

**CURSO  
DE INICIACIÓN  
A LA PROGRAMACIÓN  
DE VIDEOJUEGOS  
CON EL LENGUAJE  
FENIX v0.92a  
(en Windows)**

**Oscar Torrente Artero**



Esta obra está bajo una licencia Reconocimiento-No comercial-Compartir bajo la misma licencia 2.5 España de Creative Commons.

Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

## **Agradecimientos**

Quiero agradecer el empeño y las ganas que le ponen todas aquellas personas que tiran para adelante este proyecto tan fantástico que es Bennu/Fénix y a toda la gente que participa en el foro aportando toda la ayuda que está en su mano. Gracias por todo.

## **A quién está dirigido este texto**

Este curso está pensado para usuarios de Windows con ciertos conocimientos medios (p. ej: saber qué es un megabyte, saber crear un acceso directo, saber qué es la extensión de un archivo, haber oído hablar alguna vez de la línea de comandos...) pero que no hayan programado nunca, y que quieran aprender utilizando para ello el lenguaje Fénix. Por lo tanto, está pensado para gente que se quiera iniciar en el mundo de la programación a través de la excusa de la creación de videojuegos, objetivo éste apasionante y realmente reconfortante. Y finalmente, está pensado para gente con ilusión para ilusionar (valga la redundancia) a los demás creando mundos y experiencias mágicas y únicas.

En este manual el lector no encontrará temas avanzados tales como el manejo de punteros y gestión de memoria, o la creación de dlls: el objetivo de este texto es ser para el futuro programador una introducción elemental en el fascinante mundo de la algoritmia y del desarrollo de software mediante un lenguaje potente y flexible como Fénix, asumiendo que el lector parte de un nivel de conocimiento escaso en lo que se refiere a los conceptos básicos de programación. Por tanto, la intención de este libro es enseñar a programar a aquellos que lo intentan por primera vez y facilitar la adquisición de los procedimientos básicos necesarios para el desempeño en esta actividad, utilizando como excusa los videojuegos y como medio el lenguaje Fénix. Por eso no se han incluido temas que sobrepasarían el objetivo inicial planteado.

Así mismo, este curso no es una referencia o compendio exhaustivo de las funciones y comandos que Fénix aporta. El lector experimentado notará que en las páginas siguientes faltarán por mencionar y explicar multitud de funciones interesantes (de manejo del CD, de manejo del joystick, de gestión de memoria dinámica, de control de animaciones FLI/FLC, de todas las funciones relacionadas con la manipulación de paletas de color, etc). No se ha pretendido nunca que este texto albergara toda la información sobre Fénix, y sólo se ha incidido en aquellos aspectos del lenguaje que han sido considerados más relevantes, importantes o útiles en el día a día de un programador de este lenguaje.

## CAPÍTULO 0: ANTES DE EMPEZAR...

### Conceptos básicos de programación para el principiante:

#### ¿Qué es "programar"?

Pulsamos una tecla y los datos bailan en la pantalla o la impresora empieza a trabajar el papel. Un torno moldea con destreza el trozo de madera. El brazo robótico de la cadena de montaje aplica los puntos de soldadura con precisión. La máquina de refrescos nos ofrece nuestra bebida favorita y además nos devuelve el cambio con exactitud...

Detrás de todas estas acciones están los programadores que son personas que se han encargado de elaborar unos programas (unas "recetas" que especifican qué acciones se han de realizar y cómo) para cada máquina determinada. Es decir, los programas no son más que un conjunto de instrucciones ordenadas, entendibles por las máquinas, y que les permiten realizar tareas concretas como las enumeradas al principio, (y muchas más que vemos a nuestro alrededor).

Así pues, para el buen funcionamiento de los aparatos anteriores podemos deducir que entran en juego dos elementos fundamentales: el procesador y el programa. El procesador es la parte física: se compone de una gran cantidad de elementos electrónicos (transistores en su mayoría) miniaturizados y encapsulados en una pastilla llamada microchip. Hay de muchos tipos y su estudio es muy interesante, pero no entraremos en más detalles. Y luego está el programa. No se trata de algo físico: ya hemos dicho que está compuesto de órdenes que van a guiar al procesador en sus tareas de manipulación de la información. Es decir, podemos definir finalmente un programa como un ***conjunto de comandos que un procesador ejecuta siguiendo un orden determinado para lograr alcanzar un objetivo propuesto.***

Como programadores que somos, escribiremos esas órdenes en un archivo, como si escribiéramos un documento de texto, y luego se lo daremos como alimento al procesador. En el momento de la verdad, cuando queramos "poner en marcha" el programa, esas instrucciones que escribimos una vez van a llegar al procesador en forma de impulsos eléctricos, y el conjunto de señales que interprete nuestro procesador dará como resultado la ejecución de nuestro programa.

#### Los lenguajes de programación

Si el procesador recibe las órdenes en forma de impulsos eléctricos, ¿cómo se escriben los programas?, ¿cómo codifico esas señales?. Afortunadamente para los programadores las cosas han cambiado mucho en los últimos años. En la prehistoria de la informática electrónica (hace unos cincuenta años), los programas debían escribirse en un sistema que representara directamente los estados de las señales eléctricas que entendía el procesador.

De hecho, los únicos estados eléctricos que un procesador entiende son dos: o le llega corriente o no le llega. Por lo que a cada una de estas señales se la denominó bit y los dos posibles estados que podía recibir el procesador se representarían con un "1" (pasa corriente) o con un "0" (no pasa corriente). Es decir, que los programadores tenían que escribir directamente una secuencia de señales eléctricas codificadas en 0 y 1 (bits) en un orden determinado para que la circuitería de la máquina pudiera entender lo que tenía que hacer: es lo que se llama el código máquina. Así que un programa (ficticio) podría ser, por ejemplo:

```
10010111010101011001100101101010101010101101010100110101010110
```

Hay que decir, que aparte de ser tremendamente difícil escribir un programa en código máquina (y tremendamente fácil equivocarse), el código máquina válido para un procesador no lo es para otro, debido a su propia construcción electrónica interna. Por lo tanto, un mismo programa se tenía que

codificar en tantos códigos máquina como en procesadores se quisiera que funcionara.

A medida que los adelantos tecnológicos permitían diseñar ordenadores más complejos y con funciones más amplias, se podía ir aliviando la tarea del programador. El primer gran cambio lo aportó la llegada del lenguaje Assembler( o Ensamblador). Se trata de un lenguaje de programación que asocia una instrucción (o un comando, es lo mismo) con una determinada subsecuencia de ceros y unos que un procesador concreto entiende. Así, no hay que escribir tiras inmensas de 0 y 1: simplemente se escriben instrucciones predefinidas que son simples sustituciones de subtiras concretas.

Seguidamente os pongo un ejemplo de programa Assembler que escribe la palabra "Hola" en pantalla.

```
.MODEL SMALL
.CODE
Programa:
MOV AX,@DATA
MOV DS,AX
MOV DX,Offset Palabra
MOV AH,9
INT 21h
MOV AX,4C00h
INT 21h
.DATA
Palabra DB 'Hola$'
.STACK
END Programa
```

*Si tienes curiosidad por ejecutarlo, te alegrará saber que Windows incorpora de serie un programa que es capaz de traducir el código Assembler a código máquina (es decir, un programa que puede ejecutar el programa escrito arriba). Para arrancar este programa debes ir a Inicio->Ejecutar y escribri "debug" (sin comillas). Verás que aparece una ventana negra intimidatoria. Antes de escribir ningún código deberás teclear "a" (sin comillas), para preparar el programa para aceptar órdenes de Assembler. Una vez hayas acabado de escribirlas,deberás decirle que guarde el texto en un fichero, mediante la orden "w ruta\_del\_fichero".Si quieres ejecutar el programa, primero tendrás que cargar en memoria el fichero que creaste, con "l ruta\_del\_fichero" y ejecutarlo seguidamente con "n ruta\_del\_fichero".Para salir del programa "debug" es pulsando "q".*

Examinando el ejemplo nos damos cuenta enseguida de dos cosas: la primera, de lo complicado que es, no sólo utilizar, sino entender estas líneas de código.La segunda, aunque quizá no lo apreciemos tan claramente al no conocer otros lenguajes, de lo largo que resulta un programa que tan sólo escribe la palabra "Hola en pantalla.

Precisamente por la característica de representar directamente las instrucciones de código máquina que soporta el procesador,el lenguaje Ensamblador pertenece a la categoría de los lenguajes llamados de bajo nivel. Afortunadamente para los programadores, la cuestión de los lenguajes ha evolucionado mucho y podemos disfrutar actualmente de los lenguajes de alto nivel. La diferencia más importante en estos lenguajes es que las instrucciones se van a escribir usando una nomenclatura muy próxima al lenguaje humano. Ahora, leer el código de un programa nos puede servir para saber, más o menos, qué es lo que hace el programa sin que tengamos que tener grandes conocimientos de programación. Uno de los lenguajes de alto nivel más importante es sin duda el lenguaje C. El mismo programa antes descrito podría escribirse en lenguaje C de la siguiente manera:

```
#include <stdio.h>
void main (void)
{ printf("Hola"); }
```

La instrucción `printf("Hola");` resulta mucho más comprensible que todo el código en Ensamblador. Y ahora sí que podemos ver la diferencia de tamaño en el código total del programa.

*Si quisieras ver el resultado de ejecutar este programa, no es tan fácil como antes con la herramienta "debug". En Windows no disponemos de ninguna aplicación que nos permita ejecutar códigos fuentes escritos en C (es decir, compiladores de C). Deberíamos instalarlo por nuestra cuenta. Si te interesa el tema, mírate el entorno integrado de programación libre Dev-Cpp (<http://www.bloodshed.net>), que es una completa aplicación que, entre otras cosas, incorpora GCC, el compilador libre más famoso de C. O como alternativa, también tienes el IDE libre para C/C++ CodeBlocks (<http://www.codeblocks.org>), o incluso el IDE libre multilenguaje Eclipse (<http://www.eclipse.org>). También puedes mirarte <http://www.mingw.org>, donde encontrarás un compilador libre de C para Windows – de hecho, es una versión de GCC-, y otro más en <http://www.cs.virginia.edu/~lcc-win32>. Hay que decir también que Microsoft ofrece un entorno de programación en C/C++ llamado "Visual C++ 2005 Express Edition", que aunque no es libre, se puede descargar gratuitamente desde <http://msdn.microsoft.com/vstudio/express/downloads/>*

Existe una gran variedad de lenguajes de programación. La mayoría es de propósito general, esto es, sirven para dar solución a problemas de ámbitos muy dispares. Lo mismo se usan para crear (o más técnicamente, desarrollar) aplicaciones científicas que para videojuegos. De este tipo de lenguajes tenemos el lenguaje C (aunque su aprovechamiento máximo se produce en la programación de sistemas a bajo nivel), C++ (una evolución de C), C#, Delphi, Java, Basic, etc.

Otros lenguajes son de propósito específico -como Fénix- y están muy bien optimizados para la resolución de problemas de un campo muy concreto del saber. De este tipo tenemos lenguajes orientados al tratamiento de las base de datos empresariales como Cobol, Clipper, SQL, etc; lenguajes orientados a la resolución de problemas matemáticos como Fortran, MatLab, etc; lenguajes orientados a la generación de páginas web como XHTML, Javascript, Php, ASP.NET, etc; lenguajes orientados a la creación de videojuegos como Blitz3D o el propio Fénix, etc, etc.

Todos ellos (los de ámbito general y ámbito específico) forman parte de los denominados lenguajes de alto nivel pues su sintaxis es muy próxima al lenguaje humano. Sin embargo, sigue siendo imprescindible traducir estos lenguajes al lenguaje que entiende el procesador, el código máquina.

Ya que estamos hablando de que los ordenadores sólo entienden 0 y 1, conviene que nos paremos un momento para comentar un tema importante: los sistemas de numeración.

Un sistema de numeración es eso: un sistema para crear y manipular símbolos que representan cantidades. Todos los cálculos que realizan los ordenadores, en realidad se reducen a simples sumas, restas, multiplicaciones, divisiones de diferentes cantidades (números) y poca cosa más. Hasta el más espectacular gráfico 3D no es más que el resultado de estas operaciones matemáticas. Incluso el texto que escribes en un procesador de textos se reduce al final a operaciones aritméticas y lógicas. Y sabemos que estos números dentro del ordenador están representados siempre como 0 y 1. Así pues, podemos decir que los ordenadores trabajan con números representados en el sistema binario (este sistema se llama así porque sólo utiliza dos símbolos para representar todos los números posibles).

El número 0 en el sistema binario se representa lógicamente por el 0. Y el 1 por el 1. ¿Pero el 2? Si sólo tenemos el 0 y el 1, ¿cómo entiende el 2 el ordenador? Bueno, según la teoría matemática de los sistemas de numeración (tema en el que no vamos a profundizar aquí), el número 2 se representa así: "10". Y el 3 así: "11". A continuación tienes una tabla de equivalencias de los 16 primeros números representados en sistema binario y en sistema decimal, que es el sistema que utilizamos los humanos (este sistema se llama así porque utiliza diez símbolos para representar todos los números posibles: 0,1,2,3,4,5,6,7,8 y 9).

Sistema decimal	Sistema binario
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Fíjate en un detalle que luego veremos que es muy importante: dependiendo del número que se trate, el ordenador necesitará más bits o menos de memoria para poder representarlo y trabajar con él: para usar el número tres sólo necesita dos bits, pero para el número 14 necesitará cuatro. Si hubiéramos representado el número 16 veríamos que habría necesitado 5 bits.

Puedes ver claramente que el sistema binario es un engorro para los humanos. Si tuviéramos que escribir en un papel un número muy grande, no acabaríamos de escribir 1 y 0s. Para evitar esto, se inventó el llamado sistema hexadecimal. El sistema hexadecimal es una manera de representar los números binarios pero que simplifica su escritura, con la ventaja adicional que es muy fácil pasar los números de hexadecimal a binario y viceversa. Diríamos que el hexadecimal es un binario "para vagos". El truco consiste en coger las primeras 16 (por eso se llama hexadecimal) combinaciones posibles de cuatro bits, -que justamente son las 16 combinaciones que he escrito antes -, y asignarle a cada una un símbolo, de la siguiente manera:

Sistema binario	Sistema hexadecimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8

1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Así, un número binario como 10010011 lo podríamos escribir en hexadecimal así: 93 (número que en decimal sería el 147). ¿Cómo lo hemos hecho? Muy fácil: empezamos por la derecha del número binario y vamos haciendo grupos de cuatro bits (si el grupo de más a la izquierda tiene menos de cuatro, se rellena por la izquierda de 0 hasta llegar a cuatro). Cada grupo lo vamos sustituyendo por su símbolo hexadecimal de la tabla anterior: y ya está. Y a la inversa es igual de fácil: vamos sustituyendo cada símbolo hexadecimal por su equivalencia en binario.

Por cierto, ¿cómo sabemos si un número está escrito en hexadecimal o en decimal? Se pueden confundir, como acabamos de ver. Existe un convenio que dice que los números hexadecimales irán seguidos de la letra "h" para indicar que efectivamente son números hexadecimales. O sea, que el 93 anterior lo tendríamos que haber escrito así: 93h. Otro convenio equivalente dice que el número hexadecimal ha de ir precedido de los símbolos "0x", con lo que tendríamos que escribir por ejemplo 0x93.

Fíjate por último que un número hexadecimal puede incorporar símbolos alfabéticos entre la A y la F. Un número tal como AB3FE24h es perfectamente válido.

Finalmente, comentar que, sea cual sea la representación de los números, éstos seguirán siendo los mismos y se podrán realizar las mismas operaciones con las que estamos tan acostumbrados en el sistema decimal: sumas, restas, multiplicaciones, divisiones, etc. Eso sí, con ciertas reglas específicas a seguir en cada uno de los sistemas de numeración, como es natural.

### **Editores, intérpretes y compiladores**

Si podemos utilizar estos lenguajes de alto nivel es gracias a la aparición de los intérpretes y de los compiladores. Los programadores escriben sus programas siguiendo la sintaxis (las normas) de un lenguaje determinado, por ejemplo C, y guardan el programa en un archivo como si se tratara de un texto. De hecho, el programa lo podremos escribir con cualquier editor de texto que tengamos disponible, incluso el "Bloc de notas" de Windows. Este archivo de texto que contiene el código de nuestro programa es el que se denomina código fuente del programa. A continuación interviene el intérprete o el compilador (o los dos, como el caso de Fénix) para traducir ese código fuente en código máquina, y posibilitar la ejecución del programa por el procesador. Es decir, los intérpretes y los compiladores son programas especiales cuya tarea es traducir el texto que hemos escrito nosotros a la secuencia de 0 y 1 que entiende el procesador.

Existe una diferencia fundamental entre los intérpretes y los compiladores, que va a decidir si el programador se decanta por el uso de un lenguaje de programación interpretado o compilado.

Los lenguajes interpretados son leídos y traducidos por el intérprete instrucción por instrucción, a la misma vez que se está ejecutando el programa. Es decir, el intérprete envía las instrucciones traducidas al procesador a la misma vez que las va leyendo del código fuente. Y esto ocurre cada vez que se ejecuta el programa.

<i>Inconvenientes</i>	<i>Ventajas</i>
*Esta manera de funcionar los hace más lentos, y además, para su correcto funcionamiento tiene que existir un intérprete allí donde el código deba ejecutarse.	*Si el usuario tiene instalado el intérprete para un lenguaje concreto y lo que le suministramos es el código fuente, no importa el tipo de procesador que esté utilizando ni el sistema operativo que tenga instalado. Un programa en Java (lenguaje pseudointerpretado) podrá funcionar en un Pentium IV con Windows o con Linux, o en un Motorola montado en un Apple con el sistema MacOS instalado. Si existe un intérprete de ese lenguaje para una plataforma hardware determinada y para un sistema operativo concreto, se puede ejecutar allí.

Lenguajes interpretados son: Php, Python, Tcl, Perl, Javascript,...

Cuando se utiliza un lenguaje compilado, como C, la cosa cambia. El compilador se encargará de traducir el código fuente en código ejecutable directamente para el procesador y generará un archivo separado incluyendo este código binario. En Windows esos archivos binarios suelen llevar la extensión .exe (ejecutables). Así, el usuario solamente tendrá que ejecutar este archivo que tiene las instrucciones ya traducidas al código máquina.

<i>Inconvenientes</i>	<i>Ventajas</i>
*Precisamente el hecho de generar el código final a ejecutar por el procesador es lo que trae su mayor desventaja. El código máquina se generará para un procesador de una familia determinada -hemos comentado antes que los códigos máquina son muy poco transportables- , por ejemplo para los Pentium, y no podrá ejecutarse en otra plataforma, como un Macintosh. Y no sólo para las plataformas hardware, sino también para los sistemas operativos. Un programa compilado para Linux no funcionará bajo Windows. Esto obliga a los programadores a recompilar el programa para las diferentes plataformas y entornos en los que va a trabajar, generando diferentes versiones del mismo programa. Además el programador también se encuentra con el trastorno de tener que recompilar el programa cada vez que le hace una modificación o corrección al código fuente.	*El archivo binario obtenido está muy optimizado y su ejecución es muy rápida, ya que no requiere de más traducciones. Además, el usuario no necesita ningún programa adicional para poder ejecutarlo y así el programador puede distribuir sus productos con más facilidad.

Lenguajes compilados son: C, C++, Delphi, Cobol, Fortran...

Si quieres ver un resumen de la evolución que han sufrido los distintos lenguajes de programación existentes a lo largo de su historia (cuándo aparecieron, a partir de qué lenguajes anteriores se crearon, cuándo dejaron de existir los que ya no se utilizan, etc) es muy interesante ver los árboles genealógicos accesibles en <http://www.levenez.com/lang>

¿Y Fénix, qué es? ¿Lenguaje interpretado o compilado? Pues, como Java, C# o VB.NET, es las dos cosas. Pero eso lo veremos en el siguiente capítulo...

### Las librerías

Cuando programamos en un lenguaje determinado, hacemos uso de una serie de instrucciones, órdenes o comandos (son sinónimos) que son los propios de ese lenguaje y que son, en definitiva, los que le dan cuerpo. Por ejemplo, para imprimir "Hola" por pantalla en C hemos visto que podemos hacer servir el comando `printf("Hola");`, en C# usaríamos el comando `Console.WriteLine("Hola");`, en Java el comando `System.out.println("Hola");`, etc. Cuando se diseñó cada uno de los lenguajes existentes, se dotó de un

conjunto más o menos extenso de comandos para poder hacer algo con ellos: algo así como el vocabulario básico con el que poder trabajar con ese lenguaje.

La gente pronto se dió cuenta que con ese número limitado de comandos inicial era capaz de crear programas que, a su vez, podían ser utilizados como un comando más "no estándar" del lenguaje utilizado. Es decir, que existía la posibilidad de generar nuevos comandos para un lenguaje determinado, a partir de los comandos iniciales de ese mismo lenguaje, y hacer disponibles esos nuevos comandos para los demás programadores. La utilidad era obvia: ¿para qué escribir un programa de 100 líneas que me dibujara un círculo en la pantalla cada vez que quería dibujar el círculo ? Lo más inteligente era escribir ese programa una vez, y convertirlo en un comando más del lenguaje, para que así todo el mundo a partir de entonces no tuviera que empezar de 0 y escribir siempre esas 100 líneas, sino que solamente tuviera que escribir ese nuevo comando que él solito dibujaba un círculo. Así pues, cada programador se podía crear sus propios comandos personales que le facilitaban la faena: los codificaba una vez (es decir, escribía lo que quería que hiciera ese comando, a partir de comandos más elementales) y los podía utilizar a partir de entonces cuando quisiera. Era una forma perfecta para encapsular la dificultad de un programa y acelerar el desarrollo. Evidentemente, estos nuevos comandos - a partir de ahora los llamaremos funciones- se compartían entre los programadores, y a los que no habían diseñado esa función "no estándar", no les interesaba mucho cómo estaba hecho por dentro esa función, lo que les importaba es que hiciera bien lo que tenía que hacer.

Para poder compartir mejor entre los programadores estas nuevas funciones, pronto surgió la necesidad de agrupar las más populares o las más necesarias en conjuntos más o menos amplios: las librerías (o mejor, bibliotecas) de funciones. Las bibliotecas de funciones son, como su nombre indica, almacenes de funciones disponibles para un lenguaje concreto, que los programadores pueden hacer servir para facilitar tareas laboriosas que requerirían un montón de líneas de código si se partiera desde cero; gracias a que alguien previamente ya las ha codificado y las ha puesto disponibles en forma de función alojada en una biblioteca, se pueden evitar muchas horas de trabajo.

Para cada lenguaje hay disponibles bibliotecas de muchos tipos. Normalmente, las librerías viene agrupadas por utilidad: hay librerías de entrada/salida, que almacenan funciones que manejan el ratón y el teclado, hay librerías de sonido, que almacenan las funciones que gestionan la posibilidad de reproducir o grabar diferentes formatos de sonido, hay librerías GUI, que almacenan funciones que pintan ventanas botones automáticamente sin tener que empezar desde cero, hay librerías de red, que posibilitan tener funciones de comunicación entre ordenadores, etc. Hay que saber también que un nombre técnico para llamar a las librerías en general es el de API (Application Programming Interface -Interfaz de programación de aplicaciones), así que en nuestro programa podríamos incorporar una API de entrada/salida, una API GUI, una API de red,..., y hacer uso de las funciones incluidas en ellas.

De hecho, ya veremos que el lenguaje Fénix incorpora un mecanismo específico para poder utilizar librerías de todo tipo, escritas en C y/o Fénix, -normalmente, las APIs estarán disponibles para aquel lenguaje con el cual ellas mismas han sido creadas: Fénix se sostiene sobre C - dotando así al lenguaje de gran flexibilidad, extensibilidad y potencia, sin alterar para nada su núcleo central, el cual se mantiene pequeño y optimizado.

Para acabar, antes hemos hablado de que para programar necesitas como mínimo un editor de textos, y un programa intérprete y/o compilador (depende del lenguaje). Pero estarás preguntando: las librerías, físicamente, ¿qué son? ¿Dónde están? Normalmente las librerías son ficheros binarios -es decir, ya compilados- que para hacer uso de ellos has de tener copiados/instalados en tu sistema, bien en la misma carpeta que el archivo de texto con tu código fuente, bien en una localización estándar accesible. En Windows generalmente tienen la extensión dll. Y evidentemente, si quiere usar las funciones que incluyen, tienes que hacer referencia a estos ficheros desde nuestro código fuente con un protocolo determinado según el lenguaje.

## Las librerías gráficas

Desengañémonos: C++ (y C) es el lenguaje más utilizado para crear videojuegos a nivel profesional. Así que si queremos ser "profesionales" -¡cuidado con esa palabra!-, hay que conocer C++. C++ es un lenguaje muy potente y muy flexible, pero también bastante difícil de aprender. Además, C++ no incorpora funciones "nativas" (o sea, que pertenezcan al propio lenguaje en sí) para la gestión de gráficos ni multimedia, por lo que si nos decidiéramos por usar este lenguaje tendríamos que echar mano de alguna librería gráfica disponible y aprender a utilizarla.

Por librería gráfica se entiende aquella que te facilita mediante programación la tarea de dibujar en pantalla, ya sea bien en 2D o bien en 3D; y por multimedia que además dé soporte al control y gestión de sonido, música, vídeo, interacción teclado/ratón/joystick/..., etc.

Así pues, para incluir gráficos en los juegos y también incluir sonido y sistemas de control efectivos en C++ contamos con librerías pre-programadas. Si vamos a usar este lenguaje, sería una locura de nuestra parte y un suicidio profesional rechazarlas y hacer nuestros propios códigos (o rutinas, es lo mismo), desde cero: basta con aprender cómo funcionan las funciones incluidas en la librería que elijamos.

Nosotros no programaremos en C++ sino en Fénix, un lenguaje creado específicamente para hacer videojuegos - C++ es de propósito general - y infinitamente mucho más fácil de aprender. Evidentemente, hay algo a cambio de la facilidad que tiene Fénix: no es tan flexible como C++ y hay muchas cosas que no podremos hacer con él, pero ya hablaremos de eso en el siguiente capítulo.

A pesar de que programaremos en Fénix, y en principio no tendríamos por qué saberlo, como programadores de videojuegos que seremos nos interesará conocer como mínimo de oído las librerías (o APIs) gráficas/multimedia más extendidas e importantes que existen para C++, y así algún día, quién sabe, dar el salto.

---

Existen cuatro librerías principales, dos de ellas consideradas "amateur" y otras dos consideradas "profesionales". Las primeras son Allegro y SDL, las segundas DirectX y OpenGL. Todas se pueden utilizar básicamente en C o C++:

***Allegro*** (<http://www.allegro.cc/>) : Excelente librería para gráficos en 2 dimensiones que también maneja sonido, joystick, teclado, mouse y temporizador. Tiene la expansión AllegroGL (que permite utilizar OpenGL con Allegro) para poder crear gráficos en 3D. Una de las principales ventajas que tiene es que fue específicamente diseñado para la programación de videojuegos, por lo que todas las rutinas que utiliza están hechas para ser fáciles de manejar y sobretodo eficiente desde el punto de vista de un juego. Además de que tiene ya preconstruidas ciertas funciones para realizar algunos efectos que normalmente tendrías que programar a mano. Otro tanto a favor es que es multiplataforma, lo que quiere decir que se puede generar código para Windows, Linux y otros sistemas sin modificar línea alguna. Es una librería bastante simple de utilizar y suficientemente potente para una primera incursión en el mundo gráfico. No obstante, no es muy usada en juegos profesionales (o al menos no muchos aceptan que lo usen), quizás porque está diseñada con una visión general de cómo funciona un videojuego -por tanto, es ideal para novatos y como hobby-, pero en un juego profesional a los programadores les gusta exprimir hasta las raíces de lo que están utilizando; por ejemplo, Allegro no tiene soporte para aceleración gráfica.

Un curso de programación gráfica en C++ (con una parte teórica bastante interesante) basado en la librería Allegro y AllegroGL que puede resultar útil se puede encontrar en [http://sp4br75.digiweb.psi.br/curso\\_programacion\\_grafica/pg\\_frames.htm](http://sp4br75.digiweb.psi.br/curso_programacion_grafica/pg_frames.htm)

---

***SDL - Simple DirectMedia Layer-*** ( <http://www.libsdl.org/>): Comparable a Allegro, es una muy recomendable librería para el que recién empieza porque además de ser simple de programar es rápida y además tiene la posibilidad de trabajar con OpenGL para el manejo de aceleración gráfica, ya que, aunque SDL por sí misma realiza bastantes tareas, de vídeo, sonido, entrada y red, no tiene la potencia requerida para labores exigentes (éste es el caso de los gráficos 3D en tiempo real), por lo cual necesita asociarse con otras librerías, como por ejemplo, OpenGL. Algo importante a tener en cuenta es que es también multiplataforma: es portable a diferentes sistemas operativos como Windows, Linux, BeOS, MacOS, Solaris, IRIX, y FreeBSD sin tener que cambiar ni siquiera una sola línea de código. Esta librería fue escrita en C, pero trabaja en forma nativa con C++, y puede utilizarse fácilmente con otros lenguajes de programación como Ada, Eiffel, ML, Perl, Python y Ruby. Y otra de las características muy importantes que tiene SDL es que se trata de un proyecto libre, es decir el código fuente está disponible para prácticamente cualquier persona que quiera echarle un vistazo, y si lo deseas puedes arreglarlo a tu gusto, por lo tanto actualmente existen una gran cantidad de personas que continuamente, como hobby, están aportando algo nuevo a esta librería en sus ratos libres, ya sea mejorándolo en su desempeño o añadiéndole nuevas funciones.

Si se quiere utilizar la librería SDL para programar sobre la plataforma NET (en lenguajes como C# o VB.NET) en vez de con C/C++ clásico, existe una variante -un "port"- de SDL enfocada para ello, la SDL.NET, en <http://cs-sdl.sourceforge.net>

---

***DirectX*** ( <http://msdn.microsoft.com/directx> ) :DirectX es la librería de Microsoft por lo que únicamente puedes hacer Juegos para Windows con ella. Esa no es la única desventaja, también es difícil de programar. Para que quede más claro, no es recomendable comenzar por DirectX a menos que te decidas a hacer Juegos para Windows exclusivamente y tengas un sólido respaldo de conocimientos sobre programación general. Recomiendo dejar la dificultad de DirectX para una segunda etapa, cuando ya sepas manejar gráficos y demás elementos multimedia. Tarde o temprano tendrás que aprender a trabajar con ella si quieres hacer videojuegos ambiciosos, pero es mejor si te introduces en el mundo de las DirectX después de haber programado ya en otras librerías. DirectX fue diseñada por Microsoft con el fin de permitir a los programadores escribir programas multimedia para sistemas Windows de manera fácil, sin tener que preocuparse sobre qué hardware está corriendo y como funciona, como es el caso del tipo de tarjeta de video, etc, es decir, sin tener que escribir código específico de hardware. Para la mayoría de las aplicaciones tradicionales, el acceso a funciones de hardware se hacen a través del sistema operativo; sin embargo, para las aplicaciones multimedia, que son más exigentes, esta vía puede resultar bastante lenta; DirectX es un conjunto de librerías que permitieran acceso a funciones de hardware de forma más directa, sin pasar por todos los callejones que usualmente crea el sistema operativo. Obviamente esto hace que, además de ser un producto de Microsoft, DirectX está íntimamente ligado con el sistema operativo Windows. Por eso, esto hace que todas aquellas aplicaciones sean, en extremo, difíciles de portar a otras plataformas, es decir, una vez que hayas terminado de programar tu juego estará condenado a trabajar solamente en Windows, y cuando quieras convertir tu juego para que pueda correr en otras computadoras que usen Mac o Linux tendrás que hacerle muchos cambios al código, lo cual no es nada deseable a menos que sepas que el único mercado al que va dirigido tu juego son personas con una computadora con Windows. No obstante, una de las principales ventajas que tiene es que no está hecho solamente para la parte gráfica, sino que con DirectX puedes manejar toda la multimedia de la computadora como sonidos, música, dispositivos de Entrada/Salida como teclado, joystick, y varias otras cosas más, por lo que en este sentido es ideal para programas videojuegos. Realmente DirectX esta compuesto de 7 componentes:

-*DirectDraw*: proporciona capacidades de gráficos en 2D y sirve como base de procesos de rendering para otros servicios de video.

-*Direct3D*: es un motor de rendering para gráficos 3D en tiempo real que integra una API de bajo nivel para el render de polígonos y vértices y uno de alto nivel para la

manipulación de escenas complejas en 3D.

*-DirectShow:* es una arquitectura multimedia que divide el procesamiento de tareas multimedia, como la reproducción de video, en un conjunto de pasos conocidos como filtros: éstos tienen un número de entradas y salidas que se conectan entre ellas, ofreciendo gran flexibilidad y modularidad al desarrollar, que puede combinar los filtros según sus necesidades.

*-DirectInput:* permite recoger información en tiempo real del ratón, teclado y joysticks

*-DirectSound:* ofrece drivers y mezcladores de sonido con soporte Dolby, los cuales posibilitan un rendimiento óptimo de sonido posicional en 3D, permitiendo a los programadores de juegos situar eventos de sonido en cualquier punto del espacio perceptual del usuario.

*-DirectMusic:* es mucho más que simplemente reproducir sonidos; provee un sistema completo para implementar bandas sonoras dinámicas y sincronizadas que aprovechan la aceleración hardware y efectos de posicionamientos 3D avanzado, entre muchas otras ventajas.

*-DirectPlay:* proporciona protocolos independientes para funciones de red, para videojuegos de varios jugadores en Internet.

En el PC existe una gran variedad de tarjetas gráficas, de sonido, y demás hardware. Esto crea un problema a la hora de programar, ya que cada uno de estos dispositivos se programa de una manera diferente. DirectX, al igual que las otras librerías, es un puente entre el programador y la máquina. Dicho puente intenta evitar, en la medida de lo posible las diferencias que existen entre los distintos tipos de hardware que hay en el mercado. Por ejemplo, para hacer un sonido usaremos una instrucción de DirectSound, la cual será igual para todas las tarjetas de sonido, ya que es el propio DirectSound quien se encarga de dar las órdenes específicas a la tarjeta de sonido que tenemos conectada a nuestro PC. Si no existiera DirectSound, tendríamos que hacer rutinas de sonido diferentes para modelos de tarjetas de sonido que no fueran compatibles entre sí.

***-OpenGL -Open Graphics Library-*** (<http://www.opengl.org/>) : OpenGL es una librería 3D profesional multiplataforma, y esa es su mayor ventaja. En la actualidad muchas personas utilizan Linux u otros sistemas ajenos a Microsoft lo que abre nuevos e importantes nichos de mercado. Aprender a programar en OpenGL nos facilita el ingreso a estos mercados y amplía nuestros horizontes. Es interesante conocerla e implementarla con SDL para hacer tus primeros trabajos serios usando C. Es la competencia de Direct3D, ya que, a diferencia de DirectX (en conjunto), esta API sólo posee rutinas para el manejo de gráficos y no tiene soporte para sonido, entrada, red, etc, el cual se ha de buscar en otras librerías (SDL, Allegro o más específicas). La especificación del API de OpenGL es controlada por un grupo llamado Architecture Review Board, conformado por varias empresas importantes del sector de las tarjetas gráficas como nVidia, ATI, Creative Labs, SGI, entre otras. La implementación libre por excelencia de OpenGL se llama MESA (<http://www.mesa3d.org>). OpenGL está pensado para funcionar en sistemas con aceleración hardware, por lo que los fabricantes de tarjetas han de incluir soporte OpenGL en sus productos.

Para que no te quedes con las ganas, ahí va un código de un programa escrito en C usando OpenGL, para que veas qué pinta tiene. Este ejemplo en concreto lo que único que hace es, otra vez, escribir "Hola" en la barra de título de la ventana y poner un fondo azul a ella.

```
#include <windows.h>
#include <conio.h>
#include <gl\gl.h>
#include <gl\glaux.h>
```

```

void main (void)
{
auxInitDisplayMode(AUX_SINGLE| AUX_RGBA);
auxInitPosition(100,100,250,250);
auxInitWindow("Hola");

glClearColor(0.0f,0.0f,1.0f,1.0f);
glClear(GL_COLOR_BUFFER_BIT);
glFlush();

getch();
}

```

Evidentemente, para que podamos ver el resultado de ejecutar este código, necesitamos tener instalada la librería OpenGL (no son más que unos pocos archivos con extensión lib y dll que actualmente vienen de serie con Windows), hacer referencia a ella en nuestro entorno de programación, y un compilador C que nos cree el ejecutable.

Por lo tanto, finalmente podríamos concluir, a modo de esquema, que en el panorama de la programación multimedia existe hoy en día podría resumirse en esta correspondencia:

<i>Librerías no portables</i>	<i>Librerías portables (y libres)</i>
DirectDraw, DirectInput	SDL, Allegro
Direct3D	OpenGL
DirectMusic	MikMod ( <a href="http://mikmod.raphnet.net/">http://mikmod.raphnet.net/</a> ), etc
DirectSound	OpenAL ( <a href="http://www.openal.org">http://www.openal.org</a> )
DirectPlay	Alguna librería de red...
DirectShow	

*(MikMod es una librería - entre otras- para reproducir cierto tipo de archivos musicales (archivos mod's) usada además por Fénix. OpenAL es una librería libre para el manejo del audio tridimensional; tiene la capacidad de acceder directamente a funciones avanzadas de la tarjeta de sonido y controlar la llamada aceleración de audio.)*

Un detalle que salta a la vista es que Microsoft ofrece una solución totalmente integrada y completa para todos los aspectos que conllevan la programación de un videojuego, y que en el otro lado tenemos multitud de librerías que cubren partes del todo, teniendo que buscarlos la vida con varias de ellas; estas librerías están dispersas y son muy heterogéneas, y su uso por parte de los desarrolladores implica bastantes problemas a la hora del desarrollo. En este sentido, es remarcable el proyecto libre Tao Framework (<http://www.taoframework.com>), el cual intenta integrar en un todo multimedia multiplataforma coherente -mejor dicho, basado en la plataforma NET- algunas de estas librerías independientes, como son SDL, OpenGL (+Glut), OpenAL, y otras más específicas, como ODE (<http://ode.org>) -un simulador de dinámica de cuerpos rígidos-, DevIL (<http://openil.sourceforge.net>) -una librería para el tratamiento de imágenes-, Cg -computer graphics- e incluso incorpora un lenguaje de programación completo como es Lua (<http://www.lua.org>), intentando ofrecer así una competencia global a DirectX.

## Lenguajes de programación de videojuegos

Todo lo que hemos escrito en el apartado anterior respecto a las librerías nos sería útil en la práctica si programáramos videojuegos en C o C++. No obstante, hemos dicho que ambos lenguajes son bastante complejos y difíciles de aprender para un iniciado, con lo que fácilmente uno se podría desesperar y dejarlo todo correr.

Es por eso que existe Fénix. Un lenguaje de programación fácil y asequible para cualquiera. Con Fénix el programador no necesita introducirse en los intrincados mundos del C y de sus librerías: Fénix ya hace el “trabajo sucio” por él: así, el programador puede dedicar todas sus energías al diseño y programación de su videojuego, en vez de pelearse con el manejo de las tarjetas gráficas, la memoria y otros monstruos.

No obstante, como es natural, Fénix no es la única alternativa para desarrollar videojuegos de forma “casera”. No es superfluo conocerlas por encima, para saber si nos podrían interesar, y conocer otras soluciones a los mismos objetivos.

Para empezar, tenemos el **XNA Game Studio** (<http://msdn2.microsoft.com/es-es/xna/default.aspx>), que es básicamente un intérprete y un conjunto de librerías asociadas, ofrecido por Microsoft gratuitamente -sólo en la versión Express- para el desarrollo específico de videojuegos. En realidad, XNA no incorpora ningún lenguaje propio, sino que es un añadido al lenguaje C# para facilitar su uso en el cometido concreto de la creación de videojuegos, por lo que de hecho el desarrollador programará en C#, y por tanto, es evidente que para programar con esta plataforma se necesita tener instalado previamente de forma obligatoria un intérprete de C#, y es muy recomendable contar también con un IDE adecuado a este lenguaje. (Todo estos prerequisites se pueden encontrar -gratuitamente también- en <http://msdn.microsoft.com/vstudio/express/downloads> ). Finalmente, además también se necesita tener instalada la librería DirectX.

XNA está especialmente pensada para desarrolladores aficionados, estudiantes o independientes, y es una estrategia de Microsoft para crear una comunidad online de desarrolladores y juegos disponibles -previa suscripción en muchos casos- para Windows y también para su consola Xbox 360. Para saber más sobre esta plataforma, se puede consultar, a parte de la dirección anteriormente citada del XNA Developer Center, ésta otra: <http://creators.xna.com> o también <http://www.xnadevelopment.com/index.shtml>

En otro nivel, tenemos **3D GameStudio** (<http://www.3dgamestudio.com>) . Según lo que dice en su página web, “es la suite líder para creación de aplicaciones en tiempo real 2D y 3D. Combina la programación en lenguaje C-Script con un motor 3D de alto nivel, un motor 2D, un motor de dinámica física, un editor de modelos, un editor de niveles y un editor de terrenos, más librerías extendidas de objetos 3D, arte gráfico y modelos de juegos prefabricados”. Según ellos nunca ha sido tan fácil crear juegos de 1ª persona (plano subjetivo) , de 3ª persona, RPGs (Rol Playing Games: es decir, juegos tipo “Zelda” para entendernos), juegos de plataforma, simuladores de vuelo, juegos de deportes, aplicaciones 3D, presentación en tiempo real, etc. Ofrece tres niveles de creación de juegos: con simples clics de ratón a partir de plantillas prediseñadas; programando en C-Script usando su compilador y depurador (programa que se dedica a detectar errores en el código fuente) o bien incluyendo el código C-Script a modo de bloque interno dentro de un código mayor programado en C++ o Delphi. Promete que incluso si ningún conocimiento de programación, siguiendo sus tutoriales paso a paso, un juego simple puede ser construido en una tarde, y usando el C-Script juegos de calidad comercial pueden ser creados y distribuidos con éxito. En la página web hay demos de juegos, sección de preguntas frecuentes (FAQs), un magazine de usuarios, etc.

Otra alternativa también a tener en cuenta es **BlitzMax** (<http://blitzbasic.com>) . Según lo que dice su página web, “BlitzMax proporciona un entorno simple pero poderoso para la creación de videojuegos – simple porque se basa en el popular y cómodo lenguaje de programación BASIC, y poderoso gracias a un optimizado motor 2D/3 que trabaja en segundo plano. Incluye muchos comandos para ayudar al programador en la creación del juego, pero no demasiados: en vez de confundir con toneladas de

comandos, el conjunto de éstos ha sido cuidadosamente diseñado para proveer la máxima flexibilidad con el mínimo esfuerzo. Blitzmax es la nueva generación del lenguaje de programación para videojuegos de la casa Blitz Research: mantiene las raíces del lenguaje BASIC de anteriores productos, como Blitz3D y BlitzPlus, pero añade un montón de nuevas características interesantes, como la mejora de su lenguaje BASIC (añade punteros a funciones, polimorfismo y herencia, arrays dinámicos, paso de parámetros por referencia, cadenas UTF16, soporte para conectar con código C/C++ y soporte para programar directamente en OpenGL, etc), soporte multiplataforma para Windows, Linux y MacOS, diseño modular en ficheros independientes, conjunto de comandos 2D extremadamente fáciles de usar, inclusión de un IDE – Integrated development environment- completo, etc. Y si el programador tiene problemas, siempre puede consultar la excelente documentación o acudir a la inmensa comunidad de usuarios en los diferentes foros de Blitz para pedir ayuda.”. Un ejemplo de éstos últimos lo puedes encontrar en el estupendo foro de GameDevelopers <http://portalxuri.dyndns.org/blitzbasico/index.php> , donde podrás preguntar en castellano todas tus dudas sobre esta herramienta.

Otra alternativas (no están todas, ni mucho menos) para Windows que pueden ser tanto o más recomendables que las anteriores son:

<b>PlayBasic</b> ( <a href="http://underwaredesign.com">http://underwaredesign.com</a> )	Lenguaje para videojuegos 2D de estilo Basic
<b>DarkBasic</b> ( <a href="http://www.darkbasic.com">http://www.darkbasic.com</a> )	Similar al anterior pero permite trabajar también con 3D.
<b>Torque 2D</b> ( <a href="http://www.garagegames.com">http://www.garagegames.com</a> )	Motor de juegos en 2D multiplataforma, más IDE incorporado para programarlos en C++ o en un lenguaje propio, TorqueScript.
<b>Clanlib</b> ( <a href="http://www.clanlib.org">http://www.clanlib.org</a> )	Librería libre y multiplataforma para la programación de videojuegos 2D utilizando C++.
<b>SDLBasic</b> ( <a href="http://sdlbasic.sourceforge.net/flatnuke">http://sdlbasic.sourceforge.net/flatnuke</a> )	Lenguaje interpretado basado en la librería SDL, multiplataforma y libre. Puede utilizar toda la potencia de SDL con la sintaxis Basic
<b>FreeBasic</b> ( <a href="http://www.freebasic.net/index.php">http://www.freebasic.net/index.php</a> )	Lenguaje de ámbito general de estiloBasic. Libre y multiplataforma.
<b>Xblite</b> ( <a href="http://perso.orange.fr/xblite">http://perso.orange.fr/xblite</a> )	Lenguaje de ámbito general de estilo Basic pero con la velocidad de C. Libre; para Windows.
<b>PureBasic</b> ( <a href="http://www.purebasic.com">http://www.purebasic.com</a> )	Lenguaje de ámbito general de estilo Basic, multiplataforma
<b>Real Basic</b> ( <a href="http://www.realsoftware.com">http://www.realsoftware.com</a> )	Lenguaje de ámbito general de estilo Basic, multiplataforma.
<b>Panda 3D</b> ( <a href="http://www.panda3d.org">http://www.panda3d.org</a> )	Librería 3D para ser utilizada en programas escritos con el lenguaje Python (lenguaje libre y multiplataforma - <a href="http://www.python.org">http://www.python.org</a> -). Vale la pena también mirarse <a href="http://www.pygame.org">http://www.pygame.org</a> (permite utilizar la librería SDL con Python en vez de C)
<b>Crystal Space</b> ( <a href="http://www.crystalspace3d.org">http://www.crystalspace3d.org</a> )	Librería 3D libre para ser utilizada en programas escritos en C/C++ preferentemente para videojuegos. Incorpora también un motor propio.
<b>GDT</b> ( <a href="http://gdt.sourceforge.net">http://gdt.sourceforge.net</a> )	Librería libre para C/C++ enfocada en el creación de videojuegos, desarrollada por la comunidad hispana de los foros de Game Developers.
<b>Lua</b> ( <a href="http://www.lua.org">http://www.lua.org</a> )	Lenguaje de script embebible, ligero y rápido,

	especialmente diseñado para el desarrollo de videojuegos y multimedia. Libre y multiplataforma.
<b>Basic4GL</b> ( <a href="http://www.basic4gl.net">http://www.basic4gl.net</a> )	Lenguaje Basic que incorpora un compilador con soporte para OpenGL. Permite así programar en esta librería 3D sin necesidad de utilizar C/C++. Es libre pero sólo para Windows.
<b>FANG</b> ( <a href="http://www.fangengine.org">http://www.fangengine.org</a> )	Paquete libre con clases Java que facilitan la programación en este lenguaje de videojuegos 2D con soporte de red. Especialmente pensado para estudiantes de programación
<b>Phrogram</b> ( <a href="http://www.phrogram.com">http://www.phrogram.com</a> )	Anteriormente conocido como KPL -Kid's Programming Language-, es un lenguaje de programación diseñado para ser fácil y divertido, y especialmente adecuado para la programación de multimedia/ videojuegos, más un IDE, con cierta similitud a Visual Basic. Se ha de ejecutar sobre una plataforma NET.
<b>Processing</b> ( <a href="http://www.processing.org">http://www.processing.org</a> )	Lenguaje y entorno de programación libre destinado a la creación de imágenes, animaciones e interacciones. Especialmente pensado para artistas y diseñadores

Por último, no me gustaría dejar de comentar al **Adventure Game Studio** (<http://www.adventuregamestudio.co.uk/>), el **RPGMaker** ([http://www.enterbrain.co.jp/tkool/RPG\\_XP/eng](http://www.enterbrain.co.jp/tkool/RPG_XP/eng)) o el **FPSCreator** (<http://www.fpscreator.com>) los cuales son programas especializados en generar aventuras gráficas el primero, RPGs el segundo y Shooters el tercero, sin prácticamente programación: a golpe de ratón. Pero claro, siempre llega un punto donde no son suficientemente completos ni tienen ni mucho menos la flexibilidad que un lenguaje de programación ofrece.

A destacar, no obstante, dentro de esta familia, el software generalista **GameMaker** (<http://www.yoyogames.com/make>), en cuya web además se pueden descargar bastantes recursos como músicas, fondos y sprites de diversa índole. Una alternativa libre a este último programa, todavía en desarrollo, es **Flexlay** (<http://flexlay.berlios.de/>).

*Por sprite (a partir de ahora se utilizará bastante esta palabra) se entiende cualquier imagen que se mueva en la pantalla.*

Bueno, las diferentes propuestas parecen la panacea: ofrecen poder crear supervideojuegos, -3D además, cosa que Fénix de momento por sí solo no puede-, sin demasiado esfuerzo. Y probablemente sea verdad, pero la mayoría tienen un pequeño inconveniente (llamémoslo así). Aparte de que son productos comerciales y por tanto, para adquirirlos hay que pagar un precio, no son libres. Y eso, creo, que es un factor relativamente importante.

### **Pero...¿qué es eso de "software libre"? ¿Y eso de Linux?**

Has leído en el apartado anterior que las diferentes propuestas de lenguajes de videojuegos no son libres, o en apartados anteriores decía que tal librería era o no libre, etc. Y que Fénix sí que es libre. Pero, ¿sabes qué significa realmente eso de "software libre"? ¿Por qué es tan importante que un programa sea o no libre? ¿Quiere decir que es gratis?...En este apartado aclararé lo que significa "software libre", y

por qué es un concept tan importante, no sólo para el mundo de la informática, sino para la vida en general. Lo siguiente es un extracto de la definición de software libre obtenido de la web del Proyecto GNU (<http://www.gnu.org>), el cual se encarga de fomentar el desarrollo de software libre a nivel mundial.

*“El "Software Libre" es un asunto de libertad, no de precio. "Software Libre" se refiere a la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software. De modo más preciso, se refiere a cuatro libertades de los usuarios del software:*

*-La libertad de usar el programa, con cualquier propósito (libertad 0).*

*-La libertad de estudiar cómo funciona el programa, y adaptarlo a tus necesidades (libertad*

*1). El acceso al código fuente es una condición previa para esto.*

*-La libertad de distribuir copias, con lo que puedes ayudar a tu vecino (libertad 2).*

*-La libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie. (libertad 3). El acceso al código fuente es un requisito previo para esto.*

*Un programa es software libre si los usuarios tienen todas estas libertades. Así pues, deberías tener la libertad de distribuir copias, sea con o sin modificaciones, sea gratis o cobrando una cantidad por la distribución, a cualquiera y a cualquier lugar. El ser libre de hacer esto significa (entre otras cosas) que no tienes que pedir o pagar permisos. También deberías tener la libertad de hacer modificaciones y utilizarlas de manera privada en tu trabajo u ocio, sin ni siquiera tener que anunciar que dichas modificaciones existen. Si publicas tus cambios, no tienes por qué avisar a nadie en particular, ni de ninguna manera en particular. La libertad para usar un programa significa la libertad para cualquier persona u organización de usarlo en cualquier tipo de sistema informático, para cualquier clase de trabajo, y sin tener obligación de comunicárselo al desarrollador o a alguna otra entidad específica.*

*La libertad de distribuir copias debe incluir tanto las formas binarias o ejecutables del programa como su código fuente, sean versiones modificadas o sin modificar (distribuir programas de modo ejecutable es necesario para que los sistemas operativos libres sean fáciles de instalar). Está bien si no hay manera de producir un binario o ejecutable de un programa concreto (ya que algunos lenguajes no tienen esta capacidad), pero debes tener la libertad de distribuir estos formatos si se encontrara o se desarrollara la manera de crearlos.*

*Para que las libertades de hacer modificaciones y de publicar versiones mejoradas tengan sentido, debes tener acceso al código fuente del programa. Por lo tanto, la posibilidad de acceder al código fuente es una condición necesaria para el software libre.*

*"Software libre" no significa "no comercial". Un programa libre debe estar disponible para uso comercial, desarrollo comercial y distribución comercial. El desarrollo comercial del software libre ha dejado de ser inusual; el software comercial libre es muy importante."*

Para que un programa sea considerado libre ha de someterse a algún tipo de licencia legal de distribución, entre las cuales se encuentran la licencia GPL (General Public License), o la LGPL, entre otras –hay muchas: el tema de las diferentes licencias es un poco complicado a nivel legal: son como los contratos de jugadores de fútbol: con muchas cláusulas; si quieres saber más, mírate <http://www.opensource.org/licenses/category> -. Estas licencias son genéricamente conocidas como licencias GNU y fueron redactadas en su día por la Free Software Foundation, fundación que se encarga de mantener con fuerza este movimiento.

Supongo que conocerás, al menos de oído porque lo he nombrado varias veces, a Linux, la estrella de los programas libres. Linux es un sistema operativo competencia de Windows, libre y gratuito. Es decir, que en vez de que tu ordenador arranque en Windows –que se supone que lo has tenido que pagar-, podría arrancar Linux gratuitamente, accediendo igualmente a multitud de programas de todo tipo (ofimática, multimedia, juegos, etc). Técnicamente, un sistema operativo es un programa que se comunica directamente con el hardware de la computadora (procesador, memoria, discos duros) y hace de intermediario entre éste

y los demás programas de todo tipo que se puedan instalar en la máquina, ofreciendo a cada programa los recursos hardware que necesita en cada momento. Cuando decimos que un programa es portable o no, lo que decimos es que puede ser utilizado con varios intermediarios (con varios sistemas operativos) o no. Linux es una implementación escrita básicamente en C del sistema operativo UNIX (uno más de entre los numerosos clónicos del histórico Unix), que sale muy bien parado al compararlo con otros sistemas operativos comerciales. Comenzó como proyecto personal del -entonces estudiante- Linus Torvalds, pero a estas alturas el principal autor es la red Internet, desde donde una gigantesca familia de programadores y usuarios aportan diariamente su tiempo aumentando sus prestaciones y dando información y soporte técnico mutuo. La versión original -y aun predominante- comenzó para PCs compatibles (Intel 386 y superiores), existiendo también en desarrollo versiones para prácticamente todo tipo de plataformas hardware: PowerPC, Sparc, Alpha, Mips, etc.

Si quisieras ver qué tal pinta eso de tener un Linux en tu ordenador, lo puedes descargar de multitud de sitios en Internet e instalarlo. Ya que Linux es libre, ha habido gente que ha adaptado el Linux original a sus propias necesidades, de forma que actualmente hay muchas versiones -llamadas distribuciones- diferentes, aunque todas ellas son compatibles a nivel básico y más o menos semejantes. Algunas distribuciones importantes son:

**Ubuntu** (<http://www.ubuntulinux.org> ó <http://www.ubuntu-es.org>)

**Fedora** (<http://fedoraproject.com>)

**Debian** (<http://www.debian.org>)

**OpenSuse** (<http://es.opensuse.org>)

**Mandriva** (<http://www.mandriva.com>)

**Gentoo** (<http://www.gentoo.org>)

Si quieres ver más distribuciones, visita <http://www.distrowatch.com>

Pero es más: si no te atreves a instalarte tu Linux en casa porque temes destrozar tu Windows de toda la vida (que puede pasar si no vas con cuidado), existe la posibilidad de descargarte distribuciones Linux Live-CD. Estas distribuciones las puedes grabar en un CD y no hace falta instalar nada en el ordenador para que funcionen: simplemente metes el CD en la lectora, arrancas el ordenador y ya estás en Linux. Cuando te canses, quitas el CD y al volver a arrancar, aquí no ha pasado nada: continúa Windows igual que estaba. La distribución Live-CD reina es Knoppix (<http://www.knoppix.org>), muy aconsejable para iniciarse en el mundo Linux, pero también hay versiones Live-CD de las distribuciones "estándar", como Ubuntu o Fedora.

Si quieres ver más distribuciones Live-CD, clasificadas además según su temática (ya que existen distribuciones específicas para trabajar en ámbitos concretos, como por ejemplo distribuciones enfocadas a juegos, a investigación científica, a hacking, a diagnóstico del PC, a recuperación de desastres, a multimedia, etc) visita <http://www.livecdlist.com>

Y bueno, finalmente resulta que Fénix es libre. ¿Qué quiere decir eso? Pues que cualquiera que quiera -y sepa, ojo!- puede ser parte partícipe del desarrollo de lenguaje. Es decir, tú mismo puedes contribuir a mejorar el lenguaje Fénix. Fénix no tiene dueño que diga cómo ha de ser. Todo el mundo es dueño de Fénix, y todos pueden hacer que evolucione como convenga. Esto va mucho más allá de si Fénix es gratis o no. Lo importante es que todos podemos acceder al código fuente de Fénix (es decir, Fénix no es más que un programa más: accediendo a su código podemos ver cómo está hecho por dentro) y ver qué mejoras o características se pueden implementar, y hacerlo, de tal manera que se cree una comunidad de personas que colaboren mutuamente para llevar a cabo todo este gran proceso altruista. Ésa es la gran diferencia de Fénix con los demás lenguajes de videojuegos: no es una caja secreta.

## Algunos recursos web sobre programación de videojuegos

Algunos sitios web (algunos además incluyen foros de usuarios que te podrán echar una mano) que pueden ayudar a iniciarte en la programación de juegos son:

<a href="http://www.vjuegos.org">http://www.vjuegos.org</a>	Comunidad Iberoamericana de Desarrolladores de Videojuegos. Portal en español enfocado a impulsar el desarrollo de videojuegos a nivel profesional
<a href="http://www.gamasutra.com">http://www.gamasutra.com</a>	El portal más importante sobre desarrollo de videojuegos a nivel profesional
<a href="http://www.gamedev.net">http://www.gamedev.net</a>	Excelente página para desarrollar videojuegos con cualquier librería.
<a href="http://www.devmaster.net">http://www.devmaster.net</a>	Web llena de recursos para programadores de videojuegos (foro -muy útil-, artículos, noticias, wiki, software...)
<a href="http://www.gdmag.com">http://www.gdmag.com</a>	Revista para Desarrolladores de Videojuegos
<a href="http://www.gdconf.com">http://www.gdconf.com</a>	Congreso Internacional de Desarrolladores de Videojuegos
<a href="http://www.igda.org/">http://www.igda.org/</a>	International Game Developers Association
<a href="http://www.adva.com.ar">http://www.adva.com.ar</a>	Asociación de Desarrolladores de Videojuegos Argentina
<a href="http://nehe.gamedev.net">http://nehe.gamedev.net</a>	Página con tutoriales sobre OpenGL y más.
<a href="http://www.webgamebuilder.com">http://www.webgamebuilder.com</a>	Portal donde podrás descargar múltiples herramientas (entre otras, el Blitz3D) para diseñar tus propios juegos, enfocados preferentemente a la web, y consultar los foros relacionados.
<a href="http://www.stratos-ad.com">http://www.stratos-ad.com</a>	Punto de encuentro para desarrolladores de videojuegos hispanos. Incluye bolsa de trabajo
<a href="http://www.codepixel.com">http://www.codepixel.com</a>	Artículos y tutoriales sobre programación gráfica y animación 3D.
<a href="http://www.ambrosine.com/resource.html">http://www.ambrosine.com/resource.html</a>	Listado exhaustivo de enlaces a distintos programas creadores de juegos tipo GameMaker, diferentes lenguajes de programación de videojuegos, recursos como música, sprites, fondos, etc listos para descargar

Mención aparte requiere la estupenda web "The Game Programming Wiki" (<http://www.gpwiki.org>), completísimo sitio donde se pueden encontrar artículos y referencias relacionados con los más dispares aspectos de la programación de videojuegos: desde exhaustivísimas y completísimas comparativas de diferentes lenguajes, librerías gráficas y motores gráficos, hasta tutoriales sobre planificación y diseño de proyectos; desde listados interminables de herramientas de diseño gráfico/creación multimedia e IDEs hasta especificaciones oficiales de formatos de archivos; desde formales artículos matemáticos que tratan sobre simulaciones físicas hasta direcciones legales de industrias del videojuego; desde entradas a diferentes portales de comunidades de desarrolladores hasta tutoriales específicos para Allegro, SDL, OpenGL, DirectX, DevIL, OpenAL, Java, C/C++, C#, Lua...; desde manuales de programación genéricos avanzados (estructuras dinámicas de datos, patrones de diseño, etc) hasta artículos sobre los diferentes métodos de Inteligencia Artificial o los algoritmos de encaminamiento tipo A\*, etc, etc. IMPRESCINDIBLE.

## Conceptos básicos de multimedia para el principiante:

### **Gráficos**

Seguro que tarde o temprano necesitarás imágenes para tu videojuego, y seguro que también necesitarás tratarlas (colorearlas, enfocarlas, ampliarlas, encuadrarlas, trucarlas...) La primera pregunta es obvia: ¿de dónde podemos sacar las imágenes que necesito?

Si necesitas una imagen determinada puedes hacer una búsqueda en Internet, y si la tienes ya en la pantalla puedes hacer una impresión de toda la pantalla o de una parte para convertirla en una imagen (ya sabrás que con la tecla "ImpPant" colocarás lo que se ve en la pantalla en el portapapeles de Windows). Si dispones de la imagen impresa (en libros, revistas, fotografías...) puedes digitalizarla con un escáner. Si dispones de una cámara fotográfica digital puedes crear imágenes digitales a partir de la realidad. Si dispones de una cámara de video digital puedes extraer "fotogramas" o "encuadres" con el programa adecuado. Si tienes tarjeta capturadora de video puedes extraer fotogramas de cualquier grabación en video o de la señal de video de una cámara. Y siempre tienes el recurso de crear tú mismo la imagen con un programa de edición gráfica, ayudándote si quieres de algún periférico como una tableta gráfica, para poder dibujar mejor.

Una vez digitalizada, la imagen se convierte en un fichero. Dependiendo del tipo de fichero con que se guarde conservará características más o menos fieles a su fuente, tendrá más o menos calidad, será más o menos fácil retocarla sin perder calidad, será más o menos fácil de interpretar por cualquier ordenador... Todos estos factores vendrán determinados por 4 parámetros diferentes: la profundidad de color, la resolución, el peso (el tamaño que ocupa en el disco duro) del fichero de la imagen, y el formato de la imagen.

#### **Profundidad de color:**

La tarjeta de vídeo proporciona la comunicación necesaria entre el PC y el monitor. Recibe las señales digitales provenientes del procesador y las convierte a un formato analógico que pueda utilizar el monitor para crear una imagen visible. El software que se está ejecutando en el sistema transmite a la tarjeta información sobre los píxeles individuales que tienen que presentarse en la pantalla. La tarjeta de vídeo es en realidad una computadora ella sola: tiene su propio procesador y su propia memoria, pudiendo realizar la mayor parte de sus funciones de forma independiente del procesador principal del sistema. Las capacidades de la tarjeta de vídeo pueden tener un efecto importante en el rendimiento global de un PC; puede suceder que un equipo con el procesador más rápido y con una gran cantidad de memoria funcione con lentitud con una tarjeta de vídeo por debajo del estándar: esto es especialmente cierto con las aplicaciones intensivas de vídeo, especialmente con los juegos.

Un píxel es cada punto de luz que emite la pantalla; ésta está formada por multitud de ellos. La profundidad de color se refiere a la cantidad de bits que necesita cada píxel para guardar información de la imagen. La cantidad de memoria que incorpora la tarjeta de vídeo es la que determina las resoluciones de pantalla y profundidades de color que un sistema puede soportar. Actualmente, gracias a la avanzada capacidad gráfica de las tarjetas, casi todas las imágenes son del tipo "true color" (color verdadero). Eso quiere decir que para cada píxel se necesitan 24 bits de memoria gráfica, o lo que es lo mismo, 3 bytes (8 bits corresponden a un byte), 8 bits para cada canal de color RGB "Red, Green, Blue" (RVA: Rojo, Verde, Azul), ya que la formación de color en la imagen se basa en la mezcla de estos tres colores primarios. La profundidad color verdadero proporciona una paleta de colores casi real para el ojo humano. No obstante, cuanto más profundidad de colores tenga una imagen mejor se verá (más información habrá para representar cada píxel), pero más espacio ocupará en la memoria de la tarjeta, y de rebote, más espacio ocupará el fichero en disco.

Los datos expuestos en el cuadro siguiente ayudarán a entenderlo mejor:

<i>Número de Bits</i>	<i>Número de colores</i>
1	2 (blanco y negro)
2	4
4	16
8	256
16	65.536
24	16 millones

### **Resolución:**

La resolución define la cantidad de píxeles que contiene una imagen, en términos de anchura y altura de ésta. La resolución más corriente en los adaptadores de video fabricados actualmente es de 1024x768 píxeles, lo que significa que una imagen que ocupe toda la pantalla estaría formada por 30.720 píxeles. La relación más corriente entre el ancho y el alto de la imagen es de 4:3 de manera que los adaptadores que proporcionan una resolución mayor lo hacen utilizando la misma proporción. En general, utilizaremos una resolución de 640x480 píxeles en monitores de 14", 800x600 en monitores de 15", 1024x768 en monitores de 17", 1280x1024 con un monitor de 21". Evidentemente, a más resolución, más pequeñas se ven las cosas: los píxeles son más pequeños. Cuanto más elevado es el número de píxeles existentes por unidad de superficie en pantalla, mayor es la sensibilidad y la calidad de la imagen.

La memoria de la tarjeta gráfica está formada por bits que están dispuesto –teóricamente- en tres dimensiones: al altura, que es el número de píxeles desde la parte superior a la parte inferior de la pantalla; la anchura, que es el número de bits desde la izquierda hasta la derecha de la pantalla, y la profundidad, que es el número de bits de memoria utilizados para cada píxel. Si una tarjeta utiliza por ejemplo un color de 4 bits, dedica cuatro bits de memoria para cada píxel. Esto quiere decir que cada píxel puede tener 16 colores –combinaciones- posibles. Para utilizar esta profundidad de color con una resolución de 640x480 el adaptador tiene que tener 1.228.800 (640x480x4) bits, que equivalen a 150 kilobytes (KB) de memoria. Dada una tarjeta gráfica con una determinada memoria, sólo se podrán conseguir combinaciones de las tres dimensiones (anchura, altura, profundidad) que puedan ser admitidas por la memoria que tenga. Para los estándares actuales, una resolución de 640x480 con 16 colores es el mínimo admisible. Se conoce como resolución VGA estándar, y los sistemas la utilizan normalmente de forma predeterminada mientras no disponga de un controlador (un "driver") de video diseñado específicamente para la tarjeta instalada.

### **Peso:**

Siempre que necesites reducir el peso de una imagen tendrás que renunciar a alguna característica de la imagen original: tendrás que prescindir de algunos datos (compresión con pérdida), o guardarla con menos colores de los que tiene en realidad (bajar la profundidad de color), o hacer que se componga de menos puntos de color que el original, y por tanto, que sea más "granulosa" (bajar la resolución) o hacerla más pequeña en tamaño (bajar los valores de las dimensiones de altura y anchura).

### **Formato:**

El formato de la imagen es la manera como se codifica la información de ésta "físicamente" dentro de un fichero. Hay dos grandes familias de imágenes según su formato: los gráficos vectoriales y los mapas de bits.

#### Gráficos vectoriales

Los ficheros con gráficos vectoriales contienen instrucciones (vectores) con información

matemática sobre las posiciones, color, etc de las líneas y curvas que contienen las imágenes; o sea, describen las imágenes a partir de sus características geométricas. La visualización de la imagen en la pantalla se basa en el cálculo de estos vectores y la consecuente reconstrucción, por parte del programa adecuado, de las formas y los colores.

<i>Inconvenientes</i>	<i>Ventajas</i>
*Cuanto más complejas son las imágenes, el ordenador tarda más tiempo en calcularlas y en representarlas.	*Son independientes de su tamaño de visualización, es decir, se escalan automáticamente para aparecer nítidas (a eso se le llama no perder definición). Cuando editas o modificas la posición, forma, tamaño y color de un gráfico vectorial, éste no pierde calidad ya que lo que modificas son las propiedades de las líneas y curvas que lo definen.
*No son aptos para mostrar fotografías o imágenes complejas.	*Al estar definidas por vectores, las imágenes no pesan tanto en comparación con los mapas de bits, aunque esto depende mucho de la imagen y de la calidad que se desee: las imágenes formadas por colores planos o degradados sencillos son más factibles de ser vectorizadas. A menor información para crear la imagen, menor será el tamaño del fichero.
	*Cada pieza de la imagen puede ser manipulada separadamente. Es posible mover objetos individuales alrededor de la pantalla, alargarlos, rotarlos, duplicarlos o introducir distorsiones, etc.
	*Algunos formatos permiten animación. Está ser realiza de forma sencilla mediante operaciones básicas como traslación o rotación y no requiere un gran acopio de datos.

Los formatos de los archivos vectoriales ( y sus correspondientes extensiones) pueden ser muy variados: PS y PDF (formato PostScript), SVG (el estándar internacional), SWF (Flash), WMF (Windows MetaFiles), CDR, DFX, ...

#### Mapa de bits

En contraste, se encuentran los gráficos formados por una cuadrícula de puntos de color (píxels): los bitmap, también llamados gráficos rasterizados. Una imagen de este tipo está definida por la posición y por el color de cada uno de los píxeles que la configuran. Al modificar una imagen de mapa de bits se modifican los píxeles y no las líneas o curvas. La representación del gráfico en pantalla es pues un mosaico formado por piezas de colores diferentes.

<i>Inconvenientes</i>	<i>Ventajas</i>
*Al depender del número de píxels o resolución, estas imágenes pueden perder calidad al ser ampliadas (es decir, al escalarlas pierden definición).	*Estas imágenes tiene la ventaja de reproducir rápidamente y con mucha fidelidad gradaciones sutiles de sombras y colores. Los bitmaps son típicamente usados para reproducir imágenes que contienen muchos detalles, sombras y colores: fotografías, negativos de películas y otras ilustraciones.

\*En comparación, pesan más que las imágenes vectoriales, ya que los bitmaps contienen información específica sobre cada píxel representado en la pantalla.

Los formatos más corrientes de mapa de bits son GIF, JPG, BMP, PCX, TIF...

Además, cada programa de retoque de imágenes suele tener un formato propio (por ejemplo, el Paint Shop Pro tiene el formato \*.PSP que sólo entiende él).

Comentemos los formatos de mapa de bits más importantes:

### Formato GIF

\*Se desarrolló específicamente para gráficos online, los cuales requieren tener poco peso para poder ser visualizados y descargados rápidamente.

\*Admiten transparencia. Eso quiere decir que tienen la posibilidad de convertir en transparente o invisible un solo color, de manera que el fondo que tenga ese color sea invisible.

\*Permiten hacer animación con una técnica de poner muchas imágenes en el mismo archivo GIF.

\*-Utiliza un algoritmo de compresión sin ningún tipo de pérdida. El formato \*.GIF consigue comprimir las imágenes rebajando el número de colores, o sea, rebajando la profundidad de color (es decir, la cantidad de información sobre la imagen que cada píxel necesita, expresada en bits). Así hablaremos de imágenes de 32 bits o 24 bits ("color verdadero"), 16 bits (miles de colores), 8 bits (256 colores), de 4 bits (16 colores) o de 1 bit (monocromos). Como hemos comentado antes, cuanto más profundidad de color tenga una imagen más información sobre color tendrá que tener; por tanto, será de más calidad pero también ocupará más espacio en la memoria y el disco.

\*Los gráficos están limitados a una paleta de 256 colores (8 bits).

**Importante:** Hay que tener claro el concepto de paleta gráfica.

Cuando se elige una/s paleta/s determinada/s para usarla/s en nuestro programa –si no se hace esto, se elige por defecto la paleta genérica del sistema Windows–, al mostrar las imágenes cada píxel tendrá un valor asociado que representa un índice dentro de esa paleta o tabla de color. Esa tabla de colores indexados se podrá construir según convenga, o bien distribuyendo en la paleta uniformemente unos colores seleccionados, disponibles a lo largo del espacio de color RGB, desde el violeta al rojo pasando por todos los colores intermedios (aunque hay que tener en cuenta que la cantidad de colores susceptibles de ser elegidos para incluirse en una paleta depende de la profundidad de color usada en ese momento), o bien llenando la paleta de gradaciones sutiles de un mismo color, y utilizar una u otra paleta allí donde sea necesario. Por ejemplo, si vamos a pintar un paisaje nevado, será más eficiente utilizar una paleta llena de colores con diferentes tonalidades de blanco, azul... Si vamos a pintar una selva, será mejor disponer de una paleta especializada en colores verdes, marrones... Así pues, resumiendo, cada píxel tendrá asociado en cada momento un valor numérico que es un índice dentro de la paleta, el cual le dice al ordenador qué color utilizar cuando se visualiza dicho píxel.

Por ejemplo, en un modo de vídeo paletizado de 1 bit, cada píxel sólo puede tener dos valores posibles (0 ó 1), con lo que las imágenes que utilicen este tipo de paletas sólo podrán ser en blanco y negro. Con una paleta de 4 bits, cada píxel podrá tener un color dado por una de las 16 posiciones posibles en esa paleta. En un modo de vídeo paletizado de 8 bits, cada píxel es un valor que varía de 0 a 255. Por lo tanto, la paleta puede contener 256 colores y punto, que es lo que ocurre con las imágenes en formato GIF; la gracia está en crear una paleta de 256 colores que representen los colores que más usaremos.

### Formato JPEG (JPG)

\*Se desarrolló como un medio para comprimir fotografías digitales.

\*El algoritmo que utiliza descarta algunos bloques de datos de las imágenes cuando las comprime. Por esta razón es mejor sólo guardar una vez la imagen con compresión JPEG, porque cada vez que se utilice este algoritmo se eliminarán más datos.

\*Permite escoger el grado de compresión: cuanto más compresión más pérdida y el peso del fichero es menor.

\*Las imágenes en formato JPG pueden conservar una profundidad de color de millones de colores, reduciendo considerablemente el peso del fichero. Eso hace que este formato sea muy utilizado en Internet para facilitar las descargas, en las cámaras digitales (que tienen que poner cuantas más fotografías mejor en un espacio reducido), i , en general, siempre que se quiera trabajar con imágenes con muchas gradaciones de sombras y colores y se precise ahorrar espacio.

### Formato BMP

\*El formato gráfico estándar de mapa de bits sin compresión usado en el entorno Windows. Con la herramienta MSPaint se pueden crear fácilmente ficheros BMP que tengan diferentes paletas: de 1, 4, 8 bits...

### Formato PNG

\*Utiliza un algoritmo de compresión sin pérdidas, libre de patentes, que permite una mejora del 5-25 % mejor que el GIF.

\*Que sea libre de patentes quiere decir que cualquier desarrollador puede diseñar programas que generen imágenes PNG, y utilizarlas sin ningún tipo de restricción. Esto no es así con otros formatos como el GIF: si un desarrollador quiere escribir un programa que en algún momento hace uso del algoritmo de creación de imágenes GIF, ha de pagar una suma económica por ello. Los orígenes del formato Portable Network Graphics se remontan a 1977 cuando dos investigadores israelitas publicaron un algoritmo para la compresión sin pérdidas de imágenes que se denominó LZ78. En 1983 la empresa Unisys desarrolló una variante de LZ78 que se denominó LZW. Éste solicitó una patente por el algoritmo que le fue concedida por la oficina de patentes de EEUU en 1985. En 1987 la empresa CompuServe diseñó el formato GIF, que utiliza el algoritmo LZW para comprimir las imágenes. Concedida la patente a Unisys y conociendo ésta en que era utilizado en el formato GIF en 1993 inició acciones contra CompuServe. En 1994 CompuServe y Unisys llegaron a un acuerdo para pedir royalties al software que soporte GIF, por lo que la creación de archivos GIFs sin permisos puede ser perseguida por la ley. Este hecho desató iras en los grupos de usuarios de Internet, uno de los cuales comenzó el desarrollo de un nuevo formato libre: el PNG.

\*No solamente el formato PNG está libre de patentes para su creación y uso, sino que es un formato libre. Es decir, la especificación del formato es pública y cualquiera puede modificarlo para mejorar el algoritmo de compresión de manera que toda la comunidad se beneficie de los avances en la evolución del formato.

\*Existen dos profundidades de color para este formato: 8 y 16 bits lo que permiten 256 y 16,7 millones de colores respectivamente.

\*Una de las principales mejoras que incluye este formato es el denominado canal alfa que permite hasta 256 niveles de transparencia en el PNG-24 (24 bits), es decir que es posible tener transparencias con distintos grados.

\*Con este formato no pueden realizarse animaciones aunque se ha desarrollado uno nuevo basado en este MNG (Multiple Network Graphics) que si lo permite.

**Importante:** Debido a ser un formato libre ( y a que es de gran calidad), el formato PNG es el formato nativamente soportado por Fénix. Esto quiere decir que Fénix está específicamente diseñado para

trabajar con imágenes PNG, y por tanto, preferentemente todos los gráficos que se utilicen en un juego hecho en Fénix tendrían que ser PNG. Aunque es verdad Fénix soporta otros formatos, el recomendado sin duda es el PNG, por su integración y compatibilidad con la plataforma Fénix, su calidad intrínseca y su licencia libre.

## Video

### **Procesamiento del video digital:**

Las 4 etapas básicas en el procesamiento del video digital son:

-Captura/Digitalización: el proceso de captura implica disponer de un ordenador con una placa de captura. Su función es la conversión analógica-digital de la señal de video para la grabación al disco duro y posterior edición. La digitalización del video es un proceso que consiste en grabar la información en forma de un código de dígitos binarios a través del cual el ordenador procesa los datos electrónicamente. Una buena digitalización requiere una acción combinada de las prestaciones del ordenador y de la tarjeta digitalizadora (capturadora). La calidad se manifiesta en la velocidad de digitalización, variando de 1 a 10 Mb/s, el número de colores y las dimensiones de la imagen. De tarjetas capturadoras hay de todo tipo en el mercado: normalmente permiten entrar imágenes y sonido desde una cámara de video analógica o digital, o bien un magnetoscopio, y exportar a un televisor y magnetoscopio, o al disco duro directamente, de manera conjunta o por separado. Es necesario, no obstante, hacer los correspondientes nexos a través de un cableado adecuado dependiendo del caso (RCA o S-Video para conexiones analógicas, Firewire – IEE1394- para conexiones digitales, etc).

-Edición: una vez realizada la digitalización, se dispone de herramientas informáticas que facilitan la edición (el tratamiento) de imagen y sonido y la generación de efectos: ordenar, editar con corte para desprenderse de los trozos no deseados, inclusión de transiciones, filtros, transparencias, títulos, créditos...

-Compresión: si se prevee difundir secuencias de video, a las que son en formato digital se les ha de aplicar un proceso de compresión para reducir el tamaño de los ficheros. Existen diversos estándares de compresión, que se escogerán en función de la plataforma final de visualización.

-Difusión: una vez acabado el proceso de edición, es cuando se entra en la última fase : la exportación a un destino particular, que puede ser a una cinta de video, un CD, un DVD, Internet...

### **Formatos de video digital:**

En el mundo de la informática personal se pueden encontrar principalmente los siguientes formatos de video digital:

-Quicktime: sistema de compresión y visualización de video digital desarrollado por la empresa Apple. Es un formato biplataforma (Mac/PC) porque dispone de programas visualizadores para los dos entornos. Los ficheros se llaman movies y tienen extensión MOV.

-Video for Windows (AVI): sistema de compresión y visualización de video digital en ordenadores PC con el entorno Windows, desarrollado por la empresa Microsoft. Los ficheros tienen la extensión AVI (Audio Video Interlaced). Es importante saber que cada fichero AVI tiene un códec específico, propietario o genérico, y por tanto, no todos los archivos AVI son iguales. Por otra parte, es un formato ideal para editar video aunque una excepción son los archivos \*.avi codificados con el códec DivX, que es un formato final no editable (ya que, aunque tenga extensión avi, el video tiene estructura

interna tipo mpeg).

-**MPEG**: destacaremos el subformato MPEG-1 (Moving Picture Expert Group) que fue el estándar establecido en 1992. Se orienta al archivo y distribución de video digital. El MPEG-1 está pensado para un flujo de imágenes de 352x288 píxeles con 16 millones de colores a 1.5Mbits/s (calidad VHS). Posteriormente apareció el MPEG-2, pensado para la TV digital (cable y satélite) y es el formato utilizado en el DVD. Las pruebas actuales permiten distribuir secuencias de video de 720x485 píxeles. Es formato final, nunca para editar video.

-**Streaming** –Video para la red internet: con extensión \*.rm, \*.mov y \*.wmv (y algun derivado). Siempre es formato final, no editable, y de muy poca calidad –apto para la visualización en tiempo real a través de la red).

### **Códecs:**

En la actualidad, el gran problema que tiene el video digital es el gran peso de los archivos generados, aumentando éste todavía más cuando en el momento de la edición se añade sobre las imágenes un texto, un filtro o cualquier otro efecto especial.

Los datos siguientes ayudarán a entenderlo mejor: un fotograma de video con millones de colores, sin ningún tipo de compresión, ocupa alrededor de 1,2 Mbit. Un segundo de video (25 fotogramas), sin comprimir, alrededor de 30Mbit. Un minuto (1.500 fotogramas), 1.5 Gbit aproximadamente. Es evidente que se necesitaría un disco duro con mucha capacidad para poder almacenar todos los cplis que se quieren editar, mucha memoria y un procesador muy rápido.

El problema de capacidad y la velocidad de transmisión se resuelve mediante la compresión, imprescindible para reducir el peso de las imágenes de diferentes algoritmos de compresión, llamados códecs, compresores o codificadores, elementos que forman parte del software de las tarjetas capturadoras de video y audio.

Los codificadores más estandarizados actualmente son los siguientes:

-**El M-JPEG (Motion-JPEG)**: Es una variedad del JPEG (Joint Photographic Experts Group), un sistema de compresión de fotografía que ha servido como modelo a muchos de los sistemas de compresión de video, entre los cuales hay el M-JPEG, el Indeo, el Cinepak, etc, que consideran el video como una sucesión de fotografías. Cuando comprime, digitaliza toda la información de cada fotograma de video, independientemente de los otros. Es lo que se conoce como Intraframe. Aunque ocupa mucho espacio, es el codificador que garantiza mayor calidad. Por esta razón, es el más usado cuando se ha de digitalizar material original que se ha de editar. El problema que presenta este sistema es la falta de compatibilidad entre las diferentes tarjetas digitalizadoras (capturadoras) existentes con el codec M-JPEG, ya que cada fabricante ha desarrollado su propia variante del formato.

-**El MPEG (Moving Picture Experts Group)**: Es un sistema de compresión de video y audio implantado por la Unión Internacional de Telecomunicaciones. Al contrario que el anterior, es un codificador universal, no presenta incompatibilidades. Comprime completamente diferentes fotogramas a la vez, y crea dependencia entre ellos. Es lo que se conoce como Interframe. Es un sistema más complejo. En el proceso de digitalización, comprime unos fotogramas principales, que son los de video y se llaman keyframes; y otros, secundarios, que contienen la información de las imágenes y se llaman deltaframes. Ahorra mucho espacio, pero no es muy utilizado para la edición. El hecho de no digitalizar completamente cada frame por separado comporta una pérdida de detalles no procesados, y consecuentemente una recogida de información incompleta que interfiere considerablemente en la calidad final, con imágenes poco definidas. En cambio, por su característica universal, es muy apropiado para ver videos ya editados.

Hay 3 tipos de MPEG, algunos ya se han comentado antes.

\*MPEG 1: Diseñado para poder introducir video en un CD-ROM, con un calidad de imagen y sonido razonable.

\*MPEG 2: Pensado para la televisión digital y para los DVD, por su alta resolución

\*MPEG 4: Con un formato y una resolución baja para facilitar su transmisión, está ideado para videoconferencias e Internet, básicamente.

A parte de estos códecs, hay muchos más que hacen implementaciones propias del estandar MPEG o que son totalmente diferentes. Algunos de estos codificadores ya están instalados por defecto en el sistema operativo del ordenador. Otros se instalan automáticamente cuando se instala la tarjeta capturadora de imagen y sonido, o bien algún programa de edición de video. Hay que tener presente que éste es un mundo en continua evolución, y que , por tanto, conviene irse actualizando e incorporando los nuevos códec que aparecen en el mercado según las necesidades.

La elección de un codec u otro depende del destino final de la película (cinta magnética, CD, Internet, videoconferencia, etc), el método de compresión que se hará servir y la forma como gestiona la imagen y el sonido. Códecs importantes son el Cinepak, el Sorenson, el Indeo... Si alguna vez tuvieras un video que no lo puedes visualizar correctamente debido presuntamente a que te falta el códec adecuado, para saber qué codec concreto es el que te falta y poderlo ir a buscar a Internet y descargarlo (un buen sitio para probar es buscarlo en <http://www.free-codecs.com> ), puedes utilizar la estupenda herramienta Gspot, descargable desde <http://www.headbands.com/gspot> .

### **3D**

Antes de nada, hay que aclarar que Fénix es un lenguaje que -a día de hoy- sólo permite programar juegos en 2D, por lo que no en este manual no tocamos nada del mundo tridimensional. Sin embargo, en apartados anteriores han aparecido salpicados conceptos propios del mundillo 3D tales como aceleración 3D, renderizado, etc, que es posible que te suenen a chino y probable que vuelvan a aparecer en este texto más adelante, por lo que es bueno explicarlos un poquito en este apartado para así tener las cosas claras.

#### **¿Qué es animar?**

La animación, ya sea 2D o 3D, se puede entender como una simulación de la realidad pero con unas leyes y unos objetos que le son propios. De hecho, la palabra viene del latín "animus", que quiere decir "alma". La animación es posible gracias a la llamada "persistencia retiniana", que es el efecto óptico por el cual las imágenes quedan durante unos instantes grabadas en nuestra memoria. Si hacemos pasar rápidamente diferentes imágenes ante nuestros ojos éstas se mezclan creando una sensación de movimiento. Para crear una buena sensación de movimiento hace falta pasar aproximadamente un mínimo de doce imágenes por segundo. A mayor número de imágenes por segundo la definición del movimiento mejorará. El cine como fotografía en movimiento necesita de un mayor número de imágenes por segundo, sobre todo en su grabación. Las primeras películas se rodaban a 16 o menos imágenes por segundo, de aquí el efecto estroboscópico de su movimiento. En la actualidad, tanto el cine como la televisión emiten y se graban a un mínimo de 24 imágenes por segundo ( o fps, frames por segundo).

La animación es el arte del movimiento: el movimiento parte de dos situaciones: A y B, y un desplazamiento con un tiempo determinado. Para ir de A hasta B necesitaremos un número determinado de pasos intermedios. Si queremos que el movimiento sea uniforme en la misma unidad de tiempo los espacios tendrán que ser iguales. Si lo que queremos es dar la sensación de aceleración o desaceleración tendremos que juntar o separar los espacios. La línea que marca la dirección del movimiento y sobre la cual marcamos los pasos se denomina línea directriz. Además del desplazamiento sobre el plano, las figuras generan unos desplazamientos sobre ellas mismas, es decir, hace falta crear diferentes posiciones de la figura. Hace falta tener claro qué son las posiciones inicial y final que denominaremos posiciones clave (en dibujos animados se los denomina layouts, y en infografía keyframes). Estas posiciones nos muestran las

características de lo que será el movimiento. El número de intervalos que queramos poner entre las dos posiciones principales dependerá de la velocidad de movimiento y la distancia, entre ellos del ritmo de aceleración o desaceleración que le queramos dar en cada momento.

Las animaciones 3D se ven actualmente sólo en 2D, pero su creación agrega realismo a las texturas, la iluminación, la profundidad de campo y otros muchos efectos que hacen que las imágenes parezcan más reales. La animación puede generarse, o bien en tiempo real, en el cual cada cuadro se crea con el tiempo del espectador o en tiempo simulado. Las animaciones de tiempo real son a menudo muy lentas puesto que la potencia informática necesaria para crear animaciones de alta calidad es sumamente elevada y, por lo tanto, muy caras. Actualmente, el ordenador genera todavía los cuadros, que se imprimen y fotografían o se envían a un dispositivo de salida de vídeo. De este modo, un ordenador puede estar segundos, minutos, o horas generando cada cuadro, pero al reproducir la cinta o la película pasa cada cuadro en un fracción de segundo.

### **Proceso de animación 3D**

El proceso de animación por ordenador es una tarea compleja. En sus más amplios términos, la animación 3D se divide en las tareas de: guión y storyboard, modelaje, aplicación de texturas, animación, iluminación, sombreado y renderizado.

\*Guión y StoryBoard: La creación de una película animada casi siempre empieza con la preparación de un "storyboard", una serie de bocetos que muestran las partes importantes de la historia y que incluyen una parte del diálogo. Se preparan bocetos adicionales por establecer los fondos, la decoración y la apariencia y temperamento de los personajes. En ciertas secuencias, la música y el diálogo se graban antes de ejecutar la animación final para que la secuencia final de imágenes sea gobernada por las pistas de sonido. En otros, la serie final de imágenes se realiza primero y, por lo tanto, controla la composición y el arreglo de la música y otros efectos de sonido así como el estilo y el ritmo del diálogo. A veces, se usan ambos tipo de sincronización dentro de la misma producción

\*Modelaje: Es el proceso de crear y manipular una forma geométrica 3D para representar un objeto como en la vida real (o en la fantasía) en un ordenador, es decir, consta de ir dando forma a objetos individuales que luego serán usados en la escena. Se utilizan numerosas técnicas por conseguir este efecto que tienen coste y calidad diferentes: Constructive Solid Geometry, modelado con NURBS y modelado poligonal son algunos ejemplos. Básicamente, la técnica consiste en crear puntos en un "espacio virtual" que usa coordenadas X,Y,Z que representan anchura (X), altura (Y) y profundidad (Z). O conectar estos puntos mediante polígonos, curvas o splines -un tipo de línea especial-, según la calidad deseada. Una alternativa a la creación de modelos propios, es emplear una gran colección de modelos ya creados, que está disponible en compañías especializadas en este campo, como Viewpoint Fecha Labs, Reyes Infográfica, 4 Bytes y otras.

\*Aplicación de texturas: Es aplicar texturas a las formas 3D creadas con el modelaje. Este proceso es el que contribuye principalmente a dar realismo a la visión del modelo acabado. No es suficiente con dar simplemente el mismo color al modelo que el objeto que se desea emular. Hay otros atributos que necesitan también ser considerados, como la reflexión, la transparencia, la luminosidad, la difusividad, el relieve, el mapeo de la imagen, el lustre, la suavidad y otros.

\*Animación: Es el arte de fabricar el movimiento de los modelos 3D. Hay muchas maneras de conseguirlo que varían en complejidad, coste y calidad. Una manera muy cara y realista de animar a un ser humano es la "Captura del movimiento". Para lo cual, se ponen sensores (captadores electrónicos de cambios de posición) en una persona y cuando esta lleva a término mejor poner "realiza" los movimientos requeridos, un ordenador graba el movimiento a partir de las señales procedentes de los sensores. Los datos grabados pueden aplicarse después a un modelo 3D. Otro método de producir una fluidez en los movimientos es una técnica denominada Cinemática Inversa (normalmente IK). Diferentes objetos (modelos) se unen jerárquicamente a otras formando como una cadena. Cuando uno de los objetos se

mueve, los otros son influidos por esta fuerza y se mueven de acuerdo con los parámetros fijados, como si tuvieran un esqueleto. El animador sólo necesita poner sólo ciertas posiciones importantes conocidas como "keyframes" (cuadros clave) y el ordenador completa el resto. Con la ayuda de la técnica de keyframing, en lugar de tener que corregir la posición de un objeto, su rotación o tamaño en cada cuadro de la animación, sólo se necesita marcar algunos cuadros clave (keyframes). Los cuadros entre keyframes son generados automáticamente, lo que se conoce como 'Interpolación'.

\***Iluminación:** Es muy parecido a la de un estudio de televisión o de cine. Allí, según cómo y dónde se coloquen los diferentes tipos y cantidades de luz, queda alterada dramáticamente la percepción de la escena. La iluminación de la animación 3D utiliza los mismos principios y técnicas que en la vida real.

\***Sombreado:** Son las diferencias de color de la superficie de un objeto debidas a las características de la citada superficie y de su iluminación. También afecta al sombreado la precisión con la que cada modelo particular es renderizado. Iría después de la iluminación.

\***Renderizado:** Es el proceso de plasmar en una sola imagen o en una serie de imágenes todo el complejo proceso de cálculo matemático que se ha ido efectuado a partir del modelado, aplicación de texturas, animación, iluminación, etc. hecho hasta el momento por una escena concreta. Es convertir un modelo matemático tridimensional en una imagen (o video) de aspecto real, aplicando a cada superficie las adecuadas texturas, juegos de sombras y luces, etc... Es el acto de "pintar" la escena final haciéndola visible, generada gracias a todos los cálculos necesarios. Es obtener la forma definitiva de la animación a partir de todos los aspectos del proceso de animación 3D, dibujando el cuadro completo. Esto puede ser comparado a tomar una foto o lo caso de la animación, a filmar una escena de la vida real. Hay muchas maneras de renderizar. En orden de menor a mayor calidad son: Flat, Gouraud, Phong y Ray Tracing. Las técnicas van desde las más sencillas, como el render de alambre (wireframe rendering), pasando al render basado en polígonos, hasta las técnicas más modernas como el Scanline Rendering, el Raytracing, la radiosity o el Mapeado de fotones. El método más complejo de renderizado es el que cada cuadro necesita más tiempo por ser generado. El proceso de render necesita una gran capacidad de cálculo, pues requiere simular gran cantidad de procesos físicos complejos. En escenas muy complejas como las de "Toy story", cada cuadro de animación puede necesitar hasta 12 horas por completarse. ¡No es sorprendente que "Toy story" necesitara 800.000 horas de máquina de renderizado! El software de render puede simular efectos cinematográficos como el lens flare, la profundidad de campo, o el motion blur (desenfoque de movimiento). Estos artefactos son, en realidad, un producto de las imperfecciones mecánicas de la fotografía física, pero como el ojo humano está acostumbrado a su presencia, la simulación de dichos efectos aportan un elemento de realismo a la escena. Se han desarrollado técnicas con el propósito de simular otros efectos de origen natural, como la interacción de la luz con la atmósfera o el humo. Ejemplos de estas técnicas incluyen los sistemas de partículas que pueden simular lluvia, humo o fuego, el muestreo volumétrico para simular niebla, polvo y otros efectos atmosféricos, y las cáusticas para simular el efecto de la luz al atravesar superficies refractantes.

### **La aceleración hardware:**

Los videojuegos son una de las aplicaciones más exigentes, es por eso que a mediados de los años noventa, los desarrolladores se dieron cuenta de que los procesadores principales (CPUs) de los computadores personales no bastaban para los requerimientos de operaciones como el renderizado 3D. Fue entonces cuando a la industria se le ocurrió que los PC deberían tener un procesador adicional, que se encargara exclusivamente de las operaciones relacionadas con los gráficos. A estos nuevos procesadores se les llamó GPU o Unidades de Procesamiento Gráfico; su objetivo es precisamente el de liberar a la CPU de las operaciones relacionadas con gráficos.

Las GPUs se colocan en las tarjetas de vídeo, es lo que ahora se conoce como tarjetas aceleradoras 3D. De hecho, en el mercado se pueden encontrar tarjetas de vídeo 2D "a secas", tarjetas aceleradoras 3D "a secas" -que obviamente en un ordenador han de ir acompañadas por una tarjeta de vídeo de las anteriores - y tarjetas de vídeo con aceleradora incorporada, que proveen de cierto grado de

aceleración por hardware. Es evidente, pues, que la calidad de una tarjeta aceleradora viene dada por dos factores: la cantidad de memoria que aloja, para realizar operaciones complejas, y la potencia de su GPU.

Con la aceleración 3D por hardware, el renderizado tridimensional usa el procesador gráfico en su tarjeta de video en vez de ocupar valiosos recursos de la CPU para dibujar imágenes 3D. También se le conoce como "aceleración por hardware" en vez de "aceleración por software" debido a que sin esta aceleración 3D, el CPU está obligada a dibujar todo por sí misma usando bibliotecas de renderizado por software diseñadas para suplir esa carencia hardware, (como las bibliotecas de Mesa), lo que ocupa una considerable potencia de procesamiento. La aceleración tridimensional vía hardware es valiosa en situaciones que requieran renderizado de objetos 3D tales como juegos, CAD 3D y modelamiento.

Los gráficos 3D se han convertido en algo muy popular, particularmente en juegos de computadora, al punto que ya hemos visto que se han creado APIs especializadas para facilitar los procesos en todas las etapas de la generación de gráficos por computadora. Estas APIs han demostrado ser vitales para los constructores de hardware para gráficos por computadora, ya que proveen un camino al programador para acceder al hardware de manera abstracta, aprovechando las ventajas de tal o cual placa de video, sin tener que pasar por el molesto intermediario que representa el sistema operativo. Es por eso que una de las características de una tarjeta aceleradora es su compatibilidad con tal o cual librería gráfica que pretenda acceder a sus capacidades. Actualmente la mayoría de las tarjetas de vídeo toleran todas las librerías software nombradas en apartados anteriores (Direct3D, OpenGL,...) elegir se trata simplemente de optar por la que más nos convenga.

Pero, si en Fénix no se pueden hacer juegos 3D, ¿para qué necesitarás entonces las herramientas de modelaje 3D comentadas anteriormente? La posibilidad más inmediata es para convertir una animación 3D en un video que se pueda mostrar en el juego a modo de intro, por ejemplo. De esta manera, via video, las animaciones 3D podrán incrementar el atractivo visual de tu juego en varios puntos.

## **Sonido**

El sonido es la sensación por la cual percibimos los cambios de presión y densidad del aire que se transmiten en forma de vibraciones. Tal y como hemos visto, los aparatos digitales pueden trabajar únicamente con secuencias de cifras numéricas. Igual que pasa cuando escaneamos una imagen, por ejemplo, el cual lee los colores de los puntos de la fotografía y los transforma en valores numéricos, si queremos trabajar el sonido con el ordenador nos hará falta obtener una representación numérica de estas vibraciones. Primero hace falta convertir las vibraciones del aire en oscilaciones de una corriente eléctrica. De esto se encargan los micrófonos y otros aparatos similares. El segundo paso consistirá a medir la intensidad de esta señal eléctrica a intervalos regulares. La colección de los valores obtenidos será una representación digital del sonido.

### **El proceso de digitalización:**

En el proceso de digitalización intervienen tres factores:

\*La frecuencia con que se mide la intensidad de la señal eléctrica, que se indica en Hz (Hercio: número de lecturas por segundo). No se tiene que confundir esta magnitud con la frecuencia del sonido, donde los Hz indican el número de vibraciones por segundo.

Los valores más usuales empleados en las grabaciones digitales son:

- 11.025 Hz para grabaciones de voz
- 22.050 Hz para grabaciones de música con calidad mediana
- 44.100 Hz para grabaciones de música con alta calidad.

\*La resolución con qué se anotan los valores de las lecturas.

Siempre que se hace una medida hay un redondeo. No es el mismo pesar con unas balanzas de precisión, que nos permiten afinar hasta los miligramos, que con una báscula doméstica donde siempre se acaba redondeando a decenas o centenares de gramos. Debido al sistema binario de numeración que usan los ordenadores, tenemos dos posibilidades:

-8 bits (un byte por lectura) Permite usar una escala de 256 valores posibles. Viene a ser como pesar con una báscula doméstica.

-16 bits (dos bytes por lectura) Ofrece una escala de 65.536 valores. Sería el equivalente a las lecturas que podamos obtener de una balanza de precisión.

\*El número de canales.

La digitalización se puede hacer a partir de una señal monofónica (un solo registro sonoro) o estereofónico (dos registros simultáneos).

### **Digitalización de alta y baja calidad. Codecs:**

Los valores que escogemos para cada uno de estos tres parámetros determinarán la calidad de la digitalización, y nos indicarán también el número de bytes que necesitaremos para almacenar los datos recogidos. Los datos provenientes de una grabación digital de audio podan ocupar mucho de espacio, especialmente si la digitalización se ha realizado a alta calidad. Por ejemplo, por digitalizar una canción de 3 minutos de duración a 44.100 Hz se realizan casi 8 millones de lecturas:

$$44.100(\text{muestras por seg.}) \times 3(\text{min}) \times 60(\text{seg. cada minuto}) = 7.938.000 \text{ muestras}$$

Si la grabación es estereofónica hará falta multiplicar por 2, y si las lecturas se hacen a 16 bits (que es el normal) necesitaremos 2 bytes por almacenar cada una de las cifras recogidas. En total la grabación ocupará:

$$7.938.000(\text{muestras}) \times 2(\text{canales}) \times 2(\text{bytes por muestra}) = 31.752.000 \text{ bytes}$$

¡Es decir, casi 32 millones de bytes por sólo una canción!

Los códecs son algoritmos matemáticos que permiten comprimir los datos, haciendo que ocupen mucho menos espacio. La palabra códec viene de la contracción de las expresiones COder y DECoder. Ya hemos dicho en un apartado anterior que a vuestro ordenador hay instalados unos cuantos códecs especializados en audio, y de otros especializados en vídeo.

Siempre que se usa un códec se pierde algo de calidad, puesto que se acostumbran a sacrificar los datos que nuestros sentidos no pueden percibir (por ejemplo, suprimiendo los armónicos más agudos que quedan fuera de las frecuencias audibles por los humanos). Por esto conviene hacer el proceso de compresión una sola vez, cuando ya haguen realizado todas las modificaciones deseadas a los datos originales.

Los códecs de audio más usuales son:

\*MPEG Layer 3, también conocido como "MP3" : Es el códec más tendido. Permite comprimir el sonido digital hasta 1/10 de su medida original sin que se pierda demasiada calidad. Se utiliza en muchos tipos de dispositivos portátiles, y es el rey de los códecs en el intercambio de música por internet.

\*Ogg Vorbis: A diferencia de la MP3, que tiene un complejo sistema de patentes, este formato se basa en estándares de código abierto y libre, y la calidad es similar a de la MP3, si no mejor. El caso del codec Ogg en el tema de licencias sería lo equivalente al formato PNG en gráficos: ya que el formato MP3

está patentado su creación o utilización puede estar limitada por las restricciones que imponga el propietario de la patente (actualmente, el Instituto Fraunhofer), -básicamente el pago de royalties-, se vio la necesidad de crear otro formato de compresión de audio, con altas capacidades, que fuera libre, y gratuito. Que sea libre y gratuito quiere decir que los desarrolladores comerciales pueden escribir aplicaciones independientes que sean compatibles con la especificación Ogg sin ninguna carga económica y sin restricciones de ningún tipo. Es evidente, otra vez, que este formato de audio es el preferido para ser usado con la plataforma Fénix. Se soporta de forma nativa, con lo que la optimización está asegurada, y además su calidad es excepcional.

\***GSM**: Es el códec empleado por los teléfonos móviles. Está pensado para comprimir el sonido de la habla. Tiene una relación de compresión muy alta, pero la calidad obtenida es también muy limitada.

\***Real Audio**: Este códec es el que utiliza la empresa Real Networks en sus sistemas de transmisión de audio en tiempo real por internet.

\***PCM**: Son las iniciales de "Pulse Code Modulation". De hecho el PCM no es un códec, sino el nombre que reciben los datos de audio digital sin comprimir. Lo incluimos en esta lista por ayudar a identificar los diversos formatos de codificación de datos.

### **Formatos de almacenamiento:**

Las secuencias numéricas provenientes de una digitalización de audio se pueden almacenar y transmitir en formatos muy diferentes:

\***CDAudio**: Los CD de música que usamos a los reproductores domésticos contienen los datos provenientes de una digitalización a 44,1 KHz, 16 bits y estéreo, sin ningún tipo de compresión. En un CD de 700 Mb caben unos 80 minutos de audio.

\***Audio a chorro o streaming audio**: Consiste en la transmisión por internet de datos de audio digital comprimidos, que el receptor recibe y consume al momento, convirtiéndolos en sonido. Se utiliza, entre otras cosas, para escuchar una emisora de radio en tiempo real o programas "a la carta". Los sistemas más empleados son Real Audio y Windows Media, aun cuando empieza a haber emisoras virtuales que emiten raigs de datos en MP3 y Ogg Vorbis. Podéis localizar emisoras de todo el mundo a <http://www.radiolocator.com>

### **Ficheros:**

El más usual es desar los datos de audio digital en un fichero. Normalmente la extensión de un fichero (las tres últimas letras) indica el formato de sus datos:

**.wav**: Es la extensión que se acostumbra a emplear en el Windows para identificar los ficheros de audio digital. Proviene de la contracción de "wave" (ola en inglés). Los datos de los ficheros .wav podan estar en formato PCM (sin comprimir) o pueden haber sido comprimidos con cualquiera de los códecs disponibles para Windows.

**.ave y .aiff**: Los ficheros con extensión .ave y .aiff son los que se acostumbran a emplear en sistemas Mac y Linux. Contienen datos sin comprimir.

**.mp3 y .ogg**: Los ficheros con estas extensiones contienen datos comprimidos en formato MPEG-III o Ogg Vorbis

**.ra**: La extensión .ra se acostumbra a utilizar en los ficheros codificados con el códec Real Player.

Hay otras extensiones y formatos, pero estos son quizás los más usuales.

Los programas reproductores son capaces de reproducir todos estos formatos de ficheros, siempre que los codecs correspondientes estén instalados en el sistema. Hablando además bajo nivel, es la tarjeta de sonido la encargada de transformar la información digitalizada en un impulso eléctrico otra vez, el cual a través de unos altavoces se vuelve a convertir en sonido.

### **¿Qué es el MIDI?**

El MIDI (acrónimo de Musical Instruments Digital Interface) es un sistema estándar de comunicación de información musical entre aparatos electrónicos. Para hacer posible esta comunicación, el MIDI describe dos cosas a la vez:

-Un método para codificar la música, basado en cifras numéricas que se transmiten una última la otra.

-Un sistema de conexiones físicas entre los aparatos: cables, conectores, tensiones eléctricas, frecuencias...

Es importante tener presente que el MIDI no sirve para transmitir sonidos, sino sólo información sobre como se tiene que reproducir una determinada pieza musical. Sería el equivalente informático a la partitura, al carrete de una pianola o al tambor de agujas de una caja de música: contiene toda la información necesaria para interpretar una pieza musical, pero si la queréis convertir en sonidos necesitaréis un buen instrumento y un músico o un mecanismo capaz de interpretarla.

Actualmente todos los fabricantes de sintetizadores y aparatos musicales incorporan en sus equipos las conexiones y el circuitos atendiendo a la normativa estándar MIDI. Esto hace que siempre sea posible la comunicación entre los equipos, aunque no provengan del mismo fabricante o utilicen técnicas diferentes de síntesis y proceso de datos.

### **MIDI y sonido digital**

Tal y como se ha explicado anteriormente, hace falta diferenciar los ficheros MIDI de los ficheros de sonido digital (WAV, MP3, Ogg...). El MIDI es el equivalente a una partitura, y por esto sólo contiene las instrucciones que harán que un dispositivo compatible active los sonidos. El WAV, en cambio, contiene una descripción detallada de la ola de sonido resultante. Esto hace que la medida que ocupa un fichero MIDI sea muy menor a la de un fichero de sonido digital.

La ventaja de los ficheros de sonido digital es que ofrecen la misma calidad de sonido, independientemente del ordenador dónde los escuchamos. Por el contrario, la calidad del sonido de un fichero MIDI dependerá del sintetizador o la tarjeta encargada de interpretar la "partitura".

Otro aspecto a destacar del MIDI es su flexibilidad: la música se puede editar y modificar muy fácilmente, cambiando el tiempo, la alzada de las notas, los diferentes timbres utilizados, etc.

### **Aparatos MIDI:**

El sistema MIDI es empleado por varios tipos de aparatos musicales con funciones diversas. Podríamos agruparlos en tres grandes familias:

\*Aparatos generadores de sonido: Son los aparatos que reciben información que proviene de algún otro y la transforman en sonido. Los aparatos MIDI pueden realizar esta función principalmente de dos maneras diferentes:

-Los sintetizadores generan el sonido de una manera totalmente artificial, basándose en combinaciones de funciones matemáticas por obtener los diferentes timbres. La tarjeta de sonido

del ordenador incorpora un sintetizador MIDI.

-Los mostreadores (denominados también samplers) reproducen muestras grabadas de un instrumento tradicional. Las muestras de sonido se toman en un estudio y se almacenan digitalmente en la memoria del aparato MIDI. Posteriormente, son manipuladas por adaptarlas a diferentes niveles de intensidad y frecuencia. El sonido obtenido por este método puede tener una calidad parecida a la de una grabación en disco compacto hecho con el instrumento de dónde provienen las muestras. Algunas tarjetas de sonido de gama alta tienen, además del sintetizador, un sampler.

\*Controladores: Reciben este nombre los dispositivos especializados al emitir información MIDI. Podemos encontrar al mercado controladores que adoptan la forma de instrumentos convencionales (teclados, saxofones, flautas, guitarras, acordeones, baterías...) y de otras con diseños específicos (sensores de luz, de sonido o de movimiento, mesitas sensibles, platos giratorios...).

\*Procesadores de datos: Se especializan en recibir, almacenar, procesar y generar información MIDI. En este grupo encontramos los secuenciadores y los ordenadores.

Algunos aparatos integran en una misma unidad elementos que pertenecen además de un grupo de los descritos. Por ejemplo, un sintetizador con teclado es a la vez un generador de sonido y un controlador, y puede traer también un secuenciador incorporado. En este caso la comunicación MIDI entre los diferentes componentes se efectúa internamente, sin cables ni conectores visibles, pero es importante entender qué de las funciones utilizamos en cada momento.

Finalmente, por "Renderizar" un archivo MIDI se entiende que se convertirlo en un archivo WAV, lo que sería parecido a grabarlo en un cassette, por ejemplo, y luego grabarlo de nuevo como un programa de grabación como el Audacity o similares.

### **Las tarjetas de sonido:**

Las señales analógicas de audio que llegan a la tarjeta, pasan primero por un amplificador que les proporciona un nivel adecuado para la digitalización. El amplificador amplifica más las entradas del micrófono que las entradas de línea, por lo que si se tiene una señal de una fuente que sea relativamente débil, se deberá conectar al puerto del micrófono en vez del de línea. Las tarjetas que funcionan en estéreo utilizan dos canales amplificadores. A continuación, el amplificador pasa la señal al convertidor analógico digital, y finalmente se pasa las señales digitalizadas al sistema principal para su almacenamiento o su uso en una aplicación.

Las señales digitales de audio generadas por una aplicación que ejecuta un archivo de forma de onas llegan a la tarjeta de sonido y pasan a través del mezclador. El mezclador combina las salidas de diferentes fuentes y entrega el resultado a otro amplificador que prepara la señal para su salida a los altavoces, auriculares o al sistema de sonido externo. Este proceso es similar a la reproducción de sonido via CD-Audio.

Cuando se utiliza una aplicación para reproducir un archivo midi, la tarjeta recibe la señal igual que antes, pero la información no se envía al mezclador directamente porque el midi contiene instrucciones de programación, no audio digitalizado. En su lugar, la información midi se envía al sintetizador, que genera el audio digital a partir de las instrucciones y pasa las señales de audio digital al mezclador, repitiéndose el proceso de antes.

No hay ningún estándar para tarjetas de audio. Antiguamente, existían pocos fabricantes de tarjetas y la mayoría de diseñadores de juegos tenían pocas dificultades en incluir en sus programas los drivers necesarios para soportarlas a todas. Pero cuando aumentó el número de fabricantes y éstos comenzaron a sacar varios modelos de tarjetas, se hizo cada vez más difícil para los diseñadores, mantener ese ritmo, y la

compatibilidad se convirtió en un asunto crucial. Cuando Windows conquistó el mercado de los PC, la compatibilidad perdió importancia porque sólo era necesario instalar un controlador para su adaptador de audio en Windows, no en cada aplicación individual. Añadiendo a esto el hecho de que la función Plug&Play de Windows puede detectar de forma automática e instalar un driver para la mayoría de las tarjetas de audio del mercado, la compatibilidad ya no es un problema.

### Lista de algunos programas útiles (para Windows) y recursos multimedia

Es evidente que para poder crear un juego mínimamente vistoso, debemos mostrar unos gráficos mínimos: una nave, un asteroide, un fondo de estrellas, un disparo... Para poder incluir estos dibujos en nuestro programa, previamente deberemos crear los diferentes dibujos y sprites (recordemos que un sprite es un dibujo en movimiento por la pantalla), e incluso si es necesario, crear nuestros sonidos para incluirlos también, animaciones 3D, etc. Para todo eso deberemos utilizar unas aplicaciones concretas fuera del entorno Fénix.

Se sale totalmente de los propósitos de este curso profundizar en el uso de las diferentes herramientas que un desarrollador de videojuegos necesitará para crear su mundo particular. Editores de imágenes, editores de vídeo, editores de sonido, suites 3D, etc son aplicaciones esenciales para poder generar un juego creíble, atractivo, interesante y profesional. Este curso sólo pretende iniciar al lector en el conocimiento de un lenguaje de programación concreto como es Fénix, dejando a un lado una explicación pormenorizada de todas estas herramientas que también son imprescindibles. En este sentido, el lector tendrá que hacer un trabajo de búsqueda y estudio particular de aquellas aplicaciones que en cada momento le convenga más o le ayuden mejor en su tarea de dibujar, crear músicas, incluir animaciones, etc.

Las posibilidades que se presentan al desarrollador son casi infinitas. Existen muchas aplicaciones para todo lo necesario, y en la elección de una herramienta particular influirán las necesidades particulares de un juego, el conocimiento multimedia del programador, la posibilidad de adquisición de la aplicación (algunas son libres) o incluso la apetencia personal.

A continuación ofrezco una lista incompleta, ni mucho menos rigurosa ni definitiva, de aquellas aplicaciones de las cuales tengo conocimiento (preferentemente libres), y que considero interesante conocer. Dejo al lector la libertad de elegir la que considere más oportuna para sus propósitos, y repito que será tarea suya el aprendizaje del funcionamiento y posibilidades últimas de la aplicación escogida. Para un listado más exhaustivo, recomiendo visitar <http://www.gpwiki.org/index.php/Tools:Content>

#### Editores de gráficos vectoriales

**Inkscape** (<http://www.inkscape.org>): Inkscape es un excelente editor libre de SVG (EL editor), el formato recomendado por el consorcio empresarial W3C para gráficos vectoriales. Inkscape es una escisión de un proyecto anterior llamado Sodipodi.

**OpenOffice Draw** (<http://www.openoffice.org>): Dentro de la suite ofimática libre OpenOffice, aparte del procesador de textos, la hoja de cálculo, etc nos encontramos con un simple pero muy funcional y eficaz diseñador de gráficos vectoriales. No se puede comparar en funcionalidad a los editores dedicados (OpenOffice Draw no deja de ser una parte más de una suite ofimática), pero para un usuario medio cumple de sobra su cometido. Para lo que haremos en este curso responderá de sobras a lo que podamos pedirle.

**Skencil** (<http://www.skencil.org>): Programa de dibujo vectorial similar a CorelDraw o Illustrator, programado en Python, pero libre..

### **Alternativas no libres:**

**Anime Studio** (<http://www.lostmarble.com>): Altamente recomendable por su sencillez de uso y su gran potencia y versatilidad. Está orientada a la creación de dibujos y animaciones sin interacción; especialmente pensada para creadores de “dibujos animados”. Las herramientas que incorpora son muy flexibles y útiles a la hora de trabajar con eficacia y rapidez. Viene provisto también de una serie de tutoriales perfectamente progresivos, estructurados y coherentes que facilitan en extremo el aprendizaje de esta herramienta. Altamente recomendable, pues. No obstante, no es libre y es de pago.

*Un programa parecido al anterior, es decir, un programa para la animación profesional 2D clásica dirigido a la industria de los dibujos animados, es **Ktoon** (<http://ktoon.toonka.com>). Este programa es libre y gratuito. Igual que Anime Studio, trabaja con diversos formatos y tiene la posibilidad de utilizar papel calco, secuencias, sonidos, además de guardar en varios formatos de video y en formato Flash. No obstante, sólo existe versión para Linux. De todas maneras, contamos con otra alternativa libre y esta vez multiplataforma que tiene los mismos objetivos -animación 2D-, aunque todavía no tan desarrollada: **Synfig** (<http://www.synfig.com>)*

**Flash** (<http://www.adobe.com>): Aplicación de referencia en lo que atañe al mundo multimedia en 2D. Creador de dibujos y animaciones tanto interactivas como no interactivas con inclusión de múltiples posibilidades multimedia, desde sonido y música hasta video. Es el programa original creador del formato SWF, el cual se utiliza en infinidad de webs para aumentar su atractivo visual. Incorpora además extensos tutoriales y documentación de referencia. Soporta además la posibilidad de programación de script mediante el cada vez más potente y flexible lenguaje ActionScript, transformando las últimas versiones de Flash en todo un entorno integrado para el diseño gráfico con la ayuda de la programación. No obstante, debido a sus posibilidades puede ser excesivamente complejo para el iniciado, además de no ser ni libre ni gratis. Algunas webs de soporte y ayuda con ejemplos de código ActionScript o animaciones son <http://www.flash-es.net>, <http://www.flashkit.com> o <http://x-flash.org>. Una aplicación similar, también comercial, es **Swish** (<http://swishzone.com>)

**FreeHand** (<http://www.adobe.com>): Otro excelente editor de gráficos vectoriales de la casa Adobe, más centrado en el mundo editorial y de papel impreso. De esta misma casa también se puede obtener un programa muy similar, **Illustrator**.

**CorelDraw** (<http://www.corel.com>): El equivalente al FreeHand, pero de la casa Corel. Una página de soporte a los usuarios de CorelDraw interesante es <http://www.coreldev.org>

**ArtRage** (<http://ambientdesign.com/artrage.html>) : Programa de dibujo que simula el dibujo sobre un lienzo utilizando pinceles, lápices, tizas, etc. Se puede manejar con el ratón o con una tableta gráfica.

### Editores de mapa de bits

**MSPaint** (de serie en Windows, aunque no libre): Práctico, sencillo, rápido y sin opciones rebuscadas. Elemental e intuitivo. Y lo mejor, viene instalado de serie en Windows. Seguramente, será el editor de mapbits que usaremos más durante el curso, para hacer los bocetos de los dibujos que usaremos en nuestros juegos, sin necesidad de perfilarlos ni añadir detalles: para dibujar triángulos, cuadrados, líneas... De igual manera, su ayuda es concisa y clara: buena idea echarle un vistazo.

**The Gimp** (<http://www.gimp.org>): La competencia libre al Adobe PhotoShop, la cual no le tiene nada que envidiar en prestaciones, potencia y calidad de resultados. Demostración palpable de que se puede desarrollar una aplicación libre que compita de igual a igual con los mejores programas propietarios de su rama. Altamente recomendable para editar las imágenes y fotografías con un resultado excelente. No obstante, al principio al usuario novel puede costarle entender el funcionamiento del programa, no es demasiado intuitivo. Atención, para poder hacerlo funcionar se necesita que previamente se haya instalado la librería gráfica GTK+ en el sistema. Dicha librería y la versión para Windows del Gimp las podrás encontrar si vas directamente a <http://gimp-win.sourceforge.net>. Una página web interesante de soporte al usuario en español, con muchos tutoriales, es <http://www.gimp.org.es>

**MtPaint** (<http://mtpaint.sourceforge.net>): Sencillo programa de dibujo especialmente pensado para el Pixel Art -es decir, creación de dibujos mediante la edición de pixel a pixel- y la edición simple de fotografías.

*Existen además aplicaciones muy interesantes llamadas trazadoras de imágenes, cuya función principal es convertir imágenes que son mapa de bits en imágenes vectoriales, para su manipulación posterior más cómoda. Un ejemplo de estas aplicaciones sería Potrace (<http://potrace.sourceforge.net>).*

#### **Alternativas no libres:**

**PhotoShop** (<http://www.adobe.com>): El gran monstruo de la edición gráfica. El programa más potente, versátil y funcional. Con multitud de plugins que amplían el radio de acción de esta aplicación hasta el infinito. Con él se podrán hacer todos los retoques imaginables a fotografías y dibujos de forma totalmente profesional, además de aplicar filtros y efectos diversos. Posiblemente, tal cantidad de posibilidades pueda abrumar al usuario novel, ya que no es tan fácil de utilizar al principio como lo pueda ser un PaintShopPro o un Office Picture Manager, pero las ventajas que obtenemos con él no tienen comparación. Una web interesante de soporte con ayuda al usuario en español es <http://www.solophotoshop.com>

**Expression Studio** (<http://www.microsoft.com/expression>) : Suite de diseño gráfico de Microsoft formado por varias herramientas, entre las cuales se encuentra un editor de mapa de bits llamado Expression Design.

**Paint Shop Pro** (<http://www.corel.com>): Excelente editor de dibujos y fotografías para el usuario medio. Ofrece las funcionalidades básicas que se le requieren a un programa de este tipo de una forma totalmente solvente, fácil y rápida. Dentro del paquete de instalación incorpora una utilidad muy práctica, **Animation Shop**, que es un generador de animaciones GIF.

**Office Picture Manager** (<http://www.microsoft.com>): Similar en prestaciones al Paint Shop Pro: pensado para el usuario medio que no necesite de grandes prestaciones sino de las más elementales –a la par que más usadas-. Sencillo, rápido y eficaz. Este programa viene integrado como un paquete más a instalar en la suite ofimática Office.

**Fireworks** (<http://www.adobe.com>): Aplicación pensada específicamente para la producción de elementos gráficos que se utilizarán en la web. Es decir, permite hacer dibujos, manipular fotografías y generar gráficos animados para Internet así como optimizar esas imágenes e incorporarles interactividad avanzada.

**Graphics Gale** (<http://www.humanbalance.net>) : Editor de gráficos y animaciones Gif muy fácil de utilizar, enfocado principalmente al PixelArt.

---

No obstante, si no quieres entretenerte creando tus propios dibujos/animaciones y deseas aprovechar las creaciones gráficas que existan ya realizadas por otras personas, en Internet hay multitud de webs de donde puedes descargarte gran cantidad de sprites, imágenes de fondo, iconos, etc gratuitamente para poderlos insertar en tus videojuegos inmediatamente. Puedes mirar por ejemplo <http://www.gsarchives.net> , <http://www.panelmonkey.org> , <http://www.molotov.nu/?page=graphics> , <http://www.grsites.com/> , <http://www.sprites-inc.co.uk/> , <http://www.sprite-town.tk/> , <http://www.cvrpg.com/sprites.php> , <http://www.kingdom-hearts2.com/zerov/sprites.html> , <http://rcz.saphiria.net/html/sprites/sonic.php> , <http://www.nes-snes-sprites.com/> , <http://www.goldensun-syndicate.net/sprites/> , <http://dioxaz.free.fr/objets.htm> , <http://www.drshnaps.com/drshnaps/kirbyorigins/extras/sprites/ksprites.html> , <http://www.geocities.com/spritepage2003/update.html> ,

<http://lostgarden.com/2006/07/more-free-game-graphics.html>  
<http://reinerstileset.4players.de/englisch.htm>  
[http://forum.thegamecreators.com/?m=forum\\_view&t=85024&b=4](http://forum.thegamecreators.com/?m=forum_view&t=85024&b=4)  
<http://www.angelfire.com/sc/Shining/Metal.html>  
[http://www.gamedev.net/community/forums/topic.asp?topic\\_id=272386](http://www.gamedev.net/community/forums/topic.asp?topic_id=272386) ,  
<http://local.wasp.uwa.edu.au/~pbourke/texture/> (texturas)

Otra posibilidad, no siempre legal, es obtenerlos -"ripearlos" - de un videojuego que poseamos.

### Suites de animación 3D

**Blender** (<http://www.blender3d.org>): Blender es una suite integrada de modelación, animación, renderización, post-producción, creación interactiva y reanimación 3D-ideal para juegos-, con características tan interesantes como soporte a la programación bajo el lenguaje Python. Blender tiene una particular interfaz de usuario, que es implementada enteramente en OpenGL y diseñada pensando en la velocidad. Hoy en día todavía no exprime todas las posibilidades que los grandes productos ofrecen, pero éste es un proyecto que está gozando de una rápida evolución debido a la gran participación de la comunidad involucrada, así que es de esperar mejoras importantes en plazos cortos de tiempo.

**Art of Illusion** (<http://www.artofillusion.org>): Art of Illusion es un estudio de modelado y renderizado 3D. Es un programa Java, por lo que se necesita tener instalado Java Runtime Environment 1.4 (o posterior).

**AutoQ3D** (<http://autoq3d.sourceforge.net>): Programa de modelado 3D potente, flexible y fácil de utilizar

**Zmodeler** (<http://www.zmodeler2.com>): Otro programa de modelado 3D libre y gratuito.

**Gmax**: (<http://www.turbosquid.com/gmax>): Software libre y gratuito de modelaje y animación 3D especialmente pensado para videojuegos.

**Anim8or** (<http://www.anim8or.com>) : Otro software libre y gratuito de modelaje y animación 3D.

**MilkShape3D** (<http://chumbalum.swissquake.ch/ms3d/index.html>) : Animador de esqueletos 3D.

También nos podemos encontrar programas más específicos que no abarcan todo el proceso entero sino que se dedican a modelar determinado tipo de imágenes. Por ejemplo, podemos utilizar un modelador libre de polígonos con mallas –sin animación- llamado **Wings 3D** (<http://wings.sourceforge.net>), o **Pov-Ray** (<http://www.povray.org>) , herramienta para crear imágenes tridimensionales fotorealistas mediante la técnica del trazado de rayos, o motores 3D listos para incorporar a proyectos de desarrollo, como **Irrlicht** (<http://irrlicht.sourceforge.net>) , que es libre y multiplataforma, igual que **Ogre** (<http://www.ogre3d.org>) o **NeoEngine** (<http://www.neoengine.org>) . También nos podemos encontrar con generadores de explosiones, partículas y efectos especiales 3D, como **ExGen** (<http://exgen.thegamecreators.com/>) -de pago-, o **Particle Illusion** (<http://www.wondertouch.com>) -de pago también-.

Algunas webs de soporte y ayuda referentes al tema del diseño gráfico 3D en general (sin asociarse a ningún producto en concreto pueden ser: <http://www.3drender.com/ref>, <http://www.highend3d.com>, <http://www.beyond3d.com> , <http://humus.ca> o <http://www.es.3dup.com> -ésta última en español- entre otros.

### *Alternativas no libres:*

**Maya** (<http://www.autodesk.com/alias>): Es quizá el software más popular en la industria. Es utilizado por muchos de los estudios de efectos visuales más importantes en combinación con RenderMan, el motor de

rénder fotorrealista de la empresa Pixar. Webs que pueden ayudar al usuario de esta aplicación pueden ser <http://www.simplymaya.com> y <http://www.mayatraining.com>

**3Dstudio Max** (<http://www.discreet.com>; <http://www.autodesk.com>): . Es el líder en el desarrollo de 3D en la industria de juegos y usuarios hogareños.

**Lightwave 3D**(<http://www.newtek.com>) : Otro “monstruo” del sector. Como los anteriores, se compone de dos partes: modelador y renderizador.

**TrueSpace** (<http://www.caligari.com>) : Aplicación 3D integrada, con una apariencia visual muy intuitiva. Una característica distintiva de esta aplicación es que todas las fases de creación de gráficos 3D son realizadas dentro de un único programa, no como los demás que se componen de módulos separados. No es tan avanzado como los paquetes líderes Maya,3Dstudio,Ligtwave),pero provee características como simulación de fenómenos físicos (viento, gravedad, colisiones entre cuerpos) muy interesantes.

**Cinema 4D** (<http://www.maxon.net>): Es un rápido motor de rénder.

**RealSoft 3D** (<http://www.realssoft.fi>): Modelador 3D para Linux y Windows. Incluye rénder también.

**SoftImage XSI** (<http://www.softimage.com/xsi>): Excelente software de modelado, renderizado y animación 3D enfocado a videojuegos y cine.

**DeleD** (<http://www.delgine.com>) : Modelador 3D enfocado en el diseño para videojuegos 3D.Tiene una versión Lite gratuita.

#### Editores de sonido

**Audacity** (<http://audacity.sourceforge.net>) : Editor de audio que permite reproducir, grabar en múltiples pistas, editar, mezclar, aplicar efectos, etc a archivos de formato de onda, de una manera cómoda y sencilla.Ideal para usuarios noveles o no tanto.Permite la utilización tanto de Wav como de Ogg, y Mp3 si se le incorpora el code LameMp3.

**WaveSurfer** (<http://www.speech.kth.se/wavesurfer>): Herramienta para la visualización y manipulación de sonido en formato de onda, cuyas capacidades pueden ampliarse mediante plug-ins.

**Sweep** (<http://www.metadecks.org/software/sweep>): Otro editor de sonidos en formato de onda (Wav y Ogg)

**ReZound** (<http://rezound.sourceforge.net>): Y otro. Es parecido al CoolEdit. Permite aplicar filtros,efectos, loops, cue points,etc

**Mixere** (<http://mixere.sourceforge.net>): Este programa libre no es un editor de sonido propiamente dicho: sólomente es un mezclador de ficheros de sonido. Muy sencillo pero perfecto para quien desee solamente esta funcionalidad a partir de ficheros ya editados Wav,Ogg,Mp3... para obtener curiosos efectos.

#### *Alternativas no libres:*

**Audition** (<http://www.adobe.com>) : Editor anteriormente llamado Cool Edit Pro. Ideal para manipular archivos de onda: incorpora múltiples efectos, filtros y posibilidades de edición de cualquier sonido Wav. Permite cambiar el formato del archivo y grabarlo en disco como Ogg.La herramienta completa para los usuarios medios.

**Nero Wave Editor** (<http://www.nero.com>): Aplicación que viene integrada dentro de la suite de grabación Nero, la cual reúne todo lo necesario para grabar y editar archivos de formato de onda de manera clara y directa. Las opciones que incorpora son las básicas, pero trabaja de forma muy optimizada. No obstante, por defecto no puede abrir ni grabar ficheros en formato Ogg, solamente Wav y otros (mp3,etc).

**Goldwave** (<http://www.goldwave.com>): Otra excelente herramienta

A otro nivel muy diferente, ya completamente profesional, podríamos hablar de otro tipo de herramientas, no ya simples editores de sonido sino completas suites de composición, grabación y manipulación completa de música tanto MIDI como de formato de onda, como sería el **Protools** (<http://www.digidesign.com>), el **CuBase VST** (<http://www.steinberg.net>), el **SoundForge** (<http://www.sonicfoundry.com>) o el **CakeWalk** (<http://www.cakewalk.com>), las cuales son usadas en los estudios de grabación más importantes. Las alternativas libres a este tipo de programas todavía no están lo suficientemente evolucionadas para competir en condiciones de igualdad con las anteriores nombradas. Actualmente las más relevantes podrían ser los secuenciadores: **Rosegarden** (<http://www.rosegardenmusic.org>), el cual incorpora secuenciador MIDI y editor de partituras o, en menor medida **Ardour** (<http://ardour.org>). Ambos funcionan sólo en Linux.

También podríamos hablar en este apartado de otro tipo de programas, que no son propiamente editores de audio, sino que lo que permiten es extraer -"ripear"- sonidos y músicas de CDs de audio y convertirlos en ficheros de formato de onda para, por ejemplo, editarlos y poderlos utilizar (siempre respetando la legalidad sobre los derechos de autor) en tus propios juegos. Por ejemplo, tenemos como aplicaciones libres:

**AudioGrabber** (<http://www.audiograbber.com-us.net>): Extrae canciones de los CD y los copia en formato de archivo Wav al disco duro en formato Wav y Ogg, entre otros.

**Cdex** (<http://cdexos.sourceforge.net>): Extrae canciones de los CD y los copia en formato de archivo Wav y Ogg al disco duro, entre otros.

**Exact Audio Copy** (<http://exactaudiocopy.de>): Extrae canciones de los CD y los copia en formato de archivo Wav y Ogg al disco duro, entre otros.

**BeSweet** (<http://dspguru.doom9.net/>): Este programa libre no extrae canciones de CD: simplemente transforma el formato de los archivos de sonido seleccionados en otro. Soporta los formatos MP3, AC3, WAV, MP2, AVI, Aiff, VOB y Ogg.

Respecto a los MIDIS, no podemos hablar propiamente de editores de sonido, sino de programas creadores y editores de MIDIS. Algunos podrían ser:

**Band in a Box** (<http://www.pgmusic.com>): Atención: este programa NO es libre. Es un generador de canciones. Aplicación ideal para componer melodías con multitud de arreglos armónicos y rítmicos. No obstante, se ha de saber un mínimo de solfeo y armonía musical para poderlo utilizar.

**Visual Music** (<http://shitalshah.com/vmusic>): Permite tocar hasta 128 instrumentos musicales y grabar el resultado. También puedes escribir scripts para crear tus propias composiciones. Guarda el resultado en ficheros MIDI.

**Jazz++** (<http://jazzplusplus.sourceforge.net>): Secuenciador de audio que permite grabar, editar y reproducir archivos MIDI.

**Hydrogen** (<http://hydrogen-music.org>): Sintetizador de batería por software que se puede usar por sí solo, emulando una caja de ritmo basada en patrones, o vía un teclado/secuenciador MIDI externo por software. Otro software libre similar es **HammerHead** (<http://www.threechords.com/hammerhead>).

Respecto la existencia de conversores de ficheros MIDI a Wav y viceversa se ha de decir que es escasa, debido a la dificultad que entraña la diferencia de estructura que poseen ambos formatos: uno está formado por instrucciones y otro por sonido digitalizado. Así que la transformación de uno a otro es muy costosa y no siempre con resultados óptimos. Puedes mirar en <http://www.midconverter.com> o en <http://www.mp3-converter.biz> a ver si encuentras la aplicación que más se ajuste a tus necesidades.

Por otro lado, también puedes recurrir a sitios web donde te puedes descargar sonidos digitalizados de los más variados ámbitos: explosiones, truenos, ladridos, sonidos de oficina, etc. Estas web son almacenes de recursos muy interesantes para explorar. Ejemplos son: <http://recursos.cnice.mec.es/bancoimagenes/sonidos>, <http://www.flashkit.com/soundfx>, <http://www.hispasonic.com>, <http://3dcafe.com/asp/sounds.asp>, <http://www.findsounds.com>, <http://www.therecordist.com/pages/downloads.html>, <http://sounds.wavcentral.com>, y, si queremos midis: <http://www.midisite.co.uk>. Si queremos archivos en formato .mod, podemos ir a <http://modarchive.org>, <http://www.modplug.com> (donde podemos encontrar también software de creación de ficheros mod), o <http://www.mirsoft.info>.

Otro recurso a tener en cuenta es la posibilidad de utilizar música libre (al igual que lo es el software), descargable de Internet de forma totalmente legal y gratuita, fomentando además la promoción de grupos y artistas que apoyan este tipo de iniciativas. Puedes echarle un ojo a <http://www.musicalibre.info>, <http://www.musicalibre.es>, <http://www.jamendo.com>, y si quieres más, hay un listado completo en [http://parolas.thebbs.org/sincanon/Musica\\_libre.html](http://parolas.thebbs.org/sincanon/Musica_libre.html)

#### Editores de video

**Movie Maker** (de serie en Windows, pero no libre): Lo que es el MsPaint en el mundo de los editores gráficos, es el Movie Maker en el mundo de los editores de video. Elemental, sencillo, con pocas prestaciones pero las más usuales y eficaz. Para todos aquellos que no deseen elaborar videos demasiado complejos y que se contenten con un montaje sencillo y efectivo. Ideal para comenzar. Viene instalado de serie en Windows.

**VirtualDub** (<http://www.virtualdub.org>): La gran alternativa libre en el mundo de la edición de video se llama VirtualDub. Está a la altura de cualquier editor comercial importante como Premiere o Studio. Ofrece las mismas funcionalidades y su eficiencia y calidad son comparables. Otro ejemplo de programa libre que no tiene nada que envidiar a los "monstruos" de la rama. Comentar que existe una variante del programa -lo que se llama un "fork"- llamada **VirtualDubMod**, en teoría más completa y funcional pero que hace tiempo que no se actualiza; se puede encontrar en <http://virtualdubmod.sourceforge.net>

**Cinelerra** (<http://heroinewarrior.com/cinelerra.php3>): Otro estupendo editor de video libre.

**Wax** (<http://www.debugmode.com/wax>): Este programa no es propiamente un editor de video sino un programa de composición de efectos especiales de video 2D y 3D. Puede utilizarse como programa independiente o como plugin de Premiere. Otro programa libre y multiplataforma de edición de video y tratamiento de efectos especiales, con soporte para OpenGL y OpenML (<http://www.khronos.org/openml>), es **Jahshaka** (<http://www.jahshaka.org>).

**Cinepaint** (<http://www.cinepaint.org>): Más que un editor de video al uso, este programa es más bien un

editor de fotogramas individuales de una película, siendo pues más similar a un editor de bitmaps realmente, como Gimp.

**LimSee2.** (<http://limsee2.gforge.inria.fr/>): Herramienta libre de autoría basado en el lenguaje SMIL (estándar abierto internacional - <http://www.w3.org/AudioVideo/> - ), el cual permite integrar audio, video, imágenes, texto o cualquier otro contenido multimedia dentro de una presentación multimedia. Los navegadores web estándar son capaces de reproducir contenido SMIL, pero un reproductor dedicado a utilizar puede ser **Ambulant** (<http://ambulant.sourceforge.net>)

*Existen unos cuantos editores de video libres más a tener en cuenta, aunque la mayoría sólo están disponibles para GNU/Linux. Un ejemplo ideal para el usuario doméstico sería **Kino** (<http://www.kinodv.org>), y una aplicación más enfocada al profesional sería **LiVES** (<http://lives.sourceforge.net>), entre otros..*

#### **Alternativas no libres:**

**Studio** (<http://www.pinnaclesys.com>): Aplicación ideal para la edición de video caseros. Permite todo tipo de prestaciones para el montaje y tratamiento de las capturas y grabaciones digitales.

**Premiere** (<http://www.adobe.com>): Competencia directa del Pinnacle Studio, aunque tal vez orientado un poco más hacia el mercado profesional: ofrece más funcionalidad que el Studio y es más utilizado para películas de larga duración y con un montaje más elaborado.

Si todavía no tienes suficiente con la lista anterior (ni con la de Gdwiki), todavía podrás encontrar una cantidad brutal de enlaces a muchísimas más herramientas de diseño y multimedia en el espectacular post del foro de GameDevelopers: <http://portalxuri.dyndns.org/gamedevelopers/modules.php?name=Forums&file=viewtopic&t=3141&highlight=mont%F3n++informaci%F3n+%FAtil>

Finalmente, te comento que si quieres tener una lista bastante completa de programas libres, no ya solamente de los ámbitos tratados aquí sino de cualquier ámbito (procesadores de texto, hojas de cálculo, juegos, etc) que funcionen en Windows –y en Linux también-, échale un vistazo a la estupenda web <http://www.cdlibre.org>, y también a la no menos fantástica <http://alts.homelinux.net>.

Es más, si lo que quieres es tener un entorno donde ya tengas preinstalados la mayoría de programas libres para multimedia que he mencionado anteriormente (Inkscape, Skencil, Gimp, Blender, Kino, Cinelerra, Audacity, Rosegarden, Ardour, Hydrogen,...) sin preocuparte de descargarlos, de manera que estés listo para empezar a trabajar inmediatamente, puedes encontrar bastantes distribuciones de Linux dedicadas al mundo gráfico/video/3D/sonido con todo preparado y configurado. Incluso en formato Live-CD, para que no tengas que cambiar nada de tu ordenador. Échale un ojo por ejemplo a UbuntuStudio (<http://www.ubuntustudio.org>)

También existe una distribución especializada solamente en software musical digna de mención: Musix (<http://www.musix.org.ar>)

De todas maneras, no está de más comentar (algo obvio, por otra parte) que es que en la creación y desarrollo de un videojuego medianamente complejo las tareas se han de dividir y repartir entre diversos trabajadores, ya que si no es imposible tirar adelante un proyecto profesional. Es decir, la plantilla estará formada por guionistas, los cuales habrán pensado, escrito y corregido toda la historia y sus personajes mediante técnicas al efecto -como storyboards-; estará formada también por personal diseñador gráfico, que se dedicará a crear la ambientación (personajes, paisajes, etc) con las herramientas que se acaban de enumerar anteriormente; también puede haber parte de los trabajadores dedicados a la creación musical; y un pequeño grupo dentro de la plantilla será el de programadores propiamente dichos, que exclusivamente se dedicarán a programar. Pueden haber más departamentos, y todos ellos serán coordinados finalmente por

una comisión jefe de proyecto. Es decir, que un programador profesional no tiene por qué conocer las herramientas de diseño y viceversa.

No obstante, como nosotros desarrollaremos videojuegos amateur, donde todo lo tendremos que hacer nosotros, será necesario tocar muchas teclas y tener mínimas nociones de todo un poco: redacción de guiones, diseño gráfico, música, programación, planificación de proyectos, etc. Una ardua y completa labor, pero a la vez muy gratificante.

## **CAPÍTULO 1: PRIMER CONTACTO CON FÉNIX**

### **¿Qué es Fénix? Propósito y características :**

Fénix es un sencillo lenguaje de programación compilado/interpretado diseñado para crear y ejecutar específicamente cualquier tipo de juego 2D. El lenguaje está claramente orientado al manejo de gráficos 2D; es decir, que el propio lenguaje se preocupa de dibujar los gráficos en pantalla, ahorrándole al programador el trabajo que eso supone, que es mucho. Fénix incluye un motor de renderizado 2D por software que convierte la programación de juegos en una tarea fácil pero potente. En general, cualquier juego que no emplee objetos 3D real es posible realizarlo con Fénix. Sus características principales son:

- Dibujo rápido de sprites con rotado, escalado, animación y grados de transparencia.
- Detección de colisiones a nivel de pixel
- Procesos (programación multihilo)
- Rutinas de scroll tipo "parallax"
- Múltiples regiones en pantalla con o sin scroll
- Entrada por teclado, ratón y joystick
- Modos de 256 y 65536 colores (8 y 16 bits).
- Soporte del formato gráfico PNG, entre otros.
- Soporte (incompleto) para Modo 7
- Reproducción de sonidos en formato WAV, PCM y Ogg Vorbis
- Librería de sonido basada en MikMod
- Reproducción de módulos de música en formato IT, MOD y XM
- Soporte de librerías externas (en Windows, DLL).
- Multiplataforma: funciona en Windows 95/98/Me/2000/XP, GNU/Linux sobre chips Intel, MacOSX, BeOS, GP32 y Dreamcast. Además se prevee que pueda ser usado en Dreamcast, PSP,GP32 y GPX2 en futuro cercano. Esta característica es posible gracias a que Fénix está basado a más bajo nivel en la librería gráfica SDL, la cual es portable a todos estos sistemas operativos mencionados.

Incluso programadores experimentados encontrarán en Fénix una herramienta útil. El lenguaje en sí soporta muchas de las características comunmente encontradas en los lenguajes procedurales, tales como múltiples tipos de dato (INT, WORD, BYTE, STRING, FLOAT), punteros, tablas multidimensionales, estructuras, y las sentencias habituales de control de flujo.

Los autores de Fénix creen en la filosofía de la libertad de uso del software y en la distribución del código fuente. El lenguaje Fénix, su compilador y su intérprete se distribuyen gratuitamente bajo los términos de la GNU General Public License, con lo que las fuentes están disponibles y eres libre de extenderlas o hacer tu propia versión, tal como comenté en el primer capítulo. La distribución de Fénix bajo la licencia GNU GPL implica que, si realizas cambios en Fénix, estás obligado a distribuir el código fuente que hayas cambiado junto con tu versión, sin ningún coste, y bajo la misma licencia, permitiendo su redistribución. Los juegos realizados con Fénix están libres de cualquier limitación, así como cualquier

DLL realizada por tí que enlace con dichos juegos, salvo las derivadas de terceros (dlls con las que linke tu librería por ejemplo).

Hay que decir que Fénix es un proyecto en desarrollo: NO ESTÁ TERMINADO. Actualmente está en un período "alfa": todavía faltan características por implementar, y el código sufre cambios relativamente importantes entre versiones. Se espera que existan todavía numerosos bugs.

### **Historia de Fénix:**

En la década de los 90 el entonces estudiante Daniel Navarro Medrano creó como proyecto final de carrera una herramienta orientada a la creación de videojuegos de 32 bits bajo MS-DOS. El nuevo lenguaje, de nombre DIV Games Studio, combinaba características de C y Pascal con un entorno completo que permitía la creación y edición de todos los aspectos de los proyectos: programación, edición gráfica y sonora y un largo etc.

Fénix, inicialmente bajo el nombre DIVC y de naturaleza GNU y gratuita, apareció de la mano de Jose Luis Cebrián como una herramienta capaz de compilar y ejecutar esos juegos en Linux. El nombre fue cambiado en la versión 0.6 del compilador, que además introducía otras mejoras, como la aparición de un fichero intermedio entre el entorno de compilación y el entorno de ejecución. Ya no era necesario distribuir el código fuente de un juego para poder jugar a los juegos. La ventaja principal de esa práctica (similar en concepto a Java) era clara, compilar en una plataforma y ejecutar en muchas. En la versión 0.71 el proyecto quedó parado, lo que dio lugar a múltiples versiones derivadas que corregían fallos o añadían nuevas características.

La versión oficial de Fénix fue retomada por Slàinte en el año 2002, viejo conocido de la comunidad DIV por ser el webmaster de una de las páginas web más importantes para la comunidad, quien continuó el proyecto bajo el nombre de Fénix - Proyecto 1.0 al que pronto se reincorporaría su creador y cuyo primer objetivo era limpiar el compilador de errores y estabilizarlo. Desde entonces el compilador ha sufrido numerosos cambios y mejoras, dejando de un lado la compatibilidad con el lenguaje DIV, el desarrollo del cual quedó paralizado hace tiempo en su versión 2 desde la quiebra de la empresa que lo comercializaba, Hammer Technologies. (De hecho, DIV2 sólo es compatible con Ms-Dos y Windows 95/98, pero no con Windows 2000 o XP).

Tras un periodo de relativa inactividad en donde Fénix se había estancado en su versión 0.84/0.84a, a mediados del 2006, Juan alias "SplinterGU" retomó con fuerza el desarrollo de nuevas versiones del compilador/intérprete, incorporándole muchas mejoras en rendimiento y velocidad, añadiendo funciones nuevas y corrigiendo bugs crónicos, hasta llegar a la versión actual, la 0.92a. No obstante, debido a ciertas desavenencias con determinados miembros de la comunidad hicieron que Juan se decidiera por abandonar el desarrollo oficial de Fénix y crear un fork (una variante independiente) a partir del código de Fénix llamado Bennu, el cual a día de hoy (año 2007) todavía no ha salido a la luz.

Actualmente, el foro de Fénix (<http://forum.divsite.net>) es un lugar muy activo lleno de gente que intercambia información, trucos y experiencias sobre este lenguaje.

### **Críticas y carencias de Fénix v.092a:**

Fénix ha sido y/o es criticado por varios motivos.

1) Falta de documentación actualizada. De todas maneras, se están poniendo en marcha proyectos como el Wiki de Fénix (<http://www.fenixwiki.se32.com>) o su versión inglesa (<http://www.fenixdocs.com>), donde se pretende recoger la descripción de todas las funciones, variables, características sintácticas y entresijos del lenguaje. Cuando esté acabada, sería útil como manual de referencia, pero no enseñaría a

programar ni contendría tutoriales. Espero que con este curso se solucione en parte este problema ;)

2) Carencia de una buena base multilingüe, lo cual provoca en los usuarios no hispanoparlantes un cierto recelo en su uso. Esta dificultad está siendo trabajada en las últimas versiones CVS, aún por completar.

3) Inexistencia de IDEs completos para plataformas no-Windows, lo que conlleva cierta dificultad en su uso para el resto de plataformas.

(Por si no lo sabéis, un IDE no es más que un programa que integra en un solo entorno el editor, el intérprete, el compilador, el depurador si lo hubiera, la ayuda del lenguaje, etc; de manera que todas las herramientas necesarias para el programador están accesibles inmediatamente dentro de ese entorno de forma coherente, cómoda y sobretodo, visual. Si no se trabaja dentro de un IDE, las diferentes herramientas –que en el caso de Fénix son de consola (es decir, ejecutables vía línea de comandos –“la ventana negra del MS-DOS”- y por tanto no visuales)- son para el desarrollador programas independientes, inconexos entre sí, con lo que la utilización conjunta de uno u otro es más pesada e incómoda, y no es visual).

La explicación de que no haya un IDE oficial que acompañe a las versiones oficiales del lenguaje es porque el compilador/intérprete Fénix son programas multiplataforma, y su entorno oficial debe obligatoriamente funcionar en cualquier plataforma donde funcione Fénix. El mejor modo de conseguir esto sería programar dicho entorno en el propio Fénix, seguramente empleando alguna DLL a medida con funciones comunes, pero todavía no hay ningún resultado en este aspecto.

Aparte, hay algunas carencias de Fénix que a primera vista podrían sorprender al recién llegado:

**Inexistencia de soporte 3D:** Existen DLLs no oficiales que permiten parcialmente trabajar en 3D (como la VTE.dll o la M8ee.dll) , pero Fénix ha sido diseñado para realizar juegos 2D. Un motor 3D es un cambio fundamental que implica cambiar prácticamente todo el intérprete.

Además, no existe ningún motor 3D que sea óptimo para cualquier tipo de juego (por ejemplo un Quake es un programa muy distinto a un juego de estrategia tipo Starcraft).

Por otra parte hacer un juego 3D requiere multitud de medios y conocimientos. Ningún lenguaje va a ayudar al programador a la hora de controlar la cantidad de polígonos en pantalla o escribir la física de tu juego. Si ya tienes los conocimientos necesarios, no necesitas Fénix.

Aunque Fénix siga siendo un lenguaje en 2D, es posible emplear las características de las aceleradoras gráficas para optimizar el rendimiento de los juegos. Sin embargo la librería gráfica de Fénix está escrita completamente por software para permitir la máxima compatibilidad en todo tipo de ordenadores y plataformas, independientemente de la calidad de la tarjeta gráfica, memoria o procesador. No obstante, se está estimando portar la librería gráfica a OpenGL –acelerada por hardware-.

**Inexistencia de soporte de modos gráficos de 24/32 bits:** La calidad visual de los modos de 16 bits es indistinguible de los modos de 24/32 bits en un juego. Sin embargo, los modos de 32 bits tienen ciertas ventajas:

- Precisión completa de 0 a 255 para las componentes de los colores, lo que permitiría usar mapas de durezas de forma preciso (ya veremos lo que es eso de mapa de durezas).
- Misma codificación de color para todas las tarjetas, lo que significa que un número dado representa el mismo color en todas partes.
- Más facilidad para escribir rutinas que hagan operaciones de color (transparencias pixel a pixel, transformaciones, etc)

Estas características los hacen atractivos, pero un modo de 32 bits es como mínimo cuatro veces más lento que uno de 256 colores, lo cual lo hace poco útil para el desarrollo de juegos a no ser que se utilice aceleración por hardware. Por lo tanto, la aparición de estos modos no se hará realidad al menos hasta que se introduzca ésta (ver punto anterior).

**Inexistencia de soporte para el formato MP3:** Nunca se soportará el formato MP3, ya que requiere costosas licencias. Pero puedes usar el formato OGG, gratuito, que funciona de forma similar pero ofrece incluso más calidad que el MP3 y es libre.

**Inexistencia (aparente) de rutinas de red, para juegos on-line:** Lo que existe es una DLL oficial (la Fsock.dll). Sin embargo, este tipo de DLL requerirá que el programador haga la mayor parte del trabajo enviando y recibiendo paquetes de red, y controlando por su cuenta paquetes perdidos o recibidos fuera de secuencia y demás problemas habituales. Programar un juego en red es un arte complejo.

### **Obtención, instalación y funcionamiento de Fénix:**

Para descargarte Fénix, tiene que ir a la siguiente dirección: <http://sourceforge.net/projects/fenix>. (Sourceforge es un portal web que aloja multitud de proyectos de software libre, no solamente Fénix). Bueno, una vez en esta página, lo único que tenemos que hacer es clicar sobre el botón "Download Fenix". Verás que vas a parar a otra página desde la cual te podrías descargar tres cosas que vienen indicadas bajo la columna llamada "Package": un paquete de códigos y ejemplos, un conjunto de librerías DLL oficiales (que aportan modularidad y funcionalidad extra al núcleo del lenguaje Fénix, y que ya están listas para incluir en nuestros videojuegos y aprovechar así sus comandos definidos: les dedicaremos a ellas un capítulo entero), y el propio Fénix, identificado por el paquete "Fénix", la release del cual -release quiere decir versión en inglés- es en estos momentos la 0.92a.

Si clicas sobre el botón que pone "Download" correspondiente al nombre del paquete que deseamos descargar (en este caso, el package "Fenix"), verás que vuelves a ir a otra página. En esta ocasión, aparecerá una lista de las distintas versiones de Fénix disponibles para descargar. Evidentemente, nos interesará la última. Dentro del apartado para la versión 0.92a vemos que podemos descargarnos dos archivos: "fenix092a-src-release.tgz" y "fenix092a-win32-binary.zip". El primero es el código fuente del propio Fénix, y sólo nos interesaría si deseáramos observar su funcionamiento interno o bien mejorar alguna de sus características, aspecto éste muy avanzado y que requiere altos conocimientos de lenguaje C y de la librería SDL. El archivo que nos interesa es el otro, el cual es nuestro Fénix listo para usar en Windows.

Desde la sección de "Descargas" de esta web te podrás descargar diferentes versiones oficiales del intérprete y del compilador de Fénix para cada plataforma disponible (Windows, Linux, BeOS, etc). Estos programas están todos incluidos dentro de un único zip: es todo lo único realmente necesario para poder empezar a programar y ejecutar los juegos que escribas. Fíjate que te encontrarás que para cada plataforma puedes elegir entre obtener bien la versión de Fénix ya compilada, lista para usar, o bien el código fuente, para que tú mismo puedas hacer las modificaciones que creas oportunas y lo compiles optimizado para tu ordenador, (con cualquier compilador de C). Clicamos sobre el nombre de este archivo, y tras esperar unos instantes, se iniciará su descarga. El fichero ocupa menos de 1Mb.

Una vez que lo descargues, basta con que descomprimas el archivo en una carpeta allí donde quieras instalar Fénix y ya está. Verás que dentro de la carpeta hay unos cuantos archivos. Echémosles un ojo a los más importantes:

**FXC.EXE:** Compilador de Fénix

**FXI.EXE:** Intérprete de Fénix para Windows

Ambos archivos son programas que se ejecutan en la ventana de comandos, antigua ventana de MSDOS: ejecutándolos con doble clic no hará nada, ya que necesitan unos parámetros determinados para que hagan su función, como luego veremos.

Vamos a explicar un poco la razón de ser de estos dos archivos. Ya dije en el primer capítulo que Fénix se compilaba y se interpretaba. Puede que te sorprendiera en su momento, pero ahora verás que no

tiene ningún secreto. El proceso a la hora de escribir y ejecutar un programa en Fénix es el siguiente:

1º) Nosotros escribimos el código Fénix de nuestro juego con cualquier editor de texto (el **Bloc de Notas** mismo de Windows vale), y guardamos ese código fuente en un archivo que ha de tener la extensión **.PRG**.

2º) Seguidamente, compilamos con **FXC.EXE** este archivo **.PRG**. Obtendremos como resultado un nuevo archivo, en formato binario, cuya extensión se generará automáticamente y que es **.DCB**

3º) Una vez obtenido este archivo **.DCB**, cada vez que queramos ejecutar nuestro juego tendremos que interpretar dicho archivo mediante **FXI.EXE**, y así poner en marcha nuestro programa. ¡Y ya está!, esto es todo lo que necesitas para programar en Fénix.

Resumiendo: primero compilamos con **FXC.EXE** los archivos con extensión **PRG** (donde está escrito el código fuente) y luego, si queremos poner en marcha el juego ejecutamos con **FXI.EXE** el archivo con extensión **DCB** obtenido de la compilación previa.

Es decir, el compilador de Fénix deja el programa en un pseudo-código (¡que no es código máquina!) y luego el intérprete de Fénix (el exclusivo para esa plataforma concreta) ejecuta el programa. Por eso Fénix es multiplataforma: existe un intérprete para cada máquina diferente.

Uno se podría preguntar, si lo que se pretende es ser multiplataforma, por qué es necesario el compilador previo: se podría interpretar directamente los códigos fuentes y ya está. Bueno, hay varias razones para seguir la estructura existente. La primera es que al haber compilado el código previamente, la entrada que recibe el intérprete está mucho más optimizada y le facilita mucho su trabajo, con lo que esto redundará en la velocidad e integridad de la ejecución de los juegos. Y otra razón es que gracias al compilador, si no te interesa enseñar el código fuente por ahí –eso no es libre (!?)–, puedes repartir tu juego simplemente ofreciendo el archivo **.DCB**, sabiendo que seguirá siendo igual de portable.

A parte de **FXC.EXE** y **FXI.EXE** verás que hay más archivos en el zip descargado. Hay varios archivos **DLL**: **SDL.DLL** y **SDL\_MIXER.DLL**, que precisamente es la librería **SDL**, librería gráfica sobre la que se sustenta Fénix; (de hecho, Fénix se puede interpretar como un recubrimiento, una capa superior de esta librería, que encapsula, que oculta la dificultad intrínseca que tiene a la hora de programar con ella en C); **LIBPNG-3.DLL**, necesaria para la manipulación por parte de Fénix de imágenes en formato **PNG**; **OGG.DLL**, **VORBIS.DLL** y **VORBISFILE.DLL**, necesarias para la manipulación de sonidos en formato **OGG**; y **SMPEG.DLL**, necesaria para la manipulación de videos en formato **MPG**.

También hay (a parte de algún que otro archivo más prescindible) dos ejecutables más, a parte del compilador y el intérprete, que son **MAP.EXE** y **FPG.EXE**. El primero sirve para manipular imágenes en un formato propio de Fénix, (el **MAP**), pero no comentaremos nada más de este ejecutable en este manual porque no lo utilizaremos: nosotros utilizaremos siempre imágenes en formato **PNG**. El segundo ejecutable sí que lo utilizaremos alguna vez, y por eso su funcionamiento está explicado más adelante, cuando se introduzca el concepto de contenedor **FPG**.

### **La versión 0.93preview9 de Fénix:**

Existe una versión más moderna, con menos errores y más optimizada que la versión oficial 0.92a. Es la llamada versión 0.93preview9. Esta versión la podrás conseguir en dos sitios: o bien te la podrás descargar desde la página principal de <http://www.fenixdocs.com> , o bien directamente de esta otra

dirección: [http://fenixworld.se32.com/e107\\_plugins//depot/files/fw247.0.93preview9.zip](http://fenixworld.se32.com/e107_plugins//depot/files/fw247.0.93preview9.zip) En ambos casos, lo que al final tendrás descargados tan sólo son dos archivos: FXC.EXE y FXI.EXE, nada más.

Lo que tendrás que hacer con estos dos ejecutables es sustituirlos por los existentes en la versión 0.92a, y ya está. No te engañes: el resto de archivos (Dlls y demás) que vienen con el paquete Fénix estándar y que son necesarios en la 0.92a, también son imprescindibles para la 0.93, pero como en ambas versiones de Fénix todos estos archivos extra son exactamente iguales, podemos aprovechar los que ya tenemos de la 0.92a. Es decir, que para poder utilizar la 0.93preview9 deberemos de haber descargado previamente la 0.92a, porque sólo ésta bien completa con todos los archivos necesarios para funcionar. La 0.93preview9 no deja de ser un parche (como su nombre indica, “preview”) de tan sólo dos archivos ejecutables (el compilador y el intérprete), y ya está.

Es muy recomendable que “actualices” tu Fénix a esta versión 0.93preview9, porque se solucionen unos cuantos errores relativamente graves. No obstante, si te interesara utilizar esta versión en Linux, no está disponible. Ni tampoco su código fuente.

### **Tu primer programa en Fénix:**

(extraído del tutorial de EvilDragon y Josebita, en <http://gp32x.de/Fénix>)

Bueno, se acabó la teoría. Vamos a crear nuestro primer programa en Fénix de una vez. Abre el Bloc de Notas o cualquier editor de texto que te guste y escribe el siguiente código (con las tabulaciones tal como están):

```
Program MiPrimerPrograma;

Private
    int mivar1;
End

Begin
    mivar1=10;
    while(mivar1<320)
        delete_text(0);
        mivar1=mivar1+2;
        write(0,mivar1,100,1,"¡Hola mundo!");
        frame;
    end
end
```

Un detalle importante antes de continuar es saber que Fénix es lo que se llama un lenguaje “case-insensitive”; es decir, que es igual que escribas todas las palabras del código en mayúsculas o minúsculas, o alternándolas si quieres (?): es lo mismo escribir “while” que “While” que “whiLe” que “wHiLe”.... Esto, que parece una tontería, en algunos lenguajes como C o Java no es así y hay que ir con cuidado con ellos. Pero Fénix no tiene ese “problema”.

Bien. Guarda el texto que has escrito en un archivo con extensión PRG en la misma carpeta de Fénix.

Ahora abre la línea de comandos de Windows. Si no te acuerdas cómo se hacía, ve al menú Inicio->Ejecutar y escribe “cmd.exe” (sin comillas). Verás que aparece la típica ventana negra. Muévete a la carpeta donde tienes Fénix ( acuérdate del comando cd...). Esto es muy importante porque si no te mueves

a dicha carpeta, seguramente tendrás errores y no podrás hacer nada. Una vez allí, pues, escribe

**fxc <miprograma>.prg**

donde <miprograma> sea el nombre de fichero para tu PRG (p.e. para compilar el programa de test de Fénix utiliza fxc test.prg). Si todo va bien veras un listado del resultado de la compilación, o un mensaje de error si ha habido algun problema.

Ya has compilado, pero... ¿dónde está tu juego? El compilador genera un fichero resultado con la extensión .DCB y el mismo nombre que el fichero PRG que ha compilado (p.e. test.prg genera test.dcb), en la misma carpeta donde estaba el fichero prg. En nuestro caso, como el fichero prg estabre en la misma carpeta de Fénix, pues el nuevo archivo dcb estará también allí.

Para ejecutar ahora tu programa debes hacerlo usando el interprete (FXI) de Fénix. Para ejecutar el programa teclea en la linea de comandos lo siguiente:

**fxi <miprograma>**

Si todo va bien tu programa se estará ejecutando (p.e. teclea fxi test para ejecutar el test de Fénix). Esta chulo lo que sale, ¿no?: has hecho que la frase "Hola mundo" se mueva con scroll de un lado a otro, y con sólo este código tan corto. Eso muestra un poco de la potencia de Fénix, ¿verdad?. Si no te funcionó, asegúrate de haberlo escrito correctamente y haber elegido la ruta del Fénix correctamente.

No es imprescindible que grabes los archivos PRG en la misma carpeta de Fénix. Siempre podrás hacer, desde dicha carpeta, algo así como: *fxc x:\ruta\donde\esta\mi\prg\archivo.prg*, donde x es la letra que sea. Evidentemente, el DCB se te creará en la misma carpeta donde está el PRG. Es más, ni tan siquiera es imprescindible moverse a la carpeta de Fénix para ejecutar FXC y FXI: podrías ejecutarlos en la línea de comandos desde cualquier ruta donde estuvieras si tienes la precaución de escribir ANTES de hacer nada la siguiente línea:

**PATH=%PATH%;x:\carpeta\de\Fénix**

Esta línea es simplemente un comando de Windows que lo que hace es permitir que todos los programas dentro de, en este caso, la carpeta de Fénix se puedan ejecutar desde cualquier sitio. Vamos, lo que queríamos.

Existe otra manera alternativa de poder compilar y ejecutar tus programas sin tener que recurrir a la consola de comandos, y sin tener que grabar los \*.prg en una carpeta determinada. Basta con arrastrar con el ratón el icono que representa el archivo prg que has escrito, desde donde lo hayas guardado, y soltarlo sobre el icono que representa el fxc.exe (tendrás que tener una ventana abierta con el contenido de la carpeta del Fénix visible). De esta manera, obtendrás igualmente el archivo dcb. Y para ejecutar, tendrás que arrastrar el archivo dcb y soltarlo sobre el icono del fxi.exe.

A lo mejor, después de haber arrastrado y soltado el archivo prg sobre fxc.exe intentas buscar el archivo dcb y no lo encuentras, y a cambio notas que se ha creado un archivo llamado "stdout.txt" . Esto indica que algo has escrito mal en el código, porque el compilador no ha sido capaz de generar el dcb. Si quieres saber cuál ha sido tu error, dentro de "stdout.txt" verás el mensaje de error correspondiente (por ejemplo, si se te ha olvidado escribir un signo) , indicándote además en qué línea de tu código se ha detectado. Así que, si te ocurre esto, comprueba qué es lo que tienes que rectificar en tu código, hazlo y vuélvelo a intentar.

El archivo "stdout.txt" se generará igualmente también si no ha habido ningún error en la compilación, además del dcb. En este caso, la información que contiene es puramente técnica.

Si eres observador, también te darás cuenta que cuando arrastramos el archivo deb y lo soltamos sobre fxi.exe se generará otro archivo llamado "stderr.txt". La tarea de este archivo es, al igual que el anterior "stdout.txt" mostraba errores de compilación, es mostrar si ha habido algún error en la interpretación. Si ves que tu programa no se ejecuta como tú esperabas, puedes observar el contenido de este archivo. De todas maneras, no lo busques mucho: si todo ha ido de forma correcta y no ha habido errores, este archivo se autodestruye cuando se acaba la ejecución del programa.

Respecto a las tabulaciones del texto, has de saber que no son en absoluto necesarias para que la compilación se produzca con éxito. Simplemente son una manera de escribir el código de forma ordenada, clara y cómoda para el programador, facilitándole mucho la faena de “leer” código ya escrito y mantener una cierta estructura a la hora de escribirlo. Enseguida sabrás lo que quiero decir.

Y tranquilo: pronto explicaré qué es lo que hace cada una de las líneas del programa, paso a paso: este ejemplo simplemente era para que vieras cómo funciona la dinámica de editar, compilar e interpretar.

### **El IDE FlameBird2:**

Ya es visto el proceso que se ha de seguir para escribir, compilar e interpretar nuestro programa. Y estarás de acuerdo en que es bastante incómodo utilizar estas herramientas independientes por línea de comandos.

Por eso, algunas personas han creado por su cuenta varios IDEs (“Integrated Development Environment”: recuerda que es básicamente un editor de código con un montón de funciones útiles - corregir códigos, compilar y ejecutar, y muchas funciones más-), a falta de uno oficial. Está en proyecto de desarrollar, ya lo he escrito antes, un entorno oficial multiplataforma hecho en Fénix mismo, pero de momento no hay nada.

De entre los IDEs existentes, creo que el más completo que puedes encontrar para Windows es el FlameBird2. Puedes bajarlo de la web oficial: <http://fbtwo.sourceforge.net> O también de FénixWorld: <http://fenixworld.se32.com>, en el apartado de descargas.

*Por cierto, FénixWorld (no lo he dicho antes), es un portal donde los fans de Fénix escriben sus impresiones sobre los diferentes juegos de Fénix, cuelgan sus proyectos y sus extensiones al lenguaje en forma de dlls, intercambian opiniones y experiencias... bastante recomendable*

Una vez descargado el FlameBird2 y descomprimido su contenido en una carpeta, no hay que hacer nada más. No hay que instalar nada. Dentro de la carpeta ya viene el ejecutable listo para abrir el IDE. Y ya está. Si quieres, puedes crearte un acceso directo desde el menú Inicio.

Al iniciar FlameBird2 te preguntará que tipo de archivos debería asociar con él para que al hacer doble click en alguno de ellos se abriera el FlameBird2 automáticamente. Selecciona “.PRG”, de momento, y “.FBP” también. Y también tienes que seleccionar donde dice “Abrir archivos .deb con el intérprete de Fénix”: esto es para poder interpretar el juego –y por lo tanto, ejecutarlo- desde dentro del propio entorno FlameBird2, sin tener que salir fuera a probar el FXI.

Una vez iniciado por primera vez el FlameBird2, lo que hay que hacer primero es decirle justamente en qué carpeta tienes instalado Fénix, para que el FlameBird2 sepa dónde ir a buscar el compilador y el intérprete cada vez que lo necesites, sin tener que hacerlo tú. Para eso tienes que ir al menú “Editar”, y seleccionando “Preferencias”. Allí habrá una pestaña “Compilación” donde se le puede decir la ruta de la carpeta de Fénix. Si da error en este paso es que no pusiste bien la ruta de Fénix, corrígela. Y ya

tendrás el FlameBird2 listo para usar.

Verás que arriba tienes los iconos típicos (Nuevo -código fuente, proyecto-, Abrir, Guardar, Copiar, Pegar, Compilar y ejecutar, etc.) En medio tienes el espacio para programar. A la izquierda controlas todos los ficheros que necesitará tu programa. A la derecha verás todas las variables, procesos e identificadores que usa tu programa para que puedas tenerlo todo a mano. Abajo aparecen los textos de error al compilar, te acostumbrarás a ello con el uso.

Puedes probar de escribir, grabar el prg en una carpeta cualquiera y ejecutar el mismo programa de antes. Verás que es mucho más cómodo porque lo tienes todo a mano.

Mediante FlameBird2 podrás crear archivos individuales de código fuente, o bien proyectos, que vienen a ser conjuntos de varios archivos de código fuente relacionados entre sí (cuando un programa empieza a ser complejo es conveniente dividir su código fuente en diferentes archivos). Si creas un nuevo proyecto, verás que aparece un nuevo tipo de fichero aparte de los PRG y DCB: los ficheros **FBP**. Estos archivos son los archivos de proyectos del FlameBird2 (si eliges este IDE), y son los que guardan toda la información sobre las relaciones entre los distintos códigos fuente de un mismo proyecto y la configuración general del proyecto. Cuando hagas un proyecto, el archivo que tendrás que seleccionar para abrir todo el proyecto, con sus diferentes PRGs, en el FlameBird2 será éste (acuérdate de que hace cuatro párrafos hemos asociado esa extensión al FlameBird2). En un proyecto se seguirá generando un único DCB.

### **Otros programas importantes: “FPG Edit” y “FNT Edit”**

Existen por ahí otro par de programas que, pese a no ser imprescindibles, son de una gran ayuda porque facilitan un montón la faena del programador, son muy cómodos y son muy útiles.

Se trata del “FPG Edit”, y del “FNT Edit”. Ahora no es el momento de explicar cómo funcionan ni sus características, lo haremos más adelante. Baste con decir que el “FPG Edit” es una especie de gestor de todas las imágenes que usaremos en nuestro juego –las clasifica, ordena, etc-, y el “FNT Edit” es un creador de fuentes de letras específicas para nuestro juego. Ya hablaremos de ellos cuando toque, pero creo que es importante señalarlos aquí como programas que necesitaremos más adelante.

Ambas herramientas las puedes bajar de aquí:

<http://cdiv.sourceforge.net/html/download/down.htm>. O también de FénixWorld:

<http://fenixworld.se32.com/download.php?list.10>. Instálalos, porque los usaremos mucho en este curso.

### **El Fénix Pack:**

Es posible que estés pensando que es un rollo tener que ir descargándote tantas cosas -en realidad no son tantas, pero bueno- desde tantos sitios diferentes: que si Fénix de un lugar, que si el Flamebird de otro, que el FPGEdit y el FNTEdit de otro... Esto mismo es lo que pensaron los chicos del grupo de desarrolladores Coldev, y lo que han hecho ha sido reunir en un único paquete instalable todas estas herramientas -y más-, para disfrute y comodidad de los usuarios de Windows. Es lo que ellos llaman el Fénix Pack, y se puede encontrar en <http://fenixpack.blogspot.com>.

Es decir, en vez de tener que instalar una a una las diferentes herramientas, aspecto un poco pesado sobre todo para el principiante -y peor si no se conocen-, con el Fénix Pack v2.7 dispondremos de un único fichero "Setup", el cual instalará (creando sus correspondientes accesos directos del menú Inicio de Windows y también un desinstalador accesible desde el menú "Agregar/quitar programas" del panel de control de Windows) los siguientes programas, todos ellos de una sola vez, como si fueran uno solo:

\*Compilador e intérprete Fénix 0.92a

\*IDE FlamebirdMX 0.53

\*FPGEdit 2005 0.5.440

\*FNTEdit 2006 0.3.285

\*Generador de sprites desarrollado por el propio grupo Coldev (<http://coldev.blogspot.com>) con soporte para modelos en formato MD2, X, 3DS o B3D. El FenixPack también incorpora otro generador de sprites llamado Charas.EX.

\*Generador de explosiones Explogen (<http://www.kengine.illusionfxnet.com>)

\*Generador de paquetes autoinstalables PakAto de Xwolf (<http://semitec.iespana.es/>)

\*Editores gráficos Idraw3 y EA Graphics Editor (<http://www.nba-live.com/eagraph>)

\*Multitud de códigos fuente de ejemplo, clasificados en tres niveles de dificultad (muchos de los cuales se han utilizado en este manual como soporte para las explicaciones). El nivel avanzado de ejemplos corresponde de hecho a juegos totalmente acabados y funcionales de todo tipo (lucha, carreras, plataformas, rpgs...). Para más información, consultar el último capítulo, donde se detalla más en profundidad el contenido de estos ejemplos.

### **Sobre el uso de la ayuda de Fénix:**

Es completamente imprescindible que consultes y explores la/s ayuda/s online que ofrece/n Fénix, aunque en verdad es éste uno de los aspectos donde este proyecto flaquea más, ya que no existe todavía una ayuda oficial completa, ni completamente actualizada. El centro de información que pretende llegar a ser eso, la referencia central completa y actualizada del lenguaje, ya lo hemos comentado, es <http://fenixwiki.se32.com> (aunque está en proceso de completado), y en inglés, <http://www.fenixdocs.com>. Con el tiempo, aquí tendrás toda la información necesaria para poder programar con plena seguridad: es la referencia completa de los comandos de Fénix (cómo se utilizan, qué parámetros tienen, ejemplos...). Imprescindible.

Si te encontraras en el caso de buscar alguna información en esto dos sitios y no encontrarla porque no está todavía, puedes probar suerte y consultar <http://jlceb.cir.es/fenix>. Ésta es la antigua referencia del lenguaje, para las versiones 0.84/0.84b, que aunque incompleta también, contiene valiosa información todavía.

De todas maneras, a día de hoy, la mejor fuente de información sobre Fénix es, evidentemente, el foro de Fénix: <http://forum.divsite.net>, integrado por una comunidad dispuesta en todo momento a ayudar y a resolver las dudas que surjan respecto a este lenguaje.

Ten en cuenta que el presente manual sólo abarca los conceptos más básicos de la programación, y no repasa ni mucho menos –tampoco es su objetivo– toda la funcionalidad Fénix que puede aportar. Para ampliar y profundizar en el conocimiento del lenguaje y de su entorno, la ayuda es, pues, visita obligada.

### **Recursos web sobre Fénix:**

A continuación tienes una lista que reúne algunas de las más importantes direcciones de recursos web sobre Fénix en español existentes actualmente, para que la tengas como referencia a lo largo de todo el curso:

<http://forum.divsite.net>

Foro Español de Fénix donde la comunidad participa activamente. Muy recomendable. La cantidad de información que se reúne en sus post es de un valor incalculable. La asiduidad de sus miembros y la gran voluntad en aportar nuevas ideas y soluciones y ayudar a los más novatos lo convierten un lugar permanente de intercambio.

<http://fenixwiki.se32.com>

Referencias Online del lenguaje

<a href="http://jlceb.cir.es/fenix">http://jlceb.cir.es/fenix</a> (desactualizada)	<p>En este wiki con el tiempo vendrán explicadas todas las especificaciones sobre cada una de las funciones del lenguaje Fénix (qué parámetros usan, qué devuelven, para qué sirven,etc), sobre las diferentes variables (de qué tipo son, para qué sirven, qué valores pueden tener), sobre las palabras claves del lenguaje (construcción de bucles, condicionales, declaración de variables, creación de procesos,etc), incluso códigos de ejemplo que muestran el uso de las funciones.</p> <p>Es una visita obligada para todo aquél que quiera programa en Fénix. De hecho, recomiendo descargar su contenido en el disco duro local para poderla tener más a mano.</p> <p>Para realizar la descarga de múltiples páginas web de un mismo sitio podemos recurrir a aplicaciones como WebCopier (<a href="http://www.maximuxsoft.com">http://www.maximuxsoft.com</a> ) -comercial- , o HTTrack (<a href="http://www.httrack.com">http://www.httrack.com</a> ) -libre-, o incluso Acrobat (<a href="http://www.adobe.com">http://www.adobe.com</a> ), el cual convertirá toda la referencia en un único pdf muy práctico.</p>
<a href="http://fenixworld.se32.com">http://fenixworld.se32.com</a>	<p>Portal Español especializado en Fénix. Es el punto de reunión de los desarrolladores en Fénix, donde se intercambian impresiones, trucos y donde se cuelgan los proyectos acabados. En esta web hay un apartado donde también se pueden descargar diferentes herramientas que nos pueden ayudar a la hora de programar en Fénix, como pueden ser "FPG Edit" y "FNT Edit", las cuales utilizaremos en este curso ( y el IDE Flamebird2 también).</p>
<a href="http://fbtwo.sourceforge.net">http://fbtwo.sourceforge.net</a>	<p>Web oficial de FlameBird2, el IDE de Fénix que utilizaremos en este curso, desde donde lo podremos descargar.</p>
<a href="http://fenixpack.blogspot.com">http://fenixpack.blogspot.com</a>	
<a href="http://www.fenixdocs.com">http://www.fenixdocs.com</a>	<p>Para la documentación inglesa. También tiene una sección de tutoriales muy interesante, y sobre todo, la sección de descargas, donde están disponibles multitud de librerías DLL que aportan funcionalidad extra al lenguaje Fénix (la Lib3Dm8e.dll, la LibDPlay.dll, la LibOdbc.dll, la LibRegistry.dll, la WPF.dll, la VTE.dll, la net.dll, etc) además de varios IDEs aparte del FlameBird2, el FPGEdit, y el propio Fénix en sus versiones para diferentes plataformas.</p>
<a href="http://div.france.online.fr/">http://div.france.online.fr/</a>	<p>Portal de la comunidad francesa de DIV/Fénix. Con bastante contenido en tutoriales y juegos. Existe un foro.</p>
<a href="http://www.gp32spain.com/foros/forumdisplay.php?f=27">http://www.gp32spain.com/foros/forumdisplay.php?f=27</a>	<p>Foro dedicado específicamente al desarrollo y utilización de Fénix en la consola GP32.</p>

Si te atreves a lidiar con el código fuente de Fénix, por ejemplo para compilar un binario de Fénix más optimizado para tu máquina particular, (ya advierto que se necesitan elevados conocimientos del lenguaje C) , ya sabes que te lo puedes descargar desde la misma página desde donde te descargaste el binario para Windows. Pero si deseas estar a la última de los más recientes cambios que puedan haberse producido en el código, tienes que consultar el repositorio CVS de Fénix. Me explico.

El código que hay en Sourceforge puede sufrir leves modificaciones y mejoras que no se reflejan públicamente hasta que son suficientes para sacar una nueva "release" -versión-, y todos esos pequeñitos cambios se van acumulando en este llamado repositorio CVS, hasta que se decide sacar una nueva versión oficial. Por lo tanto, el código del CVS siempre será una versión un poquito más moderna que la oficial de Sourceforge.

Para acceder a él lo más recomendable es utilizar algún cliente CVS. Windows no tiene ninguno por defecto, así que tendremos que recurrir algún cliente gratuito. Uno libre que no está mal es el WinCVS (<http://cvsgui.sourceforge.net> o <http://www.wincvs.org> ). Una vez instalado, lo único que tendrás que hacer para acceder al código fuente CVS de Fénix es ir a al

menú "Admin"->"Commandline" y escribir el siguiente comando:

```
cvs -z3 -d:pserver:anonymous@fenix.cvs.sourceforge.net:/cvsroot/fenix co -P Fenix
```

Verás que se empieza a descargar el código fuente de Fénix, a tu carpeta personal.

### **La distribución de tus programas:**

Todavía no hemos aprendido a programar ni un solo código nosotros solos, pero hay un detalle que es importante que sepas ya cómo va: el tema de pasar tus juegos a otras personas, y que les funcione en su ordenador. Cuando ya hayas acabado de desarrollar tu juego, y en tu ordenador vaya perfectamente, ¿qué es lo que tienes que dar a otra persona para que lo pueda disfrutar en su ordenador?

Salta a la vista que lo primero imprescindible para que a otra persona le vaya el juego es darle el fichero DCB y ADEMÁS, el intérprete adecuado a su plataforma para que lo pueda ejecutar: FXI.EXE.

Para ello, lo más habitual es incluir estos ficheros necesarios (de momento ya tenemos dos, pero ahora veremos que habrá más), dentro de un paquete tipo Zip. De esta manera, el usuario del juego recibiría un sólo archivo dentro del cual, una vez descomprimido, se encontrarían todos los archivos necesarios para poder ejecutar el juego sin problemas.

Existen bastantes programas compresores-empaquetadores libres y gratuitos que permiten trabajar con multitud de formatos de compresión, entre los cuales está el formato casi omnipresente Zip, pero también el formato RAR, el ACE, etc además de sus formatos propios. Programas de este tipo que puedes utilizar podrían ser: 7-Zip ([www.7-zip.org](http://www.7-zip.org)) o IZARC ([www.izarc.org](http://www.izarc.org)).

El programa FXI.EXE, no obstante, necesita a su vez de otros archivos en su misma carpeta para poder funcionar, que son: **SDL.DLL** y **SDL\_MIXER.DLL** (siempre necesarias) y **LIBPNG-3.DLL** (necesaria si en tu juego usas imágenes PNG), **OGG.DLL**, **VORBIS.DLL** y **VORBISFILE.DLL** (necesarias si en tu juego usas sonidos OGG) y **SMPEG.DLL** (necesaria si en tu juego usas videos MPEG). Así que además de incluir el fichero DCB y FXI.EXE, en el paquete tipo Zip que contenga tu juego distribuible deberás incluir también -en la misma carpeta que FXI.EXE- las DLL que se necesiten (SDL.DLL, SDL\_MIXER.DLL siempre, y las otras depende).

Dije anteriormente que Fénix es multiplataforma, pero si usas estos archivos, por ejemplo, en Linux, el juego no responderá, y es porque el FXI no es compatible. El archivo DCB y los otros (los de imágenes, vídeo y sonido) sí lo son, lo único que tienes que conseguir son los ejecutables de la versión Linux (tendrías que compilar el código fuente descargable de Fénix para ello, tema que requiere ciertos conocimientos elevados de programación y del propio Linux) y sustituir el FXI.exe y las .DLL por las de la versión LINUX y listo; lo mismo vale para otros sistemas operativos

Si tu juego tiene imágenes -las tendrás- ,es evidente que las tendrás que meter también dentro del paquete que quieres distribuir; así que tendrás que incluir todos los archivos FPG que hayas utilizado, o bien, si en tu código has cargado las imágenes individualmente -con *load\_png* por ejemplo, tendrás que incluir en el paquete una a una las imágenes PNG por separado (del tema imágenes ya hablaremos más adelante). Además, si incluyes sonidos, también tendrás que meterlos dentro del paquete a distribuir, incluyendo pues cada uno de los archivos WAV,OGG,MOD,PCM,XM... individuales. Lo mismo si has utilizado videos, o cualquier otro tipo de archivo auxiliar.

Es importante que te acuerdes de poner (dentro del paquete a distribuir) todos los videos,sonidos e imágenes en las direcciones relativas que usaste en la carga de éstos dentro del código fuente. Es decir, si pusiste *load\_fpg("gráficos.fpg")* tendrás que guardar el archivo graficos.fpg en la misma carpeta que el

DCB, pero si pusiste *load fpg("imagenes/graficos.fpg")* debes crear una carpeta dentro de la que tienes el DCB llamada "imágenes" y meter en ella el archivo "graficos.fpg". Esto lo entenderás mejor cuando veamos el tema de la carga de imágenes –o video o sonido, es igual-.

Otro detalle muy importante es que en otros sistemas operativos no Windows no es lo mismo "Uno" que "uno", es decir, que distinguen entre mayúsculas y minúsculas, y es necesario que al escribir las direcciones donde tienes los ficheros te asegures que escribes las mayúsculas y minúsculas de la misma manera.

Ya tienes el paquete creado con todos los archivos necesarios en su interior. Lo importante del tema ahora es que si entregas por ejemplo tu programa en un Zip que incluye tu DCB, más los archivos de imágenes/sonidos/videos, más el FXI.EXE más las DLL necesarias, el usuario final que lo único que quiere es jugar, no tendrá ni idea de lo que tiene que hacer cuando reciba este conjunto de archivos. ¿Dónde clicas: en el EXE? Le saldrá un pantallazo negro...

El truco es el siguiente: renombras el archivo FXI.EXE a <nombrearchivodcb>.EXE. Es decir, si tienes inicialmente el archivo DCB que se llama "Pepito.dcb", haces que el intérprete FXI.EXE se llame ahora "Pepito.exe". Y ya está. El jugador potencial hará clic en el único exe que ve y ahora sí, el juego se ejecutará sin más problema.

Incluso podrías cambiarle además la extensión al archivo DCB y ponerle la extensión DAT –queda como más profesional, ¿no?- , y funcionará también. Puedes probarlo, si quieres, con nuestro primer programa Fénix que hemos visto y un amigo con su ordenador.

Ya has visto que el icono del FXI.EXE es una especie de ave de fuego, y si renombramos el archivo como nuestro nuevo juego, nuestro juego tendrá como icono ese mismo ave. Existe una manera (que yo conozca) de cambiar este icono y poner el que nosotros queramos, pero de eso hablaremos en otro capítulo posterior, cuando comentemos precisamente el uso de este tipo de imágenes.

Un aspecto curioso respecto al tema de la distribución de nuestro juego es que disponemos de la posibilidad de crear "stubs". Y esto ¿qué es? Pues generar a partir de nuestro archivo PRG, en vez de un archivo DCB, un archivo EXE. Este archivo EXE incorporará en su interior el intérprete FXI.exe más el código DCB del juego. Es decir, que en vez de tener el intérprete por un lado y el código compilado por el otro, podremos tener las dos cosas juntas (el intérprete "con mochila") en un ejecutable. Esto se hace en el proceso de compilación: usaremos nuestro compilador fxc.exe de forma normal, pero ahora haciendo uso de un parámetro concreto del fxc.exe, el -s, así:

**fxc -s fxi.exe nombreDelJuego.prg**

Lo que hemos hecho aquí es indicar explícitamente que al archivo generado en la compilación (lo que hasta ahora había sido un archivo DCB) se le va a "incluir dentro" un ejecutable, indicado justo después del parámetro -s, que en este caso (y siempre) será el "fxi.exe". El resultado será un ejecutable llamado nombreDelJuego.EXE.

Pero todavía podemos ir más allá: podemos incluir dentro de este archivo nombreDelJuego.EXE a más a más todos los archivos que utilizemos en el juego (archivos de sonido, imágenes, videos, archivos FPG,etc,etc), de tal manera que podamos distribuir nuestro juego en forma de un único archivo ejecutable, sin nada más. (Ésta sería una manera posible de dificultar la apropiación de parte de otras personas de los archivos de recursos del juego -músicas, imágenes,etc-). Para lograr esto, tenemos que escribir:

**fxc -a -s fxi.exe nombreDelJuego.prg**

Fíjate que simplemente hemos añadido el parámetro -a. Un detalle importante es que, para que esta opción funcione, todos los recursos que se incrusten en el ejecutable han de estar dentro de carpetas cuyas rutas relativas especificadas dentro del código fuente tienen que ser las mismas que respecto al directorio donde está ubicado el fxc.exe (es decir, se toma la carpeta donde está fxc.exe como directorio base). Cuidado con esto.

Con la opción -a (y la -s también, obviamente) hay un tipo de archivos -sólo ése- que no se incluye dentro del ejecutable: las librerías DLL. Esto es así por la propia naturaleza de estas librerías. Así que, si utilizas la opción -a (junto con la -s) lo que obtendrás finalmente será un ejecutable único, pero necesitarás incorporar además en el paquete Zip las DLL necesarias para que el juego se pueda ejecutar correctamente.

Finalmente, existe otra opción interesante del compilador que es la opción -f seguida del nombre/ruta de un fichero, así:

**fxc -f graficos.fpg -s fxi.exe nombreDelJuego.prg**

Esta opción sirve para lo mismo que la opción -a (es decir, para incluir ficheros dentro del ejecutable resultante de la compilación) pero sólo para incluir un fichero concreto ( el especificado después de -f). Es decir, en vez de incluir todos los ficheros de recursos, sólo se incluye uno, el que se le diga.

El compilador de Fénix tiene más opciones interesantes: puedes ver el listado completo si escribes simplemente **fxc** y nada más en la línea de comandos y consultas el texto escrito en el archivo "stdout.txt".

De todas maneras, supongo que estarás pensando que distribuir tu juego dentro de un simple archivo Zip es demasiado "cutre". A tí te gustaría que tu juego se pueda distribuir en un único ejecutable "Setup", y que la gente, al clicar en él, instalara el juego en su ordenador como cualquier otro programa: con un asistente que copiara automáticamente todos los archivos a una ruta especificada por el usuario, que fuera preguntando si se desea una entrada en el menú Inicio de Windows o si quiere crear un acceso directo en el escritorio, etc...el procedimiento habitual en una instalación de cualquier programa en Windows.

Para conseguir esto, necesitas utilizar algún programa diseñado para esto: los generadores de instalaciones. Aquí tienes una lista -incompleta y limitada- de algunos de estos programas que te pueden ayudar, por si te los quieres mirar.

<b>Nsis</b> ( <a href="http://nsis.sourceforge.net">http://nsis.sourceforge.net</a> )	Es libre.
<b>Wix</b> ( <a href="http://sourceforge.net/projects/wix">http://sourceforge.net/projects/wix</a> )	Es libre
<b>Inno Setup Compiler</b> ( <a href="http://www.jrsoftware.org">http://www.jrsoftware.org</a> )	Es freeware (gratis pero no libre)
<b>Setup Generator</b> ( <a href="http://www.gentee.com/setupgen">http://www.gentee.com/setupgen</a> )	Es freeware
<b>AdminStudio</b> ( <a href="http://www.macrovision.com">http://www.macrovision.com</a> -dentro del apartado FLEXnet InstallShield-	
<b>Advanced Installer</b> ( <a href="http://www.advancedinstaller.com">http://www.advancedinstaller.com</a> )	
<b>CreateInstall</b> ( <a href="http://www.createinstall.com">http://www.createinstall.com</a> )	
<b>InstallConstruct</b> ( <a href="http://www.mg-india.com">http://www.mg-india.com</a> )	
<b>Installer VISE</b> ( <a href="http://www.mindvision.com">http://www.mindvision.com</a> )	

### Y de regalo, tu segundo programa en Fénix:

¿Te has quedado con las ganas de escribir alguna otro programilla y ver qué hace? Venga, prueba con éste (respeta las tabulaciones):

```
Program Misegundoprograma;

private
    int mivar1;
end

begin
    loop
        delete_text(0);
        mivar1=rand(1,10);
        if (mivar1<=5)
            write(0,200,100,2,"Menor o igual que 5");
        else
            write(0,200,100,2,"Mayor que 5");
        end
        frame;
    end
end
```

A ver si eres capaz de intuir qué es lo que hace este código, aún sin saber nada del lenguaje. Comprueba lo que ocurre al ejecutarlo.

## CAPÍTULO 2: EMPEZANDO A PROGRAMAR CON FÉNIX

### Explicación paso a paso de “Mi primer programa en Fénix”

Recordemos el código que utilizamos para ver ese scroll tan bonito de la frase “Hola mundo”:

```
Program MiPrimerPrograma;  
  
Private  
    int mivar1;  
End  
  
Begin  
    mivar1=10;  
    while(mivar1<320)  
        delete_text(0);  
        mivar1=mivar1+2;  
        write(0,mivar1,100,1,"¡Hola mundo!");  
        frame;  
    end  
end
```

Vamos a ir viendo poco a poco el significado de las diferentes líneas de este código, el cual nos servirá para introducir diferentes conceptos importantes sobre programación en general y programación en Fenix en particular.

#### **Palabra reservada “Program”:**

Fijémosnos en la primera línea:

```
Program MiPrimerPrograma;
```

Esto es lo que se llama la cabecera del programa. Todos los programas deben comenzar con la palabra reservada PROGRAM seguida del nombre del programa (cualquier nombre) y un símbolo ; (punto y coma).

Por “palabra reservada” se entiende cualquier palabra o comando propio e intrínseco del lenguaje Fenix, (es decir, es un nombre que está reservado como palabra clave del lenguaje de programación) y que por tanto, se ha de escribir tal cual –aunque recordad que **Fenix es “case-insensitive”**-, y no se puede utilizar para ninguna otra función que no sea la que Fenix tiene prevista para ella.

Esta cabecera es obligatoria en todos los programas. Acordarse del punto y coma final.

Antes de la misma sólo puede aparecer, de forma opcional, uno o varios comentarios.

#### **Comentarios:**

Un comentario, por cierto, es simplemente una nota aclaratoria sobre el programa. Los comentarios no son necesarios para el correcto funcionamiento del programa, aunque son muy útiles para entender su funcionamiento. Sirven sobre todo para explicar dentro del código fuente partes de éste que puedan ser más o menos complejas. Hay dos tipos de comentarios:

-De una sola línea, comienzan con el símbolo // y terminan al final de la línea en que se definen.

-De varias líneas, comienzan con el símbolo /\* y terminan con el símbolo \*/. También pueden también comenzar y terminar en la misma línea.

Todos los textos incluidos en un comentario son ignorados por el compilador. Se pueden poner tantos comentarios en un programa como sea necesario, y en cualquier punto del programa.

Por ejemplo, nuestro programa se podría haber escrito:

```
//Esto es un comentario de una línea.  
Program MiPrimerPrograma;  
  
/*Y éste  
comentario es  
un comentario  
de varias*/  
Private  
    int mivar1;  
End  
  
Begin  
    mivar1=10;  
    while(mivar1<320)  
        delete_text(0);  
        mivar1=mivar1+2;  
        write(0,mivar1,100,1,"¡Hola mundo!");  
        frame;  
    end  
end
```

### **Bloque “Private/end”:**

Los ordenadores tienen una memoria en la que pueden almacenar valores. Una “variable” es una posición concreta de la memoria del ordenador que contiene un valor determinado que es utilizado en un programa. Ese valor puede ser un número entero, decimal, un carácter, una cadena de caracteres, etc. En su programa el desarrollador puede asignar a las variables un nombre, como por ejemplo “x”; y en ese programa “x” se utilizaría para referirse al valor que contiene.

Declarar una variable es “crear” esa variable; es decir: reservar un espacio de la memoria del ordenador con un nombre, disponible para almacenar allí un valor determinado. Declarar una variable es como dejar sitio en el cajón de la habitación, y ponerle una etiqueta, para saber que ese lugar está reservado para colocar tus calcetines, por ejemplo.

En Fénix se delimita muy claramente la parte del programa que se encarga de declarar variables –al principio-, de la parte que el código a ejecutar propiamente dicho. Primero se declaran las variables que pretendemos usar, y una vez hecho esto, se comienza el programa en sí.

Fijarse también que esta vez las líneas PRIVATE y END no acaban en punto y coma.

Es muy importante saber que hay cuatro clases de variables (también llamadas “datos”) claramente diferentes en Fénix según sus prestaciones y objetivos –y excluyentes entre sí- : las variables

privadas –la que acabamos de ver- , las variables públicas, las variables locales y las variables globales. Cuando hable de procesos en el siguiente capítulo explicaré la diferencia entre estas clases y el porqué de tal importante distinción.

Baste saber por ahora que para declarar una variable local, lo único que tendríamos que hacer es reemplazar la palabra “private” por “local”; y para declarar una variable global, habría que reemplazarla por “global”. Las variables públicas funcionan de una manera diferente y no se tendrán en cuenta en este manual hasta más adelante.

Así pues, nada nos impide poder declarar variables de estos tres tipos (privadas, locales y globales) diferenciados en un mismo programa, lo cual en general tendría finalmente el siguiente aspecto:

```
Program miprograma;  
Global  
//Declaración de variables privadas  
End  
  
Local  
//Declaración de variables locales  
End  
  
Private  
//Declaración de variables globales  
End  
  
Begin  
//Programa principal  
End
```

De hecho, lo que acabo de escribir es una plantilla que siguen todos los programas en Fénix: la sentencia “program” inicial, la sección de declaraciones de las variables que son necesarias de los tres tipos posibles (el orden es indiferente, y si no hay variables de un tipo determinado, evidentemente no hace falta escribir su sección de declaraciones correspondiente), y la sección del código principal.

### **Línea: “int mivar1;”. Tipos de datos en Fénix:**

Ésta línea es la que propiamente declara una variable, que la llamamos “mivar1”, y además dice de qué tipo es.

El nombre de la variable nos lo podemos inventar absolutamente (hay algunas excepciones exóticas: no podemos darle un nombre que sea igual a algún comando existente de Fénix por razones obvias –habría confusión-; o que el nombre no puede empezar por un número; o que no puede incluir espacios en blanco – lógicamente, porque Fénix no sabría qué es la segunda palabra-,etc).

El tipo de la variable indica qué contenido es el que la variable está preparada para contener. Fénix necesita saber siempre qué tipo de valores puede o no albergar una variable determinada. No es lo mismo que una variable sea declarada para poder guardar números enteros que para guardar cadenas de caracteres. Si declaramos una variable como entera, si quisiéramos asignarle un valor de cadena de caracteres daría un error. Así pues, siempre, cuando se declara la variable, hay que decirle que será de un tipo determinado, y que por tanto, los valores que incluirá, sean los que sean, siempre serán de ese tipo determinado. Dependiendo de lo que queramos hacer con esa variable, nos convendrá crearla de un tipo o de otro, según lo que tengamos previsto almacenar en ella.

Los **tipos** de variables disponibles en Fénix son:

<b>BYTE</b>	Almacenan valores numéricos enteros que pueden valer entre 0 y 255. Una variable de este tipo necesita 8 bits de memoria (sin signo y sin coma decimal) para almacenar ese valor.
<b>SHORT</b>	Almacenan valores numéricos enteros que pueden valer entre -32767 y +32768 Una variable de este tipo necesita 16 bits de memoria (con signo y sin coma decimal) para almacenar ese valor.
<b>WORD</b>	Almacenan valores numéricos enteros que pueden valer entre 0 y 65535. Una variable de este tipo necesita 16 bits de memoria (sin signo y sin coma decimal) para almacenar ese valor.
<b>INT</b>	Almacenan valores numéricos enteros que pueden valer entre -2147483648 y +2147483647. Una variable de este tipo necesita 32 bits de memoria (con signo y sin coma decimal) para almacenar ese valor.
<b>DWORD</b>	Almacenan valores numéricos enteros que pueden valer entre 0 y 4294967296. . Una variable de este tipo necesita 32 bits de memoria (sin signo y sin coma decimal) para almacenar ese valor.
<b>FLOAT</b>	Almacenan valores numéricos decimales, de poca precisión. Estos números decimales son del tipo llamados "con coma flotante"; es decir: el número de decimales no es fijo porque puede variar en función de las operaciones matemáticas que se realicen con ellos. A la hora de escribir un número float , el símbolo utilizado para separar la parte entera de la parte decimal es el punto: "." Una variable de este tipo necesita 32 bits de memoria (con signo y con coma decimal) para almacenar ese valor.
<b>STRING</b>	Almacenan cadenas de texto (una serie de varios caracteres que también puede estar vacía), de longitud variable -potencialmente ilimitada-, y que estarán comprendidas entre comillas dobles o simples. Una variable de este tipo necesitará más o menos bits de memoria para almacenar un valor según la cantidad de caracteres que tenga esa cadena: más corta la cadena, menos espacio necesitará.
<b>CHAR</b>	Almacenan un único carácter (una letra, signo, espacio o carácter especial de control). Es posible realizar operaciones numéricas con un carácter o asignar su valor a una variable de tipo entero, con lo que se extrae el valor ASCII del carácter. Una variable de este tipo necesita 8 bits de memoria para almacenar este valor; por lo tanto, hay 256 (2 <sup>8</sup> ) posibles caracteres diferentes reconocidos por Fénix, normalmente usando el juego de caracteres ISO8859-1, similar al estándar de Windows.
<b>POINTER</b>	Hacen referencia a una dirección de memoria donde está alojada una variable, NO al valor contenido en ella. No lo veremos en este curso.

También podríamos crear nosotros, como programadores que somos, nuestros propios tipos de datos, con el bloque TYPE/END, pero este tema lo abordaremos más adelante.

Llama la atención que haya tantos tipos de datos diferentes para almacenar números enteros. Esto es porque si sabemos que los valores de una variable nunca van a ser mayores que 255 por ejemplo, sería una tontería declarar una variable de tipo INT porque se estaría desperdiciando gran cantidad de memoria del ordenador para nada. Es como si tuviéramos un gran cajón del cual sólo utilizamos una pequeña parte. Debemos ajustar el tipo de las variables a los valores que preveemos que van a contener para ajustar recursos y no despilfarrar memoria con variables que la requieren pero que luego no se aprovecha.

Otra cosa importante: los tipos de dato enteros tienen un rango limitador: cuando sumas o restas una

cantidad que provoca que el número salga fuera del rango que admiten, los bits más significativos del nuevo número se pierden. En otras palabras menos técnicas: se provoca que el número "dé la vuelta": si a una variable WORD que contiene 65535 le sumas 1, el valor resultante es 0 en lugar de 65536, ya que 65536 queda fuera del rango admitido por los tipos WORD.

Todos los datos de tipo entero vistos en el apartado anterior permiten especificar como prefijo **SIGNED** o **UNSIGNED** para obtener versiones de los mismos con o sin signo. Esto equivale a usar un tipo de dato entero del mismo número de bits pero con la salvedad del signo. Por ejemplo, **UNSIGNED INT** equivale exactamente a **DWORD**. La única variante de este tipo que no tiene una palabra clave como alias es **SIGNED BYTE**, un entero de 8 bits que puede contener un número entre -128 y 127.

Es posible que alguna vez os encontréis con algún código donde declaren las variables sin especificar su tipo. Es una práctica que yo no recomiendo, pero basta saber que Fenix asume por defecto que esa variable es tipo INT.

Vemos claramente pues, volviendo a nuestro ejemplo, que la variable "mivar1" es de tipo INT. Para definir el tipo de una variable cuando se declara, pues, es siempre

```
tipo nombre;
```

Opcionalmente, a la vez que escribimos esta línea donde declaramos la variable y definimos su tipo, podríamos también a la vez rellenar de valor la variable, lo que se dice inicializar la variable –darle su valor inicial-. Si la declaramos y ya está, automáticamente su valor queda establecido por Fénix a 0 (cero). Para hacer esto, simplemente es de la forma

```
tipo nombre= valor;
```

En el ejemplo no lo hemos hecho así, pero perfectamente podríamos haber escrito

```
int mivar1=10;
```

y habernos ahorrado la línea posterior *mivar1=10;* (escribiendo pues en una sola línea la declaración y la inicialización de la variable), ya que de hecho lo único que hace esa línea precisamente es darle un valor a "mivar1" (ahora lo veremos).

Un último detalle que conviene comentar es el hecho de que Fénix tiene predefinidas una serie de variables que sirven para tareas concretas –normalmente son variables locales-. Esto quiere decir que no podremos crear nosotros variables que tengan el mismo nombre que estas variables "de sistema" para evitar confusiones. Cada una de estas variables predefinidas realizan una tarea específica diferente de las demás y su utilidad es muy grande, como iremos viendo poco a poco. Por ejemplo, Fénix incorpora en su implementación el uso de una variable denominada "x", que ya veremos para qué sirve. Esta variable "x" la podremos usar siempre que necesitemos recurrir a las ventajas concretas que nos pueda ofrecer, y hemos de procurar no llamar a ninguna de nuestras variables personales "x", porque se podría confundir con la predefinida, aunque no la estemos usando. Hay que decir que para utilizar las variables predefinidas no hay que declararlas ni nada. Profundizaremos en esto más adelante.

Fijarse que ahora volvemos a acabar la línea con punto y coma.

### **Bloque “Begin/end”:**

El código principal de un programa –donde están los comandos que va a ir ejecutando- siempre comienza con la palabra reservada BEGIN (“empezar”), tras la cual puede aparecer cualquier número de sentencias(=órdenes=comandos,etc), y siempre finaliza con la palabra reservada END (“acabar”). (En nuestro código hay dos “ends”: el del begin es el último de todos).

Es decir, que de hecho, el mínimo programa que podríamos escribir en Fénix (que no use variables) y que funcionaría sería algo así como

```
Program MínimoPrograma;  
Begin  
End
```

Por supuesto, si lo ejecutaras, no pasaría nada, como mucho la pantalla se pondría negra un momento y puede que saliera otra diciendo que el programa ha finalizado, y es que realmente no hemos hecho nada.

Fijarse que esta vez las líneas BEGIN y END no acaban en punto y coma.

### **Línea “mivar1=10;”:**

Un programa puede poner en cualquier momento de su ejecución un valor concreto en una variable.

Para poner el valor numérico 123 en la variable “x” por ejemplo, el programa utilizará una sentencia (orden) como la siguiente, (si la variable no la hemos declarado previamente como de tipo entero, puede que dé error):

```
x = 123;
```

Los programas pueden también consultar el valor de las variables, o utilizarlos en diferentes expresiones; por ejemplo, para poner en una variable entera llamada "y" el valor de "x" más 10, se utilizaría una sentencia como la siguiente:

```
y = x + 10;
```

Antes de que el ordenador ejecutara esta sentencia, “y” podría tener cualquier valor pero, después de ejecutarla, la variable contendría exactamente el resultado de sumarle 10 al valor numérico que tuviera la variable "x" en ese momento.

Queda claro después de todo lo dicho, que la tercera línea de nuestro programa lo que hace es asignarle a una variable “mivar1” creada por nosotros el valor de 10. Con ese valor almacenado en memoria con el nombre de “mivar1” y que le hemos dado el valor 10 se supone que en líneas siguientes haremos algo.

Fijarse que ahora volvemos a acabar la línea con punto y coma.

Por cierto, digo ya que las operaciones que podemos realizar con valores numéricos son la suma (+), la resta (-), la multiplicación (\*), la división (/) y el módulo (%), que es en realidad el resto de una división –es decir, que 12%5 sería 2, 27%4 sería 3, 3%2 sería 1,etc-.

Un detalle muy importante que hay que tener en lo que respecta a estas operaciones aritméticas

básicas es que el tipo de datos del resultado (int,word,byte,float...) vendrá dado por el tipo de datos de los operandos. Es decir, que una suma de dos números Int dará un Int, la multiplicación de dos números Float dará un Float, etc.

Esto es especialmente delicado cuando estamos realizando estas operaciones (división, multiplicación,etc) entre números decimales y sin darnos cuenta asignamos el resultado a una variable de tipo entera. Lo que ocurrirá es que el resultado se trunca sin redondear, por lo que si éste era por ejemplo 32.94, se convertirá en un 32; si era un 86.351 será un 86.

Cuando un operando es de un tipo (por ejemplo, byte) y el otro operando es de otro tipo (por ejemplo, int), el tipo del resultado será siempre igual a aquél tipo de los dos que esté por encima en la jerarquía siguiente: float>dword>int>word>short>byte.

### **Bloque “While/End”:**

La sentencia WHILE (“mientras”) es una sentencia que implementa un bucle, es decir, que es capaz de repetir un grupo de sentencias un número determinado de veces. Para implementar este bucle se debe especificar entre paréntesis la condición que se debe cumplir para que se ejecute el grupo de sentencias a continuación de la palabra reservada WHILE. Tras especificar esta condición, se pondrán todas las sentencias que se necesita que se repitan y, finalmente, se marcará el final del bucle con la palabra reservada END (no importa que dentro del bucle aparezcan más palabras END si éstas forman parte de sentencias interiores a dicho bucle).En nuestro ejemplo, el END correspondiente al While sería el segundo empezando por el final.

Así que la estructura genérica sería:

```
While (condición)
    ...
End
```

Cuando se ejecute la sentencia WHILE se realizará la comprobación que se especifica y, si ésta resulta cierta, se ejecutarán las sentencias interiores; en caso contrario, se continuara el programa a partir del END que marca el final del WHILE.

Si se han ejecutado las sentencias interiores (lo que se denomina realizar una iteración del bucle), se volverá a comprobar la condición y, si ésta vuelve a ser cierta, se realizará otra iteración (se volverán a ejecutar las sentencias interiores). Se denomina iteración a cada ejecución del grupo de sentencias interiores de un bucle. Este proceso continuará hasta que, al comprobarse la condición del WHILE, ésta resulte falsa.

Si se llega por primera vez a una sentencia WHILE la condición resulta falsa directamente, entonces no se ejecutarán las sentencias interiores ninguna vez.

Las sentencias interiores a un bucle WHILE pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles WHILE

En nuestro ejemplo concreto, tenemos el bucle:

```
while(mivar1<320)
    delete_text(0);
    mivar1=mivar1+2;
    write(0,mivar1,100,1,";Hola mundo!");
frame;
```

*end*

Podemos comprobar que hay una condición –que “mivar1” sea menor que 320-, que es la que dirá si hay que ejecutar o no (y si es que sí, hasta cuándo) las siguientes cuatro líneas. Justo en la línea de antes hemos visto que dábamos el valor de 10 a dicha variable. Pues bien, la condición del while no hace más que comprobar si el valor de esa variable “mivar1” es menor que 320. Como nada más llegar al while eso es verdad ( $10 < 320$ ), se pasa a ejecutar una a una las sentencias incluidas dentro del bucle, hasta llegar al end. Cuando la ejecución del programa llega al end, se vuelve otra vez a la línea del while y se vuelve a comprobar la condición a ver si continúa siendo verdadera. Si “mivar1” continúa siendo menor que 320, se vuelve a ejecutar el interior del bucle, y así, y así hasta que llegue un momento que a la hora de efectuar la comparación otra vez, “mivar1” ya no valga menos que 320. En ese caso, el programa salta todas las líneas internas del bucle y continúa su ejecución justo después del end del while. Se supone que en algún momento durante la ejecución de las sentencias del interior del bucle, el valor de “mivar1” cambiará, porque si no cambiara, “mivar1” siempre valdría 10 y la condición siempre sería verdadera, por lo que siempre se ejecutarían las sentencias del interior del bucle y eso daría lugar a un bucle infinito: el programa no se acabaría nunca.

Fijarse que en nuestro programa cuando la condición sea falsa y el programa vaya a la siguiente línea después del end del while, se encontrará con que ya no hay nada más: (el end del programa); con lo que la ejecución del programa se acabará cuando la condición del while sea falsa.

Fijarse también que esta vez las líneas WHILE y END no acaban en punto y coma.

### **Orden “Delete\_text”:**

La primera línea que nos encontramos dentro del bucle del while es el primer comando propiamente dicho que “hace algo”. Este comando se encarga, como su nombre indica, de borrar de la pantalla todos los textos que haya en ese momento. ¡Pero si de momento no hay ningún texto que borrar: sólo le hemos dado un valor a una variable!, dirás. Es verdad, pero el texto. escribiremos más adelante. Pero entonces, dirás otra vez: ¿por qué no escribimos el texto, y luego, si hay que borrarlo, se borra? ¿Por qué lo que se hace es borrar primero la pantalla cuando no hay nada y luego escribir en ella? Bueno, tiempo al tiempo, pronto lo explicamos...

Es importante, ya que nos hemos topado con el primer “comando” de Fenix, aclarar el concepto de parámetro. Verás que cuando vayan apareciendo más y más comandos, TODOS llevarán al final de su nombre un par de paréntesis –()- y el consabido punto y coma. Dentro del paréntesis puede o no haber nada, o haber un número/letra/palabra/signo... (como en el caso de delete\_text , o haber dos números/letras/palabras/signos... separados por comas, o haber tres números/letras/palabras/signos... separados por comas, o haber cuatro,etc. Cada uno de los valores que aparecen dentro de los paréntesis se denominan parámetros de ese comando, y el número de ellos y su tipo (si son números, letras, palabras,etc) depende de cada comando en particular. ¿Y para qué sirven los parámetros? Para modificar el comportamiento del comando en algún aspecto.

Me explico: los comandos que no tienen parámetros son los más fáciles, porque hacen una cosa, su tarea encomendada, y punto. No hay posibilidad de modificación: siempre harán lo mismo de la misma manera. Cuando un comando tiene uno o más parámetros, también hará su función preestablecida, pero el cómo la hará viene dado por el valor de cada uno de los parámetros que tenga, los cuales modificarán alguna característica concreta de esa acción a realizar.

Por ejemplo: supongamos que tenemos un comando, el comando “lalala” que tiene –que “recibe”, técnicamente dicho- un parámetro. Si el parámetro vale 0 lo que hará el comando será imprimir por pantalla “Hola, amigo”; si el parámetro vale 1 lo que hará el comando será imprimir por pantalla “¿Qué tal

estás?” y si el parámetro vale 2 imprimirá “Muy bien”,etc. Evidentemente, en las tres posibilidades el comando “lalala” hace esencialmente lo mismo: imprimir por pantalla (para eso es un comando, sino podríamos estar hablando de tres comandos diferentes), pero dependiendo del valor de su único parámetro –numérico, por cierto: no valdría en este caso dar otro tipo de valor (como una letra) porque si no seguramente daría error- su acción de imprimir por pantalla un mensaje es modificada en cierta manera. Así,

Lalala(0);   □ Imprime “Hola amigo”  
Lalala(1);   □ Imprime “¿Qué tal estás?”  
Lalala(2);   □ Imprime “Muy bien”

Para entendernos: los comandos serían como los verbos, que hacen cosas, que ejecutan órdenes, y los parámetros serían como los adverbios, que dicen cómo se han de hacer estas cosas, de qué manera se ejecutan esas órdenes (más lento, más suave, en la esquina superior o inferior de la pantalla,etc).

Dicho esto, vemos claramente que delete\_text recibe un único parámetro que ha de ser numérico. Y ¿qué significa en concreto que ese parámetro valga 0? Pues significa lo que hemos dicho ya: que delete\_text se dedicará a borrar TODOS los textos que aparezcan por pantalla. Poniendo otros valores como parámetros –ya lo veremos-, se podría especificar más y decirle a delete\_text que borrara tal o cual texto concreto.

Por cierto, poner como valor del parámetro de delete\_text la palabra "all\_text" (sin comillas) es equivalente a poner el número 0.

### **Línea “mivar1=mivar1+2;”:**

He aquí una línea que puede sorprender a quien no haya programado nunca: una “igualdad” que no es igual...¿Cómo es eso? Pues porque esto no es una igualdad: es una asignación. El valor de mivar1+2 se le asigna como nuevo valor a mivar1. Las asignaciones siempre hay que leerlas de derecha a izquierda.

Hemos comentado que las variables son trocitos de memoria del ordenador con un nombre donde se guardan valores usados por el programa.Pues bien, lo que hace esta línea es coger el valor que tiene mivar1, sumarle dos, y el resultado guardarlo en mivar1 otra vez (con lo que perderíamos su antiguo valor). Es como abrir un cajón –la variable- y coger su contenido, sumarle dos, y posteriormente poner ese nuevo contenido sumado en el mismo cajón de antes (podría haber sido otro cajón, también).

De hecho, en Fenix, cuando queramos especificar que algo ES IGUAL a algo, no usaremos el signo = -signo de “asignación” -, sino que se usará el signo == (dos iguales) –signo de “igualdad”. Cuidado porque son signos diferentes con significado diferentes y pueden dar muchos problemas para el que se despista. Un ejemplo de utilización del signo == sería por ejemplo en un while:

```
While (mivar1 == 30)
...
End
```

En este ejemplo, lo que se estaría comprobando es si mivar ES IGUAL a 30, y mientras eso sea verdad, realizar el bucle. Hubiera sido un gran error haber puesto

```
While (mivar1 = 30)
...
End
```

ya que la condición del while en este caso SIEMPRE sería verdad, porque lo que estaría pasando es que se le asignaría el valor 30 a la variable mivar1 (y eso siempre es verdad), y por lo tanto, nunca se incumpliría una condición que de hecho no es tal.

Y para acabar, normalmente, en vez de hablar de signos, en la jerga técnica hablaremos de operadores (son sinónimos).

Por cierto, seguramente cuando empieces a leer códigos de otros desarrolladores te percatarás de que utilizan operadores un tanto peculiares. Estos operadores son del tipo ++,--,+=,=-,\*=,/=, etc.No son imprescindibles, porque lo que hacen se puede hacer con los operadores de toda la vida (+,-,\*,/,etc) pero se utilizan mucho. En este manual no los utilizaremos por razones pedagógicas: para así facilitar aún más la comprensión de los códigos fuente a aquellos que se inician en el mundo de la programación; sin embargo, es probable que cuando cojas un poco más de soltura empieces a usarlos porque facilitan mucho la escritura (y lectura) del código y para el programador son muy cómodos de utilizar.

Pero, ¿qué es lo que hacen? A continuación presento una tabla de “equivalencias” –incompleta- entre el uso de algunos de los operadores “raros” de los que estamos comentando y su correspondencia con el uso de los operadores “normales” equivalente.

Así pues, las expresiones:

a++	Equivale a	a=a+1 .(Al operador “++” se le llama operador “incremento”)
a--	Equivale a	a=a-1 (Al operador “--” se le llama operador “decremento”)
a+=3	Equivale a	a=a+3
a-=3	Equivale a	a=a-3
a*=3	Equivale a	a=a*3
a/=3	Equivale a	a=a/3

### Orden “Write”:

Podéis observar que este nuevo comando tiene bastantes parámetros. Su tarea principal es, como su nombre indica, escribir un texto determinado en pantalla. Pero,¿qué texto? ¿En qué sitio de la pantalla?...eso lo dirán el valor de sus parámetros.

No obstante, antes de saber cómo funcionan dichos parámetros, es necesario aprender algo: el origen de coordenadas en Fenix está en la esquina superior izquierda de la ventana donde se visualiza el videojuego (y si el videojuego se ve a pantalla completa, pues será la esquina superior izquierda de la pantalla).Ese punto es el punto (0,0).Podréis comprobarlo fácilmente si ponéis esas coordenadas en write junto con una alineación que conozcáis. A partir del punto (0,0), si nos movemos para la derecha estaremos incrementando la coordenada X, y si nos movemos para abajo estaremos incrementando la coordenada Y. Un esquema sería:

```
(0,0): extremo superior izquierdo de la ventana madre
-----□ Coord. X
|
|
|
V
```

Ahora sí, el significado de los parámetros de la orden "write" son:

**-1r parámetro (de tipo entero):** Indica un código correspondiente a la fuente de letra que se va a utilizar para el texto. Si no queremos usar ninguna fuente concreta, la fuente predeterminada del sistema se representa por el código 0.

**-2º parámetro (de tipo entero):** coordenada horizontal del texto en píxeles

**-3r parámetro (de tipo entero):** coordenada vertical del texto en píxeles

**-4º parámetro (de tipo entero):** código correspondiente al tipo de alineación.

Indica cómo se han de interpretar las coordenadas del 2º y 3r parámetro. Por ejemplo, si este 4º parámetro vale 0, la coordenada escrita en el 2º y 3r parámetro corresponderá al extremo superior izquierdo del texto; si este 4º parámetro vale 1, la coordenada corresponderá al punto centrado horizontalmente en el texto de su límite superior; si este 4º parámetro vale 2, la coordenada corresponderá al extremo superior derecho; si vale 3, corresponderá al punto centrado verticalmente del extremo izquierdo; si vale 4, el texto estará centrado horizontalmente y verticalmente alrededor de ese punto; si vale 5, corresponderá al punto centrado verticalmente del extremo derecho; si vale 6, la coordenada corresponderá al extremo inferior izquierdo, si vale 7 el punto estará centrado horizontalmente en el extremo inferior; y si finalmente vale 8, la coordenada corresponderá al extremo inferior derecho. Lo más rápido es jugar con la función cambiando el valor de este parámetro y ver los resultados.

De forma alternativa, en vez de asignar a este parámetro el valor numérico directamente, se pueden utilizar palabras concretas que significan lo mismo y son más entendibles a primera vista. Su tabla de equivalencias es la siguiente:

Valor real del 4º parámetro	Cadena posible a utilizar como alternativa
0	ALIGN_TOP_LEFT
1	ALIGN_TOP
2	ALIGN_TOP_RIGHT
3	ALIGN_CENTER_LEFT
4	ALIGN_CENTER
5	ALIGN_CENTER_RIGHT
6	ALIGN_BOTTOM_LEFT
7	ALIGN_BOTTOM
8	ALIGN_BOTTOM_RIGHT

**-5º parámetro (de tipo cadena de caracteres –string-):** el texto a escribir (entre comillas).

Queda claro pues que la línea `write(0,mivar1,100,1,"¡Hola mundo!");` lo que hace, primero de todo, es escribir el texto "¡Hola mundo!" –vaya descubrimiento!-, con la fuente predeterminada del sistema (todavía no sabemos manejar con Fénix los diferentes tipos de fuente disponibles), y con la alineación de tipo 1, que quiere decir que la coordenada horizontal especificada en `write` estará en el centro del texto y la coordenada vertical será el "techo" justo debajo del cual se escribirá el mensaje.

La coordenada vertical es fácil de ver: vale 100 píxeles. Si valiera más veríamos como el texto aparece más abajo en la ventana, y si vale menos, pues saldría más arriba; con lo dicho tendrías que deducir cómo saldría el texto con el tercer parámetro igual a 0.

Pero la coordenada horizontal es un poco más difícil: vale `mivar1`. Y claro, `mivar1` vale un

número, ya lo sabemos, pero ¿cuál?

Déjame que acabe de explicar la última orden, y te explico cómo podemos averiguar el valor del 2º parámetro del *write*. (De hecho, ya deberías de ser capaz de adivinarlo con lo explicado).

### **Orden “Frame”:**

Este comando es muy especial y FUNDAMENTAL: en TODOS los programas Fénix aparece, y fíjate que es de los pocos que no llevan paréntesis alguno.

Antes de nada, prueba de ejecutar el mismo programa sin esta orden (bórrala del código fuente, o mejor, coméntala detrás de las //: así el compilador no la leerá pero bastará retirar otra vez las // para tenerla otra vez disponible). Verás que el programa no parece ejecutarse: no pasa nada. Vaya.

Imagínate que el comando Frame es como si fuera una puerta, un agujero o algo parecido que permite visualizar por pantalla todo lo que el programa ha estado ejecutando hasta el momento durante un instante minúsculo. Si no hay Frame, por mucho que el programa haga, no se verá nada porque esa puerta, ese agujero estará cerrado. Cada vez que se lea el comando Frame, ¡zas!, se abre la puerta, se visualiza todo lo que en todas las líneas anteriores (desde el Frame anterior) ha estado haciendo el programa y se vuelve a cerrar la puerta, así hasta el siguiente Frame.

De esta manera, un programa Fénix se basa en trozos de código que se van ejecutando pero que realmente no se ven sus resultados en pantalla hasta que llega un Frame. A partir de él, si hay más líneas de código se vuelven a ejecutar, pero sin llegar a visualizarse hasta que no se encuentre con el siguiente Frame. Y así.

Es por eso, y esto es muy importante que lo recuerdes, TODOS los programas en Fenix contienen un bucle principal donde en su interior aparece un Frame al final. Lo que hay en el bucle es el programa propiamente dicho que se va ejecutando hasta que se llega al Frame, que visualiza el resultado, y luego se va a la siguiente iteración, donde el programa vuelve a ejecutarse y se vuelve a llegar al Frame, que vuelve a visualizar los resultados –que serán diferentes de la primera vez dependiendo del valor que vayan cogiendo las distintas variables, etc-, y se vuelve a ir a la siguiente iteración, y así, y así se consigue un programa cuya ejecución será visible durante un tiempo: el tiempo que se tarde en salir de ese bucle principal (bien porque es un bucle finito por una condición –como el while del ejemplo-, o bien porque se sale forzosamente de un bucle infinito).

Ésta es una explicación más o menos de estar por casa: posteriormente volveremos a explicar más profundamente el sentido de este comando (cuando hablemos de los procesos), pero por ahora esto es lo que debes entender.

De hecho, si has entendido bien el funcionamiento de Frame, podrás entonces contestar por qué hemos puesto primero `delete_text` cuando no había ningún texto que borrar y luego hemos puesto el `write`. Acabo de decir que el Frame se encarga de poner en pantalla el resultado de todas las operaciones que ha venido efectuando el programa hasta ese momento (lo que se dice el “estado” del programa). Si ponemos el `write` primero, y luego el `delete_text`, primero escribiremos algo y luego lo borraremos, con lo que cuando se llegue al Frame en ese momento no habrá ningún texto que visualizar en pantalla y se verá una pantalla negra y ya está. Pruébalo. Podrás comprender pues la importancia de saber colocar en el lugar apropiado de tu código el Frame: será ese lugar donde estés seguro de que quieres mostrar el estado de tu programa; generalmente el Frame es la última línea del bucle principal del programa.

## **Funcionamiento global del programa:**

Ahora que hemos acabado de explicar las líneas una a una, podríamos ver cuál es el funcionamiento global del programa, ¿no? Primero se le da el valor de 10 a `mivar1`. Luego se entra en lo que es el bucle principal –ineludible en Fenix- del programa. Es un `while` que se ejecutará siempre que `mivar1` sea menor que 320. Evidentemente, cuando se llega aquí por primera vez esto es cierto porque acabamos de darle el valor de 10 a `mivar1`; por lo tanto, entramos en el bucle. Borrarnos todos los textos que hubiera en pantalla, aumentamos el valor de `mivar1` en 2 –con lo que en esta primera iteración ahora `mivar1` vale 12- y escribimos un texto en unas coordenadas por `(mivar1,100)`, que en esta primera iteración serán `(12,100)`. Finalmente llegamos a `Frame`, que lo que hará es imprimir en pantalla el resultado de nuestro programa, que no deja de ser poner un texto en las coordenadas `(12,100)`. Segunda iteración: borramos todos los textos que hubiera en pantalla aumentamos `mivar1` en 2 –ahora valdrá 14-, y escribimos el mismo texto en –ahora- `(14,100)`, con lo que al llegar al `Frame`, ahora en la segunda iteración el texto aparece un poco más a la derecha que la primera vez. Tercera iteración: mismo proceso pero ahora se escribe el texto en las coordenaas `(16,100)`. Y así. ¿Y así hasta cuando? Pues hasta que `mivar1`, que define la coordenada horizontal del texto, valga 320.

¿Y por qué es 320? Fíjate que da la casualidad que “`mivar1`” vale 320 justo cuando el texto (o mejor dicho, según el 4º parámetro del `write`, el punto horizontal central del texto) ha llegado al otro extremo de la ventana. Por tanto, podemos concluir que el extremo derecho de la ventana está en `x=320`. De hecho, date cuenta que nosotros en ningún momento hemos establecido el tamaño de la ventana de nuestro juego; esto se puede hacer, pero si no se hace, por defecto Fénix crea una ventana de anchura 320 píxels y de altura 240 píxels, por lo que sabremos siempre que la coordenada horizontal de los puntos del extremo derecho de la ventana valdrá 320. ¿Qué pasaría si en el `while`, en vez de poner 320 ponemos por ejemplo 3200? Pues que el texto se movería igual hasta llegar al extremo derecho de la ventana (`x=320`), pero como el bucle del `while` todavía no habría acabado, aunque el texto habría desaparecido por la derecha y ya no sería visible, el programa continuaría ejecutándose –nosotros veríamos sólo una ventana negra- hasta que por fin, después de un rato, “`mivar1`” llegara a valer 3200, momento en el que el programa acabaría. ¿Y si hiciéramos al revés, en vez de poner en el `while` 320 ponemos 32? Pues ocurriría lo contrario: el programa acabaría antes de que el texto pudiera llegar al extremo derecho, de hecho acabaría enseguida porque el texto pronto llega a esa coordenada.

También nos podemos preguntar otras cosas. ¿Qué pasaría si, en el programa original, asignáramos un valor inicial de 100 a “`mivar1`”? Pues, evidentemente, que al iniciar el programa el texto ya aparecería en la ventana en una posición más centrada horizontalmente, y tendría menos recorrido para llegar hasta la coordenada `x=320`, con lo que el programa también duraría menos. ¿Y si inicializáramos “`mivar1`” a 1000? Pues que no veríamos nada de nada porque nada más empezar la condición del `while` resulta falsa, y como tal, el `while` es saltado y se acaba el programa. También podemos probar de inicializar “`mivar1`” a 1000 y además cambiar la condición del `while` para que “`mivar1`” < 3200. En este caso, el programa se ejecutaría un rato, hasta que “`mivar1`” llegara a valer 3200, pero no veríamos nada, porque el valor inicial de “`mivar1`” ya de entrada es muy superior a la coordenada derecha de la ventana del juego, y por tanto, el texto estará fuera de la región visible.

Otro efecto curioso es el de cambiar el incremento en el valor de la variable “`mivar1`”. Así, en vez de hacer `mivar1=mivar1+2`; podríamos haber escrito `mivar1=mivar1+10`; por ejemplo. En este caso, veríamos como el texto se mueve mucho más rápido, ya que en cada iteración, el texto se desplaza una cantidad mayor de píxels y por tanto recorre la ventana en menos iteraciones.

Podemos jugar con el “`write`” también. Si ponemos un valor constante en su segundo parámetro, es evidente que el texto no se moverá, pero aún así, veremos que el programa tarda un rato en cerrarse. Esto es, evidentemente, porque aunque no hayamos utilizado “`mivar1`” para hacer el scroll del texto, el bucle del

while sigue funcionando, y cuando se llega a la condición falsa se sale igual: estaremos borrando y escribiendo en cada iteración una y otra vez el mismo texto en la misma posición hasta que la variable “mivar1” valga 320. También podríamos probar de poner “mivar1” en el tercer parámetro del “write”; de esta manera, haríamos un scroll vertical del texto. En este caso, el extremo inferior de la pantalla es 240, así que si no queremos esperar un rato con la pantalla negra después de que el texto haya desaparecido por debajo de la ventana, tendríamos que modificar la condición del while consecuentemente. Y podríamos incluso hacer un scroll en diagonal, simplemente añadiendo otra variable y poniendo las dos variables en el 2º y 3º parámetro del “write”. Algo así como esto:

```
Program MiPrimerPrograma;
```

```
Private
```

```
    int mivar1;  
    int mivar2;
```

```
End
```

```
Begin
```

```
    mivar1=10;  
    mivar2=10;  
    while(mivar1<320)  
        delete_text(0);  
        mivar1=mivar1+2;  
        mivar2=mivar2+2;  
        write(0,mivar1,mivar2,1,";Hola mundo!");  
        frame;
```

```
    end
```

```
end
```

Fíjate que lo único que hemos hecho es declarar, inicializar e incrementar correspondientemente una nueva variable que representará la coordenada vertical donde se escribirá el texto. La condición del while puede continuar siendo la misma o depender de la nueva variable, según nos interese.

Deberías intuir ya con lo explicado hasta ahora cómo se podría hacer, por ejemplo, un scroll vertical que fuera de abajo a arriba. El código sería éste:

```
Program MiPrimerPrograma;
```

```
Private
```

```
    int mivar1;
```

```
End
```

```
Begin
```

```
    mivar1=240;  
    while(mivar1>10)  
        delete_text(0);  
        mivar1=mivar1-2;  
        write(0,100,mivar1,1,";Hola mundo!");  
        frame;
```

```
    end
```

```
end
```

Lo que he hecho ha sido: primero, darle un valor inicial a “mivar1” de 240; segundo, cambiar la condición del while poniendo que el bucle se acabará cuando “mivar1” deje de ser mayor que 10; y tercero, el incremento de la “mivar1” lo hago negativo. Con estos tres cambios, lo que consigo es que inicialmente el texto se escriba en el extremo inferior de la ventana, y que en cada iteración se vaya escribiendo 2 píxeles

más arriba –recordad que la coordenada vertical decrece si vamos subiendo por la ventana- y este proceso se irá ejecutando hasta que lleguemos al momento donde “mivar1” vale menos de 10, es decir, que el texto se ha escrito casi en el extremo superior de la ventana.

Otro efecto interesante es el de comentar la línea `delete_text`. Al hacer esto, estamos evitando que en cada iteración el texto acabado de escribir se borre, con lo que estaremos escribiendo cada vez un nuevo texto sobre el anterior, desplazado unos píxeles a la derecha. El efecto de superposición que se crea es un línea continua curiosa.

Y otra cosa que puedes probar es (con el `delete_text` descomentado) cambiar el valor de su parámetro -o- por otro número cualquiera. La explicación de lo ocurrido lo dejaremos para más adelante.

### **Notas finales:**

Fíjate en las tabulaciones: he tabulado todo el código entre el BEGIN/END y luego, otra vez, he vuelto a tabular el código entre el WHILE/END. Esto, ya lo comenté anteriormente, es por claridad. Tabulando se tiene una rápida visión de dónde empiezan y dónde acaban los distintos trozos de código que componen nuestro programa, y podemos localizar rápidamente zonas concretas. Tabulando sabemos que un conjunto de sentencias pertenecen a un mismo bucle, condicional, programa, etc, aportando así una gran claridad en la lectura del código. Y sobretodo, ayudando en gran medida para corregir posibles errores de escritura. Es muy recomendable acostumbrarse a tabular los distintos tipos de bucles, condiciones, procesos, etc, para una buena salud mental y ocular. Ya iréis viendo ejemplos de programas donde se pone en práctica esta manera de organizarse (por otra parte, estándar en el mundo de la programación).

Otra cosa que posiblemente no hayas visto es cuándo en general se escriben los puntos y coma y cuándo no. En general, si no digo explícitamente lo contrario, los puntos y coma hay que escribirlos SIEMPRE al final de una sentencia (como puede ser un comando, una asignación, o cualquier cosa). La única excepción –que de hecho no es tal porque no son sentencias- es cuando abrimos y cerramos un bloque de código, como puede ser un bucle (WHILE/END es un ejemplo), un condicional (IF/END...) o el bloque principal del programa (BEGIN/END). En general, todas las parejas de palabras que tengan la palabra END no llevan punto y coma.

Y un último comentario. Te habrás fijado que, independientemente del trozo de bloque que cerremos (bucle, condicional, begin principal...), todos acaban con la misma palabra: END. Esto a la larga puede ser un poco lioso porque cuando tienes cuatro o cinco ENDS seguidos no sabes cuál corresponde a qué bloque. Por eso siempre es muy recomendable añadir un comentario al lado del END diciendo a qué bloque corresponde, de esta manera:

```
Program MiPrimerPrograma;
```

```
Private
```

```
    int mivar1;
```

```
End
```

```
Begin
```

```
    mivar1=10;
```

```
    while(mivar1<320)
```

```
        delete_text(0);
```

```
        mivar1=mivar1+2;
```

```
        write(0,mivar1,100,1,"¡Hola mundo!");
```

```
        frame;
```

```
    end //End del while
```

```
end //End del begin
```

Así no tendrás tantas posibilidades de confusión.

## Explicación paso a paso de “Mi segundo programa en Fénix”

Recordemos el código:

```
Program Misegundoprograma;
```

```
private
```

```
    int mivar1;
```

```
end
```

```
begin
```

```
    loop
```

```
        delete_text(0);
```

```
        mivar1=rand(1,10);
```

```
        if (mivar1<=5)
```

```
            write(0,200,100,2,"Menor o igual que 5");
```

```
        else
```

```
            write(0,200,100,2,"Mayor que 5");
```

```
        end
```

```
        frame;
```

```
    end
```

```
end
```

De aquí, lo único novedoso es lo que hay entre el BEGIN/END, o sea:

```
loop
```

```
    delete_text(0);
```

```
    mivar1=rand(1,10);
```

```
    if (mivar1<=5)
```

```
        write(0,200,100,2,"Menor o igual que 5");
```

```
    else
```

```
        write(0,200,100,2,"Mayor que 5");
```

```
    end
```

```
    frame;
```

```
end
```

Podemos ver que esto es un bloque definido en el inicio por la palabra LOOP y terminado al final por –siempre- la palabra END. Como bloque que es, su interior ha sido tabulado para facilitar la lectura

### **Bloque “Loop/End”. Sentencias BREAK y CONTINUE:**

La sentencia LOOP (bucle) es una sentencia que implementa un bucle infinito, es decir, que repite indefinidamente un grupo de sentencias. Para implementar este bucle vemos que se debe comenzar con la palabra reservada LOOP, seguida de las sentencias que se quieren repetir continuamente y la palabra

reservada END al final. Cuando un programa se encuentra una sentencia LOOP...END se ejecutarán a partir de entonces, una y otra vez, todas las sentencias interiores a dicho bucle. Las sentencias interiores a un bucle LOOP pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles LOOP.

Aprovecho aquí para comentar un par de órdenes muy ligadas al bucle LOOP y a todos los bucles en general, aunque no aparezcan en el ejemplo

### **Sentencia BREAK;:**

Si en algún momento quisiéramos finalizar un bucle LOOP, se puede utilizar la sentencia BREAK; que, al ejecutarse dentro de un bucle de este tipo, forzará al programa a seguir a continuación del END. Es decir, una sentencia BREAK; dentro de un bucle lo finalizará de forma inmediata, continuando el programa por la sentencia siguiente a dicho bucle. Esta sentencia puede ponerse, además de dentro del bucle LOOP, dentro de los bucles WHILE, REPEAT, FOR o FROM.(ya los veremos). Por ejemplo, una sentencia BREAK dentro de un bucle WHILE funcionará de la misma manera: finalizará el bucle de forma inmediata, continuando el programa a partir de la sentencia siguiente a dicho bucle.

En caso de haber varios bucles anidados (unos dentro de otros) la sentencia BREAK saldrá únicamente del bucle más interior de ellos.

Y la sentencia BREAK no es válida para finalizar secuencias IF, ni SWITCH ni sentencias CLONE (ya las veremos)

### **Sentencia CONTINUE;:**

La sentencia CONTINUE;, dentro de un bucle, finalizará la iteración actual y comenzará la siguiente (el programa continuará ejecutándose tras la palabra LOOP). Es decir, una sentencia CONTINUE; dentro de un bucle forzará al programa a finalizar la iteración actual del mismo y comenzar la evaluación de la siguiente. Aparte del bucle LOOP, puede usarse esta sentencia en los bucles WHILE, REPEAT, FOR o FROM (ya los veremos). Por ejemplo, una sentencia CONTINUE dentro de un bucle WHILE forzará al programa a comprobar la condición inicial inmediatamente, y si ésta es cierta, volverá a ejecutar las sentencias interiores desde el principio (tras el WHILE), y si la condición resultara falsa, la sentencia CONTINUE finalizará el bucle.

En caso de haber varios bucles anidados (unos dentro de otros) la sentencia CONTINUE tendrá efecto únicamente en el bucle más interior de ellos. Y la sentencia CONTINUE no es válida para finalizar secuencias IF, ni SWITCH ni sentencias CLONE (ya las veremos).

Vamos a ver estas sentencias con ejemplos. Si tenemos este código:

```
Program ejemplo;
Private
    Int mivar1=0;

End
Begin
    Loop
        mivar1=mivar1+1;
        If (mivar1==30)
            Break;
        End
    write_var(0,100,100,4,mivar1);
```

```

                                Frame;
                                End
                                End

```

lo que tendrá que ocurrir es que aparecerá escrito en pantalla un número que irá aumentando en una unidad desde el 0 hasta el número 29. ¿Por qué? Porque inicializamos la variable `mivar1` con el valor 0, y en cada iteración le incrementamos en una unidad su valor y lo escribimos. Esto sería un proceso infinito si no fuera porque en cada iteración se comprueba una condición –el “if” lo veremos enseguida, pero aquí es fácil ver cómo funciona– que dice que si “`mivar1`” en ese momento vale exactamente 30, entonces se haga un `break`. Y hacer un `break` implica salir automáticamente del interior del bucle donde se está en este momento. Como en este ejemplo, al salir del bucle ya no hay nada más, se acaba el programa. Hay que puntualizar que sólo se verá hasta el número 29 porque la condición de comprobación está antes del “write”, y por tanto, cuando “`mivar1`” sea igual a 30, se saldrá del bucle sin pasar por el “write”, y por tanto, el valor 30 no se escribirá. Otra cosa sería –es fácil verlo–, si hubiéramos escrito el mismo programa pero cambiando el orden:

```

Program ejemplo;
Private
    Int mivar1=0;
Begin
    Loop
        mivar1=mivar1+1;
        write_var(0,100,100,4,mivar1);
        If (mivar1==30)
            Break;
        End
    End
End

```

Un par de detalles: fijarse que para escribir el valor de una variable no hemos utilizado el comando “write” sino otro muy parecido que es “write\_var”. Este comando funciona igual que “write”, pero este último sólo sirve para escribir textos concretos y fijos establecidos como 5º parámetro entre comillas, y en cambio “write\_var” sirve normalmente para escribir el valor que tiene una variable, pasada también como 5º parámetro (sin comillas).

Más correctamente, la gran diferencia entre “write” y “write\_var” es que el valor que se escriba con “write” quedará permanentemente inalterado en pantalla una vez escrito, y lo que se escriba con “write\_var” (valores de variables), en cada frame se comprobará automáticamente si ha cambiado; en ese caso ese dato se refrescará en pantalla mostrando su nuevo valor. Por tanto, la función “write\_var” es la más adecuada para hacer un seguimiento “en tiempo real” frame a frame de la evolución del valor de una variable.

Otra cosa a tener en cuenta es, como siempre, recordar la existencia de la orden “frame;” al final del bucle principal del programa, ya que si no está no se ve nada, tal como hemos visto.

Ahora cuando veamos el funcionamiento del “if” lo veremos, pero fijate que en la condición de igualdad `if(mivar1==30)` he escrito dos iguales seguidos (==) y no uno. Esto es porque ya he comentado que el signo = a secas es un operador de asignación, pero no de igualdad. Para escribir que a es igual a b, hemos de escribir `a==b`; si escribiéramos `a=b` estaríamos diciendo que el valor de b pasa a ser el valor de a, y por tanto, una sentencia como `if (a=b)` siempre sería verdad, porque siempre sería verdad si no hay errores que le paso el valor de b a a.

Ahora pongo otro ejemplo para ver el funcionamiento del CONTINUE:

```
Program ejemplo;
Private
    Int mivar1=0;
Begin
    Loop
        mivar1=mivar1+1;
        If (mivar1==30)
            continue;
        End
        write_var(0,100,100,4,mivar1);
        Frame;
    End
End
```

Es el mismo código que antes, pero cambiando BREAK por CONTINUE. Lo que ocurre, en cambio, es que se genera un bucle infinito con el número que aparece en pantalla incrementando su valor en una unidad hasta el infinito (de hecho en teoría hasta su valor máximo que puede albergar según el tipo de variable que lo contiene, pero ocurre un error antes, pero no nos incumbe ahora). Entonces, ¿qué hace el CONTINUE? El programa se ejecuta muy rápido para verlo, pero lo que ocurre es que el número 30 no aparece: se visualiza hasta el 29 y después se ve el 31 y siguientes. Lo que ocurre es que cuando se realiza la comprobación del if y se ve que en esa iteración “mivar1” vale 30, se salta todo lo que queda de iteración y se vuelve otra vez para arriba, a la palabra LOOP, y se empieza la siguiente iteración. Es decir, hacer un CONTINUE es como poner un END en medio de la iteración y dejar inutilizadas las sentencias que hay dentro del bucle posteriores a él. Así pues, cuando “mivar1” vale 30, no se hace el write\_var y se vuelve a LOOP, donde “mivar1” pasa a valer 31.

En este caso, es indiferente poner el if antes o después del write\_var, porque en ambos casos la sentencia Frame es la que se salta si se ejecuta el CONTINUE. Si se salta la sentencia frame, nada se muestra por pantalla.

### **“mivar1=rand(1,10);”.Valores de retorno de las funciones:**

Esta línea es una simple asignación de un valor a una variable, pero en vez de ser un valor concreto como podría ser *mivar1=10;*, por ejemplo, lo que se hace es utilizar el comando rand.

Ves que el comando rand que es una orden con dos parámetros. Lo que no hemos comentado hasta ahora es que los comandos, a parte de recibir parámetros de entrada –si lo hacen-, y a parte de hacer la tarea que tienen que hacer, también devuelven valores de salida, generalmente INT. Estos valores enteros que devuelven pueden servir para varias cosas dependiendo del comando; el significado de ese valor devuelto depende: algunos valores son de control, avisando al ordenador que el comando se ha ejecutado bien o mal, otros sirven para generar números específicos, (como rand), etc.

De hecho, no lo hemos visto pero el comando write por ejemplo devuelve un valor entero cada vez que se ejecuta. Si nos interesa recoger ese valor, se lo asignamos a una variable y lo utilizamos, y si no, pues no hacemos nada. Ahora no nos interesa saber cual es el significado de este número que devuelve el write, pero para que lo veas, ejecuta esto:

```
Program ejemplo;
Private
    Int mivar1;
```

```

End
Begin
  Loop
    mivar1=write(0,100,100,4,"Hola");
    write_var(0,150,150,4,mivar1);
    Frame;
  End
End

```

Verás que se escribe la palabra “Hola” y que además aparece un número que va aumentando, hasta que ocurre un error –el mismo que con el ejemplo del CONTINUE-. Este número ¿qué es?. Podrás intuir que es el valor que en cada iteración devuelve el comando write. Efectivamente: en este ejemplo, cada vez que se llama a write, éste realiza su conocida tarea de imprimir por pantalla la palabra “Hola”, pero además, recogemos el valor entero que devuelve –cosa que antes no hacíamos porque no nos interesaba, aunque write lo devuelve siempre- en una variable, para seguidamente visualizar dicho valor también por pantalla.

En el caso del write, el valor devuelto representa un número identificador del texto impreso en pantalla; es como un DNI de la frase escrita, un número al que podremos hacer referencia en nuestro programa posteriormente y mediante el cual podremos manipular ese texto cuando lo deseemos. Es decir, que si escribimos un texto con write y recogemos lo que devuelve, allí tendremos el número que identificará ese texto, y si luego en nuestro programa queremos cambiarlo o hacer lo que sea con él, simplemente tendremos que recurrir a ese número para saber de qué texto estamos hablando y hacer lo que sea con él (ya veremos ejemplos, tranquilo).

Respecto al error que aparece, puede que no entiendas lo que dice: “Demasiados textos en pantalla”. ¡Cómo, si sólo escribo “Hola” y ya está!. Pues no, fíjate bien. Si acabo de decir que lo que devuelve el write es un identificador del texto escrito, y ese identificador vemos que sube como la espuma, eso quiere decir que estamos escribiendo muchos textos diferentes, cada uno de los cuales, como es lógico, tiene un identificador diferente. Es decir, que si vemos que el número que aparece sube hasta el 500, es porque hemos escrito 500 textos diferentes, cada uno de los cuales con un identificador diferente. ¿Y esto cómo es? Pues porque en cada iteración ejecutamos la orden write de nuevo, por lo que en cada iteración estamos escribiendo un nuevo texto, que se superpone al anterior. Es por esta razón que al final aparece este error, porque hemos escrito tantos textos en pantalla que la memoria del ordenador se desborda. Entonces, ¿cómo se puede escribir un texto fijo que esté presente durante toda la ejecución del programa sin que dé el error? Prueba esto:

```

Program ejemplo;
Private
  Int mivar1;
Begin
  mivar1=write(0,100,100,4,"Hola");
  write_var(0,150,150,4,mivar1);
  Loop
    Frame;
  End
end

```

Deberías entender lo que ocurre. Escribimos una sola vez “Hola”, vemos como el identificador de ese texto, efectivamente es el número 1 (siempre se crean los identificadores a partir del 1 para el primer texto escrito y a partir de allí para arriba) y es sólo después cuando nos metemos en el bucle principal del programa con el frame, para que se pueda visualizar la frase fija todas las veces que haga falta.

Otra solución al mismo problema sería:

```

Program ejemplo;
Private
    Int mivar1;
Begin
    Loop
        delete_text(0);
        mivar1=write(0,100,100,4,"Hola");
        write_var(0,150,150,4,mivar1);
        Frame;
    End
end

```

Hemos hecho el mismo truco que en nuestro primer programa en Fénix.

Ahora es el momento donde se puede entender mejor cuál es el significado del parámetro del `delete_text`. Ya sabemos que si ponemos un 0 quiere decir que borrará todos los textos que haya en pantalla –escritos con el comando `write`-. Prueba de ejecutar el mismo programa anterior pero poniendo un 1 como valor del `delete_text`. Y luego poniendo un 2, etc. Verás que se vuelven a escribir muchos textos otra vez y se vuelve a generar el error que hemos visto. ¿Por qué? Porque si no ponemos un 0 en `delete_text` y ponemos cualquier otro número, ese número representará el identificador de un texto concreto, y sólo borrará ese texto concreto. Es decir, que si con `write` hemos creado un texto con identificador 1, `delete_text(1)` sólo borrará ese texto y ninguno más. Por eso, con este código, por ejemplo:

```

Program ejemplo;
Private
    Int mivar1;
Begin
    Loop
        delete_text(1);
        mivar1=write(0,100,100,4,"Hola");
        write_var(0,150,150,4,mivar1);
        Frame;
    End
end

```

lo que ocurre es que `delete_text` sólo borrará un texto (el que tenga identificador 1) de entre todos los textos que se escriban –que van a ser tantos como iteraciones haya: infinitos, y cada uno con un identificador diferente-. Por tanto, el error volverá a aparecer porque sólo hemos borrado el primer texto que se escribió, pero a partir del segundo (con identificador 2) para adelante no se ha borrado ninguno.

Bueno. Una vez explicado todo este rollo sobre los valores de retorno de las funciones (o comandos) de Fénix, vamos al ejemplo que teníamos entre manos. Teníamos la línea:

```
mivar1=rand(1,10);
```

Claramente, “mivar1” va a valer un número que será lo que devuelva el comando `rand`. El comando `rand` precisamente sirve para devolver un número, cuya característica más importante es que es pseudo-aleatorio.

Ves que tiene dos parámetros enteros: el primero es el límite inferior y el segundo es el límite superior –ambos incluidos- de los valores que puede tomar el número que `rand` va a generar. Cada llamada a esta función generará un número diferente, siempre dentro del rango especificado. Por lo tanto, cada vez que se ejecute la línea anterior, “mivar1” va a valer un número diferente, entre 1 y 10. Y ya está.

## **Bloque “If/Else/End”. Condiciones posibles:**

La sentencia IF no es ningún bucle. Como dice su nombre, simplemente sirve para ejecutar un bloque de sentencias solamente si se cumple una condición determinada. Y opcionalmente, también es posible incluir una sección ELSE con otro bloque de sentencias que se ejecutarían en caso de que NO se cumpla esa condición, teniendo pues el programa respuesta para las dos posibilidades. Tanto si se ejecutan las sentencias de la sección IF como si se ejecutan las sentencias de la sección ELSE, cuando se llega a la última línea de esa sección, se salta a la línea inmediatamente posterior al END para continuar el programa.

Es decir, en general, la sintaxis del bloque IF/ELSE será:

```
IF ( condición)
    Sentencias -una o más-que se ejecutan si la condición es verdad;
ELSE
    Sentencias –una o más- que se ejecutan si la condición es falsa;
END
```

Y cuando se hayan ejecutado todas las sentencias de una u otra sección(IF o ELSE), se continúa en la línea inmediatamente posterior al END.

He dicho que el ELSE es opcional. Si no ponemos ELSE, el IF quedaría:

```
IF ( condición)
    Sentencias -una o más-que se ejecutan si la condición es verdad;
END
```

y, en este caso, si la condición fuera falsa, el if no se ejecuta y directamente se pasa a ejecutar la línea inmediatamente posterior al END (el programa no hace nada en particular cuando la condición es falsa).

Es posible anidar sentencias IF sin ningún límite, es decir, se pueden poner mas sentencias IF dentro de la parte que se ejecuta cuando se cumple la condición (parte IF) o dentro de la que se ejecuta cuando la condición no se cumple (parte ELSE).

¿Y qué tipo de condiciones podemos escribir entre los paréntesis del IF? Cuando vimos el WHILE en el primer programa que hemos hecho, vimos los operadores para las condiciones de menor (<) o mayor que (>) un número, y el de la condición de igualdad (==), cuyo operador, ¡ojo!, recordad que son dos signos iguales, ya que un solo signo representa el operador de asignación. Pero hay muchas más operadores para diferentes condiciones:

==	Comparación de igualdad
<>	Comparación de diferencia (también sirve !=)
>	Comparación de mayor que
>=	Comparación de mayor o igual que (también sirve =>)
<	Comparación de menor que
<=	Comparación de menor o igual que (también sirve =<)

Además, también se pueden utilizar los operadores llamados lógicos, usados para encadenar dos o más comprobaciones dentro de una condición. Los operadores lógicos son:

<b>OR</b>	Comprueba que, al menos, una de dos expresiones sea cierta. (también sirve    )
<b>AND</b>	Comprueba que las dos expresiones sean ciertas. (también sirve &&)
<b>XOR</b>	Comprueba que sólo una de las dos expresiones sea cierta (también sirve ^^ )
<b>NOT</b>	Comprueba que no se cumpla la siguiente condición (también sirve ! )

Y los paréntesis (), para establecer el orden de comparación de varias expresiones, pues siempre se comparan primero las expresiones situada dentro de los paréntesis, o en el más interior, si hay varios grupos de paréntesis anidados.

Ejemplos de condiciones podrían ser:

<code>mivar1==100 AND mivar2&gt;10</code>	Esta condición comprueba que “mivar1” sea exactamente 100 Y QUE ADEMÁS, “mivar2” sea mayor que 10
<code>mivar1&lt;&gt;0 OR (mivar2&gt;=100 and mivar2&lt;=200)</code>	Esta condición primero comprueba lo que hay dentro del paréntesis, o sea, que “mivar2” sea mayor o igual que 100 Y QUE ADEMÁS “mivar2” sea menor o igual que 200. Eso, O A VER SI “mivar1” es diferente de 0

El anidamiento de condiciones puede ser todo lo complejo que uno quiera, pero conviene asegurarse de que realmente se está comprobando lo que uno quiere, cosa que a veces, con expresiones muy largas, es difícil.

No está de más finalmente tener presente un par de reglas matemáticas que cumplirán todas las posibles expresiones que podamos escribir:

<code>!(condicion1 OR condicion2)</code>	Equivale a	<code>!condicion1 AND !condicion2</code>
<code>!(condicion1 AND condicion2)</code>	Equivale a	<code>!condicion1 OR !condicion2</code>

**Un dato importante:** a la hora de realizar comprobaciones de condiciones en cláusulas IF, por ejemplo, el resultado de la comprobación está claro que sólo puede ser o VERDADERO o FALSO. Pero estos valores lógicos un ordenador, y Fénix en concreto, no los entiende, porque sólo entienden de ceros y unos. Así que Fénix entiende internamente que si una condición es falsa en realidad él entiende que vale 0, y si es verdadera, en realidad él entiende que vale diferente de 0 (1,-1 o lo que sea).

Esta asociación (FALSO=0 y VERDADERO!=0) es común a otros muchos lenguajes de programación (C,Java,Php,Python,etc). No obstante, por comodidad, en todos estos lenguajes de programación -incluido Fénix- existe la posibilidad de utilizar dos constantes lógicas: "TRUE" y "FALSE", que equivalen a sus valores numéricos, en vez de éstos.

En el caso concreto del programa que nos ocupa, teníamos, recordemos, el siguiente código:

```
Program Misegundoprograma;
private
    int mivar1;
```

```

end
begin
    loop
        delete_text(0);
        mivar1=rand(1,10);
        if (mivar1<=5)
            write(0,200,100,2,"Menor o igual que 5");
        else
            write(0,200,100,2,"Mayor que 5");
        end
        frame;
    end
end

```

Está claro que lo que hace el programa es, una vez que la función rand ha devuelto un número aleatorio entre 1 y 10 –ambos incluidos- y lo ha asignado a la variable “mivar1”, comprueba el valor de dicha variable con un if. Si el recién valor de “mivar1” es menor o igual que 5, saldrá un texto en pantalla, en una coordenada concreta que dirá “Menor o igual que 5”. En cambio, si “mivar1” es mayor que 5, saldrá otro texto, en las mismas coordenadas, que diga “Mayor que 5”.  
Y ya está.

### **Funcionamiento global del programa:**

El programa básicamente, pues, hace lo siguiente. Nada más comenzar se introduce en un bucle infinito (LOOP), por lo que, ya que no vemos ninguna sentencia BREAK, sabemos que este programa en principio nunca finalizará su ejecución – a no ser que nosotros le demos al botón de cerrar de la ventana, menos mal que podemos hacer eso-. Nada más entrar en el bucle, borramos todo el texto que pueda haber. Recordad que ponemos primero el delete\_text antes de cualquier write porque si lo hiciéramos al revés, al llegar a la sentencia FRAME; no habría ningún texto que mostrar y no saldría nada en pantalla: hemos de recordar siempre de dejar justo justo antes del FRAME todo lo que queremos mostrar tal como lo queremos mostrar. Seguidamente ejecutamos la función rand, que nos devuelve un entero aleatorio entre 1 y 10 y lo asignamos a una variable declarada previamente como privada. A partir de entonces esa variable tiene un valor que puede ser cualquiera entre 1 y 10. Seguidamente se realiza la comprobación mediante el IF/ELSE del valor de esa variable, y si es mayor o igual que 5 se escribirá un texto en pantalla y si es menor se escribirá otro, pero eso no se hará hasta que se llegue a FRAME. Una vez que salimos de ejecutar la correspondiente línea del bloque IF/ELSE, es cuando aparece precisamente FRAME, porque deseamos poner en pantalla en este momento el resultado del proceso del programa: es decir, la impresión de la frase de la sección IF/ELSE que se haya ejecutado. Llegado este punto, la iteración acaba y comenzamos una nueva iteración: borramos el texto, asignamos a la variable un nuevo valor aleatorio que no tendrá nada que ver con el que tenía antes, y se vuelve a hacer la comprobación. Como los valores que tendrá “mivar1” en las distintas iteraciones son aleatorios, unas veces serán mayores o iguales que 5 y otras veces serán menores que 5, por lo que en algunas iteraciones aparecerá una frase y otras la otra, mostrando así el efecto que vemos por pantalla: la frase cambia rápidamente de una en otra casi sin tiempo para leerla. Y así hasta el infinito.

¿Qué pasaría si quitáramos –comentáramos- el bloque LOOP/END? Pues que la ejecución del programa dura un suspiro, ya que sólo se ejecuta una sólo vez su código –no hay iteraciones que se repitan-, y por lo tanto, sólo se ejecuta una vez el FRAME, y por lo tanto, sólo se muestra en una fracción de segundo lo que sería una única frase, antes de llegar al END del programa. Lógico, ¿no?

## Otros bloques de control de flujo. Switch,for,from,repeat:

En los dos programas anteriores hemos visto diferentes maneras de hacer bucles y condiciones. En concreto hemos visto los bucles WHILE/END y LOOP/END y el condicional IF/ELSE/END. Pero en Fénix hay más bloques que permiten la existencia de iteraciones o condiciones.

En general, a todos estos bloques se les llama bloques de control de flujo, porque controlan dónde tiene que ejecutar el programa la siguiente instrucción, es decir, controlan el devenir del programa y dicen cuál es la siguiente línea a ejecutar.

Vamos a ver ahora los bloques de control de flujo que nos faltan para poder tener ya todas las herramientas necesarias para poder escribir programas flexibles:

### **Bloque “SWITCH/END”:**

```
SWITCH ( variable o expresión )
    CASE valores:
        Sentencias;
    END
    CASE valores:
        Sentencias;
    END
    CASE valores:
        Sentencias;
    END
    ...
    DEFAULT:
        Sentencias;
    END
END
```

Una sentencia SWITCH/END consta en su interior de una serie de secciones CASE/END y, opcionalmente, una sección DEFAULT/END.

El SWITCH/END es como una especie de Mega-IF. Cuando se ejecuta una sentencia SWITCH, primero se comprueba el valor de la variable o expresión que hay entre los paréntesis, y después, si el resultado está dentro del rango de valores contemplados en la primera sección CASE, se ejecutarán las sentencias de la misma y se dará por finalizada la sentencia. En caso de no estar el resultado de la expresión en el primer CASE se pasará a comprobarlo con el segundo CASE, el tercero, etc. Y por último, si existe una sección DEFAULT y el resultado de la expresión no ha coincidido con ninguna de las secciones CASE, entonces se ejecutarán las sentencias de la sección DEFAULT.

Hay que hacer notar que una vez ejecutada una de las secciones CASE de una sentencia SWITCH ya no se ejecutarán más secciones CASE, aunque éstas especifiquen también el resultado de la expresión.

No es necesario ordenar las secciones CASE según sus valores (de menor a mayor, o de mayor a menor), pero sí es imprescindible que la sección DEFAULT (en caso de haberla) sea la última sección. No puede haber más que una sección DEFAULT. Es posible anidar sentencias SWITCH sin ningún límite, es decir, se pueden poner nuevas sentencias SWITCH dentro de una sección CASE (y cualquier otro tipo de sentencia).

En una sección CASE se puede especificar:

- Un valor
- Un rango de valores mínimo..máximo (es importante separar estos valores extremos por dos puntos, no por tres)
- o -Una lista de valores y/o rangos separados por comas.

Un ejemplo:

```

program pepito;
private
    byte mivar1=220;
end
begin
    loop
        delete_text(0);
        switch (mivar1)
            case 0..3:
                write(0,200,100,4,"La variable vale entre 0 y 3, incluidos");
            end
            case 4:
                write(0,200,100,4,"La variable vale 4");
            end
            case 5,7,9:
                write(0,200,100,4,"La variable vale 5 o 7 o 9");
            end
            default:
                write(0,200,100,4,"La variable vale cualquier otro valor diferente de los anteriores");
            end
        end // switch
    frame;
    end //loop
end // begin

```

### **Bloque "FOR/END":**

```

FOR(valor_inicial_contador;condicion_final_bucle;incrment_cont)
    Sentencias;
END

```

La sentencia FOR es una sentencia que implementa un bucle. Se deben especificar, entre paréntesis, tres partes diferentes, separadas por símbolos ; (punto y coma) tras la palabra reservada FOR. Estas tres partes son opcionales (pueden omitirse) y son las siguientes:

-Valor inicial del contador:. En esta parte se suele codificar una sentencia de asignación que fija el valor inicial de la variable que va a utilizarse como contador de iteraciones del bucle. Un ejemplo puede ser la sentencia de asignación  $x=0$ ; que fijaría la variable  $x$  a cero a inicio del bucle (valor para la primera iteración).

-Condición final del bucle:. En esta parte se especifica una condición; justo antes de cada iteración se comprobará que sea cierta para pasar a ejecutar el grupo de sentencias. Si la condición se evalúa como falsa, se finalizará el bucle FOR, continuando el programa tras el END del bucle FOR. Un ejemplo de condición puede ser  $x<10$ ; que permitirá que se ejecute el grupo interior de

sentencias únicamente cuando la variable x sea un número menor que 10.

-Incremento del contador: En la última de las tres partes es donde se indica el incremento de la variable usada como contador por cada iteración del bucle; normalmente, ésto se expresa también con una sentencia de asignación. Por ejemplo, la sentencia  $x=x+1$ ; le sumaría 1 a la variable x tras cada iteración del bucle.

Tras la definición del bucle FOR, con sus tres partes, es donde debe aparecer el grupo de sentencias interiores del bucle que se van a repetir secuencialmente mientras se cumpla la condición de permanencia (parte segunda). Tras este grupo de sentencias la palabra reservada END determinará el final del bucle FOR.

Cuando en un programa llega a una sentencia FOR se ejecuta primero la parte de la inicialización y se comprueba la condición; si ésta es cierta se ejecutará el grupo de sentencias interiores y, después, la parte del incremento, volviéndose seguidamente a comprobar la condición, etc. Si antes de cualquier iteración la condición resulta falsa, finalizará la sentencia FOR inmediatamente.

Como se ha mencionado, las tres partes en la definición del bucle son opcionales; si se omitieran las tres, sería equivalente a un bucle LOOP...END.

Además, en un bucle FOR pueden ponerse varias partes de inicialización, condición o incremento separadas por comas, ejecutándose todas las inicializaciones primero, luego comprobándose todas las condiciones de permanencia (si cualquiera resultara falsa, el bucle finalizaría), las sentencias interiores, y al final, tras cada iteración, todos los incrementos.

Una sentencia BREAK dentro de un bucle FOR lo finalizará de forma inmediata, continuando el programa por la sentencia siguiente a dicho bucle. Una sentencia CONTINUE dentro de un bucle FOR forzará al programa a ejecutar directamente la parte del incremento y, después, realizar la comprobación de permanencia y, si ésta es cierta, volver a ejecutar las sentencias interiores desde el principio. Si la condición resulta falsa, la sentencia CONTINUE finalizará el bucle FOR.

Las sentencias interiores a un bucle FOR pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles FOR.

Un ejemplo:

```
program pepito;
private
    byte mivar1;
end
begin
    for (mivar1=1; mivar1<10; mivar1=mivar1+1)
        write(0,(mivar1*25)+30,100,4,mivar1);
        frame(2000);
    end
loop
    frame;
end
end
```

Este ejemplo tiene bastante miga. Vamos a repasarlo línea por línea. Nada más empezar el programa, nos metemos en el bucle FOR. Lo primero que ocurre allí es que asignamos el valor de 1 a “mivar1”, declarada previamente como de tipo BYTE. Después comprobamos si el valor de “mivar1” es menor que 10. Evidentemente,  $1 < 10$ , y por lo tanto, se ejecutarán las sentencias internas del for (en la primera iteración no se tiene en cuenta para nada la sección del FOR tras el último ;). Una vez ejecutadas, se volverá arriba y antes de nada, se efectuará, ahora sí, el incremento, pasando así “mivar1” a valer 2. (Fijarse que a partir de la segunda iteración ya no se usa para nada la sección del FOR de delante del primer ;). Después de hacer el incremento, se volverá a comprobar la condición. Como todavía  $2 < 10$ , se volverá a ejecutar el interior del FOR. Una vez hecho esto, se vuelve para arriba, se incrementa “mivar1” para que valga 3, se vuelve a comprobar la condición, y si es verdadera, se vuelve a ejecutar el interior del FOR, y así hasta que llegue una iteración donde al comprobar la condición ésta sea falsa. En ese momento, la ejecución del programa seguirá en la línea inmediatamente posterior al END del FOR.

Las sentencias interiores del FOR son dos. La primera lo que hace es escribir el valor de “mivar1”, que cambia en cada iteración, en posiciones diferentes según también el valor de “mivar1”, de manera que se vea claramente los incrementos y el momento donde el bucle acaba. Primero se imprime el número 1 (el primer valor de “mivar1”), después, un poco más a la derecha –fijarse en el segundo parámetro del write- se imprime el número 2 (el nuevo valor de “mivar1”), y así, hasta imprimir el número 9. Es importante recalcar que el número 10 no se imprimirá, porque, tal como hemos comentado, el proceso es primero incrementar, y luego hacer la comprobación: en la última iteración tendríamos que “mivar1” vale 9, entonces se va arriba, se incrementa a 10, se comprueba que 10 no es menor que 10, y se sale del FOR, por lo que cuando “mivar1” vale 10 ya no tiene la oportunidad de ser imprimida. Puedes jugar con los valores iniciales del contador, la condición de finalización o los valores de incremento. Así lo verás más claro.

La otra sentencia interior del FOR es la conocida FRAME para que cada vez que se realice una iteración, en la cual se quiere escribir un valor diferente de “mivar1”, se haga efectivo esa impresión. Como siempre, FRAME está dentro de un bucle, en este caso el FOR. Lo diferente es ese parámetro que no conocíamos que tuviera. ¿Para qué sirve?

Primero prueba de quitar ese parámetro y dejar el FRAME a secas como siempre. Verás que el programa se ejecuta tan rápidamente que casi no hay tiempo de ver que los números van apareciendo uno seguido de otro.

El parámetro del FRAME lo que indica es cuánto tiempo ha de pasar para que el programa espere a que el “agujero” del FRAME se abra y muestre el resultado del programa. Es decir, si no ponemos ningún parámetro, o ponemos el valor 100 –es lo mismo-, cada vez que se llegue a la línea FRAME, la “puerta” se abre y se muestra lo que hay ipsofacto. Este proceso es lo que se llama imprimir el siguiente fotograma. El fotograma es la unidad de tiempo utilizada en Fénix, (profundizaremos en ello en el siguiente capítulo), pero imagínate por ahora que el ritmo de aparición de los distintos fotogramas es como un reloj interno que tiene el programa, y que viene marcado por FRAME: cada vez que se “abre la puerta”, se marca el ritmo de impresión por pantalla. De ahí el nombre al comando FRAME: cada vez que se llega a ese comando es como gritar “¡Mostrar fotograma!”.

Si se pone otro número como parámetro del FRAME, por ejemplo el 500, lo que ocurrirá es que cuando se llegue a la línea FRAME, la “puerta” no se abrirá inmediatamente sino que se esperará 5 fotogramas para abrirse. Es como si el programa se quedara esperando un rato más largo en la línea FRAME, hasta ejecutarla y continuar. Así pues, este parámetro permite, si se da un valor mayor de 100, ralentizar las impresiones por pantalla. En el caso concreto de nuestro ejemplo, pues, cada iteración tardará en mostrarse 20 veces más que lo normal (el parámetro vale 2000). Si quieres, puedes cambiar su valor. También es posible poner un número menor de 100. En ese caso, lo que estaríamos haciendo es “abrir la puerta” más de una vez en cada fotograma: si pusiéramos por ejemplo 50, estaríamos doblando la velocidad normal de impresión por pantalla. Puedes comprobarlo también.

Para entender mejor esto último que estamos comentando, en un apartado del cuarto capítulo de este manual he escrito un pequeño código que creo que te podrá ayudar a entender mejor el sentido de este parámetro que puede llevar el frame, aunque en este momento este programa contiene elementos que no hemos visto todavía.

Volvamos al código que teníamos. Fíjate que después de haber escrito el bucle FOR, el programa no acaba ahí sino que se añade un bucle LOOP con una única sentencia FRAME. Prueba de quitar ese LOOP. Verás que el programa se ejecuta pero cuando ya se ha impreso todos los valores de "mivar1", evidentemente, el programa se acaba. Si queremos mantener el programa permaneciendo visible aún cuando se haya acabado el bucle del FOR, nos tenemos que inventar un sistema que también sea un bucle, infinito, cuya única función sea seguir mostrando el contenido de la pantalla tal cual. Esto es justo lo que hace el LOOP: va ejecutando el FRAME infinitamente para poder seguir visualizando la lista de números.

Por cierto, ¿qué pasaría si ahora comentamos la línea *frame(2000)*? Verás que el programa se ejecuta igual, pero que los números no van apareciendo por pantalla uno seguido de otro, sino que ya aparecen de golpe los 10 números. Esto es fácil de entender: si eliminamos la orden frame del bucle FOR, lo que estamos haciendo es evitar que en cada iteración se muestre por pantalla el resultado de dicha iteración justamente. Es decir, si eliminamos el frame, el bucle se realizará igual, iteración a iteración, pero hasta que no encuentre un frame no va a poder visualizar el resultado de sus cálculos. Por eso, cuando el programa entra en el Loop y se encuentra un frame (el primero), pone en pantalla de golpe todo aquello que llevaba haciendo hasta entonces, que es precisamente todas y cada una de las iteraciones del bucle FOR.

En apartados anteriores he comentado que para poder imprimir variables y no textos fijos se tenía que utilizar la función *write\_var* en vez de *write*, cosa que no he hecho en este ejemplo. ¿Por qué? En realidad, la función *write*, tal como acabamos de comprobar, sí que permite la impresión de valores de variables, incluso podríamos escribir algo así como

```
mivar1=10;  
write(0,100,100,4,"Esta variable vale:" + mivar1);
```

donde puedes ver que el signo + sirve para poder unir trozos de frases y valores separados para que aparezcan como un único texto (el signo + es pues en Fénix el llamado técnicamente operador de "concatenación" de cadenas).

La razón de haber elegido en este ejemplo la función *write* en vez de *write\_var* viene de su principal diferencia de comportamiento, ya explicada anteriormente, respecto la impresión por pantalla de valores variables. Recordemos que con *write* una vez que se escribe el valor de la variable, éste pasa a ser un valor fijo en pantalla como cualquier otro y no se puede cambiar y que en cambio, con *write\_var*, los valores de las variables que se hayan escrito, a cada fotograma se comprueba si han cambiado o no, y si lo han hecho, se actualiza automáticamente su valor. Es decir, que con *write\_var* los valores impresos de las variables cambiarán al momento que dichos valores cambien en el programa, permanentemente.

Lo puedes comprobar: si cambias la función *write* por *write\_var*, verás que a cada iteración se va imprimiendo un nuevo número, pero los números impresos anteriormente también cambian al nuevo valor: esto es así porque como estamos cambiando los valores a "mivar1", todos los valores de esa variable, impresos en cualquier momento de la ejecución del programa cambiarán también a cada fotograma que pase.

Un detalle curioso es ver que, si usamos *write\_var*, sí que se llega a imprimir el número 10. ¿Por qué? Porque, tal como hemos comentado, en la última iteración "mivar1" vale 9, entonces se sube arriba, se le incrementa a 10, se comprueba que sea cierta la condición, y al no serlo se sale del for. Entonces,

aparentemente el 10 no se tendría que escribir, ¿verdad? La clave está en darse cuenta que “mivar1” llega a valer 10 en el momento de salir del FOR. Y como seguidamente nos encontramos con un LOOP, este bucle nos permite continuar con la ejecución del programa, y por tanto, permite la aparición de sucesivos fotogramas. Como he dicho que a cada fotograma del programa se realiza una comprobación y actualización automática de todos los valores impresos por write\_var, como en ese momento “mivar1” vale 10, cuando se llega la primera vez al FRAME del LOOP, se llega a un fotograma nuevo, y así, se realiza la actualización masiva de todos los números. A partir de allí, como el programa ya únicamente consiste en ir pasando fotogramas y ya está, el valor 10 de “mivar1” va a continuar siendo el mismo y ya no se ve ningún cambio en pantalla.

Una última apreciación: fijate que el bucle FOR no es imprescindible para programar. Cualquier bucle FOR se puede sustituir por un bucle WHILE. Fíjate en los cambios que se han producido al ejemplo que estamos trabajando: hemos cambiado el FOR por un WHILE, pero el programa sigue siendo exactamente el mismo. Es fácil de ver:

```
program pepito;
private
    byte mivar1;
end
begin
    mivar1=1;
    while (mivar1<10)
        write(0,(mivar1*25)+30,100,4,mivar1);
        mivar1=mivar1+1;
        frame(2000);
    end
loop
    frame;
end
end
```

Y ya para acabar, a ver si sabes qué hace este código antes de ejecutarlo:

```
program nuevo;
private
    int a,b ;
end
begin
    for (a=0,b=10;a<5 AND b>5;a=a+1,b=b-1)
        write(0,0,10*a,0,"a="+a+" b="+b) ;
    end
loop
    frame;
end
end
```

### **Bloque “REPEAT/UNTIL”:**

```
REPEAT
    Sentencias;
UNTIL ( condición )
```

La sentencia REPEAT/UNTIL es una sentencia muy similar a WHILE.

Debe comenzar con la palabra reservada REPEAT, seguida de las sentencias que se quieren repetir una o más veces y el final de la sentencia se determinará poniendo la palabra reservada UNTIL –“hasta”- seguida de la condición que se debe cumplir para que se dé por finalizada la sentencia.

Cuando se ejecute una sentencia REPEAT se ejecutarán primero las sentencias interiores (las que están entre el REPEAT y el UNTIL) y, tras hacerlo, se comprobará la condición especificada en el UNTIL y si ésta continúa siendo falsa, se volverán a ejecutar las sentencias interiores. El proceso se repetirá hasta que la condición del UNTIL resulte cierta, continuando entonces la ejecución del programa en la sentencia siguiente al bucle.

Las sentencias interiores a un bucle REPEAT pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles REPEAT.

Una sentencia BREAK dentro de un bucle REPEAT lo finalizará de forma inmediata, continuando el programa a partir de la sentencia siguiente a dicho bucle. Una sentencia CONTINUE dentro de un bucle REPEAT forzará al programa a comprobar la condición del UNTIL inmediatamente, y si ésta es falsa, volverá a ejecutar las sentencias interiores desde el principio (tras la palabra reservada REPEAT). Si la condición resulta cierta, la sentencia CONTINUE finalizará el bucle.

Cada vez que se ejecutan las sentencias interiores se dice que se ha realizado una iteración del bucle. La sentencia REPEAT...UNTIL (literalmente traducida como REPETIR...HASTA (que se cumpla la) condición) siempre ejecutará las sentencias interiores al menos una vez, ya que comprueba la condición siempre tras ejecutarlas. Ahí está de hecho la diferencia con el bucle WHILE. Como en el WHILE la condición se comprueba antes de ejecutar las sentencias interiores, puede darse el caso que la primera vez que se llegue a un WHILE la condición ya sea directamente falsa, y por tanto, las sentencias interiores no se lleguen a ejecutar nunca. En cambio, como en el REPEAT la condición se comprueba después de ejecutar las sentencias interiores, como mínimo éstas siempre se ejecutarán una vez, porque no se comprobará la falsedad de la condición hasta después de haber ejecutado dichas sentencias interiores.

La diferencia comentada anteriormente entre un WHILE/END y un REPEAT/UNTIL la puedes ver en estos ejemplos:

```
program ej1;
private
  byte mivar1=1;
end
begin
  while(mivar1<1)
    write(0,100,100,4,mivar1);
    frame;
  end
loop
  frame;
end
end

program ej2;
```

```

private
  byte mivar1=1;
end
begin
  repeat
    write(0,100,100,4,mivar1);
    frame;
  until (mivar1<1)
  loop
    frame;
  end
end

```

En el primer ejemplo sólo veremos una pantalla negra, y en el segundo el número 1 impreso. Esto es porque en el primer ejemplo, antes de nada, se comprueba si la condición es verdadera. Como no lo es, el while/end no se ejecuta y se pasa directamente al Loop, donde no hay nada que mostrar. En cambio, en el segundo ejemplo, primero se realiza el write y el frame siguientes al repeat, y después se comprueba la condición. Al ser falsa, no se realiza ninguna iteración más y se va a loop, pero como mínimo se ha impreso el número debido a la única ejecución del write que ha existido.

### **Bloque “FROM/END”:**

```

FROM variable = valor_num TO valor_num STEP valor_num;
  Sentencias;
END

```

La sentencia FROM implementa un bucle también. Para ello, se necesita una variable que sirva como contador del bucle. De hecho, su funcionamiento es casi idéntico al FOR/END, lo único que el FROM tiene una sintaxis más sencilla y sólo sirve para bucles incrementales/decrementales mientras que el FOR permite muchísimos otros usos gracias a que se puede definir la comparación y la función de incremento.

Antes de las sentencias que conformarán el grupo interior de sentencias se debe poner la palabra reservada FROM seguida del nombre de la variable contador, el símbolo de asignación (=), el valor inicial de la variable, la palabra reservada TO y, finalmente, el valor final de la variable. Tras esta declaración del bucle FROM se debe poner el símbolo ; (punto y coma). Después de esta cabecera definiendo las condiciones del bucle vendrá el grupo interior de sentencias que se pretende repetir un número determinado de veces y, al final, la palabra reservada END.

La primera iteración se hará con el valor inicial en la variable usada como contador; tras esta iteración se le sumará 1 a esta variable (si el valor inicial es menor que el valor final) o se le restará 1 (en caso contrario). Tras actualizar el valor de la variable, se pasará a la siguiente iteración siempre que el valor de la variable no haya llegado (o sobrepasado) el valor final del bucle.

Las sentencias interiores a un bucle FROM pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles FROM.

Una sentencia BREAK dentro de un bucle FROM lo finalizará de forma inmediata, continuando el programa por la sentencia siguiente a dicho bucle (tras el END). Una sentencia CONTINUE dentro de un bucle FROM forzará al programa a incrementar inmediatamente la variable usada como contador y, después, si no se ha sobrepasado el valor final, comenzar con la siguiente iteración.

Como opción, es posible poner tras los valores inicial y final de la sentencia, la palabra reservada STEP seguida de un valor constante que indique el incremento de la variable contador tras cada iteración del bucle, en lugar de +1 o -1, que son los incrementos que se harán por defecto si se omite la declaración STEP (paso).

Los valores inicial y final de un bucle FROM deben ser diferentes. Si el valor inicial es menor que el valor final, no se puede especificar un valor negativo en la declaración STEP, y viceversa.

## Profundizando en la escritura de texto en la pantalla:

Hasta ahora para trabajar con textos en pantalla hemos utilizado las funciones *write*, *write\_var*, *delete\_text* y poco más. Pero hay más funciones relacionadas con los textos.

Por ejemplo, tenemos *move\_text*, que sirve para mover a otras coordenadas de la pantalla un texto ya impreso antes, cuyo identificador (generado por el *write* utilizado la primera vez que se imprimió ese texto) se ha de pasar como parámetro

Recordemos el primer código que aprendimos en este curso:

```
Program MiPrimerPrograma;
Private
    int mivar1;
End

Begin
    mivar1=10;
    while(mivar1<320)
        delete_text(0);
        mivar1=mivar1+2;
        write(0,mivar1,100,1,"¡Hola mundo!");
        frame;
    end
end
```

Recuerda que lo que hacía este programa era hacer un scroll horizontal de la frase “¡Hola mundo!”. Podríamos ahora modificar este programa para que haga exactamente lo mismo, pero utilizando la función *move\_text*. Es más, esta nueva versión incluso sería “mejor” que la primera porque se pueden evitar los posibles parpadeos que se producen cuando se borra y escriben los textos constantemente a una elevada velocidad, tal como se hace en el programa original. Bien, pues la versión con *move\_text* será así:

```
Program MiPrimerPrograma;
Private
    int mivar1;
End

Begin
    mivar1=10;
    write(0,10,100,1,"¡Hola mundo!");

    while(mivar1<320)
        mivar1=mivar1+2;
```

```

                move_text(1,mivar1,100);
                frame;
            end
        end
end

```

*Move\_text* tiene tres parámetros: el identificador del texto que se quiere mover, y la nueva coordenada X y Y donde se quiere mover. Fíjate que como valor del primer parámetro en el ejemplo ponemos un 1, porque recuerda que *write* devuelve siempre (aunque nosotros no lo recojamos) el identificador único del texto que escribe, y siempre empieza por el número 1 y va aumentando (2,3...) por cada nuevo texto que se imprime. Así pues, en vez de borrar y reescribir el mismo texto, simplemente se mueve a la coordenada (mivar1,100).

Fíjate además en otro detalle. Para poder utilizar *move\_text* hemos tenido que escribir previamente el texto con *write* (para conseguir su identificador). Es decir, que el funcionamiento es imprimir una vez el texto en una posición inicial de pantalla, y posteriormente, irlo moviendo. Por tanto, el *write* lo hemos sacado fuera del *while*, porque sólo imprimiremos el texto una sola vez, la primera. Y el *delete\_text* lo hemos quitado porque ya no hace falta.

Bueno, ahora cambiaremos un poco el programa y añadiremos otro texto que diga “¡Y adiós!”, pero que permanecerá fijo en pantalla. Y lo que haremos será cambiarle el color. Así:

```

Program MiPrimerPrograma;
Private
    int mivar1;
End

Begin
    mivar1=10;
    write(0,10,100,1,"¡Hola mundo!");
    set_text_color(25);
    write(0,250,50,1,"¡Y adiós!");

    while(mivar1<320)
        mivar1=mivar1+1;
        move_text(1,mivar1,100);
        frame;
    end
end

```

Lo que hemos introducido de novedad es la función *set\_text\_color*. Esta función cambia el color de los textos impresos por TODOS los comandos *write* que aparezcan a partir de entonces en el código el código fuente. ¿Y cuál es el nuevo color? El que se especifique como valor en el único parámetro que tiene.

Este número es, en el modo de 16 bits, el resultado de mezclar unas componentes RGB determinadas. Me explico.

Todos los colores en la pantalla del ordenador se forman a partir de tres colores básicos, que son el rojo, el azul y –atención- el verde. El negro absoluto es la falta de color, y por tanto es aquel color que no contiene ni pizca de los tres colores básicos, es decir, el que tiene sus componentes RGB (Red-

Green-Blue, o Rojo-Verde-Azul en español) a cero. El blanco entonces sería aquel que tiene cada uno de sus componentes RGB al máximo valor (porque el blanco es la suma de los colores primarios en plenitud). Cualquier color de entremedio tendrá una cierta cantidad de R, una cierta cantidad de G y otra de B.

Se denomina modo de 16 bits porque se reserva esta cantidad de memoria para generar el color: en concreto, 5 bits para el rojo, 6 para el verde y 5 para el azul. En este tipo de gráficos, no se utiliza paleta, a diferencia del de 8 bits.

Has de saber que también existen gráficos de 24 bits (donde cada componente tiene reservado 8 bits), con lo que el número de combinaciones (y por tanto, de colores) posibles aumenta de forma muy sensible respecto los demás modos gráficos. La mala noticia es que Fénix aún no los soporta. Y para rematar el asunto, también existen los gráficos de 32 bits, que son iguales que los de 24 pero reservan 8 bits para información sobre la posible transparencia.

En este manual vamos a trabajar siempre, salvo advertencia expresa, con gráficos de 16 bits. En este modo gráfico, para poder utilizar un color determinado a partir de sus componentes RGB lo normal es hacerlo con cualquier programa de edición de imágenes. No hay ningún secreto: se eligen las componentes de color a partir del cuadro de diálogo correspondiente y se pinta con la herramienta seleccionada. Por defecto, tanto el MSPaint como el Gimp (por poner dos ejemplos de programas que usaremos más) guardan las imágenes que sean PNG en 24 bits, pero esto no será ningún problema porque Fénix al cargarlas las convierte automáticamente en 16 bits -perdiendo información en el proceso, eso sí- ,y por tanto, no tendremos que preocuparnos de nada.

Si quisieras crear con el Gimp una imagen de 8 bits paletizada (en algún lugar de este manual lo necesitaremos, como por ejemplo en el apartado del seguimiento de caminos con las funciones `PATH_*`, o en la creación de scrolls Modo7), deberías, antes de guardarla, de cambiarla a modo 8 bits, yendo al menú "Imagen->Modo->Indexado" y elegir la paleta deseada.

No obstante, si el color lo quieres utilizar programando en Fénix, tendrás que usar una nueva función, la función `RGB()`. Esta función tiene tres parámetros, que son respectivamente la cantidad de rojo que queremos para nuestro nuevo color, la cantidad de verde y la cantidad de azul. El mínimo es 0 -no hay nada de esa componente en nuestro color- y el máximo es 255. Así pues, para generar el negro tendríamos que escribir algo así como `rgb(0,0,0)`, y para generar el blanco `rgb(255,255,255)`. Para generar el rojo más intenso, tendríamos `rgb(255,0,0)`; asimismo, para el verde más intenso `rgb(0,255,0)`; para el azul más intenso, `rgb(0,0,255)`; y a partir de aquí, con todas las combinaciones que se nos ocurran podremos crear los colores que queramos.

Pero si nos fijamos bien en el ejemplo anterior, la función `set_text_color` tiene como valor de su parámetro el número 25. ¿Qué quiere decir esto?. En realidad, cada color viene identificado por un número. De hecho, lo que hace la función `rgb()` es devolver un número, un único número resultado de realizar complicados cálculos matemáticos a partir de las cantidades de las componentes. Y este número entero es el que realmente identifica al color en sí. Por lo tanto, ese número 25 identifica a un determinado color de entre todos los posibles que se podrían obtener a partir de todas las combinaciones de las componentes. Como este resultado del complejo cálculo de mezcla puede depender de la tarjeta gráfica que se tenga, más que poner un número directamente es más recomendable utilizar la función `RGB`, ya vista, porque así tendremos más control sobre el color que queremos que aparezca. Es decir, que poniendo el número 25 podríamos encontrarnos que para un ordenador el color nº 25 es parecido al fucsia y en otro ordenador es más parecido al marrón. Por lo tanto, sería más seguro poner `set_text_color(rgb(255,0,0))`; para imprimir los textos que venga a continuación de color rojo puro, por ejemplo.

Existe una función inversa a `RGB()` llamada `GET_RGB()`, que a partir de un número entero que identifica a un color, obtiene la cantidad correspondiente de su componente roja, verde o azul. Esta función es un poco más avanzada (y menos útil) y la veremos en próximos capítulos

También existe otra función, `GET_TEXT_COLOR()`, la cual no tiene ningún parámetro y lo que devuelve es un número que corresponde al color con el que en este momento se están escribiendo los textos.

Es decir, que en ejemplo anterior, si se utilizara esta función,devolvería el número 25.

Volvamos a nuestro ejemplo. Nota que el nuevo texto permanecerá fijo porque el `move_text` sólo afecta al texto con código 1, y como “¡Y adiós!” se imprime después de “¡Hola mundo!”, “¡Y adiós!” tendrá automáticamente el código 2.

Por último, un ejercicio. ¿Cómo haríamos para hacer que el texto “¡Y adiós!”, que ahora está fijo, realice un scroll horizontal, y que lo además haga de derecha a izquierda? Pues así:

```
Program MiPrimerPrograma;
Private
    int mivar1;
End
Begin
    mivar1=0;
    write(0,10,100,1,"¡Hola mundo!");
    set_text_color(25);
    write(0,320,50,1,"¡Y adiós!");

    while(mivar1<320)
        mivar1=mivar1+2;
        move_text(1,mivar1,100);
        move_text(2,320-mivar1,50);
        frame;
    end
end
```

Hacemos que el segundo texto (código identificador 2) se mueva también,procurando que la coordenada X inicial sea la más a la derecha posible (320) y que en cada iteración vaya disminuyendo para así irse moviendo hacia la izquierda.Cuando “mivar1” valga 0, el primer texto se imprimirá en (0,100) y el segundo en (320,50); cuando “mivar1” valga 100 el primer texto se imprimirá en (100,100) y el segundo en (220,50), cuando valga 320 el primer texto se imprimirá en (320,100) y el segundo en (0,50)...

Otro ejemplo un poco más avanzado y bonito: un texto oscilante. En este caso utilizamos junto con `move_text` la función `get_disty`, que no me hemos visto y que se estudiará más adelante.

```
Program lala;
private
    int var=10;
    int idtexto;
end
Begin
    set_mode (800,600,16);
    idtexto=write(0,100,30,4,"texto oscilante");
    Loop
        var=var+22500;
        move_text(idtexto,100+get_disty(var,20),30);
    end
```

```
Frame;  
If(key(_esc)) Break; End  
End  
End
```

Dejando de banda los ejemplos anteriores, hay otros detalles importantes relacionados con los textos en pantalla, como la variable entera global predefinida **TEXT\_Z** que, al modificarla, estamos cambiando la profundidad a la que se escriben los textos que se escribirán a partir de entonces. Esta variable es útil por si nos interesa que los textos queden detrás o delante de algún gráfico (ya que éstos -ya lo veremos- también tienen una profundidad regulable), o detrás del fondo para hacerlos invisibles....

Cuanto más "atrás", más alejado queramos poner el texto, mayor será su valor -hasta el máximo que permite el tipo Int-, y cuanto más "adelante", más cercano queramos escribirlo, menor será su valor. Por defecto, si no se indica nada, los textos se escriben con un valor de TEXT\_Z igual a -256.

Que esta variable sea "predefinida" significa que no es necesario declararla en ningún sitio: se puede usar directamente con ese nombre en cualquier parte de nuestro código. Eso sí, para la tarea que tiene asignada (cambiar la profundidad del texto), no para otra cosa. Lo de que sea "global" no nos importa ahora: ya se explicará en posteriores capítulos.

También disponemos de las funciones `text_height()` y `text_width()`, dos funciones que, pasándoles como primer parámetro el identificador correspondiente a la fuente de letra que se quiera utilizar (si no queremos usar ninguna concreta y con la predefinida del sistema nos basta, tendremos que escribir un 0), y como segundo parámetro un texto cualquiera, nos devuelven el alto y el ancho, respectivamente, de ese texto en píxeles (es decir, la altura o ancho del carácter más grande que vaya a dibujarse de toda la cadena), pero sin llegar a escribir nada: algo importante para calcular en qué posición escribir ese u otro texto, antes de hacerlo de forma efectiva.

### Algunos ejemplos más de códigos fuente:

A continuación mostraré unos cuantos códigos fuente, sin explicarlos, a modo de síntesis de todo lo que ha explicado hasta ahora. En estos códigos no se introduce ningún concepto nuevo: han de servir como refresco y ejemplo práctico de todo lo leído hasta ahora. Si se han comprendido las ideas básicas de programación (condicionales, bucles, variables...) no debería de ser difícil adivinar qué es lo que hacen cada uno de los códigos siguientes, (antes de ejecutarlos, por supuesto).

#### **Primer ejemplo:**

```
program ejemplo1;  
private  
    int mivarX=33;//Un valor inicial cualquiera  
    int mivarY=174; //Un valor inicial cualquiera  
end  
begin  
    set_mode(320,200,16);  
  
    loop  
        mivarX=rand(0,320);  
        mivarY=rand(0,200);
```

```

                if(mivarX>100 and mivarX <250 and mivarY>50 and mivarY<150)
                    continue;
                end
                write(0,mivarX,mivarY,4,".");
                frame;
            end
        end
end

```

Con este código se podría jugar a poner un *delete\_text(0)* justo antes del *write*, para ver qué efecto curioso da (y así tampoco dará el error de “Demasiados textos en pantalla”), aunque haciendo esto no se notará el efecto del bloque IF/END.

### **Segundo ejemplo:**

```

program ejemplo2;
private
    int columna;
    int fila;
end
begin
    set_mode(320,200,16);
    for(columna=1;columna<=5;columna=columna+1)
        for(fila=0;fila<5;fila=fila+1)
            write(0,(columna*20)+30,(fila*20)+30,4,"*");
            frame;
        end
    end
    loop
        frame;
    end
end

```

Con este código también se podría jugar a poner un *delete\_text(0)* delante del *write* para observar mejor cómo cambian en las distintas iteraciones los valores de las variables “columna” y “fila”.

Explico un poco el código: fijate que hay dos FOR anidados: esto implica que por cada iteración que se produzca del FOR externo *for(columna=1;...)* -en total son 5: de 1 a 5-, se realizarán 5 iteraciones del FOR interno *for(fila=1;...)* -de 0 a 4-, por lo que se acabarán haciendo finalmente 25 iteraciones con los valores siguientes: columna=1 y fila=0, columna=1 y fila=1, columna=1 y fila=2, columna=1 y fila=3, columna=1 y fila=4, columna=2 y fila=0, columna=2 y fila=1, etc. Luego ya sólo basta imprimir el signo \* en las coordenadas adecuadas para dar la apariencia de cuadrado.

Un detalle interesante es observar qué pasa si eliminamos la línea *frame;* del interior de los bucles FOR. El cuadrado se verá igual, pero de golpe: no se verá el proceso de impresión gradual de los asteriscos. Esto es fácil de entender. Si mantenemos la orden *frame* dentro de los FOR, en cada iteración estaremos obligando al programa a mostrar por pantalla el resultado de los cálculos que ha realizado desde el último *frame* anterior. Pero en cambio, si no lo ponemos, los cálculos se realizarán igualmente -las iteraciones y la orden *write* se ejecutarán- pero no se mostrará ningún resultado visible hasta que se encuentre con alguna orden *frame;*, orden que aparece en el bloque LOOP/END del final (necesario para permitir que el programa continúe ejecutándose indefinidamente y no se acabe ipsofacto). Por tanto, cuando, una vez que el programa haya salido de los FOR anidados, se encuentre la primera vez con el *frame;* del interior del LOOP, “vomitará” de golpe en la pantalla todo aquello que ha venido calculando y “escribiendo” hasta entonces, por lo que el cuadrado de asteriscos aparecerá directamente completo.

Una variante interesante de este código es el siguiente:

```
program ejemplo2;
private
    int columna;
    int fila;
end
begin
    set_mode(320,200,16);
    for(columna=1;columna<=5;columna=columna+1)
        for(fila=1;fila<=columna;fila=fila+1)
            write(0,(columna*20)+30,(fila*20)+30,4,"*");
            frame;
        end
    end
    loop
        frame;
    end
end
```

Lo único que se ha cambiado son los valores de los elementos que componen el FOR más interno. Fíjate que ahora este FOR realizará tantas iteraciones como el valor de “columna” le permita, porque ahora el límite máximo de iteraciones de este FOR viene impuesto por dicho valor. Es decir, que tendremos este orden de valores para las dos variables: columna=1 y fila=1, columna=2 y fila=1, columna=2 y fila=2, columna=3 y fila=1, columna=3 y fila=2, columna=3 y fila=3, etc. Por eso sale ese triángulo.

Si queremos que salga otro tipo de triángulo podríamos hacer:

```
program ejemplo2;
private
    int columna;
    int fila;
end
begin
    set_mode(320,200,16);
    for(fila=1;fila<=5;fila=fila+1)
        for(columna=1;columna<=fila;columna=columna+1)
            write(0,(columna*20)+30,(fila*20)+30,4,"*");
            frame;
        end
    end
    loop
        frame;
    end
end
```

O bien, otra manera sería:

```
program ejemplo2;
```

```

private
    int columna;
    int fila;
end
begin
    set_mode(320,200,16);
    for(fila=5;fila>=1;fila=fila-1)
        for(columna=5;columna>=fila;columna=columna-1)
            write(0,(columna*20)+30,(fila*20)+30,4,"*");
            frame;
        end
    end
    loop
        frame;
    end
end
end

```

Podemos "pintar" más figuras curiosas como ésta:

```

program ejemplo2;
private
    int i;
end
begin
    set_mode(320,200,16);
    for(i=100;i<=200;i=i+10)
        write(0,i,50,4,"*");
        write(0,i,100,4,"*");
    end
    for(i=50;i<=100;i=i+10)
        write(0,100,i,4,"*");
        write(0,200,i,4,"*");
    end
    loop
        frame;
    end
end
end

```

O ésta:

```

program ejemplo2;
private
    int i;
end
begin
    set_mode(320,200,16);
    for(i=75;i<=125;i=i+10)
        write(0,i,65,4,"*");
    end
    for(i=50;i<=100;i=i+10)

```

```

                                write(0,100,i,4,"*");
    end
    loop
        frame;
    end
end

```

Y ya para acabar, un juego. Evidentemente, el siguiente código no escribirá nada en pantalla. ¿Por qué?

```

program ejemplo2;
private
    int columna;
    int fila;
end
begin
    set_mode(320,200,16);
    for(fila=0;fila>=1;fila=fila-1)
        for(columna=5;columna>=fila;columna=columna-1)
            write(0,(columna*20)+30,(fila*20)+30,4,"*");
            frame;
        end
    end
    loop
        frame;
    end
end
end

```

### **Tercer ejemplo:**

El siguiente ejemplo imprimirá por pantalla los 20 primeros números de la serie de Fibonacci. La serie de Fibonacci es una secuencia de números enteros que tiene muchas propiedades matemáticas muy interesantes, y que en la Naturaleza se ve reflejada en multitud de fenómenos.

La serie de Fibonacci comienza con dos números 1. A partir de ahí, los números que forman parte de esta serie han de cumplir lo siguiente: ser el resultado de la suma de los dos números anteriores. Es decir, el siguiente número ha de ser la suma de los dos 1. Por tanto, 2. El siguiente número ha de ser la suma de un 1 con ese 2. Por tanto, 3. Y así. Es fácil ver, por tanto, que el resultado que queremos que obtenga nuestro programa ha de ser el siguiente:

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

Antes de nada, hay que decidir qué bucle utilizaremos para imprimir estos números (porque está claro que utilizaremos un bucle, ¿no?). Como queremos exactamente 20 números, lo más sencillo y directo es utilizar un bucle FOR, porque con este bucle tenemos controlado cuántas iteraciones exactamente queremos realizar (en este caso, 20). Así que el código fuente es el siguiente:

```

program ejemplo3;
private
    int num=1;
    int num2=1;

```

```

        int suma;
        int cont;
end
begin
    set_mode(640,480,16);
    //Imprimimos los dos primeros números de la serie (1 y 1)
    write(0,200,10,4,num);
    write(0,200,20,4,num2);
    for(cont=3;cont<=20;cont=cont+1)
        suma=num+num2;
        write(0,200,cont*10,4,suma);
/*Corremos los números."num" se pierde,"num2" pasa a ser "num" y "suma" pasa a ser "num2"*/
        num=num2;
        num2=suma;
        frame;
    end
    loop
        frame;
    end
end
end

```

Otro ejemplo de programa "matemático", el cual escribe por pantalla los quince primeros números que cumplen ser múltiplos de 3 pero no de 6:

```

program ejemplo2;
private
    int i=0;
    int cont=0;
end
begin
    set_mode(320,200,16);
    loop
        i=i+1;
        if(i%3==0 and i%6!=0)
            write(0,20+3*i,100,4,i + ",");
            cont=cont+1;
        end
        if(cont==15) break; end
    end
    loop
        frame;
    end
end
end

```

¿Sabrías decir antes de ejecutar este código, qué es lo que saldrá por pantalla?

```

program ejemplo2;
private
    int i=0;
    int result;
end

```

```

begin
    set_mode(320,200,16);
    for(i=1;i<=20;i++)
        result=i*i;
        write(0,150,10*i,4,result);
    end
loop
    frame;
end
end

```

¿Entiendes las operaciones matemáticas que se utilizan en este ejemplo?

```

program hola;
private
    int a=57;
end
begin
    write(0,160,90,4,"Tengo billetes de 5 euros y monedas de 2 y 1 euro.");
    write(0,160,100,4,"Para conseguir " + a + " euros necesito");
    write(0,160,110,4,a/5 + " billetes y " + (a%5)/2 + " monedas de 2 y " + (a%5)%2 + " monedas de 1");
loop
    frame;
end
end

```

## CAPÍTULO 3: INCLUSIÓN DE GRÁFICOS

Todos los ejemplos que hemos hecho hasta ahora no han incluido ningún tipo de gráfico: sólo textos. Es evidente que para diseñar un juego necesitarás gráficos, además de posiblemente música y otros efectos. En este capítulo se tratará las diferentes maneras de incluir los gráficos de nuestro juego.

### Los modos de pantalla:

Hasta ahora, la pantalla donde se visualizaba cualquier programa era de un tamaño de 320x240 píxeles, ya que no habíamos indicado nada que dijera lo contrario. Una de las cosas que debemos aprender primero es configurar el modo de pantalla, es decir, la resolución de nuestro juego. Esto es un paso importante porque determina muchas cosas del juego, como el tamaño de los gráficos o los recursos necesarios: a grandes resoluciones tenemos gran cantidad de píxeles (puntos) en la pantalla, gracias a eso conseguimos una mayor nitidez en la imagen y se reduce el efecto de bordes dentados o los escalones, pero a cambio, los gráficos deben ser más grandes y por lo tanto aumenta la cantidad de cálculos que tiene que hacer el ordenador.

Para establecer la resolución de nuestro juego (o sea, establecer el "modo gráfico"), se utiliza la función *set\_mode*.

Es válido llamar a esta función aunque ya haya un modo gráfico en funcionamiento, sea para cambiar la resolución de pantalla, o pasar de ventana a pantalla completa, por ejemplo, aunque se recomienda llamar a *set\_mode* nada más comenzar el programa, para evitar efectos de parpadeo al iniciar el modo gráfico dos veces.

Esta función tiene cuatro parámetros, (los dos últimos opcionales). Los dos primeros son justamente la resolución horizontal y vertical del juego respectivamente. Algunos valores posibles son: 320x200, 320x240 –por defecto–, 320x400, 360x240, 376x282, 400x300, 512x384, 640x400, 640x480, 800x600, 1024x768 o 1280x1024. En el caso de que se escriba una resolución no soportada, Fenix tratará de emularla estableciendo la resolución siguiente de mayor tamaño y dibujando un borde negro automáticamente alrededor, creando un efecto Cinemascope. Probémoslo:

*Program ejemplo;*

*Begin*

```
Set_mode(640,480);
```

```
Loop
```

```
Frame;
```

```
end
```

*End*

El tercer parámetro de *set\_mode* puede tener dos valores: el número 8 (o de forma equivalente, la palabra `MODE_8BITS`) o el número 16 (o de forma equivalente, la palabra `MODE_16BITS`). Este parámetro sirve para indicar la profundidad de color a la que funcionará el juego. Fíjate que un juego en Fénix sólo puede usar pues la profundidad de 8 bits (256 colores) o 16 bits (color de alta densidad). Una profundidad de 16 es más que suficiente para alcanzar todas las tonalidades que el ojo humano es capaz de distinguir, así que en principio no es imprescindible profundidades mayores, las cuales acarrearían más carga de procesamiento por parte del intérprete. Ya que este parámetro es opcional, si no se escribe nada es sobreentendiéndose que la profundidad será de 8 bits.

Por defecto, esta función inicializa un modo gráfico en ventana. El cuarto parámetro permite especificar un modo gráfico a pantalla completa, además de otros parámetros adicionales. Si este parámetro

se especifica –es opcional- , puede ser una combinación (suma) de uno o más de los valores -sin comillas- siguientes:

<b>MODE_FULLSCREEN</b>	Crea un modo a pantalla completa. Trabajando a pantalla completa podemos usar cualquier resolución que admita nuestra tarjeta de vídeo y el monitor. Nosotros vamos a trabajar en modo ventana por varios motivos: por ejemplo, si trabajamos en pantalla completa no podemos ver los errores que cometemos: si hay demasiados textos en pantalla debería salir una ventana informándonos de ello y al aceptar se cerraría el programa, pero en pantalla completa la ventana no se ve y el programa se cuelga.
<b>MODE_WINDOW</b>	Crea un modo en ventana –por defecto-.
<b>MODE_2XSCALE</b>	Dobla la resolución gráfica de nuestro juego. Los bordes de los gráficos aparecerán suavizados.
<b>MODE_HARDWARE</b>	Crea una superficie en buffer de video para la ventana o pantalla. Normalmente, Fenix dibuja sobre la memoria RAM del ordenador y copia el resultado en la memoria de la tarjeta gráfica. Con este parámetro activado, Fenix dibujará directamente sobre la memoria de vídeo. Esto acelera algunas operaciones, pero enlentece otras (como por ejemplo, el dibujo con transparencias).
<b>MODE_DOUBLEBUFFER</b>	Habilita un segundo buffer. Su interés suele venir en combinación con <b>MODE_HARDWARE</b> .
<b>MODE_MODAL</b>	Hace que la ventana se abra en modo MODAL, evitando que la ventana pierda el foco y confinando el movimiento del ratón a su superficie.
<b>MODE_FRAMELESS</b>	Crea una ventana sin bordes.

De manera alternativa, en vez de especificar los valores de la tabla anterior como cuarto parámetro de la función *set\_mode*, también existe la posibilidad de asignarlos a una variable global predefinida llamada **GRAPH\_MODE**. Esa asignación se ha de producir antes de la llamada a *set\_mode* para que funcione.

Por ejemplo, el siguiente código hace uso indistinto o bien de la variable **GRAPH\_MODE** o bien de utilizar su valor como cuarto parámetro de *set\_mode*. Si lo ejecutas verás que este ejemplo pondrá la ventana del programa inicialmente a pantalla completa, al cabo de un segundo cambiará la resolución y pondrá la ventana "normal", seguidamente volverá a cambiar la resolución y pondrá la ventana sin marcos, y finalmente, volverá a cambiar la resolución y pondrá la ventana en modo modal:

```

program hola;
begin
  set_fps(1,1); //Pronto se explicará para qué sirve esta función. Para entender el ejemplo no es importante.
  graph_mode=mode_fullscreen;
  set_mode(640,480,16);
  frame;
  set_mode(320,240,16, mode_window);
  frame;
  set_mode(800,600,16, mode_frameless);
  frame;
  graph_mode=mode_modal;
  set_mode(640,480,16);
  frame;
loop
  frame;
end

```

*end*

Y todavía un dato más: la variable `GRAPH_MODE` también puede valer, aparte de los valores anteriores, los dos valores del tercer parámetro de `set_mode`: `mode_8bits` o `mode_16bits`, de manera que esta variable puede servir para especificar ambas cosas a la vez (profundidad de color y modo gráfico) sumando los valores respectivos. Por ejemplo, si estableciéramos `GRAPH_MODE=mode_8bits+mode_frameless`; tendríamos una profundidad de 8 bits en una ventana sin marco. Si hiciéramos `GRAPH_MODE=16`, tendríamos una profundidad de 16 bits y como no se especifica nada más, el modo gráfico sería el por defecto, que es la ventana "normal". Depende de los valores que se le asignen, entonces no haría falta especificar ni tercer ni cuarto parámetro en `set_mode`, el cual sólo serviría para establecer la resolución.

Que sepas que también existe otra variable global predefinida llamada ***FULL\_SCREEN***, la cual puede valer "true" (1) o "false" (0) que se puede utilizar como alternativa tanto al cuarto parámetro de `set_mode` como a la propia variable `GRAPH_MODE` si se desea especificar simplemente si el juego funcionará a pantalla completa o no, respectivamente.

Si la función `set_mode` crea una ventana de juego, ésta utilizará por título el nombre del fichero DCB y un icono estándar. La función `set_title` establece el título de la ventana del juego (el cual se pasa entre comillas como el único parámetro que tiene), y la función `set_icon` usa un gráfico determinado como icono para mostrar en la barra de título de la ventana del juego (el funcionamiento de esta función lo estudiaremos posteriormente, cuando hayamos introducido el uso de gráficos). Es recomendable siempre usar estas funciones (`set_title` y `set_icon`) inmediatamente antes de cada llamada a `set_mode` que se haga en el programa, ya que si después de estas funciones no aparece la función `set_mode`, no se visualizarán sus efectos.

Por ejemplo, si quisiéramos hacer que nuestro juego funcionara a 800x600 de resolución, con profundidad de 16 bits, con ventana modal sin bordes, tendríamos que escribir lo siguiente:

*Program ejemplo;*

*Begin*

```
Set_mode(640,480,MODE_16BITS,MODE_MODAL + MODE_FRAMELESS);
```

```
Loop
```

```
Frame;
```

```
end
```

*End*

Si queremos que se vea un título, hemos de hacer que se vean los bordes:

*Program ejemplo;*

*Begin*

```
Set_title("Prueba");
```

```
Set_mode(640,480,MODE_16BITS,MODE_MODAL);
```

```
Loop
```

```
Frame;
```

```
end
```

*End*

### **Configuración de los frames per second (FPS):**

Lo siguiente que haremos será definir el número de imágenes por segundo que vamos a visualizar.

Ya he comentado que cualquier animación o imagen en movimiento se consigue haciendo dibujos similares con pequeñas diferencias, por ejemplo, de posición. Esto es lo que se hace en la televisión, se muestran imágenes fijas a mucha velocidad, una tal que el ojo humano no pueda diferenciarlo. Pues nosotros podemos decidir cuantas imágenes por segundo vamos a ver, o como se dice en inglés, los “frames per second” (FPS).

Si estás curtido en videoconsolas, ya habrás oído aquello de juegos a 50Hz o a 60Hz (herzios) que no es más que la medida de imágenes por segundo que es capaz de “pintar” la televisión, y ciertamente hablamos de valores muy elevados, esto viene a que es importante recalcar la utilidad de escoger un buen “frame rate”: el ojo humano, se ha demostrado, que no es capaz de distinguir imágenes fijas a más de 32 imágenes por segundo, pero realmente, si tienes buena vista, puedes notar que la imagen va a saltos, y a mayor número de imágenes por segundo más suave se verá el juego, y por supuesto menos perjudicial para la vista; pero en el otro extremo tenemos los requisitos del sistema: un ordenador necesita hacer cientos de operaciones entre imagen e imagen, como calcular la posición de los objetos, “pintar” la pantalla, manejar inteligencia artificial y demás, y para ello es necesario dejarle el tiempo suficiente, por eso, cuantas más imágenes por segundo tengamos, menos tiempo tiene el ordenador para realizar todas las operaciones y más rápido debe ser, de lo contrario, el juego se detendría hasta que se hubieran completado todas las operaciones, es lo que normalmente se le llama “ralentización”.

A la hora de diseñar un videojuego es muy importante tener presente en qué tipo de ordenadores se va a poder jugar: si la potencia no es un problema (están destinados a unas pedazo de máquinas que cuestan 1200€) podemos poner un número elevado de imágenes por segundo, pero si es para ordenadores más modestos hay que reducir ese valor, incluso sacrificar algo de suavidad en los movimientos a favor de la velocidad general del juego. Yo te recomiendo que, a la hora de valorar la situación, te mantengas en unos valores entre 30 y 60 imágenes por segundo, aunque realmente, con las máquinas que hay ahora, los juegos que vas a realizar y el rendimiento general que están consiguiendo el equipo de desarrollo de Fénix, es fácil que puedas usar los 60 FPS en casi todos tus juegos sin problemas.

Para especificar un valor de fps, se utiliza la función *Set\_fps*. Si escribimos por ejemplo *Set\_fps(60,1)*, lo que estaremos diciendo al ordenador es que trabaje a 60 imágenes por segundo.

Quizás te llame la atención el segundo parámetro. Ese 1 es algo que puede ayudar al rendimiento, porque le da permiso al ordenador literalmente a “saltarse 1 frame”. Es decir, le dice al ordenador que en cada segundo, si le falta tiempo, puede ahorrarse una vez el “dibujar” en la pantalla, lo que puede provocar un salto en la secuencia de la imagen. Visualmente esto no es aceptable, pero darle un cierto margen al ordenador para que termine su trabajo a tiempo es algo más que recomendable, sobre todo si los equipos que van a ejecutar tu juego son más bien modestos. La cifra puede variar entre 0 (sin saltos) y el número de saltos por segundo que tú quieras.

Como nota adicional, si especificamos un valor de 0 FPS, le estamos diciendo al ordenador que ejecute el juego tan rápido como sea posible, de tal manera que cuanto más rápido sea el ordenador más ligero irá el juego (esta utilidad está especialmente recomendada para programas que deben mostrar resultados en pantalla cuanto más rápido mejor, como dibujar una gráfica o contar bucles, pero si se usa para juegos piensa en como irá en un Pentium a 133Mhz y luego en un Pentium IV a 1.7GHz). También, desde el mismo equipo de desarrollo de Fénix, se recomienda usar, al menos, un salto de frame (yo pondría tres) para mejorar el rendimiento general.

Para jugar un poco con esta función, recuperemos el primer programa que hicimos, el del scroll horizontal de un texto:

```
Program MiPrimerPrograma;
```

```
Private
```

```

        int mivar1;
End
Begin
    mivar1=10;
    while(mivar1<320)
        delete_text(0);
        mivar1=mivar1+2;
        write(0,mivar1,100,1,"¡Hola mundo!");
        frame;
    end
end

```

Vemos que aquí no hay ninguna función Set\_fps. Si no aparece, se toma por defecto el valor de 25 fps. Añadamos la orden, con un valor de 60 fps. ¿Qué es lo que ocurrirá?:

```

Program MiPrimerPrograma;
Private
    int mivar1;
End
Begin
    Set_fps(60,1);
    mivar1=10;
    while(mivar1<320)
        delete_text(0);
        mivar1=mivar1+2;
        write(0,mivar1,100,1,"¡Hola mundo!");
        frame;
    end
end

```

El scroll se mueve más rápido. Esto es lógico: estamos obligando a que se pinten más fotografías en cada segundo. Es decir, estamos forzando a que el ritmo de ejecución del programa sea tal que la orden frame; se ejecute 60 veces por segundo, o al menos 59. Es otra manera de decir que Set\_fps establece la velocidad del juego. Es fácil ver esto si ponemos que se vea un frame cada segundo:

```

Program MiPrimerPrograma;
Private
    int mivar1;
End
Begin
    Set_fps(1,1);
    mivar1=10;
    while(mivar1<320)
        delete_text(0);
        mivar1=mivar1+2;
        write(0,mivar1,100,1,"¡Hola mundo!");
        frame;
    endi

```

*end*

Vemos que el texto se mueve en cada segundo, porque es la frecuencia a la que se ejecuta FRAME.

Existe una manera de visualizar en todo momento a cuántos FPS funciona un programa mientras éste se está ejecutando (ya que este valor, según lo que hayamos puesto como segundo parámetro de `set_fps`, puede ir variando). Esto es interesante para detectar cuellos de botella en determinados momentos de la ejecución de un programa debido al uso excesivo de recursos de la máquina (memoria, disco, CPU...) por parte de éste. Simplemente tienes que poner en cualquier sitio entre el BEGIN/END del programa principal -aunque se recomienda al principio de todo, justo después de `set_fps`-:

```
write_var(0,100,100,4,fps);
```

Lo que estamos haciendo simplemente es imprimir por pantalla el valor actualizado de una variable, llamada obligatoriamente "FPS", que nos marcará precisamente lo que queremos: a cada frame nos dirá a cuántos FPS está funcionando nuestro programa. Esta variable NO LA TIENES que declarar: ya viene declarada por defecto en Fénix: es decir, ya está lista para usar (dentro de `write_var` como hemos puesto o en cualquier otro sitio) y no tienes que hacer nada más. Esta variable es una (entre otras muchas) de las llamadas "variables predefinidas" de Fénix, y en concreto, es una variable predefinida global. Tranquilo: sobre los tipos de variables ya hablaremos en el capítulo siguiente.

La función `set_fps` la puedes invocar en cualquier punto de tu código, con lo que podrás alterar la frecuencia de frames por segundo según convenga (si el personaje está en una fase u otra del juego,etc).

### **Concepto de FPG. La aplicación “FPGEdit” y la utilidad de línea de comandos “fpg.exe”:**

Todavía no hemos pintado ningún gráfico. Ya es hora. Antes de nada, hay que disponer de los gráficos a utilizar. Éstos se pueden crear, ya lo hemos visto, de múltiples maneras. Aconsejo utilizar el Inkscape para creación de gráficos vectoriales muy eficaces o el Gimp para mapas de bits. Si no los tienes instalado, puedes usar cualquier otro programa. A malas, siempre podrás usar el limitado MSPaint, que ya es suficiente para hacer nuestras pruebas.

Una vez que tenemos los gráficos, ¿cómo los incluimos en el código de nuestro programa para que se muestren? Hay dos maneras: o referenciar uno a uno cada gráfico que utilicemos en el juego de forma independiente, o haber generado antes un fichero único que sirva de paquete contenedor de los gráficos, de manera que solamente se referencie en el código ese archivo, a partir del cual se pueden localizar los gráficos individuales. Este archivo contenedor ha de tener la extensión FPG. Así que el FPG es como un cajón, un almacén donde se guardan todos los gráficos. Ambas soluciones son válidas, aunque cuando el número de gráficos empieza a ser elevado, es conveniente irlos empaquetando en unos pocos FPG dependiendo de su función específica dentro del juego, para tener una clasificación más o menos clara de los gráficos: así, tendríamos los archivos `fondos.fpg`, `enemigos.fpg`, etc conteniendo los distintos gráficos de fondos, enemigos, etc respectivamente.

La pregunta que surge ahora es: si ya se tienen los gráficos, y si quieren utilizar ficheros `fpg`, ¿cómo se crean éstos? Existen varias aplicaciones que realizan este proceso, e incluso se la puede programar uno mismo con Fénix, pero la aplicación más extendida y solvente actualmente es “FPGEdit” disponible en la web de recursos de Fénix, FenixWorld (<http://fenixworld.se32.com>), en concreto en el enlace <http://fenixworld.se32.com/download.php?view.36>, o también en SourceForge: <http://osdn.dl.sourceforge.net/sourceforge/cdiv> ó <http://cdiv.sourceforge.net/html/down/down.htm>. También se encuentra incluido en el Fenix Pack (<http://fenixpack.blogspot.com>)

El FPGEdit es muy sencillo de utilizar: basta recorrer el disco duro mediante el árbol que aparece en la zona superior izquierda de la ventana del programa en busca de las imágenes que queremos incluir en el FPG. A medida que nos movamos por las diferentes carpetas, irán apareciendo listadas las imágenes incluidas en su interior, en la parte superior derecha de la ventana del programa. Seguidamente, hemos de decirle que queremos crear un nuevo fichero FPG e introducir en él los gráficos que deseemos. Esto se hace utilizando la parte inferior de la ventana del programa, donde crearemos el FPG. En el cuadro que aparece cuando queremos crear un FPG nuevo, escogeremos la ruta donde se grabará el archivo FPG con el nombre que queramos y como tipo de FPG el de 16 bits de Fénix (¡esto es muy importante!) y a partir de ahí iremos seleccionando desde la zona superior derecha de la ventana los gráficos que deseamos incluir (se pueden seleccionar varios manteniendo pulsada la tecla CTRL), arrastrándolos y soltándolos en la zona inferior de la ventana

Fíjate, es importante, que cada gráfico, una vez incluido en el FPG, tendrá asociado un número de tres cifras. Este número será el identificador único de ese gráfico en aquel código fuente donde se utilice el FPG. Es decir, que cuando se quiera hacer referencia a ese gráfico en el código, aparte de decir en qué FPG está incluido, habrá que decir cuál es el número concreto que corresponde al gráfico buscado. Como podrás deducir, "sólo" caben hasta 999 gráficos en un FPG (el 000 no vale), y puedes comprobar que ese código lo puedes cambiar según tus intereses (con el botón "Edit" de la barra superior de la zona inferior de la ventana), siempre que mantengas que es un número único para ese FPG. Una vez hayamos seleccionado los gráficos que queríamos, hemos de guardar el fichero FPG. Y siempre podremos abrir un archivo FPG ya creado para editarlo: quitar o poner nuevas imágenes o cambiar sus códigos identificativos, etc.

Si entras en la carpeta donde tienes guardados el intérprete y el compilador de Fénix ("fxi.exe" y "fxc.exe") verás que además hay dos ejecutables más: "map.exe" y "fpg.exe". El primero sirve para manipular ficheros gráficos de tipo MAP, que no utilizaremos en este manual, pero el segundo sí que es una utilidad interesante.

"Fpg.exe" permite hacer operaciones básicas de edición de un fichero FPG, como crear un nuevo archivo FPG, añadir imágenes dentro él, extraerlas, etc. Es algo así como un "FPGEdit" que viene de serie, pero sin tantas opciones (ni mucho menos), y además, funciona en forma de línea de comandos, como el intérprete y el compilador. Evidentemente, el FPGEdit es mucho más cómodo, flexible y versátil de utilizar, pero no está de más conocer la funcionalidad básica de este ejecutable, el cual es oficial y forma parte de cada versión de Fénix (cosa que no pasa con el FPGEdit)

Para probar este ejecutable, lo más fácil será que creamos una carpeta concreta en cuyo interior almacenaremos las imágenes que van a formar parte del FPG, y donde se guardará también el FPG creado. En el ejemplo la ruta de esta carpeta será "C:\hola".

Para crear un nuevo FPG de 16 bits vacío: nos situamos en la carpeta de instalación de Fénix y escribimos

**FPG -n "C:\hola\mifpg.fpg"**

donde la ruta entre comillas indica donde guardaremos el archivo recién creado, y su nombre (con extensión)

Una vez creado el FPG, queremos introducir varias imágenes en su interior. Para ello hacemos:

**FPG -a "C:\hola\mifpg.fpg" "C:\hola\a.png" "C:\hola\a.png" "C:\hola\a.png"**

La primera ruta es la del FPG de destino, y todas las rutas siguientes a partir de esta primera, indican las rutas completas de las diferentes imágenes que se van a introducir en el FPG destino. El orden con el que se escriban las rutas de las imágenes definirá su código interno dentro del FPG (001,002, etc).

Puedes comprobar que el fichero FPG contiene lo que se espera si lo abres por ejemplo con el FPGEdit.

En vez de haber especificado los nombres de cada una de las imágenes a introducir, también se podrían haber utilizado los comodines que ofrece el S.O Windows para el tratamiento múltiple de ficheros (el comodín \*, el comodín ?, el comodín [ - ]), aunque en ese caso las rutas de las imágenes no han de ir entre comillas. No obstante, este tema se sale un poco del ámbito de este manual y no profundizaremos en ello: baste comentar que la línea anterior sería equivalente a ésta:

**FPG -a "C:\hola\mifpg.fpg" C:\hola\\*.png**

Hay que tener muy presente que la opción "-a" siempre añadirá las imágenes al FPG desde el principio, partiendo del código 001. Así que sobrescribirá (eliminándolas) las imágenes que ya pudieran haber anteriormente en esas posiciones.

No es necesario crear el archivo FPG vacío primero y luego introducir las imágenes: se puede hacer las dos cosas a la vez, así:

**FPG -n "C:\hola\mifpg2.fpg" "C:\hola\a.png" "C:\hola\a.png" "C:\hola\a.png"**

Otro caso frecuente es el de eliminar una imagen del interior del FPG. Para ello se ha de especificar el código identificador que tiene esta imagen dentro del contenedor, así:

**FPG -d "C:\hola\mifpg.fpg" 2**

Con esta orden se eliminará la imagen con código 002 del interior del archivo FPG.

Si se quiere listar el contenido de un archivo FPG (es decir, saber qué imágenes contiene en su interior), se puede escribir indistintamente:

**FPG "C:\hola\mifpg.fpg" ó FPG -v "C:\hola\mifpg.fpg" ó FPG -l "C:\hola\mifpg.fpg"**

Si lo pruebas, verás que obtienes información sobre la ruta original de cada imagen, el código identificador que tienen éstas dentro del FPG, el tamaño en altxancho de cada imagen y el tipo de imagen que es -normalmente, "bitmap"-.

Para saber qué otras opciones ofrece este ejecutable, escribir: **FPG** a secas.

### **Carga de la imagen de fondo:**

Lo primero que vamos a hacer es utilizar una imagen PNG como imagen de fondo de pantalla. *Como he dicho, siempre tendremos la posibilidad de utilizar o bien imágenes individuales o bien imágenes incluidas dentro del contenedor FPG. En los ejemplos siguientes utilizaremos los dos sistemas para que los compares, pero posteriormente utilizaremos uno u otro indistintamente.*

Una vez que tengas creada la imagen de fondo, (y de hecho, todas las imágenes que vayas a utilizar en el juego), lo más cómodo es que la/s guardes en la misma carpeta donde tienes el PRG que estarás haciendo. Así no habrá problemas de rutas y se encontrarán las imágenes inmediatamente.

Si se supone que nuestra imagen PNG de fondo se llama "fondo.png", escribe el siguiente código:

*Program MiPrimerFondo;*

*private*

```

        int id1;
end
Begin
    set_mode(600,400,16);
    id1=load_png("fondo.png");
    put_screen(0,id1);
    loop
        frame;
    end
end

```

Fíjate que hemos hecho que el juego tenga una profundidad de 16 colores. A partir de ahora siempre trabajaremos así. De esta manera nos evitamos posibles problemas a la hora de utilizar PNGs que han sido creados con esta profundidad –que es lo habitual-, ya que si quisiéramos visualizar una imagen de 16 bits teniendo definida una profundidad de 8 bits, nos daría error.

Lo novedoso empieza en la línea `id1=load_png("fondo.png");`. Fíjate que esta función tiene un parámetro que es el nombre de la imagen que queremos mostrar. En realidad, esta función lo único que hace es cargar –de ahí su nombre- en memoria RAM dicha imagen, posibilitando así su manipulación posterior por parte del programa. Es importante saber que cualquier imagen o sonido o video con el que deseemos trabajar en nuestro código siempre tendremos que cargarlo previamente desde el disco duro a la RAM, mediante una función `load_XXX` –dependiendo de lo que carguemos-; esto es una condición indispensable. De hecho, de aquí viene la importancia de tener una memoria RAM lo más grande posible: para que quepan cuantas más imágenes, sonidos y videos mejor. Y también hay que tener claro que cargar una imagen en memoria no significa que se vaya a ver nada de nada: la carga es un proceso necesario pero que es interno del programa, en la pantalla no se va a ver nada.

Si la imagen no estuviera en la misma carpeta que el archivo DCB, tendríamos que poner como parámetro la ruta de la imagen a partir del lugar donde está el archivo DCB, en vez de sólo el nombre. Es decir, que si tenemos por ejemplo una carpeta “Juego” que contiene tanto el archivo DCB como una carpeta llamada “imágenes” donde está a su vez nuestro “fondo.png deberíamos poner

```
id1=load_png("imagenes\fondo.png");
```

o bien, de forma alternativa, poner toda la ruta desde la unidad donde estemos –la ruta absoluta que se dice-. Si tenemos por ejemplo la carpeta “Juego” en la raíz C:, pues sería:

```
id1=load_png("C:\Juego\imagenes\fondo.png");
```

Puedes observar que esta función devuelve un valor entero, que se lo asignamos a una variable creada por nosotros llamada “id1”. Este valor entero es un identificador único, que nos servirá para referenciar en cualquier momento del código a la imagen que hemos cargado. Es decir, a partir de ahora, para manipular y gestionar la imagen “fondo.png nos referiremos a ella mediante esa variable “id1 que contendrá el “DNI” particular e intransferible de dicha imagen: en vez de nombrarla cuando sea necesario con su nombre, la nombraremos con ese número identificador, (el cual pocas veces nos interesará saber cuál es en concreto y simplemente lo asignaremos a la variable entera con la que se trabajará).

Puedes ver también que rápidamente utilizamos la variable “id1 porque en la línea siguiente aparece la función `put_screen`, con dos parámetros, que valen 0 e “id1” respectivamente. Esta función lo que hace es pintar sobre el fondo un gráfico que no cambia ni se mueve. Ese gráfico viene dado por el segundo parámetro, el cual es el valor numérico identificativo del PNG que acabamos de cargar. El primer

parámetro siempre ha de valer 0 si lo que queremos es visualizar una imagen cargada individualmente, tal como hemos hecho; si hubiéramos cargado un archivo FPG –ahora lo veremos-, este primer parámetro ha de valer otra cosa.

La función `put_screen` no necesita que indiques la posición del gráfico, pues este siempre se pondrá en el centro de la pantalla, independientemente de la resolución, por eso normalmente la imagen usada con esta función ocupa toda la pantalla o gran parte de ella. Si no llegara a ocupar toda la pantalla lo que falta se rellena en negro.

Ya que estamos hablando de la función `put_screen`, aprovecho para comentar la existencia de la función `clear_screen`. Esta función, que no tiene ningún parámetro, borra el gráfico del fondo de la pantalla. Puedes comprobarlo ejecutando este pequeño ejemplo:

```
Program Test_ClearScreen;
private
  int id1;
end
Begin
  set_mode(600,400,16);
  set_fps(1,0);
  id1=load_png("fondo.png");
  put_screen(0,id1);
  frame;
  clear_screen();
  frame;
  put_screen(0,id1);
  frame;
  clear_screen();
  frame;
  put_screen(0,id1);
  loop
    frame;
  end
end
```

Ahora vamos a hacer lo mismo que veníamos haciendo hasta ahora -es decir, poner un gráfico de fondo y ya está-, pero en vez de cargar la imagen individualmente con `load_png`, vamos a incluirla dentro de un fichero FPG –que de momento sólo contendrá esta imagen-, y lo que vamos a hacer es cargar el fichero FPG, con lo que todas las imágenes que contuviera quedarían cargadas automáticamente y no se tendría que ir cargándolas una a una.

Lo primero que hay que hacer es crear el archivo FPG con FPGEdit. Lo nombraremos “imágenes.fpg” e incluiremos “fondo.png” con el código 001. Seguidamente, escribimos este programa:

```
Program MiSegundoFondo;

Private
  int id2;
end
Begin
  set_mode(640,480,16);
  id2=load_fpg("prueba.fpg");
```

```

    put_screen(id2,1);
  Loop
    Frame;
  End
end

```

Lo que hacemos aquí es utilizar la función `load_fpg`, la cual es equivalente a `load_png`, pero en vez de cargar en memoria un archivo PNG, carga un archivo FPG. Sólo a partir de entonces podrán estar accesibles en el código las imágenes que contiene. Igual que antes, esta función devuelve un número entero –aleatorio–, que se recoge en la variable `id2`, que identificará al archivo FPG unívocamente a lo largo del programa para lo que sea necesario. Y como antes también, cargar un archivo de imágenes no quiere decir que se vea nada de nada. Para ello hay que decirle más cosas al ordenador. Si queremos por ejemplo, volver a poner “fondo.png” como imagen de fondo estática, hemos de volver pues a utilizar la función `put_screen`.

La diferencia con el ejemplo anterior está en que ahora el primer parámetro no vale 0 sino el identificador devuelto por `load_fpg`. Siempre que carguemos el FPG, cuando queramos hacer referencia a alguna de las imágenes que contiene, tendremos primero que indicar en qué FPG se encuentra, y posteriormente decir qué código de imagen tiene dentro de él. Es lo que hacemos aquí: decimos que queremos cargar una imagen que está en el fichero identificado por “id2 cuyo código interno es 001 (el valor del segundo parámetro). Es por esta razón que cuando cargamos las imágenes directamente, el primer parámetro vale 0, porque no necesitamos referenciar a ningún FPG.

### **Descarga de imágenes:**

Acabamos de decir que para poder utilizar imágenes en nuestro juego hay que cargarlas previamente en memoria RAM. Pero la RAM es finita. Si empezáramos a cargar y cargar imágenes sin control, al final no tendríamos suficiente RAM disponible y nuestro juego se colgaría. Para evitar esto, es necesario descargar las imágenes que ya no estén siendo utilizadas, para dejar libre espacio de RAM que pueda ser usado otra vez.

Antes de continuar, he de decir que no he logrado descubrir la manera de descargar las imágenes si se cargan de manera individual con `load_png`. Parece ser que no existe una función “`unload_png`” (?) que realice esta tarea, así que por el momento solamente podremos descargar las imágenes si las cargamos con `load_fpg`, mediante la función “`unload_fpg`”. Este hecho es un punto a favor más para que la carga de imágenes se realice mediante FPGs y no por separado. No obstante, hay que tener en cuenta que si se descarga un FPG, ninguna de las imágenes incluidas en él se podrán utilizar en el programa, por lo que hay que estar seguro de que ninguna imagen del FPG será ya más utilizada.

Para aprender cómo se descargan las imágenes cargadas mediante un FPG con `load_fpg`, primero modificaremos el ejemplo que acabamos de hacer en el apartado anterior. Modifícalo así:

```

Program MiSegundoFondo;

Private
  int id2;
end
Begin
  set_mode(640,480,16);
  id2=load_fpg("prueba.fpg");
  put_screen(id2,1);
end

```

```

Loop
    if (key(_esc))
        break;
    end
Frame;
End
end

```

Fíjate que lo único que hemos hecho ha sido añadir el siguiente if:

```

if (key(_esc))
    break;
end

```

*(Este if también se podría haber escrito en una sola línea, así: if(key(\_esc)) break; end , para facilitar la lectura del código. De hecho, el salto de línea en el código fuente, Fénix no lo tiene en cuenta nunca para nada: es como si no estuviera).*

Lo que dice este if es que si es cierta la condición que contiene entre paréntesis (ahora veremos qué es), se ha de salir del LOOP. Al salir del LOOP, de hecho se acaba el programa. Por lo que la condición del if que hemos puesto sirve para salir del programa en el momento que “key(\_esc)” sea cierto. Mientras que no lo sea, el if no se ejecutará y el LOOP irá funcionando.

¿Y qué quiere decir esto de key(\_esc)? Pues key es una función, la cual devuelve o “verdadero” o “falso” – en realidad, 1 ó 0 respectivamente- si se ha pulsado una determinada tecla del teclado, o no. Es decir, key(\_esc) devolverá “verdadero” -1- si se ha pulsado la tecla ESC, y “falso” -0- si no se ha pulsado.

Por lo tanto, lo que hace el if es comprobar, a cada iteración de LOOP, si se ha pulsado la tecla ESC. Si no se ha pulsado, el if no se hace y se ejecuta FRAME y el programa continúa. Si llega un momento en que sí se ha pulsado ESC, entonces la condición key(\_esc) es verdadera, se ejecuta el interior del if y por tanto se sale del bucle y se acaba el programa. ¡Acabamos de descubrir una nueva forma de acabar el programa que no sea dándole al botón de Cerrar de la ventana!: pulsando una tecla determinada podemos hacer lo mismo.

Evidentemente, la función key no sólo funciona con la tecla ESC. Aquí tienes la lista de posibles teclas (son fáciles de deducir) que la función key puede tratar:

_ESC	_J	_SCROLL_LOCK
_1	_K	_HOME
_2	_L	_UP
_3	_SEMICOLON	_PGUP
_4	_APOSTROPHE	_C_MINUS
_5	_WAVE	_LEFT
_6	_L_SHIFT	_C_CENTER
_7	_BACKSLASH	_RIGHT
_8	_Z	_C_RIGHT
_9	_X	_C_PLUS
_0	_C	_END
_MINUS	_V	_DOWN
_PLUS	_B	_PGDN
_BACKSPACE	_N	_INS
_TAB	_M	_DEL
_Q	_COMMA	_F11
_W	_POINT	_F12

<code>_E</code>	<code>_SLASH</code>	<code>_LESS</code>
<code>_R</code>	<code>_C_BACKSLASH</code>	<code>_EQUALS</code>
<code>_T</code>	<code>_R_SHIFT</code>	<code>_GREATER</code>
<code>_Y</code>	<code>_PRN_SCR</code>	<code>_ASTERISK</code>
<code>_U</code>	<code>_ALT</code>	<code>_R_ALT</code>
<code>_I</code>	<code>_SPACE</code>	<code>_R_CONTROL</code>
<code>_O</code>	<code>_CAPS_LOCK</code>	<code>_L_ALT</code>
<code>_P</code>	<code>_F1</code>	<code>_L_CONTROL</code>
<code>_L_BRACKET</code>	<code>_F2</code>	<code>_MENU</code>
<code>_R_BRACKET</code>	<code>_F3</code>	<code>_L_WINDOWS</code>
<code>_ENTER</code>	<code>_F4</code>	<code>_R_WINDOWS</code>
<code>_CONTROL</code>	<code>_F5</code>	
<code>_A</code>	<code>_F6</code>	
<code>_S</code>	<code>_F7</code>	
<code>_D</code>	<code>_F8</code>	
<code>_F</code>	<code>_F9</code>	
<code>_G</code>	<code>_F10</code>	
<code>_H</code>	<code>_NUM_LOCK</code>	

¿Y por qué explico todo este rollo? Porque con el añadido del if, podemos parar el programa por código cuando deseemos. Y justo en ese momento, antes de salir de la ejecución, insertaremos la función de descarga de imágenes, para acabar el programa limpiamente sin dejar ningún residuo en la memoria RAM. Para lograr esto, lo que debemos hacer es utilizar la función de descarga `unload_fpg`.

*Program MiSegundoFondo;*

*Private*

*int id2;*

*end*

*Begin*

*set\_mode(640,480,16);*

*id2=load\_fpg("prueba.fpg");*

*put\_screen(id2,1);*

*Loop*

*if (key(\_esc))*

*break;*

*end*

*Frame;*

*End*

*Unload\_fpg(id2)*

*end*

Se ve que una vez salido del bucle, y justo antes de acabar el programa se descarga el FPG que hayamos utilizado. Para ello, la función `unload_fpg` requiere un parámetro, que vuelve a ser el identificador del fichero FPG que se quiere descargar entero. Y ya está.

Prueba por último este otro código curioso. A ver si sabes qué es lo que ocurre; (necesitarás para poderlo ejecutar un archivo FPG con cuatro imágenes en su interior como mínimo).

*program ejemplo2;*

*private*

*int id1;*

```

end
begin
    set_mode(640,480,16);
    id1=load_fpg("graficos.fpg");
    loop
        put_screen(id1,rand(1,4));
        frame;
    end
end
end

```

### **Iconos. Set\_icon(). Editores de iconos y de recursos:**

Un icono es, como sabes, un tipo de imagen con unas características muy concretas. Suelen ser imágenes de un tamaño reducido (16x16,32x32,48x48 píxeles) que se utilizan por ejemplo como indicadores de acciones en una barra de botones de una aplicación, o aparecen en un listado de una carpeta de Windows, etc. Este tipo de imágenes tiene un formato propio (diferentes del PNG) y normalmente tiene la extensión \*.ICO.

*(Además, en Windows también existen un tipo de imágenes con un formato muy parecido que es formato utilizado para mostrar cursores de ratón: son los ficheros con extensión CUR, pero en Fénix no se trabaja con éstos).*

No obstante, la mayoría de iconos que se pueden ver en Windows (el icono “carpeta”, el icono “documento de texto”, el icono “Mi PC”, etc) no son archivos ICO. ¿Por qué? Porque se utiliza otro sistema para visualizar iconos.

Cualquier ejecutable (y por ejecutable se entiende un fichero con extensión EXE o bien DLL), a parte de incluir el código compilado en binario del propio programa, tiene un apartado donde, dentro del mismo fichero, almacena uno o más iconos. ¿Y esto por qué? Porque si te has fijado, cada programa tiene un icono distintivo: sería una lata que además del archivo EXE (o DLL) el programador que ha creado el juego tuviera que acompañarlo de uno o más archivos ICO, con la posibilidad de que se borrara alguno accidentalmente y quedara pues una referencia en el ejecutable a ese archivo inválida. La alternativa es incorporar dentro del propio fichero EXE (o DLL) los iconos, integrándolos en el código binario como una bacteria fagocitada por una ameba. De esta manera, se tiene sólo el archivo ejecutable con todos los iconos que usará, en su interior.

Y respecto los iconos de Windows, la mayoría utilizan este sistema: no son archivos ICO independientes sino que están integrados dentro de los distintos ejecutables del sistema. Haz una prueba: abre el Explorador de Windows y en el menú “Herramientas” selecciona “Opciones de carpeta”. Allí, elige la pestaña “Tipo de fichero” y clicas en el botón “Avanzadas”. En la ventana que sale, clicas en el botón “Cambio de icono...” y allí podrás navegar y elegir cualquier archivo EXE o DLL del sistema. Verás que aparecerán todos los iconos que existen en el interior de éstos. Concretamente, los iconos estándares del escritorio de Windows están en el archivo “shell32.dll”, dentro de la carpeta “System32” del sistema. También hay iconos chulos en el archivo “moricons.dll”.

Anteriormente comenté que si queremos que en la barra de títulos (es decir, la barra de color azul que aparece en la parte superior de la ventana) de nuestro juego –si éste no se ejecuta a pantalla completa– aparezca otro icono diferente que no sea el del fénix naranja, tal como he dicho deberemos utilizar la función `set_icon`. Esta función tiene dos parámetros: el primero es la librería FPG que contiene el gráfico (es obligatorio utilizar alguna librería FPG donde esté incluido el icono: los iconos no se pueden cargar independientemente, entre otras cosas porque no existe ninguna función del tipo “load\_ico”) y el segundo es el código del gráfico dentro del FPG. Posiblemente te sorprenda que podamos incluir un icono dentro de

un FPG. El formato ICO es un formato plenamente aceptado en Fénix, y por tanto, no hay ningún problema en utilizar este otro formato de imágenes diferente del PNG, y por tanto, en incluirlos dentro de cualquier FPG. Incluso se puede perfectamente incluir en un mismo FPG imágenes en formato PNG y en formato ICO indistintamente.

Recuerda que es necesario llamar a esta función antes que llamar a *set\_mode*, ya que ésta es la función que creará y visualizará dicha ventana. Importante: para esta función, debes utilizar obligatoriamente sólo un gráfico de 32x32 pixels. El gráfico puede estar a 8 ó 16 bits de color, y no debes nunca borrar este gráfico de memoria usando funciones como *unload\_map* o *unload\_fpg*. No es posible cambiar el icono de una ventana ya inicializada, pero se puede volver a llamar a *set\_mode* para activar los cambios.

Por ejemplo: si tenemos un archivo llamado “*iconos.fpg*” que representa un FPG que incluirá el icono de nuestro juego, el cual tendrá el código 001, el programa necesario para mostrar el icono deseado en la barra de títulos sería algo parecido a esto:

*Program ejemplo;*

*Private*

*Int iconfpg;*

*End*

*Begin*

*Iconfpg=load\_fpg(“iconos.fpg”);*

*Set\_title(“Prueba”);*

*Set\_icon(iconfpg,1);*

*Set\_mode(640,480,MODE\_16BITS,MODE\_MODAL);*

*Loop*

*Frame;*

*end*

*End*

En este momento te surgirá la duda de saber cómo se pueden conseguir y crear iconos. Se pueden utilizar los editores gráficos mencionados en el primer capítulo, pero, no obstante, existen en el mercado editores específicos de iconos (y cursores) que permiten personalizar al máximo el diseño y comportamiento de los iconos que queramos crear desde cero o bien modificar a partir de alguno ya existente.

Algunos editores de iconos podrían ser:

**MicroAngelo** (<http://www.microangelo.us>): El editor de iconos líder. El más versátil, potente, funcional y flexible.

**IconArt** (<http://www.conware-pro.com>): Freeware

**LiquidIcon** (<http://www.x2studios.com>): Freeware

**IconEdit** (<http://www.iconedit2.com>)

**Awicons** (<http://www.awicons.com>)

**IconCoolEditor** (<http://www.iconcool.com>)

**IconPackager**, **Image Icon Converter**, **IconCoolEditor**, **Coffe IconEditor** y muchos otros más: (<http://www.popularshareware.com/010-rating-1.html>)

Después de todo lo explicado, y después de haber probado el código de ejemplo anterior, es posible que hagas algún juego utilizando *set\_icon* y lo quieras distribuir. Tal como comenté en un capítulo anterior, para ello hay que renombrar el FXI.exe con el nombre de tu juego. Pero queda un detalle pendiente y es que el icono de FXI.exe es un cuadrado mostrando un fénix naranja, pero tu querrás que el icono de tu ejecutable sea el icono que has diseñado tú y que aparece –eso sí– en la barra de título mientras el juego está en marcha. ¿Cómo se pueda cambiar el icono del FXI.exe entonces?

El icono del fénix naranja no es ningún icono ICO: está integrado dentro del propio fichero FXI.exe. Por lo tanto, la solución sería “destripar” el código binario del FXI.exe, “extirpar” el icono del fénix naranja e “implantar” nuestro icono dentro, para que el ejecutable resultante tuviera ahora sí una apariencia totalmente profesional. Y este proceso se puede realizar mediante utilidades llamadas editores de recursos.

Un recurso es un icono, o un cursor, o una cadena de texto que está integrada dentro de un ejecutable. Por tanto, con un editor de recursos podríamos modificar, suprimir o añadir iconos, cursores o frases a un ejecutable, sin alterar su correcto funcionamiento (puede ser una manera, por ejemplo, de traducir una aplicación: editando los recursos de cadenas de texto). Nosotros lo podríamos usar para lo que te he comentado: para explorar y localizar dentro del ejecutable el icono propio de FXI.exe y sustituirlo por el nuestro.

Editores de recursos pueden ser:

**Resource Explorer** (<http://www.sharewaresoft.com> o <http://www.brothersoft.com>, entre otros)

**PE Resource Explorer:** <http://www.wilsonc.demon.co.uk/d7resourceexplorer.htm>

**Resource Hunter e Icon Hunter:** <http://www.boilsoft.com/rchunter.html> e <http://www.boilsoft.com/iconhunter.html>

### **Los nuevos formatos: FGC, FBM y FPL:**

El lenguaje Fénix surgió a partir del lenguaje y entorno DIV. Una consecuencia de esa herencia es que los formatos gráficos con los que trabajaba DIV continúan siendo los formatos con los que trabaja Fénix actualmente. Estos formatos gráficos en concreto son tres: FPG, MAP y PAL.

El primero ya lo conocemos. El segundo es un formato de archivo gráfico con posibilidad de animación, como podría ser también el GIF o el PNG animado, pero con la particularidad de que el formato MAP es propio de DIV/Fénix, y éste lo utiliza internamente cuando carga gráficos en memoria y tiene que operar con ellos, por ejemplo. La razón de utilizar un formato propio internamente, y no PNG, por ejemplo, es para aprovechar las características nativas del lenguaje, especialmente diseñadas para apoyar la creación de videojuegos. Además, estos formatos están especialmente optimizados para guardar los gráficos a la misma calidad que se muestran en pantalla, por lo que suelen ocupar menos que otros formatos gráficos de uso habitual (salvo formatos que perjudican la calidad del gráfico y no suelen ser por lo tanto aptos para un videojuego donde debe preservarse cada gráfico pixel a pixel). Por último, el formato PAL es el formato de las paletas de color de 8 bits (hablaremos de qué son las paletas muy pronto) propio de DIV/Fénix.

No obstante, DIV pasó a ser propiedad legal de una empresa británica, Fastrack (actualmente en quiebra), que tenía en mente comercializar este lenguaje en el Reino Unido. Al tener ellos la licencia, puede ocurrir que en cualquier momento esa empresa demande a los desarrolladores de Fénix por seguir utilizando los mismos formatos gráficos que ellos, ya que legalmente ahora son de su propiedad. Así que para que la comunidad Fénix no incurra en un posible delito, se propuso el cambio de formatos en estas tres áreas: contenedores de imágenes, formatos de imágenes en sí y paletas, para así poder desarrollar sus propios ficheros independientes de DIV y desmarcarse por completo de su antecesor.

Así pues:

El sustituto del formato FPG es el llamado FGC (Fenix Graphic Container).  
El sustituto del formato MAP es el llamado FBM (Fenix BitMap)  
El sustituto del formato PAL es el llamado FPL (Fenix Palette)

No obstante, tenemos un problema. La utilidad "FPGEdit" de momento sólo es capaz de trabajar (abrir,editar,guardar) archivo con formato FPG. Y la utilidad "fpg.exe" de línea de comandos también. Así que provisionalmente nos veremos obligados a utilizar los formatos "antiguos", hasta que hayan disponibles utilidades que permitan trabajar con los nuevos, especialmente el FGC.

A raíz de estos cambios de formato, han habido ciertos cambios en algunas funciones del lenguaje Fénix que trabajaban con ellos. Los he resumido en la siguiente tabla (no he puesto las funciones que trabajan con paletas PAL y FPL ya que en este manual no utilizarán):

<b>FUNCIONES "ANTIGUAS" (todavía operativas)</b>	<b>FUNCIONES "NUEVAS"</b>	<b>OBSERVACIONES</b>
Load_fpg()	Load_fgc()	Incompatibles. La primera sólo es capaz de cargar en memoria un archivo en disco con formato FPG y la segunda lo mismo pero con archivos con formato FGC
Unload_fpg()	Unload_fgc()	Son funciones equivalentes (sinónimas)
Load_map()	Load_fbm()	Cargan en memoria un archivo de disco con formato MAP o FBM, respectivamente. No se verán en este manual, en favor de otras funciones como Load_png() o Load_fpg()/Load_fgc().
Unload_map()	Unload_fbm()	Son funciones equivalentes (sinónimas)
-	Save_fgc()	Graba en el disco duro un archivo con formato FGC. Es la única manera actual de generar ficheros de este tipo, para poder cargarlos posteriormente con Load_fgc() y poderlos usar en nuestros programas.
-	Save_fbm()	Graba en el disco duro un archivo con formato FBM. No se verá en este manual, en favor de otra función como Save_png(), que hace lo mismo pero en formato PNG.

**Uso de paletas de colores:**

Anteriormente hemos comentado de pasada la existencia de las llamadas paletas de colores, (en el primer capítulo se explicó qué es lo que son), e incluso hemos comentado que hay archivos en Fénix (los antiguos .PAL y los nuevos .FPL) que se dedican a guardarlas en disco. Pero en realidad, estas paletas sólo tiene sentido utilizarlas cuando trabajemos con gráficos de 8 bits, y como en este manual se trabajará siempre con gráficos de 16 bits, no profundizaremos en este tema. Por ejemplo, Fénix aporta muchas funciones para trabajar con paletas como CONVERT\_PALETTE, ROLL\_PALETTE, FIND\_COLOR, GET\_COLORS, SET\_COLORS, LOAD\_FPL, SAVE\_FPL,etc que no se explicarán. No obstante, merece la pena introducir el posible uso que podrían tener las paletas en el caso que se hicieran servir.

Acabamos de decir que las paletas se usan con modos graficos de 8 bits, en los cuales ya sabemos que como máximo se pueden usar 2<sup>8</sup> colores en pantalla = 256 colores. Y hay que tener cuidado, ya que sólo se puede tener una paleta cargada a la vez

Dado que este número de colores no es muy alto como para crear imagenes muy realistas (degradados de color suaves, etc), se suelen "elegir" los 256 colores que se van a utilizar para representar una imagen. Por eso, para mostrar una imagen de una selva, se usarían muchos tonos de verdes, algunos naranjas o rojos, y menos azules. O para representar un cielo,la mayoría de los colores serian azules, en sus

tonalidades desde el negro hasta el blanco, etc.

Cuando se trabaja en modo 16 bits no se necesitan paletas porque hay muchísimos más colores  $2^{16} = 65536$  colores. Entonces ya no tenemos que elegir colores :se pueden utilizar tantos que las imágenes tendrán buena calidad. En ese modo simplemente llamamos a los colores con la función RGB(r,g,b), por ejemplo, estableciendo la intensidad deseada de cada componente de color.

Si queremos que nuestros juegos vayan rápidos independientemente del ordenador en el que vayan a ejecutarse, una de las cosas que tendría sentido plantearnos es usar paletas de 8 bits (256 colores). Ya hemos dicho que al hacerlo tendremos que elegir bien estos 256 colores para sacarles el mayor partido posible: tendremos que construir nuestra paleta de 256 colores de acuerdo a los que vayamos a utilizar , mediante programas de edición de gráficos como el Gimp y con la ayuda de varios comandos de Fénix de manipulación de paletas que podrán ser usados durante la ejecución de un juego, bien para cambiarlas por completo, bien para cambiar colores individualmente, etc. De todas maneras, piensa que nada te impide usar varias paletas por turnos (por ejemplo, en un juego de plataformas con una 1ª fase de hielo -predominarán los blancos y los azules- y una 2ª de fuego -tonos rojos-; al pasar de la fase 1 a la 2 se descargaría de memoria (o simplemente se dejaría de utilizar) los gráficos (contenidos en un fpg, por ejemplo) de la 1ª y se cargaría los gráficos contenidos en otro fpg (y por consiguiente las paletas) de la 2ª.

Hemos dicho que podemos usar un editor gráfico para crear una paleta. Pero, ¿eso cómo se hace? En general no será necesario crear explícitamente ninguna paleta, porque cuando se crea una imagen de 8 bits, ésta ya incorpora dentro su propia paleta. También es verdad que la mayoría de programas gráficos dan la opción, a partir de una determinada imagen de 8 bits, de guardar las paletas por separado, para así al crear una nueva imagen se le pueda asociar alguna paleta guardada previamente.

El uso de 256 colores tiene otras ventajas, y no sólo el ahorro de memoria o mayor velocidad de ejecución: te permite tener control total sobre los colores que usas, pues manejar 256 colores es mas sencillo que los 65000 y pico; sobre todo porque cada "casilla" (es decir, cada color de la paleta) lo puedes modificar para conseguir efectos de brillos, cambios de color... e incluso, con conocimientos de los efectos del color, modificando una paleta entera puedes usar una misma imagen para representar el día, la noche y todas las tonalidades intermedias, cosa que en 16 bits puede ser algo tedioso porque tendrías que modificar todos los graficos, en lugar de 256 "casillas".

## CAPÍTULO 4: DATOS Y PROCESOS

### Concepto de proceso:

Se llama proceso a cada uno de los diferentes objetos que hay dentro de un juego. Por ejemplo, en un juego de marcianos será un proceso la nave que maneja el jugador, otro proceso cada enemigo, cada explosión, cada disparo, etc. Normalmente se equiparan a los diferentes gráficos en movimiento (también llamados sprites), aunque pueden existir procesos que no aparezcan en pantalla.

Los procesos son “partes” de código. Cada proceso tiene su propio código que se ejecuta y se relaciona con los demás procesos según sus órdenes concretas. A diferencia de otros lenguajes, Fénix se basa en programación concurrente; esto quiere decir que en lugar de seguir línea por línea el programa en un orden, éste se ejecuta en trozos que avanzan al mismo tiempo, los procesos. Esto al principio resulta confuso, pero una vez te acostumbras, ves que gracias a ello cada proceso actúa casi de forma independiente y es muy útil por ejemplo en juegos como donde cada personaje es un proceso independiente y así no tiene que esperar a que se ejecuten antes los demás.

De esta manera, podemos crear programas más flexibles muy fácilmente. Además, la detección de errores es también más sencilla, porque si todo funciona excepto el movimiento de tu nave, sabrás exactamente dónde tienes que buscar. Pero esta manera de codificar también exige una planificación previa con lápiz y papel y decidir antes de ponerse a teclear código como un loco cuántos procesos se van a crear, de qué tipo, etc, para tenerlo claro antes de perderse. No es broma: es muy importante no llegar al momento donde están en mitad del código pensando por qué no funciona sin tener ni idea...

Un proceso, o bien consta de un bucle (infinito o no), o bien sólo se ejecuta una vez hasta que llega a su fin y se termina.

Expliquémoslo con un ejemplo. Básicamente, el ordenador ejecuta los programas línea a línea, desde arriba hasta el final, parecido a leer un libro. Entonces, por ejemplo, si hacemos un videojuego de matar enemigos, sería algo así como:

*Principio:*

*Dibujar tu nave*

*Dibujar los enemigos*

*Dibujar los disparos*

*Chequear si hay colisiones*

*Si un disparo toca un enemigo, matarlo*

*Si un del enemigo te toca: matarte*

*Chequear si hay algún movimiento de joystick*

*Calcular la nueva posición acorde al movimiento del joystick*

*Crear un nuevo disparo si has apretado el botón de disparar*

*Calcular los movimientos de los enemigos*

*Ir al Principio.*

Este ejemplo es muy básico y espero que se entienda. Hace todo lo necesario desde arriba hasta abajo, y vuelve a empezar, haciendo lo mismo una y otra vez. Evidentemente, se necesitan más partes del programa, como concretar qué es lo que pasa cuando un enemigo o tú muere, pero este ejemplo sólo es para mostrarte la manera de programar que en Fénix NO se utiliza. ¿Por qué? Porque cuando el videojuego se vuelve cada vez más complicado, el bucle del programa se hace cada vez mayor y el código se vuelve pronto desordenado y caótico. La solución, ya digo, pasa por la multitarea (programación concurrente). Multitarea significa que más de un programa (los procesos) están ejecutándose al mismo tiempo. Así, en vez de un único bucle inmenso como el anterior, donde las líneas una tras otra se ejecutan para hacer todos los movimientos, colisiones, chequeos, etc, se tiene más programas/procesos pequeños que hacen todo esto

continuamente por separado.

Entonces, básicamente tendríamos los siguientes procesos funcionando al mismo tiempo para lograr lo mismo que con el ejemplo anterior:

*TuNave (joy polling, movimiento, dibujo, proceso\_creación\_disparos, chequeo\_colisiones)*

*Enemigo (movimiento, dibujo, chequeo\_colisiones)*

*Disparo (movimiento, dibujo, chequeo\_colisiones)*

En vez de tener un código grande y desordenado, se tiene tres códigos simples funcionando a la vez.

Otro detalle a tener en cuenta es que es posible ejecutar múltiples instancias de un proceso. Es decir, si cada enemigo y disparo es un proceso, podremos ejecutar ese proceso más de una vez y simultáneamente, por lo que tendríamos tantos enemigos o disparos en pantalla como procesos de ese tipo estuvieran funcionando a la vez, de forma independiente cada uno de ellos.

### **Las variables locales predefinidas GRAPH, FILE, X e Y:**

Hasta ahora hemos visto que para poder utilizar una variable cualquiera (sólo hemos usado variables privadas, pero para las locales y las globales pasaría lo mismo) le tenemos previamente que declarar, porque si no el programa no puede reconocer dicha variable en el código. Esto es así para todas las variables que nosotros creamos.

No obstante, Fénix dispone de un conjunto de variables (que son o globales o locales, pero nunca privadas o públicas) que existen y que pueden ser usadas en nuestro código sin ninguna necesidad de declararlas. Son variables listas para usar, predefinidas con un tipo de datos concreto –normalmente INT-, y están disponibles en cualquier programa Fénix. Evidentemente, nosotros no podemos crear variables que tengan el mismo nombre que alguna de las predefinidas existentes, porque si no habría confusiones en los nombres. Ya vimos en un capítulo anterior un ejemplo de este tipo de variables: TEXT\_Z.

¿Por qué existen estas variables predefinidas? Porque cada una de ellas realiza una tarea muy específica y especialmente encomendada a ella. Es decir, que estas variables predefinidas cada una de ellas existe por una razón determinada; y se tendrán que usar en consecuencia. No son variables generales como las que podamos crear nosotros, en las que diciendo que son de un tipo de datos determinado pueden almacenar cualquier cosa. Cada variable predefinida tiene un significado concreto, y los valores que pueda almacenar en consecuencia serán unos determinados. Las variables predefinidas se utilizan, básicamente, para controlar los diferentes dispositivos del ordenador (ratón, teclado, pantalla, tarjeta de sonido, joystick...) y los gráficos de los juegos.

A continuación vamos a conocer el significado de dos variables predefinidas (locales) enteras muy importantes, como son GRAPH y FILE.

Como primera aproximación, ya hemos dicho que podemos asumir que en un juego de matar enemigos, por ejemplo, cada enemigo representará un proceso diferente, los disparos otro, nuestra nave otro, etc. Lo primero que tendríamos que hacer para que todos estos procesos se vieran en pantalla es asociarles a cada uno de ellos una imagen: los enemigos tienen una imagen (o varias si queremos hacer enemigos diferentes), los disparos también, nuestra nave también, etc. Aquí es donde interviene la variable GRAPH.

Cuando creamos un proceso, la manera de asignarle una imagen es estableciendo un valor para la variable GRAPH dentro de dicho proceso. Cada proceso puede tener asignado un valor diferente a GRAPH de forma independiente: los distintos valores no se interferirán unos con otros porque el código de cada proceso se ejecuta aislado del de los demás, y la variable GRAPH guarda la asociación de los distintos valores que puede tener en los diferentes procesos con el proceso al que le corresponde (de hecho, esto que

acabo de decir es la definición de variable local, que ya la veremos más adelante). El valor que puede tener esta variable GRAPH es un número entero, que resulta ser el identificador devuelto por load\_png que hace referencia a la imagen asociada a ese proceso.

Si en vez de utilizar load\_png utilizamos load\_fpg, aparte de GRAPH necesitaríamos otra variable local predefinida entera: FILE. Esta variable toma como valor el identificador del archivo FPG cargado mediante load\_fpg, de tal manera que entonces GRAPH pasa a valor el código numérico interno que tiene la imagen buscada dentro del archivo FPG. Así pues, para definir la imagen de un proceso que ha sido cargada con load\_fpg, tendríamos que indicarle en qué archivo FPG se encuentra, pasándole el identificador de ese archivo a la variable FILE, y dentro del FPG, de qué imagen se trata, pasándole el código numérico interno a la variable GRAPH.

Además de GRAPH y FILE, existen otras dos variables locales predefinidas más de las que querría decir cuatro cosas ahora: son la variable X y la variable Y. Éstas tienen como valor la coordenada x e y respectivamente del centro de la imagen dada por GRAPH (y FILE en su caso). Es decir, sirven para establecer las coordenadas del centro de la imagen asociada al proceso donde se definen. Igual que GRAPH, cada proceso puede dar valores diferentes a X e Y de forma independiente unos de otros porque cada proceso no interfiere – a priori- en los demás: es como si cada proceso tuviera una copia particular de las variables X e Y con sus valores propios.

Supongo que toda esta parrafada habrá sido un poco difícil de digerir. Pronto veremos ejemplos de todo esto, donde se verá mucho más claro lo explicado arriba.

### **Creación de un proceso:**

Cada proceso debe empezar con la palabra reservada PROCESS, el nombre del proceso y la lista entre paréntesis de parámetros –especificando su tipo- que puede tener, si es que tiene. Los parámetros son una lista de variables en los que el proceso va a recibir diferentes valores cada vez que sea invocado (llamado o utilizado desde otro proceso. Los paréntesis son obligatorios incluso cuando el proceso no tenga parámetros. Seguidamente viene la declaración de variables que necesite, las cuales sólo podrán ser del tipo “privadas” –las que venimos usando hasta ahora-, y luego un BEGIN, el código del proceso y un END. Es decir:

```
PROCESS NombreProceso ( tipo param1, tipo param2,...)
  PRIVATE
    Declaraciones de variables privadas;
  END
  BEGIN
    Código del proceso;
  END
```

Fíjate que la estructura de un proceso se parece mucho a lo que hemos visto hasta ahora, que es la estructura del programa principal. De hecho, el programa principal no es más que otro proceso más.

Ese proceso puede ejecutar todas las órdenes que se quieran, llamar a otros procesos, "matarlos" (hacer que dejen de ejecutar su código), congelarlos (hacer que paren de ejecutar su código, pero sin "matarlo"), dormirlos (congelarlo haciendo que su gráfico no aparezca en pantalla), interactuar con ellos y cualquier otra cosa que se nos ocurra. Como hemos dicho, se puede hacer más de una llamada a un proceso concreto (ej.: cada vez que el protagonista tenga que disparar se llama al mismo proceso disparo, ahorrando muchísimas líneas de código). Y pueden existir infinidad de procesos a la vez, cada uno ejecutando su código único (lo que puede ralentizar el ordenador si hay demasiados, pero nada es perfecto).

Es importante diferenciar entre los bloques PROCESS de los programas, que definen el comportamiento de un tipo concreto de proceso, y lo que es un proceso del juego en tiempo de ejecución, que es un objeto del juego cuyo comportamiento estará regido por uno de los bloques PROCESS del programa, según del tipo que sea (lo que se llama técnicamente "instancia" de un proceso).

El concepto de los procesos espero que lo hayas entendido, pero ahora te surgirá la duda de, ¿cómo insertar el código de los procesos dentro del programa principal PROGRAM, que conocemos tan bien? (De hecho, el programa principal no es más que otro proceso más del programa, el primer proceso). ¿Y dónde aparecen esas variables GRAPH y FILE comentadas antes?

He aquí un ejemplo, que es vital entenderlo para poder continuar, así que atento (necesitaremos un archivo FPG llamado "imágenes.fpg" con una imagen –de 30x30 estaría bien- en su interior de código 1):

```
PROGRAM ejemplo_procesos;
GLOBAL
  INT id1;
END
BEGIN
  set_mode(640,480,16,MODE_FULLSCREEN);
  id1=load_fpg("imagenes.fpg");
  personaje();
END
```

```
PROCESS personaje()
BEGIN
  x=320; y=240; file=id1; graph=1;
  LOOP
    IF (key(_up)) y=y-10; END
    IF (key(_down)) y=y+10; END
    IF (key(_left)) x=x-10; END
    IF (key(_right)) x=x+10; END
    FRAME;
  END
END
```

Fíjate que el código se estructura en dos partes, que son los dos procesos de los que consta el programa. La primera parte es la que ya conocemos, que es el proceso correspondiente al programa principal, y la segunda a un proceso llamado "personaje" que hemos creado. Si hubiera más procesos, la estructura se repetiría con los correspondientes bloques PROCESS/END para cada proceso extra.

Lo primero que salta a la vista es que, a diferencia de hasta ahora, hemos declarado una variable en el programa principal de tipo "global". Esto es necesario para que funcione, (si fuera privada no lo haría), pero el por qué lo explicaré dentro de poco, ahora no es relevante.

Seguidamente puedes observar ponemos la resolución y la profundidad del juego y decimos que el programa se vea a pantalla completa, cargamos un archivo FPG al cual le damos el identificador "id1", y finalmente, y aquí está la novedad, llamamos –o mejor dicho, creamos- al proceso "personaje".

Fíjate que la manera de crear un proceso es simplemente escribir su nombre, y si tuviera parámetros, poner los valores que queramos que tengan a la hora de la creación del proceso. En este caso, "personaje" es un proceso sin parámetros, y por tanto, sólo se ponen los paréntesis vacíos. Al crearse este proceso, si éste es visible porque está asociado a un gráfico, se verá ese gráfico por pantalla. Si se volviera a ejecutar la línea *personaje()*; otra vez (porque estuviera dentro de un bucle), se crearía otro proceso idéntico en el juego y por tanto tendríamos dos procesos "personaje" funcionando, y por tanto, dos imágenes iguales en

pantalla. Y así. Como ves, es una forma fácil de crear muchos enemigos a atacar!

Pero, ¿qué es lo que ocurre exactamente cuando se crea un proceso? Pues que se comienza a ejecutar el código que hay entre el BEGIN y el END de ese proceso. De manera, y ESTO ES IMPORTANTE, que podríamos tener ejecutándose a la vez muchos procesos, porque mientras el proceso principal crea un proceso "hijo" y las líneas de código de éste se empiezan a ejecutar, el programa principal puede tener todavía líneas por ejecutar –no en este ejemplo, pero sí es lo normal-, por lo que el ordenador empezará a ejecutar en paralelo los códigos de todos los procesos existentes en ese momento.

Miremos el código del interior del proceso “personaje”. Lo primero que hacemos es establecer valores a las variables locales predefinidas X, Y, GRAPH y FILE. Es decir, decimos de qué FPG (FILE) vamos a sacar la imagen (GRAPH) que va a visualizar el “personaje” en pantalla, y además decimos en qué coordenada inicial se va a situar el centro de esa imagen. Como hemos puesto una resolución de 640x480, fíjate que el gráfico aparecerá en el centro de la pantalla, inicialmente.

A continuación nos encontramos un LOOP, que comienza un bucle infinito, mientras el proceso exista, claro. Y los Ifs lo que comprueban es si alguna de las teclas de las flechas del teclado (izquierda, derecha, arriba, abajo) se ha pulsado. Si alguna condición de éstas se cumple, corrige las coordenadas del gráfico, es decir, lo mueve, porque varía los valores de las coordenadas X o Y, y por tanto, reubica el centro de la imagen dada por GRAPH.

Finalmente llegamos a la sentencia FRAME. Ésta sentencia es fundamental y conviene hacer unas cuantas apreciaciones. Sabéis que FRAME indica cuándo está el proceso listo para volcar la siguiente imagen a la pantalla. Lo importante –y novedoso- del asunto es que, cuando nuestro programa consta de más de un proceso, hay que saber que el programa se pausará hasta que todos los procesos activos hayan llegado hasta una sentencia FRAME, momento en el cual el programa –s decir, el conjunto de procesos- reanuda su ejecución. Es decir, como podemos tener varios procesos, cada uno de los cuales con una sentencia FRAME particular, Fénix coordina el funcionamiento del programa de tal manera que espera a que todos los procesos hayan llegado, en su código particular, hasta la sentencia FRAME. El primero que llegue se tendrá que esperar al segundo y así. Esto es para que la impresión en la pantalla se haga de una manera sincronizada, con todos los procesos a la vez. Es como mandar la orden de “¡Visualizar fotograma!” para todos los procesos al unísono, de manera coordinada y al mismo tiempo. Si no se hiciera así, sería un caos, porque un proceso que tuviera poco código llegaría pronto al FRAME, mientras que uno con mucho código llegaría después y entonces se estarían visualizando fotogramas en tiempos diferentes para cada proceso, lo que sería un verdadero caos. Por lo tanto, los procesos que terminen antes su ejecución tendrán que esperarse siempre a los procesos más lentos para ir todos al mismo compás que marca el FRAME. De todo esto se puede deducir, por ejemplo, que si escribimos un bucle infinito sin sentencia FRAME el programa posiblemente se cuelga. ¿Por qué? Porque el proceso que tenga ese bucle infinito nunca llegará al FRAME, y los demás procesos estarán pausados esperándole sin remedio. O sea que cuidado.

Para acabar de rematar el concepto de proceso, por si todavía quedan algunas dudas, vamos a modificar el ejemplo anterior para introducir otro proceso, además del que ya tenemos, de manera que aparezcan por pantalla dos objetos independientes, los cuales podrán ser movidos por teclas diferentes.

El gráfico del nuevo proceso puede ser cualquiera. Puedes reutilizar el mismo gráfico que hemos utilizado para el primer proceso (no hay ningún problema para que procesos diferentes tengan el mismo valor de GRAPH), o bien, que es lo más normal, utilizar otro gráfico para este nuevo proceso, en cuyo caso tendríamos que incluirlo lógicamente dentro del FPG correspondiente y usarlo como ya sabemos. Vamos a hacer esto último: crea una nueva imagen, de 30x30 por ejemplo, e inclúyela dentro del archivo "imagenes.fpg", con el código 2.

Lo que deberíamos de escribir para que aparecieran por pantalla estos dos procesos, y que pudiéramos controlarlos de forma independiente con teclas diferentes sería algo parecido a esto:

```
PROGRAM ejemplo_procesos;
GLOBAL
  INT id1;
END
BEGIN
  set_mode(640,480,16,MODE_FULLSCREEN);
  id1=load_fpg("imagenes.fpg");
  personaje();
  personaje2();
END

PROCESS personaje()
BEGIN
  x=320; y=240; file=id1; graph=1;
  LOOP
    IF (key(_up)) y=y-10; END
    IF (key(_down)) y=y+10; END
    IF (key(_left)) x=x-10; END
    IF (key(_right)) x=x-10; END
    FRAME;
  END
END

PROCESS personaje2()
BEGIN
  x=100; y=100; file=id1; graph=2;
  LOOP
    IF (key(_w)) y=y-10; END
    IF (key(_s)) y=y+10; END
    IF (key(_a)) x=x-10; END
    IF (key(_d)) x=x-10; END
    FRAME;
  END
END
```

Fíjate en los cambios, marcados en negrita. Lo primero que hacemos es llamar al proceso "personaje2()" desde el programa principal, para comenzar la ejecución del código que hay en su interior (en otras palabras: para que "personaje2()" se pueda ejecutar, y por tanto, entre otras cosas, visualizarse).

Y luego, lo único que hemos hecho ha sido copiar prácticamente el mismo código de "personaje()" en "personaje2()". Un detalle que se ha cambiado ha sido la posición inicial. Efectivamente, comprueba que "personaje2()" ha de salir en la pantalla más a la izquierda y más arriba, porque sus X e Y son menores. Lo hemos hecho así porque si ambos procesos tuvieran la misma posición inicial, está claro que el gráfico de un proceso solaparía al otro -el último proceso que aparece en el código principal (en este caso "personaje2()") sería el que estaría por encima de los demás procesos anteriores-.

Otro detalle que se ha cambiado ha sido el valor de GRAPH, para que "personaje2()" tenga asociado otra imagen, incluida no obstante en el mismo FPG, ya que si te fija, la variable FILE no se ha cambiado y continúa haciendo referencia al mismo archivo "imagenes.fpg".

Y finalmente, se han cambiado las teclas de control del movimiento de "personaje2()". Si se pulsa la tecla "W", "personaje2()" se moverá hacia arriba, si se pulsa la tecla "S", se moverá hacia abajo, etc. No habría ningún problema si se asocian las mismas teclas de movimiento que a "personaje()" -los cursores-, pero entonces ¿qué pasaría?. Es fácil. Que cuando apretáramos la tecla "Left", por ejemplo, AMBOS procesos se moverían a la vez hacia la izquierda, y así con los demás cursores igual. Pruébalo.

Importante recalcar otra vez la presencia de la orden FRAME; en el nuevo proceso "personaje2()", al igual que estaba en "personaje()". Recuerda que TODOS los procesos han de tener como mínimo una orden FRAME para poderse ejecutar -y visualizar si es el caso- en cada fotograma evitando así los cuelgues del programa.

A partir de aquí, sólo nuestra imaginación nos pondrá límites a lo que queramos hacer. Por ejemplo, ¿qué se tendría que modificar del ejemplo anterior para que pulsando la tecla "n", el gráfico del "proceso2()" pasara a ser el mismo que el del "proceso1()", y pulsando la tecla "m" el gráfico del "proceso2()" volviera a ser el inicial? Efectivamente, tendríamos que incluir las siguientes dos líneas en el interior del bucle LOOP/END del "proceso2()" -antes de la orden Frame;-

```
if(key(_n)) graph=1; end  
if(key(_m)) graph=2; end
```

Pruébalo y juega con las posibilidades que se te ocurran.

### **Variabes globales, locales y privadas:**

Bueno, por fin ha llegado el momento de definir eso que hemos estado evitando durante todo el manual: ¿qué diferencias hay entre una variable global, local ,privada (y pública)? No le he explicado hasta ahora porque antes era necesario aprender el concepto de proceso poder entender dichas diferencias.

La diferencia entre estos tres tipos de variables está en su alcance.

#### **\*Variables GLOBALES:**

Las variables globales tienen alcance en todo el programa; esto quiere decir que cuando se declara una variable global, ésta será una variable –una posición de memoria- que puede accederse desde cualquier punto del programa. Se entiende por cualquier punto del programa, cualquier zona de sentencias, bien sea la zona de sentencias generales (en el programa principal), o bien sea cualquier zona de sentencias pertenecientes a cada proceso.

Por lo tanto, cuando se dice que un dato es global es porque puede utilizarse el mismo dato en el programa principal y en cualquiera de los procesos.

Y por tanto, y esto es importante, cualquier cambio que se realice en el valor de esta variable desde cualquier punto del programa repercutirá inmediatamente en el resto del código.

Es como si fuera un cajón compartido, el cual todos los procesos pueden abrir y cambiar su contenido cuando les tercie, de manera que cada proceso que abra el cajón se encontrará con lo que haya metido en él algún proceso anterior. Así que hay tener cuidado cuando se cambie el valor de una variable global en un proceso concreto, porque ese nuevo valor será el que vean todos los demás procesos a partir de entonces.

Sólo se podrán declarar variables globales en el proceso principal; en ningún otro proceso se pueden declarar –cosa lógica por otra parte-. Para declarar variables globales, si son necesarias, simplemente hemos visto que es necesario escribir antes del bloque BEGIN/END del proceso principal lo siguiente:

```
GLOBAL
  Declaraciones_y_posibles_inicializaciones;
END
```

En general, se declaran como datos globales todos aquellos que establecen condiciones generales del juego que afecten a varios procesos; un ejemplo pueden ser los puntos obtenidos por el jugador, que podrían almacenarse en la variable global “puntuación”, de modo que cualquier proceso del juego pudiera incrementarla cuando fuera necesario.

### **\*Variables LOCALES:**

Las variables locales son variables que existen en cada proceso con un valor diferente pero que tienen el mismo nombre en todos.

Por ejemplo, si se crea una variable local “puntuación”, ésta existirá y estará disponible en todos los procesos, pero si en el proceso\_1 se le asigna un valor determinado, ese valor sólo estará asignado para el proceso\_1. El proceso\_2 también podrá tener propio valor para la variable “puntuación”, diferente del de la “puntuación” de proceso\_1. Y si se cambia el valor de “puntuación” del proceso\_2, no afectará para nada al valor de “puntuación” del proceso\_1. Las variables GRAPH o X son ejemplos de variables locales (predefinidas): todos los procesos tienen una variable llamada GRAPH o X, pero cada uno de ellos les asigna un valor distinto. Es decir, que cuando en un programa se utilice el nombre X, dependiendo de en qué proceso se utilice, se accederá a un valor numérico u otro. Cada proceso accederá con la variable local X, a su propia coordenada X

En resumen, todos los procesos poseen el mismo conjunto de variables locales (predefinidas y definidas por nosotros), pero cada proceso asigna un valor diferente a estas variables locales.

Además, otro aspecto importante de las variables locales es que sus valores para un proceso concreto pueden ser consultados/modificados desde otros procesos diferentes. Es decir, que si desde el código de un proceso A se quiere consultar/modificar el valor de la variable GRAPH del proceso B, es posible (y viceversa)

Sólo se podrán declarar variables locales en el proceso principal; en ningún otro proceso se pueden declarar, ya que las variables locales tienen que existir para todos los procesos por igual y la única manera de conseguirlo es declarándolas en el main. Para declarar variables locales, si son necesarias, simplemente hemos visto que es necesario escribir antes del bloque BEGIN/END del proceso principal lo siguiente:

```
LOCAL
  Declaraciones_y_posibles_inicializaciones;
END
```

En general, se declaran como datos locales todos aquellos que se consideren informaciones importantes de los procesos. Un ejemplo puede ser la energía que le queda a un proceso (puede ser la energía de una nave, la de un disparo, la del protagonista, etc). Esta información podría almacenarse en la

variable local “energía” de modo que cualquier proceso pudiera acceder y modificar su propia energía, e incluso la de los demás (por ejemplo, cuando colisionara con ellos les podría quitar energía), ya que también es posible leer y escribir los valores de una variable local de un proceso desde otro. Eso ya lo veremos. Así pues, en general este tipo de variables se usan cuando hay cosas en común en varios procesos distintos, como pueden ser energías, vidas, puntos de fuerza, de magia, o contadores.

### **\*Variables PRIVADAS:**

Por último, los datos privados son datos que se utilizan exclusivamente en un proceso del programa.

Por ejemplo, si un objeto del juego requiere una variable para contar de 0 a 9 guardará este valor en una variable privada, pues los demás procesos no necesitan utilizar este valor. Es decir, que si se define en un proceso un dato privado llamado "micontador", éste será un nombre que no significará nada dentro de otro proceso.

Es por esta razón, por ejemplo, que en el primer código de ejemplo de este capítulo (el programa “ejemplo\_procesos”) se declaró como global la variable “id1”, ya que si no el ejemplo no funcionaba. Esto es porque si se declaraba privada, la variable “id1” solamente tendría sentido dentro del proceso del programa principal, pero no dentro de los demás procesos, donde “id1” no significaría nada. Como en el proceso “personaje” se utiliza la variable “id1”, ya que su valor se le intenta asignar a FILE, es necesario que “id1” sea reconocida por ambos procesos. Podría ser pues o una variable local o una global, pero está claro que ha de ser global porque “id1” solamente va a contener un valor único para todos los procesos que existan: el identificador –único- del fichero FPG cargado. No tendría ningún sentido que la variable fuera local.

Otra diferencia importante de las variables privadas respecto las locales es que en el caso de las privadas NO es posible consultar/modificar sus valores para un proceso concreto desde otros procesos diferentes. Es decir, que si desde el código de un proceso A se quiere consultar/modificar el valor de una variable privada del proceso B, NO es posible (y viceversa)

Para declarar variables privadas, si son necesarias, simplemente hemos visto que es necesario escribir antes del bloque BEGIN/END de cualquier proceso lo siguiente:

```
PRIVATE
  Declaraciones_y_posibles_inicializaciones;
END
```

Esta sección puede aparecer tanto en el programa principal como en cualquier otro proceso del programa, ya que el programa principal se considera como otro proceso más.

En general se declaran como datos privados todos aquellos que vayan a ser utilizados como contadores en un bucle, las variables para contener ángulos o códigos identificadores secundarios, etc. Si un dato declarado como privado necesita consultarse o modificarse desde otro proceso (identificador.dato), entonces se deberá declarar dicho dato como local, (dentro de la sección LOCAL del programa); de esta forma, todos los procesos poseerán el dato, pudiendo acceder cada uno a su valor o al valor que tenga dicho dato en otro proceso.

En resumen: a los datos globales se puede acceder desde cualquier punto del programa; los locales son

aquellos que tienen todos los procesos (cada proceso tiene su propio valor en ellos) y se pueden consultar/modificar entre procesos, mientras que los privados son los que pertenecen a un solo proceso determinado y no pueden ser consultados/modificados por ningún otro proceso diferente.

Los datos globales se utilizan principalmente para controlar dispositivos como la pantalla, el ratón, el joystick, el teclado o la tarjeta de sonido. Los datos locales sirven para conocer cosas como cuál es el gráfico de un proceso, en qué posición aparece, cuál es su tamaño, su ángulo, su plano de profundidad, etc. De hecho, éstos son valores que pueden indicarse en diferentes variables locales predefinidas de los procesos; y todas tienen un valor válido por defecto, por lo que únicamente se tendrán que establecer aquellos valores que quieran alterarse. Los datos privados se utilizan básicamente para contadores y variables de soporte.

He dicho que los valores de las variables locales de un proceso pueden ser manipulados desde otro proceso. Veremos cómo se hace esto cuando conozcamos (un poquito más adelante) otro concepto: los llamados identificadores de procesos.

### **Ámbito de las variables:**

Ya conocemos tres de los cuatro tipos de variables que existen en Fénix (globales, locales y privadas: las públicas las trataremos más adelante). Y ya sabemos lo que son los procesos. Es posible que te estés preguntando en este momento si es posible o no declarar cualquiera de los tres tipos dentro de cualquier proceso, o si sólo se pueden declarar en el programa principal... vamos, en definitiva: en qué lugar del código se han de declarar las variables de distinto tipo.

Vamos a partir del ejemplo que estuvimos viendo unos párrafos más atrás: lo vuelvo a escribir aquí:

```
PROGRAM ejemplo_procesos;
GLOBAL
  INT id1;
END
BEGIN
  set_mode(640,480,16,MODE_FULLSCREEN);
  id1=load_fpg("imagenes.fpg");
  personaje();
  personaje2();
END
```

```
PROCESS personaje()
BEGIN
  x=320; y=240; file=id1; graph=1;
  LOOP
    IF (key(_up)) y=y-10; END
    IF (key(_down)) y=y+10; END
    IF (key(_left)) x=x-10; END
    IF (key(_right)) x=x+10; END
    FRAME;
  END
END
```

```
PROCESS personaje2()
BEGIN
  x=100; y=100; file=id1; graph=2;
```

```

LOOP
  IF (key(_w)) y=y-10; END
  IF (key(_s)) y=y+10; END
  IF (key(_a)) x=x-10; END
  IF (key(_d)) x=x-10; END
  FRAME;
END
END

```

La única sección de declaraciones que vemos es la sección de variables globales, en el programa principal. ¿Tendría sentido escribir alguna sección más de variables globales dentro de algún proceso? Es decir, escribir por ejemplo:

```

PROCESS personaje()
GLOBAL
  int varGlobalPersonaje;
END
BEGIN
  x=320; y=240; file=id1; graph=1;
  LOOP
    IF (key(_up)) y=y-10; END
    ...
  END
END

```

Pues no, no tiene ningún sentido, porque ya hemos visto que las variables globales son como un único cajón compartido, donde todos los procesos pueden acceder a él y leer y escribir valores. Así que las variables globales no son propias de ningún proceso, y por lo tanto, **la única sección de declaraciones de variables globales estará sólo en el programa principal.**

Vayamos ahora a por las variables locales. ¿Dónde se declaran? En el programa principal o también dentro de los procesos que queramos? Piensa en lo que significa ser una variable local: es una variable de la cual TODOS los procesos tienen una copia (eso sí, con valores posiblemente diferentes). Es decir, si TODOS los procesos tienen una misma variable local "X", podemos concluir que **la única sección de declaraciones de variables locales estará sólo en el programa principal**, ya que es el único sitio al cual tienen acceso todos los procesos sin problemas.

¿Y las variables privadas? Tal como su nombre indica, estas variables son particulares de cada proceso :ningún proceso tiene por qué saber qué variables privadas tiene declaradas otro, y viceversa. Así que queda claro que en el caso de las variables privadas, **existirá una sección de declaraciones de variables privadas dentro del código interno de cada proceso (y del código principal) donde sea necesario.**

Así pues, para escribir correctamente un programa que utilizara estos tres tipos de variables, tendríamos que hacer lo siguiente (tomando el código de los ejemplos anteriores):

```

PROGRAM ejemplo_procesos;
GLOBAL
  INT id1;
END
LOCAL
  INT varLocal; //No se usa, pero para declararla lo tengo que hacer aquí
END
PRIVATE

```

```

INT varPrivadaCodigoPrincipal; //No se usa, pero para declararla lo tengo que hacer aquí
END
BEGIN
    set_mode(640,480,16,MODE_FULLSCREEN);
    id1=load_fpg("imagenes.fpg");
    personaje();
    personaje2();
END

PROCESS personaje()
PRIVATE
    INT varPrivadaPersonaje; //No se usa, pero para declararla lo tengo que hacer aquí
END
BEGIN
    x=320; y=240; file=id1; graph=1;
    LOOP
        IF (key(_up)) y=y-10; END
        IF (key(_down)) y=y+10; END
        IF (key(_left)) x=x-10; END
        IF (key(_right)) x=x+10; END
        FRAME;
    END
END

PROCESS personaje2()
PRIVATE
    INT varPrivadaPersonaje2; //No se usa, pero para declararla lo tengo que hacer aquí
END
BEGIN
    x=100; y=100; file=id1; graph=2;
    LOOP
        IF (key(_w)) y=y-10; END
        IF (key(_s)) y=y+10; END
        IF (key(_a)) x=x-10; END
        IF (key(_d)) x=x+10; END
        FRAME;
    END
END

```

El orden en el que aparezcan las secciones de declaraciones es irrelevante.

Si te fijas, verás que en el código anterior, dentro del código principal, hemos escrito una sección GLOBAL, LOCAL y PRIVATE. Las variables globales y locales ya sabemos que sirven para todos los procesos y por eso está claro que han de ir declaradas en el código principal, pero, ¿esta sección PRIVATE, de qué proceso es privada?

Resulta que lo que hasta ahora habíamos llamado “código principal” no deja de ser un proceso más como otro cualquiera. La única particularidad es que es el único proceso que obligatoriamente ha de existir, y es el proceso que se ejecutará primero de todos. ¿Es un proceso?, te estarás preguntando. ¿Entonces por qué no aparece la línea PROCESS... por ningún sitio? Pues porque en el caso de este proceso, (el “proceso-código principal”), esta línea es opcional. Por eso no la hemos escrito nunca, pero si quisiéramos, la podríamos escribir. Lo único que tienes que tener en cuenta es que este proceso ha de tener un nombre muy concreto y fijo: se ha de llamar “MAIN” -”principal”, en inglés-. Esto es así porque Fénix

necesita saber cuál de los múltiples procesos que pueden existir es el que primero se ha de ejecutar, y esto lo sabe si llamamos a dicho proceso así. Por lo tanto, podríamos reescribir el código anterior añadiendo la línea PROCESS...que falta.

```
PROGRAM ejemplo_procesos;
GLOBAL
  INT id1;
END
LOCAL
  INT varLocal; //No se usa, pero para declararla lo tengo que hacer aquí
END

PROCESS MAIN()
PRIVATE
  INT varPrivadaCodigoPrincipal; //No se usa, pero para declararla lo tengo que hacer aquí
END
BEGIN
  set_mode(640,480,16,MODE_FULLSCREEN);
  id1=load_fpg("imagenes.fpg");
  personaje();
  personaje2();
END

PROCESS personaje()
...
END

PROCESS personaje2()
...
END
```

Y ¡atención!, fijate que hemos tenido gran cuidado en dejar fuera del código del proceso “main” las declaraciones de variables globales y locales (las de las privadas no, evidentemente) , porque, repito, aquéllas no pertenecen a ningún proceso, y por lo tanto, han de ir fuera de cualquier código particular de ningún proceso. Así pues, podemos concluir que las declaraciones de variables globales y locales irán siempre después de la línea “program...” y antes de la línea “process main()”, si es que ésta existiera.

Ahora que ya sabemos dónde ubicar las distintas declaraciones de las distintas variables, podremos comprender el funcionamiento del siguiente ejemplo, que muestra algunas de las características definitorias de los tres tipos de variables comentados hasta ahora (globales, locales, privadas):

```
PROGRAM ejemplo_procesos;
GLOBAL
  int varGlobal;
END
LOCAL
  int varLocal;
END
PROCESS Main()
PRIVATE
  int varPrivCodPrinc;
END
BEGIN
  set_mode(320,240,16);
```

```

personaje();
personaje2();
write_var(0,100,80,4,varGlobal);
write_var(0,150,80,4,varLocal);
write_var(0,200,80,4,varPrivCodPrinc);
LOOP
    if(key(_a)) varGlobal=varGlobal+1; end
    if(key(_d)) varLocal=varLocal+1; end
    if(key(_g)) varPrivCodPrinc=varPrivCodPrinc+1; end
FRAME;
END
END

PROCESS personaje()
PRIVATE
    int varPrivPers1;
END
BEGIN
write_var(0,100,100,4,varGlobal);
write_var(0,150,100,4,varLocal);
write_var(0,200,100,4,varPrivPers1);
LOOP
    if(key(_b)) varGlobal=varGlobal+1; end
    if(key(_e)) varLocal=varLocal+1; end
    if(key(_h)) varPrivPers1=varPrivPers1+1; end
FRAME;
END
END

PROCESS personaje2()
PRIVATE
    int varPrivPers2;
END
BEGIN
write_var(0,100,120,4,varGlobal);
write_var(0,150,120,4,varLocal);
write_var(0,200,120,4,varPrivPers2);
LOOP
    if(key(_c)) varGlobal=varGlobal+1; end
    if(key(_f)) varLocal=varLocal+1; end
    if(key(_i)) varPrivPers2=varPrivPers2+1; end
FRAME;
END
END

```

Ejecuta el programa. Verás nueve 0 dispuestos en tres columnas de tres filas. ¿Qué pasa si pulsamos la tecla “a”? Que aumentarán los valores de los números de la columna de más a la izquierda ¿Por qué? Si te fijas en el código, cuando pulsamos “a” lo que hacemos es aumentar el valor de “varGlobal”, y su nuevo valor lo imprimimos en tres sitios diferentes -fijate en el write\_var correspondiente dentro de cada uno de los tres procesos que hay en el código-. ¿Por qué ponemos en pantalla ese valor en tres sitios? Para demostrarte que el valor de una variable global lo podemos modificar desde cualquier proceso y ese valor será el que usarán de forma compartida el resto: apreta la tecla “b” o la tecla “c” y verás que ocurre lo mismo que pulsando “a”, ya que, aunque lo hagamos desde un proceso u otro diferente, estamos modificando la misma variable “varGlobal”.

Si pulsas “d”, estarás aumentando sólo el número de la fila superior de la columna central, que se corresponde con el valor de la variable local del proceso “main”. Otra variable local diferente pero con el mismo nombre pertenecerá al proceso “personaje” y otra también diferente pero homónima pertenecerá a “personaje2”: el valor de la primera se modifica con la tecla “e” y el de la segunda con la tecla “f”, observando el correspondiente aumento en las cifras central e inferior-central, respectivamente.

Si pulsamos “g” estaremos modificando el valor de la variable privada del código principal, si pulsamos “h” el de “personaje” y si pulsamos “i” el de “personaje2”. En este ejemplo no se puede ver la diferencia que existe entre variables locales y privadas porque no estamos accediendo desde procesos diferentes a sus valores, (esto ya veremos más adelante cómo se hace con las variables locales), momento donde se nota la diferencia entre éstas, aunque ahora lo que sí que puedes probar es escribir por ejemplo dentro del proceso “personaje” (después del begin) una línea como ésta:

```
write_var(0,100,200,4,varPrivPers2);
```

Verás que al intentar ejecutar el programa ocurre un error, ya que el proceso “personaje” está intentando acceder a una variable privada de otro proceso, cosa que no puede hacer.

Siguiendo con el ejemplo anterior, ¿qué ocurriría si tenemos dos variables distintas con el mismo nombre? ¿Ocurriría algún error o funcionaría bien el programa? De entrada, decir que es una práctica MUY poco recomendable utilizar el mismo nombre para distintas variables, porque se presta a confusión y a medio plazo es una fuente inagotable de líos, errores y dolores de cabeza. Pero si aun así lo quieres probar, has de saber que solamente se permiten dos casos en los que dos variables diferentes pueden llamarse igual (en el resto de posibilidades da error de compilación):

- 1.-Dos variables privadas de distintos procesos se pueden llamar igual ya que son completamente independientes entre sí.
- 2.-Una variable global y una variable privada se pueden llamar igual. No obstante, siempre que se haga referencia al nombre de ambas variables dentro del código del proceso donde se ha definido la variable privada homónima, dicho nombre hará referencia siempre a la variable privada, y no la global.

A lo mejor el segundo punto es el más enredado de todos: nada más fácil que verlo en un ejemplo:

```
PROGRAM ejemplo_procesos;  
GLOBAL  
int hola=4;  
END  
PROCESS Main()  
BEGIN  
set_mode(320,240,16);  
personaje();  
personaje2();  
write_var(0,100,80,4,hola); //Se muestra el contenido de la variable global  
LOOP  
FRAME;  
END  
END  
  
PROCESS personaje()  
PRIVATE  
int hola=2;
```

```

END
BEGIN
  write_var(0,100,100,4,hola); //Se muestra el contenido de la variable privada
  LOOP
    FRAME;
  END
END

PROCESS personaje2()
BEGIN
  write_var(0,100,120,4,hola); //Se muestra el contenido de la variable global

  LOOP
    FRAME;
  END
END

```

Por cierto, no lo he comentado hasta ahora, pero una regla básica (entre otras más esotéricas) a la hora de nombrar a las variables de nuestro programa, -sean del tipo que sean-, es que su nombre no puede empezar por un dígito numérico. Es decir, variables llamadas “1var” o “9crack” son inválidas y provocarán un error de compilación. Existen más normas que prohíben la inclusión de determinados caracteres especiales en el nombre de las variables (como %,\$,etc), pero a menos que seas un poco extravagante, no te deberás preocupar por esto, ya que siempre usaremos para nombrar variables caracteres alfabéticos, o en todo caso alfanuméricos.

### **Constantes:**

Las constantes son nombres (como los que se dan a las variables) que se utilizan como sinónimos de valores. Se pueden entender como variables que siempre tienen un mismo valor definido, un valor que no se puede cambiar. Estos valores pueden ser números enteros, decimales o cadena de caracteres, pero en ningún caso, a diferencia de las variables, se especificará explícitamente su tipo de datos: simplemente se asignará al nombre de la constante su valor correspondiente, del tipo que sea,y ya está.

Utilidad: se podría utilizar, por ejemplo, una constante denominada “altura\_maxima” como un sinónimo permanente del valor numérico 100. Es decir, que será indiferente utilizar “altura\_maxima” o 100 en el programa.

Las constantes se utilizan para ver el listado de un programa de una forma más clara. En el ejemplo anterior se haría para informar a las personas que leyeran el programa de que el número 100 que utiliza el programa es la altura máxima (de cualquier cosa u objeto del juego que fuera).

Otro uso de las constantes es el siguiente. Si en un juego se establece en varias ocasiones 3 como el número de vidas del protagonista, cuando se quiera modificar para aumentarlo o disminuirlo se tendría que buscar y sustituir ese número por todo el programa, corriendo además el riesgo de sustituir otro número 3 que pudiera aparecer en el programa para otra cosa. Una alternativa es declarar una constante denominada, por ejemplo, “maximo\_vidas” como un sinónimo del valor numérico 3 y utilizar en el programa dicha constante en lugar del número; ahora, cuando se quiera cambiar este valor por otro, simplemente habrá que hacerlo una sola vez en la declaración de constante “maximo\_vidas”.Y ya está.

Para declarar constantes, si son necesarias, simplemente es necesario escribir antes del bloque BEGIN/END de cualquier proceso –aunque lo más habitual es hacer siempre en el proceso principal por lógica- lo siguiente:

```

CONST
  Nombre_Constante= Valor_numérico;
END

```

Fíjate que no es necesario especificar el tipo de la constante, tal como se ha explicado anteriormente. Igual que pasa con las variables, también hay constantes predefinidas. Un ejemplo de constantes predefinidas podría ser MIN\_INT, que es un sinónimo del menor valor numérico que puede tomar un entero tipo INT (-2147483648) o MAX\_INT, que es sinónimo del mayor valor numérico que puede tomar un entero tipo INT (2147483648).

### **Parámetros de un proceso:**

Ahora veremos ejemplos de programas con procesos que admiten parámetros, para ver cómo funcionan. En general, los procesos pueden recibir parámetros en los siguientes tipos de datos:

Una variable local predefinida (como X,GRAPH,SIZE,FLAGS,...), **por lo que no habrá de declararlas en ningún sitio, NI TAMPOCO SERÁ NECESARIO ESPECIFICAR SU TIPO EN LA CABECERA DEL PROCESO.**

Una variable privada del PROPIO PROCESO. En este caso, la definición se realiza directamente en la cabecera del proceso y por tanto, **NO SE HA DE ESCRIBIR NINGUNA SECCION PRIVATE/END en el cuerpo del proceso.**

Una variable global definida por nosotros dentro de la sección GLOBAL del programa principal.

Una variable local definida por nosotros dentro de la sección LOCAL del programa principal.

Una variable pública definida por nosotros dentro de la sección PUBLIC del programa principal.

Como ejemplo, ahí va un programa con un proceso que recibe cuatro parámetros diferentes de los tipos indicados anteriormente:

```

PROGRAM mi_juego;
GLOBAL
  INT puntos;
END
LOCAL
  INT energia;
END
BEGIN
  mi_proceso(1,2,3,4);
  //Más instrucciones
END

PROCESS mi_proceso(x,int energia,int puntos,int n)
BEGIN
  //Más instrucciones
END

```

Vemos que el funcionamiento es el siguiente: a la hora de crear el proceso se ponen los valores que queramos que tengan los parámetros, y en la cabecera del proceso –en la línea del *PROCESS etc-* se escriben los nombres de las variables a las cuales se les va a asignar esos valores, en el mismo orden. Es decir, que en este ejemplo, cuando se cree *mi\_proceso*, la variable local predefinida *X* de dicho

proceso valdrá 1, la variable local “energía” valdrá 2, la variable global “puntos” valdrá 3 y la variable privada del proceso “n” valdrá 4. Y con estos valores se ejecutará el código interno de ese proceso.

Fíjate que en la cabecera de “mi\_proceso()” no hemos especificado el tipo de X porque no hace falta: es una variable predefinida, pero sí que lo hemos hecho con las otras. Y además, la variable “n” es privada al proceso (no aparece en las secciones GLOBAL/END ni LOCAL/END) pero no tiene ninguna sección de declaración PRIVATE/END dentro de éste porque el aparecer en la cabecera, este hecho ya implica que se declare automáticamente.

Fíjate también que con este sistema podemos ir creando diferentes procesos que tengan el mismo código pero que tengan distintos valores iniciales para algunas variables: un ejemplo es crear un proceso “enemigo”, que según los valores que pongamos a la hora de crearlo tenga asociado un gráfico u otro, una posición u otra, una energía u otra, creando así múltiples enemigos que son ligeramente diferentes mediante un único código. Bien, ¿no?

Ahora, a ver si entiendes el siguiente código:

```
PROGRAM mi_juego;
PRIVATE
    id2;
END
BEGIN
    id2=mi_proceso(160,100);
    //Más instrucciones
END

PROCESS mi_proceso(x,y)
PRIVATE
    n;
END
BEGIN
    set_mode(640,480,16);
    graph=load_png("imagen.png");
    FROM n=0 TO 99;
        x=x+2;
        y=y+1;
        FRAME;
    END
END
```

En este ejemplo, cuando se llama a un proceso, éste devuelve un número entero, que lo recogemos en una variable privada del código principal, “id2”. No lo hemos dicho hasta ahora, pero cada vez que creamos un proceso, éste nos devuelve un número –aleatorio- el cual podremos utilizar para hacer referencia a ese proceso a lo largo de todo el código (para matarlo, o dormirlo por ejemplo). De hecho, en el programa de ejemplo no se usa para nada, pero se ha escrito para que veas la posibilidad.

Este código lo que hace es crear un proceso llamado mi\_proceso, el cual tiene dos parámetros y ya está. Si luego nos fijamos en el código de este proceso, podemos ver que lo que representan estos dos parámetros es la posición inicial x e y (porque así está escrito en la cabecera del proceso) del gráfico asociado al proceso, el cual se obtiene a partir del archivo “imagen.png”. Una vez, pues, definida la posición inicial y el gráfico, se entra en un bucle el cual él solo va moviendo la coordenada del centro del dibujo hacia la derecha y para abajo, hasta que finalmente el contador “n” llega a 99, momento en el cual el proceso mi\_proceso muere. Fíjate que si cambiamos los valores de los parámetros de mi\_proceso,

estaremos cambiando la posición inicial donde se dibuja el proceso.

### **Código identificador de un proceso. La variable local predefinida ID y la función “Get\_id()”:**

Un código identificador es un número entero diferente de 0 que Fénix asigna automáticamente a cada proceso y lo identifica individualmente mientras el juego se está ejecutando. Los ID son como el DNI de cada uno de los procesos individuales que en un momento determinado estén ejecutándose en nuestro juego.

Que un ID sea asignado a un proceso en tiempo de ejecución –es decir, cuando el juego se ejecuta y no antes-, significa que no podemos saber cuáles serán esos números mientras programamos ni tampoco podremos asignarlos o cambiarlos. De todas formas, es muy sencillo obtener ese valor: está almacenado en la variable local ID de cada proceso y también –importante- es el valor de retorno que devuelven los procesos cuando son llamados.

Los ID son fundamentales a la hora de interactuar con procesos, ya que gracias a su ID podemos leer y modificar las variables locales y públicas de un proceso (las privadas no) desde otro. Por lo tanto, podríamos mover un proceso desde otro, cambiar su gráfico, rotarlo o hacer cualquier cosa que podamos hacer mediante variables locales.

Pongamos un ejemplo ahora que es importante y nos va a servir para dos cosas: 1º) Para aclarar un poco las diferencias entre estos los tres tipos de variables vistos hasta ahora (globales, locales, privadas); 2º) Para observar la utilización de los ID de los procesos para manipular los valores de las variables locales:

```
program ejemplo2;
global
  int varglobal=20;
  int idproc1, idproc2;
  int idfpg;
end
local
  int varlocal;
end
begin
  set_mode(640,480,16);
  idfpg=load_fpg("graficos.fpg");
  idproc1=proceso1();
  idproc2=proceso2();
  write(0,500,80,4,"El identificador de proceso1: " + idproc1);
  write(0,500,180,4,"El identificador de proceso2: " + idproc2);
  loop
    frame;
  end
end

process proceso1()
private
  int varprivada1=1;
end
begin
  file=idfpg;
  graph=1;
  x=200;
```

```

y=300;
varlocal=10;
write(0,200,80,4,"El identificador de proceso1 visto desde él mismo: " + id);
write(0,200,100,4,"La variable local de proceso1 vista desde él mismo: "+ varlocal);
//write(0,200,120,4,"La variable local de proceso2 vista desde proceso1: "+ idproc2.varlocal);
write(0,200,140,4,"La variable privada de proceso1 vista desde él mismo: " + varprivada1);
    loop
        varglobal=varglobal+1;
        frame;
    end
end

process proceso2()
begin
file=idfpg;
graph=2;
y=100;
varlocal=20;
write(0,200,180,4,"El identificador de proceso2 visto desde él mismo: " + id);
write(0,200,200,4,"La variable local de proceso2 vista desde él mismo: "+ varlocal);
write(0,200,220,4,"La variable local de proceso1 vista desde proceso2: "+ idproc1.varlocal);
//write(0,200,240,4,"La variable privada de proceso1 vista desde proceso2 : " + idproc1.varprivada1);
    loop
        x=varglobal;
        frame;
    end
end
end

```

En este código hay mucha chicha. Por un lado tenemos una variable global llamada "varglobal", la cual tiene inicialmente un valor de 20. Si te fijas, desde "proceso1" se accede a ella modificando en cada iteración (por tanto, en cada fotograma) su valor: como es global podemos hacerlo. Pero además, desde "proceso2" también se accede a ella en cada iteración, haciendo que el valor de la posición x de su gráfico coja el valor que tiene en ese momento "varglobal". Como las variables globales son únicas para todo el programa, si "proceso1" ha aumentado en 1 su valor, esto repercutirá en que en ese mismo fotograma la posición horizontal de "proceso2" será de un pixel más a la derecha.

Por otro lado, tenemos una variable local llamada "varlocal". Esto quiere decir que todos los procesos del programa tienen una variable llamada así. Si te fijas, en "proceso1" le asigno el valor de 10 y en "proceso2" le asigno el valor de 20. El punto interesante está en que es posible acceder/modificar al valor de "varlocal" de "proceso1" desde "proceso2" así como al valor de "varlocal" de "proceso2" desde "proceso1". Y esto lo conseguimos gracias a los identificadores de cada proceso.

Recuerda que cuando se llama a un proceso, éste devolverá siempre su identificador (que no es más que un número entero único en ese momento). Así pues, dicho identificador lo recogemos en el instante que creamos cada proceso, almacenándolo en una variable (en nuestro caso, "idproc1" e "idproc2").

Pero a partir de aquí, cuando ya tenemos guardado en estas variables el identificador de cada proceso, ¿cómo accedemos al valor de alguna de sus variables locales? Pues esto se hace en general con la sintaxis:

*identificadorProceso.nombreVariableLocal*

...que es lo que hemos hecho en el ejemplo: fijate que en "proceso2" logro saber cuál es el valor de "varlocal" de "proceso1" gracias a la notación "idproc1.varlocal".

Si eres perspicaz te habrás fijado en un detalle: la línea que nos tenía que imprimir en pantalla el valor encontrado desde "proceso1" de la "varlocal" de "proceso2" está comentada...Si pruebas a descomentarla verás que el programa lanza un error desagradable. Y esto es por una razón sutil: ocurre que hasta que todos los procesos no llegan por primera vez a la línea frame; -al primer fotograma-, cada proceso no es consciente de la existencia de los demás procesos que han sido creados posteriormente a él. Es decir, en el código principal primero creamos a "proceso1" y luego a "proceso2". Bien, hasta que "proceso1" no llegue a ejecutar el primer frame (y con él, todos los demás procesos, claro), dentro de su código no se podrá hacer referencia a ningún otro proceso que haya sido creado después de él, porque no sabe de su existencia.

Prueba un experimento: intercambia el orden de creación de los procesos (o sea, pon primero la línea que crea "proceso2", antes que "proceso1" y descomenta la línea problemática. Verás que continúas teniendo el mismo problema, pero ahora causado por otra línea: la que pretende acceder desde "proceso2" al valor de "varlocal" de "proceso1". Y esto, ¿cómo lo solucionamos?. Una manera sería poner las referencias que se hagan a los procesos posteriores al actual después de la primera línea frame; Es decir, si cortas la línea problemática y la pegas justo después del frame del mismo "proceso1" -y la descomentas-, verás que al ejecutar el programa funciona perfectamente. De todas maneras, la solución óptima es utilizar un bloque DECLARE/END, cosa que veremos posteriormente.

Fíjate también que mostramos por pantalla los identificadores de "proceso1" y "proceso2" de dos maneras distintas: por un lado, en el programa principal a partir de las variables (globales, para que se puedan usar cada una de ellas con su valor único en todo el programa) "idproc1" y "idproc2", pero también, dentro de cada proceso, haciendo uso de una variable local predefinida -al igual que lo son las variables X,Y,GRAPH o FILE, ya conocidas- llamada ID. Esta variable ID almacena, para cada proceso particular, su identificador.

Por último, tenemos en "proceso1" la variable privada "varprivada1". Podríamos intentar acceder desde "proceso2" al valor de "varprivada1" utilizando la misma notación por ejemplo que para las variables locales (y de hecho, hay una línea comentada que lo haría), pero si la descomentas verás que se produce un error sin solución: no se reconoce "varprivada1" porque las variables privadas sólo se pueden usar dentro del proceso donde se han definido.

Hasta ahora hemos visto cómo averiguar el código identificador de una instancia de un proceso concreto (bien obteniéndolo a partir del valor devuelto en la creación de ésta, o bien mediante la variable local ID para esa instancia de proceso en cuestión). Pero existen ocasiones en que nos puede interesar obtener no uno solo, sino todos los códigos identificadores de los procesos activos que son de un tipo concreto. Es decir, obtener los Ids de todos los procesos activos que sean "enemigos", o todos los procesos activos que sean "disparo", o "bola", o "proceso1", etc...

Fénix aporta una función que nos facilita la vida en este aspecto: la función "get\_id()".

### GET\_ID(TIPO)

Esta función devuelve el identificador del primer proceso activo que encuentre que corresponda al tipo especificado. Si no encontrase ninguno, esta función devolverá 0.

En las siguientes llamadas a esta función que se realicen utilizando el mismo parámetro, GET\_ID devolverá sucesivamente los identificadores de los siguientes procesos de ese tipo que se encuentren en memoria, hasta que ya no quede ninguno, en cuyo caso devolverá 0.

Es posible llamar a GET\_ID con un parámetro de tipo de proceso 0. En este caso, GET\_ID devolverá los identificadores de todos los procesos en memoria, de cualquier tipo.

GET\_ID no devolverá nunca identificadores de procesos muertos o marcados para matar el próximo frame mediante la función SIGNAL.

El estado de GET\_ID se reinicia cada frame, con lo que si los procesos del tipo buscado han variado (han aparecido nuevos, han muerto algunos), GET\_ID lo reflejará..

#### PARÁMETROS:

INT TIPO : Un tipo de proceso, dado por la expresión *TYPE nombre\_del\_proceso\_en\_el\_código*

Un ejemplo de esta función puede ser el siguiente. Necesitarás dos archivos PNG de unos 30x30 píxeles, llamados “circulo.png” y “cuadrado.png”, respectivamente:

```
program Test_GET_ID;
global
  int IDP;
  int cont;
  int png1,png2;
end
begin
  set_mode(320,200,16);
  png1=load_png("circulo.png");
  png2=load_png("cuadrado.png");
  from cont=1 to 10
    bola(rand(0,299),rand(0,199),png1);
    cuadrado(rand(0,299),rand(0,199),png2);
  end
  frame; //Para ver los 10 círculos y los 10 cuadrados en su posición inicial
  repeat
    /*Voy del 1 al 10 porque sé que hay 10 procesos de tipo cuadrado y por tanto, he de ejecutar la función
    get_id 10 veces para encontrar sus 10 IDs*/
    from cont=1 to 10
      /*En cada iteración IDP valdrá el identificador de un proceso "cuadrado" distinto.*/
      IDP=get_id(type cuadrado);
      /*Este if hace simplemente que los cuadrados se detengan en el extremo derecho de la pantalla y no vayan
      más allá.*/
      if(IDP.x < 320) IDP.x=IDP.x + 10; end
    end
  frame;
  until(key(_esc))
  exit();
end

process cuadrado(x,y,graph)
begin
  loop
    frame;
  end
end

process bola(x,y,graph)
begin
  loop
```

```
frame;  
end  
end
```

Puedes comprobar, si ejecutas el código, que lo que ocurre es que inicialmente tenemos diez procesos de tipo “bola” y diez procesos de tipo “cuadrado”. Y únicamente los de tipo “cuadrado” se mueven hacia la derecha: los de tipo “bola” se mantienen inmóviles en sus posiciones iniciales. Esto es porque la función `get_id()` solamente devuelve los identificadores de los procesos “cuadrado”, y por tanto, sólo variamos los valores de la variable local X de dichos procesos.

### **Jerarquías de procesos:**

Otro concepto que deberías tener claro trabajando con procesos en Fénix es el de las jerarquías.

Si un proceso a partir de su código crea otro proceso, dentro del código de éste siempre se podrá hacer referencia a su proceso creador utilizando una variable local llamada **“Father”** –padre- que guarda el código identificador de éste, y, desde el código de ese proceso padre, el proceso creado se podrá identificar con su código guardado en otra variable local: **“Son”** –hijo-, siempre que éste sea el último proceso creado (es decir, si el padre genera muchos hijos, solamente el último de ellos podrá ser llamado "Son" en el código del padre). Es decir, dicho de otra manera, un proceso Father es aquel que crea a otro, el cual será el proceso Son respecto el primero (hasta que Father genere un nuevo hijo, que entonces será el nuevo proceso Son).

Si un proceso no crea otros procesos, no tendrá ningún hijo y "Son" valdrá 0. Si se destruye un proceso que ha generado previamente otros procesos, éstos se dice que se han quedado huérfanos, y su variable local “father” valdrá 0.

También podemos utilizar la variable local **“Bigbro”** -hermano mayor- para identificar al último proceso que el creador -el padre- del proceso actual creó antes que éste, (y valdrá 0 si no se había creado ningún proceso anterior al actual). Y **“Smallbro”** servirá para referenciar al primer proceso que el creador del proceso actual creó después de éste, (y también valdrá 0 si no se ha creado ningún proceso posterior al actual).

Así pues, resumiendo: en un proceso “proceso1” cualquiera, la variable FATHER guarda el código identificador del proceso que llamó a “proceso1”, la variable SON guarda el código identificativo del último proceso “proceso1” ha llamado, la variable BIGBRO guarda el código identificador del proceso que el padre de “proceso1” llamó justo antes que “proceso1” y SMALLBRO guarda el del proceso que el padre de “proceso1” llamó justo después de él.

Como sabemos, el primer proceso que se ejecuta siempre es el proceso principal (también conocido como Main), el cual es el Father de todos los procesos, porque él es el que crea (llama) a todos los demás procesos que se van a ejecutar. Puesto que estos procesos tienen el mismo padre, se puede decir que son hermanos. A su vez, estos procesos pueden llamar a otros procesos, por lo que se convertirán a su vez en padres de estos procesos, que serán sus hijos y hermanos entre ellos, etc.

Esta nomenclatura tiene una razón de ser.

Por ejemplo, si se pausa o muere el Father, ineludiblemente también se pausarán o terminarán sus procesos hijos (todos, no solamente el último referenciado por "Son". ¿y esto, para qué sirve? Imagínate la siguiente situación con el juego de matar enemigos: hay sobre 20 disparos y 10 enemigos en pantalla (que hacen ya 30 procesos). Entonces te matan, el juego se acaba y la pantalla de créditos aparece. Cuando esta pantalla aparece, todos los enemigos y disparos ya no deberían de estar en pantalla: tienes que terminar sus

procesos. Podrías terminar los 30 procesos uno a uno, pero lo mejor es terminar el Father, ya que así, siempre que lo hagas de la manera adecuada, terminas todos sus procesos hijo de un plumazo. Enseguida lo veremos.

Otra utilidad fundamental de esta nomenclatura es la posibilidad de acceder a variables locales del proceso deseado. Por ejemplo, usando el identificador "Father", podemos acceder desde el código de cualquier proceso hijo a variables como *father.x* o *father.y*, que nos devolverán en todo momento la posición en pantalla del proceso padre (muy útil por ejemplo si queremos seguirle). De esta manera, con el uso de las variables *father*, *son*, *bigbro* y *smallbro* podemos tener controlada a toda la parentela cercana de un proceso con facilidad, consultando o modificando los valores que se deseen de X,Y,GRAPH,FILE...y otras variables locales que iremos viendo (como ANGLE,SIZE,Z,etc).

Un ejemplo que ilustra el significado de las variables locales predefinidas de jerarquía (*father*,*son*,*bigbro*,*smallbro*) podría ser éste:

```
//AUTHOR : COLDEV
```

```
/*
```

```
Si father crea son1, después son2 y después son3, entonces:
```

```
son1.bigbro = 0
```

```
son2.bigbro = son1
```

```
son3.bigbro = son2
```

```
son1.smallbro = son2
```

```
son2.smallbro = son3
```

```
son3.smallbro = 0
```

```
father.son = son3
```

```
*/
```

```
program jerarquias;
```

```
global
```

```
int count=0;
```

```
end
```

```
local
```

```
int var1,var2;
```

```
end
```

```
begin
```

```
set_mode(320,240,16);
```

```
myProcess1();
```

```
end
```

```
process MyProcess1()
```

```
begin
```

```
var1= 777;
```

```
var2= 111;
```

```
MyProcess2();
```

```
//Imprime el valor de var1 del proceso actual (MyProcess1), ya que el nieto de su abuelo es él mismo
```

```
write(0,100,20,0, son.son.father.father.var1);
```

```
loop frame;end
```

```
end
```

```
process MyProcess2()
```

```
private
```

```
int i;
```

```
end
```

```
begin
```

```

i=MyProcess3();
MyProcess3();
MyProcess3();
MyProcess3();
//Imprime el valor de var1 del proceso smallbro del primer proceso MyProcess3 creado
write(0,100,40,0, i.smallbro.var1 );
//Imprime el valor de var1 del proceso bigbro del último proceso MyProcess3 creado
write(0,100,60,0, son.bigbro.var1 ); //Por curiosidad: ¿qué sería entonces “son.bigbro.bigbro.var1”?
loop frame;end
end

process MyProcess3()
begin
count++;
var1=count;
//Imprime el valor de var2 del abuelo del proceso actual (es decir, MyProcess1)
write(0,100,80,0, father.father.var2 );
loop frame; end
end

```

Fíjate en los valores mostrados por pantalla y deduce, a partir del código, por qué son los que son.

### Señales entre procesos. Comando “Signal()”:

Un proceso A puede enviar un aviso, una señal a otro proceso B.

Para ello, el proceso A ha de ejecutar la orden “signal()”. Este comando tiene dos parámetros: el primero será el código identificador del proceso destino (proceso B); el segundo es la señal que se mandará, la cual se ejecutará inmediatamente, sin esperar al siguiente frame (excepto cuando la señal es enviada al mismo proceso que la generó -cuando se “autoenvía” la señal-, circunstancia en la que sí se espera al siguiente frame para hacer efectiva la señal).

Existen ocho señales:

<b>S_KILL</b>	Mata al proceso, por lo que su código dejará de ejecutarse y desaparecerá de la memoria y de la pantalla para siempre. Una vez muerto no se podrá acceder a sus variables locales ni modificarlo, pues ya no existe.
<b>S_SLEEP</b>	Duerme el proceso, por lo que su código no se ejecutará ni aparecerá en pantalla, pero el proceso seguirá existiendo y por tanto se podrá seguir accediendo a sus variables locales y modificarlas desde otros procesos.
<b>S_FREEZE</b>	Congela el proceso, por lo que su código no se ejecutará pero el gráfico seguirá quieto en pantalla (y los demás procesos podrán detectar colisiones con su gráfico), y también se podrá seguir accediendo y modificando sus variables locales.
<b>S_WAKEUP</b>	Despierta a los procesos dormidos o congelados de manera que sigan ejecutando su código donde lo dejaron.
<b>S_KILL_TREE</b>	Igual que S_KILL, pero además de actuar sobre el proceso destino que pongamos en <i>signal</i> , mata a toda su descendencia (procesos hijos, “nietos”, etc)
<b>S_SLEEP_TREE</b>	Igual que S_SLEEP, pero además de actuar sobre el proceso destino que pongamos en <i>signal</i> , duerme a toda su descendencia
<b>S_FREEZE_TREE</b>	Igual que S_FREEZE, pero además de actuar sobre el proceso destino que

	pongamos en <i>signal</i> , congela a toda su descendencia
<b>S_WAKEUP_TREE</b>	Igual que S_WAKEUP, pero además de actuar sobre el proceso destino que pongamos en <i>signal</i> , despierta a toda su descendencia

La razón de la existencia de las señales \*\_TREE es fácil de ver. Si se mata a un proceso, toda su descendencia se queda huérfana, con lo que los diferentes procesos hijo pierden su relación directa entre ellos (los hermanos ya dejan de tener una relación común porque el padre ha muerto y cada uno va por su lado -parece una telenovela ¿verdad?-), y mandar señales a estos procesos se convierte en tarea complicada. Por eso es conveniente en más de una ocasión eliminar a todo el árbol. Para eso existen las señales \_TREE, que afectan a todo el árbol que crea el proceso que le pasamos por parámetro en *signal*.

Como primer parámetro de la orden *signal* también se pueden poner directamente las variables FATHER,SON,etc (recordemos que su valor no es más que el identificador de un proceso determinado, como cualquier otro).

Hasta ahora he comentado que la orden *signal* era capaz de enviar una señal a una instancia de un proceso especificado. Pero esto no es todo. Esta orden también se puede utilizar para enviar la misma señal a la vez a un conjunto amplio de instancias de procesos de un tipo concreto.

Es decir, si estás programando un matamarcianos, es posible que necesites en algún momento matar a todos los enemigos de golpe (porque el jugador ha conseguido un mega-cañón ultrasónico que los deja a todos fritos de golpe con su onda expansiva). Recuerda que cada enemigo es una instancia concreta (con su ID propio) de un proceso llamémosle “enemigo”. Si utilizáramos la orden *signal* como hasta ahora se ha explicado, tendríamos que ingeniarnos un complicado sistema para recoger todos los identificadores de los diferentes enemigos existentes en el juego (todos ellos, recordemos, instancias del proceso “enemigo”) e ir llamando dentro de un bucle a la orden *signal* pasándole en cada iteración un ID diferente, para ir matando a los enemigos uno a uno.

¿No sería más fácil enviar una señal a todos los procesos que sean instancias de un proceso determinado? Algo así como “enviar esta señal a todos los procesos de tipo 'enemigo'”, en nuestro caso. Y ahorrarnos un montón de faena...Pues se puede, con la misma orden *signal*.

Si quieres saber cómo se haría para enviar una señal a un conjunto de procesos del mismo tipo, lo único que hay que escribir es como primer parámetro de la orden *signal* –el destinatario de la señal–, en vez de un código identificador, la sentencia *TYPE nombre\_del\_proceso\_en\_el\_código*. Es decir, que si quisiéramos destruir todos los enemigos de golpe, tendríamos que escribir:

```
signal(TYPE disparo,s_kill);
```

y así todos los procesos activos de tipo “enemigo” recibirían la señal S\_KILL.

Un ejemplo del envío de señales mediante *signal* podría ser el siguiente. Necesitarás un archivo FPG llamado “fpg.fpg” con un gráfico de unos 30x30 píxeles con código 001:

```
program Test_SIGNAL;
global
int fpg;
int IDB;
/*Valdrá 1 si el proceso “Bola” está vivo y 0 si no. Nos servirá para que, en el momento que “Bola” se detecte que esté muerta, se vuelva a generar otro proceso “Bola” y se pueda continuar el programa. Un proceso dejará de estar vivo SÓLO si ha recibido una señal de tipo KILL.*/
```

```

int bolaActiva;
/*Valdrá 1 si el proceso "Bola2" está vivo y 0 si no. Nos servirá para que, en el momento que "Bola2" se
detecte que esté muerta, se vuelva a generar otro proceso "Bola2". Un proceso dejará de estar vivo SÓLO
si ha recibido una señal de tipo KILL.*/
int bolaActiva2;
end
begin
set_mode(640,480,16);
write_var(0,600,450,4,fps);
fpg=load_fpg("fpg.fpg");
write(0,10,30,3,"1) S_KILL");
write(0,10,40,3,"2) S_FREEZE");
write(0,10,50,3,"3) S_SLEEP");
write(0,10,60,3,"4) S_WAKEUP");
write(0,10,70,3,"5) S_KILL_TREE");
write(0,10,80,3,"6) S_FREEZE_TREE");
write(0,10,90,3,"7) S_SLEEP_TREE");
write(0,10,100,3,"8) S_WAKEUP_TREE");
IDB=Bola(); bolaActiva=1;
repeat
if(key(_1) && bolaActiva==1) signal(IDB,s_kill);bolaActiva=0;end
if(key(_2) && bolaActiva==1) signal(IDB,s_freeze);end
if(key(_3) && bolaActiva==1) signal(IDB,s_sleep);end
if(key(_4) && bolaActiva==1) signal(IDB,s_wakeup);end
if(key(_5) && bolaActiva==1) signal(IDB,s_kill_tree);bolaActiva=0; bolaActiva2=0; end
if(key(_6) && bolaActiva==1) signal(IDB,s_freeze_tree);end
if(key(_7) && bolaActiva==1) signal(IDB,s_sleep_tree);end
if(key(_8) && bolaActiva==1) signal(IDB,s_wakeup_tree);end
/*Este if lo único que hace es, si la bola de arriba está muerta, esperar un poco y volver a crear una nueva
bola. No te preocupes por el parámetro -no visto todavía- que tiene la orden frame: sólo decir que sirve
para pausar un poquitín el juego de manera que se note que durante unos instantes no hay bola de
arriba.*/
if(bolaActiva==0)
frame(2000);
IDB=Bola(); bolaActiva=1;
end
frame;
until(key(_esc))
exit();
end

process Bola()
begin
graph=1;
x=180;
y=40;
/*Este if lo único que hace es, si la bola de abajo está muerta, esperar un poco y volver a crear una nueva
bola. No te preocupes por el parámetro -no visto todavía- que tiene la orden frame: sólo decir que sirve
para pausar un poquitín el juego de manera que se note que durante unos instantes no hay bola de
abajo.*/
if(bolaActiva2==0)
frame(2000);
Bola2(); bolaActiva2=1;
end

```

```

loop
  while(x<600) x=x+5; frame; end
  while(x>180) x=x-5; frame; end
end
end

```

```

process Bola2()
begin
  graph=1;
  x=180;
  y=80;
  loop
    while(x<600) x=x+5; frame; end
    while(x>180) x=x-5; frame; end
  end
end
end

```

Si ejecutas este código, lo que podrás ver será dos gráficos que no paran de moverse constantemente de un lado a otro de la pantalla, y un menú con 8 opciones para aplicar cada una de las 8 señales existentes a alguno (o todos) de los procesos activos. El gráfico de arriba es un proceso (“Bola”) y el gráfico de abajo es otro (“Bola2”). El de arriba es padre del de abajo, al menos al principio de la ejecución del programa. Estudiemos lo que pasa y su relación con el código fuente de la siguiente manera:

\*Si pulsamos la tecla “2”, siempre que “Bola” esté vivo, lo congelamos. Se puede comprobar que el gráfico correspondiente se para y deja de moverse, pero al gráfico de debajo no le pasa nada porque es otro proceso diferente que no resulta afectado por la señal.

\*Si seguidamente pulsamos la tecla “4”, siempre que “Bola” esté vivo, lo descongelamos ó despertamos, según el caso. En este caso, al descongelarlo, lo que veremos es que el gráfico comienza otra vez a moverse de forma normal desde el sitio donde permanecía congelado.

\*Si ahora pulsamos la tecla “3”, siempre que “Bola” esté vivo, lo dormimos. Se puede comprobar que el gráfico correspondiente desaparece de la pantalla, pero al gráfico de debajo no le pasa nada porque es otro proceso diferente que no resulta afectado por la señal. Si volvemos a pulsar otra vez la tecla “4”, veremos que despertamos a “Bola”, con lo que volveremos a ver el gráfico en pantalla y éste comenzará otra vez a moverse de forma normal desde el sitio donde desapareció.

\*Si pulsamos la tecla “6”, siempre que “Bola” esté vivo, congelamos el proceso “Bola” y toda su descendencia, que en este caso es sólo el proceso “Bola2”. Es decir, en definitiva congelamos dos procesos con una sola orden *signal*. Podemos comprobar como en este caso, ambos gráficos se paran y dejan de moverse.

\*Si seguidamente pulsamos la tecla “8”, siempre que “Bola” esté vivo, descongelamos ó despertamos, según el caso, al proceso “Bola” y toda su descendencia, que en este caso es sólo el proceso “Bola2”. Lo que veremos como resultado es que ambos gráficos comienzan a la vez a moverse de nuevo de forma normal desde el sitio donde permanecían congelados, como si no hubiera pasado nada.

\*Si ahora pulsamos la tecla “7”, siempre que “Bola” esté vivo, dormimos el proceso “Bola” y toda su descendencia, que en este caso sólo es “Bola2”. Podemos comprobar como en este caso, ambos gráficos desaparecen de la pantalla. Si volvemos a pulsar otra vez la tecla “8”, veremos que despertamos a “Bola” y a toda su descendencia (o sea, “Bola2” también), con lo que volveremos a ver ambos gráficos en pantalla y éstos comenzarán otra vez a moverse de forma normal desde el sitio donde desaparecieron cada uno.

\*Si pulsamos la tecla "5", siempre que "Bola" esté vivo, matamos a éste y toda su descendencia (es decir, al proceso "Bola2" también lo matamos). Como resultado, ambos procesos desaparecerán de la pantalla. No obstante, al cabo de unos instantes, el código fuente del ejemplo ha sido escrito de tal manera que se vuelve a generar un nuevo proceso "Bola" -y éste, un nuevo proceso "Bola2"- volviendo a empezar de nuevo.

\*Si pulsamos la tecla "1", siempre que "Bola" esté vivo, lo matamos. Se puede comprobar que el gráfico correspondiente desaparece, pero al gráfico de debajo no le pasa nada porque es otro proceso diferente que no resulta afectado por la señal. No obstante, al cabo de unos instantes, el código fuente del ejemplo ha sido escrito de tal manera que se vuelve a generar un nuevo proceso "Bola" -y éste, un nuevo proceso "Bola2"-, con lo que tendremos tres gráficos en pantalla: el perteneciente a los nuevos "Bola" y "Bola2", y el perteneciente al antiguo "Bola2" que ha quedado huérfano. Si ahora pruebas por ejemplo de pulsar la tecla 5 (la señal S\_KILL\_TREE) verás que sólo morirán los dos procesos acabados de generar, porque son los únicos que están emparentados: el antiguo "Bola2" tiene un padre que ya no existe, y por tanto, no pertenece a la misma familia que los nuevos. Esto implica que, tal como está escrito el código de ejemplo, no le podemos enviar ninguna señal a este proceso huérfano, por lo que permanecerá en pantalla a perpetuidad, independientemente de la creación/destrucción de nuevas familias de procesos.

Otro ejemplo, similar al anterior, de utilización de señales dirigidas a toda la descendencia completa de un proceso padre (todo un árbol), es el siguiente. Para poderlo visualizar necesitarás crear una imagen PNG llamada "circulo.png" de unos 40x40 píxeles -que será como su nombre indica, una bola-.

*//Author: Wakroo*

*Program pausa;*

*Global*

*Int mapCirculo;*

*Int idCirculator;*

*int texto;*

*Int pausado = 0;*

*end*

*Begin*

*set\_title("Ejemplo de Pausa");*

*set\_mode (640,480,16);*

*mapCirculo = load\_png("circulo.png");*

*idCirculator = circulator();*

*write(0,0,0,0,"P para pausar todos los hijos");*

*write(0,0,10,0,"O para quitar la pausa de todos los hijos");*

*write(0,0,20,0,"S para pausar el último hijo ('Son')");*

*write(0,0,30,0,"A para quitar la pausa del último hijo");*

*write(0,0,40,0,"ESC para salir");*

*Loop*

*If (key(\_esc)) exit(); End*

*If (key(\_p) AND NOT pausado)*

*signal(idCirculator,s\_freeze\_tree);*

*texto = write(0,320,240,4,"Pausa total");*

*pausado = 1;*

*End*

*If (key(\_o) AND pausado)*

*signal(idCirculator,s\_wakeup\_tree);*

*delete\_text(texto);*

*pausado = 0;*

*End*

*Frame;*

*End*

*End*

```

Process circulator()
Begin
  circulo(50,100);
  circulo(100,170);
  circulo(150,240);
  circulo(200,310);
  circulo(250,380);
Loop
  if (key(_s) AND NOT pausado)
    signal(son,s_freeze);
    texto=write(0,320,240,4,"Pausado sólo el último hijo");
    pausado=1;
  end
  if (key(_a) AND pausado)
    signal(son,s_wakeup);
    delete_text(texto);
    pausado=0;
  end
Frame;
End
End

Process circulo(x,y)
Private
  Int direc = 10;
end
Begin
  graph = mapCirculo;
Loop
  x =x+ direc;
  If (x < 50) direc = 10; End
  If (x > 590) direc = -10; End
Frame;
End
End

```

En este caso tenemos un proceso padre, “circulator”, que genera cinco procesos hijos (y hermanos entre sí), “circulo”, por lo que enviando desde el programa principal una señal de tipo \*\_TREE a “circulator”, controlamos todos los círculos a la vez. Además en este ejemplo, dentro del propio proceso “circulator”, podemos controlar individualmente el último de los cinco procesos creados por éste mediante la variable local “Son”.

### **Variables públicas y sentencia Declare:**

Una novedad de las últimas versiones de Fénix es la inclusión de un nuevo tipo de variables anteriormente inexistentes: las variables públicas.

Una variable pública es una variable que es específica de un proceso/función, igual que lo es una variable privada. Sin embargo, a diferencia de ésta, una variable pública puede ser accedida desde el resto del programa, igual que como se utilizaría una variable local (a partir del identificador de ese proceso).

Es decir, dicho de otra forma, una variable pública sería igual que una variable local, pero sólo existente para el proceso concreto donde se haya definido, no para todos.

La manera de acceder al valor de una variable pública de un proceso desde otro es similar a como se hacía con las variables locales: `identificadorProceso.nombreVariablePública`.

No obstante, una particularidad novedosa de este tipo de variables es que la variable “`identificadorProceso`” ha de declararse obligatoriamente no como `Int` -que es lo normal- sino como un tipo de datos “nuevo” cuyo nombre es igual al nombre de dicho proceso. Es decir, para poder acceder a las variables públicas de un proceso desde otro proceso, esa variable se ha de declarar no como `int`, sino con el nombre del proceso. (En realidad el tipo de dato finalmente acaba siendo un `int`, pero se ha de declarar tal como se comenta).

Esta particularidad tiene una consecuencia importante que ahora veremos. Prueba este código:

*Program example;*

*Public*

*int speed = 50;*

*String name = "Galactica";*

*End*

*Global*

*Nave ship;*

*End*

*Begin*

*ship = Nave();*

*write(0,100,100,4,ship.name + ":" + ship.speed);*

*Repeat*

*frame;*

*Until(key(\_esc))*

*End*

*Process Nave()*

*Begin*

*Loop*

*frame;*

*End*

*End*

En principio, este código lo único que haría es escribir en pantalla el valor de dos variables locales del proceso “Nave”. Fíjate, tal como he comentado, que el identificador del proceso no lo declaramos como `Int` sino con el nombre del proceso al que pertenecen las variables locales que queremos usar: “Nave”. No obstante, se produce un error en la compilación.

Fíjate en este otro código:

*Program example;*

*Process Nave()*

*Begin*

*Loop*

*frame;*

*End*

*End*

*Public*

*int speed = 50;*

```

    String name = "Galactica";
End
Global
    Nave ship;
End
Begin
    ship = Nave();
    write(0,100,100,4,ship.name + ":" + ship.speed);
    Repeat
        frame;
    Until(key(_esc))
End

```

Si te fijas, es el mismo programa de antes, pero con una diferencia. Hemos escrito el código del proceso "Nave" entre la primera línea ("program ...") y la sección de declaraciones de variables. Esto no lo habíamos explicado hasta ahora, pero es perfectamente posible alterar el orden "habitual" de nuestro código, que hasta ahora consistía en Línea program+declaraciones de variables+código programa principal + código procesos por este otro, donde el código de los diferentes procesos se pueden escribir (casi) al principio de todo.

Si ejecutas este programa, verás que ahora funciona. ¿Por qué? Porque en la primera versión del programa, al declarar la variable ship como "Nave", éste no es ningún tipo de dato reconocido y el compilador lanza un error, ya que el código del proceso "Nave" está escrito después de esa declaración, y allí el compilador todavía no ha llegado en ese momento. En cambio, en la segunda versión del programa, al haber escrito el código del proceso "Nave" antes de las declaraciones, cuando el compilador llega a leer éstas, él ya sabe qué significa la palabra "Nave".

De esta manera ya podemos utilizar las variables públicas sin problemas. No obstante, si no te acabas de acostumbrar al nuevo orden de escritura del código, existe una alternativa. El truco está en mantener el orden del código como siempre ( Línea program+declaraciones de variables+código programa principal + código procesos), pero declarar las variables públicas de otra manera. Con la sentencia DECLARE.

La sentencia DECLARE tiene la siguiente sintaxis:

```

DECLARE PROCESS nombreProceso (tipo param1,tipo param2,...)
...
END

```

donde los parámetros que se especifiquen serán los mismos que tiene el propio proceso.

Dentro de este bloque es donde se escribirá el bloque de declaraciones PUBLIC/END para las variables públicas, y todo en conjunto se escribirá en la sección de declaraciones de variables del programa principal, como hasta ahora.

Es decir, que la tercera versión de nuestro programa, con la orden DECLARE, quedaría así:

```

Program example;
Declare Process Nave()
    Public
        int speed = 50;
        String name = "Galactica";
    End
End

```

```

Global
  Nave ship;
End
Begin
  ship = Nave();
  write(0,100,100,4,ship.name + ":" + ship.speed);
  Repeat
    frame;
  Until(key(_esc))
End

```

```

Process Nave()
Begin
  Loop
    frame;
  End
End

```

Fíjate que cerramos con un END el bloque PUBLIC y con otro END el bloque DECLARE.

Pongo otro ejemplo muy similar al anterior para afianzar estos nuevos conceptos:

```

//Author: Coldev
program hola;
DECLARE PROCESS mi_proceso()
public
  int hola;
  int adios;
END
END

process main()
begin
  mi_proceso();
  otro_proceso();
end

process mi_proceso()
begin
  hola=20;
  adios=30;
  loop frame; end
end

process otro_proceso()
private
//Asignas el tipo mi_proceso a la variable "consulta" y ya está: ya puedes usar variables públicas
  mi_proceso consulta;
end
begin
  consulta=get_id(type mi_proceso);
  WRITE_var(0,100,100,4,consulta.hola);
  loop FRAME; end
end

```

En este ejemplo, declaro dos variables públicas llamadas “hola” y “adios” para el proceso “mi\_proceso()”, asignándoles de paso un valor inicial. Y desde otro proceso diferente, accedo al valor de una de estas variables -“hola”-, a partir del identificador del proceso “mi\_proceso()” obtenido gracias a la función `get_id`, y lo muestro por pantalla.

Fíjate (repito otra vez) que la manera de acceder al contenido de una variable pública desde otro proceso es idéntica a la manera de acceder al de una variable local, con la salvedad de que el identificador del proceso que tiene definida la variable pública a la que se quiere acceder ha de ser declarado no como `int` sino con el nombre de dicho proceso, tal como se puede observar también en el ejemplo.

Más en general, la sentencia `DECLARE`, como su nombre indica, sirve para declarar no ya variables, sino un proceso (o una función: en seguida veremos qué es lo que son) antes de utilizarlo en el código del programa. Es decir, `DECLARE` permite definir procesos y funciones antes de que vayan a ser implementados. Esto es útil, como hemos visto, cuando el compilador necesita saber de la existencia de ese proceso antes de que el código de éste lo haya leído.

La aplicación más habitual es la que hemos visto: declarar variables públicas de ese proceso en concreto, pero también es posible incluir dentro del bloque `DECLARE/END` un bloque `PRIVATE/END` para declarar también las variables privadas que utilizará el proceso en cuestión. **No obstante, hay que tener en cuenta que si se declaran variables privadas en un bloque `DECLARE`, no se podrá declarar después ninguna otra variable privada dentro del código específico de ese proceso.**

### Los comandos “`Let_me_alone()`” y “`Exit()`”:

Como bien sabes, nuestros programas constan de un código principal, a partir del cual se van llamando a diferentes procesos, y éstos a otros, etc. A partir de ahí, cada proceso “es como si” fuera un programa independiente que está ejecutándose en paralelo.

Esto quiere decir que, por ejemplo, si hacemos que el código principal termine en pulsar la tecla `ESC`, esto no quiere decir ni mucho menos que nuestro programa deje de ejecutar, ya que posiblemente tendremos multitud de procesos continuando su ejecución gracias a sus bucles `loop` infinitos. Es probable que hayas notado esta circunstancia en alguno de los ejemplos anteriores cuando intentaste acabar la ejecución de un programa y aparentemente éste no respondía: el código principal sí que había terminado pero el código de los diferentes procesos todavía estaba funcionando, a perpetuidad.

Para solucionar éste y otros posibles problemas que pudieran surgir, tenemos la función `let_me_alone()` --“déjame solo” literalmente-. `let_me_alone()` acaba con todos los procesos existentes SALVO el que ejecuta esta instrucción.

Si por ejemplo colocamos esta función justo al final del código principal, nos aseguramos de matar todos los procesos primero, para seguidamente acabar la ejecución del programa de forma limpia, sin tener que controlar cuántos procesos activos hay.

Esta solución es útil también cuando se quiere salir del juego y volver a alguna especie de menú, introducción o parecido.

Un ejemplo que creo que no necesita demasiada explicación (necesitas un archivo `FPG` llamado “`graficos.fpg`” con un gráfico en su interior de código 001 y 20x20 píxeles):

```
program Test_LET_ME_ALONE y EXIT;  
global  
int fpg;
```

```

    int bolas=1;
end
begin
    set_mode(640,480,16);
    fpg=load_fpg("graficos.fpg");
    write(0,160,30,4,"SPACE = elimina todos los procesos excepto el programa principal");
    write(0,160,50,4,"ENTER = crea de nuevo los procesos gráficos");
    bola(160,60);
    bola(200,100);
    bola(160,140);
    bola(120,100);
    repeat
/*La condición de "bolas==1" es para que sólo se realice el interior del if si los gráficos son visibles en
ese momento*/
        if(key(_space) && bolas==1)
            let_me_alone();
            bolas=0;
        end
/*La condición de "bolas==0" es para que sólo se realice el interior del if si los gráficos no son visibles
en ese momento*/
        if(key(_enter) && bolas==0)
            bola(160,60);
            bola(200,100);
            bola(160,140);
            bola(120,100);
            bolas=1;
        end
    frame;
    until(key(_esc))
    exit();
end

process Bola(x,y)
begin
    graph=1;
    loop
        frame;
    end
end
end

```

La función *let\_me\_alone()* es parecida a *exit()*, pero *exit()* -ya lo habrás visto en algún ejemplo anterior- acaba con TODOS los procesos existentes y por tanto, finaliza el juego, sin esperar ni siquiera al siguiente frame.

Existe una variable global predefinida interesante llamada **EXIT\_STATUS**, que pasa a valer 1 cuando el usuario trata de cerrar la aplicación desde el sistema operativo (por ejemplo, si el juego se ejecuta en una ventana, cuando pulsa el botón de cerrar de su barra de título). Su valor normal y original es 0. Para que un juego funcione correctamente y esté bien integrado en el sistema operativo, se recomienda consultar esta variable con un proceso y salir del juego (por ejemplo mediante la función EXIT cuando esta variable valga 1. La consulta debe realizarse cada frame, pues no se garantiza que la variable siga valiendo 1 en frames posteriores al intento de cerrar la aplicación.

## El comando “Exists()”:

Otro comando interesante relacionado con el reconocimiento y detección de procesos activos durante la ejecución del programa es el comando *Exists()*.

Esta función devuelve 1 (cierto) si el proceso cuyo identificador recibe como primer y único parámetro sigue en ejecución (esto es, no ha sido eliminado de memoria, ni marcado para matar mediante una función como SIGNAL), y devuelve 0 (falso) si no hay ni rastro de ese proceso.

Es posible pasarle como parámetro un tipo de proceso obtenido mediante la instrucción TYPE, en cuyo caso devolverá 1 si hay en memoria al menos un proceso del tipo indicado. En caso de no encontrar ningún proceso, EXISTS devolverá 0.

Un ejemplo de uso de este comando podría ser el siguiente (necesitas un archivo FPG llamado “graficos.fpg” con un gráfico en su interior de código 001 y 20x20 píxeles):

```
program Test_EXISTS;
global
  int fpg;
  int IDB;
  int ex;
  int ebola=1;
end
begin
  fpg=load_fpg("graficos.fpg");
  set_mode(640,480,16);
  write(0,160,30,4,"Proceso Bola Existente...");
  write_var(0,160,40,4,ex);
  write(0,160,60,4,"SPACE = Elimina el proceso 'Bola'");
  write(0,160,80,4,"ENTER = Vuelve a crear el proceso 'Bola'");
  IDB=Bola();
  repeat
    ex=exists(IDB);
    /*La condición de “ebola==1” es para que sólo se realice el interior del if si el proceso “Bola” existe en
    ese momento*/
    if(key(_space) && ebola==1)
      signal(IDB,s_kill);
      ebola=0;
    end
    /*La condición de “ebola==0” es para que sólo se realice el interior del if si el proceso “Bola” no existe
    en ese momento*/
    if(key(_enter) && ebola==0)
      IDB=Bola();
      ebola=1;
    end
  frame;
  until(key(_esc))
  let_me_alone();
end

process Bola()
begin
  graph=1;
  x=160;
```

```
y=100;
loop
  frame;
end
end
```

### **La cláusula ONEXIT:**

Hay casos donde el programador asigna recursos a un determinado proceso ( impresión de un texto, carga de un sonido o un fpg, creación de una tabla blendop,etc) y en un momento dado ese proceso es “killeado” por otro proceso diferente -por ejemplo: cuando un enemigo es matado por nuestra nave-. En ese momento, cuando ese proceso está a punto de morir, es posible que nos interese realizar la descarga de los recursos que estaba utilizando, porque se dé el caso que éstos ya nos los utilizará ningún proceso más. El hecho de que un proceso muera no implica que los recursos que esté utilizando sean descargados de la memoria, ni mucho menos, (con el desperdicio de memoria y el ralentizamiento del juego que esto comporta). Para facilitar esta labor, disponemos de la cláusula ONEXIT:

Su sintaxis es la siguiente:

```
Process ... //Cabecera del proceso
//Sección de declaraciones de variables
Begin
//Código del proceso
OnExit
//Código de salida (normalmente, para descargar recursos utilizados por el proceso)
End
```

Por lo que puedes ver, la cláusula OnExit se usa al final de un bloque BEGIN/END de un proceso. Y el código que hay posterior a ella se ejecutará cuando el proceso/función/programa principal termine su ejecución, ya sea de “muerte natural” al llegar al final de su código o bien porque haya recibido una señal Kill.

En realidad, después de la cláusula OnExit se puede poner cualquier código que se te ocurra, pero ya he comentado que su utilidad más evidente es escribir código de descarga de recursos. Piensa, por ejemplo, que si se escribe después de la cláusula OnExit algo como *Loop frame;End*, el juego se quedará bloqueado, ya que habrá un proceso que no terminará de morir nunca y se quedará en un estado inestable.

Un ejemplo:

```
Program example;
global
    int idfpg;
    int idproc;
end
Begin
  set_mode(640,480,16);
  idfpg=load_fpg("tetris.fpg");
  idproc=procesol();
  Repeat
    if(key(_x)) signal(idproc,s_kill); end
  frame;
  Until(key(_esc))
End
```

```

Process proceso1()
Private
  int idtexto;
End
Begin
  file=idfpg;
  graph =1;x=320;y=240;
  idtexto=write(0,100,100,4,"Hola");
  Loop
    frame;
  End
OnExit
  unload_fpg(file);
  delete_text(idtexto);
End

```

Otro ejemplo más ilustrativo, si cabe:

```

//AUTHOR:COLDEV
program hola;
private
  int i;
end
begin
  SET_MODE(640,480,16);
  i= test_process();
  loop
    if (key(_e))
      if (EXISTS(i)) SIGNAL(i,s_kill); end
    end
    frame;
  end
end

process test_process()
begin
  write(0,150,200,0,"Proceso creado - Pulsa (E) para matarlo");
  loop
    frame;
  end
OnExit
  write(0,150,300,0,"Proceso matado.");
end

```

**Sobre la ejecución paralela de los procesos. La variable local predefinida PRIORITY:**

Ejecuta el siguiente código:

```

program hola;
begin
  set_mode(320,240,16);
  proceso1();

```

```

proceso2());
loop
    if(key(_esc)) exit();end
frame;
end
end

process proceso1()
begin
x=320;
loop
if(key(_a)) x=x+1; end
delete_text(0);
write(0,20,10,4,x);
frame;
end
end

process proceso2()
begin
x=220;
loop
if(key(_b)) x=x+1; end
delete_text(0);
write(0,50,10,4,x);
frame;
end
end

```

Miremos qué hace este programa con detenimiento. Vemos que el código principal básicamente llama *PRIMERO* a “proceso1” y *DESPUÉS* a “proceso2”. Tanto “proceso1” como “proceso2” son “invisibles” porque no hemos asociado ningún gráfico con la variable GRAPH, pero eso no quita que estén funcionando como cualquier otro proceso. En concreto, a ambos le asignamos una coordenada X diferente. Y a partir de entonces, en ambos entramos en el bucle Loop. En “proceso1” lo que hacemos indefinidamente es borrar todos los textos de la pantalla, escribir el valor actual de su X y plasmarlo en un fotograma (con la opción previa de haber modificado su valor pulsando una tecla). En “proceso2” hacemos lo mismo: borramos todos los textos de la pantalla, escribimos el valor actual de su X y lo plasmamos en el fotograma.

Un rápido (y erróneo) análisis de este código puede llevar a pensar que lo que se tendría que imprimir en pantalla son los dos valores de cada una de las X de los dos procesos. Pero no. Sólo se imprime la X de “proceso2”. ¿Por qué?

He explicado en apartados anteriores que todos los procesos muestran en pantalla de forma sincronizada el resultado de sus líneas de código anteriores a cada línea frame; se encuentren en su ejecución. De forma sincronizada quiere decir que hasta que todos los procesos activos existentes no hayan llegado a una línea frame; el resto de procesos esperará en su línea frame; correspondiente: y cuando por fin todos los procesos hayan llegado a ella, es cuando el fotograma se muestra por pantalla ofreciendo a la vez el resultado de todos los cálculos efectuados en las líneas anteriores de cada proceso.

Pero hay que tener cuidado. Siguiendo esta interpretación, uno podría pensar del código anterior lo siguiente: pongo en marcha los dos procesos, en uno se borra toda la pantalla, se escribe su X y se llega al frame y en el otro se borra la pantalla, se escribe su X y se llega al frame, con lo que cuando llegaran al frame los dos procesos, uno ha de escribir su X y el otro la suya. ¡NO!

Porque el código de los procesos se ejecuta se forma secuencial: un código de proceso después de otro. Es decir, el primer proceso que sea creado desde el programa principal ejecutará primero sus líneas de código hasta llegar a su frame, momento en el que se para; el proceso que haya sido creado justo después del anterior es en ese momento cuando comenzará la ejecución de su código, hasta llegar a su frame; y así con todos los procesos hasta que todos hayan llegado al frame, momento en el que, entonces sí que sincronizadamente, se visualiza el resultado en pantalla.

Esto implica, en el ejemplo anterior, que primero se ejecutan las líneas de “proceso1” hasta el frame, y luego las líneas de “proceso2” hasta el frame, y es en ese momento cuando (como ya no quedan más procesos más) se visualiza el resultado. Así, lo primero que ocurre es que se borran todos los textos de la pantalla, luego se escribe el valor de X de “proceso1”, luego se vuelve a borrar toda la pantalla, luego se escribe el valor de X de “proceso2” y es en ese instante cuando se muestra en pantalla el resultado de todo esto. Puedes entender ahora perfectamente ahora por qué sólo se ve la X de “proceso2”: porque el *delete\_text(0)* que aparece en “proceso2” borra todos los textos anteriormente escritos, que en este caso es la X de “proceso1”.

Si cambias el orden de invocación de los procesos en el programa principal (es decir, llamas primero a “proceso2” y luego a “proceso1”), verás que ahora el resultado es al revés: sólo se ve en pantalla la X de “proceso1”, por el mismo motivo.

**NOTA:** evidentemente, existe una manera más óptima de visualizar el valor cambiante de una variable que la mostrada en el ejemplo anterior: escribiendo un *write\_var* fuera del loop de los procesos. Pero el código se ha escrito como excusa para demostrar la explicación dada de sincronía y secuencialidad.

Después de toda la explicación precedente, ya estamos en condiciones de entender que la orden Frame no dibuja nada, sino que lo que realmente es ceder la ejecución al próximo proceso: aunque aparentemente Fénix ejecute los procesos de forma concurrente, realmente el procesador sólo los puede atender de uno en uno, y para que se pase al siguiente proceso a ejecutar, o se termina con el proceso actual o se lee la instrucción Frame. Solamente cuando todos los procesos hayan terminado de ejecutarse o han llegado al Frame, se dibujan en pantalla, se espera el tiempo que sobre (ver el comando “set\_fps()”) y se pasa al siguiente Frame.

Seguramente te estarás preguntando ahora si no tenemos más remedio que limitarnos a respetar el orden de ejecución impuesto por el orden de creación de los procesos. ¿No hay ninguna manera de hacer que el código de un proceso se ejecute antes que el de otro, aunque éste haya sido creado antes que aquél? Pues sí, hay una manera: utilizando la variable local predefinida **PRIORITY**.

La variable local PRIORITY define el nivel de prioridad del proceso pertinente. Por defecto su valor es 0.

Los procesos que tienen una prioridad mayor se ejecutan antes ya que a la hora de dibujar cada FRAME del juego el intérprete utiliza el valor de esta variable para determinar el orden de ejecución. Recuerda que los procesos se ejecutan primero y sólo cuando ya no queda ningún proceso por ejecutar en el FRAME actual, se dibujan en pantalla.

Para que veas en acción esta variable, jugaremos con el ejemplo siguiente (necesitarás dos imágenes de unos 50x50 píxeles de tamaño, llamadas “dibujo.png” y “dibujo2.png”):

```
program prioridades;
```

```
process main()  
begin
```

```

set_mode(640,480,16);
procesol();
loop if(key(_esc)) exit();end frame; end
end

process procesol()
begin
graph=load_png("dibujo.png"); x=320;y=240;
procesos2();
loop
    if(key(_left)) x=x-10;end
    if(key(_down))y=y+10;end
    if(key(_right))x=x+10;end
    if(key(_up))y=y-10;end
    frame;
end
end

process procesos2()
begin
graph=load_png("dibujo2.png"); x=320;y=240;
loop
    x=father.x;
    y=father.y;
    frame;
end
end

```

En este ejemplo tenemos un proceso “procesol” que podremos mover con el cursor, y además es padre de “procesos2”, el cual, lo único que hace seguir en cada fotograma la posición de “procesol”.

Si ejecutas el ejemplo, verás que lo único que se ve es “procesos2”, ya que al ser el último proceso creado, éste siempre se pintará por encima de los demás (éste es un tema que ahora no nos interesa: ya volveremos a él cuando hablemos de la profundidad de pintado de los procesos y la variable local predefinida Z). Hay que notar que este hecho, el de que el último proceso creado es el que se pinta encima de los anteriores, siempre será así independientemente del valor de PRIORITY: para cambiar esto repito que lo que es necesario es utilizar otra variable que no tiene nada que ver, Z, de la cual ya se hablará más adelante.

Lo importante del asunto es que te fijas que, ya que “procesol” ha sido creado antes que “procesos2” (de hecho, éste es hijo de aquél), el código de “procesol” se ejecutará antes que el de “procesos2”, y una vez que ambos códigos hayan llegado a la orden FRAME -junto con la del proceso principal-, es cuando se mostrará el resultado por pantalla. Estudiemos esto con más detenimiento.

Si pulsamos una tecla del cursor, cambiamos la X e Y de “procesol”, y llegamos al Frame. Entonces pasaremos la ejecución al código de “procesos2”, en el cual se toman los valores actuales de X e Y de “procesol” y se asignan a la X e Y de “procesos2”, y llegamos al Frame. Una vez ya hemos llegado al Frame de todos los procesos, es cuando se pinta la pantalla: en este momento la X e Y de “procesol” y de “procesos2” valen exactamente lo mismo, por lo que veremos lo que ya hemos comentado: “procesos2” se mueve a la vez que “procesol” y lo tapa.

Pero, ¿qué pasa si añadimos la línea siguiente justo después del BEGIN de “procesos2”?

```
priority=1;
```

Estaremos haciendo que la prioridad de “proceso2” sea mayor que la de “proceso1” ( $1 > 0$ ). Así pues, la cosa cambia. Si ejecutas ahora el programa, verás que cuando pulsas los cursores, parece que el gráfico de “proceso1” se avanza un poquito al de “proceso2”, dejándose entrever; aunque cuando dejamos de pulsar los cursores ambos gráficos vuelven a superponerse perfectamente. ¿Por qué es esto?

Ahora el código de “proceso2” se ejecuta primero hasta la línea Frame; después se ejecuta el código de “proceso1” hasta el Frame y cuando todos los procesos han llegado a su Frame, se pinta por pantalla el resultado. Estudiemos esto con más detenimiento.

Está claro que “proceso2” tomará la X e Y de la posición que en este momento tiene su padre (“proceso1”), la cual puede cambiar si se está pulsando los cursores. Pero “proceso1” se ejecuta después que “proceso2”, con lo que en cada frame, la X e Y de “proceso2” será la X e Y que tenía “proceso1” en el frame anterior, por lo que “proceso2” siempre irá retrasado un frame respecto a “proceso1”, y es por eso el efecto. Es decir, que “proceso2” no se enterará de que su padre ha cambiado su posición hasta que no pase el siguiente frame, porque el código de su padre siempre se ejecuta después de el suyo propio, con lo que los movimientos que pueda sufrir el padre ya nos notará hasta que se vuelva a reiniciar una vuelta entera de ejecuciones para el nuevo frame. Parece un poco lioso pero si lo piensas un poco verás que tiene su lógica.

### **El parámetro de la orden Frame:**

En el capítulo 2 ya se vió un ejemplo de uso del parámetro -opcional- de la orden Frame; pero ya que posiblemente éste es uno de los conceptos que más cuesta de entender al iniciado en Fénix, no está de más mostrar un ejemplo específico. Es un código bastante simple con el que confío se acabe de entender la utilidad de este parámetro.

El parámetro del FRAME lo que indica es cuántos fotogramas más han de pasar extra para que la orden frame sea realmente efectiva. Es decir, si no ponemos ningún parámetro, o ponemos el valor 100 —es lo mismo—, cada vez que se llegue a la línea FRAME, el fotograma de ese proceso está listo para ser imprimido de ipsofacto (a falta de esperar que los otros posibles procesos lleguen también a sus respectivas líneas FRAME).

Si se pone otro número como parámetro del FRAME, por ejemplo el 500, lo que ocurrirá es que cuando se llegue a la línea FRAME, el fotograma del proceso no estará listo inmediatamente sino que ese proceso esperará a que los demás procesos hayan ejecutado 5 veces una línea FRAME para ejecutar él la suya (es decir, esperará a que se hayan visualizado 5 fotogramas por parte de los demás procesos antes de visualizar nada él). Es como si el programa se quedara esperando un rato más largo en la línea FRAME, hasta ejecutarla y continuar.

Así pues, este parámetro permite, si se da un valor mayor de 100, ralentizar las impresiones por pantalla de un proceso determinado. También es posible poner un número menor de 100. En ese caso, lo que estaríamos haciendo es “abrir la puerta” más de una vez en cada fotograma: si pusiéramos por ejemplo 50, estaríamos doblando la velocidad normal de impresión por pantalla. Puedes comprobarlo.

Aquí presento un ejemplo que espero que ayude a entender este concepto:

```
Program ejemplo_avisos;
Global
  int nivel_actual;
end
Local
  Int info=1;
end
```

```

Begin
  set_mode(640,480);
  set_fps(10,0); //Diez pantallazos por segundo.
  titulo_nivel(info);
End

Process titulo_nivel(int info)
Private
  int id_text;
  int id_numero;
  int contador=6;
end
Begin
  Repeat
    If(key(_esc)) exit();End
    id_text=write(0,340,140,5,"NIVEL");
    id_numero=write_var(0,370,140,3,info);
    Frame(400); // Espera 4 pantallazos
    delete_text(id_text);
    delete_text(id_numero);
    Frame(400); // Espera 4 pantallazos
    contador=contador-1;
    // el aviso se repite 5 veces
  Until (contador<1)
  nivel();
End

Process nivel()
Begin
  info=nivel_actual;
  nivel_actual=nivel_actual+1;
  Frame (8000); //pausa de 80 pantallazos (8 segundos a FPS=10)
  titulo_nivel(nivel_actual+1);
  write(0,50,50,0,"CUANDO TE CANSES DE AVISOS PULSA Esc");
  Frame;
End

```

### **Concepto y creación de una función:**

Una función es un comando que ejecuta un cierto código y que al finalizar nos devolverá un resultado numérico entero, (el cual podremos recoger, o no, según nos interese).

Una función es una manera para “encapsular” con un nombre una porción de código fuente de forma que cuando sea necesario éste, y las veces que sea necesario, no haya que escribirlo siempre íntegramente sino que simplemente se llame (se “invoque”) por su nombre y entonces su código interno –escrito solamente una vez- se ejecutará. Utilizar funciones es una forma de trabajar mucho más elegante, eficiente y fácil de depurar.

Hemos visto varias funciones hasta ahora: load\_png, key, let\_me\_alone... Estas funciones son pequeños programas en sí mismos que se ejecutan siempre que las llamamos: a nosotros no nos interesa cómo se han escrito internamente estas funciones: lo que nos interesa es que realicen una determinada tarea, devolviendo o no un valor numérico entero que podremos utilizar posteriormente para lo que consideremos oportuno.

Las funciones no son capaces de devolver otra cosa que no sea un número entero. No devolverán números decimales o cadenas (a no ser que utilicemos punteros, un tema demasiado avanzado para este manual).

Pues bien, nosotros también podemos crear nuestras propias funciones. Pero, ¿qué diferencia hay entre crear una función y crear un proceso?

Cuando creamos un proceso, este nuevo proceso se pondrá en marcha a la vez que continúa el proceso padre, de manera que los dos procesos se ejecutarán de forma paralela, (concurrente). Cada uno de estos procesos tendrá el comando Frame; y cada uno de ellos llegará por su cuenta a este comando sincronizándose en ese momento y ejecutándose pues a la vez en cada fotograma.

Una función es diferente: una función es un trozo de código igual que un proceso, pero cuando una función es llamada, el proceso que la llamó se detiene automáticamente (se “congela”) hasta que la función acabe de hacer lo que tiene encomendado. En ese instante, el proceso que la llamó se “despierta” automáticamente y continúa la ejecución de su propio código.

Por lo tanto, las funciones NUNCA tienen la orden Frame;, porque no son más que trozos de código que se ejecutan de forma secuencial al estilo C, parando mientras tanto la ejecución del proceso que lo llamó. Piensa un poco: por ejemplo, la función load\_png (y todas las demás) hacen esto: hasta que no se haya cargado una imagen PNG el código del proceso no continúa.

Acabamos de decir que mientras se está ejecutando el código del interior de una función, el proceso que la ha llamado se queda en espera de que la función termine -y opcionalmente, devuelva algún valor-, para continuar en la siguiente de su propio código. En principio, pues, el llamar a una función sólo “paraliza” a un proceso, pero fíjate que como en Fénix hasta que todos los procesos no han llegado a una orden Frame; no se pasa al siguiente fotograma, si existe un proceso que se ralentiza por el uso de una determinada función -es posible que ocurra-, esto provocará el ralentizamiento general del juego ya que los demás procesos están esperando a aquél.

Lo acabado de comentar implica algo muy importante: las funciones no pueden contener bucles infinitos bajo ningún concepto. La razón es evidente: si una función tuviera un bucle infinito, nunca finalizaría su ejecución y por tanto todos los procesos del programa permanecerían congelados en espera un tiempo infinito, con lo que tendríamos nuestro programa “colgado”.

Que las funciones nunca incorporan en su código la orden Frame no es cierta del todo. Pueden incorporarla, pero entonces gran parte (aunque no todas) de sus características se asimilan a la de un proceso normal y corriente, por lo que la diferencia entre ambos se hace más ténue y no resultan tan útiles. En este curso no veremos ninguna función con Frame.

Así pues, ¿qué hay que escribir para crear una función?

Las funciones se crean igual que un proceso cualquiera, como los que hemos creado hasta ahora, pero tienen dos o tres pequeñas grandes diferencias:

**Nuestra función NO ha de tener ningún comando FRAME ni ningún bucle infinito** (ya hemos comentado este aspecto).

**En vez de escribir la palabra PROCESS en la cabecera de nuestra función, escribiremos la palabra FUNCTION.** Opcionalmente, igual que los procesos, puede recibir parámetros. En ese caso, la regla de escritura de éstos en la cabecera son las mismas que para los procesos.

**Nuestra función podrá devolver opcionalmente un valor numérico entero. Para ello, se utilizará el comando *RETURN(valor\_numérico)*. Pero cuidado porque con RETURN se acaba la ejecución de la función, allí donde esté, por lo que hay que saberla colocar en el lugar adecuado.**

Lo que hace la sentencia RETURN() es finalizar –“matar”- la función actual de forma inmediata, como si se llegara al END de su BEGIN.

Se pone entre paréntesis un valor –escrito explícitamente o bien a través de una variable que lo contenga- que queremos devolver al proceso que llamó a dicha función (el cual recogerá ese valor en alguna variable suya, seguramente). Es decir, escribir RETURN(29); hará que la función finalice su ejecución y devuelva al proceso invocante el valor 29; y ese proceso recogerá ese valor y ya hará con él lo que estime oportuno.

Pero el valor escrito entre los paréntesis del RETURN es opcional. También no se puede escribir nada, dejando los paréntesis vacíos (o incluso, que es lo mismo, sin escribir ni siquiera los paréntesis: sólo RETURN;). En este caso, la función termina su ejecución igual que en la otra manera, pero ahora el valor de retorno de la función -que no se suele utilizar demasiado en realidad- será el código identificador de ésta (ya que, las funciones al igual que los procesos tienen cada una un código identificador).

Si la función carece de sentencia RETURN (que es perfectamente posible), cuando llegue al final de su código -el último END- también devolverá el código identificador de dicha función al proceso que lo llamó.

La sentencia RETURN de hecho también se puede utilizar dentro del código perteneciente a un proceso, aunque sólo en su forma más simple (es decir, sin escribir ningún valor dentro de los paréntesis), así RETURN;

Esta sentencia dentro de un proceso tendrá el mismo efecto que dentro de una función: finaliza inmediatamente la ejecución de su código (y por tanto, en el caso de los procesos, destruye esa instancia concreta de proceso de la memoria).

Si escribes la sentencia RETURN en el código principal lo finalizará, pero recuerda que si quedan procesos vivos éstos se seguirán ejecutando. Ya sabes que finalice la ejecución del código principal no implica que finalice la ejecución del programa, pues ésta continuará si no se ha tenido la precaución de matar todos los procesos existentes. Por eso, acuérdate en estos casos de utilizar la función *let\_me\_alone()* justo antes del END que marca el final del código principal, para eliminar al resto de procesos que puedan quedar vivos.

Recuerda además que también se puede finalizar en cualquier punto del programa su ejecución mediante la función *exit()* que automáticamente eliminará todos los procesos.

Es evidente que, al no ser un proceso, las funciones no tienen definidas ninguna de las variables locales predefinidas de cualquier proceso: así que GRAPH,X,ANGLE...no tendrán ningún sentido para una FUNCTION.

Posiblemente, después de haber leído los párrafos anteriores tan técnicos no creo que hayas entendido muy bien todavía cómo funcionan las funciones. Básicamente lo que tienes que recordar es que para escribir una función (que no deja de ser una porción de código que no se visualiza en pantalla pero que realiza algún determinado cálculo) se ha de hacer como si escribiéramos un proceso normal, pero éste NO ha de tener la orden FRAME, la cabecera ha de poner FUNCTION en vez de PROCESS y si se pretende

que la función retorne algún resultado, ha de tener la orden RETURN(valor numérico), o simplemente RETURN si se quiere salir del código de la función en algún momento antes de su END.

Un ejemplo, que espero que lo aclare:

```
program hola;
private
    int c;
end
begin
    loop
        c=mifuncion(rand(2,8));
        write_var(0,100,100,4,b);
        frame;
    end
end

function mifuncion(int a)
private
    int b;
end
begin
    b=a*a;
    return (b);
end
```

Lo que hemos hecho aquí es crear un nuevo comando o función llamada “mifunción” que recibe un parámetro entero. El objetivo de esta función es realizar un cálculo y devolver el resultado (en este caso, la multiplicación por sí mismo del valor pasado por parámetro). Fíjate que en el programa principal se recoge lo que devuelve “mifunción” en una variable y a partir de aquí se continúa ejecutando el programa principal.

La gracia de este ejemplo es ver que el programa principal no continúa hasta que “mifunción” haya acabado (se deja de ejecutar el código del programa principal para comenzarse a ejecutar el código de la función “mifunción”, hasta que éste acabe), y que, como siempre, la sincronización con otros procesos (si el proceso principal se ha retrasado un pelín respecto los demás) se realizará en su frame; correspondiente.

Por experiencia, aconsejo que si se quiere que una función devuelva algún valor, éste se guarde en una variable privada de la función, que será la que se coloque entre paréntesis del RETURN, tal como he hecho en el ejemplo. Como ves, no hay ninguna necesidad de nombrar a las diferentes variables involucradas (“a”, “b”, “c” en el ejemplo) de la misma manera, porque cada una de ellas son variables diferentes del resto.

A continuación presento un código rebuscado a propósito: no tiene ninguna utilidad práctica pero sirve para comprobar si se ha entendido todo lo explicado hasta ahora. ¿Qué es lo que saldrá por pantalla?

```
program fafafa;
global
    int v=2;
    int w;
end
```

```

begin
    p();
end

function p()
begin
    v=3;
    q();
end

function q()
begin
    w=v;
    r();
end

process r()
begin
    write(0,1,1,0,w);
    loop
        frame;
    end
end

```

Las funciones pueden ser recursivas: no existe ningún inconveniente (de hecho, los procesos también). Una función recursiva es la que dentro de su código se llama a sí misma. Un ejemplo, (original de Animanegra):

```

program hola;
private
    int resultado;
end
begin
    resultado=prueba(4);
    write(0,100,100,4,resultado);
    loop frame; end
end

function prueba(int a)
begin
    if(a < 10)
        a = prueba(a+1);
    end
return(a);
end

```

¿Sabrías razonar por qué se ve un “10” en la pantalla si ejecutas el código anterior?

Cuando hablamos de las variables públicas comentamos la existencia de la sentencia DECLARE, que servía para hacer saber a Fénix de la existencia de un proceso antes de llegar a su definición, escrita posteriormente en el código. Pues bien, podemos hacer lo mismo con funciones. Para utilizar la sentencia DECLARE con funciones, su sintaxis es la siguiente:

```
DECLARE FUNCTION tipoDatoRetornado nombreFuncion (tipo param1, tipo param2,...)
...
END
```

donde los parámetros que se especifiquen serán los mismos que tiene la propia función, y tipoDatoRetornado es el tipo de datos del valor que será devuelto con la sentencia RETURN (si es que ésta lo hace).

Un ejemplo:

```
Program lala;
Declare Function string ejemplofunc( int param1)
  Private
    int privint;
  End
End
Private
  string texto;
End
Begin
  texto=ejemplofunc(1);
  write(0,150,10,4,texto);
  texto=ejemplofunc(2);
  write(0,150,20,4,texto);
  loop
    if(key(_esc)) exit(); end
  frame;
  end
End

Function string ejemplofunc( int param1)
Begin
  privint=param1;
  if (privint==1) return ("¿Qué taaaal?!?!");end
  if (privint==2) return ("¡¡¡Muy bien!!!!");end
End
```

En el ejemplo anterior hemos usado la sentencia DECLARE para declarar una variable privada (“privint”) de la función “ejemplofunc”.

Si lo que queremos es otra cosa diferente, que es utilizar dentro de una función una variable pública de un proceso externo, lo deberíamos hacer de la siguiente manera:

```
DECLARE process coche()
public
Int gasolina;
end
END

function coche_llenar_gasolina(coche idcoche)
begin
idcoche.gasolina=100;
```

*end*

En este caso, tenemos una función “coche\_llenar\_gasolina” con un parámetro que será el identificador de una instancia de un supuesto proceso “coche”. Fíjate (ya lo hemos visto otras veces) que dicho parámetro está definido no como INT sino como el nombre del proceso:”coche”. La función lo que hace es modificar el valor de una variable pública llamada “gasolina”, de esa instancia concreta del proceso “coche”.

En párrafos anteriores se ha comentado que las funciones sólo son capaces de devolver valores enteros (a no ser, repito, que se utilicen punteros, cosa que no veremos). No obstante, ya debes saber que en Fénix hay varios tipos de datos que se corresponden con valores enteros: BYTE, WORD, INT, DWORD...¿Cuál es el tipo de entre éstos que se devuelve realmente?. Por defecto, si no se especifica nada, este tipo será INT. Pero es posible especificar otro tipo de datos -siempre que sea entero- para el valor de retorno de una función. Y ¿esto cómo se hace?.

Simplemente hay que indicar el tipo de datos a devolver en la cabecera de la función, así:

*FUNCTION **tipodatosretorno** nombrefuncion(tipoparam1 nombreparam1, tipoparam2 nombreparam2,...)*

y ya está. Con el siguiente ejemplo se verá mejor:

*program devoluciones;*

*global*

*int i;  
byte b;  
word w;  
dword dw;  
float f;  
string c;  
tipo t;*

*end*

*process main()*

*begin*

*write(0,150,60,4,cero());  
write(0,150,70,4,uno());  
write(0,150,80,4,dos());  
write(0,150,90,4,tres());  
write(0,150,100,4,cuatro());  
write(0,150,110,4,cinco());  
write(0,150,120,4,seis());*

*Loop frame; end*

*end*

*//Una función típica de las de toda la vida*

*function cero()*

*begin*

*i = 0;return i;*

*end*

*//Equivalente a la función "cero"*

```

function int uno()
begin
i = 1;return i;
end

//Especificamos explícitamente que el valor a devolver es BYTE
function byte dos()
begin
b = 2;return b;
end

//Especificamos explícitamente que el valor a devolver es WORD
function word tres()
begin
w = 3;return w;
end

//Especificamos explícitamente que el valor a devolver es DWORD
function dword cuatro()
begin
dw = 4;return dw;
end

//No devuelve lo que uno esperaría. No funciona
function float cinco()
begin
f = 5.5; return f;
end

//No devuelve lo que uno esperaría. No funciona
function string seis()
begin
c="Seis"; return c;
end

```

### **La sentencia #DEFINE:**

Esta sentencia permite definir -“crear”- macros ( más exactamente, asignar un texto a una macro). Y una macro es simplemente una palabra que, una vez definida, al aparecer más adelante en el código es sustituida por un texto determinado, especificado también en la sentencia #define..

Una macro es similar a una constante, pero tiene alguna diferencia importante. La más evidente es que, a diferencia de en las constantes, los valores de las macros pueden ser expresiones aritméticas/lógicas donde pueden intervenir diferentes operandos, los cuales pueden tener un valor diferente cada vez que la macro sea utilizada, ya que éstas -otra diferencia con las constantes- pueden admitir parámetros. En este sentido, una macro se parecería más a la definición de una función extremadamente simple.

La sintaxis de la sentencia #DEFINE es la siguiente:

```
#DEFINE Macro(param1,param2,...) ExpresiónAsignadaALaMacro
```

Los paréntesis que incluyen los parámetros de la macro son opcionales.

¡Fijarse que no escribimos el punto y coma al final (aunque si lo escribes tampoco pasa nada! Ni tampoco especificamos el tipo de los parámetros: siempre serán Int.

Un ejemplo:

```
#define CALCULO 2+3
Program lala;
Begin
  write(0,100,100,4,4*CALCULO);
    loop
      frame;
    end
End
```

En este caso hemos definido una macro con el nombre de “CALCULO”, sin parámetros: cada vez que aparezca la palabra CALCULO en el código será sustituida por su valor, que es 2+3.

Fíjate en un detalle. El resultado mostrado es 11, ya que, evidentemente,  $11=4*2+3$ . Pero, ¿qué resultado obtenemos si ejecutamos este código?

```
#define CALCULO (2+3)
Program lala;
Begin
  write(0,100,100,4,4*CALCULO);
    loop
      frame;
    end
End
```

Obtenemos 20, ya que  $20=4*(2+3)$ . Como puedes ver, es muy importante el uso de los paréntesis en expresiones matemáticas definidas en macros, porque un despiste nos puede hacer que obtengamos valores que no esperábamos.

Otro ejemplo, con una macro que admite parámetros:

```
#define CALCULO(res,a,b) res=a+b
Program lala;
Private
  int mivar;
End
Begin
  CALCULO(mivar,2,3);
  write(0,100,100,4,4*mivar);
    loop
      frame;
    end
End
```

En este caso hemos utilizado una macro casi como si de una función se tratara. Puedes ver que el resultado del cálculo lo asignamos a una variable “mivar”. Fíjate también ahora que poner paréntesis o no ponerlos es lo mismo, ya que “res” (y por tanto, “mivar”), siempre va a valer 5.

Podemos complicar las macros con parámetros hasta un nivel de sofisticación que lleguen a tener

una versatilidad prácticamente igual a la de una función estándar con la ventaja además de que las macros, por su naturaleza, se ejecutan más rápidamente que las funciones. Por ejemplo, a ver si sabes interpretar lo que obtienes de ejecutar el siguiente código:

```
#define operacion(a,b,c) if(b==0) b=c+1; else a=b+1; end
program hola;
private
  int var1,var2=3;
end
begin
  operacion(var1,var2,3);
  write(0,100,100,4,var1);
  write(0,100,120,4,var2);
  loop FRAME; end
end
```

Por cierto, para que no cometas posibles errores en tus futuras macros, has de saber que si asignas algún valor a alguno de sus parámetros dentro de la macro (como hacemos en el ejemplo anterior con “a” y “b”), cuando invoques a la macro dentro del código, el lugar que ocupan dichos parámetros deberá ser rellenado con nombres de variables (que recogerán el valor asignado correspondiente) y no con valores explícitos, porque si no dará un error. De hecho, esto es un comportamiento lógico y esperable, ya que si lo piensas bien, no tiene sentido hacerlo de otra manera.

Seguramente te estarás preguntando por qué esta sentencia tiene ese misterioso símbolo “#” delante. Bien, este símbolo indica que el “comando” DEFINE en realidad es lo que se llama una directiva de preprocesador.

El preprocesador es un pequeño programa que filtra el contenido de tu código antes de pasárselo al compilador, y una de sus misiones es interpretar ciertas directivas que pueden estar presente en cualquier punto del programa -aunque se recomienda por claridad que estén al principio de todo, antes incluso de la línea “program...” -y se que caracterizan por ocupar una línea completa y empezar por el carácter # al principio de la misma (opcionalmente, precedido por espacios).

Así pues, cada vez que el preprocesador encuentre una línea que contiene una macro previamente definida, la sustituye por su valor real, y una vez ha acabado todo este proceso, es cuando transfiere el código fuente al compilador para que éste realice su función.

Aparte de la directiva #DEFINE, existen unas cuantas más, como por ejemplo #IFDEF, que sirve para crear secciones de código compilables opcionalmente (es decir, zonas que sólo se compilan si se cumple una condición determinada), entre otras. Para más información, consulta la ayuda online de Fénix.

## CAPÍTULO 5: TABLAS Y ESTRUCTURAS

Las clases de datos que hemos explicado en el capítulo anterior (las variables de todo tipo y las constantes) son las que hemos utilizado en los ejemplos hasta ahora exclusivamente. Éstos son los datos más simples: un nombre asociado con valor guardado en una determinada posición de la memoria del ordenador, cuyo contenido puede variar (variables) o no (constantes). No obstante, existen tipos más complejos de datos: las tablas y las estructuras.

### Tablas

Una tabla es una lista de variables, es decir, una tabla es un nombre que se relaciona, no con una sola posición de memoria (un solo valor) sino con tantas (tantos valores) como sea necesario.

Igual que las variables, las tablas pueden ser globales, locales, privadas o públicas.

Para definir por ejemplo una tabla global formada por números enteros en un programa se hace una declaración –e inicialización- como la siguiente:

```
GLOBAL
        Int mitabla1[3]=0,11,22,33;
END
```

Estas líneas declararían en un programa "mitabla1" como una lista de 4 variables. Se debe tener en cuenta que se comienza a contar siempre desde la posición 0, y esta tabla tiene hasta la posición 3, como indica el número entre los símbolos [ ] (que se denominan corchetes y no se deben confundir con los paréntesis): ese número indica pues el número de la última posición. En este ejemplo, cuatro posiciones de la memoria del ordenador se reservarían para "mitabla1", y se pondría inicialmente (antes de comenzar el programa) los valores, 0, 11, 22 y 33 en dichas posiciones.

Esta inicialización no es necesaria: también se podría haber declarado la tabla y ya está:

```
GLOBAL
        Int mitabla1[3];
END
```

en cuyo caso todos los valores de los elementos de la tabla valdrían 0 por defecto.

Hay que saber que si en la declaración de la tabla a la vez se inicializa (es decir, ya se introducen los valores de los elementos, como es el primer caso tratado), el número entre corchetes es opcional ponerlo, ya que si por ejemplo en ese momento se introdujeran cinco valores, Fénix reservaría sitio para cinco elementos en esa tabla, ni uno más ni uno menos, igual que si hubiéramos escrito un "4" entre los corchetes.

Evidentemente, nada evita que se pueda inicializar una tabla con un determinado número de valores y poner entre corchetes un número mayor de elementos: todos aquellos elementos que no sea rellenados por los valores en ese momento serán puestos a 0.

Cuando en un programa se tiene que consultar o modificar una de estas posiciones de memoria, se debe indicar el nombre de la tabla y, entre los corchetes, un valor numérico para especificar qué posición de la tabla se quiere consultar o modificar. Por ejemplo, una sentencia para poner el valor 999 en la posición 1 de "mitabla1" (que inicialmente valdría 11) sería como sigue:

```
mitabla1[1]=999;
```

Asimismo, sería posible definir tablas de mas dimensiones, mediante corchetes extra; por ejemplo, en una tabla global entera:

```
GLOBAL
      Int mitabla2[2][4];
END
```

crearía una tabla con 3 filas x 5 columnas = 15 posiciones de memoria.

Fíjate que todos los elementos de cualquier tabla han de ser del mismo tipo: o enteros, o decimales, o cadenas... No puede haber dentro de una misma tabla elementos de distintos tipos.

Las tablas son muy útiles cuando necesitamos almacenar muchos valores que de alguna manera están relacionados entre sí por su significado o propósito. Podríamos trabajar igualmente con variables “normales”, pero es mucho más fácil declarar una única tabla (un nombre) con 100 posiciones por ejemplo que no declarar 100 variables diferentes (100 nombres) para guardar cada uno de esos 100 valores.

Un sinónimo de “tabla” es la palabra “array”. También verás que a veces se les llama “vectores” a las tablas de una dimensión, y “matrices” a tablas de dos o más dimensiones.

Pongamos un ejemplo donde se vea todo más claro. En el siguiente código (imaginemos que estamos dentro de una sección PRIVATE/END) hemos declarado cinco tablas, cada una de forma distinta.

```
int tabla1[]=5,3,65;
int tabla2[4];
string tabla3[]="hola","que","tal";
string tabla4[4];
int tabla5[2][3];
```

La primera es una tabla (un vector) de elementos enteros a la que inicializamos en el mismo momento de declararla; al no haber especificado entre corchetes el número de elementos, ésta tendrá los mismos que valores se le introduzcan en la inicialización, es decir, 3.

La segunda también es un vector de enteros, pero con un número determinado de elementos -cinco- y que no se inicializa, por lo que los valores iniciales de estos cinco elementos será 0.

La declaración de la tercera tabla es similar a la de la primera con la salvedad de que aquélla es un vector de cadenas. De igual manera, la cuarta tabla es una tabla de cinco cadenas, inicialmente vacías.

La quinta tabla es una tabla bidimensional (una matriz). Imagínatela como si fuera la rejilla de una hoja de cálculo o similar, formada por celdas dispuestas en filas y columnas. Cada celda vendrá identificada por dos índices, que vienen dados por los números dentro de los dos corchetes; el primer corchete indicará el número de la fila donde está situada la celda, y el segundo corchete indicará el número de la columna donde está situada la celda, empezando siempre por 0, y desde arriba hacia abajo y desde izquierda a derecha.. En la declaración de la tabla, el primer corchete indicará pues el último elemento de todas las filas -por lo tanto, nuestra tabla tiene tres filas- y el segundo corchete indicará el último elemento de todas las columnas -por lo tanto, nuestra tabla tiene cuatro columnas-.

Las dos preguntas más evidentes cuando trabajamos con tablas es: ¿cómo recupero los distintos

valores de todos los elementos de una tabla? y ¿cómo puedo asignar nuevos valores a dichos elementos?

La respuesta a la primera pregunta es: mediante un bucle que recorra la tabla elemento a elemento y vaya trabajando como sea necesario con el valor que obtiene en cada iteración. El bucle que se utiliza más típicamente a la hora de recorrer tablas es el bucle FOR, porque sabemos cuándo empezamos (en el primer elemento) y sabemos cuándo acabamos (en el último elemento, que sabemos qué posición ocupa).

Así, la manera más normal de recorrer un tabla -la "tabla1" anterior, en este caso- y mostrar por ejemplo por pantalla los distintos valores de los elementos por los que se va pasando, sería algo así:

```
for (i=0;i<3;i++)
    write(0,100,10+10*i,4,tabla1[i]);
end
```

Fíjate en un detalle muy importante. Este for va a realizar tres iteraciones, cuando i valga 0, 1 y 2, ya que en el segundo parámetro del for se especifica que i ha de ser siempre MENOR que 3. Y lo hemos hecho así precisamente porque sabemos que "tabla1" tiene sólo 3 elementos, tabla1[0], tabla1[1] y tabla1[2]. Es decir, que hubiera sido un error haber puesto algo así como (es bastante frecuente que pase):

```
for (i=0;i<=3;i++)
    write(0,100,10+10*i,4,tabla1[i]);
end
```

ya que en este bucle se haría una iteración de más, en la cual tabla1[3] no existe y provocaría un error. Hay que tener ojo con esto.

De igual manera, podemos recorrer cualquier otro tipo de tablas, como las de cadenas. Si probamos con "tabla4":

```
for (i=0;i<5;i++)
    write(0,100,10+10*i,4,tabla4[i]);
end
```

veremos que no aparece nada en pantalla. Esto es porque esta tabla está vacía y no tiene ningún valor que mostrar (a diferencia de las tablas numéricas, las tablas de cadena no se rellenan automáticamente con ningún valor).

Podemos ya por tanto escribir un programa completo que recorra las cuatro tablas unidimensionales de este ejemplo y muestre por pantalla los valores almacenados en ellas (si es que tienen), así:

```
program ejemplo;
private
    int tabla1[]=5,3,65;
    int tabla2[4];
    string tabla3[]="hola","que","tal";
    string tabla4[4];
    int tabla5[2][3];
    int i;
end
begin
    set_mode(640,480,16);
    //Recorremos la primera tabla
    for (i=0;i<3;i++)
        write(0,100,10+10*i,4,tabla1[i]);
```

```

end

//Recorremos la segunda tabla (vacía)
for (i=0;i<5;i++)
    write(0,200,10+10*i,4,tabla2[i]);
end

//Recorremos la tercer tabla
for (i=0;i<3;i++)
    write(0,300,10+10*i,4,tabla3[i]);
end

//Recorremos la cuarta tabla (vacía)
for (i=0;i<5;i++)
    write(0,400,10+10*i,4,tabla4[i]);
end
loop
    frame;
end
end

```

La segunda pregunta que teníamos era que cómo podemos asignar nuevos valores a los elementos de una tabla. Es bien sencillo. Tal como se ha comentado anteriormente, si se quiere cambiar el valor de un elemento concreto, simplemente hay que asignarle su valor como haríamos con cualquier otra variable. Por ejemplo, así:

```
tabla1[1]=444;
```

asignaríamos el valor 444 al elemento 1 (el segundo) de la tabla "tabla1". Si quisiéramos cambiar los valores de un conjunto de elementos de la tabla, otra vez tendríamos que recurrir a un bucle para ir elemento a elemento y cambiar su valor. Por ejemplo, si haces así:

```

for (i=0;i<3;i++)
    tabla1[i]=444;
    write(0,100,10+10*i,4,tabla1[i]);
end

```

verás como antes de imprimirse por pantalla los valores de los elementos de "tabla1" se les da a todos por igual el valor de 444, sobrescribiendo el valor anterior que tuvieran. Si introduces este bucle en el ejemplo anterior podrás comprobar como se imprimirán 3 números 444 correspondientes a los 3 elementos de "tabla1".

Pero, ¿qué pasa con la tabla bidimensional? ¿Cómo se asignan los valores? Podríamos asignarlos a la vez que se declara:

```

program ejemplo;
private
    int tabla5[2][3]=1,2,3,4,5,6,7,8;
end
begin
    set_mode(640,480,16);
    write(0,100,10,4,tabla5[0][0]);
    write(0,100,20,4,tabla5[0][1]);
    write(0,100,30,4,tabla5[0][2]);

```

```

write(0,100,40,4,tabla5[0][3]);
write(0,100,50,4,tabla5[1][0]);
write(0,100,60,4,tabla5[1][1]);
write(0,100,70,4,tabla5[1][2]);
write(0,100,80,4,tabla5[1][3]);
write(0,100,90,4,tabla5[2][0]);
write(0,100,100,4,tabla5[2][1]);

```

```

loop
    frame;
end
end

```

Fíjate el orden que se sigue para ir llenando los elementos: primero es el [0][0], después el [0][1], después el [0][2], luego el [1][0], y así.

También se puede asignar un valor en mitad del código a un elemento concreto:

```

tabla5[1][2]=444;

```

O también se puede asignar nuevos valores a múltiples elementos recorriendo de arriba a abajo todos los elementos de la matriz. Y ¿eso cómo se hace? Si piensas un poco, lo deducirás. Si hemos necesitado un bucle para recorrer los elementos uno detrás de otro de un vector, necesitaremos dos bucles para recorrer las filas por un lado y las columnas por otro de esa matriz bidimensional. No obstante, la forma correcta de hacerlo puede que te sorprenda un poco: estos dos bucles irán uno dentro de otro (lo que se llaman bucles anidados). Vamos a poner primero un ejemplo de recorrido de una matriz para leer los valores de sus elementos (hace lo mismo que el ejemplo anterior pero usando bucles), y luego pondremos otro ejemplo que no los lea, sino que los modifique por un valor dado. El ejemplo de lectura es:

```

program ejemplo;
private
    int tabla5[2][3]=1,2,3,4,5,6,7,8;
    int i,j,a=10;
end
begin
    set_mode(640,480,16);
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            write(0,100,10+a ,4,tabla5[i][j]);
            a=a+10;
        end
    end
end

loop
    frame;
end
end

```

El ejemplo de escritura es simplemente añadir una línea al anterior:

```

program ejemplo;
private
    int tabla5[2][3]=1,2,3,4,5,6,7,8;

```

```

        int i,j,a=10;
end
begin
  set_mode(640,480,16);
  for(i=0;i<3;i++)
    for(j=0;j<4;j++)
      tabla5[i][j]=444;
      write(0,100,10+a ,4,tabla5[i][j]);
      a=a+10;
    end
  end
end
loop
  frame;
end
end

```

Y los dos bucles ¿por qué? Fíjate. En la primera iteración del bucle más exterior,  $i=0$ . En la primera iteración del bucle más interior,  $j=0$ . A partir de aquí, se van a ir sucediendo iteraciones en el bucle interior, de manera que mientras  $i$  vale 0,  $j$  valdrá consecutivamente 1, 2 y 3. Es decir, se habrán realizado cuatro iteraciones con los valores  $i=0$  y  $j=0$ ,  $i=0$  y  $j=1$ ,  $i=0$  y  $j=2$  y  $i=0$  y  $j=3$ . En este momento, se ha llegado al límite de iteraciones en el bucle interior, se sale de él y se va a la siguiente iteración del bucle exterior, con lo que  $i=1$ , y se vuelve a repetir el proceso de las cuatro iteraciones internas con  $i=1$ , por lo que ahora tendremos los valores  $i=1$  y  $j=0$ ,  $i=1$  y  $j=1$ ,  $i=1$  y  $j=2$  y  $i=1$  y  $j=3$ . Y así hasta llegar a la última iteración del bucle exterior, que es cuando  $i=2$ . Si te fijas, con este método hemos logrado recorrer todos los elementos de la matriz.

Y con una matriz tridimensional ¿qué? Pues se usarán de forma similar tres bucles anidados.

Un ejemplo práctico de uso de tablas (para implementar por ejemplo un sistema de diálogos entre personajes de un juego):

```

Program texto_dialogos;
Private
  int contador=0;
// Todos los textos que aparecerán
string textosjuego[] =
  "Volver...",
  "con la frente marchita",
  "las nieves del tiempo",
  "platearon mi sién.",
  "Sentir...",
  "que es un soplo la vida,",
  "que veinte años no es nada,",
  "¡qué febril la mirada!",
  "Errante, la sombra",
  "te busca y te nombra.",
  "Vivir...",
  "con el alma aferrada",
  "a un dulce recuerdo",
  "que lloro otra vez.";
end

```

```

Begin
  set_mode(640,480,16);
  Loop
    delete_text(0);
    If(key(_down))
      contador++;
      //Hasta que no se suelte la tecla no pasa nada
      while(key(_down) <> 0) frame; end
    End
    //Muestro la cadena que ocupa la posición dada por el índice "contador"
    write_var(0,300,120,4,textosjuego[contador]);
    //Si llego al final de la tabla, vuelvo a empezar
    If(contador=>14) contador=0;End
    If(key(_esc)) exit();End
  Frame;
End
End

```

### **Estructuras y tablas de estructuras:**

Una estructura es como cada una de las fichas típicas de oficina que contienen información sobre una persona diferente (cada una de ellas) como el número de identidad, el nombre, dirección, teléfono, etc.

Normalmente no se utiliza una estructura sola, sino que se trabajan con tablas de estructuras. Las tablas de estructuras serían entonces como el cajón que alberga todas las fichas/estructuras. A las estructuras también se les llama a menudo registros. Y cada anotación dentro de una ficha (el nombre, dirección, teléfono...) se denomina campo.

Aunque no es formalmente correcto, es muy común denominar, para abreviar, estructura a lo que realmente es una tabla de estructuras.

Igual que las variables, las estructuras pueden ser globales, locales, privadas o públicas.

Por ejemplo, en un juego se podría definir la siguiente estructura global para guardar la información sobre la posición en pantalla de tres enemigos:

```

GLOBAL
  STRUCT posicion_enemigos[2]
    Int coordenada_x;
    Int coordenada_y;
  END = 10,50,0,0,90,80;
END

```

Este cajón de fichas se llamaría `posicion_enemigos`, contendría 3 fichas para 3 enemigos (con los números 0,1 y 2, como indica el 2 entre corchetes), y cada ficha dos anotaciones, la coordenada horizontal de un enemigo y la vertical.

En la jerga correcta, `posicion_enemigos` es una estructura de 3 registros, cada uno con 2 campos, que serían `coordenada_x` y `coordenada_y`. La lista de valores siguiente posicionaría el primer enemigo en la coordenadas (10,50), el segundo en (0,0) y el tercero en (90,80). Como verás, el 10 es el valor de `coordenada_x` para `posicion_enemigos[0]`, el 50 es el valor de `coordenada_y` para `posicion_enemigos[0]`, el 0 es el valor de `coordenada_x` para `posicion_enemigos[1]`, etc.

Fijarse en el ejemplo anterior que como a la vez de declarar la estructura, se inicializan con valores sus campos, al final de dichos valores se escribe un punto y coma. Si se declara la estructura y ya está (dejando inicialmente los campos vacíos), no se escribirá ningún punto y coma detrás del END, como es habitual.

Para acceder después en el programa a uno de estos valores numéricos se debería indicar el nombre de la estructura, el número de registro y el nombre del campo. Por ejemplo, para poner el valor 66 en la posición de memoria que guarda las coordenada vertical (y) del segundo enemigo (el registro número 1, ya que el primer enemigo tiene sus coordenadas en el registro 0), se utilizaría la siguiente sentencia:

```
posicion_enemigos[1].coordenada_y = 66;
```

Y para visualizarlo por pantalla, se podría utilizar por ejemplo:

```
write(0,100,100,4,posicion_enemigos[1].coordenada_y);
```

En realidad es muy parecido a las tablas, pero indicando después del número de registro el símbolo "." (un punto) y el nombre del campo.

La gran diferencia que tienen con las tablas es que los campos pueden no ser del mismo tipo.

De hecho, cada campo de la estructura puede ser una variable, una tabla u otra estructura completa, con sus diferentes registros o campos. Por ejemplo, sería completamente correcto crear una estructura global como ésta:

```
GLOBAL
    STRUCT datos_clientes [9]
        Byte codigo_interno;
        String DNI;
        String nombre;
        String apellidos;
        Int num_articulos_comprados;
        Float dinero_que_debe;
    END
END
```

En este caso, habríamos creado 10 fichas para los datos de nuestros clientes, en las cuales incluiríamos, para cada uno de ellos, datos de diferente tipo.

Fíjate aquí que no es necesario inicializar la estructura cuando se declara: se pueden rellenar los valores de los campos más adelante. En este caso, en el momento de la declaración de la estructura todos los valores de sus campos se inicializan a 0 por defecto.

Si quisiéramos darle valores a un registro concreto (por ejemplo el 2), ya sabemos que no tendríamos más que hacer:

```
datos_clientes[2].codigo_interno=34;
datos_clientes[2].DNI="89413257F";
datos_clientes[2].nombre="Federico";
datos_clientes[2].apellidos="Martínez Fernández";
datos_clientes[2].num_articulos_comprados=2;
datos_clientes[2].dinero_que_debe=789.04;
```

Y repito, para consultar el valor de un campo concreto de ese registro, un write bastaría:

```
write(0,100,100,4, datos_clientes[2].nombre);
```

Veamos un ejemplo de código que utilice una tabla de estructuras para almacenar las posiciones de un determinado número de enemigos al principio de la partida y al final también. Asignaremos valores a los campos de cada registro y luego los comprobaremos mostrándolos por pantalla.

La tabla de estructuras del ejemplo contendrá la posición inicial y final de hasta un máximo de 10 procesos enemigos (sería como una caja con 10 fichas, cada una de ellas indicando la (x,y) inicial y la (x,y) final de un proceso):

```
STRUCT movimiento_enemigos[9]
    Int x_inicial;
    Int y_final;
    Int x_inicial;
    Int y_final;
END
```

Vemos que esta tabla tiene diez registros y en cada uno cuatro campos. El ejemplo primero se encargará, mediante bucles que recorrerán las 10 estructuras, de asignar valores a todos los cuatro campos de cada registros. Y posteriormente los visualizará, utilizando para recorrer las 10 estructuras otra vez bucles.

```
program ejemplo;
private
STRUCT movimiento_enemigos[9]
    Int x_inicial;
    Int y_inicial;
    Int x_final;
    Int y_final;
END
    int i,a;
end
begin
    set_mode(640,480,16);
    for(i=0;i<10;i++)
        movimiento_enemigos[i].x_inicial=i;
        movimiento_enemigos[i].y_inicial=i;
        movimiento_enemigos[i].x_final=i+1;
        movimiento_enemigos[i].y_final=i+1;
    end

    for(i=0;i<10;i++)
        write(0,10+a,10,4,"Enemigo " + i);
        write(0,10+a,20,4,movimiento_enemigos[i].x_inicial);
        write(0,10+a,30,4,movimiento_enemigos[i].y_inicial);
        write(0,10+a,40,4,movimiento_enemigos[i].x_final);
        write(0,10+a,50,4,movimiento_enemigos[i].y_final);
        a=a+67;
    end
end
loop
```

```
end  
frame;  
end
```

Otro ejemplo más complicado. Si se hace la siguiente declaración:

```
STRUCT a[2]  
    Int b;  
    Int c[1];  
END = 1,2,3,4,5,6,7,8,9;
```

Los valores inicializados, ¿a qué campo van a parar?

Sabemos que la estructura a[] tiene 3 registros (desde a[0] hasta a[2]) y en cada registro 3 campos (b, c[0] y c[1]), luego la anterior declaración inicializaría la estructura de la siguiente forma:

```
a[0].b = 1;  
a[0].c[0] = 2;  
a[0].c[1] = 3;  
a[1].b = 4;  
a[1].c[0] = 5;  
a[1].c[1] = 6;  
a[2].b = 7;  
a[2].c[0] = 8;  
a[2].c[1] = 9;
```

Se puede comprobar haciendo el bucle pertinente que recorra los elementos y los muestre por pantalla, como ya sabemos.

### **La función SIZEOF():**

Esta función devuelve el número de bytes que ocupa en memoria el tipo de dato, tabla o estructura (o tabla de estructuras) que se le pase por parámetro.

Explico esta función aquí porque es muy útil y práctica en muchos casos donde se necesita saber el tamaño total que ocupa determinada tabla o estructura (que es bastante frecuente).

Por ejemplo, sabemos que un dato Int ocupa 4 bytes en memoria, pues la sentencia

```
variable= sizeof(int);
```

debería de asignar el valor de 4 a "variable". Y así con todos los tipos de datos, incluyendo además tablas y estructuras. Si tenemos una tabla de 20 elementos Int, ¿cuánto ocupará en memoria? Pues  $20 \times 4 = 80$  bytes, por lo que la sentencia

```
variable=sizeof(mitabla);
```

debería de asignar el valor de 80 a "variable".

Hay que hacer notar que sizeof devuelve el tamaño TOTAL de la tabla en memoria, el cual se calcula a partir de sumar el tamaño del total de elementos, ya sean elementos que hayamos llenado nosotros explícitamente como también aquellos que aparentemente estén vacíos, pero que en realidad no lo están porque ya sabes que si no se les asigna un valor explícitamente, los elementos de una tabla siempre son

asignados al valor 0, por lo que siempre tendremos las tablas llenas completamente y su tamaño en memoria siempre será el mismo tenga más o menos elementos rellenos "a mano".

Igualmente pasa con estructuras. Si tenemos una estructura llamada "miestructura", simplemente deberíamos de escribir `sizeof(miestructura)` para saber cuánto ocupa en memoria esa estructura.

Ojo, porque si estamos hablando de una tabla (ya sea de elementos simples como de estructuras con varios registros) llamada por ejemplo "mitabla", `sizeof(mitabla)` devolverá el tamaño completo que ocupan todos los registros que forman parte de la tabla, y en cambio `sizeof(mitabla[0])` devolverá el tamaño que ocupa sólo el primer registro -que es el mismo que cualquiera de los otros-. A partir de aquí es fácil ver que una operación -bastante común- como esta división: `sizeof(mitabla)/sizeof(mitabla[0])` lo que devolverá será el número de elementos que puede albergar dicha tabla.

Veamos un ejemplo clarificador:

```
program misizeof;
private
  byte a;
  short b;
  word c;
  int d;
  dword e;
  float f;
  string g;
  char h;
  int tabla1[3];
  byte tabla2[1][1];
  struct struct1
    int a;
    string b;
  end
  struct struct2[9]
    int a;
    float b;
    string c;
  end
end
begin
  set_mode(640,480,16);
  write(0,300,10,4,"Byte: " + sizeof(byte));
  write(0,300,20,4,"Short:" + sizeof(short));
  write(0,300,30,4,"Word: " + sizeof(word));
  write(0,300,40,4,"Int: " + sizeof(int));
  write(0,300,50,4,"Dword: " + sizeof(dword));
  write(0,300,60,4,"Float: " + sizeof(float));
  write(0,300,70,4,"String: " + sizeof(string));
  write(0,300,80,4,"Char: " + sizeof(char));
  write(0,300,90,4,"Vector de 4 enteros: " + sizeof(tabla1));
  write(0,300,100,4,"Matriz de bytes 2x2: " + sizeof(tabla2));
  write(0,300,110,4,"Estructura con un campo Int y otro String: " + sizeof(struct1));
  write(0,300,120,4,"Tabla de 10 estructuras con un campo Int, otro Float y otro String: " + sizeof(struct2));
  write(0,300,130,4,"Un registro de la tabla de 10 estructuras anterior: " + sizeof(struct2[0]));
loop
  frame;
```

```
end
end
```

Y otro ejemplo interesante:

```
//Author: SplinterGU
program hola;
private
  int Array1[9]; //Tiene 10 elementos reservados en memoria, todos llenos de 0
  float Array2[] = 1.0,2.0,3.0; /*Tiene 3 elementos reservados en memoria, ya que al no especificar un
  índice entre los corchetes, se reservarán tantos elementos como valores se especifiquen*/
  int SizeArray;
end
begin
  SET_MODE(320,240,16);
  SizeArray = sizeof(Array1)/sizeof(Array1[0]);
  WRITE(0,150,50,4,"El tamaño de Array1 es: " + SizeArray); //Ha de mostrar un 10
  SizeArray = sizeof(Array2)/sizeof(Array2[0]);
  WRITE(0,150,150,4,"El tamaño de Array2 es: " + SizeArray); //Ha de mostrar un 3
  loop FRAME; end
end
```

La función SIZEOF también se puede usar para averiguar el espacio que ocupan los datos definidos por el usuario con TYPE (visto a continuación).

### Los tipos definidos por el usuario. TYPE:

Un paso más allá en el concepto de estructuras es el de tipo definido por el usuario (TDU).

Un TDU es una estructura/ficha/registro (ojo, no una tabla de estructuras), con una diferencia: no es ninguna variable y por tanto no se puede usar directamente, sino que se utiliza para definir a su vez a otras variables. Es decir, el tipo de estas otras variables no será ni Int ni float ni ningún otro tipo predefinido, sino que será precisamente el TDU, el cual es un tipo de datos formado a partir de otros. Y estas variables podrán ser incluso tablas de elementos de ese TDU.

Para crear un TDU se utiliza el bloque TYPE/END. Su sintaxis es exactamente igual a la de un bloque STRUCT/END sin inicialización de valores, con la salvedad que se ha de escribir obligatoriamente fuera de cualquier sección de declaraciones de variables. Es decir, fuera de los bloques GLOBAL/END, LOCAL/END o PRIVATE/END. Y también fuera de cualquier BEGIN/END. Normalmente se suelen escribir justo después de la línea program.

Es decir, resumiendo, el bloque TYPE/END declara un tipo de dato definido por el usuario y compuesto a su vez por varios datos de otro tipo, y no crea ninguna variable concreta en el momento en el que aparece: simplemente, le otorga un significado al identificador utilizado y permite que sea usado posteriormente en cualquier declaración de otras variables.

Un uso típico de las tablas de TDU es por ejemplo, para almacenar en cada registro determinada información sobre un enemigo (gráfico, posición x e y, vida, nivel...), o una pieza de un rompecabezas (gráfico, posición x e y, ancho, alto...), etc. Ya verás que sus aplicaciones son múltiples, y se te irán ocurriendo más cuando los necesites.

Veamos un ejemplo. En el siguiente código se crea un TDU compuesto de un campo entero, otro decimal y otro de cadena. También se crea otro TDU compuesto de una tabla de enteros y de una tabla de caracteres. A partir de aquí, en la sección PRIVATE/END se declaran dos variables del primer TDU y una del segundo. El programa lo único que hace es asignar valores a los campos de las tres variables declaradas y seguidamente mostrarlos por pantalla.

```
program mistipos;
type tipo1
    int a;
    float b;
    string c;
end
type tipo2
    int a[20];
    char b[15];
end
private
    tipo1 hola,adios;
    tipo2 quetal;
end
begin
set_mode(640,480,16);

hola.a=2;
hola.b=4.5;
hola.c="Lalala";

adios.a=87;
adios.b=96.237;
adios.c="Lerele";

quetal.a[0]=3;
quetal.a[1]=64;
quetal.a[2]=9852;
quetal.b[0]="c";
quetal.b[1]="g";

write(0,100,10,4,hola.a);
write(0,100,20,4,hola.b);
write(0,100,30,4,hola.c);
write(0,100,40,4,adios.a);
write(0,100,50,4,adios.b);
write(0,100,60,4,adios.c);
write(0,100,70,4,quetal.a[0]);
write(0,100,80,4,quetal.a[1]);
write(0,100,90,4,quetal.a[2]);
write(0,100,100,4,quetal.b[0]);
write(0,100,110,4,quetal.b[1]);

loop
    frame;
end
end
```

Como se puede ver, la mayor utilidad de los TDU es poder crear un tipo estructura con un determinado nombre identificador para declarar posteriormente estructuras a partir simplemente de ese identificador, y cuyo contenido será el escrito en el interior del bloque TYPE. Se puede deducir de esto que TYPE es mucho más cómodo que STRUCT si se tienen que crear múltiples variables en lugares diferentes del programa con la misma estructura.

En definitiva, si, por ejemplo, al principio del programa aparece este código:

```
TYPE color
  BYTE r, g, b;
END
```

sería posible declarar una o más variables del nuevo tipo "color" en cualquier sección de datos, por ejemplo GLOBAL:

```
GLOBAL
  color p;
END
```

siendo en este ejemplo, la variable "p" es una estructura con tres componentes de tipo byte (r, g, b).

### **Copia y comparación entre dos tablas o dos estructuras. Gestión básica de memoria:**

En ciertas ocasiones nos puede interesar copiar el contenido completo de una tabla en otra, o de una estructura a otra. Lo más inmediato que se nos ocurriría sería probar esto:

```
program laringe;
private
struct struct1
  Int c;
  Int d;
end =1,4;
struct struct2
  Int c;
  Int d;
end
int tabla1[1]=6,2;
int tabla2[1];

begin
  set_mode(640,480,16);

  //Las estructuras han de ser idénticas en lo que se refiere a número,tipo y orden de campos
  struct2=struct1; //Copio los valores de los campos de struct1 a struc2
  /*Los elementos de las dos tablas han de ser del mismo tipo, y el tamaño de tabla2 ha de ser igual o mayor
  al de tabla1.*/
  tabla2=tabla1; //Copio los valores de los elementos de tabla1 a tabla2

  //Muestro que efectivamente se han copiado bien.
  write(0,100,10,4,struct2.c); //Debería de salir un 1
  write(0,100,20,4,struct2.d); //Debería de salir un 4
```

```
write(0,100,30,4,tabla2[0]); //Debería de salir un 6
write(0,100,40,4,tabla2[1]); //Debería de salir un 2
```

```
loop
    frame;
end
end
```

Bien. Si intentamos ejecutar esto el intérprete nos devolverá un error en la línea donde se supone que copio el contenido de struct1 a struct2. Parece pues que no se hace así la copia. Si comentamos esa línea para probar al menos la copia de las tablas, veremos otra cosa curiosa: el programa se ejecuta sin errores pero resulta que sólo se ha copiado el valor del primer elemento: los valores del resto de elementos no. O sea que tampoco funciona copiar tablas de esta manera.

Copiar tablas o estructuras no es tan fácil como copiar valores entre variables simples. Para lograr hacerlo correctamente, necesitamos hacer uso de la función MEMCOPY

### MEMCOPY(&DESTINO,&ORIGEN,BYTES)

Esta función copia un bloque de memoria de tamaño BYTES desde la tabla/estructura especificada por su nombre ORIGEN (precedido del símbolo &), a la tabla/estructura especificada por su nombre DESTINO (precedido del símbolo &).

*Te estarás preguntando por qué hay que poner el & delante de los dos primeros parámetros de este comando (y de muchos otros que iremos viendo). Este símbolo indica que no se trabaja con valores sino con sus direcciones de memoria.*

*La memoria del ordenador está organizada en forma de casillas, donde cada una tiene una dirección, gracias a la cual el ordenador sabe dónde tiene guardados los datos: la memoria RAM del ordenador es como un inmenso apartado de correos. Cada vez que veas un símbolo & delante del nombre de alguna variable, estaremos utilizando no el valor que contiene esa variable, sino la dirección de memoria donde está alojado esa variable (y por tanto, su valor). De hecho, por eso el comando MEMCOPY se llama así: porque lo que hace en realidad es copiar un bloque entero de memoria de un tamaño dado por BYTES, desde una dirección de memoria dada por donde empieza a guardarse ORIGEN, a una dirección de memoria que es donde empieza a guardarse DESTINO, sobrescribiéndolo en este caso. El tipo de datos POINTER ("puntero") es el requerido para este tipo de parámetros*

*En la práctica lo único que hay que acordarse cuando se quiera utilizar esta función es añadir un símbolo & delante de los nombres -ojo, nada de valores concretos- de las variables estándar que serán el primer y segundo parámetro de la función*

Ambas zonas de memoria ORIGEN y DESTINO pueden solaparse, en cuyo caso la zona de destino contendrá tras la ejecución los valores que antes estaban en la de origen, machacando cualquier posición compartida.

Es recomendable NO usar esta función con bloques de memoria (tablas o estructuras) que incluyan elementos de tipo cadena, pues el intérprete de Fénix puede quedar en estado inconsistente.

#### PARAMETROS:

POINTER DESTINO : Puntero a la zona de destino

POINTER ORIGEN: Puntero a la zona de origen

INT BYTES : Número de bytes a copiar

Es decir, que para copiar el contenido de una tabla a otra (o de una estructura a otra), lo que se tendrá que hacer es copiar el bloque de memoria donde está almacenada la tabla/estructura de origen a otro

sitio de la memoria del ordenador, el cual será el bloque de destino. Parece muy extraño, pero en verdad es muy sencillo si volvemos a intentar el ejemplo anterior haciendo uso de esta nueva función:

```
program laringe;
private
struct struct1
    Int c;
    Int d;
end =1,4;
struct struct2
    Int c;
    Int d;
end
int tabla1[1]=6,2;
int tabla2[1];

end
begin
    set_mode(640,480,16);
    //Las estructuras han de ser idénticas en lo que se refiere a número,tipo y orden de campos
    memcpy(&struct2,&struct1,sizeof(struct1));//Copio los valores de los campos de struct1 a struc2

    /*Los elementos de las dos tablas han de ser del mismo tipo, y el tamaño de tabla2 ha de ser igual o
    mayor al de tabla1.*/
    memcpy(&tabla2,&tabla1,sizeof(tabla1));//Copio los valores de los elementos de tabla1 a tabla2

    //Muestro que efectivamente se han copiado bien.
    write(0,100,10,4,struct2.c); //Debería de salir un 1
    write(0,100,20,4,struct2.d); //Debería de salir un 4
    write(0,100,30,4,tabla2[0]); //Debería de salir un 6
    write(0,100,40,4,tabla2[1]); //Debería de salir un 2

loop
    frame;
end
end
```

Podemos comprobar que ahora sí se realiza la copia correctamente.

No es necesario utilizar la función MEMCOPY si solamente se desea copiar un campo concreto de una estructura a otra, o un elemento concreto de una tabla a otra. En estos casos, una simple asignación del tipo:

```
tabla2[indiceelementoconcreto]=tabla1[indiceelementoconcreto];
```

ó

```
struct2.uncampoconcreto=struct1.uncampoconcreto;
```

funcionará perfectamente. El uso de MEMCOPY está diseñado para copiar tablas o estructuras enteras (y también variables de TDU).

Nos encontraremos con un problema similar al acabado de comentar en los párrafos precedentes

si lo que deseamos es comparar el contenido completo de una tabla con otra, o de una estructura con otra. Lo más inmediato que se nos podría ocurrir podría ser esto:

```
program laringe;
private
struct struct1
    Int c;
    Int d;
end =1,4;
struct struct2
    Int c;
    Int d;
end = 3,8;
int tabla1[1]=6,2;
int tabla2[1]=6,5;

end
begin
    set_mode(640,480,16);

    if (struct1 == struct2)
        write(0,100,100,4,"Todos los valores de las estructuras son iguales");
    else
        write(0,100,150,4,"Algún valor de las estructuras es diferente");
    end

    if (tabla1 == tabla2)
        write(0,100,200,4,"Todos los valores de las tablas son iguales");
    else
        write(0,100,250,4,"Algún valor de las tablas es diferente");
    end
loop
    frame;
end
end
```

Nos vuelve a pasar lo mismo. El if que compara las estructuras devuelve un error que no nos permite ejecutar el programa, y la comparación de tablas sólo compara el primer elemento, obviando los siguientes. No es así pues como se comparan tablas, estructuras o TDU. Necesitamos para ello la función MEMCMP.

### MEMCMP(&BLOQUE1,&BLOQUE2,BYTES)

Esta función compara el contenido de un bloque de memoria dado por el primer parámetro con el contenido de otro bloque de memoria dado por el segundo parámetro. Ambos bloques tendrán un tamaño idéntico en bytes, dado por el tercer parámetro.

Esta función devolverá 0 si los bloques son iguales, y -1 si no lo son

#### PARAMETROS:

POINTER BLOQUE1 : Puntero a comienzo de un bloque de memoria

POINTER BLOQUE2: Puntero al comienzo del otro bloque de memoria cuyo contenido se

comparará con el primero

INT BYTES : Tamaño en bytes de los bloques de memoria (es el mismo para los dos, lógicamente)

VALOR RETORNADO: INT : Resultado de la comparación (0 si son iguales, -1 si son diferentes)

Sabiendo esto, modificaremos nuestro ejemplo para que ahora funcione:

```
program laringe;
private
struct struct1
    Int c;
    Int d;
end =1,4;
struct struct2
    Int c;
    Int d;
end = 3,8;
int tabla1[1]=6,2;
int tabla2[1]=6,3;

end
begin
    set_mode(640,480,16);

    if (memcmp(&struct1,&struct2, sizeof(struct1))==0)
        write(0,200,100,4,"Todos los valores de las estructuras son iguales");
    else
        write(0,200,150,4,"Algún valor de las estructuras es diferente");
    end

    if (memcmp(&tabla1,&tabla2, sizeof(tabla1))==0)
        write(0,200,200,4,"Todos los valores de las tablas son iguales");
    else
        write(0,200,250,4,"Algún valor de las tablas es diferente");
    end
loop
    frame;
end
end
```

Otro ejemplo muy similar de la función MEMCMP , pero esta vez utilizando tipos definidos por el usuario podría ser éste:

```
program test;

type tipoper
    int var1;
    int var2;
end

global
    tipoper v1;
    tipoper v2;
    int i;
end
```

```

begin
v1.var1 = 1; v1.var2 = 2;
v2.var1 = 1; v2.var2 = 3;
i=memcmp(&v1,&v2,sizeof(tipoper));
write(0,10,10,0,i);

v2.var1 = 1; v2.var2 = 2;
i=memcmp(&v1,&v2,sizeof(tipoper));
write(0,10,20,0,i);

while(!key(_ESC))
    frame;
end
end

```

No es necesario utilizar la función MEMCMP si solamente se desea comparar el valor contenido en un campo concreto de una estructura y en otra, o en un elemento concreto de una tabla y en otra. En estos casos, una simple comparación del tipo:

```
if(tabla2[indiceelementoconcreto]==tabla1[indiceelementoconcreto])
```

ó

```
if(struct2.uncampoconcreto==struct1.uncampoconcreto)
```

funcionará perfectamente. El uso de MEMCMP está diseñado para comparar tablas o estructuras enteras (y también variables de TDU, como hemos visto).

### **Pasar un vector (tabla unidimensional) o un TDU como parámetro de un proceso/función:**

En páginas anteriores se ha explicado cómo pasar parámetros en un proceso o función. Pero estos parámetros eran siempre de tipos simple (enteros, decimales, cadenas). ¿Cómo se puede pasar un vector o un TDU? Es algo que nos interesará saber y no se hace de la misma manera.

El hecho que se haga de otra forma viene dado porque se utilizan direcciones de memoria y punteros, conceptos avanzados que no se tratarán en este curso. Es decir, que para entender completamente el proceso de paso por parámetro de vectores necesitaríamos tener más conocimientos sobre programación. No obstante, a pesar de no ser muy pedagógico, a continuación voy a resumir en forma de receta los pasos que hay que hacer siempre para lograr conseguir nuestro objetivo, sin profundizar más en ello. Simplemente hay que hacer dos cambios respecto como pasaríamos un parámetro simple:

1) En la cabecera del proceso/función, hemos de especificar que el parámetro es un vector escribiendo, en el lugar que le corresponde como parámetro que es:

```
tipo pointer nombrevector
```

donde "tipo" es el tipo de datos de los elementos del vector, "pointer" es una palabra que se ha de escribir tal cual y "nombrevector" es el nombre del vector que queramos ponerle para trabajar con él en el interior del código del proceso/función. A partir de ahí, en ese código utilizaremos el vector como normalmente.

Si estamos hablando de TDUs, se hace muy parecido. En la cabecera de la función hay que especificar que el parámetro de es un tipo personalizado escribiendo, en el lugar que le corresponde como parámetro que es: *tipoPropio pointer variableDeEseTipo* , donde "tipoPropio" es el nombre del TDU, "pointer" es una palabra que se ha de escribir tal cual y "variableDeEseTipo" es el nombre de la variable de ese tipo de datos que utilizaremos.

Fíjate que estamos hablando siempre de vectores o de TDUs, no de tablas multidimensionales. Para lograr pasar estos datos por parámetro, el proceso es diferente y todavía un poco más "rebuscado" (repito que se necesitan tener conocimientos de gestión de memoria) con lo que no se explicará en este manual.

2) En la llamada al proceso/función, se ha de preceder con el símbolo "&" el nombre del vector (y no poner corchetes ni nada) que se pasa por parámetro o bien el de una variable cuyo tipo sea el TDU , así:

```
proceso(&mivector);
```

o

```
proceso(&mivariabledeuntipopersonalizado);
```

Veamos un ejemplo. Vamos a crear un proceso que reciba un vector por parámetro y se dedique y mostrar por pantalla el valor de sus elementos:

```
program ejemplo;
private
  string mitabla[3]="aswert","asd2","asdfa","ppaw";
end
begin
  set_mode(640,480,16);
  miproceso(&mitabla);
end

process miproceso(string pointer lalala)
private
  int i;
end
begin
  for(i=0;i<4;i++)
    write(0,100,10+10*i ,4,lalala[i]);
  end
  loop
    FRAME;
  end
end
```

Un detalle importante: cuando pasamos vectores como parámetros, Fénix está diseñado de tal manera que cualquier cambio que hagamos dentro del código interno del proceso/función en cuestión a los valores de los elementos de ese vector, esos cambios permanecerán fuera de dicho proceso/función, con lo que si accedemos a esos elementos desde otro sitio, esos cambios se verán reflejados. Esto se puede ver fácilmente con este ejemplo:

```
private
```

```

    string mitabla[2]="Popo","Lela","Asdfa";
end
begin
    set_mode(640,480,16);
    leervector(&mitabla,100);
    cambiarvector(&mitabla);
    leervector(&mitabla,200);
    loop
        frame;
    end
end

function leervector(string pointer lalala, int posx)
private
    int i;
end
begin
    for(i=0;i<3;i++)
        write(0,posx,10+10*i,4,lalala[i]);
    end
    return;
end

function cambiarvector(string pointer lalala)
private
    int i;
end
begin
    for(i=0;i<3;i++)
        lalala[i]="CAMBIADO";
    end
    return;
end

```

Igual que hemos hecho con vectores, también podríamos utilizar tipos definidos por el usuario (TDU) como parámetros de funciones, con (otra vez) la particularidad de que cualquier modificación en los valores de sus campos efectuada dentro del cuerpo de la función se mantendrán una vez acabada la ejecución de ésta (en la continuación de la ejecución del programa). Un ejemplo:

```

program ratata;

type tipo
    int a;
    string b;
end

process main()
private
    tipo t;
end
begin
    t.a=1;
    t.b="a";

```

```
    hola(&t);  
    write(0,100,100,4,t.a);  
    write(0,100,110,4,t.b);  
    Loop frame; end  
end
```

```
function hola(tipo pointer pepito)  
begin  
    pepito.a = 2;  
    pepito.b = "b";  
end
```

Verás que lo que aparecerá por pantalla serán los nuevos valores de los campos de la variable "t" -del TDU "tipo"-, una vez se haya ejecutado la función "hola()".

Si piensas un poco, verás que lo que hemos hecho en estos ejemplos, sin decirlo, es crear una función que devuelve no ya enteros como hasta ahora -que también: la sentencia return se puede seguir usando- sino que es capaz de devolver otro tipo de valores, como vectores o TDUs, debido a que las modificaciones hechas internamente en la función se mantienen después. Esta manera de hacer se podría hacer extensible a otros tipos como cadenas o decimales, con lo que ya podríamos construir funciones que pudieran devolver cualquier tipo de datos. ¿Te animas?

## CAPÍTULO 6:TUTORIAL PARA EL JUEGO DEL LABERINTO

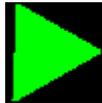
(extraído del tutorial de Wakroo, en <http://divnet.divsite.net>)

En este capítulo realizaremos paso a paso nuestro primer juego, el juego de conseguir salir de un laberinto mediante los cursores del teclado. No es nada espectacular, pero por algo se empieza, ¿no crees?

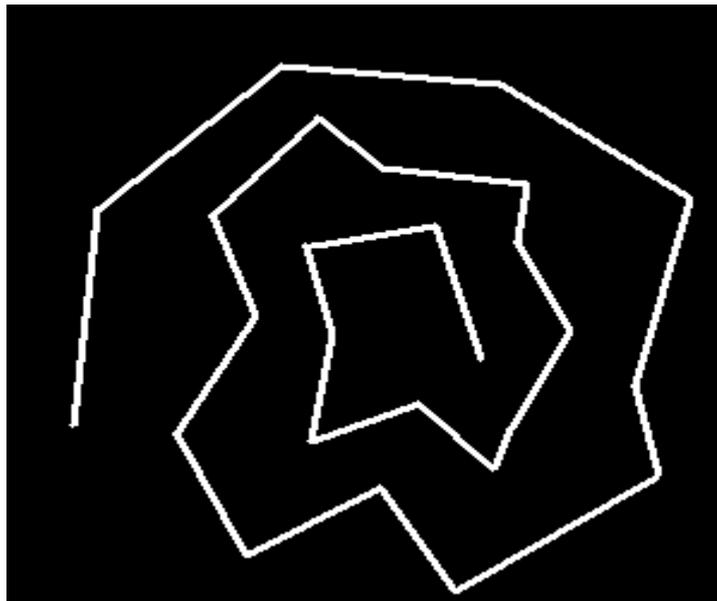
### Imágenes y transparencias:

Lo primero que vamos a hacer es crearnos un fichero FPG –llamado “imágenes.fpg” con dos imágenes PNG. Una será de 30x30 píxeles y tendrá el código 001: representará el personaje que moveremos nosotros con el cursor. La otra representará el laberinto; será de 640x480 –la pantalla completa- y tendrá el código 002.

La imagen 001 puede ser algo parecido a esto:



y la imagen 002 –a escala- a esto:



Un detalle importantísimo antes de continuar: fíjate que ambas imágenes tienen el negro absoluto como color de fondo. Esto no es porque sí. En Fénix, para las imágenes de profundidad de 16 bits –que son las que usaremos siempre- el color negro absoluto en un trozo de la imagen indica que ese trozo es como si no existiera: es “transparente”, no se pinta. Esto quiere decir que no podremos dibujar nunca nada de color absoluto porque no se verá, pero no te preocupes, eso no es mucho problema porque luego veremos que podemos pintar lo que sea de un color casi casi casi negro, donde nuestro ojo no notará la diferencia pero Fénix sí y lo tratará como un color normal.

De todas formas, y a pesar de lo dicho, cuando ejecutes el juego verás que el color de fondo es negro: esto es porque no se ha definido ninguna imagen de fondo con `put_screen`. Si la definiéramos, lo que se vería –si hemos puesto el negro absoluto como fondo del personaje y el laberinto- es la imagen de fondo

perfectamente y encima sólo visibles las líneas del laberinto y el triángulo del personaje, pero no sus rectángulos completos respectivos (que sí que se verían, y taparían entonces el fondo de la pantalla si hubiéramos puesto como fondo del personaje y del laberinto otro color que no fuera el negro).

Así pues, para Fenix la zona negra absoluta no forma parte de ninguna imagen. Éste es el mecanismo que tenemos para poder dibujar imágenes que no sean rectangulares. O sea, que si hemos creado un personaje cuadrado de 30x30 como es el caso, pero hemos dibujado un triángulo en su interior y el resto lo hemos puesto negro, para Fénix el dibujo del personaje sólo será el triángulo rojo, con lo que eso comporta (contorno de la imagen, transparencias respecto otros dibujos de debajo, etc). Y lo que es más importante para nosotros, el laberinto también tiene el color negro de fondo. Piensa un poco: si hubiéramos dibujado las líneas que forman el laberinto en un fondo blanco por ejemplo, Fenix interpretaría que la imagen que corresponde al proceso laberinto es todo el rectángulo de la imagen. En cambio, si ponemos el color de fondo negro, el gráfico correspondiente al proceso laberinto sólo será las líneas de su contorno. Esto es vital para nuestro programa, porque si lo que queremos es comprobar cuándo nuestro personaje choca contra el laberinto, si el laberinto Fenix lo toma como todo el rectángulo de la imagen, el personaje siempre estaría chocando con él a pesar de no tocar ninguna línea –el personaje siempre está dentro de la imagen rectangular del laberinto–, pero si en cambio hacemos que Fénix tome como el gráfico del laberinto sólo las líneas, haciendo transparente todo lo demás, entonces podremos detectar las colisiones del personaje con las líneas solamente cuando éstas se produzcan, ya que el resto de la imagen del laberinto no se toma como tal: no es imagen.

### **El juego básico. El comando "Collision()":**

Bueno, empecemos. Vamos a aprovechar el primer ejemplo de código que escribimos en el capítulo anterior, el “ejemplo\_procesos” para empezar a escribir nuestro juego. Aquí os lo vuelvo a pasar para que os acordéis.

```
PROGRAM juego_labirinto;  
GLOBAL  
  INT id1;  
END  
BEGIN  
  set_mode(640,480,16,MODE_FULLSCREEN);  
  id1=load_fpg("imagenes.fpg");  
  personaje();  
END
```

```
PROCESS personaje()  
BEGIN  
  x=320; y=240; file=id1; graph=1;  
  LOOP  
    IF (key(_up)) y=y-10; END  
    IF (key(_down)) y=y+10; END  
    IF (key(_left)) x=x-10; END  
    IF (key(_right)) x=x+10; END  
    FRAME;  
  END  
END
```

Con este código, recordarás que lo que teníamos es nuestro personaje que se puede mover por cualquier sitio de nuestra pantalla, incluso salir de ella. Ahora vamos a añadir el laberinto, el cual, evidentemente, será otro proceso.

Recuerda que para hacer esto, hay que hacer dos cosas. Primero modifica el programa principal así:

```

PROGRAM juego_laberinto;
GLOBAL
  INT id1;
END
BEGIN
  set_mode(640,480,16,MODE_FULLSCREEN);
  id1=load_fpg("imagenes.fpg");
  personaje();
  laberinto();
END

```

Fijate que lo único que hemos hecho ha sido añadir después de la creación del personaje, la llamada al proceso “laberinto()” para que se cree y visualice el laberinto. Está claro que lo segundo que tendremos que hacer es codificar este nuevo proceso “laberinto()”. Para eso, después del END que cierra el código del proceso “personaje()”, escribe lo siguiente:

```

PROCESS laberinto()
BEGIN
  x=320;y=240;graph=2;
  LOOP
    frame;
  END
END

```

Con esto lo único que hacemos es visualizar el laberinto en el centro de la pantalla: (establecemos valores a las variables X,Y y GRAPH y nos metemos en un bucle infinito para ir mostrando siempre la imagen), pero nuestro personaje, -el cual ha de aparecer en el centro del laberinto según las X e Y que hemos puesto-, se puede seguir moviendo igual.

Lo que vamos a hacer ahora es obligar a que si el personaje se mueve y choca contra el laberinto, vuelva al centro de la pantalla. Para ello, hemos de modificar el proceso “personaje()” de la siguiente manera:

```

PROCESS personaje()
BEGIN
  x=320; y=240; file=id1; graph=1;
  LOOP
    IF (key(_up)) y=y-10; END
    IF (key(_down)) y=y+10; END
    IF (key(_left)) x=x-10; END
    IF (key(_right)) x=x+10; END
    IF (collision(TYPE laberinto)) x=320;y=240; END
    FRAME;
  END
END

```

Fijate que sólo hemos añadido una línea más a las que había: la línea *IF (collision(TYPE laberinto)) x=320;y=240; END*.

Nos encontramos con una nueva orden: collision(TYPE laberinto);. Esta sentencia devuelve cierto si el gráfico del proceso que la ejecuta -o sea, "personaje"- y el gráfico del proceso cuyo nombre se escribe

entre paréntesis seguido de TYPE -o sea, "laberinto"- se superponen –colisionan- de alguna manera. Por lo tanto, el IF es cierto y se ejecutan las sentencias del interior. Y ahí nos encontramos dos órdenes, que vuelven a poner el proceso personaje en el centro de la pantalla. ¡Ya hemos hecho nuestro primer juego!

**ACLARACIÓN IMPORTANTE:** Fíjate que dentro de los paréntesis de la orden *collision*, antes de poner el nombre del proceso con el que se quiere comprobar la colisión, hemos escrito la palabra reservada TYPE. ¿Por qué? Porque en realidad, poniendo esta palabra, lo que estamos intentando detectar son todas las colisiones que tenga el proceso "personaje" con TODAS las instancias del proceso "laberinto" que estén activas en este momento. Es decir, que si hubiéramos hecho varias llamadas dentro de nuestro código al proceso "laberinto", tendríamos por cada una de ellas una instancia de él (se verían múltiples laberintos en nuestro juego), y con la palabra TYPE lo que hacemos es detectar las colisiones con cualquiera de ellas.

Evidentemente, en este juego no sería necesario utilizar TYPE porque sólo tenemos una instancia de "laberinto" funcionando (o sea, sólo creado el proceso "laberinto" una vez), pero funciona igual. De todas maneras, alguna vez nos encontraremos con la necesidad de detectar la colisión de nuestro personaje con una instancia CONCRETA de un proceso (como por ejemplo, un enemigo concreto de entre tantos creados a partir de un único proceso "enemigo"). ¿Cómo lo haríamos? En ese caso, el parámetro de *collision* debería ser el ID de esa instancia concreta con la que se desea detectar si hay colisión o no.

Es decir, resumiendo, podemos utilizar la orden *collision* de dos formas. O bien:

```
collision(TYPE nombreProceso);
```

donde detectamos las colisiones del proceso actual con TODAS las instancias de un proceso concreto dado por nombreProceso, o bien:

```
collision(idProceso);
```

donde detectamos las colisiones del proceso actual sólo con aquella instancia que tenga un identificador dado por idProceso.

**OTRA ACLARACIÓN IMPORTANTE:** En el caso de utilizar *collision* de la manera *collision(TYPE nombreProceso);* en realidad, *collision* no devuelve "verdadero" o "falso" simplemente. Lo que devuelve en realidad es el ID del proceso con el que colisiona. Como un ID válido (es decir, correspondiente a un proceso activo existente) es siempre mayor que 0, y Fénix entiende que cualquier valor diferente de 0 es tomado como "verdadero", nos aprovechamos de este hecho para utilizar *collision* de esta manera: si devuelve 0 es que no ha habido colisión, y si devuelve otra cosa, sí que ha habido, y en realidad esa "otra cosa" es el ID del proceso con el que se ha colisionado. Esto es importante porque aunque ahora no lo necesitemos, más adelante veremos que nos interesará recoger este ID para saber con qué proceso hemos colisionado en concreto.

Un detalle: *collision* se comprueba en cada frame del proceso actual, pero puede ocurrir que antes de llegar a él, varios procesos hayan colisionado con el proceso actual. En ese caso, *collision* sólo devolverá el ID del último proceso con el que ha colisionado antes del fotograma, perdiéndose los otros ID anteriores. Esto se podría evitar, por ejemplo, usando un bucle y un vector, del tipo:

```
i=0;
```

```
while (vectorids[i]=collision (type otroProc))
```

```
... /*Vectorids[i] guarda el identificador de la instancia concreta de otroProc con la que se ha colisionado, las cuales pueden ser varias a la vez. Cuando se haya acabado de registrar todas esas
```

colisiones, se habrán rellenado unos determinados elementos de dicho vector y el bucle se termina porque la condición pasa a ser 0 -falsa-, (que es lo que devuelve collision cuando no se detecta colisiones). Justo después de este bucle se puede proseguir con el código normal del juego pudiendo utilizar todos los elementos de Vectoids.\*/  
i++;  
end

En el caso de utilizar *collision* en el modo *collision(idProceso)*;, cuando existe colisión no se devuelve *idProceso*, sino simplemente 1.

Fijaros finalmente que en este ejemplo, si quisiéramos hacer que se moviera más despacio o más deprisa tendríamos que alterar las cuatro cifras. Si en vez de 10 hubiésemos creado en la sección de declaraciones del programa principal una constante llamada "velocidad", por ejemplo, bastaría con alterar el valor de la constante al principio del programa para alterar todo el movimiento, lo que en un programa extenso nos ahorraría muchísimo tiempo y nos aseguraría que no nos hemos dejado nada por cambiar.

Tampoco sería mala idea crear dos constantes más, llamadas por ejemplo RESHOR y RESVER (de "resolución horizontal" y "resolución vertical", respectivamente, y asignarles el valor de 640 y 480. De esta manera, cuando usemos la orden *set\_mode*, podríamos hacer *set\_mode(RESHOR,RESVER,16)*; y cuando queramos situar al gráfico del laberinto y al personaje en el centro de la pantalla simplemente tendríamos que asignar a sus variables X e Y los valores RESHOR/2 y RESVER/2 respectivamente. Con esto ganamos que, si algún día decidiéramos cambiar la resolución de la pantalla del juego por otra (por ejemplo 800x600), no tuviéramos que cambiar "a mano" todos los valores numéricos de las coordenadas de los diferentes procesos (es decir, el 320 que vale la X por el 400 y el 240 de la Y por el 300): simplemente tendríamos que cambiar los valores de las dos constantes y automáticamente todo quedaría ajustado sin hacer nada más

### **Añadiendo giros al movimiento del personaje.El comando "Advance()" y la variable local predefinida ANGLE:**

Ahora veamos el siguiente caso: avanzar y girar. **Cuando creamos un gráfico en Fenix siempre estará "mirando" hacia la derecha.** Por lo tanto al dibujar objetos vistos desde arriba la parte de delante debe estar mirando hacia la derecha. Así nos evitaremos muchos problemas (fijaros que yo el triángulo lo he dibujado hacia la derecha). Para girar nuestro triángulo y hacer que "mire" hacia otro sitio habría que sustituir los cuatro IF de movimiento por estos otros:

```
IF (key(_up)) advance(10); END  
IF (key(_down)) advance(-10); END  
IF (key(_left)) angle=angle+7500; END  
IF (key(_right)) angle=angle-7500; END
```

Como veis es muy simple. La sentencia *advance()*; hace avanzar el gráfico el número de píxeles indicado entre paréntesis en el sentido en el que está "mirando" el gráfico. Si el número es negativo lo hace retroceder

**ANGLE** es una variable local predefinida, y representa el ángulo que existe entre una flecha imaginaria apuntando hacia la derecha hasta la orientación actual donde "mira" el proceso correspondiente.

Esta variable se mide en milésimas de grado y va en sentido contrario a las agujas del reloj. Por lo tanto si

ponemos angle=90000; el proceso estará mirando hacia arriba, ANGLE=180000; mirará hacia la izquierda, ANGLE=270000; mirará hacia abajo y ANGLE=360000; o ANGLE=0; hacia la derecha.

Tened cuidado con los ángulos, puesto que Fenix, al pasar de 360000 no ajusta el ángulo a 0, sino que sigue hacia adelante. Lo mismo pasa al bajar de 0. Se podría dibujar el gráfico en otro sentido y luego ajustarlo, pero lo más sencillo es ponerlo mirando hacia la derecha y evitar complicaciones.

Así pues, si apretamos el cursor de arriba o abajo veremos que lo que hace el triángulo es desplazarse 10 píxeles en la dirección en la que está "mirando" en ese momento, y si apretamos derecha o izquierda el triángulo no se mueve de donde está pero sí que varía su orientación (gira) para un lado o para el otro.

### **Inclusión de múltiples enemigos diferentes. La variable local predefinida SIZE:**

¿Y si ahora quisieramos incluir enemigos que dificulten nuestro juego? La idea es que, además de no poder chocar contra el laberinto, nuestro personaje evite chocar contra enemigos que irán surgiendo al azar por la pantalla.

Para que apareciera un enemigo sería suficiente con crear un proceso llamada, por ejemplo, "enemigo();" y hacer una llamada a ese proceso. ¿Pero y si queremos crear más de un enemigo? Basta con hacer más de una llamada. Así:

```
PROGRAM juego_labirinto;
GLOBAL
  INT id1;
END
BEGIN
  set_mode(640,480,16,MODE_FULLSCREEN);
  id1=load_fpg("imagenes.fpg");
  personaje();
  labirinto();
  LOOP
    enemigo();
    Frame;
  END
END
```

Fíjate: hemos puesto un LOOP en el proceso principal y dentro una llamada al proceso enemigo();. Además, hemos escrito –antes no estaba- la sentencia FRAME dentro del bucle porque recordad que si no aparece una sentencia FRAME en un proceso con un bucle infinito, el programa se puede colgar (porque los demás procesos estarán pausados siempre hasta que el que no tiene FRAME llegue a él, y si está en un bucle infinito sin FRAME, eso nunca ocurrirá).

Tal como lo hemos hecho, en cada frame se creará un enemigo. Pero como por defecto el programa funciona a 25 frames por segundo (fps, o sea, imágenes que muestra en un segundo; y que recordad que se puede cambiar con la sentencia set\_fps();) se crean demasiados enemigos -¡25 por segundo!-. Por lo tanto hay que hacer algo para que aparezcan menos. Para solucionar eso vamos a usar la conocida sentencia rand(); dentro del proceso principal. Así:

```
PROGRAM juego_labirinto;
GLOBAL
  INT id1;
```

```

END
BEGIN
  set_mode(640,480,16,MODE_FULLSCREEN);
  id1=load_fpg("imagenes.fpg");
  personaje();
  laberinto();
  LOOP
    IF (rand(1,100)== 1)
      enemigo();
    END
  Frame;
  END
END

```

Esto haría que en uno de cada 100 frames se creara un enemigo, porque hay una probabilidad de 1 contra 100 de que el valor devuelto de rand sea igual a 1, y que por tanto se ejecute el interior del IF. Como es aleatorio a veces serán más, a veces serán menos. Pero es muy poco probable que aparezcan todos a la vez, o incluso más de uno o dos cada segundo. Naturalmente, podéis poner la frecuencia que os de la gana. Ej.: IF (rand(1,100)<11),... Está claro por ejemplo que si pusiéramos IF(rand(1,100)<=100) no estaríamos haciendo nada de nada,¿no?.

Ahora toca escribir el código del proceso “enemigo()”. Pero antes, fijate que llamando tal cual al proceso “enemigo()” así a secas, todos los enemigos serán iguales. La manera más práctica –hay otras que se te pueden ocurrir- para solucionar esto existen los parámetros.

Los parámetros recuerda que son datos que le pasamos al proceso en el momento de crearlo, y que se ponen entre los paréntesis al final de la llamada al proceso; luego, en la cabecera del código del proceso en sí, debemos poner qué es cada dato. Por ejemplo, -sólo lo estoy recordando: esto ya lo vimos-, en la llamada al proceso personaje(); podríamos haber puesto personaje(320,240,1); , y en la cabecera de su proceso PROCESS personaje(x,y,graph), omitiendo entonces en su código las asignaciones de los valores X ,Y , GRAPH. De esta manera le hemos asignaríamos directamente en la llamada al proceso esos valores a esas variables (el primer valor a la primera variable, el segundo valor a la segunda variable,...). Por lo tanto, podríamos hacer lo mismo con el proceso enemigo(); , siendo así muy fácil modificarlos y hacer que un enemigo salga, por ejemplo, en un lugar concreto de la pantalla.

Vamos allí: el código del enemigo, suponiendo que los enemigos aparecen en la parte superior de la pantalla y que van hacia abajo, podría ser algo parecido a esto:

```

PROCESS enemigo(x,y,graph,size,int incrementox, int incrementeoy)
BEGIN
  LOOP
    x=x+incrementox;
    y=y+incrementeoy;
    IF (x<-20) BREAK; END
    IF (x>660) BREAK; END
    IF (y>500) BREAK; END
  FRAME;
END

```

END

Y la llamada al proceso desde el programa principal quedaría finalmente:

```
enemigo(rand(1,640),-50,3,rand(50,150),rand(-10,10),rand(5,10));
```

De esta manera se crearía un enemigo en una x al azar entre 1 y 640, 50 puntos por encima de la pantalla, con gráfico 3 (habrá que crear un gráfico y meterlo en el fpg), con tamaño aleatorio (*SIZE* es una variable local predefinida que indica el tamaño del gráfico con respecto al original, en tanto por ciento: 50 quiere decir la mitad y 200 el doble) y cantidad de movimiento horizontal y vertical aleatorio (variables estas dos últimas creadas por nosotros: no son predefinidas). Con estos datos es muy poco probable que se creen dos enemigos exactamente iguales.

Los tres IF que se han escrito son por motivos prácticos. Que el gráfico salga de la pantalla no quiere decir que el proceso haya terminado. Por lo tanto seguirá ejecutando el código para siempre (o hasta que se termine el programa, una de dos). Puede no parecer importante, pero hasta el ordenador más rápido terminará por dar saltos cuando tenga que manejar demasiados procesos, puesto que se están creando nuevos procesos continuamente. Por eso al proceso se le obliga a terminar cuando sale de la pantalla. Fíjate que aparece un BREAK;, el cual fuerza al programa a romper el bucle y continuar justo después del END del mismo. En este caso, como después del END del LOOP viene el END del BEGIN (es decir, no hay más órdenes que ejecutar) el proceso finaliza.

Utilizando parámetros de esta manera, es mucho más fácil crear procesos similares pero con características diferentes. Imagínate que quieres en algún momento queréis crear un enemigo con estas características: x=100; , size=150; , incrementox=10; , incremtoy=5; y un gráfico diferente (graph=4:habría que meter ese gráfico también). Con los parámetros sería suficiente con llamar a ese proceso pasándole esos datos. Sin los parámetros habría que crear un nuevo proceso con su propio código (en el que la mayoría sería idéntico al del otro proceso, pero bueno). Como ves los parámetros son muy útiles.

Pero por ahora los enemigos son inofensivos. Lo único que hacen es aparecer en la pantalla, darse un paseíto y desaparecer. Ahora los vamos a convertir en un peligro para el personaje. En el proceso "personaje();" , después de la comprobación de colisión con el laberinto, habría que hacer una comprobación de colisión con los enemigos. El código a añadir en su proceso sería el siguiente:

```
IF (collision(TYPE enemigo)) BREAK; END
```

De esta manera, al chocar con algún enemigo (cualquiera), la sentencia BREAK; rompería el bucle LOOP, continuando después del END y terminando el proceso (puesto que después se llega al END del proceso), y por tanto, haciendo desaparecer al protagonista y no teniendo pues el jugador nada que controlar, aunque el programa principal seguirá ejecutándose.

Vamos a jugar con ese BREAK;. Si después del END del LOOP escribimos alguna sentencia, ésta se cumplirá una vez que el personaje choque con algún enemigo, y antes de finalizarse. Por lo tanto

podríamos poner algún comando que, por ejemplo, finalice el programa principal. De esta manera, cuando finalice el proceso “personaje()”, éste hará finalizar también el proceso principal. La sentencia para ello, ya la vimos, es exit();

### Añadiendo explosiones:

Aunque la verdad es que eso de salir así, sin más, queda un poco soso. Pues vamos a hacer que el personaje explote. Para eso crearemos otro proceso, llamado “explosion()”. La llamada la haremos después del LOOP del proceso “personaje()” ya que queremos mostrar la explosión justo después de que un enemigo cualquiera haya colisionado con el personaje y justo antes de que el proceso del personaje esté a punto de finalizar. Además, quitaremos de allí el exit() que acabamos de poner, porque si no no podríamos ver la explosión, ya que el programa principal acabaría ipso facto sin dar tiempo a nada más. Por lo que de momento volveríamos otra vez a la situación de que después de ver la explosión el juego continuaría pero sin ningún personaje al que controlar.

**TRUCO IMPORTANTE:** Como queremos que la explosión suceda en el lugar donde está el personaje habrá que pasarle como parámetros la X y la Y del personaje, para que se conviertan en la X y la Y de la explosión. ¿Y cómo se hace eso? Muy sencillo. Supongamos que desde el proceso\_1 se llama a proceso\_2. En general, si en la llamada que hacemos desde proceso\_1 a proceso\_2 ponemos como parámetro el nombre de alguna de las variables de proceso\_1, éste le pasará el valor actual de esa variable a la variable de proceso\_2 que ocupe la posición correspondiente dentro de su cabecera. Por lo que si desde “personaje()” hacemos la llamada así: *explosion(x,y)*; le estamos pasando el valor actual de X e Y del proceso “personaje()” a los dos parámetros de “explosion()”, y si en la cabecera de “explosion()” estos dos parámetros precisamente se llaman X e Y, estaremos dándole los valores de X e Y del proceso personaje a los valores X e Y del proceso explosión, y por lo tanto la explosión tendrá lugar en el sitio donde está el personaje. Justo lo que queríamos. El código de “personaje()” queda así finalmente:

```
PROCESS personaje()
BEGIN
  x=600; y=240; graph=1;
  LOOP
    IF (key(_up)) advance(10); END
    IF (key(_down)) advance(-10); END
    IF (key(_left)) angle=angle+7500; END
    IF (key(_right)) angle=angle-7500; END
    IF (collision(TYPE laberinto)) x=320; y=240; END
    IF (collision(TYPE enemigo)) BREAK; END
  FRAME;
END
  Explosion(x,y);
END
```

Y el nuevo proceso “explosion()” es –cuidado con las tildes: no es lo mismo llamar a un proceso “explosión()” que “explosion()”, serían dos procesos diferentes-:

```
PROCESS explosion(x,y)
BEGIN
  FROM graph=11 TO 20; FRAME; END
```

END

Así de simple. Le hemos pasado como parámetros las coordenadas X e Y del proceso personaje, que es el proceso donde se llamará a “explosion()”.

**TRUCO IMPORTANTE:** Habrás visto que usamos es una sentencia FROM (FOR también vale). Fíjate que como contador se ha puesto la variable GRAPH, por lo que a cada iteración el gráfico asociado al proceso “explosion()” será diferente. Como a cada iteración hemos puesto que se ejecute la sentencia FRAME; a cada iteración se visualizará el cambio de gráfico en pantalla, por lo que se creará un efecto que parecerá que haya una explosión –si las imágenes las has dibujado que parezcan de fuego y que vayan creciendo en tamaño-, ya que se verán 10 imágenes rápidamente seguidas en el mismo sitio. Éste es un truco muy eficaz, pero depende de lo bueno que seas dibujando los momentos diferentes de la explosión. En el ejemplo he usado diez imágenes, y he empezado a contar desde el 11 por una cuestión de orden, nada más: para dejar las imágenes 4-10 para otros usos todavía no especificados (los códigos de los FPGs no hace falta que sean correlativos: puede haber “saltos” entre ellos!).

También se podría aplicar el mismo método de las explosiones a las colisiones con el laberinto (cambiando el reseteo de los valores de X e Y por una sentencia BREAK;).

Ahora deberíamos decidir qué pasa cuando ya ha pasado la explosión: o se acaba el programa definitivamente, o podemos volver a empezar mostrando otra vez el protagonista intacto en el centro de la pantalla como si no hubiera pasado nada. En ambas opciones deberíamos modificar el proceso “explosion()”. Veamos la primera:

Ya vimos que tuvimos que quitar el exit() del proceso personaje porque no nos daba tiempo de ver la explosión. Pero esto no ocurre si ponemos el exit() después del FROM, así:

```
PROCESS explosion(x,y)
BEGIN
  FROM graph=11 TO 20; FRAME; END
  exit();
END
```

¿Por qué? Porque si ponemos el exit dentro de explosion, primero se hará el FROM entero, y una vez que acabe –cuando ya hayamos visto toda la explosión- entonces se dice al programa principal que acabe. De la manera anterior, metiendo exit() en el proceso “personaje()”, cuando se llegaba a la línea de llamada de “explosión()” ésta comenzaba a ejecutarse, pero “personaje()” por su lado también sigue ejecutándose línea a línea –recordad el tema de la concurrencia y la multitarea-, de manera que cuando “explosión()” iniciaba su ejecución, “personaje()” continuaba con la línea siguiente, que era el exit(), por lo que el programa se acababa directamente sin tener la posibilidad de verse la explosión completa.

La otra alternativa, volver a empezar, se realiza modificando el proceso “explosion()” así:

```
PROCESS explosion(x,y)
BEGIN
  FROM graph=11 TO 20; FRAME; END
  personaje();
END
```

Fíjate que lo que hacemos es una nueva llamada, una vez se ha visto la explosión, al proceso "personaje()", con lo que se vuelve a ejecutar el código de ese proceso, que define las variables X,Y y GRAPH, y los movimientos de nuestro protagonista. Fíjate también que cuando "explosion()" llama a "personaje()" para crearlo de nuevo, el proceso "personaje()" anterior hace tiempo que dejó de existir –y por eso sólo se verá un personaje cada vez y no más-, ya que justamente cuando desde "personaje()" se llama a "explosion()", a "personaje()" se le acaba el código y muere. Por lo que digamos que lo que ocurre es que "personaje()" justo antes de morir crea a "explosion()", la cual, a su vez, justo antes de morir, vuelve a crear a "personaje()", con lo que se convierte en un bucle infinito de creación/destrucción de los procesos "personaje()" / "explosion()".

### **ANGLE, SIZE y otras variables locales predefinidas: SIZE\_X, SIZE\_Y, ALPHA y FLAGS:**

Ya hemos acabado nuestro juego del laberinto. No obstante, no está de más comentar con más calma y profundidad las posibilidades que nos ofrecen las nuevas variables introducidas en párrafos anteriores (ANGLE y SIZE), y de paso introducir también el conocimiento y uso de otras variables locales predefinidas muy interesantes, que se utilizarán extensamente a partir de ahora en este manual. Se trata de las variables SIZE\_X, SIZE\_Y, ALPHA, ALPHA\_STEPS, FLAGS y XGRAPH.

Miremos primero un ejemplo donde se insiste en el uso de ANGLE y SIZE. Utilizaremos para ello el mismo archivo "imagenes.fpg" del juego del laberinto (en concreto, su gráfico 001, el triángulo). Lo que hace este programa es cambiar la orientación (el ángulo) y el tamaño del gráfico en forma de triángulo según pulsemos la tecla adecuada del cursor del teclado. Lee los comentarios del código.

```

Program ROTZOM;
  Global
    int rzarrowid;
    int fpg;
  End
  Begin
    set_mode(640,480,16);
    fpg=load_fpg("imagenes.fpg");
    /*Creamos un nuevo proceso llamado "flecha" y almacenamos su ID en la variable global "rzarrowid",
    para que podamos modificar su ángulo y su tamaño posteriormente*/
    rzarrowid = flecha();
    write_var(0,60,0,0,rzarrowid.angle); //Escribimos el ángulo actualizado del proceso "flecha"
    write_var(0,60,10,0,rzarrowid.size); //Escribimos el tamaño actualizado del proceso "flecha"
  Loop
    if(key(_left))rzarrowid.angle=rzarrowid.angle+2000; end //Incrementamos el ángulo en 2 grados
    if(key(_right))rzarrowid.angle=rzarrowid.angle-2000; end //Decrementamos el ángulo en 2 grados
    if(key(_up))rzarrowid.size=rzarrowid.size+10; end //Incrementamos el tamaño en un 10%
    if(key(_down))rzarrowid.size=rzarrowid.size-10;end //Decrementamos el tamaño en un 10%
  If(key(_esc)) exit();End;
  Frame;
  End
End

/*Este proceso mostrará el triángulo en la pantalla. Modificando su variable ANGLE o SIZE podremos
rotarlo o hacerle zoom. Esto se podría hacer desde dentro del propio proceso (más fácil) o, como en este
ejemplo, obteniendo el identificador del proceso en el momento de su creación y modificando esas
variables desde el programa principal*/
Process flecha()

```

```

Begin
  x = 320;
  y = 240;
  file=fpg;
  graph = 1;
  Loop
    Frame;
  End
End

```

Si pruebas el programa, podrás ver los valores que cogen ambas variables a medida que pulsemos las diversas teclas del cursor.

Ya se comentó anteriormente que cuando se gira una vuelta entera una imagen, el valor de la variable ANGLE no se pone a 0 automáticamente sino que continúa creciendo más allá de las 360000 milésimas de grado. Y esto puede ser un lío, y a la larga un problema. Si queremos que a cada vuelta, la variable ANGLE vuelva a valer 0, el truco está en aplicar al valor que queremos que no sobrepase nunca el límite, el módulo con la cantidad máxima a partir de la cual se produce el “reseteo”. Es decir, en nuestro caso, 360000. Pruébalo: sustituye las dos líneas donde se modifica el valor de ANGLE del triángulo (cuando se pulsán los cursores izquierdo y derecho) por éstas otras:

```

if(key(_left))rarrowwid.angle=(rarrowwid.angle+2000)%360000; end
if(key(_right))rarrowwid.angle=rarrowwid.angle-2000%360000; end

```

Si vuelves a ejecutar el programa, verás que ahora el gráfico gira igual, todas las vueltas que quieras, pero cada vez que se acaba de girar una vuelta, el valor de ANGLE vuelve a resetearse a 0, con lo que su valor máximo siempre será 360000.

Este efecto es resultado de aplicar la teoría de la matemática modular, que en general viene a decir que cualquier operación matemática que obtenga un resultado, si a éste se le aplica el módulo de un número determinado, el resultado permanecerá invariable si es menor que ese número, pero si es mayor, el resultado pasará a valer la parte "sobrante". Es decir,  $(2*5+3)\%15=13$ , pero  $(2*5+3)\%10=3$ . Es otra manera (más práctica) de entender lo que llamamos el resto de una división.

Tal como se ha dicho antes, tenemos otras variables locales predefinidas aparte de las ya vistas que son importante que conozcas:

**SIZE\_Y:** Esta variable especifica el escalado vertical que se va a realizar sobre el gráfico. A diferencia de SIZE el escalado sólo se aplica al alto del gráfico, siendo el valor por defecto 100 (100% del alto original). Cuando se especifica un valor distinto de 100 en esta variable o en SIZE\_X , se utilizan siempre los valores de SIZE\_X y SIZE\_Y , despreciándose el valor de SIZE.

**SIZE\_X:** Esta variable especifica el escalado horizontal que se va a realizar sobre el gráfico. A diferencia de SIZE el escalado sólo se aplica al ancho del gráfico, siendo el valor por defecto 100 (100% del ancho original). Cuando se especifica un valor distinto de 100 en esta variable o en SIZE\_Y , se utilizan siempre los valores de SIZE\_X y SIZE\_Y , despreciándose el valor de SIZE.

Un ejemplo (donde utilizaremos el mismo FPG del juego del laberinto, "imagenes.fpg", y mostraremos el gráfico del laberinto, ya que al ser más grande que el del triángulo, se podrá apreciar mejor el efecto)

bastante similar al anterior sería éste:

```
Program SIZES;
Global
    int idlab;
    int fpg;
End
Begin
    set_mode(640,480,16);
    fpg=load_fpg("imagenes.fpg");
    /*Creamos un nuevo proceso llamado "laberinto" y almacenamos su ID en la variable global "idlab"*/
    idlab = laberinto();
    write_var(0,60,0,0,idlab.size_x); //Escribimos el tamaño horizontal del gráfico en % respecto el original
    write_var(0,60,10,0,idlab.size_y); //Escribimos el tamaño vertical del gráfico en % respecto el original
    write_var(0,60,20,0,idlab.size); /*Para comprobar cómo SIZE no se tiene en cuenta cuando se utilizan
    SIZE_X ó SIZE_Y*/
    Loop
        if(key(_left))idlab.size_x=idlab.size_x+10; end
        if(key(_right))idlab.size_x=idlab.size_x-10; end
        if(key(_up))idlab.size_y=idlab.size_y+10; end
        if(key(_down))idlab.size_y=idlab.size_y-10;end
        If(key(_esc)) exit();End;
    Frame;
End
End

Process laberinto()
Begin
    x = 320;
    y = 240;
    file=fpg;
    graph = 2;
    Loop
        Frame;
    End
End
```

**ALPHA:** Contiene un valor entre 0 y 255 para indicar el grado de transparencia que tiene el gráfico. La transparencia total viene dada por ALPHA=0 y la transparencia original de la imagen no se ve alterada con ALPHA = 255. Para que esta variable sea tenida en cuenta por Fénix, la variable FLAGS -vista a continuación- ha de valer 8 (o la suma de 8 más otros valores:8+1,8+2,8+2+1,etc).

ALPHA está íntimamente relacionada con otra variable predefinida GLOBAL, que es **ALPHA\_STEPS**. Esta otra variable indica los niveles de transparencia necesarios. Contiene un valor entre 1 y 255 (por defecto, 16) para indicar la cantidad de niveles de transparencia que se van a definir. Dicho en otras palabras, indica el número de saltos visibles de transparencia que puede haber entre el valor 0 y 255 de ALPHA. Por ejemplo, si ALPHA\_STEPS vale 4, sólo habrá 4 cambios apreciables de transparencia a lo largo de todos los valores posibles de ALPHA. Para saber cada cuántos valores de ALPHA se producirá el cambio de transparencia, se puede usar la fórmula  $256/ALPHA\_STEPS$ . Si, para ALPHA\_STEPS=4, habrá  $256/4=64$  valores de ALPHA consecutivos que ofrecerán la misma transparencia hasta llegar a un salto. Es evidente que a mayor valor de ALPHA\_STEPS, mayor "sensibilidad" existe al cambio de valores de ALPHA. Los casos extremos serían ALPHA\_STEPS=1 (donde no se produciría ningún cambio de

transparencia en el gráfico a pesar de cambiar el valor de su ALPHA) y ALPHA\_STEPS=255, donde cada cambio en una unidad de ALPHA se traduciría visualmente en un cambio de transparencia en el gráfico.

Un ejemplo (donde utilizaremos el mismo FPG del juego del laberinto, "imagenes.fpg", y mostraremos el gráfico del laberinto, ya que al ser más grande que el del triángulo, se podrá apreciar mejor el efecto) bastante similar al anterior sería éste:

*Program ALPHAS;*

*Global*

*int idlab;*

*int fpg;*

*End*

*Begin*

*set\_mode(640,480,16);*

*fpg=load\_fpg("imagenes.fpg");*

*write\_var(0,60,10,0,alpha\_steps);*

*idlab = laberinto();*

*write\_var(0,60,0,0,idlab.alpha);*

*Loop*

*if(key(\_space))while(key(\_space))frame; end idlab.alpha=(idlab.alpha-1)%255; end*

*if(key(\_enter))while(key(\_enter))frame; end alpha\_steps=(alpha\_steps+10)%255;end*

*If(key(\_esc)) exit();End;*

*Frame;*

*End*

*End*

*Process laberinto()*

*Begin*

*x = 320;*

*y = 240;*

*file=fpg;*

*graph = 2;*

*Loop*

*Frame;*

*End*

*End*

En general, por “flag”(bandera en inglés) se entiende aquel parámetro –o variable según el caso- que dependiendo del valor que tenga,modifica la configuración de algo de una manera o de otra: es como un cajón de sastre donde cada valor que puede tener ese parámetro o variable “flag” cambia un aspecto diferente de aquello a lo que afecta.

**FLAGS:** El valor de esta variable activa operaciones especiales que se realizarán en el momento de dibujar el gráfico del proceso en la pantalla. Puede contener uno o la suma de varios de los valores siguientes (si es la suma, los efectos se añadirán uno sobre otro):

1	Espejado horizontal
2	Espejado vertical
4	Transparencia del 50%
8	La transparencia se tomará del valor de ALPHA

16	Blit aditivo
32	Blit sustractivo
128	NO_COLOR_KEY (sin transparencias)

\*El valor por defecto de esta variable es 0.

\*El espejado es mucho más rápido de dibujar que una rotación convencional.

\*El indicador de transparencia (valor 4) corresponde siempre a una transparencia del 50% mientras que el de alpha trabaja en combinación con la variable local ALPHA .

\*El blit aditivo y sustractivo consiste en hacer que el gráfico sume/reste sus componentes de color pixel a pixel con el valor que tenia antes de dibujar el gráfico ese pixel de pantalla, por tanto colores con componentes de color elevadas tiende a la saturación (se acercan al blanco) en el caso del blit aditivo y tienden a 0 en el caso del sustractivo.

\*Para gráficos grandes que no deban respetar transparencias, es interesante utilizar como valor de flags el de NO\_COLOR\_KEY ya que el dibujado sin transparencia es mucho más rápido que el dibujado con transparencias.

También existe la posibilidad de asignar como valor de FLAGS una constante (que equivale a alguno de los valores numéricos anteriores), en vez de directamente dicho valor. Así se gana más claridad en el código. La tabla de correspondencias es la siguiente:

Valor numérico real	Constante equivalente
1	B_HMIRROR
2	B_VMIRROR
4	B_TRANSLUCENT
8	B_ALPHA
16	B_ABLEND
32	B_SBLEND
128	B_NOCOLORKEY

Un ejemplo trivial:

```

Program ALPHAS;
  Global
    int idlab;
    int fpg;
  End
  Begin
    set_mode(640,480,16);
    fpg=load_fpg("graficos.fpg");
  
```

```

    idlab = laberinto();
    write_var(0,60,0,0,idlab.flags);
    Loop
        if(key(_enter))while(key(_enter))frame; end idlab.flags=(idlab.flags+1)%128;end
        If(key(_esc)) exit();End;
        Frame;
    End
End

```

```

Process laberinto()

```

```

    Begin
        x = 320;
        y = 240;
        file=fpg;
        graph = 2;
        Loop
            Frame;
        End
    End
End

```

A continuación presento un ejemplo donde se muestra el efecto de establecer la variable FLAGS a 4 (transparencia). Para lograr observar el resultado, necesitarás tener un FPG llamado "mezclas.fpg" que contengan tres gráficos: un cuadrado rojo de 250x250 píxeles, un cuadrado verde de 250x250 y un cuadrado azul de 250x250. Lo que verás si ejecutas este código es simplemente estos tres cuadrados superpuestos, pero como todos tienen transparencia del 50%, se podrán observar las mezclas de colores que se producen en sus intersecciones. Además, uno de los cuadrados aparecerá y desaparecerá de la pantalla a intervalos regulares, para poder apreciar mejor dichas mezclas.

```

Program mezclas_colores;

```

```

Global
    int contador;
end
Begin
    set_mode(640,480,16);
    load_fpg("mezclas.fpg");
    cuadrado(250,320,1);
    cuadrado(350,280,2);
    parpadeo();
End

```

```

Process cuadrado(x,y,graph)

```

```

    Begin
        flags=4;
        Loop
            Frame;
        End
    End
End

```

```

Process parpadeo()

```

```

    Begin
        cuadrado(150,180,3);
        Loop
            contador++;

```

```
If(contador>10)
    signal(son,s_sleep);
    If(contador>20)
        signal(son,s_wakeup);
        Frame(500);
        contador=0;
    End
End
If(key(_esc)) exit();End
Frame;
End
End
```

Finalmente, si quisiéramos por ejemplo, un espejado horizontal y vertical a la vez con transparencia, FLAGS tendría que valer  $1+2+4=7$ .

## CAPÍTULO 7: TUTORIAL DE UN PING-PONG

En este capítulo vamos a crear nuestro segundo juego escrito en Fénix: el ping-pong clásico de toda la vida: con dos palas a los lados que se moverán arriba y abajo y una pelota que irá rebotando en ellas o perdiéndose fuera de la pantalla si algún jugador no es suficientemente rápido.

Lo primero, como siempre, los gráficos. Sólo necesitaremos dos: el de la pelota y el de las dos palas.

El gráfico de las dos palas será un rectángulo blanco de 30 píxeles de anchura x 100 píxeles de alto, y el gráfico de la pelota será un círculo blanco de unos 30 píxeles de diámetro. Ambas imágenes las incluiremos en un archivo FPG llamado "pingpong.fpg" (el rectángulo que representa las palas con código 001 y el círculo que representa la pelota con código 002).

Comencemos. Lo primero que vamos a escribir (de un tirón) va a ser el esqueleto semivacío de nuestro programa, incluyendo la creación desde el programa principal de los diferentes procesos necesarios, de tal manera que empezemos visualizando todos los elementos que van a jugar un papel en nuestro programa (bola y palas).

Es posible que, en este momento, te preguntes si es necesario que creamos un proceso diferente para cada pala (izquierda y derecha), o si con un mismo proceso ya vale, ya que tienen el mismo gráfico y su comportamiento es muy similar...(igual que hemos hecho en el capítulo anterior con los enemigos, en este caso sólo habría que indicar unas coordenadas en pantalla diferentes para cada pala, pero serían el mismo proceso...)

La respuesta es clara: necesitarás un proceso por cada pala. Las razones pueden ser múltiples, pero la más sencilla y rápida de entender es que para mover cada pala independientemente una de la otra, necesitarás definir teclas diferentes de movimiento. Es decir, si queremos que la pala izquierda se mueva arriba podríamos hacer que esto ocurra pulsando la tecla "q", pero si queremos que sea la pala derecha la que suba, tendremos que utilizar otra tecla diferente. Si tuviéramos un sólo proceso, pulsando "q" subirían las dos palas a la vez...Sólo por eso, ya tenemos un motivo para hacer dos procesos separados.

Así pues, el código que será nuestro punto de partida será algo así como éste (espero que no tengas ningún problema en entenderlo):

```
program hola;
global
    int id1;
end
begin
    set_mode(640,480,16);
    id1=load_fpg("pingpong.fpg");
    pala1();
    pala2();
    bola();
end

process pala1()
begin
    x=90;
    y=240;
    file=id1;
```

```

graph=1;
loop
    if(key(_q)) y=y-10;end
    if(key(_a)) y=y+10;end
frame;
end
end

process pala2()
begin
    x=550;
    y=240;
    file=id1;
    graph=1;
    loop
        if(key(_up)) y=y-10;end
        if(key(_down)) y=y+10;end
frame;
    end
end

process bola()
begin
    x=320;
    y=240;
    file=id1;
    graph=2;
    loop
        frame;
    end
end
end

```

Vemos que el programa principal se encarga básicamente de crear los tres procesos involucrados en el juego, y ¡atención! morir seguidamente. Es la primera vez que no vemos un bucle LOOP/END en un programa principal. Te preguntará por qué no se acaba entonces la ejecución: bueno, porque aunque el código principal ya no esté en ejecución, él antes ha puesto en marcha tres procesos (“pala1”, ”pala2”, y “bola”) los cuales sí tienen un bucle LOOP/END, con lo que estos tres procesos estarán ejecutándose en paralelo perpetuamente. Mientras quede un sólo proceso en funcionamiento, el programa todavía estará corriendo.

Fíjate también que ya hemos hecho que las dos palas se muevan para arriba y para abajo con sendas teclas. No hemos limitado su movimiento vertical a fuera de la pantalla, pero sería fácil hacerlo.

Ahora lo que nos falta es hacer que la bola se mueva, y que al rebotar en alguna de las palas, cambie su dirección de movimiento. Además haremos que si la bola sale de los límites de la pantalla (esto es, ninguna pala ha llegado a tocarla), se acabe el programa.

Para ello, vamos a implementar un sistema provisional que luego mejoraremos. La idea es que nada más crearse el proceso “bola”, ésta empiece a moverse mediante la función *advance()*. Como esta función dirige el gráfico en el sentido que viene marcado por su variable ANGLE, y como el valor de esta variable al inicio de cada proceso es 0 por defecto, la bola empezará a moverse hacia la derecha horizontalmente.

Lo que tenemos que hacer también es que cuando la bola colisione con la pala derecha (la primera que se encontrará), la bola cambie su orientación y se siga moviendo en esa nueva orientación. El cálculo

de la nueva orientación se puede hacer de muchas maneras más o menos complicadas, pero ahora lo haremos fácil: ya que la bola inicialmente se mueve horizontalmente lo que haremos de momento será que solamente se pueda mover en esta dirección, pero cambiando el sentido. O sea, si la bola va de izquierda a derecha antes de impactar con la pala, después tendrá que ir de derecha a izquierda.

Para ello, fíjate lo que tenemos que hacer: la bola inicialmente va a la derecha porque su ANGLE vale 0. Si quisiéramos que fuera hacia la izquierda, tendría que valer 180 grados. Así que lo que haremos será que cuando colisione con la pala1, ANGLE valga 180000. Lo probamos. Modificamos el proceso “bola” de la siguiente manera (el resto del código permanece igual):

```
process bola()
begin
    x=320;
    y=240;
    file=id1;
    graph=2;
    loop
        advance(10);
        if(collision(type pala2))angle=180000;end
        frame;
    end
end
```

Si lo pruebas, verás que efectivamente, al chocar la bola con la pala de la derecha, rebota y va directa a la pala de la izquierda, la cual no reacciona porque todavía no lo hemos dicho que lo haga. Es lo siguiente que haremos.

Un paréntesis antes de continuar: es posible que te hayas preguntado dónde colocar la detección de la colisión: o en el proceso “bola” o en el proceso “pala2”. Si hubiéramos optado por hacerlo en “pala2” no habría sido más difícil que escribir (no lo hagas) la siguiente condición dentro de su código:

```
if(collision(type bola)) identificadorProcesoBola.angle=180000;end
```

No obstante, y esto es una recomendación para todos los juegos que hagas, es MUCHO MAS fácil, cómodo y limpio detectar las colisiones en aquel proceso que vaya a colisionar con más procesos. Es decir, “bola” colisionará con “pala1” y “pala2”. En cambio, las palas sólo colisionarán con “bola”. Esto quiere decir que si detectáramos las colisiones en las palas, ya hemos visto que tendríamos que escribir una línea como la anterior en “pala2” pero otra de muy similar en el código de “pala1”, con lo que tendríamos en dos procesos distintos la colisión con un proceso “bola”. ¿No es mucho más fácil detectar desde el propio proceso “bola” las posibles colisiones que éste puede tener con los demás procesos -en este caso, “pala1” y “pala2”?

Es por tanto buena idea centralizar la detección de colisiones en aquél proceso que vaya a colisionar con más procesos diferentes. Es un consejo

Bueno, volvamos a nuestro ping-pong. ¿Cómo hacer para que cuando la bola choque con la pala de la izquierda (“pala1”), cambie su sentido otra vez 180 grados para volver a dirigirse horizontalmente hacia la derecha? Viendo lo que hemos escrito antes, parece claro que lo que tendremos que escribir justo antes del frame del loop de “bola” es una línea como ésta:

```
if(collision(type pala1))angle=0;end
```

¡Ya tenemos nuestra bola que rebota!

Ahora lo que nos falta es hacer que cuando la bola se pierda en la inmensidad, fuera de los límites de la pantalla, porque una pala se ha movido y no ha logrado colisionar con ella, el programa se acabe. Esto es simple, sólomente hay que añadir una última línea dentro del loop de “bola”:

```
if(x<0 or x>640) exit(); end
```

Nuestro ping-pong ya funciona tal como en principio hemos decidido. No obstante, vamos a hacer el código fuente un poco “más elegante”: sustituye los dos ifs que detectan las colisiones de “bola” con “pala1” y “pala2” dentro del código de “bola” por este único if:

```
if(collision(type pala1) or collision(type pala2))angle=(angle+180000)%360000;end
```

Esta línea hace lo mismo que las otras dos que hemos quitado. ¿Cómo lo hace? Para empezar, la condición comprueba si la bola colisiona con “pala1” O “pala2”. En cualquiera de estas dos colisiones, hará lo mismo, que es cambiar el ANGLE de la bola por un valor obtenido de una expresión un tanto exótica.

Estudieemos qué valores obtenemos. Inicialmente, cuando la bola se mueve horizontalmente a la derecha, su ANGLE es 0, así que cuando choca con “pala2”, su nuevo ANGLE es el que tenía antes (es decir, 0) más 180000 módulo 360000: es decir, 180000. Correcto. Ahora la bola va horizontalmente hacia la izquierda. Cuando choca con “pala1” su nuevo ANGLE es el que tenía antes (es decir, 180000) más 180000 módulo 360000, es decir 360000 módulo 360000, es decir, 0. ¡Volvemos a la situación inicial, con la bola dirigida horizontalmente a la derecha! Es decir, que con una expresión un poco más ingeniosa, podemos hacer las cosas más bien hechas.

Supongo que estarás pensando ahora mismo que este ping-pong es un churro: la pelota sólo se mueve en horizontal, no hay marcador de puntuaciones, no se vuelve a empezar la partida cuando la bola sale fuera de la pantalla...Vamos a intentar solventar estas carencias.

Lo único que será modificado respecto el código que tenemos en estos momentos es el proceso “bola”: el resto de procesos y el código principal serán los mismos.

Un consejo: por cada cosa que cambiemos en el código, pruébalo a ejecutar, para ir observando cómo el juego va cambiando poco a poco: si introduces todos los cambios de golpe y lo pruebas al final, no habrás notado las pequeñas mejoras que pasito a pasito se han ido incorporando, y te habrás perdido parte del encanto de este tutorial.

Lo primero que vamos a hacer es que la bola se pueda mover en cualquier dirección de la pantalla, no solamente en horizontal. Para ello, haremos que cuando el proceso “bola” se cree, su orientación sea aleatoria. Esto consiste simplemente en añadir justo antes (fuera) del bucle Loop la línea

```
angle=rand(0,360000);
```

Y también cambiaremos lo de que si la bola excede los límites laterales de la pantalla el juego se acaba. Ahora haremos que cuando ocurra esto, la bola se resitúe en el centro de la pantalla, y además, con una nueva orientación aleatoria, como si volviéramos a empezar. Para hacer esto, has de quitar el if que ejecutaba la orden exit() y poner en su lugar estos dos ifs:

```
if(x<0)
    x=320;
    y=240;
    angle=rand(0,360000);
```

```
end
```

```
if(x>640)
```

```
    x=320;
```

```
    y=240;
```

```
    angle=rand(0,360000);
```

```
end
```

¿Por qué escribimos dos ifs diferentes si hacen lo mismo? Con una condición OR -como estaba antes- ya valdría...Sí, es cierto, pero lo hemos hecho así pensando en los puntos de cada jugador. No es lo mismo que la bola haya salido por el lado izquierdo de la pantalla (momento en el que el jugador que maneja la pala derecha aumentará un punto en su marcador) que la bola haya salido por el lado derecho (momento en el que el jugador que maneja la pala izquierda aumentará un punto en su marcador). Dentro de cada uno de los dos ifs tendremos que aumentar en uno la puntuación de un jugador diferente. Es por eso que tenemos que escribir dos. Vamos a ello: a continuación presento el código del proceso “bola” tal como tiene que quedar (las líneas nuevas están marcadas en negrita) para que se vea la puntuación de cada jugador:

```
process bola()
```

```
private
```

```
    int punts1,punts2;
```

```
end
```

```
begin
```

```
    x=320;
```

```
    y=240;
```

```
    file=id1;
```

```
    graph=2;
```

```
    angle=rand(0,360000);
```

```
    write_var(0,90,50,4,punts1);
```

```
    write_var(0,550,50,4,punts2);
```

```
    loop
```

```
        advance(10);
```

```
        if(collision(type pala1) or collision(type pala2)) angle=(angle+180000)%360000;end
```

```
            if(x<0)
```

```
                punts2=punts2+1;
```

```
                x=320;
```

```
                y=240;
```

```
                angle=rand(0,360000);
```

```
            end
```

```
        if(x>640)
```

```
            punts1=punts1+1;
```

```
            x=320;
```

```
            y=240;
```

```
            angle=rand(0,360000);
```

```
        end
```

```
        frame;
```

```
    end
```

```
end
```

Vemos que hemos declarado dos variables privadas, “punts1” y “punts2” que representan respectivamente los puntos del jugador que maneja “pala1” y del que maneja “pala2”. Vemos también que mostramos los puntos por pantalla mediante write\_var (actualizándose cuando sea necesario,pues, sin

necesidad de escribir texto dentro del loop -con el engorro de estar usando `delete_text()` para evitar el error de “Demasiados textos en pantalla”-). Y vemos que si la bola sale por la izquierda de la pantalla, aumenta un punto en el marcador del jugador de la pala 2, y si sale por la izquierda, aumenta un punto en el marcador del jugador de la pala 1.

Ya que estamos, podríamos hacer ahora que cuando se llegara a un máximo de puntuación, el juego acabara. Por ejemplo, si un jugador cualquiera llega a 3 puntos, ya ha ganado y se acaba el juego. ¿Cómo hacemos esto? Por ejemplo, escribiendo el siguiente `if` justo antes de la línea `Frame`; del `Loop` del proceso “bola”.

```
if (punts1 == 3 or punts2 == 3)
    while(not key(_enter))
        delete_text(0);
        write(0,320,200,4,"GAME OVER!");
        frame;
    end
end
exit();
end
```

Estudiamos este `if`. Lo que dice es que si cualquiera de los dos jugadores llega a tres puntos ( es decir, si “`punts1`” o “`punts2`” vale 3), nos meteremos en un bucle, del cual sólo podremos salir si pulsamos la tecla `Enter`. Cuando la pulsemos, se acabará el programa. Y mientras no la pulsemos, lo que veremos será un simple texto centrado en la pantalla que dice “`Game Over`”. Fíjate que necesitamos poner la orden `delete_text()` para evitar que el bucle `while` imprima demasiados “`Game Over`” uno encima de otro -y nos dé el error ya conocido de “`Demasiados textos...`”-, y también necesitamos poner una orden `Frame`; porque recuerda que para que se vea ese texto (y el programa no se quede colgado), necesitamos lanzar un fotograma.

Fíjate también que cuando hemos llegado al `Game Over` y no hemos pulsado todavía el `Enter` (y por tanto, estamos dentro del `while`), podemos seguir moviendo las palas tranquilamente -son procesos independientes-, y la bola se ha vuelto a situar en el centro de la pantalla con una orientación al azar, pero no se mueve (evidentemente). ¿Por qué no se mueve? Porque para hacerlo se ha de ejecutar la línea que consta de la orden `advance()`, al principio del `Loop` de este proceso, cosa que ahora es imposible hacer porque estamos encerrados dentro de las iteraciones infinitas del `while` y no podemos salir de allí, a no ser que pulsemos el `Enter`, el cual no es solución porque en ese momento saldremos ya del programa.

Y llegamos a la parte posiblemente más complicada: la gestión de los rebotes. Hay muchas maneras de simularlos, y muchas maneras mejores que la que se va a presentar aquí, pero en general, las maneras más realistas y sofisticadas necesitan por parte del programador de unos conocimientos elevados de geometría y de física. Nosotros ahora no implementaremos ningún sistema de choques, deformaciones ni rebotes basado en ninguna ley física ni formularemos grandes ecuaciones, sino que programaremos un sistema sencillamente pasable y mínimamente creíble.

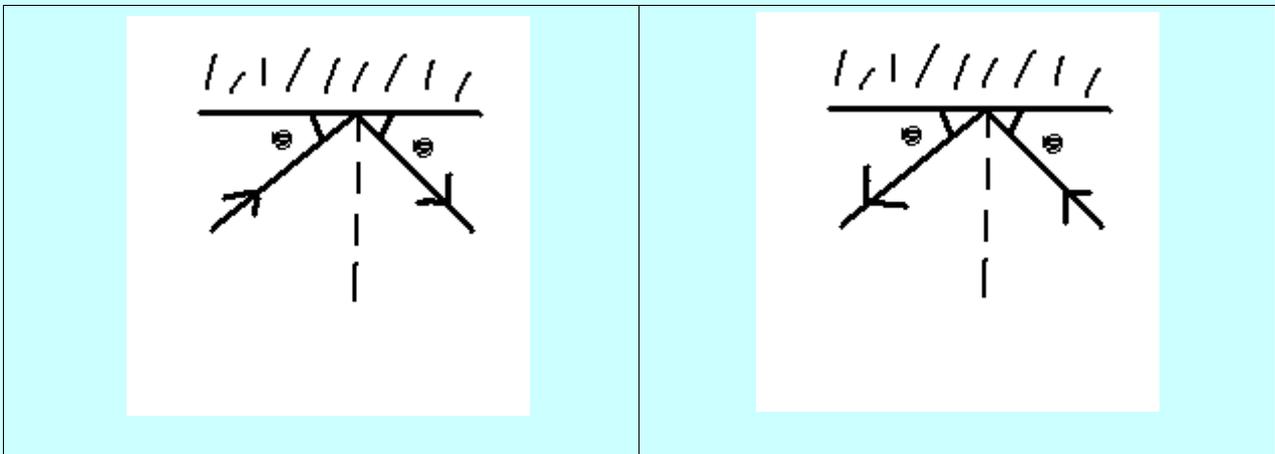
Lo que haremos será lo siguiente: distinguiremos (porque así lo queremos) entre los choques de la bola contra los límites inferior y superior de la pantalla y los choques de la bola contra las dos palas.

\*En los choques que sufre la bola contra los extremos inferior y superior de la pantalla, el ángulo de salida de la bola respecto la superficie de choque será el mismo que el ángulo de incidencia de la bola respecto ésta. O dicho de otra manera, el ángulo de reflexión será el mismo que el de incidencia (respecto la superficie, aunque de hecho, respecto la perpendicular también sería igual).

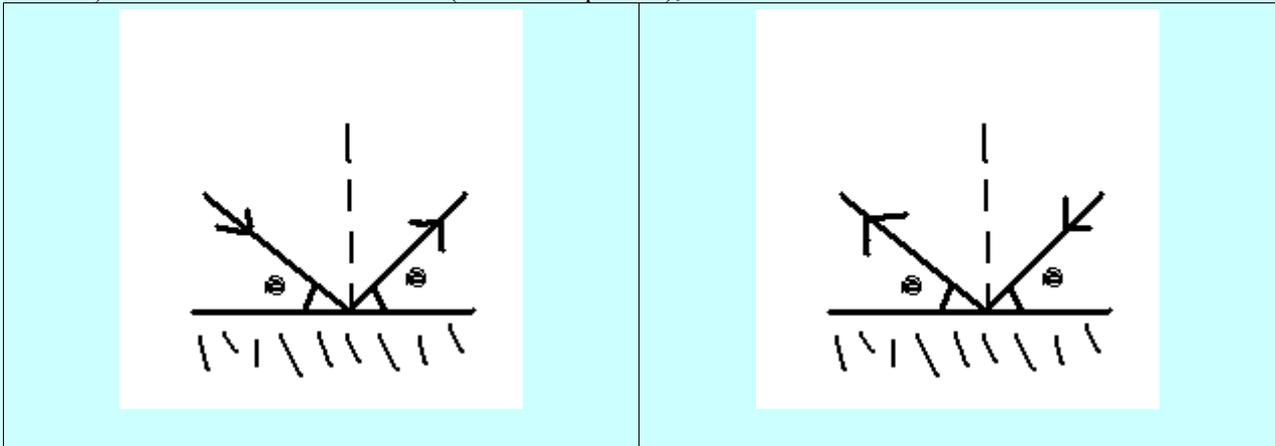
\*En los choques que sufre la bola contra las palas, para hacer el juego más interesante e imprevisto, lo que haremos será que el ángulo de salida de la bola respecto la superficie de choque sea un ángulo aleatorio, pero siempre dentro del mismo cuadrante donde estaría el ángulo de reflexión si éste fuera el mismo que el de incidencia (el caso anterior).

Una vez claro este punto, debemos estudiar los posibles casos con que podemos encontrarnos, que son varios.

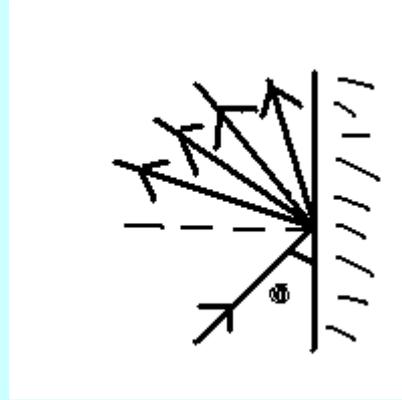
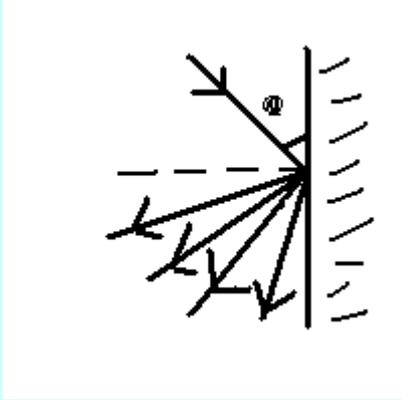
\*Si la bola choca contra el límite superior de la pantalla, puede hacerlo viniendo desde la izquierda (hacia la derecha) o viniendo desde la derecha (hacia la izquierda), o sea:



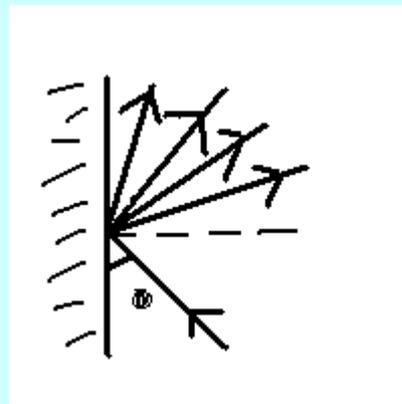
\*Si la bola choca contra el límite inferior de la pantalla, puede hacerlo viniendo desde la izquierda (hacia la derecha) o viniendo desde la derecha (hacia la izquierda), o sea:



\*Si la bola choca contra la pala derecha ("pala2"), puede hacerlo viniendo desde arriba (hacia abajo) o viniendo desde abajo (hacia arriba), o sea:



\*Si la bola choca contra la pala izquierda (“pala1”), puede hacerlo viniendo desde arriba (hacia abajo) o viniendo desde abajo (hacia arriba), o sea:



A partir de aquí, deberíamos coger lápiz y papel y empezar a dibujar triangulitos rectángulos para descubrir equivalencias de ángulos gracias a que sabemos de la escuela que la suma de los tres ángulos de cualquier triángulo es 180°. Si hacemos esto con calma y además tenemos en cuenta que ANGLE siempre será el ángulo entre la línea horizontal y donde apunte el gráfico del proceso, nos saldrán las siguientes expresiones para conseguir que la orientación de la bola después del choque sea la que nosotros queremos:

\*Si la bola choca contra el límite superior viniendo desde la izquierda (es decir, con  $ANGLE > 0^\circ$  y  $\leq 90^\circ$ ):

$angle = (360000 - angle) \% 360000$ ; o lo que es lo mismo:  $angle = -angle$ ;

\*Si la bola choca contra el límite superior viniendo desde la derecha (es decir, con  $ANGLE > 90^\circ$  y  $\leq 180^\circ$ ):

$angle = (360000 - angle) \% 360000$ ; o lo que es lo mismo:  $angle = -angle$ ;

\*Si la bola choca contra el límite inferior viniendo desde la izquierda (es decir, con  $ANGLE > 180^\circ$  y  $\leq 270^\circ$ ):

$angle = (360000 - angle) \% 360000$ ; o lo que es lo mismo:  $angle = -angle$ ;

\*Si la bola choca contra el límite inferior viniendo desde la derecha (es decir, con  $ANGLE > 270^\circ$  y  $\leq 360^\circ$ ):

$angle=(360000-angle)\%360000$ ; o lo que es lo mismo:  $angle=-angle$ ;

\*Si la bola choca contra la pala derecha viniendo desde la arriba (es decir, con  $ANGLE>270^\circ$  y  $\leq 360^\circ$ ):

*El nuevo ANGLE puede tener cualquier valor entre  $180^\circ$  y  $270^\circ$*

\*Si la bola choca contra la pala derecha viniendo desde la abajo (es decir, con  $ANGLE>0$  y  $\leq 90^\circ$ ):

*El nuevo ANGLE puede tener cualquier valor entre  $90^\circ$  y  $180^\circ$*

\*Si la bola choca contra la pala izquierda viniendo desde la arriba (es decir, con  $ANGLE>180^\circ$  y  $\leq 270^\circ$ ):

*El nuevo ANGLE puede tener cualquier valor entre  $270^\circ$  y  $360^\circ$*

\*Si la bola choca contra la pala izquierda viniendo desde la abajo (es decir, con  $ANGLE>90^\circ$  y  $\leq 180^\circ$ ):

*El nuevo ANGLE puede tener cualquier valor entre  $0^\circ$  y  $90^\circ$*

Sorprendentemente, en todos los casos donde chocamos contra los límites inferior o superior de la pantalla nos ha salido la misma expresión, y vemos también que las expresiones son diferentes para las dos palas, incluso dependiendo de la orientación inicial de la bola en cada una de ellas.

Una vez que ya hemos encontrado las expresiones adecuadas a cada caso, lo único que nos falta es escribirlas convenientemente en nuestro código fuente, y ya habremos acabado nuestro juego. A continuación presento el código fuente completo, y en negrita las líneas que tenemos que añadir para implementar todo lo que acabamos de discutir sobre los choques contra los límites de la pantalla y las palas

```
program hola;
global
    int id1;
end
begin
    set_mode(640,480,16);
    id1=load_fpg("pingpong.fpg");
    pala1();
    pala2();
    bola();
end

process pala1()
begin
    x=90;
    y=240;
    file=id1;
    graph=1;
    loop
        if(key(_q)) y=y-10;end
        if(key(_a)) y=y+10;end
        frame;
    end
end

process pala2()
```

```

begin
    x=550;
    y=240;
    file=id1;
    graph=1;
    loop
        if(key(_up)) y=y-10;end
        if(key(_down)) y=y+10;end
        frame;
    end
end

process bola()
private
    int punts1,punts2;
end
begin
    x=320;
    y=240;
    file=id1;
    graph=2;
    angle=rand(0,360000);
    write_var(0,90,50,4,punts1);
    write_var(0,550,50,4,punts2);
    loop
        advance(15);

/*Hemos quitado la sentencia que teníamos hasta ahora que controlaba el choque de la bola con las dos palas, y hemos separado en dos ifs las colisiones dependiendo de si la bola choca con pala1 o pala2, ya que su orientación será diferente dependiendo de con quien choque*/
        if(collision(type pala1))//Pala izquierda
            if(angle > 90000 and angle < 180000) //La bola viene de abajo
                angle=rand(0,90000);//Va al primer cuadrante
            end
            if(angle > 180000 and angle < 270000)//La bola viene de arriba
                angle=rand(270000,360000); //Va al cuarto cuadrante
            end
        end

        if(collision(type pala2))//Pala derecha
            if(angle > 270000 and angle < 360000)//La bola viene de arriba
                angle=rand(180000,270000); //Va al tercer cuadrante
            end
            if(angle > 0 and angle < 90000) //La bola viene de abajo
                angle=rand(90000,180000); //Va al segundo
            end
        end
    end

/*Si la bola choca con el límite superior o inferior de la pantalla, venga de donde venga, se comportará igual*/
        if(y<0 or y>480)
            angle=(360000-angle)%360000;

```

```

        end

        if(x<0)
            punts2=punts2+1;
            x=320;
            y=240;
            angle=rand(0,360000);
        end

        if(x>640)
            punts1=punts1+1;
            x=320;
            y=240;
            angle=rand(0,360000);
        end

        if (punts1 == 3 or punts2 == 3)
            while(not key(_enter))
                delete_text(0);
                write(0,320,200,4,"GAME
OVER!");
                frame;
            end
            exit();
        end

        frame;
    end
end
end

```

En verdad, el movimiento de la bola y la gestión de sus choques se podría haber hecho de una manera mucho más simple e igualmente efectiva: sólo con un poco de ingenio, y sin tanto lío con los ángulos.

A continuación presento el código completo de otro juego de pingpong muy similar (funcionará con el mismo FPG del ejemplo anterior), donde se puede apreciar una técnica diferente para hacer que la bola se mueva y detecte los choques con la parte superior e inferior de la pantalla y con las dos palas.

```

program pingpong2;
global
/*Cuando "fin" valga 1 (es decir, cuando la bola salga por primera vez por la derecha o la izquierda de la
pantalla, el juego acabará*/
    int fin = 0;
end
begin
    set_mode(320,240,16);
    load_fpg("pingpong.fpg");
    bola(160,120);
    pala1(5,120);
    pala2(315,120);
    loop
/*Este if estará pendiente de si el juego ha acabado. Si es así,y se pulsa "x", se crea otra vez el proceso

```

```

bola (que está muerto en este momento) y se vuelve a empezar a jugar*/
    if (fin == 1 and key(_x))
        delete_text(0); //Elimino el texto del GameOver
        bola(160,120);
        fin = 0;
    end
    frame;
end
end

process pala1(x,y)
begin
    graph=1;
    loop
        if(key(_q) and y>10)y=y-5;end
        if(key(_a) and y<230)y=y+5;end
        frame;
    end
end

process pala2(x,y)
begin
    graph=1;
    loop
        if(key(_up) and y>10)y=y-5;end
        if(key(_down) and y<230)y=y+5;end
        frame;
    end
end

process bola(x,y)
private
    int dx = 5; //Cantidad de píxeles que se moverá la bola cada frame en dirección horizontal
    int dy = 5; //Cantidad de píxeles que se moverá la bola cada frame en dirección vertical
    int dys=1; //Complemento a dy para calcular el movimiento vertical de forma más compleja
end
begin
    graph = 2;
    loop
//Las dos líneas siguientes definen el movimiento de la bola en cada frame
        x=x+dx;
        y=y+dy;
        if(collision(type pala1) or collision(type pala2))
            dx = -dx; //Invierto el movimiento horizontal (derecha <->
izquierda)
/*Las dos líneas siguientes establecen el valor del incremento vertical del movimiento de la bola. Lo
primero que hago es hacer que "dys" valga 1 o -1 dependiendo del valor actual de "dy" y de su valor
absoluto (ABS() es un comando de Fénix que devuelve el valor absoluto del valor numérico que se le
ponga como parámetro). Si actualmente "dy" es positivo, "dys" valdrá 1. Si actualmente "dy" es negativo,
"dys" valdrá -1. Una vez que "dys" vale 1 o -1, se establece el nuevo incremento en la posición y de la bola
a partir de un número aleatorio entre 1 y 5 multiplicado por 1 ó -1. Tal como hemos dicho, si el anterior
"dy" era positivo, "dys" será 1 y los valores posibles del nuevo "dy" serán 1,2,3,4 o 5, aleatoriamente. Si el
anterior "dy" era negativo, "dys" valdrá -1 y los valores posibles del nuevo "dy" serán -1,-2,-3,-4 o -5,
aleatoriamente. Fijate que si la bola baja ("dy" positivo), continúa bajando ya que "dy" continúa siendo

```

```

positivo- aunque posiblemente con otro valor, y por tanto, a otra velocidad-, y si la bola sube ("dy"
negativo), continúa subiendo.Sólo se cambiará el sentido vertical cuando la bola choque contra los límites
inferior o superior de la pantalla, no contra las palas.*/
        dys = dy/abs(dy);
        dy = rand(1,5)*dys;

        end
/*Si la bola choca contra los límites inferior o superior de la pantalla, invierto el movimiento vertical
(arriba <-> abajo)*/
        if(y<3 or y>237)
            dy = -dy;
        end
/*Si la bola sale fuera de los límites izquierdo o derecho de la pantalla, llamo al proceso "gameover" y
hago un suicidio del proceso "bola" mediante return*/
        if(x<0 or x>320)
            gameover();
            return;
        end
        frame;
    end
end;

/*Proceso que simplemente escribe un texto en pantalla y asigna a "fin" el valor de 1, con lo que el if del
programa principal ya se podrá ejecutar si se pulsa "X", para volver a jugar.Este proceso es muy pequeño
y se podría haber eliminado, escribiendo su código directamente en el mismo lugar donde se le llama*/
process gameover()
begin
    write(0,160,110,4,"GameOver");
    write(0,160,120,4,"Pulsa X para volver a empezar");
    fin = 1;
end

```

A continuación presento un nuevo código de un juego de ping-pong (¡otro más!) donde podemos encontrar otra manera mucho más fácil todavía de gestionar los choques de la bola contra las palas (con una única fórmula simple: ¡averigua con lápiz y papel cómo se llega ella!), y además tenemos el detalle interesante de que las dos palas están controladas por el mismo proceso "pala", en vez de con dos procesos independientes. Como novedad, la velocidad de la bola va aumentando a medida que se juega. Para poderlo ejecutar necesitarás el mismo FPG de los ejemplos anteriores, "pingpong.fpg".

```

program tutorial;
global
    int score[1]; /*El elemento score[0] es la puntuación del jugador de la izquierda y el
elemento score[1] es la puntuación del jugador de la derecha*/
    int fpg;
end
begin
    set_mode(320,240,16);
    fpg=load_fpg("pingpong.fpg");
    bola();
    pala(20,_up,_down);
    pala(300,_q,_a);

    write(0,160,20,4,"-");
    write_var(0,100,20,4,score[0]);

```

```

        write_var(0,220,20,4,score[1]);

        loop
            if(key(_esc) exit(); end
            frame;
        end
    end

Process bola()
private
    int speedo; //Velocidad de la bola
    /*Guarda el código identificador del proceso pala en caso de que colisione con él*/
    int pala_id;
end
begin
    graph=2;
    x = 160;
    y = 120;
    angle = rand(0,360) * 1000;
    speedo = 500; //Velocidad inicial. Irá aumentando progresivamente
    loop
        /*Si la bola está a punto de superar el limite inferior o superior de la pantalla (si la bola tiene un radio de
        5 píxeles...para que no se deje de ver entera*/
        if(y<5 or y>235)
            //El ángulo de entrada es el ángulo de salida!
            angle = -angle;
        end
        /*Si la bola sale por la izquierda, punto para el jugador con score[0]*/
        if(x<0)
            score[0]=score[0]+1;
            //Se resetea la velocidad a la estandar

            speedo = 500;
            //Se resetea la posición de la bola
            y = 120;
            x = 220;
        end
        /*Lo mismo para el extremo derecho de la pantalla, sólo que ahora con el marcador de la otra pala*/
        if(x>320)
            score[1]++;
            speedo = 500;
            y = 120;
            x = 100;
        end
        //Si choca contra una pala, SEA CUAL SEA
        if(pala_id = collision(type pala))
        /*Con esta fórmula tan sencilla obtenemos la nueva orientación de la bola después del choque*/
            angle = -angle + 180000;
            advance(3);
        end
        //Se incrementa lentamente la velocidad a cada frame
        speedo=speedo+3;
        //Mueve la bola en la dirección actual

```

```

    advance(speedo/100);
    frame;
end //loop
end

```

*/\*Creo un único proceso "pala" para las dos palas, pasando como parámetro las teclas que regirán el movimiento de cada una de ellas, y su posición horizontal (que son las únicas cosas que diferencian una pala de la otra, porque hemos visto que con una misma fórmula se pueden gestionar los choques de la bola con las dos palas.\*/*

```

Process pala(x,int keyup,int keydown)
begin

```

```

    y = 120;
    graph=1;
    loop

```

```

        if(key(keyup)) y =y-12; end

```

```

        if(key(keydown))y =y+12; end

```

*/\*Limito los movimientos de las raquetas a los extremos de la pantalla\*/*

```

        if(y < 20) y=20; end

```

```

        if(y > 220)y=220; end

```

```

    frame;

```

```

end

```

```

end

```

### **Variante: el juego del "Picaladrillos":**

A partir del código básico del ping-pong podemos crear múltiples juegos sencillos que se basan en los mismos principios de colisiones con bolas y rebotes. Todo dependerá de hasta dónde llega nuestra imaginación. Como ejemplo, presento a continuación un juego muy sencillo, el "rompeladrillos".

El jugador controla una raqueta que puede mover solamente de forma horizontal con los cursores, la cual dirigirá una pelota cuya función es destruir (colisionando con ellos) una serie de ladrillos colocados a lo largo del borde superior de la pantalla. Si la bola cae por el borde inferior de la pantalla se pierde: para generar otra bola habrá que apretar la tecla CTRL. El objetivo es destruir todos los ladrillos.

Para probar el juego, necesitarás un fpg llamado "breakout.fpg" compuesto por tres imágenes. La imagen 1 será la bola (de 25x25 está bien, pero no es un tamaño obligatorio), la imagen 2 será un ladrillo (que para este código de ejemplo concreto sí es conveniente -por las medidas de pantalla implicadas- que sus medidas sean 28x138) y la imagen 3 será la pala (con un tamaño de 120x120 está bien, y que para hacerlo más interesante podría tener una forma irregular, como ésta)



```

Program breakout;

```

```

global

```

```

    int blocksleft;//Contará la cantidad de ladrillos que hay aún

```

```

end

```

```

private

```

```

        int i,j;//Variables para los bucles
end
begin
    set_mode(320,240,16);
    load_fpg("breakout.fpg");
    paddle(160,220); //Raqueta
    ball(160,80);//Bola
    /*Los dos for siguientes sirven para dibujar en pantalla los ladrillos que tendremos que
destruir. La estructura es sencilla: el primer bucle sirve para darle 10 posiciones horizontales diferentes a
los ladrillos, y el segundo sirve para dar 3 posiciones verticales. Así que cada vez que lleguemos a la
llamada al proceso block, dibujaremos un ladrillo en una de las 10 posiciones horizontales posibles y en
una de las 3 posiciones verticales posibles, con lo que al final tendremos 30 ladrillos distribuidos en 3
filas y 10 columnas*/
        for(i=0;i<10;i++)
            for(j=0;j<3;j++)
                blocksleft++;/*Como creamos un ladrillo,tenemos que añadirlo al
contador*/
                    block(i*32+16,j*16+8);/*Dibujamos el ladrillo propiamente
dicho.Los números están para ajustar la posición a las medidas de la pantalla.Recuerda que para que
cuadre el gráfico del ladrillo ha de tener unas dimensiones de 28x138*/
                end
            end
        write(0,180,230,5,"Ladrillos que faltan: ");
        write_var(0,180,230,3,blocksleft);
        loop
            if(key(_control))/*Puedes crear una bola extra cuando quieras*/
                while(key(_control))frame;end/*Esperar hasta que la tecla se suelte*/
                ball(160,80);
            end
        frame;
        end
end

process block(x,y)
begin;
    graph=2;
    loop
        frame;
    end
end

process paddle(x,y)
begin
    graph=3;
    loop
        if(key(_left))x=x-5;end
        if(key(_right))x=x+5;end
        if(x<15)x=15;end
        if(x>305)x=305;end
        frame;
    end
end

process ball(x,y)

```

```

private
    int vx=0;//Su velocidad inicial (cambiará) en la dirección X e Y
    int vy=5;
    int paddle;
    int block;
end
begin
    graph=1;
    loop
        paddle=COLLISION(type paddle); //Para no tener que escribir tanto después en
los ifs
        block=COLLISION(type block); //Lo mismo
        //Si la bola choca contra un ladrillo
        if(block!=0)
            /*Comprobamos que hemos chocado contra la parte inferior o superior de un
ladrillo...*/
                if(abs(block.y-y+vy)>8)
                    vy=-vy;//...e invertimos la velocidad vertical
                end
            /*Comprobamos que hemos chocado contra la parte izquierda o derecha de un
ladrillo...*/
                if(abs(block.x-x+vx)>16)
                    vx=-vx;//...e invertimos la velocidad horizontal
                end
            /*La bola no necesariamente choca sólo contra un ladrillo a la vez, sobretodo
cuando incide de forma horizontal sobre ladrillos superpuestos, donde choca con todos ellos a la vez.*/
                while(block!=0)
                    blocksleft--;
                    signal(block,s_kill);
                /*Vemos si hay otro ladrillo con el que haya colisionado.(Recordad que cada vez que se llama a
"collision", se da el siguiente item con el que se ha colisionado, hasta que se haga un frame,
momento en el que se "resetea" la lista de objetos colisionantes y se empieza otra vez*/
                    block=COLLISION(type block);
                end
            end

            //Si la bola choca contra la raqueta...
            if(paddle!=0)
                vy=-vy;//...invertimos la velocidad vertical...
                vx=vx+(x-paddle.x)/2;//...y alteramos la velocidad horizontal
dependiendo de en qué punto de la raqueta se haya colisionado..*/
            end
            //Si la bola choca contra los límites de la pantalla (excepto el inferior)*/
            if(x<=2 or x>=318)vx=-vx;end
            if(y<=2)vy=-vy;end

            /*Limitamos la velocidad horizontal un poco (para que no se embale mucho)... */
            if(vx>20)vx=20;end
            x=x+vx;//y la bola se mueve
            y=y+vy;
            if(y>260)return;end/*Si perdemos la bola, matamos el proceso*/
            frame;
        end
    end
end

```

## CAPÍTULO 8:TUTORIAL PARA UN MATAMARCIANOS

(extraído del tutorial de Drumpi, en <http://drumpi.esp.st>)

El matamarcianos que explicaré en este tutorial es tremendamente parecido al juego del laberinto presentado en un capítulo anterior, con unas pocas alteraciones. Recomiendo leer pues aquel tutorial antes para poder seguir el tutorial presente, ya que éste se puede considerar una síntesis resumida – con algunas adaptaciones mínimas pero interesantes- del juego del laberinto, y por tanto no se volverán a explicar conceptos que ya hayan aparecido anteriormente.

### **Punto de partida:**

Empezaremos dibujando un fondo estrellado –de color azul oscuro con puntitos blancos- que representará el universo, y una nave espacial, que será el protagonista que controlaremos, de unos 70x70 píxeles, y mirando hacia arriba. Recuerda que si usamos colores de 16 bits, el color transparente será el negro absoluto.

Un problema que podemos tener es el no poder usar el negro absoluto como color en nuestra nave, pero eso se soluciona usando un negro casi absoluto (por ejemplo si las componentes RGB son 0,0,1), la diferencia no se apreciará.

Si el gráfico (y el FPG) fuera de 8 bits -256 colores, cosa que nosotros no usaremos-, no está de más saber que el color transparente se consigue gracias a que se establece que el color 0 de la paleta (el primero de la paleta) determina las partes que no se van a dibujar. Si deseas tener el negro absoluto entre los colores de tu nave debes tenerlo en otra parte de la paleta. En principio, de todas maneras, no trabajaremos con paletas de 8 bits en este manual.

Una vez que tengamos las imágenes, las añadiremos a un FPG llamado “prueba.fpg”: el fondo con código 001 y la nave con código 002, y comenzaremos nuestro programa escribiendo esto (se explica por sí solo):

*Program ejemplo;*

*Global*

*Int Graficos;*

*End*

*Begin*

*set\_mode(640,480,16);*

*set\_fps(60,1);*

*Graficos=load\_fpg(“prueba.fpg”);*

*Put\_screen(graficos,1);*

*Nave();*

*Loop*

*If (key(\_esc)) break; end*

*Frame;*

*End*

*Unload\_fpg(graficos);*

*End*

*Process nave ()*

*Begin*

*Graph=2; /\*No es necesario utilizar la variable FILE porque sólo hemos cargado un FPG\*/*

*X=320;*

*Y=435; //La nave está a 10 píxeles del extremo inferior de la ventana*

*Loop*

```

    If (key(_left)) x=x-4; end
    If (key(_right)) x=x+4; end
    If (x>600) x=600; end //Para que la nave no se salga por la derecha
    If (x<40) x=40; end //Para que la nave no se salga por la izquierda
    Frame;

```

End

end

Lo primero que añadiremos al código es la orden *let\_me\_alone()*, que la escribiremos justo antes del end final del programa principal (después del *unload\_fpg(graficos)*), para que cuando salgamos del bucle de éste, matemos todos los procesos que pudieran estar activos en ese momento (de momento sólo tenemos el proceso "Nave", pero en seguida pondremos otros, como "Enemigo" o "Disparo"), y podamos acabar el programa limpiamente y sin problemas, una vez se haya acabado de ejecutar el código del programa principal..

### **Añadiendo los disparos de nuestra nave. La variable local predefinida Z:**

Seguidamente, haremos el dibujo del disparo, de unos 10x30 píxeles. El código para guardarlo en el FPG será el 3. Los disparos son un nuevo elemento con autonomía propia dentro del juego, por lo que tendremos que crear un nuevo proceso para ellos: el proceso "disparo".

```

Process disparo ()

```

```

Begin

```

```

    Graph=3;

```

```

    Loop

```

```

        y=y-15;

```

```

        frame;

```

```

    end

```

```

end

```

Puedes ver que a cada frame, la posición vertical decrece en 15 píxeles: en cada frame el disparo va subiendo por la pantalla 15 píxeles, pues. La posición vertical del disparo al inicio la conocemos, siempre será  $y=435$ , ya que nuestra nave nunca se va a mover hacia arriba ni hacia abajo, pero ¿y la coordenada  $x$ ? Depende de la posición de la nave, por lo que usaremos el mismo truco de siempre: los parámetros. Por ahora vamos a dejar esto en suspense, vamos a decirle primero al programa cuándo tiene que crear un disparo.

El disparo se creará siempre que se pulse la tecla de disparo, pero ¿cuál crees que será el mejor sitio para crearlo dentro del código? Como el disparo depende directamente de la nave protagonista, haremos que sea este proceso el que se encargue de crear el disparo. Usaremos para ello la tecla "x" como tecla de disparo.

Por lo tanto, añadiremos una línea más –el último IF- al proceso nave:

```

Process nave ()

```

```

Begin

```

```

    Graph=2;

```

```

    X=320;

```

```

    Y=435;

```

```

    Loop

```

```

        If (key(_left)) x=x-4; end

```

```

        If (key(_right)) x=x+4; end

```

```

        If (x>600) x=600; end

```

```

        If (x<40) x=40; end

```

```

                If (key(_x)) disparo(); end
                Frame;
            end
end

```

De esta forma se creará un disparo antes de cada frame.

Volvamos al proceso “disparo()”. Decíamos que la posición de origen del disparo dependía de la nave...

Si te acuerdas de un capítulo anterior en este manual, (el del tutorial del juego del laberinto), para hacer que las explosiones aparecieran en la misma posición que nuestro personaje, lo que hacíamos era que el proceso personaje recibiera por parámetro los valores de X e Y del personaje (en aquel tutorial se pasaban más variables, como ANGLE, pero ahora no importa). Recuerda que esto se hacía en dos pasos: en la llamada al proceso "explosion()" -que recordemos que se hacía dentro del código de nuestro personaje-, poníamos como valores de sus parámetros no ningún número en concreto, sino los valores que tenían en ese momento la X e Y del personaje: es decir, escribíamos: "*explosion(x,y);*". Y luego, en la cabecera del proceso "explosion()", hacíamos que esos valores pasados fueran asignados precisamente a la X e Y de la explosión: es decir, la cabecera del proceso explosión la escribíamos así: "*process explosion(x,y)*".

Podríamos hacer ahora exactamente lo mismo: nuestro personaje ahora será el proceso "nave" y lo que en el ejemplo anterior era una explosión, ahora sería el proceso "disparo". Pero lo vamos a hacer de otra forma para que veas otras posibilidades (aunque es bastante similar, de hecho)

Tal como tenemos escrito nuestro juego hasta aquí, ahora existe una relación entre ambos procesos: “nave” ha creado a “disparo”, por lo que “nave” es el Father de “disparo” y por tanto “disparo” es Son de “nave”, y dos “disparos” son hermanos, porque tienen el mismo padre. Usaremos este hecho para utilizar las variables locales predefinidas FATHER y SON convenientemente.

¿Sabes a dónde quiero ir a parar? Modifica así el proceso disparo:

```

Process disparo ()
Begin
    Graph=3;
    Y=435;
    X=father.x;
    Loop
        y=y-15;
        frame;
    end
end

```

Ahora le estamos diciendo que se ponga en la misma coordenada X que su padre, (es decir, la nave) utilizando la sintaxis típica para acceder a los valores de las variables locales de procesos: "identificadorProceso.variableLocal". No obstante, hay que tener en cuenta que este sistema sólo lo podremos utilizar tal como está ahora si sabemos que el proceso disparo tiene como único Father posible la nave. Si quisiéramos por ejemplo reutilizar el proceso “disparo()” para que nuestros posibles enemigos pudieran disparar también, entonces la variable FATHER valdría para cada disparo en particular uno de dos valores posibles (proceso “nave()” o proceso “enemigo()”) y entonces la X del disparo cambiaría según de qué padre proviniese...

Si ejecutas ahora el juego tal como está, verás que tu nave dispara por fin, pero quizás no te

guste el detalle de que los disparos aparezcan por encima de la nave y desde el centro. Modificamos pues el proceso "disparo()" de la siguiente manera:

```
Process disparo ()
Begin
    Graph=3;
    Y=410;
    X=father.x;
    Z=1;
    Loop
        y=y-15;
        frame;
    end
end
```

Lo que hemos hecho ha sido modificar la coordenada Y del origen del disparo, poniéndola un poco más arriba, ya que tenemos la suerte de que la nave sólo se moverá horizontalmente, y por tanto, podemos fijar que los disparos aparecerán a partir de Y=410.

Si la nave se pudiera mover también verticalmente, ¿que Y tendríamos que haber puesto? Bueno, al igual que hemos recogido la posición horizontal de la nave para asignársela a la del disparo, igualmente habríamos hecho con la posición vertical, procurando restarle unos determinados píxeles para que el disparo continuara saliendo más arriba que la nave. Es decir:

```
y=father.y - 25;
```

Existiría otra posibilidad diferente y es que la nave pudiera rotar (es decir, pudiera cambiar su ANGLE). En este caso, los disparos tendrían que salir en la misma dirección en la que la nave estuviera encarada en cada momento. Lo primero que tendríamos que escribir en esta nueva situación en el proceso "disparo" (fuera del loop) es:

```
x=father.x;
y=father.y;
angle=father.angle;
```

(o bien, de forma alternativa, pasar estos tres valores como parámetro, tal como se discutió en el tutorial del juego del laberinto en el apartado de las explosiones) y, también muy importante, tendríamos que cambiar la línea de dentro de su LOOP/END que decrementaba la y ( $y=y-15;$ ) y poner en su lugar *advance(15);*, de tal manera que los disparos se muevan en la misma dirección a la que está mirando la nave en ese momento (lógicamente).

No obstante, haciéndolo así volvemos a tener el mismo problema de antes: el disparo aparece por encima de la nave, y ahora no podemos hacer el truco de restar una cantidad a la coordenada vertical porque la nave, si puede rotar, no necesariamente estará mirando "hacia arriba": quedaría fatal que la nave estuviera mirando al noroeste por ejemplo y los disparos salieran 25 píxeles desplazados por encima de la nave, creándose de la nada.

En este caso, el truco estaría en escribir justo antes (fuera) del bucle LOOP/END del proceso disparo, otra línea *advance(15);*. Es decir, que antes de que empiece el bucle se ejecute "de serie" un *advance(15);* Lo que hace esta línea es forzar a que el disparo se mueva antes de empezar el LOOP de su proceso, con lo que cuando empiece el bucle el disparo ya no estará en el centro, sino "delante" del personaje. De esta manera, el primer *advance(15);* que se ejecute de dentro del LOOP/END en la primera

iteración se acumulará al *advance(15)*; de fuera, y así, cuando se llegue al primer frame, en realidad el disparo avanzará de golpe 30 píxeles, con lo que habremos solucionado el problema.

La otra modificación que he hecho ha sido añadir la línea  $Z=1$ ; Fíjate que si no se pone, los disparos se pintarán por encima de la nave, cosa que queda feo.  $Z$  es otra variable local predefinida que indica el nivel de profundidad de un proceso dado en relación al resto. Esto sirve para lo que has visto: pintar unos procesos por encima de otros, bien ordenaditos en la tercera dimensión: a mayor valor de  $Z$ , más lejano se pintará el proceso; a menor valor de  $Z$ , más cercano estará, siendo el rango posible de valores entre -512 y 512.

¿Y por qué los disparos, si no se especifica ningún valor de  $Z$ , aparecen dibujados sobre la nave? Por una razón que has de recordar siempre a partir de ahora: en principio, el último proceso en ser creado será el que se vea por encima de los demás. Es decir, si no se especifica lo contrario, los procesos se ordenan en profundidad en función de su momento de creación (procesos posteriores,  $Z$  más cercana). En nuestro código está claro que cualquier proceso "disparo" se ha creado después del proceso "nave", porque entre otras cosas, si no hay proceso nave creado previamente, los disparos no pueden existir., por lo que, siguiendo esta regla, cualquier proceso disparo aparecerá pintado sobre la nave. Para evitarlo, tendremos que cambiar la profundidad a la que aparece el gráfico del disparo, especificando para este proceso un determinado valor de esta variable local.

¿Y cómo sabemos que tenemos que poner un 1? Porque, otra cosa que tienes que saber, es que todos los procesos creados en el programa, por defecto siempre tienen el valor de  $Z$  igual a 0. Así pues, el proceso "nave" sabemos que tiene una  $Z=0$ , pero ojo, los procesos "disparo" también, si no se le dice lo contrario. El tema está lo que acabo de comentar en el párrafo anterior: aún cuando existan varios procesos con  $Z$  iguales, el proceso de pintado es el siguiente: tendrá mayor profundidad aquel proceso que haya sido creado antes. Es decir, a medida que se van creando procesos, éstos se van "amontonando" encima de los ya existentes (repito, aún teniendo una  $Z$  igual). Por lo tanto, si queremos que los disparos se pinten bajo el proceso nave, tendremos que obligarles explícitamente a que su  $Z$  sea MAYOR que 0, que es el valor que tiene la  $Z$  de "nave" por defecto. Con 1 ya vale. Recuerda: cuanto más negativa es la  $Z$ , más hacia el frente se vienen los gráficos, y a  $Z$  más positivas, más hacia el fondo se pintan.

Es conveniente saber qué valores predeterminados puede valer la  $Z$  según el tipo de gráficos que aparezcan en pantalla. Por si te interesa, lo más al fondo que hay son los scrolls y los gráficos pintados con las funciones de la familia "put" (como "put\_screen"), que por defecto tienen una  $Z=512$ . Después están los planos abatidos Modo7, con una  $Z=256$  por defecto. Le siguen los gráficos de los procesos (ya lo hemos dicho, con  $Z=0$ ), luego los textos (con  $Z=-256$ ) y por último, lo que se pinta por encima de todo, el gráfico del cursor del ratón ( $Z=-512$ ). Según avancemos veremos que estos valores se pueden modificar, como ya hemos hecho con un gráfico de un proceso.

Puedes probar un momentito este ejemplo (saliéndonos momentáneamente de nuestro matamarcianos) para ver mejor lo que te cuento. Necesitarás un FPG llamado "cuadrados.fpg" donde hay tres gráficos con códigos 001,002,003, los cuales son cuadrados de 30x30 de diferentes colores. El programa principal creará tres procesos, cada uno de los cuales será visualizado mediante uno de estos cuadrados, que se podrán mover mediante diferentes teclas del cursor. También mostrará el valor por defecto de  $Z$  para los tres procesos, que será 0. Y ya está. Lo que quiero que veas es que, aún teniendo la misma  $Z$ , el cuadrado correspondiente a "proceso1" estará a menor profundidad que los otros dos (lo puedes comprobar si lo mueves y lo pasas por encima de los otros gráficos); el cuadrado de "proceso2" estará sobre el de "proceso3" pero bajo el de "proceso1"; y el de "proceso3" se pintará bajo los otros dos. Esto es debido al orden de creación de los procesos: primero se ha creado "proceso1", luego "proceso2" y luego "proceso3". Si quieres, puedes cambiar este orden y comprobarás que las profundidades cambian también – a pesar de continuar teniendo los mismos valores de  $Z$  todos los procesos-. Ahora bien, si definieras explícitamente un valor de  $Z$  para estos procesos, sería todo más claro, porque controlarías exactamente a qué profundidad respecto el resto deseas pintar cada proceso.

```

program tutorial;
global
    int fpg;
end
private
    int idproc1,idproc2,idproc3;
end
begin
    set_mode(320,240,16);
    fpg=load_fpg("cuadrados.fpg");
    idproc1=proceso1();
    idproc2=proceso2();
    idproc3=proceso3();
    write_var(0,100,20,4,idproc1.z);
    write_var(0,100,40,4,idproc2.z);
    write_var(0,100,60,4,idproc3.z);
    loop
        if(key(_esc)) exit(); end
        frame;
    end
end

Process proceso1()
begin
    graph=1;
    x = 160;
    y = 120;
    loop
        if(key(_up))y=y-5;end
        if(key(_down))y=y+5; end
        if(key(_left))x=x-5; end
        if(key(_right))x=x+5; end
    frame;
end
end

Process proceso2()
begin
    graph=2;
    x = 60;
    y = 20;
    loop
        if(key(_w))y=y-5;end
        if(key(_s))y=y+5; end
        if(key(_a))x=x-5; end
        if(key(_d))x=x+5; end
    frame;
end
end

Process proceso3()
begin
    graph=3;

```

```

x = 260;
y = 220;
loop
  if(key(_h))y=y-5;end
  if(key(_n))y=y+5; end
  if(key(_b))x=x-5; end
  if(key(_m))x=x+5; end
frame;
end
end

```

Una cosa importante que deberás recordar: el valor de Z no se tiene en cuenta para la detección de colisiones. Es decir, las colisiones se pueden producir entre diferentes procesos aunque éstos estén en Z diferentes. Cuando añadamos enemigos a nuestro juego (dentro de nada) verás que los enemigos estarán a otra Z diferente de los disparos, pero aun así, los disparos colisionarán perfectamente con éstos, ya que, como hemos dicho, la tercera dimensión es irrelevante en la detección de colisiones.

Volvamos al matamarcianos. e habrás puesto a disparar como un loco, has soltado ráfagas hasta que te ha dolido el dedo, y habrás comprobado que a medida que pasa el tiempo y disparas más parece que el juego se ralentiza, puede que tardes diez segundos o cinco minutos, pero llega un momento que la cosa se vuelve injugable ¿Qué está pasando? La respuesta es bien sencilla, el disparo, aunque salga de pantalla aun sigue activo, y sigue avanzando hasta que llegue al infinito, y claro, si el primer disparo sigue ejecutándose tras el 100º disparo significa que el ordenador está moviendo 100 procesos, y llega un momento en que tiene que ejecutar demasiados procesos y no le da tiempo. Es nuestro trabajo evitarlo, tenemos que hacer que el disparo se autodestruya al salir del área visible. Hay dos formas de hacerlo, una es usando la función “out\_of\_region()”, de la que encontrarás más información en un capítulo posterior, y la otra es la que vamos a usar, que es bastante más sencilla, que simplemente es comprobar si está por encima de la posición vertical -15 y salir del bucle, lo haremos añadiendo un IF así:

```

Process disparo ()
Begin
  Graph=3;
  Y=410;
  X=father.x;
  Z=1;
  Loop
    y=y-15;
    if (y< -15) break; end
  frame;
end
end

```

Haz la prueba y verás que ha mejorado notablemente, puedes tirarte horas disparando y la munición no se acabará, y el rendimiento no se verá afectado.

Quizás podamos mejorar ligeramente el rendimiento todavía un poco más. Esto lo haremos usando otro tipo de bucle que ya haga la comprobación, que será más apropiado dado que sabemos cuando va a dejar de ejecutar el bucle, tan sencillo como sustituir el bucle LOOP por REPEAT/UNTIL:

```

Process disparo ()
Begin
  Graph=3;

```

```

Y=410;
X=father.x;
Z=1;
Repeat
    y=y-15;
    frame;
until (y < -15)
end

```

Este bucle es más apropiado ya que, al menos, se ejecutará una vez pero se acabará cuando supere el borde superior de la pantalla. Quizás te preguntes por qué debe superar  $-15$  y no  $0$ , y eso se debe a que las coordenadas del gráfico están en el centro, y de borrarlo en cero aun se vería la mitad del disparo.

### **Añadiendo los enemigos:**

Abre tu editor de imagen preferido y diseña una nave enemiga del mismo tamaño que la protagonista, pero mirando hacia abajo, con código de gráfico número 4.

¿En qué lugar del código crearías los enemigos? Por el momento tenemos tres procesos. En el proceso “disparo()” no es, porque si no disparas no saldría ninguna nave enemiga. El proceso “nave()” no tiene este problema, pero no es el más adecuado, imagínate, el bueno creando el mal. Así que nos quedamos en el proceso principal.

Ahora hay que saber donde meterlo, lo lógico es que esté en el bucle principal, de lo contrario sólo se crearía un enemigo, así que vamos a añadir la línea *enemigo()*; dentro del LOOP del programa principal:

```

Program ejemplo;
Global
    Int Graficos;
End
Begin
    set_mode(640,480,16);
    set_fps(60,1);
    Graficos=load_fpg("prueba.fpg");
    Put_screen(graficos,1);
    Nave();
    Loop
        Enemigo();
        If (key(_esc)) break; end
        Frame;
    End
    Unload_fpg(graficos);
    Let_me_alone();
End

```

Y a más a más, claro, escribiremos después de todo lo que hemos escrito hasta ahora el código de este nuevo proceso:

```

Process enemigo ()
Begin
    Graph=4;
    Y=-40;

```

```

Repeat
    Y=y+5;
    Frame;
Until (y > 520)
end

```

Ves que lo que van a hacer nuestros enemigos será aparecer desde fuera de la pantalla por arriba, y avanzar hacia la parte inferior (5 píxeles) y desaparecer por abajo.

Pero también tendremos que, primero, hacer que se muevan de lado para que sea más difícil acertarles, y, segundo, hacer que cada enemigo aparezca por arriba en una X diferente de los demás.

Vayamos por lo segundo. Ya sabemos que cuando necesitamos crear procesos con unas condiciones que varían según el momento, una solución a la que podemos recurrir es el paso de parámetros, o sea, indicarle al ordenador que cuando cree un proceso le indique de qué forma hacerlo. En nuestro caso, vamos a crear el proceso enemigo indicándole qué posición horizontal –fija y común , de momento- debe tener al ser creado, y lo haremos escribiendo lo siguiente en el proceso principal:

```

Program ejemplo;
Global
    Int Graficos;
End
Begin
    set_mode(640,480,16);
    set_fps(60,1);
    Graficos=load_fpg("prueba.fpg");
    Put_screen(graficos,1);
Nave();
Loop
    Enemigo(320); //Indico un valor concreto del parámetro
    If (key(_esc)) break; end
    Frame;
End
Unload_fpg(graficos);
Let_me_alone();
End

```

Y esto en el proceso enemigo:

```

Process enemigo (x) /*El valor pasado en el programa principal se asigna a la variable local X del proceso
enemigo (por lo que todos los enemigos saldrán en el centro de la pantalla.*/
Begin
    Graph=4;
    Y=-40;
Repeat
    Y=y+5;
    Frame;
Until (y > 520)
end

```

Si compilas y ejecutas ahora, verás que aparecen un montón de naves, todas en fila, bajando por el centro de la pantalla. La verdad es que seguramente no era lo que esperabas, tu querrás que aparezcan en distintas posiciones, pero ¿cómo lo hacemos? Si ponemos un número siempre aparecerán por esa posición.

La solución –una de ellas- está en generar números aleatorios, al azar. Para eso usaremos la conocida función rand, que viene de la palabra inglesa “random” (aleatorio). Modifiquemos pues la llamada al proceso enemigo dentro del programa principal así de manera que en vez de pasarle como parámetro el número fijo 320, sea un número aleatorio, así:

```
Enemigo(rand(0,640));
```

Si haces la prueba obtendrás una auténtica lluvia de enemigos, un auténtico aluvión. Quizás hay demasiados enemigos ¿no? La pantalla está saturada y casi no se ve el espacio. Vamos a arreglarlo de la siguiente manera:

```
Program ejemplo;
Global
    Int Graficos;
End
Begin
    set_mode(640,480,16);
    set_fps(60,1);
    Graficos=load_fpg("prueba.fpg");
    Put_screen(graficos,1);
Nave();
    Loop
        If(rand(0,100)<20)
            Enemigo(rand(0,640));
        End
        If (key(_esc)) break; end
        Frame;
    End
    Unload_fpg(graficos);
    Let_me_alone();
End
```

Así generamos un número entre 0 y 100, y sólo cada vez que este número sea menor que veinte se creará una nave enemiga, es decir, que en cada frame hay un 20% de posibilidades (20 de 100) de que aparezca un nuevo enemigo, de esta manera se creará una nave cada 5 frames como media (como son datos aleatorios, a veces serán más y otras menos). Puedes ejecutar el programa y ajustar el valor a tu gusto.

Ya que le hemos pasado como parámetro desde donde empieza, vamos a aprovechar para pasarle otros parámetros, por ejemplo, cuánto debe avanzar lateralmente y verticalmente (por lo que hay una línea del proceso enemigo que hay que modificar), empecemos por modificar en el proceso principal la llamada al proceso enemigo, pasándole en vez de uno, tres valores como parámetro, así:

```
enemigo(rand(0,640), rand(-5,5),rand(4,7));
```

Evidentemente, si llamas a un proceso con tres parámetros, este debe recibir tres parámetros, y por eso en la cabecera deben figurar tres variables, y no una como tenemos ahora, así que hagamos hay que hacer, antes de hacer nada más, la siguiente modificación en el proceso enemigo:

```
Process enemigo (x,int inc_x,int inc_y)
Begin
    Graph=4;
    Y=-40;
    Repeat
        X=X+inc_x;
        Y=Y+inc_y;
```

```

                Frame;
            Until (y > 520)
end

```

Lo que hemos hecho ha sido asignar dos números aleatorios a dos variables privadas creadas por nosotros (“inc\_x” e “inc\_y”) de manera que así se desplacen de forma distinta. Horizontalmente, -la coordenada x-, se podrán desplazar tanto a izquierda como a derecha, y por eso el número aleatorio puede ser tanto positivo como negativo. Verticalmente, -la coordenada y-, sólo se desplazan hacia abajo, por eso son números positivos mayores de cero. Explicado más detenidamente: antes de pintar la nave la movemos hacia un lado modificando su variable x el número de píxeles que valga “inc\_x” (ya sea a la izquierda, por ser un valor negativo, a la derecha, por ser un número positivo, o no se moverá de lado al valer cero), luego la movemos hacia abajo (lo que valga “inc\_y”, que cuanto mayor sea, más rápido se moverá) y por último la pintamos en la nueva posición, y a repetir otra vez hasta que salga de la pantalla. Si lo ejecutas ahora, la lluvia ya no se desplaza sólo hacia abajo, ya se mueven a distinta velocidad y en diversas direcciones.

Ya que estamos, podemos hacer que las naves sean distintas entre sí, bien metiendo nuevas naves al FPG y usando “rand” en la variable “graph”, o bien haciendo que sean de distinto tamaño. Una ventaja de estos programas con respecto a otros lenguajes es que esto último podemos hacerlo de manera casi automática, gracias a la variable local SIZE -“tamaño” en inglés-. Esta variable, como ya sabemos, por defecto (es decir, si no la cambias), vale 100, e indica el porcentaje del tamaño de la imagen. Si le asignas el valor 50, el gráfico se verá al 50% de su tamaño, es decir, la mitad de grande. Así que escribiremos:

```

Process enemigo (x,int inc_x,int inc_y)
Begin
    Graph=4;
    Size=rand(65,115);
    Y=-40;
    Repeat
        X=X+inc_x;
        Y=Y+inc_y;
        Frame;
    Until (y > 520)
end

```

### **Eliminando procesos, matando enemigos:**

La muerte de los enemigos hay dos formas de tratarla, desde los disparos y desde los propios enemigos, ya que son los procesos que están interactuando. Aunque se podría hacer al revés, lo más lógico es que sea el propio enemigo el que decida si debe morir o no, porque es posible que un disparo no sea suficiente para eliminarlo. En nuestro caso, para no complicarnos mucho, el enemigo morirá al más leve roce de tu munición, así que ya tenemos una nueva condición para acabar con el bucle.

El problema es que cómo sabe la nave si está chocando con un disparo si en pantalla hay como unos 25. Podríamos calcular la distancia a cada uno, pero es que no sabemos los números de identificación de todos ellos, y si los guardamos, a ver de qué manera lo hacemos porque no sabemos cuántos hay ni los podemos almacenar en el mismo sitio. Parece que la cosa está complicada, pero por suerte nuestros lenguajes están ahí para facilitarnos las cosas: la palabra clave es, como podías esperar, *collision*. Recuerda que esta función detecta si el proceso que usa esta función está chocando con el que le pasamos por parámetro, es decir, si algún pixel NO TRANSPARENTE del proceso que usa *collision* se superpone con otro pixel NO TRANSPARENTE del que le decimos, pero ahora tenemos el problema de tener que preguntarle uno a uno por todos los procesos disparo... Pues no, la cosa se soluciona igual de fácilmente, usando “type”, como ya sabrás, y el nombre de un proceso ya le decimos que nos referimos a cualquier proceso de ese “tipo”, en nuestro caso “disparo()”. Cojamos el proceso enemigo y pongámonos a escribir:

```

Process enemigo (x,int inc_x,int inc_y)
Begin
    Graph=4;
    Size=rand(65,115);
    Y=-40;
    Repeat
        X=X+inc_x;
        Y=Y+inc_y;
        Frame;
    Until (y > 520 OR collision(type disparo))
end

```

De esta forma, cuando choque con un disparo, se saldrá del bucle y desaparecerá de pantalla porque *collision* devolverá un valor “true” (verdadero). Date cuenta de un detalle importante, y es que después de “disparo” no hemos puesto los paréntesis, porque si los ponemos le estamos diciendo al ordenador que cree un proceso disparo (y nos pedirá los parámetros). Al poner OR en medio de ambas posibilidades dentro del “Until”, se debe cumplir al menos una de ellas para salir, pues recuerda el funcionamiento de la operación OR; si hubiéramos querido que se saliera cuando ambas fueran verdaderas hubiéramos usado AND.

Otra cosa que quizás no te guste es que los enemigos desaparecen al contacto con el disparo, sin fuego ni sangre ni un mal ¡puf! No te preocupes por ahora, eso lo solucionaremos más adelante, pues antes de ponernos con los detalles “decorativos” debemos preocuparnos del aspecto “funcional”, esto significa, que si no funciona, de nada nos sirve hacer la explosión si después hay que cambiarlo todo. Pero ¿qué falla? Parece que todo va bien. Sí, la cosa va bien, pero no cuadra que si un disparo choca con la nave enemiga, éste siga como si nada.

Se podría poner una sentencia en el proceso disparo que lo finalizara al chocar con el enemigo, pero no suele funcionar: cuando el proceso enemigo choca con el disparo ejecuta su propia sentencia *collision* y por lo tanto desaparece, por lo que cuando el proceso disparo llega a su propia sentencia *collision* ya no está chocando con el enemigo y no desaparece, y viceversa. No hay manera que desaparezcan los dos a la vez. La mejor manera que he encontrado para solucionar esto es usar códigos identificadores

Recuerda que todos los números de identificación de los procesos (las ID, o los números que devuelven al ser llamados) son diferentes de 0. Así que si estamos asignando ese valor a una variable, esta valdrá siempre “true”. *Y aprovecho para decir que esto es aplicable también a los ficheros (FPG) y gráficos (PNG), de manera que sabremos en cualquier momento si se ha cargado o no (si no se carga vale 0, y si se carga valdrá otro número, es decir, “true”).* Y tras este rollo, lo que veníamos a decir, que ya lo dije en el tutorial del juego del laberinto: *collision* no devuelve “verdadero” simplemente, sino la ID del proceso con el que colisiona. Esto no es importante si el parámetro que le hemos pasado es una ID que sabíamos de antemano (el proceso “nave” u otro en concreto), pero en nuestro caso, que puede ser un proceso cualquiera, del tipo que le hemos pedido, del que hay varios “iguales”, es utilísimo, ya que nos dice, sin margen de error, quién es el responsable que está chocando con él. De esta forma, podemos saber cuál es el disparo que debemos eliminar para que no siga adelante. Lo haremos así, asignando la ID que devuelve a una variable, comprobaremos si alguno ha colisionado, y actuaremos en consecuencia “matando” al proceso:

```

Process enemigo (x,int inc_x,int inc_y)
Private
    Int ID_disparo_acertado;
end
Begin
    Graph=4;
    Size=rand(65,115);

```

```

Y=-40;
Repeat
    X=X+inc_x;
    Y=Y+inc_y;
    ID_Disparo_acertado=collision(type disparo);
    If (ID_disparo_acertado !=0)
        Signal(ID_disparo_acertado,s_kill);
    End
    Frame;
Until (y > 520 or ID_disparo_acertado)
end

```

Parece que hemos realizado bastantes cambios, vamos a analizarlos. Hemos creado una nueva variable, (privada, y por tanto no se podrá leer ni modificar desde otro proceso), para almacenar el ID del disparo que choca con nuestra nave, este valor valdrá cero mientras no haya una colisión. Una vez almacenado el dato comprobamos si ha habido colisión, y en caso afirmativo hacemos que desaparezca el disparo, para ello usamos la función conocida función *signal*. Recuerda que esta instrucción permite mandarle "señales" a otros procesos para que reaccionen de una forma u otra, para ello necesitamos dos parámetros: el primero es la ID del proceso al que le vamos a mandar la señal, cosa que no es problema en nuestro caso, y el segundo es la señal que le vamos a mandar (recuerda: S\_KILL, S\_SLEEP,S\_FREEZE, S\_WAKEUP, S\_KILL\_TREE, S\_SLEEP\_TREE,S\_FREEZE\_TREE o S\_WAKEUP\_TREE).

Por último, comprobamos si se cumple cualquiera de las condiciones para que el enemigo desaparezca, bien por salir de pantalla o por haber recibido un impacto directo gracias a tu grandiosa puntería. Si se cumple, el enemigo se autodestruye saliendo del bucle REPEAT/UNTIL. De esta manera nos aseguramos de que desaparecerán tanto el proceso enemigo como el disparo y evitamos el problema que teníamos antes de que los disparos continuaban después de colisionar con el enemigo.

También podríamos haber escrito, de forma alternativa, el código del enemigo así, ahorrándonos de escribir alguna línea :

```

Process enemigo (x,int inc_x,int inc_y)
Private
    Int ID_disparo_acertado;
end
Begin
    Graph=4;
    Size=rand(65,115);
    Y=-40;
    Repeat
        X=X+inc_x;
        Y=Y+inc_y;
        If (ID_Disparo_acertado=collision(type disparo) !=0)
            Signal(ID_disparo_acertado,s_kill);
        End
        Frame;
    Until (y > 520 or ID_disparo_acertado)
end

```

Analicemos la condición con cuidado. Se hace el mismo IF, pero con un cambio: el paréntesis tiene una asignación además de la comparación. El orden de procesamiento siempre es: 1º la asignación y 2º la comparación. Por lo tanto, primero se asigna a "ID\_Disparo\_acertado" el valor de retorno de la función *collision* (que ya sabemos que será 0 si no hay colisión o el ID del disparo concreto si la hay), y

seguidamente, en la misma línea, se compara este valor de "ID\_Disparo\_acertado" para ver si es diferente de 0. Es otra manera de escribir lo mismo, más compacto.

Vamos a ponernos un momento en la piel de un programador. Algo muy importante a la hora de hacer un programa es que este sea "eficiente": esto es, que en el menor tiempo posible se haga la mayor cantidad de trabajo posible, y esto incluye evitar repetir las operaciones o usar sólo la memoria imprescindible. Esto se consigue usando el menor número de variables del tipo más adecuado, usando procesos sin gráficos para ejecutar operaciones que se realizan a menudo (normalmente a estos procesos que "no se ven" se les llama funciones o sub-rutinas), evitando repetir operaciones que se pueden resolver con bucles, y siendo ya muy tiquismiquis, evitando acceder a memoria, sobre todo secundaria (disco duro, disquetes, CD ROM) ya que es lo que consume más tiempo. Del otro lado tenemos el orden del código, si usas una variable para siete cosas distintas al final no sabrás lo que está haciendo en cada momento, por ejemplo, usar "var1" como contador, como variable auxiliar, para ver la energía...: reduce pero con moderación. Por ahora tampoco debe preocuparte demasiado, pero sí debes tener en cuenta que la carga de archivos (imágenes, música...) tarda mucho tiempo y consume memoria, por lo que deberías cargar siempre sólo las imágenes que vayas a usar en cada momento (de aquí la importancia de saber agrupar las imágenes en los ficheros).

Bueno, supongo que mientras has estado leyendo este tostón habrás pensado en nuestro proceso y te habrás dado cuenta de que hay un problema de eficiencia: hacemos la comprobación de si el enemigo ha chocado dos veces (una en el "if" y otra en el "until") y esto hay que remediarlo ¿cómo? Bueno, la segunda comprobación sirve para salir del bucle... ¿no te suena de algo? claro, nuestra instrucción "break", si la metemos dentro de las instrucciones del "if" ya no hace falta comprobarlo de nuevo. Hemos ganado en eficiencia evitando una comprobación (e incluso el posterior "frame" y la comprobación del "until" para salir).

```
Process enemigo (x,int inc_x,int inc_y)
Private
    Int ID_disparo_acertado;
end
Begin
    Graph=4;
    Size=rand(65,115);
    Y=-40;
    Repeat
        X=X+inc_x;
        Y=Y+inc_y;
        ID_Disparo_acertado=collision(type disparo);
        If (disparo_acertado)
            Signal(ID_disparo_acertado,s_kill);
            Break;
        End
    Frame;
    Until (y > 520)
end
```

Considerarás conmigo que no es justo que nuestro protagonista pueda disparar a diestro y siniestro: no deja opción a los enemigos para que nos toquen, y a la larga hace nuestro juego aburrido. Podemos reducir el número de disparos que podemos hacer usando un contador que, mientras no sea cero, no pueda disparar. Pero para controlar el retardo ¿qué valor debemos darle? ¿5? ¿10? En realidad no lo sabemos y sólo podemos hacerlo a ojo, y para hacerlo vamos a crear una constante, de tal manera que

sólo tenemos que cambiarle el valor en esa zona para controlarlo, y así no tenemos que leer todo el código buscando todas las referencias a él (y lo más seguro es que nos equivoquemos cambiando algo). Cojamos el programa principal y añadamos la declaración de una constante llamada “Retardo”:

```

Program ejemplo;
Const
    Int Retardo=10;
End
Global
    Int Graficos;
End
Begin
    set_mode(640,480,16);
    set_fps(60,1);
    Graficos=load_fpg(“prueba.fpg”);
    Put_screen(graficos,1);
    Nave();
    Loop
        If(rand(0,100)<20)
            Enemigo(rand(0,640));
        End
        If (key(_esc)) break; end
        Frame;
    End
    Unload_fpg(graficos);
End

```

Y a continuación vamos a hacer los cambios en el proceso nave, que es el que se encarga de crear los disparos:

```

Process nave ()
Private
    Int cont=0;
End
Begin
    Graph=2;
    X=320;
    Y=435;
    Loop
        Cont=cont-1;
        If(cont<0) cont=0; end
        If (key(_left)) x=x-4; end
        If (key(_right)) x=x+4; end
        If (x>600) x=600; end
        If (x<40) x=40; end
        If (key(_x) AND cont==0 )
            Cont=retardo;
            disparo();
        end
        Frame;
    end
end

```

Como ves, hemos creado la variable “cont” y la hemos iniciado a 0, luego en el bucle la vamos

decrementando, y cuando es negativa la volvemos a poner a cero. De esta manera seguirá a cero hasta que pulses "x", que será cuando se cumplan las dos condiciones del último "if" y se entre en él, allí se generará el disparo y se pondrá "cont" al valor de "retardo", de forma que será necesario repetir el bucle "retardo" veces para volver a disparar. En caso de que no quieras que retardo sea de 10, te vas a la zona de constantes y pones el valor que te de la gana. Recuerda que para hacer una comparación se usa "==" y no "=" ya que lo segundo es una asignación (cambiar el valor de una variable).

Existen muchas otras maneras de evitar que el jugador dispare de forma infinita manteniendo la tecla pulsada y obligarle a tener que pulsar la tecla repetidas veces para disparar. Algunas ideas (trozos de código insertables en un proceso "nave" o similar) que puedes probar en los juegos que puedas crear a partir de ahora podrían ser los siguientes:

```
if (key(_space)) //Se supone que la tecla SPACE es la de disparar
    if (bloqueo==0) disparo(x,y);bloqueo=1;end
else
    bloqueo=0;
end
```

Lo que se hace en este ejemplo es tener una variable con el estado anterior de la tecla, que vale 0 cuando no se pulsó y 1 cuando si se pulsó. Si se da el caso que antes no estaba pulsada (bloqueo=0) y ahora sí (key(\_space)) crea un proceso de disparo, éste se encarga de actualizar la variable a que está pulsada y no la pone como no pulsada hasta que desaparece el disparo.

Este otro ejemplo es similar al anterior, pero fuerza además a que mientras exista un disparo en la pantalla, no se pueda generar otro:

```
if (!exists(type disparo)) // Este es el primero a comprobar... si este cumple nos ahorramos IFs
    if (key(_space))
        if (!bloqueo) disparo (x,y) ; bloqueo=1;end
    else
        bloqueo = 0 ;
    end
end
```

Otra tercera manera, donde sólo puede existir un disparo en cada momento,sería así: hay que poner lo siguiente en el proceso "nave":

```
if (bloqueo==0 and key(_space)) disparo(); end
```

y luego escribir el proceso "disparo" así:

```
process disparo()
begin
    father.bloqueo=1; //Depende cómo hayas declarado bloqueo, aquí está como LOCAL
    repeat
        //desplazas el disparo y eso
    until (y<150)
    father.bloqueo=0;
end
```

En este último ejemplo, si quisieras tener más de un disparo en la pantalla a la vez, tendrás que modificar el proceso "nave" a algo así:

```
if (bloqueo==0 and key(_space)) disparo(); bloqueo=1; end
```

```
if (bloqueo==1 and !key(_space))bloqueo=0; end
```

y en el proceso "disparo" eliminar las líneas que hacen referencia al father (la primera y la última).

### **Añadiendo explosiones:**

Para crear una explosión cuando un enemigo impacte con un disparo, vamos a hacer lo mismo que hicimos en el capítulo del juego del laberinto: una animación, es decir, vamos a crear la ilusión de movimiento y de cambio de imagen a través de imágenes fijas.

Por supuesto, nuestro primer problema es conseguir los gráficos de una explosión para después meterlo en el FPG. Hay varias maneras de hacerlo, la primera es la más obvia, y es que cojas tu editor y te dediques a pintar el fuego expandiéndose y desapareciendo, algo seguramente difícil de hacer de forma realista. También podemos buscar una explosión en las numerosas librerías gratuitas que circulan por la red (o "ripear" los gráficos de otros juegos, cosa que, además de no ser legal, quita originalidad y personalidad a tu juego). Nosotros optaremos por la primera opción.

Crea 15 imágenes, cada una de ellas con una forma levemente diferente de lo que sería la explosión: las primeras imágenes pueden ser puntos rojos pequeños y las últimas pueden ser grandes llamaradas amarillas y azules. Usa tu imaginación, pero ten en cuenta siempre que el color transparente es el negro. Finalmente, añade los gráficos creados al FPG en el orden cronológico de la explosión. En los ejemplos su código irá del 005 al 019.

Bien, ya tenemos la base de la animación, ahora es el momento de ponernos manos a la obra e indicarle al ordenador lo que queremos hacer. Para empezar, esta animación sólo ocurrirá cuando el enemigo muera, por lo que será él el encargado de generarla ¿Crearemos un proceso nuevo "explosion()" como hicimos en el tutorial del laberinto? Pues ahora no: veremos otra solución alternativa. Lo que haremos será cambiar el gráfico de la nave por el del fuego, simplemente. Esto lo haremos dentro del proceso "enemigo()", después de haber comprobado que un disparo ha impactado con él. Y la idea de generar la animación es creando un bucle, en el que irá cambiando la imagen e irá mostrándose una a una a cada "frame"; de esta forma tendremos la animación.

Tenemos que hacer un bucle, y tenemos varias formas de hacerlo, pero las mejores son aquellos bucles que tienen definido un número determinado de repeticiones, pudiendo usar "from" y "for", esto ya es cuestión de gustos. El que quizás sea más complejo sea el bucle "for", pero es el que más posibilidades da, y se usa en muchos lenguajes, por ejemplo en C++, que es de los más usados, y así tendrás un ejemplo práctico de cómo se usa. Crearemos una variable "cont" como contador, y añadiremos todas las líneas que aparecen después del bucle Repeat/Until (y justo antes del end del proceso):

```
Process enemigo (x,int inc_x,int inc_y)
```

```
Private
```

```
    Int ID_disparo_acertado;
```

```
    Int cont;
```

```
end
```

```
Begin
```

```
    Graph=4;
```

```
    Size=rand(65,115);
```

```
    Y=-40;
```

```
Repeat
```

```
    X=X+inc_x;
```

```
    Y=Y+inc_y;
```

```
    ID_Disparo_acertado=collision(type disparo);
```

```

        If (disparo_acertado)
            Signal(ID_disparo_acertado,s_kill);
            Break;
        End
    Frame;
Until (y > 520)
Cont=5;
For(cont=5;cont<=19;cont++)
    Graph=cont;
    Frame;
End
end

```

Poco más hay que explicar, cada frame muestra una imagen distinta, en orden, dando la impresión de animación. Ejecútalo y verás que efecto más fantástico. Haz pruebas con diversas explosiones de distintos tamaños, colores o número de imágenes hasta que des con lo que buscas.

### **Añadiendo energía enemiga:**

Ahora vamos a añadir algo tan interesante como es la energía para los enemigos, muy corriente para crear distintos tipos de enemigos con armaduras potentes, incluidos los pesados de los jefes finales. Nosotros vamos a crear algo muy básico aquí, todos los enemigos tendrán la misma energía, en caso de que quieras crear algo mas complejo ya te daré alguna orientación.

Por lo pronto, nuestros enemigos podrán resistir tres de nuestros ataques (obviamente más si tú lo programas). Para ello necesitaremos una variable “energía”. El tipo de esa variable viene dada por factores externos, es decir, si otro proceso va a consultar su energía no podremos declararla como privada; tampoco global, pues necesitamos una por enemigo y no sabemos a ciencia cierta cuantos va a haber como máximo. Hacerla variable local serviría en caso de que otro proceso dependiera de si un enemigo concreto está muerto, o le queda poca energía,etc, pero recuerda que entonces la variable energía también estará presente en otros procesos como disparo, el principal o incluso la nave protagonista, aunque no la necesiten para nada. Como en nuestro caso la variable “energía” no va a ser consultada para nada por ningún otro proceso, ya que nuestros enemigos aparecen, bajan y mueren sin ningún control, usaremos una variable privada.

Tras declararla la vamos a inicializar a tres, que será su energía. Cómo lo vamos a hacer es sencillo, una vez que impacte la bala, entraremos de nuevo en el “if” que “mataba” el disparo y salía del bucle, pero ahora lo que vamos a escribir es que se reduce la energía una unidad, y en caso de que sea cero, se salga del bucle y muera. Parece sencillo y lo es, sólo hay que ponerse y escribirlo:

```

Process enemigo (x,int inc_x,int inc_y)
Private
    Int ID_disparo_acertado;
    Int cont;
    Int energia=3;
end
Begin
    Graph=4;
    Size=rand(65,115);
    Y=-40;
    Repeat
        X=X+inc_x;
        Y=Y+inc_y;
        ID_Disparo_acertado=collision(type disparo);
        If (disparo_acertado)

```

```

Signal(ID_disparo_acertado,s_kill);
Energia=energia-1;
If (energia==0)
    Break;
End
End
Frame;
Until (y > 520)
Cont=5;
For(cont=5;cont<=19;cont++)
    Graph=cont;
Frame;
End
end

```

### **Añadiendo energía nuestra y su correspondiente barra gráfica. Introducción al uso de regiones:**

No es justo que los enemigos sean más duros que nosotros (aunque ahora mismo nuestra nave sea fantasma y por lo tanto inmortal), así que vamos a apañarnos una barra de energía para nosotros, aunque no vamos a tratar la muerte de nuestra nave. Lo primero que necesitamos es el gráfico de la barra de energía, esta tendrá 200 píxeles de ancho, y el alto que te de la gana (yo propongo 20). Puedes decorarlo como te de la gana, aunque lo suyo es que hagas un degradado de color (que pase de un color a otro de forma progresiva), puedes usar un color vivo, que resalte sobre el escenario y los enemigos. Métela en el FPG con código 006.

También necesitamos declarar la variable de la energía de nuestra nave, "energía\_jugador", así que de nuevo tenemos que plantearnos el tipo que será. Lo más lógico es que sea única, por lo que las posibilidades se reducen a dos: global o privada, ahora depende de cómo vamos a acceder a ella. Podemos controlarla perfectamente desde el proceso nave, o podemos crear un proceso aparte. Lo cierto es que da igual qué método usemos, es cuestión de gustos, no hay una norma fija, aunque sí es recomendable separar procesos que controlan cosas distintas, sobre todo para mantener la claridad a la hora de corregir o mostrar el programa a otra persona; aunque el motivo que nos hace decantar por una solución es otro: un gráfico necesita un proceso. Por eso crearemos uno nuevo, "energía\_nave()" que muestre la energía. Y por tanto, la variable "energía\_jugador", entonces, será global, porque va a ser utilizada por más de un proceso: como mínimo por el proceso "energía\_nave()", el proceso "nave" y -ya lo veremos- por el proceso "enemigo()".

Te estarás preguntando cómo vamos a conseguir que nuestra barra de energía se encoja y estire a voluntad según la energía que tenga nuestra nave. Si eres un poco avisado estarás pensando en cómo hacerlo con la variable SIZE o usando una imagen que la tape. Pero vamos a usar otro sistema, las regiones de pantalla. Las regiones son zonas de un tamaño definido dentro de la pantalla, que pueden tener un comportamiento diferente al resto de ella. Las regiones se estudiarán con más profundidad en el apartado correspondiente del capítulo siguiente, así que si no entiendes algo de lo que se explica aquí, te remito a ese lugar del manual.

Las regiones tienen múltiples utilidades (saber si un gráfico está fuera de una región -con la función OUT\_REGION-, definir ventanas para multijugador a pantalla partida -con la función DEFINE\_REGION,,etc) pero en este juego, la que más utilidad le vamos a dar es que podemos meter un gráfico dentro de una región, de tal manera que si éste se sale de ella, desaparece como si fuera el borde de la pantalla.

Nosotros vamos a jugar con esto último, haremos una región y meteremos dentro el gráfico, y haremos que la región se estreche y se ensanche haciendo que se muestre la porción de energía que le queda a la nave (es decir, que muestre una parte más o menos grande del gráfico). Evidentemente, habrá que encontrar una correspondencia entre la energía y la anchura en píxeles de la pantalla del gráfico.

Empecemos pues con el código. Lo primero es declarar la variable para la energía de nuestra nave. Aunque técnicamente podríamos, no es aconsejable llamarla “energía” porque ya hay una variable privada con ese mismo nombre en el proceso “enemigo”, es la importancia de buscar nombres claros pero bien elegidos, así que definela como “energía\_jugador” e inicialízala a 100 como ya sabes.

Ahora coge nuestro proceso nave y haz una llamada al nuevo proceso justo antes del LOOP. Sólo necesitamos un proceso, lo llamaremos “energía\_nave()”. No olvides que no debes usar letras con tildes y de poner los paréntesis tras la llamada, aunque no pongas parámetros, son necesarios.

Momento de crear el nuevo proceso.

```
Process energia_nave ()
Begin
    X=320;
    Y=30;
    Graph=6;
    Loop
        Frame;
    End
end
```

Poco hay que decir de esto, ya sabes que lo único que hace es mostrar un gráfico en una posición determinada. Ahora tenemos que definir la región, y para ello la función clave es *define\_region*. Definiremos la región 1, en la posición  $320-100=220$  horizontal (mitad de la pantalla menos mitad del gráfico) y  $30-10=20$  vertical (posición vertical del gráfico menos la mitad del alto de este), con un ancho del tamaño “energía\_jugador” y un alto de 20, o sea, el alto del gráfico. Para modificarlo en cada frame, (para actualizar su ancho a lo que valga “energía\_jugador”), es necesario volver a definirla, por eso será lo primero que hagamos al entrar al bucle.

```
Process energia_nave ()
Begin
    X=320;
    Y=30;
    Graph=6;
    Loop
        Define_region(1,220,20,energia_jugador,20);
        Frame;
    End
end
```

Teóricamente, como “energía\_jugador” vale 100, la energía que debería verse es exactamente la mitad (no olvidemos que el gráfico mide 200 de ancho), sin embargo pasan dos cosas: ni la energía es la mitad, ni se ve por encima de los demás procesos. Lo segundo se arregla fácilmente asignando antes del bucle el valor  $-1$  a la variable Z, como vimos en el proceso disparo, salvo que ahora queremos que se muestre por encima.

¿Qué falla en cuanto a lo de la región? Tan sencillo como que no le hemos dicho al ordenador que el proceso sólo debe ser visible dentro de esa región que definimos. Para ello, tenemos que decir explícitamente que tal proceso o tal otro "pertenecen" a una región, y por tanto, sólo serán visibles dentro de ésta. Y eso lo hacemos mediante la variable local REGION del proceso en cuestión. Asignándole el número de la región, el proceso solamente se verá dentro de ésta.

```
Process energia_nave ()
```

```

Begin
    X=320;
    Y=30;
    Z=-1;
    Graph=6;
    Region=1;
    Loop
        Define_region(1,220,20,energia_jugador,20);
        Frame;
    End
end

```

Ahí lo tienes, justo la mitad. Haz la prueba variando el valor de “energía\_jugador”. Por supuesto esto no lo podrás hacer durante el juego, hay que controlarlo según los eventos que sucedan en el juego.

La energía puede bajar por los disparos enemigos o por chocar con los enemigos. Los enemigos no van a disparar porque crear una inteligencia artificial no es algo que se haga fácilmente. Podríamos hacer que aleatoriamente generasen en cada frame un número entre 1 y 50 y que disparen cuando saquen 25 (o un número dentro de un rango) y después lo hagan hacia el jugador (es sencillo si se sabe usar las funciones “get\_angle” y “advance”, puedes intentarlo tras acabar con este tutorial). Nosotros nos centraremos en el choque con nuestra nave.

Dado que los enemigos chocarán con nosotros durante varios frames, haremos que nos quiten un poco de energía en cada uno de ellos. Si controlamos esto desde el proceso de nuestra nave, sólo se nos quitará energía como si chocásemos con una enemiga, aunque sean dos las que lo están haciendo, por eso lo controlaremos desde las naves enemigas, esta es una de las ventajas de haber definido “energía\_jugador” como variable global. A escribir:

```

Process enemigo (x,int inc_x,int inc_y)
Private
    Int ID_disparo_acertado;
    Int cont;
    Int energia=3;
end
Begin
    Graph=4;
    Size=rand(65,115);
    Y=-40;
    Repeat
        X=X+inc_x;
        Y=Y+inc_y;
        ID_Disparo_acertado=collision(type disparo);
        If (disparo_acertado)
            Signal(ID_disparo_acertado,s_kill);
            Energia=energia-1;
            If (energia==0)
                Break;
            End
        End
        If(collision (type nave))
            Energia_jugador=energia_jugador-1;
        End
    Frame;
Until (y > 520)

```

```

    Cont=5;
    For(cont=5;cont<=19;cont++)
        Graph=cont;
        Frame;
    End
end

```

Pero ¡cuidado! Si lo hacemos así llegará un momento en que “energía\_jugador” podría ser negativa, esto hay que controlarlo. El proceso que controla la energía de nuestra nave es el proceso “energía\_nave”, así que añadimos un IF:

```

Process energia_nave ()
Begin
    X=320;
    Y=30;
    Z=-1;
    Graph=6;
    Region=1;
    Loop
        If (energia_jugador<0)
            Energia_jugador=0;
        End
        Define_region(1,220,20,energia_jugador,20);
        Frame;
    End
end

```

Fíjate dónde lo hemos puesto, no es casualidad, *define\_region* no admite números negativos, por eso lo hemos puesto antes, así nos aseguramos que siempre “energía\_jugador” sea positivo.

Pero si siempre pierdes energía, el jugador se sentirá estafado y dejará de jugar por morir tan pronto, así que hay que suministrarle pilas. Podemos hacer que aparezcan procesos que al chocar con el jugador le suministre la ansiada energía. Vamos a ser un poco más originales y haremos que la energía suba sola al matar a un enemigo. Es tan fácil como añadir la línea *energia\_jugador=energia\_jugador+3*; en el proceso “enemigo()” justo antes del *break*; que hay dentro del *If(energia==0)*. Podrías pensar que podríamos haberla escrito tras salir del bucle principal REPEAT/UNTIL, ya que el proceso enemigo está a punto de finalizar, pero piensa que también se aumentaría la energía por desaparecer por debajo de la pantalla.

A más a más, acuérdate de controlar que la energía de la nave no sobrepase el valor 200, porque ésta sigue creciendo y creciendo y la región definida puede salirse de la pantalla y dar un error que te cuelgue el juego, ya sabes, añadimos otro IF:

```

Process energia_nave ()
Begin
    X=320;
    Y=30;
    Z=-1;
    Graph=6;
    Region=1;
    Loop
        If (energia_jugador<0)
            Energia_jugador=0;
        End
        If (energia_jugador>200)

```

```

                Energia_jugador=200;
            End
            Define_region(1,220,20,energia_jugador,20);
            Frame;
        End
    end

```

De momento no hemos hecho nada para cuando nuestra nave llega a energia\_jugador=0. Podríamos hacer que desapareciera de la pantalla y que se mostrara un bonito GAME OVER. Para hacer esto, es fácil de ver que lo que habría que modificar es el bloque *if(energia\_jugador<0)/end* del proceso "energia\_nave()" de la manera siguiente:

```

If(energia_jugador<0)
    energia_jugador=0;
    write(0,320,240,1,"GAME OVER!!!");
    signal(Type nave,s_kill); //Hacemos desaparecer la nave y por tanto ya no podemos jugar
    Break;
End

```

### **Disparo mejorado y disparo más retardado:**

Ahora vamos a añadir un efecto nuevo al juego: vamos a hacer que si la nave protagonista supera un umbral mínimo de energía, la frecuencia de los disparos también disminuya. Es decir, que si tienes poca energía, tus disparos serán más lentos en generarse. Puede que no te guste la idea, darle desventaja al jugador cuando más apoyo necesita, pero es cosa de los juegos, sólo hay recompensa si juegas bien.

La modificación de esto se puede llevar a cabo en varios sitios: en nuestro nuevo proceso "energia\_nave", en el proceso "nave"... pero dado que nuestra nave reaccionará más adelante de otra forma – ya lo veremos-, lo haremos en su proceso. Lo que haremos será alterar el valor de "retardo", aumentándolo, para que tarde más en disparar. A estas alturas es sencillo que lo hagas tú sólo: tienes que comprobar si la energía es menor de cierto valor y asignar el nuevo valor, y si no, restaurar el anterior. Pero vamos a usar un nuevo método: usaremos "switch".

Aunque hay un detalle que se te puede haber pasado por alto: "retardo" no es una variable, es una constante, y por lo tanto no se puede modificar. Bueno, tan sencillo como cambiar esa línea a la zona de variables globales y listo. El proceso nave queda así (hemos añadido el SWITCH):

```

Process nave ()
Private
    Cont=0;
End
Begin
    Graph=2;
    X=320;
    Y=435;
    Energia_nave();
    Loop
        Cont--;
        If (cont<0) cont=0; end
        If (key(_left)) x=x-4; end
        If (key(_right)) x=x+4; end
        If (x>600) x=600; end
        If (x<40) x=40; end
    End
End

```

```

                Switch(energia_jugador)
                    Case 0..30:
                        Retardo=10;
                    End
                    Default:
                        Retardo=5;
                    End
                end
                If (key(_x) and (cont==0))
                Cont=retardo;
                disparo();
            end
            Frame;
        end
    end
end

```

Y el proceso principal así:

```

Program ejemplo;
Global
    Int Retardo=10;
    Int Graficos;
    Int energia_jugador=100;
End
Begin
    set_mode(640,480,16);
    set_fps(60,1);
    Graficos=load_fpg("prueba.fpg");
    Put_screen(graficos,1);
    Nave();
    Loop
        If(rand(0,100)<20)
            Enemigo(rand(0,640));
        End
        If (key(_esc)) break; end
        Frame;
    End
    Unload_fpg(graficos);
    Let_me_alone();
End

```

Si quieres puedes añadir distintos retardos en función de la energía. Procura no repetir valores, de lo contrario, el primer rango que cumpla la condición será el que se ejecute y se ignorará el resto.

Vamos ahora a darle una alegría al jugador, cuando tenga mucha, pero mucha energía lo que vamos a hacer es aumentar la capacidad de disparo, pero de una forma un tanto especial: los enemigos seguirán soportando tres impactos, pero dispararemos doble: saldrán dos disparos cada vez.

Lo primero que tenemos que hacer es coger el proceso “disparo” y hacer un ligero cambio, y es que en vez de recoger el valor de su variable X a partir de la línea  $X=father.x$ ; vamos a conseguir lo mismo pero pasándole ese valor como parámetro, igual que hicimos en el tutorial del laberinto:

```

Process disparo (x)
Begin

```

```

    Graph=3;
    Y=410;
    Z=1;
    Repeat
        y=y-15;
        frame;
    until (y < -15)
end

```

Y además, modificaremos el código del proceso nave de la siguiente manera (hemos añadido el bloque IF(energia\_jugador>195)/ELSE/END):

```

Process nave ()
Private
    Int Cont=0;
End
Begin
    Graph=2;
    X=320;
    Y=435;
    Energia_nave();
    Loop
        Cont--;
        If (cont<0) cont=0; end
        If (key(_left)) x=x-4; end
        If (key(_right)) x=x+4; end
        If (x>600) x=600; end
        If (x<40) x=40; end
        Switch(energia_jugador)
            Case 0..30:
                Retardo=10;
            End
            Default:
                Retardo=5;
            End
        end
        If (key(_x) and (cont==0))
            Cont=retardo;
            If(energia_jugador>195)
                Disparo(x-10);
                Disparo(x+10);
            else
                disparo(x);
            end
        end
    end
    Frame;
end
end

```

Lo que hemos hecho simplemente ha sido poner que si la energía del jugador es mayor de 195, que dispare dos disparos en vez de uno, separados 20 píxeles uno del otro.

## **Disparos de los enemigos:**

¿Qué sería de un juego de matamarcianos si éstos no te pueden disparar? Hagámoslo.

Los disparos de los enemigos se podrían hacer de muchas maneras: que cayeran siempre verticalmente hacia abajo, que persiguieran a nuestro protagonista allí donde se moviera...lo que está claro es que los disparos de los enemigos tendrán que ser otro proceso diferente del de los disparos nuestros: éstos hemos hecho que siempre fueran para arriba, ¿recuerdas? y este comportamiento precisamente no es el que queremos para los disparos enemigos.

Haremos el caso más simple: los disparos de los enemigos simplemente se moverán incrementando su Y. La otra alternativa propuesta, que los disparos fueran teledirigidos hacia nuestra nave, no es excesivamente difícil de hacer utilizando las funciones *get\_angle* y *advance*, por ejemplo. Puedes intentarlo tú solo.

Así pues, creamos un nuevo proceso que representará el disparo enemigo, llamado “edisparo()”:

```
Process edisparo(x,y)
Begin
    graph=3; /*El gráfico puede ser otro. Aquí utilizamos el mismo gráfico que el de los disparos
nuestros*/
    Z=1;
    Repeat
        y=y+5;
        if(collision(TYPE nave))
            energia_jugador=energia_jugador-1;
        end
        frame;
    until(y>480)
FRAME;
END
```

Y en el proceso “enemigo()”, lo creamos. Escribe la siguiente línea en el interior del bloque REPEAT/UNTIL de dicho proceso, justo antes de la orden frame;.

```
if(rand(0,100)<10) edisparo(x,y); end
```

Y ya tendrás enemigos temibles: te podrán quitar energía al chocar contra ti y al acertar con sus disparos. ¡O sea que cuidado!

## **Añadiendo los puntos. Introducción al uso de las fuentes FNT:**

Lo que vamos a hacer ahora es mostrar los puntos que la nave irá sumando cada vez que haga explotar un enemigo (1 punto=1 enemigo muerto), en la parte superior de la pantalla, centrado respecto al ancho. Lo primero es crear una variable “puntos” que contenga el valor de los puntos, declárala de tipo global en el programa principal, y a continuación vamos a poner la condición para que sume un punto: esto último lo haremos añadiendo la línea *puntos=puntos+1*; en el proceso enemigo, como podemos ver a continuación:

```
Process enemigo (x,int inc_x,int inc_y)
Private
    Int Disparo_acertado;
    Int Energia=3;
    Int Cont;
```

End  
Begin

```
Graph=4;
Size=rand(65,115);
Y=Y-40;
Repeat
    X=X+inc_x;
    Y=Y+inc_y;
    Disparo_acertado=collision(type disparo);
    If (disparo_acertado)
        Signal(disparo_acertado,s_kill);
        Energia=energia-1;
        If(energia==0)
            Energia_jugador=energia_jugador+3;
            Puntos=puntos+1;
            Break;
        end
    end
    If (collision(type nave))
        Energia_jugador=energia_jugador-1;
    end
    Frame;
Until (y > 520)
Cont=5;
For(cont=5;cont<=19;cont++)
    Graph=cont;
    Frame;
End
```

end

El escribir el texto lo vamos a hacer desde el proceso principal, aunque no es lo más recomendable si quisiéramos tener el código bien agrupado y ordenado; lo bueno sería crear un proceso nuevo que controlara los textos, pero como sólo vamos a escribir éste no hará falta.

Vamos a hacerlo bonito: vamos a utilizar una fuente FNT para mostrar el texto (el tema de las fuentes FNT también se trata en el capítulo siguiente: remito el lector al apartado correspondiente para obtener más información). Antes que nada, hay que hacer lo mismo que con las imágenes, cargar la fuente, para ello hay distintas funciones. Tenemos "load\_fnt", el parámetro que hay que pasarle es la ruta absoluta o relativa –según convenga– del archivo, igual que las imágenes, y al igual que ellas también esta función devuelve un número identificador. Por ejemplo, tenemos un tipo de letra que hemos llamado "mi\_letra.fnt" y hemos declarado una variable global para almacenar su ID llamada "letra\_id", la línea para cargarlo sería "letra\_id=load\_fnt("mi\_letra.fnt");" siempre y cuando el archivo de fuente este en la misma carpeta desde donde se ejecuta el DCB.

Fénix además admite también la carga del formato TTF mediante la DLL "tff.dll" con el comando "load\_ttf", que en el mismo ejemplo sería "letra\_id=load\_ttf("arial.ttf",8);" donde el segundo número es el tamaño de la letra. Esta fuente por defecto tendría el color blanco, si quieres cambiárselo deberías usar la función *text\_color()*. El uso de DLLs lo veremos en capítulos posteriores.

En este momento podrás entender mejor cuál es el significado del primer parámetro que siempre valía 0 hasta ahora de la función write. En su momento te dije que poner 0 ahí equivalía a utilizar la fuente predeterminada del sistema, la cual no hay que cargar. En realidad, este primer parámetro lo que indica es el identificador de la fuente (FNT o TTF) con la que queremos escribir el texto. Así, si hemos

cargado una fuente con `load_fnt` y esta función nos devuelve un identificador que lo guardamos en una variable “letra\_id”, si escribimos por ejemplo `write(letra_id,100,100,1,”Hola”)`;; aparecerá la palabra “Hola” escrita con la fuente identificada por “letra\_id”, que será la que hemos cargado con `load_fnt`.

En resumen, si queremos imprimir los puntos en pantalla –en este ejemplo usaremos la fuente del sistema: si quieres utilizar otra fuente ya sabes cómo hacerlo-:

*Program ejemplo;*

*Global*

```
Int Retardo=10;  
Int Graficos;  
Int Puntos=0;  
Int energia_jugador=100;
```

*End*

*Begin*

```
set_mode(640,480,16);  
set_fps(60,1);  
Graficos=load_fpg(“prueba.fpg”);  
Put_screen(graficos,1);  
Nave();  
Write_var(0,320,10,1,puntos);  
Loop  
    If(rand(0,100)<20)  
        Enemigo(rand(0,640));  
End  
    If(key(_esc)) break; end  
    Frame;  
End  
Unload_fpg(graficos);  
Let_me_alone();
```

*End*

Fíjate que usamos `write_var` y no `write` porque queremos cada vez que la variable puntos cambie de valor allí donde lo haga automáticamente la impresión de ese valor también cambie.

Supongo que ya te habrás dado cuenta de que nuestro juego no tiene fin. Puedes estar matando enemigos toda la vida. Estaría bien, por ejemplo, que nuestro juego acabara cuando la nave hubiera eliminado un cierto número de enemigos, por ejemplo 100. En ese momento, podría aparecer sobrepresionado algún texto felicitando al jugador, desapareciendo además todo rastro de presencia enemiga, para que nuestra nave pudiera viajar a sus anchas por todo el universo. Vamos a hacerlo. Es fácil ver que lo que tendríamos que hacer es añadir en el proceso “`nave()`”, justo antes de su orden `FRAME`, el siguiente bloque `IF/END`:

```
If (puntos==100)  
    signal(Type enemigo,s_kill);  
    write(0,120,240,0,”¡Muy bien! ¡Has sorteado los asteroides con éxito!. Pulsa ESC para salir”);  
End
```

### **Introducción al uso del ratón:**

Lo que vamos a hacer ahora va a ser comprobar la posición horizontal del ratón y colocar la

nave en su posición, y no sólo eso, sino que vamos a usar los mismos botones del ratón para disparar. Otra vez recomiendo el lector la lectura previa del apartado correspondiente del capítulo siguiente para obtener información más completa y extensa sobre el uso del ratón. No obstante, para conseguir lo que queremos, es tan sencillo como cambiar un par de líneas:

```

Process nave ()
Private
    Int Cont=0;
End
Begin
    Graph=2;
    X=320;
    Y=435;
    Energia_nave();
    Loop
        Cont=cont-1;
        If (cont<0) cont=0; end
        X=mouse.x;
        Switch(energia_jugador)
            Case 0..30:
                Retardo=10;
            End
            Default:
                Retardo=5;
            End
        end
        If (mouse.left and (cont==0))
            Cont=retardo;
            If(energia_jugador>195)
                Disparo(x-10);
                Disparo(x+10);
            else
                disparo(x);
            end
        end
    end
    Frame;
end

```

Fíjate que hemos sustituido cuatro líneas que controlaban las teclas de desplazamiento lateral por una sola, y además el manejo es más sencillo para el jugador. Puedes intentar modificar este proceso para que también se desplace verticalmente, pero no olvides cambiar la posición inicial vertical del proceso “disparo”. También controlas el disparo con el ratón, incluso sigue manteniendo el retardo y únicamente cambiando una comprobación.

Los usos del ratón son tan variados como tu imaginación quiera, ahora es cosa tuya seguir investigando y probando.

### **Introducción al uso de scrolls de fondo:**

Nuestro juego ya es muy, muy completo, no te quejarás: tu nave se mueve, dispara, tiene energía, aparecen enemigos, usamos el ratón. Pero es soso, tiene el fondo ahí, estático, sin vida. Puede que si le has puesto algún planeta tenga algo de vida, y si te has atrevido a que se muevan y todo pues todavía, pero es que permanecer sin moverse en esta pequeña parcela del espacio...Vamos a hacer que el fondo se mueva.,

simulando que nuestra nave está viajando y avanzando sin descanso a través del espacio sideral.

Para ello, vamos a implementar un “scroll automático”. Automático quiere decir que consiste en un scroll que se mueve solo, sin depender de ningún otro proceso (normalmente el proceso protagonista, el cual va avanzando). Los scrolls automáticos suelen usarse para decorados de fondo con movimiento, como es nuestro caso: nuestra nave moviéndose por el espacio. El tema de los scrolls se trata ampliamente en el capítulo siguiente: si se desea profundizar en este tema y entender mejor los conceptos aquí tratados, recomiendo su lectura previa.

Para mostrar un fondo de scroll, lo que tenemos que hacer es modificar el programa principal de la siguiente manera: añadiendo las líneas de *start\_scroll* y *stop\_scroll* y quitando *put\_screen*:

*Program ejemplo;*

*Global*

```
Int Graficos;  
Int energia_jugador=100;  
Int Retardo=5;  
Int Puntos=0;
```

*End*

*Begin*

```
set_mode(640,480,16);  
set_fps(60,1);  
Graficos=load_fpg("prueba.fpg");  
Start_scroll(0,graficos,1,0,0,2);  
Nave();  
Write_var(0,320,10,1,puntos);  
Loop  
    If (rand(0,100)<20)  
        Enemigo(rand(0,640),rand(-5,5),rand(4,7));  
    End  
    If (key(_esc)) break; end  
    Frame;  
  
End  
Stop_scroll(0);  
Unload_fpg(graficos);  
Let_me_alone();
```

*End*

Vemos que creamos un scroll, el número 0, el cual estará formado por la imagen principal 001 del FPG referenciado por la variable “graficos” (es decir, el fondo estrellado), y que no tendrá imagen “de fondo” de scroll (los scrolls pueden tener dos imágenes: una más en primer plano y otra más de fondo, aunque ambas estarán por defecto a una profundidad mayor que todos los procesos, y de hecho, ambas imágenes tienen la misma Z entre sí). Como región definida tiene toda la ventana del juego, por lo que el scroll (si la imagen es lo suficientemente grande, que lo es) se producirá en toda la ventana; y como último parámetro tiene el valor 2, cosa que quiere decir que si el scroll se mueve verticalmente, lo hará de forma indefinida (cíclica).

Es evidente que el scroll ha de acabar cuando el propio proceso principal finalice: por eso se ha escrito el *stop\_scroll* al salir del bucle, inmediatamente antes de la terminación del programa.

Quizás te hayas llevado una desilusión cuando hayas visto que el fondo no se ha movido, y es que, vale, lo has iniciado, (prueba de ello es que el fondo aun está ahí aún habiendo quitado el *put\_screen*), pero no le has dicho que se mueva. Para conseguir ese movimiento hay que modificar la estructura de scroll. En concreto, los campos “x0” e “y0”, las cuales controlan la coordenada x e y del primer plano del

scroll, respectivamente.

Así que lo que nos falta por escribir en el proceso principal es la línea `Scroll[0].y0=scroll[0].y0-2;`, justo antes del frame:

*Program ejemplo;*

*Global*

```
Int Graficos;  
Int energia_jugador=100;  
Int Retardo=5;  
Int Puntos=0;
```

*End*

*Begin*

```
set_mode(640,480,16);  
set_fps(60,1);  
Graficos=load_fpg("prueba.fpg");  
Start_scroll(0,graficos,1,0,0,2);  
Nave();  
Write_var(0,320,10,1,puntos);  
Loop  
    If (rand(0,100)<20)  
        Enemigo(rand(0,640),rand(-5,5),rand(4,7));  
    End  
    If (key(_esc)) break; end  
    Scroll[0].y0=scroll[0].y0-2;  
    Frame;  
End  
Stop_scroll(0);  
Unload_fpg(graficos);  
Let_me_alone();
```

*End*

Lo que hemos hecho con esta nueva línea ha sido simplemente decirle al ordenador que en cada frame, la coordenada y de la imagen “en primer plano” –el 3r parámetro de `start_scroll`- del scroll 0 disminuya en 2 píxeles, con lo que veremos que en apariencia la imagen del cielo estrellado se va moviendo hacia arriba. Y como hemos puesto el valor 2 en el último parámetro de `start_scroll`, nos aseguramos que ese movimiento vertical sea cíclico.

Haz una última prueba: cambia el gráfico de fondo por el mismo que usamos para el scroll, es decir, modifica la función por `start_scroll(0,graficos,1,1,0,2+8)`; (no olvides cambiar el último parámetro para que el fondo también sea cíclico). Ahora salen más estrellas porque estamos viendo dos imágenes del cielo estrellado: la de “primer plano”, que se mueve, y la “del fondo” que acabamos de poner, que es la misma pero no se mueve. Para lograr esto último, ya sabes: usa `scroll[0].y1=scroll[0].y1-1;` y verás que el espacio toma profundidad. Fíjate que el incremento es diferente (en la imagen de “primer plano” cada frame la coordenada y disminuye en 2 píxeles y en la imagen “de fondo” disminuye en 1): esto es para crear un efecto de profundidad.

Puedes jugar con el campo "ratio" de la estructura scroll (`scroll[0].ratio`) si quieres: en ella se guarda la velocidad del fondo respecto al plano principal en porcentaje (así, si vale 100, el fondo se moverá a la misma velocidad que el plano principal, si vale 50 el fondo irá a la mitad de la velocidad, si vale 200 irá al doble de rápido y si vale 33 irá a un tercio, etc. Por lo tanto, en vez de especificar de forma explícita los incrementos de las coordenadas X o Y para la imagen “de fondo”, basta con hacerlo para la imagen “en primer plano” y establecer la variable ratio. Así siempre tendrás que ambas velocidades de scroll están

relacionadas una con la otra: si la del “primer plano” varía, la “del fondo” variará en consecuencia.

**Introducción al uso de bandas sonoras y efectos de sonidos:**

Remito al apartado correspondiente del capítulo siguiente, donde se explica y detalla la funcionalidad y uso de las funciones que Fénix aporta para la inclusión de músicas y efectos de sonido -para los disparos, explosiones, etc- en nuestros juegos (la familia de funciones Load\_song, play\_song, etc y Load\_wav, play\_wav, etc, respectivamente).

## CAPÍTULO 9: FUNCIONES Y ESTRUCTURAS BÁSICAS DEL LENGUAJE

### Trabajar con temporizadores:

Un temporizador es un cronómetro programable accesible en todo momento. Fénix dispone “de serie” de 10 temporizadores accesibles a través de una tabla de variables globales enteras llamadas timers (desde timer[0] a timer[9]).

Estos temporizadores se inicializan automáticamente al comenzar el programa al valor 0 y se incrementan automáticamente 100 veces por segundo. En cualquier momento puede cambiarse el valor de uno de ellos, por ejemplo inicializándolo de nuevo a 0, para controlar una cantidad de tiempo concreta.

Ya sabes que las variables globales se actualizan cada instrucción FRAME. Por tanto, es obvio que los valores leídos a cada frame en un temporizador no serán consecutivos a menos que nuestro programa se esté ejecutando a 100 frames por segundo. Es decir: supongamos que el programa se ejecuta a 40 fps. Eso quiere decir que en un segundo pasarán 40 frames, por lo que el valor del temporizador cambiará 40 veces. No obstante, en un segundo el temporizador aumenta en 100 su valor, por lo que en un segundo el temporizador habrá aumentado 100 unidades pero haciéndolo en 40 saltos.

Una cosa importante que comentar es que para realizar las comprobaciones de los valores que tienen los temporizadores en un momento dado -un `if(timer[x]==limite_tiempo)`- debemos utilizar siempre con los operadores de `>=` o `>` y nunca con el `==` ya que no se puede asegurar que tras la actualización del temporizador el valor sea exactamente igual al deseado.

Vamos a poner un ejemplo. Queremos que un nivel de un juego dure 5 segundos, y se quiere mostrar por pantalla el tiempo que queda. Habrá una variable global llamada “tiempo” que almacenará esos segundos, y un proceso “controlTiempo”, que la modifica. En este ejemplo, cuando se llegue al final de la cuenta atrás, el programa termina su ejecución, pero se podrían inventar otras reacciones. Aquí está el código:

```
program hola;
global
    int tiempo;
end
begin
    controlTiempo();
    loop
        if(key(_esc)) exit(); end
        if(tiempo==0) exit(); end //Es por esta línea que “tiempo” ha de ser global.
        frame;
    end
    let_me_alone();
end

process controlTiempo()
private
//En vez de un variable, se podría haber utilizado una constante para identificar el fin de partida
    int estadojuego=0;
end
begin
    timer[0] = 0;
    repeat
        tiempo = 5 - (timer[0] / 100);
        delete_text(0);
```

```

        write_var(0,100,100,4,tiempo);
        if (tiempo == 0) estadoJuego = 1; end
        frame;
    until (estadoJuego == 1)
end

```

Fíjate lo que hace el proceso “controlTiempo”. Cada vez que se ejecuta, resetea el valor de la variable “timer[0]” (el único temporizador que utilizaremos). Entonces se entra en un bucle repeat/until que lo que hace es decrementar el valor de “tiempo”, hasta que valga 0, momento cuando se sale del bucle y –se supone– se acaba la partida. Lo interesante está en la línea  $tiempo = 100 - (timer[0] / 100)$ ; . Esta línea lo que debería de hacer es decrementar en una unidad el valor de “tiempo” cada segundo. Para controlar que este decremento se haga efectivamente cada segundo recurrimos al timer[0]. Cada segundo hemos dicho que timer[0] (y cualquier timer de los diez que hay, aunque no se utilicen) se incrementa 100 unidades. Por lo tanto, en el primer segundo “tiempo” valdrá  $100 - (100/100)=100-1=99$ ; en el segundo segundo “tiempo” valdrá  $100 - (200/100)=100-2=98$ , etc.

A continuación escribo un pequeño y sencillo programa que registra cuánto tarda el jugador en apretar una tecla. Este código se podría ampliar de forma que se convirtiera por ejemplo en un juego donde el jugador tuviera que apretar dicha tecla dentro de unos márgenes de tiempo establecidos (no pudiendo demorarse o adelantarse demasiado), o incluso guardar los tiempos marcados en un fichero en disco para lograr hacer una lista de récords con todos los jugadores...son ideas para desarrollar.

```

Program ejemplo_timer_para_record;
Global
    int mitiempo=0;
end
Begin
    set_mode(640,480,16);
    record();
End

Process record()
Begin
    write(0,320,240,1,"PULSA ENTER para PARAR EL TIMER");
    Loop
        Repeat
            mitiempo=timer[0];
            write_var(0,320,450,1,timer[0]);
            Frame;
        Until(key(_enter))
        Break;
    End

    delete_text(0);
    write(0,320,100,1,"TU TIEMPO");
    write(0,320,250,1,"pulsa Esc para Salir");
    write_var(0,320,125,1,mitiempo);

    Loop
        If(key(_esc)) exit();End
        Frame;
    End
End

```

Hay que saber, por último, que los timers funcionan de forma independiente de todo lo que demás. Así que si, por lo que sea, el juego se ralentiza, en el ejemplo anterior la partida durará los 100 segundos, pero para el juego habría pasado menos tiempo: supongamos que, en condiciones normales el juego ejecuta 1000 frames en los 100 segundos, en caso de ralentizarse ejecutaría 950 frames en ese tiempo: el protagonista avanza menos, los enemigos avanzan menos, hay menos disparos...Esto es lo que se llama un problema de sincronización.

A partir de las versiones más recientes de Fénix podemos utilizar un nuevo tipo de temporizador, con precisión de milésimas de segundos. Su funcionamiento es muy similar al de los timers clásicos, pero con la diferencia (aparte de la precisión), de que sólo existe un timer de este tipo (en vez de los 10 anteriores) y se accede a él no a partir de elementos de un vector predefinido sino a partir de la función `get_timer()`.

Esta función, la cual no tiene parámetros, sirve para obtener de ella el valor que el temporizador tenga en un instante en particular. Veamos este ejemplo, donde se puede ver el contador en plena acción con precisión de milésimas:

```
program hola;
private
    int tiempo;
end
begin
    loop
        if(key(_esc)) exit(); end
        tiempo=get_timer();
        write_var(0,100,100,4,tiempo);
        frame;
    end
    let_me_alone();
end
```

A partir de aquí, podemos elegir según nuestras necesidades si continuamos haciendo servir los clásicos timers de precisión de centésimas o esta nueva función `get_timer()`.

No obstante, hay que tener en cuenta que la función `get_timer()` NO sirve para asignar un valor concreto al temporizador: eso no lo podemos hacer, a diferencia de los timers clásicos, donde la línea `timer[0]=0;` resetea a cero ese temporizador. Con `get_timer` es incorrecto por ejemplo escribir `get_timer()=0`, ya que no es posible reestablecer a mano un valor de este temporizador.

Entonces, ¿cómo lo hacemos para tener el control sobre un periodo de tiempo dado? Pues el truco está en recoger al principio de ese período en una variable el valor que en ese instante tenga el temporizador, y luego, al final de ese período, volver a recoger el nuevo valor y restarle el del principio. De esta manera, la diferencia que obtenemos es el número de milisegundos que han transcurrido entre un instante y el otro.

Por ejemplo, aquí mostramos el primer ejemplo que vimos en este apartado, pero esta vez utilizando `get_timer()` en vez de `timer[0]`:

```
program hola;
global
    int tiempo;
end
begin
```

```

        controlTiempo();
        loop
            if(key(_esc)) exit(); end
            if(tiempo==0) exit(); end
            frame;
        end
        let_me_alone();
end

process controlTiempo()
private
    int estadojuego=0;
    int comienzo;
end
begin
    comienzo=get_timer(); //Instante inicial
    repeat
        tiempo = 5 - (get_timer()-comienzo)/1000;
        delete_text(0);
        write_var(0,100,100,4,tiempo);
        if (tiempo == 0)
            estadoJuego = 1;
        end
        frame;
    until (estadoJuego == 1)
end

```

Un ejemplo más fácil, para acabar de entender el sistema (necesitarás un gráfico cualquiera llamado "grafproc.png"):

```

//Author: Coldev
program facil;
begin
    set_mode(640,480,16);
    write(0,10,10,0,"Pulsa (1) para esperar 1 segundo      Pulsa (2) para esperar 2 segundos ");
    loop
        if(key(_esc)) exit(); end
        if(key(_1)) ExampleImage(1000); end
        if(key(_2)) ExampleImage(2000); end
        frame;
    end
end

process ExampleImage(int duracion)
private
    int comienzo;
end
begin
    graph=load_png("grafproc.png");
    x=320;y=240;
    comienzo= get_timer();
    /*No hago nada mientras el tiempo transcurrido sea, a partir del comienzo, menor que la duración
    establecida. Cuando ya haya pasado ese tiempo, saldré del while, y al acabar el proceso, su gráfico que se

```

estaba mostrando desaparecerá. Podría haber escrito dentro de este while todo aquello que deseara que se ejecutara sólo en el periodo de tiempo dado por "duración"\*/

```
while (get_timer()) < comienzo+duracion) frame; end
end
```

¿Para qué sirve, en este código, el proceso "retraso"? (Necesitarás para hacerlo funcionar un gráfico cualquiera llamado "grafproc.png").

```
program pepe;
private
    int idproc;
end
begin
    set_mode(640,480,16);
    write(0,10,10,0,"Pulsa 1 para dormir, 2 para despertar, 3 para matar");
    idproc=procesol();
    loop
        if (key(_esc)) exit(); end;
        if (key(_1)) retraso(2000,idproc,s_sleep); end
        if (key(_2)) retraso(2000,idproc,s_wakeup); end
        if (key(_3)) retraso(2000,idproc,s_kill); end
        frame;
    end
end

process procesol()
begin
    graph=load_png ("grafproc.png");
    x=320; y=240;
    loop
        frame;
    end
end

process retraso(int duracion,int idproc,int senal)
PRIVATE
    int time_init;
    int time_end;
end
BEGIN
    time_init = get_timer();
    time_end = time_init + duracion;
    repeat frame; until (get_timer() >= time_end)
    signal(idproc,senal);
END
```

Y otra pregunta más: ¿qué diferencia hay entre los dos códigos siguientes?

```
program hola;
Global
int min=2; //minutos
int seg=4; //segundos
```

```

int nFps=10; //Numero de frames a los que cambiará el valor de seg.
int contador; //contador
End
begin
write_var(0,100,100,4,min);
write(0,108,100,4,".");
write_var(0,120,100,4,seg);
loop
    if(contador==nFps)
        seg--; contador=0;
        if(seg<0) min--; seg=59;end
    end
if(min==0 And seg==0) break;end
contador++;
frame;
end
end

```

Y:

```

program hola;
Global
int min=2; //minutos
int seg=4; //segundos
int tiempo;
End
begin
write_var(0,100,100,4,min);
write(0,108,100,4,".");
write_var(0,120,100,4,seg);
tiempo=get_timer();
loop
if(get_timer())>1000+tiempo)
    seg--;
    if(seg<0) min--; seg=59;end
    tiempo=get_timer();
end
if(min==0 And seg==0) break;end
frame;
end
end

```

Efectivamente, la velocidad de cambio en los “segundos” en el primer código depende del FPS que lleve ese programa (en concreto, el cambio se produce cada 10 frames), y en el segundo código sí que va sincronizado realmente con los segundos reales ya que utiliza un temporizador.

Uno podría pensar que si se establece una velocidad de 10 frames/segundo (con `set_fps(10,0)`), por ejemplo), los dos códigos podrían ser equivalentes, pero esto no es exactamente así, porque Fénix “trata” de ejecutar 10 frames/segundo -en este caso- PERO NO lo garantiza, no tiene suficiente precisión para asegurar que esta velocidad sea así ni constante. Esto depende de la máquina, depende la cantidad de procesos activos, etc, etc. Por eso, el uso de temporizadores siempre es más fiables ya que son independientes de todo esto. Aunque usar timers es un arma de doble filo: se tiene precisión en tiempo real en cualquier máquina, pero si el ordenador es lento, los personajes se moverán más despacio y será más difícil que lleguen a una meta en el tiempo previsto (es como si el tiempo fuera más rápido). Por eso es

recomendable usar los timers si se necesita un reloj o el juego no depende del tiempo, pero es mejor usar variables contadoras si lo que se quiere hacer es un juego contrarreloj.

### **Trabajar con ficheros:**

Fénix incorpora una familia de funciones -las llamadas “de Entrada/Salida”-que nos aportarán muchos más beneficios a la hora de gestionar el trabajo con ficheros (crearlos, escribir en ellos, leerlos...). Estas funciones nos pueden ser útiles, por ejemplo, a la hora de guardar datos de una partida en un fichero, para poderlos recuperar más adelante, y así poder incluir en nuestro juego la funcionalidad de salvar/recargar partidas.

#### **FOPEN(“fichero”,modo)**

Esta función abre un fichero en disco, operación previa imprescindible para permitir realizar lecturas y manipulaciones posteriores sobre el mismo, empleando otras funciones. Es decir, ésta es una función imprescindible, la primera que se ha de escribir para poder trabajar con un fichero determinado.

Una vez abierto, el puntero de lectura/escritura se posiciona siempre automáticamente al comienzo del fichero.

#### **PARÁMETROS:**

STRING FICHERO : Ruta completa del fichero

INT MODO : Modo de apertura. Puede ser uno de los siguientes:

O\_READ Modo de lectura

O\_READWRITE Modo de lectura/escritura

O\_WRITE Modo de escritura/creación

O\_ZREAD Similar a O\_READ pero para ficheros comprimidos escritos con O\_ZWRITE.

O\_ZWRITE Similar a O\_WRITE pero a la vez comprimiendo el fichero.

**VALOR DE RETORNO:** INT : Identificador del fichero abierto

Con el modo O\_READ, podremos leer posteriormente el contenido de este archivo desde el principio (con la función correspondiente)

Con el modo es O\_READWRITE podremos leer y escribir en ese archivo desde el principio (con la función correspondiente), sobrescribiendo en el segundo caso lo que ya hubiera. Si quisiéramos preservar el contenido del archivo y escribir nuevos contenidos al final del fichero, añadiéndose a los que ya hubiera, también usaríamos O\_READWRITE, pero deberíamos mover antes (con la función correspondiente) el puntero al fin de fichero y empezar a escribir desde allí.

En cambio, con el modo O\_WRITE, todo lo que se escriba en el fichero sobrescribirá completamente el contenido anterior principio porque éste se borra totalmente.

Un fichero abierto en modo O\_READ o O\_READWRITE debe existir previamente. En caso contrario, la función devolverá 0. En cambio, un fichero será creado si no existe previamente al ser abierto en modo O\_WRITE.

Y una cosa importante: las rutas especificadas (en este comando y en general, en todos los comandos de Fénix donde se tengan que escribir rutas como valores de parámetros) se pueden escribir utilizando los

separadores de directorios de Windows (\) o de Unix-Linux (/) indistintamente.No obstante, en las rutas absolutas de Windows es necesario indicar una letra de unidad, con lo que se pierde portabilidad. Por tanto se recomienda, en la medida de lo posible, escribir rutas relativas al propio DCB y no rutas absolutas.

### **FWRITE( fichero,variable)**

Esta función guarda datos en formato binario, contenidos previamente en una variable, en un fichero abierto con la función FOPEN . Esta función sólo puede guardar una variable a la vez. Sin embargo, esa variable puede ser una tabla o estructura: en ese caso se guardarán secuencialmente (es decir, por orden uno detrás de otro) todos los elementos de la tabla o estructura.No es válido, en cambio, almacenar datos de tipo POINTER.

#### PARÁMETROS:

INT FICHERO : Identificador de fichero devuelto por FOPEN  
VAR VARIABLE : Nombre de la variable a guardar

VALOR DE RETORNO: INT : Número de bytes escritos, 0 en caso de error

### **FREAD(fichero,variable)**

Esta función lee datos binarios de un fichero abierto con la función FOPEN y los asigna a una variable para poder así trabajar en el código fuente con la información obtenida.

Si por ejemplo la variable es de tipo int (es decir, de 4 bytes), se leerán 4 bytes del archivo. Si es de tipo word se leerán 2 bytes y si es de tipo byte se leerá uno.

Esta función sólo puede leer una variable a la vez. Sin embargo, esa variable puede ser una tabla o estructura: en ese caso se leerán cada vez todos los elementos de la tabla o estructura secuencialmente (es decir, por orden uno detrás de otro).

Una característica importante de su funcionamiento es que cada vez que se le invoque, seguirá leyendo a partir de la última posición donde el puntero se quedó en la última lectura.Es decir, que si se realiza una lectura y se rellena una tabla con dicha operación, la siguiente vez que se llame a FREAD se seguirá leyendo del archivo y se volverá a rellenar la tabla con los nuevos valores.

Como curiosidad, si se deseara procesar ficheros binarios genéricos o escritos con otra utilidad, puede usarse esta función para leer un dato de tipo BYTE cada vez, y procesarlos en secuencia.

#### PARÁMETROS:

INT FICHERO : Identificador de fichero devuelto por FOPEN  
VAR VARIABLE : Nombre de la variable a leer

VALOR DE RETORNO: INT : Número de bytes leídos, 0 en caso de error

### **FEOF(fichero)**

Esta función comprueba si quedan datos por leer de un fichero.

Devuelve 1 si el puntero de lectura/escritura traspasa el final de un fichero abierto con la función FOPEN, o 0 en caso contrario.

Puede usarse en un bucle que emplee FREAD para leer datos de un fichero, hasta que ya no queden más en el mismo.

PARÁMETROS: INT FICHERO : Identificador de fichero devuelto por FOPEN

VALOR DE RETORNO: INT : 1 si se llegó al final del fichero

### **FSEEK(fichero,posicion,desde)**

Esta función cambia la posición del puntero de lectura/escritura en un fichero abierto con la función FOPEN .

La nueva posición viene dada por la combinación de los valores del segundo y tercer parámetro.

El tercer parámetro puede indicar tres puntos concretos, que puede ser el comienzo del fichero (valor 0 ó SEEK\_SET), la posición actual del puntero –cuyo valor se puede obtener con FTELL si se desea- (valor 1 ó SEEK\_CUR) o el final del fichero (valor 2 ó SEEK\_END).

El segundo parámetro es el número de bytes que el puntero se moverá, a partir de la posición base indicada en el tercer parámetro, para realmente llegar al sitio deseado. Su valor será positivo si el movimiento del puntero es hacia adelante dentro del contenido del fichero o negativo si es para atrás. Evidentemente, si partimos de la posición base al comienzo del fichero, el segundo parámetro no podrá ser negativo, y si partimos del final de fichero, no podrá ser positivo.

En todo caso se devolverá la nueva posición real, siempre respecto al origen del fichero. FSEEK no funciona si el archivo está comprimido y por tanto se ha tenido que abrir con O\_ZREAD.

#### PARÁMETROS:

INT FICHERO : Identificador de fichero devuelto por FOPEN

INT POSICIÓN : Nueva posición deseada

INT DESDE : Tipo de posición utilizada. Puede ser:

0 ó SEEK\_SET:A partir del comienzo del fichero

1 ó SEEK\_CUR:A partir de la posición actual

2 ó SEEK\_END:A partir del final del fichero

VALOR DE RETORNO: INT : Nueva posición de lectura/escritura, en número de bytes desde el comienzo del fichero (empezando en 0)

### **FTELL(fichero)**

Esta función devuelve la posición en número de bytes del puntero de lectura/escritura de un fichero abierto con la función FOPEN , siempre respecto al origen del fichero (0 para indicar el comienzo).

Se suele utilizar junto con FSEEK, para situar el puntero en un lugar concreto a partir del lugar donde está actualmente.

PARÁMETROS: INT FICHERO : Identificador de fichero devuelto por FOPEN

VALOR DE RETORNO: INT : Posición de lectura/escritura, en número de bytes desde el comienzo del fichero (empezando en 0)

Un ejemplo:

```
program fafafa;
private
    int idfich;
end
begin
    idfich=fopen("pepito.txt",o_read);
/*Muestro en qué posición -byte- estoy. Como acabo de abrir el archivo, debo de estar al principio,o sea,
en el byte 0*/
    write(0,10,10,4,ftell(idfich));
//Desde el principio del fichero, me muevo un byte para adelante
    fseek(idfich,1,0);
/*Vuelvo a mostrar donde estoy. Como me he movido un byte desde el principio, ha de mostrar un 1*/
    write(0,10,20,4,ftell(idfich));
//Desde donde estoy -a un byte del principio-, me vuelvo a mover un byte para adelante
    fseek(idfich,1,1);
/*Vuelvo a mostrar donde estoy. Como desde donde estaba -que era en el byte 1- me he movido otro byte,
ftell mostrará un 2, desde el principio.*
    write(0,10,30,4,ftell(idfich));
    fclose(idfich);
loop
    frame;
end
end
```

### FLENGTH (fichero)

Esta función devuelve el número tamaño total, en bytes, de un fichero abierto con la función FOPEN.

Esta función no es válida para archivos abiertos con O\_ZREAD o O\_ZWRITE

PARÁMETROS: INT FICHERO : Identificador de fichero devuelto por FOPEN

VALOR DE RETORNO: INT : Número de bytes que ocupa el fichero.

Un ejemplo:

```
program test008;
private
    int fp,long;
end
begin
    set_mode(250,200,16);
    fp = fopen("test008.prg", O_READ);
    if (fp==0)
        write(0,120,80,4,"ERROR al abrir archivo test008.prg");
        write(0,120,90,4,"Presione ESC para salir");
        while(!key(_esc)) frame; end
        exit();
    end
    long = flength(fp);
```

```

fclose(fp);
write(0,120,80,4,"La length de test008.prg es "+ long);
while(!key(_ESC)) frame; end
delete_text(0);
end

```

### FCLOSE(fichero)

Esta función cierra un fichero abierto previamente por FOPEN. Todos los ficheros abiertos deben cerrarse una vez acabadas las operaciones de lectura o escritura deseadas, ya que existe un límite al número de ficheros abiertos simultáneamente, en función del sistema operativo.

PARÁMETROS : INT FICHERO : Identificador de fichero devuelto por FOPEN

Vamos a ver unos ejemplos de uso de las funciones que llevamos hasta ahora. Para crear un archivo, previamente inexistente, y escribir en él el contenido de cierta estructura (por ejemplo), haríamos lo siguiente:

```

program fafafa;
private
    int i;
    int idfich;
    struct mistruct[2]
        int a;
        string b;
    end =1,"hola",2,"que tal",3,"muy bien";
end
begin
    idfich=fopen("pepito.txt",o_write);
    fwrite(idfich,mistruct);
    fclose(idfich);
end

```

En este ejemplo hemos utilizado una tabla de estructuras para reducir el código del ejemplo y remarcar así lo importante sin perdernos en otros detalles, pero lo normal trabajando con archivos es utilizar estructuras simples.

Poniendo O\_READWRITE como valor del segundo parámetro de FOPEN también hubiera funcionado, pero si el archivo existiera previamente –con O\_READWRITE ha de existir- y contuviera ya algunos datos, los del principio del fichero se sobrescribirían con los nuevos datos, pero si éstos no fueran suficientes para sobrescribir los datos antiguos enteramente, en el fichero podrían quedar restos de los datos antiguos.

Verás que cuando ejecutes este programa aparentemente no habrá pasado nada. Pero fíjate que en la misma carpeta donde está el DCB aparecerá el archivo “pepito.txt” con los datos en binario que almacenaba la estructura.

Otro ejemplo útil sería el de añadir contenido nuevo a este archivo existente sin sobrescribir el contenido que ya había. Para eso debemos abrir el archivo en modo O\_READWRITE y lo primero, situarnos justo al final para poder empezar a escribir desde allí. O sea,

```

program fafafa;
private
    int i;
    int idfich;
    struct mistruct[2]
        int a;
        string b;
    end = 1,"ah, si?",2,"pues yo",3,"también";
end
begin
    idfich=fopen("pepito.txt",o_readwrite);
    fseek(idfich,0,2);
    fwrite(idfich,mistruct);
    fclose(idfich);
end

```

Fijate que lo único que hemos cambiado respecto el ejemplo anterior es el modo de apertura del archivo (con O\_WRITE no podremos añadir contenido), y utilizar antes de realizar ninguna acción de escritura, la orden `fseek`. En este caso, `fseek` colocará el puntero en el final (3r parámetro) del archivo "pepito.txt" (1r parámetro), y a partir de ahí se moverá 0 bytes (2º parámetro) –o sea, que se quedará en el final-.

Ahora podríamos hacer un programa que leyera el contenido de este archivo que acabamos de crear y que lo mostrara por pantalla. Pero nos surge un problema. ¿Cuántas estructuras contiene "pepito.txt"? Depende de cuántas veces se hayan escrito datos en él. En el archivo puede haberse escrito un número indeterminado de estructuras. Si ejecutamos una vez el primer ejemplo, y una vez el segundo ejemplo, tendremos en "pepito.txt" seis estructuras simples. Pero nada impide volver a ejecutar el segundo ejemplo otra vez, con lo que tendríamos 9 estructuras para leer. ¿Cómo sabremos que tenemos que leer 9 estructuras, ni más ni menos, de ese archivo? Un manera sería así,

```

program fafafa;
private
    int i=10;
    int idfich;
    struct mistruct
        int a;
        string b;
    end
end
begin
    set_mode(320,240,16);
    idfich=fopen("pepito.txt",o_read);
    //Leo las estructuras que contiene el fichero una a una hasta llegar al final del mismo
    loop
        fread(idfich,mistruct);
        if (feof(idfich)!=false) break; end
        write(0,50,i,4,mistruct.a);
        write(0,100,i,4,mistruct.b);
        i=i+10;
        frame;
    end
    fclose(idfich);
end

```

```

        loop
            frame;
        end
end

```

Fíjate que aquí ya usamos estructuras simples, que es lo habitual. Lo único que hacemos es ir repitiendo la lectura del archivo estructura a estructura –y poniendo sus valores por pantalla- hasta que ya no queden más por leer. En cada lectura se sobrescribe el contenido de la variable “mistruct”, así que ésta contendrá en cada momento los valores de la estructura acabada de leer del fichero. ¿Y cómo sabemos que ya no quedan más estructuras por leer? Cuando llegamos al final del fichero, y esto ocurre cuando la función *feof* devuelve 1 (true).

Aunque a lo mejor se te había ocurrido en vez de realizar el bucle LOOP/END, hacer lo mismo pero con un bucle WHILE/END así:

```

while(feof(idfich)==false)
    fread(idfich,mistruct);
    write(0,50,i,4,mistruct.a);
    write(0,100,i,4,mistruct.b);
    i=i+10;
    frame;
end

```

Pero este código tiene un pequeño defecto: muestra los datos bien pero también muestra un último carácter extraño que sobra. ¿Por qué muestra “de más”? Por que fíjate lo que hemos cambiado. La condición del “traspaso” del fin de fichero se comprueba ANTES de leer la siguiente estructura, de tal manera que cuando el programa lee la última estructura, en la siguiente comprobación *feof* continúa valiendo falso porque aunque se haya llegado al final, no se ha “traspasado” todavía: se está en el umbral. Por eso, en esa comprobación, al continuar *feof* valiendo falso, se ejecuta otra vez el interior del while (el FREAD). Y como en ese momento ya no hay nada que leer porque ya sí que se ha “traspasado” por completo el fin de fichero, el FREAD fracasa totalmente y devuelve resultados totalmente erráticos. Por eso, te doy un consejo: comprueba si se ha “traspasado” el fin de fichero justo DESPUÉS de leer con FREAD: de esta manera, si has te has pasado del final, lo sabrás de inmediato y el programa podrá reaccionar a tiempo.

Hasta ahora, los ejemplos vistos sólo son capaces de leer o escribir una única estructura en un archivo. Pero si queremos almacenar más de una estructura, ¿tendremos que crear un archivo diferente para cada una de ellas? No. El siguiente ejemplo (original de SplinterGU) muestra cómo, con un poquito de maña, es posible almacenar en un único archivo el número de estructuras diferentes que se necesite (en el ejemplo, dos), utilizando para ello gran parte de las funciones explicadas hasta ahora, como *fwrite*, *fread*, *ftell* o *fseek*.

La idea del programa es básicamente, a partir de tener un par de estructuras rellenas de valores, guardar éstos en un archivo, seguidamente borrarlos de las estructuras en memoria (y ver que eso es así) y finalmente recuperarlos del archivo donde se guardaron al principio. Los comentarios del código son muy completos, léelos:

```

program test007;
type struct1
    string strng;
    int intgr;
end
type struct2

```

```

        int var1;
        float var2;
end
private
    int fp;
    int i;
    struct1 st1[6] = "s10",10,"s5",5,"s210",210,"s3",3,"s33",33,"s12",12,"s3",3;
    struct2 st2[6] = 10,10.10,5,5.5,210,210.210,3,3.3,33,33.33,12,12.12,3,3.3;
    /*Estructura que nos servirá para almacenar, para cada estructura a guardar en el archivo, a qué
    distancia en bytes se encuentra del comienzo de éste. La idea es que cada estructura se almacenará
    completa una seguida de la otra, de tal manera que luego al leer el archivo, se necesitará saber en qué
    posición dentro de él comienza la estructura que se desea recuperar. Así pues, header.oStruct1 nos
    indicará la posición de comienzo de la primera estructura y header.oStruct2 lo mismo con la segunda. Si
    deseáramos escribir más estructuras dentro del archivo, deberíamos de incluir un campo más por cada
    una de ellas dentro de esta estructura header*/
    struct header
        int oStruct1 = 0;
        int oStruct2 = 0;
    end
end
begin
//Abro el fichero para escribir en él el contenido de dos estructuras
fp = fopen("test007.txt", O_WRITE);
if (!fp) //Si hay algún error...
    write(0,10,180,0,"ERROR al crear archivo test007.txt");
    write(0,30,190,0,"presione ESC para salir");
    while(!key(_esc)) frame; end //Hasta que no se pulse ESC no pasa nada
    while(key(_esc)) frame; end //Mientras se está pulsando, tampoco
    exit();
end
/*La primera estructura que se va a escribir al comienzo del fichero siempre será la estructura header, y
luego, se empezarán a escribir las demás. Pero los valores de los campos de la estructura header
son la distancia en bytes de las demás estructuras respecto al origen del fichero, y esta información no se
sabe hasta que se hayan escrito dichas estructuras en el fichero. Por lo que la línea siguiente lo
que hace es saltar "el cabezal" dentro del fichero un número de bytes igual al tamaño de header para
empezar a escribir más adelante, y así dejar reservado ese espacio del principio para que finalmente se
pueda llenar de contenido*/
fseek(fp, sizeof(header), SEEK_SET);
/*La primera estructura se escribirá justo después de que acaben los datos de la estructura header, por lo
que al haber saltado con la línea anterior hasta este punto precisamente, ya sabemos en qué posición
comenzarán los datos de str1: simplemente tenemos que usar ftell para que nos devuelva la posición
del cabezal, y este valor lo almacenaremos en el campo correspondiente dentro de header*/
header.oStruct1 = ftell(fp);
//Escribimos de un tirón el contenido de str1
fwrite(fp, st1);
/*Una vez acabados de escribir todos los datos de str1, hemos llegado al punto del archivo donde se tienen
que empezar a escribir los datos de str2. Pero antes, almacenamos en header la posición inicial de str2*/
header.oStruct2 = ftell(fp);
//Escribimos de un tirón el contenido de str2
fwrite(fp, st2);
//Situamos el cabezal al principio del fichero otra vez
fseek(fp, 0, SEEK_SET);

```

```

/*Ahora que ya sabemos las posiciones iniciales de str1 y str2 (header.oStruct1 y header.oStruct2),
podemos rellenar de valor el espacio inicial del fichero que dejamos vacío al principio. Y ¿para qué
queremos almacenar aquí las posiciones iniciales de las diferentes estructuras en el fichero? Para que en
la lectura posterior del archivo, a partir de esta información del principio, se pueda distinguir cada dato a
qué estructura pertenece.*/
fwrite(fp, header);
fclose(fp);

//Muestro por pantalla los datos almacenados en un único archivo de dos estructuras
write(0,0,0,0,"Datos almacenados");
/*Para saber el número de registros que existen en un determinado momento del TDU st1 (y así realizar
las iteraciones pertinentes) el truco está en dividir el tamaño total en bytes del TDU entero -sizeof(st1)-
entre el tamaño en bytes de un registro, típicamente el primero -sizeof(st1[0])-. Esto dará el número de
registros que tiene esa estructura en ese momento*/
for(i=0;i<sizeof(st1)/sizeof(st1[0]);i++)
    write(0,0,10+i*10,0,st1[i].strng+" , "+st1[i].intgr);
end
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
    write(0,160,10+i*10,0,st2[i].var1+" , "+st2[i].var2);
end

//Borro los datos de las dos estructuras en memoria
write(0,10,190,0,"presione ESC para limpiar los datos");
while(!key(_esc)) frame; end
while(key(_esc)) frame; end
for(i=0;i<sizeof(st1)/sizeof(st1[0]);i++)
    st1[i].strng = "";
    st1[i].intgr = 0;
end
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
    st2[i].var1 = 0;
    st2[i].var2 = 0.0;
end

//Muestro por pantalla que efectivamente estos datos han sido borrados (de la memoria)
delete_text(0);
write(0,0,0,0,"Datos borrados");
for(i=0;i<sizeof(st1)/sizeof(st1[0]);i++)
    write(0,0,10+i*10,0,st1[i].strng+" , "+st1[i].intgr);
end
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
    write(0,160,10+i*10,0,st2[i].var1+" , "+st2[i].var2);
end

/*Recupero esos datos perdidos de las dos estructuras a partir de la copia guardada en fichero al
principio del programa*/
write(0,10,190,0,"presione ESC para recuperar los datos");
while(!key(_esc)) frame; end
while(key(_esc)) frame; end
fp = fopen("test007.txt", O_READ);
if (!fp)
    write(0,10,180,0,"ERROR al abrir archivo test007.txt");
    write(0,30,190,0,"presione ESC para salir");
    while(!key(_esc)) frame; end

```

```

        while(key(_esc)) frame; end
        exit();
end
/*Lo primero que hago, una vez abierto el fichero, es mirar en qué posiciones dentro de él comienzan cada
una de las dos estructuras guardadas, para poderlas leer sin problemas. Str1 comenzará donde marque
header.oStruct1 y Str2 comenzará donde marque header.oStruct2*/
fread(fp, header);
//Me sitúo al comienzo de la primera estructura y la leo toda completa
fseek(fp, header.oStruct1, SEEK_SET);
fread(fp, st1);
//Me sitúo al comienzo de la segunda estructura y la leo toda completa
fseek(fp, header.oStruct2, SEEK_SET);
fread(fp, st2);
fclose(fp);

/*Muestro otra vez por pantalla los datos contenidos en las dos estructuras, recuperados a partir del
fichero. Se puede comprobar que son los mismos valores que los mostrados al principio del programa.*/
delete_text(0);
write(0,0,0,0,"Datos recuperados");
for(i=0;i<sizeof(st1)/sizeof(st1[0]);i++)
    write(0,0,10+i*10,0,st1[i].strng+" , "+st1[i].intgr);
end
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
    write(0,160,10+i*10,0,st2[i].var1+" , "+st2[i].var2);
end

write(0,0,190,0,"Presione ESC para salir");
while(!key(_ESC)) frame; end
while(key(_ESC)) frame; end
delete_text(0);
end

```

Dos funciones también muy importantes son FPUTS y FGETS, para trabajar con archivos de texto:

#### **FPUTS(fichero,"texto")**

Esta función escribe una cadena en un fichero de texto abierto en modo de escritura. Sirve para crear ficheros de texto, al contrario que FWRITE, que puede usarse para crear ficheros binarios.

Cada cadena se guarda en el fichero en una línea aparte. Si la cadena contiene un salto de línea o más, se guardará el carácter \ al principio de cada uno (lo cual permite volver a leerla empleando en una sola llamada).

#### PARÁMETROS:

INT FICHERO : Identificador de un fichero abierto con FOPEN  
 STRING TEXTO : Texto a escribir

#### **FGETS(fichero)**

Esta función lee la siguiente línea de un fichero abierto en modo de lectura, devolviéndola en una cadena, sin incluir el carácter o caracteres de final de línea.

Es decir, cada vez que se llame a esta función, leerá (únicamente) la siguiente línea de texto del fichero, por lo que escrita dentro de un bucle podría utilizarse para leer un archivo desde el principio hasta el final. La manera de detectar que se ha llegado al final del archivo es la misma que antes: con FEOF.

FGETS sirve para procesar ficheros de texto, al contrario que FREAD, que puede usarse para leer ficheros binarios.

Una línea que acaba con el carácter \ en el fichero original será juntada con la siguiente, sin incluir dicho carácter, pero incluyendo un carácter de salto de línea en su posición. Esto permite almacenar y recuperar cadenas arbitrarias en disco, incluso aunque contengan saltos de línea.

PARÁMETROS: INT FICHERO : Identificador de un fichero abierto con FOPEN

VALOR DE RETORNO: STRING : Texto de la siguiente línea

Miremos un par de ejemplos de estas dos últimas funciones. Si quisiéramos crear un archivo de texto y no binario (es decir, que no contenga datos numéricos ni tablas ni estructuras), podríamos hacer algo muy parecido a esto:

```
program fafafa;
private
    int idfich;
end
begin
    idfich=fopen("pepito.txt",o_write);
    fputs(idfich,"hola, ¿qué tal?. Muy bien.");
    fputs(idfich,"¿Ah,sí? Pues yo también");
    fclose(idfich);
end
```

De igual manera que en ejemplos anteriores, si quisiéramos añadir nuevo texto a un archivo que ya tuviera, tendríamos que cambiar el modo de apertura a O\_READWRITE y utilizar la orden *Fseek*.

Para leer un archivo de texto, lo que tendríamos que hacer algo parecido a lo siguiente:

```
program fafafa;
private
    int idfich;
    string cadenatotal="";
    string cadenaleida;
end
begin
    idfich=fopen("pepito.txt",o_read);
    loop
        if(feof(idfich)!=false) break; end
        cadenaleida=fgets(idfich);
        cadenatotal=cadenatotal+cadenaleida;
    end
    fclose(idfich);
    write(0,160,50,4,cadenatotal);
    loop
        frame;
    end
```

end

Fíjate cómo vamos concatenando en cada iteración la cadena recientemente leída a la cadena ya creada, actualizando de esta manera la propia cadena "total". Éste es el método típico para ir añadiendo porciones de texto a una cadena que por lo tanto va aumentando de longitud.

Otro ejemplo de *fputs* y *fgets* similar al anterior es el siguiente:

```
program test009;
global
  string linea[] = "ésta es la primer línea...",
                  "ésta es la segunda...",
                  "ésta parece ser la tercera...",
                  "creo que ya vamos por la cuarta...",
                  "ésta es la última";
  int i,fp;
end
begin
  fp = fopen("test009.txt", O_WRITE);
  /*Recordemos que si sizeof(linea) devuelve el tamaño total de la tabla y sizeof(linea[0]) devuelve el
  tamaño de un elemento de esa tabla -en este caso el primero, pero puede ser cualquiera ya que todos son
  iguales-, la división sizeof(linea)/sizeof(linea[0]) nos devolverá el número de elementos llenos que tiene la
  tabla en el momento actual.*/
  /*Así pues, este bucle recorre cada elemento de la tabla "linea", que resulta ser una cadena, la cual irá
  siendo guardada en un archivo de texto*/
  for(i=0;i<sizeof(linea)/sizeof(linea[0]);i++)
    fputs(fp,linea[i]);
  end
  fclose(fp);
  write(0,0,0,0,"Recuperando usando fgets");
  fp = fopen("test009.txt", O_READ);
  i=0;
  while(!feof(fp))
  //Recupero en cada iteración una nueva línea
    write(0,0,20+i*10,0,fgets(fp));
    i++;
  end
  fclose(fp);
  while(!key(_esc)) frame; end
end
```

Otras funciones interesantes referentes al manejo de ficheros y directorios son las siguientes:

### **FILE\_EXISTS("fichero")**

Esta función comprueba si un fichero existe o no en la ruta especificada.

Devuelve 1 si el fichero existe y 0 si no existe. El fichero se busca en la ruta actual si no se escribe ningún valor al parámetro o en la ruta que se haya especificado.

PARÁMETROS: STRING FILE : Ruta completa del fichero

## FILE (“fichero”)

Esta función abre el fichero de la ruta especificada, devuelve su contenido en una única cadena y cierra el fichero. Por lo tanto, para su correcto funcionamiento es aconsejable que el fichero sea de texto.

PARAMETROS: STRING FILE: Ruta completa del fichero (ha de existir previamente)

VALOR DEVUELTO: STRING : Cadena cuyo valor será todo el contenido del fichero.

Un ejemplo:

```
program fafafa;
private
  int i;
          string s;
end
begin
  set_mode(320,240,16);
  if(file_exists("pepito.txt")) s=file("pepito.txt"); end
  write_var(0,259,20,4,s);
    loop
      frame;
    end
end
```

No obstante, puedes encontrarte con que si el contenido del fichero incluye saltos de línea y retornos de carro -es decir, si has pulsado ENTER mientras lo escribías-, lo que se visualizará por pantalla, en vez de éstos, serán unos caracteres extraños (en realidad, los caracteres correspondientes al salto de línea y retorno de carro en la tabla ASCII -cuyos códigos numéricos son el 13 y el 10, respectivamente- ). Si quieres que por pantalla se vean los saltos de línea/retorno de carro igual que como están en el archivo, hay que trabajar un poco más. El siguiente código soluciona este problema, pero hace uso de funciones no vistas todavía de tratamiento de cadenas (*find*, *chr*, *substr*), que serán vistas -y explicadas en profundidad- en el apartado correspondiente, más adelante en este mismo capítulo:

```
program test006;
begin
  set_mode(320,240,16);
  readFileWithCRLF("juego.txt");
  while(!key(_esc)) frame; end
end

function readFileWithCRLF(string ruta)
private
  string sVar = "";
  int i,fin,fila,aux,long;
end
begin
  sVar = file(ruta);
  fila = 0;
  long = len(sVar);
  //Voy carácter a carácter
  for (i=0;i<long;i++)
    //Busco el caracter de salto de línea
    fin = find(sVar,chr(13),i);
    //Si no lo encuentro...
```

```

if (fin!=-1)
    write(0,0,filas*10,0,substr(sVar,i,fin-i));
//...y si sí
else
    write(0,0,filas*10,0,substr(sVar,i));
    break;
end

while(fin<long)
    aux=asc(substr(sVar,fin,1));
    if (aux==10 || aux==13) fin++; if (aux==13) filas++; end else break; end
end
i=fin;
end
end

```

En estos momentos es posible que te sea difícil comprender el interior de la función readfileWithCRLF (“CR” quiere decir “Carriage Return” -es decir, retorno de carro, carácter ASCII 10- y “LF” quiere decir “Line Feed” -es decir, salto de línea, carácter ASCII 13-. Lo más interesante, de todas maneras, es poder aprovechar el código tal cual de la función en cualquier otro proyecto que tengas entre manos, y aprovechar así la funcionalidad que aporta.

### CD()

Esta función devuelve el nombre del directorio actual (el directorio donde reside el DCB). Este será el directorio donde actuarán por defecto todas las operaciones con ficheros como pueden ser FOPEN o SAVE (es decir, en el caso que no se especifiquen las rutas completas de los archivos como parámetro de dichas funciones y sólo se especifique el nombre).

VALOR DE RETORNO: STRING: Nombre del directorio actual

### CHDIR(“directorio”)

Esta función cambia el directorio actual por defecto al que se especifica con el parámetro “directorio”. Devuelve 0 si ha tenido éxito y -1 en caso de no poderse efectuar el cambio al nuevo directorio, por ejemplo, si el directorio indicado no existe.

PARÁMETROS: STRING DIRECTORIO : Nuevo directorio

### MKDIR(“directorio”)

Esta función crea un nuevo directorio, indicado por el parámetro “directorio”, en el directorio actual.

Devuelve 0 si ha tenido éxito y -1 en caso de no poderse efectuar la creación del nuevo directorio, por ejemplo si el directorio ya existiese.

También se puede especificar la ruta completa en el parámetro y el directorio será creado utilizando la ruta completa en lugar de crearse a partir del directorio actual. Las rutas pueden indicarse usando el separador de directorios / o \ indiferentemente.

Un directorio no podrá crearse si existe un fichero, recurso o directorio con el mismo nombre o si la ruta especificada es incorrecta.

En los sistemas que así lo permitan, no se podrán crear directorios en la ruta indicada si no se tienen los permisos adecuados.

PARÁMETROS: STRING DIRECTORIO : Nuevo directorio

### **RMDIR("directorio")**

Esta función borra el directorio indicado por el parámetro "directorio" en el directorio actual.

Devuelve 0 si ha tenido éxito y -1 en caso de no poderse efectuar la creación del nuevo directorio, por ejemplo si el directorio ya existiese.

También se puede especificar la ruta completa en el parámetro y el directorio será creado utilizando la ruta completa en lugar de crearse a partir del directorio actual. Las rutas pueden indicarse usando el separador de directorios / o \ indiferentemente.

Un directorio no podrá ser borrado si es el directorio actual, si no está vacío o si es el directorio raíz del sistema.

En plataformas que así lo permitan, un directorio no se podrá borrar si no se tienen los permisos adecuados para ello.

PARÁMETROS: STRING DIRECTORIO : Directorio a borrar

### **RM("fichero")**

Esta función borra el fichero indicado por el parámetro "fichero".

Devuelve 0 si ha tenido éxito y -1 en caso de no poderse efectuar el borrado.

También se puede especificar la ruta completa en el parámetro y el fichero será borrado utilizando la ruta completa en lugar de borrarse del directorio actual. Las rutas pueden indicarse usando el separador de directorios / o \ indiferentemente.

En plataformas que así lo permitan, un fichero no se podrá borrar si no se tienen los permisos adecuados para ello.

PARÁMETROS: STRING FICHERO : Nombre (o ruta) del fichero a borrar

Un ejemplo de estas últimas funciones podría ser:

```
program hola;  
private  
  string a;  
end  
begin  
  mkdir("./midir");  
  chdir("./midir");  
  a=cd();
```

```

write(0,100,100,4,a);
chdir("../");
rmdir("./midir");
loop
    frame;
end
end

```

### GLOB("patrón")

Busca cualquier fichero en el directorio actual cuyo nombre coincida con el patrón indicado.

El patrón es un nombre de fichero en el cual el comodín "?" indica "cualquier carácter" y el comodín "\*" indica "uno o más caracteres cualesquiera".

El uso más habitual es "\*.\*" para buscar todos los ficheros presentes. También es posible indicar un directorio relativo (por ejemplo,"FPG/\*.") además del patrón.

Esta función devuelve el nombre del primer fichero que encuentra y, en sucesivas llamadas con el mismo patrón, va devolviendo el resto de ficheros que coincidan con el patrón. Cuando ya no queden más ficheros, o tras la primera llamada si no se encontró ninguno, devuelve una cadena en blanco. El nombre de fichero que va devolviendo esta función no contiene el directorio.

Esta función además rellena, cada vez que es llamada, la estructura global **FILEINFO**, que podemos utilizar para obtener la siguiente información:

**FILEINFO.PATH:** Directorio donde se encuentra el fichero

**FILEINFO.NAME:** Nombre completo del fichero

**FILEINFO.DIRECTORY:** Contiene TRUE si el fichero es un directorio, o FALSE si es un fichero corriente.

**FILEINFO.HIDDEN:** Contiene TRUE si el fichero está oculto.

**FILEINFO.READONLY:** Contiene TRUE si el fichero no tiene activados permisos de escritura.

**FILEINFO.SIZE:** Tamaño en bytes del fichero

**FILEINFO.CREATED:** Una cadena de texto que contiene la fecha y hora de creación del fichero.

**FILEINFO.MODIFIED:** Una cadena de texto que contiene la fecha y hora del último acceso al fichero.

PARÁMETROS: STRING PATRÓN : Patrón de búsqueda

VALOR DE RETORNO: STRING : Nombre de siguiente fichero o "" si no quedan más.

Como ejemplo de esta última función tienes el siguiente código:

```

program Test_GLOB;

```

```

Private

```

```

    String archivo; //cadena donde se guarda el nombre del archivo

```

```

    Int cont=0;

```

```

Begin

```

```

    chdir("../"); //seleccionamos una carpeta, que es la superior a la actual

```

```

    archivo=glob("*.???"); /*buscamos un archivo cuyo nombre respete el patrón indicado*/

```

```

    While (archivo!="") //comprobamos que ha encontrado algun archivo

```

```

        Write(0,10,cont*10,0, archivo); /*escribimos el nombre del archivo...*/

```

```

        Write(0,250,cont*10,0,FILEINFO.CREATED);/*y su fecha y hora de creación*/

```

```

        cont=cont+1; //avanzamos una linea
        archivo=glob("*.???"); //buscamos otro archivo

    End
    //y esperamos que se pulse escape para salir
    Loop
        If (Key(_ESC)) break; end
        Frame;
    End
End

```

Acuérdate de que tanto en Windows como en Linux, la carpeta “..” significa la carpeta inmediatamente superior a la actual, y la carpeta “.” significa la carpeta actual.

Y, evidentemente, para obtener una información más completa y técnica sobre estas funciones y todas las que no se han citado aquí, no olvides dirigirte a la ayuda de referencia oficial de Fénix.

**Cómo grabar partidas y reproducirlas posteriormente utilizando almacenamiento de datos en archivo**  
 (extraído del tutorial de Pescado, <http://www.bertesh.tk>)

Antes de empezar con este tutorial, donde se pondrá en práctica el uso de archivos para almacenamiento de datos de un programa (en este caso, las diferentes posiciones sincronizadas con el fps que tienen dos procesos cuando se mueven) y su posterior recuperación, he de comentar que para conseguir dicho objetivo no se utilizarán ninguna de las funciones de Entrada/Salida comentadas hasta ahora, tales como fopen, fwrite o fread, sino dos nuevas funciones equivalentes: Save y Load.

<b>SAVE(“fichero”, variable)</b>
<p>Esta función guarda datos en formato binario, contenidos previamente en una variable, en un fichero, cuya ruta/nombre viene dado por su primer parámetro.</p> <p>Esta función no necesita tener abierto previamente el fichero: es ella quien lo abre transparentemente para escribir en él.</p> <p>Esta función sólo puede guardar una variable a la vez. Sin embargo, esa variable puede ser una tabla o estructura: en ese caso se guardarán secuencialmente (es decir, por orden uno detrás de otro) todos los elementos de la tabla o estructura. No es válido, en cambio, almacenar datos de tipo POINTER.</p> <p style="text-align: center;"><u>PARÁMETROS:</u></p> <p style="text-align: center;">STRING FICHERO : Ruta del fichero a escribir                      VAR VARIABLE : Nombre de la variable a guardar</p> <p>Un ejemplo de su posible uso:</p> <pre> program Test_SAVE; global     struct dat[9]         int a,b,c;     end </pre>

```

end
begin
write(0,160,100,4,"1) Salvar los valores de la estructura en archivo");
repeat
if(key(_1))
from x=0 to 9
dat[x].a=x+1;
dat[x].b=x+1;
dat[x].c=x+1;
end
save("save.dat",dat);
end
frame;
until(key(_esc))
end

```

### LOAD("fichero",variable)

Esta función lee datos binarios de un fichero cuya ruta/nombre viene dado por su primer parámetro, y los asigna a una variable para poder así trabajar en el código fuente con la información obtenida.

Si por ejemplo la variable es de tipo int (es decir, de 4 bytes), se leerán 4 bytes del archivo. Si es de tipo word se leerán 2 bytes y si es de tipo byte se leerá uno.

Esta función no necesita tener abierto previamente el fichero: es ella quien lo abre transparentemente para leer de él.

Esta función sólo puede leer una variable a la vez. Sin embargo, esa variable puede ser una tabla o estructura: en ese caso se leerán cada vez todos los elementos de la tabla o estructura secuencialmente (es decir, por orden uno detrás de otro).

#### PARÁMETROS:

STRING FICHERO : Ruta del fichero a escribir  
VAR VARIABLE : Nombre de la variable a leer

Un ejemplo de su posible uso:

```

program Test_LOAD;
global
struct dat[9]
int a,b,c;
end
end
begin
from x=0 to 9
write_var(0,60,x*10+85,4,dat[x].a);
write_var(0,160,x*10+85,4,dat[x].b);
write_var(0,260,x*10+85,4,dat[x].c);
end
write(0,1,40,3,"1)Leer fichero creado antes en el ejemplo de SAVE");
write(0,10,65,3,"Valores del fichero:");
repeat
if(key(_1))

```

```

    load("save.dat",dat);
end
frame;
until(key(_esc))
end

```

Pronto se comprobará que su uso es muy similar a las ya conocidas, y es elección personal del programador decantarse por una de las dos “familias”. *Save* viene a equivaler al trío de funciones *fopen -en modo escritura-/fwrite/fclose*, y *Load* viene a equivaler al trío *fopen -en modo lectura-/fread/fclose*.

Vamos a empezar con el código inicial del juego, el cual consistirá en 2 imágenes que se mueven en la pantalla gracias al control del jugador 1 y al jugador 2. Por lo tanto, lo primero será crear un archivo FPG –llamado “fichero.fpg”– con dos imágenes –dos bolas de distintos color bastará– que representarán cada uno de los jugadores; una tendrá el código 001 y otra el 002. Empecemos escribiendo esto:

```

Program mi_juego;
global
    int mifichero_fpg;
end
Begin
    set_mode(800,600,16);  mifichero_fpg=load_fpg("fichero.fpg");

    write(0,400,290,4,"1=JUGAR");
    write(0,400,300,4,"2=VER DEMO");
    write(0,400,310,4,"3=SALIR");
    loop
        if(key(_1))
            delete_text(0);
            empieza_partida();
            break;
        end
        if(key(_2))
            delete_text(0);
            ver_una_partida();
            break;
        end
        if(key(_3))
            break;
            exit("");
        end
    end
end
frame;

end

Process empieza_partida()
BEGIN
    jugador1();
    jugador2();
    loop
        write(0,400,30,4,"PULSA ESC PARA SALIR");
        if(key(_esc))

```

```

        let_me_alone();
        breaK;
        exit("");
    end
    frame;
    delete_Text(0);
end
end

Process jugador1()
BEGIN
    file=mifichero_fpg;
    graph=1;
    loop
        if(key(_up)) y=y-2; end
        if(key(_down)) y=y+2; end
        if(key(_left)) x=x-2; end
        if(key(_right)) x=x+2; end
        frame;
    end
end

Process jugador2()
BEGIN
    file=mifichero_fpg;
    graph=2;
    loop
        if(key(_w)) y=y-2; end
        if(key(_s)) y=y+2; end
        if(key(_a)) x=x-2; end
        if(key(_d)) x=x+2; end
        frame;
    end
end

//Más tarde nos encargaremos de este proceso
Process ver_una_partida()
BEGIN
    loop
        frame;
    end
end
end

```

Ahora vamos a grabar una partida. Necesitaremos una estructura con unos campos que contengan las variables X e Y de los jugadores (para ir guardándolas cada frame). Así que en la sección de declaraciones de variables globales escribe:

```

struct rec
    int pos_jug1x[4000];
    int pos_jug1y[4000];
    int pos_jug2x[4000];
    int pos_jug2y[4000];

```

*end*

Y aqui nos encontramos con la desventaja de este metodo. ¿Como sabremos si tendremos que usar 4000 o más?. Pues eso es problema tuyo. Yo aconsejaria estimar una cantidad de acuerdo a un tiempo limite de duracion que tenga el juego o, mejor, usar arrays dinámicos. No obstante, los arrays dinámicos es un concepto avanzado de programación que en este curso ni tocamos.

Tambien vamos a necesitar una variable auxiliar que vaya recorriendo cada posicion en cada frame (enseguida lo entenderás mejor), así que crearemos la siguiente variable, en la sección global:

```
Int var_aux=0;
```

Esta variable irá incrementando su valor de a 1 cada frame. Para eso, justo antes de la orden frame del proceso “empieza\_partida()” escribe:

```
Var_aux=var_aux+1;
```

Y justo antes de la orden frame del proceso “jugador1()”, escribe:

```
rec.pos_jug1x[var_aux]=x;  
rec.pos_jug1y[var_aux]=y;
```

¿Qué estamos haciendo con esto? Guardar en cada frame, el valor actual de X e Y dentro de la estructura rec, rellenando (en cada frame también) de forma secuencial cada uno de sus hasta 4000 elementos respectivos, de manera que en cada elemento se almacenen las posiciones consecutivas por las que ha pasado el jugador 1.

Es decir, suponiendo que el jugador 1 se encuentra en la posicion x=300 y pulsa la tecla para ir a la derecha, el valor de la variable rec.pos\_jug1x[var\_aux], cuando var\_aux sea igual a 0, esta sera igual a rec.pos\_jug1x[var\_aux]=300; . Cuando var\_aux=1 (en el siguiente frame), tendremos rec.pos\_jug1x[var\_aux]=302, cuando var\_aux valga 2 (en el siguiente frame), tendremos rec.pos\_jug1x[var\_aux]=304; y así, sucesivamente, iremos guardando las coordenadas de jugador 1.

Un detalle importante es ponerle un límite a la variable var\_aux, para que nunca sea mayor que 4000, ya que en este caso, si el juego sigue y var\_aux=4001 o superior podriamos acceder a una parte de la memoria que está fuera de la estructura que hemos creado y por tanto estariamos armando un gran lio. así que debajo de la línea *var\_aux=var\_aux+1;* del proceso “empieza\_partida()”, agregaremos lo siguiente:

```
if(var_aux>=4000) var_aux=4000; end
```

Lo mismo que hemos hecho en el proceso “jugador1()”, deberemos escribirlo en el proceso

“jugador2()”, utilizando, eso sí, `rec.jug2x` y `rec.jug2y`. Es decir, justo antes de la orden `frame` del proceso “jugador2()”, escribe:

```
rec.pos_jug2x[var_aux]=x;  
rec.pos_jug2y[var_aux]=y;
```

Ahora, lo único que nos queda es hacer que todo el contenido de la estructura `rec` pase a grabarse al disco duro en forma de fichero, y tenerlo así disponible posteriormente para su visualización. Haremos que se grabe pulsando la tecla `ESC`, y utilizaremos la función `save` (ya sabemos que también podríamos haber utilizado –es más flexible– el conjunto de órdenes *fopen en modo “o\_write”/fwrite/fclose*).

Así pues, cuando pulsemos la tecla `ESC`, con la función `save()`; guardaremos en el disco la estructura `rec` que contiene la posiciones de los jugadores. Para ello, en el proceso “`empieza_partida()`”, dentro del bloque `if(key(_esc))/end` escribe lo siguiente (ojo, ha de ser la primera línea de todas, antes del `let_me_alone()`):

```
Save(“partida.dem”,rec);
```

Y ya sólo nos queda poder visualizar la partida previamente grabada. Para ello modificaremos el proceso “`ver_una_partida()`”, que ahora no hace nada, y crearemos 2 procesos mas que simularan al jugador 1 y 2, y haran lo mismo que hicieron éstos en la partida que se grabó. Para ello, sus valores `X` e `Y` tendran el valor de las variables `rec.pos_jug1x[var_aux]`, `rec_pos_jug1y[var_aux]`, etc.

Llamaremos a estos procesos “`rec_jugador1()`” y “`rec_jugador2()`”, y su código (junto con el código modificado de “`ver_una_partida()`”) es éste:

```
Process ver_una_partida()  
BEGIN  
    load("PARTIDA.dem",rec); //cargamos el archivo  
    rec_jugador1();  
    rec_jugador2();  
    loop  
        write(0,400,30,4,"PULSA ESC PARA SALIR");  
        if(key(_esc))  
            let_me_alone();  
            break;  
            exit("");  
        end  
        var_aux=var_aux+1;  
        if(var_aux>=4000) var_aux=4000; end  
        frame;  
        delete_text(0);  
    end  
end  
  
Process rec_jugador1()  
BEGIN
```

```

        file=mifichero_fpg;
        graph=1;
        loop
//X TENDRA EL VALOR DE LA VARIABLE POS_JUG1X EN LA POSICION VAR_AUX
                x=rec.pos_jug1x[var_aux];
//Y TENDRA EL VALOR DE LA VARIABLE POS_JUG1Y EN LA POSICION VAR_AUX
                y=rec.pos_jug1y[var_aux];
                frame;
        end
end

Process rec_jugador2()
BEGIN
        file=mifichero_fpg;
        graph=2;
        loop
                x=rec.pos_jug2x[var_aux];
                y=rec.pos_jug2y[var_aux];
                frame;
        end
end

```

### **Trabajar con cadenas:**

Fénix incorpora bastantes funciones que pueden ayudar al programador en el tratamiento de cadenas de texto. Veamos unas cuantas:

*\*Nota: Las funciones CHR, ASC, y SUBSTR también son funciones “de cadena”, pero están explicadas posteriormente en este capítulo, en el apartado de introducción de caracteres por teclado.*

#### **LEN(“cadena”)**

Devuelve el número de caracteres que tiene la cadena pasada como parámetro (es decir, su longitud).

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO : INT: Número de caracteres de la cadena dada

#### **LCASE(“cadena”)**

Devuelve una cadena idéntica a la original, pero convirtiendo todas las letras mayúsculas a minúsculas (incluyendo letras con acentos o tildes).

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO : STRING : Cadena resultante

#### **UCASE(“cadena”)**

Devuelve una cadena idéntica a la original, pero convirtiendo todas las letras minúsculas a mayúsculas (incluyendo letras con acentos o tildes).

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO : STRING : Cadena resultante

### **STRCASECMP("cadena","cadena2")**

Dadas dos cadenas, realiza una comparación entre las mismas sin distinguir las mayúsculas de las minúsculas, y devuelve un valor que indica su diferenciación:

-0 quiere decir que las cadenas son iguales.

-Un valor negativo indica que el primer carácter diferente entre las dos cadenas por orden alfabético va después en la primera cadena que en la segunda.

-Un valor positivo indica que el primer carácter diferente entre las dos cadenas por orden alfabético va después en la segunda cadena que en la primera.

PARÁMETROS :

STRING CADENA : Cadena

STRING CADENA2 : Otra cadena

VALOR DE RETORNO : INT : 0 si las cadenas son equivalentes

### **STRREV("cadena")**

Esta función devuelve una cadena creada tras dar la vuelta a la cadena que recibe como parámetro. Es decir, STRREV("ABCD") devolvería "DCBA".

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO : STRING : Cadena resultante

### **LPAD("cadena",tamaño)**

Esta función devuelve una cadena creada tras añadir espacios (carácter ASCII 32: " ") al comienzo a la cadena original, hasta conseguir que alcance una longitud igual a la especificada como segundo parámetro. Si la cadena original tiene un tamaño igual o superior al especificado, esta función devuelve la cadena original sin ningún cambio.

También existe la función RPAD, que expande una cadena añadiendo espacios a la derecha, hasta el tamaño dado. Funciona igual que LPAD.

PARÁMETROS:

STRING CADENA : Cadena original

INT TAMAÑO : Tamaño deseado

VALOR DE RETORNO : STRING : Cadena resultante

### TRIM("cadena")

Esta función devuelve una cadena idéntica a la original, pero eliminando cualquier espacio o serie de espacios que encuentre al principio o al final de la misma. Por espacio se entiende cualquier carácter de código ASCII 9 (tabulador), 10 (salto de línea), 13 (retorno de carro) ó 32 (espacio).

PARÁMETROS: STRING CADENA : Cadena original

VALOR DE RETORNO: STRING : Cadena resultante

Un ejemplo de estas últimas funciones podría ser:

```
program hola;
private
  string a="hola ";
  string b="adiós";
end
begin
  write(0,50,10,4,TRIM(a));
  write(0,50,20,4,LPAD(a,17));
  write(0,50,30,4,RPAD(a,17));
  write(0,50,40,4,UCASE(a));
  write(0,50,50,4,STRREV(a));
  write(0,50,60,4,STRCASECMP(a,b));
  write(0,50,70,4,LEN(a));
  write(0,50,80,4,ASC("I"));
  write(0,50,90,4,CHR(49));
  write(0,50,100,4,SUBSTR(b,2,3)); //Vista más adelante
loop
  frame;
end
end
```

A continuación comento las funciones de conversión:

### atoi("cadena")

Convierte una cadena en número entero. Dada una cadena que contenga una serie de dígitos, devuelve el número entero que representan. Si la cadena contiene caracteres adicionales después de los dígitos, estos serán ignorados. Si la cadena comienza por caracteres que no corresponden a dígitos, la función no detectará el número y devolverá 0.

PARÁMETROS : STRING CADENA : Cadena original

VALOR DE RETORNO: INT : Número resultante

### itoa(valor)

Convierte un número entero en cadena. Dado un número entero, devuelve una cadena con su representación, sin separador de miles.

PARÁMETROS: INT VALOR : Número a convertir

VALOR DE RETORNO : STRING : Cadena resultante

### **FORMAT(valor,decimales)**

Convierte un número decimal a cadena, con separador de miles y con el número de decimales indicado.

Si no se especifica el segundo parámetro, la función usará un número adecuado de decimales para representar el número (que pueden ser 0, por lo que se devolverá la cadena sin punto decimal).

PARÁMETROS:

FLOAT VALOR : Valor a convertir

INT DECIMALES : *(opcional)* Número de decimales

VALOR DE RETORNO: STRING : Cadena resultante

Además de estas funciones descritas, también existen las funciones ATOF y FTOA, que trabajan igual que ATOI y ITOA respectivamente, pero usando números en coma flotante (decimales)

Veamos un pequeño ejemplo de estas funciones:

```
program la;
private
    int entera=4;
    float decimal=9.82;
    string cadena="123Hola";
    string cadena2="12.3Hola";
end
begin
    set_mode(640,480);
    loop
        delete_text(0);
        write(0,250,50,4,"Conversión de un entero en cadena: "+itoa(entera));
        write(0,250,80,4,"Conversión de una cadena a entero: "+atoi(cadena));
        write(0,250,110,4,"Conversión de un número decimal en cadena: "+ftoa(decimal));
        write(0,250,140,4,"Conversión de un número decimal a cadena con determinados
decimales: "+format(decimal,4));
        write(0,250,170,4,"Conversión de una cadena a número decimal: "+atof(cadena2));
        frame;
    end
end
```

Y por último, comento las funciones de búsqueda:

### **FIND("cadena","buscar",posición)**

Busca la cadena especificada como segundo parámetro en el interior de la cadena especificada como

primer parámetro, y devuelve la posición de la primera coincidencia que encuentre, o -1 para indicar que no encontró ninguna. Esta posición es el número de carácter desde la izquierda de la cadena: 0 para el primer carácter, 1 para el segundo, y así sucesivamente. El tercer parámetro, opcional, indica el primer carácter donde dará comienzo la búsqueda. La parte de la cadena a la izquierda de éste, será ignorada. Si no se especifica el parámetro, se empezará por el primer carácter (posición 0).

La búsqueda se hace de izquierda a derecha. Sin embargo, es posible hacer también la búsqueda de derecha a izquierda si se especifica como tercer parámetro un número negativo, que indicará además la primera posición a tener en cuenta (-1 para el último carácter de la cadena, -2 para el penúltimo, y así sucesivamente).

Es importante destacar que FIND considera caracteres diferentes las minúsculas de las mayúsculas

#### PARÁMETROS:

STRING CADENA : Cadena original

STRING BUSCAR : Cadena a buscar

INT POSICIÓN : (*opcional*) Posición inicial de la búsqueda

VALOR DE RETORNO: INT : Posición del primer carácter de la primera aparición de la cadena buscada, o -1 si no se encontró

Un ejemplo de su uso sería:

```
program Test_FIND;
global
  string Txt="¡Hola Mundo, hola!";
begin
  write(0,10,10,3,"Test FIND para Fénix...");
  write(0,10,30,3,"Texto = ¡Hola Mundo, hola!");
  write(0,10,40,3,"FIND(H) = "+itoa(find(Txt,"H")));
  write(0,10,50,3,"FIND(Hola) = "+itoa(find(Txt,"Hola")));
  write(0,10,60,3,"FIND(Hola a partir del 5 caracter) = "+itoa(find(Txt,"Hola",4)));
  write(0,10,70,3,"FIND(M) = "+itoa(find(Txt,"M")));
  write(0,10,80,3,"FIND(Mundo) = "+itoa(find(Txt,"Mundo",-2)));
  write(0,10,90,3,"FIND(!) = "+itoa(find(Txt,"!")));
  write(0,10,100,3,"FIND(O) = "+itoa(find(Txt,"O")));
  write(0,160,190,4,"Pulsa ESC para Salir...");
repeat
  frame;
until(key(_esc))
end
```

#### **REGEX("expresión","cadena")**

Busca una expresión regular dentro de una cadena. Es decir, dada una expresión regular, comprueba si la cadena dada la cumple en algún punto, y devuelve la posición (número de carácter desde la izquierda, empezando en 0) donde la encuentra por primera vez, o -1 si no se cumple.

Una expresión regular puede entenderse como una forma avanzada de búsqueda. La expresión regular puede ser una serie de caracteres corrientes como "abc", con lo que esta función se comportará igual que la función FIND haciendo una búsqueda. Pero la verdadera potencia es que la expresión regular puede

contener toda una serie de caracteres "comodín" (metacaracteres) que tienen significados específicos.

Los siguientes metacaracteres son reconocidos:

**"."**: un punto quiere decir "cualquier carácter". Por ejemplo, "ab.de" permite encontrar "abcde" y "abJde", pero no "abde".

**"^"**: el exponencial quiere decir "principio de cadena" si la expresión regular empieza por ^. Por ejemplo, "^abc" permite encontrar "abcdef" pero no "fabcedf".

**"\$"**: el dólar quiere decir "final de cadena" si la expresión regular acaba en \$. Por ejemplo, "abc\$" permite encontrar "deabc" pero no "abcde".

**"[]"**: los corchetes quieren decir "cualquier carácter de esta lista", y en su interior puede haber una serie de caracteres corrientes, o bien rangos de caracteres.

Por ejemplo "[abc]" permite encontrar "a", "b" ó "c", mientras que "[a-z]" permite encontrar cualquier letra entre la "a" y la "z", inclusive.

Los rangos pueden ser inacabados: "[- ]" permite encontrar cualquier carácter hasta el espacio, mientras "[ -]" permite encontrar cualquier carácter del espacio en adelante.

Además, se reconocen "clases de caracteres" POSIX, que son secuencias predefinidas de caracteres que se escriben con la sintaxis "[:clase:]" y pueden ser cualquiera de las siguientes:

- [:alpha:] para cualquier letra
- [:upper:] para letras mayúsculas
- [:lower:] para minúsculas
- [:alnum:] para letras o números
- [:digit:] para números
- [:space:] para espacios.

**"()"**: los paréntesis permiten agrupar secuencias de caracteres y metacaracteres. Esto tiene varias utilidades: por ejemplo, se pueden usar los metacaracteres de repetición como "\*" o "?" después del paréntesis de cierre; o se pueden poner varias secuencias dentro de los paréntesis, separadas por el signo "|". La expresión encontrará cualquiera de las posibilidades. Por ejemplo, "(abc|efg)" encontrará "abcde" al igual que "defgh", pero no "bcdef".

**"\*"**: Indica "0 o más repeticiones de lo anterior", donde "lo anterior" se refiere al último carácter o metacarácter. Este metacarácter, al igual que los metacaracteres de repetición que detallamos a continuación, es más útil concatenado detrás de secuencias con corchetes o paréntesis.

Por ejemplo, "[abc]\*" encuentra cualquier secuencia de 0 o más caracteres "a", "b", o "c". Es decir, "[abc]\*" permite encontrar "aabc", "bbbcca", "bca" o "a". De la misma forma, "(abc)\*" permite encontrar "abcabcabc", "abcabc" o "abc", pero no "aabbcc".

Es importante tener en cuenta que 0 repeticiones también son válidas, por lo que la expresión regular "[abc]\*" en realidad se cumple siempre, ya que cualquier cadena contiene 0 o más repeticiones de cualquier secuencia.

**"+"**: encuentra 1 o más repeticiones de lo anterior, del último carácter o metacaracter anterior a este símbolo. Por ejemplo, "[abc]+" encuentra "aabbcc" y "bccba", pero no "defg".

**"?"**: encuentra 0 ó 1 repeticiones de lo anterior. Es decir, permite marcar un trozo de la expresión como opcional.

**"{"**: encuentra un rango de repeticiones de lo anterior. El rango son dos dígitos separados por comas,

aunque uno de ellos puede omitirse.

Por ejemplo, "[abc]{2,}" encuentra cualquier secuencia de dos o más caracteres "a", "b" o "c", mientras "[abc]{,4}" encuentra cualquier secuencia de hasta cuatro caracteres del mismo rango.

Hay que tener en cuenta que si se omite el primer número, 0 también es un número válido de repeticiones. Si se especifica un número sólo, busca ese número exacto de repeticiones.

Además, la función REGEX cada vez que se ejecuta rellena automáticamente una tabla global de cadenas llamado REGEX\_REG[] con cada sección entre paréntesis en la expresión original de la cadena localizada, empezando por el índice 1. Esta posibilidad requiere mención aparte, ya que es muy potente, pues permite hacer cosas como buscar una fecha y localizar sus partes. La expresión regular "(.)/(..)/(....)" buscará expresiones del estilo "10/04/1980" y rellena en REGEX\_REG[1] el día, en REGEX\_REG[2] el mes, y en REGEX\_REG[3] el año.

#### PARÁMETROS:

STRING EXPRESIÓN : Expresión regular

STRING CADENA : Cadena original

VALOR DE RETORNO: INT : Posición encontrada o -1 si no se encontró

Un ejemplo trivial sería éste (el mundo de las expresiones regulares es un universo en sí mismo):

```
program hola;
private
  string a="pepitolamanolito";
end
begin
/*Busco cadenas de cuatro caracteres de los cuales los dos últimos han de ser "la"*/
  write(0,50,10,4,"Posicion:" + regex("..la",a));
/*Busco cadenas que SÓLO se compongan de los caracteres "ol" (es decir, que antes y después de ellos
haya espacios*/
  write(0,50,20,4,"Posicion:" + regex("^ol$",a));
/*Busco cadenas de cuatro caracteres de los cuales los tres últimos han de ser "ola" y el primer alguna de
las letras a,s o d.*/
  write(0,50,30,4,"Posicion:" + regex("[asd]ola",a));
/*Busco cadenas que sean como "la", "hala", "hahala", "hahahala",etc*/
  write(0,50,40,4,"Posicion:" + regex("(ha)*la",a));
/*Busco cadenas que sean como "hala", "hahala", "hahahala",etc*/
  write(0,50,50,4,"Posicion:" + regex("(ha)+la",a));
/*Busco cadenas "la" o "hala"*/
  write(0,50,60,4,"Posicion:" + regex("(ha)?la",a));
/*Busco cadenas "hahala", "hahahala" o "hahahahala"*/
  write(0,50,70,4,"Posicion:" + regex("(ha){2,4}la",a));
/*Busco cadenas "hala" o "hola"*/
  write(0,50,80,4,"Posicion:" + regex("(ha|ho)la",a));
/*Busco cadenas íntegramente formadas por dígitos*/
  write(0,50,90,4,"Posicion:" + regex("[[:digit:]]",a));
loop
  frame;
end
end
```

## REGEX\_REPLACE("expresión","cambia","cadena")

Busca una expresión regular dentro de una cadena, y sustituye cada coincidencia por el nuevo texto indicado. Es decir, dada una expresión regular, comprueba si la cadena dada la cumple en algún punto, y sustituye lo encontrado por el nuevo texto indicado como tercer parámetro. Devuelve la cadena resultante (no se modifica la original).

Este nuevo texto es una cadena corriente, con la salvedad de que es posible usar la secuencia "\0", "\1", etc, hasta "\9", para sustituir por las secciones entre paréntesis de la expresión regular original, tal cual se encontraron en cada posición. Para ver qué es una expresión regular y qué metacaracteres son admitidos, consulta la documentación de la función REGEX

### PARÁMETROS:

STRING EXPRESIÓN : Expresión regular

STRING CAMBIA : Cadena con el nuevo texto

STRING CADENA : Cadena original

VALOR DE RETORNO: STRING : Cadena resultante con todos los cambios

Pongamos un ejemplo muy simple simple de uso de REGEX\_REPLACE:

```
program regex_replace_test;

global
    string result;
end
begin
    result=REGEX_REPLACE("caca","bombon","el caca comia caca con caca y nueces");
    write(0,10,10,0,result);
    repeat
        frame;
    until(key(_enter))
end
```

Y otro:

```
program regex_replace_test;

global
    string result;
end
begin
    result=REGEX_REPLACE("ca+","p","el caca comia caca con caca y nueces");
    write(0,10,10,0,result);
    repeat
        frame;
    until(key(_enter));
end
```

Y otro un poco más complejo:

```
program regex_replace_test;
```

```

global
    string result;
end
begin
    result=REGEX_REPLACE("(ca|co)","p","el caca comia caca con caca y nueces");
    write(0,10,10,0,result);
    repeat
        frame;
    until(key(_enter))
end

```

Y, evidentemente, para obtener una información más completa y técnica sobre estas funciones y todas las demás, no olvides dirigirte a la ayuda de referencia oficial de Fénix.

Finalmente, comentar la existencia de otras dos funciones de cadena, SPLIT y JOIN, cuya finalidad es respectivamente, partir una cadena en varias cadenas utilizando una determinada expresión regular como separador; y unir varias cadenas en una.

### SPLIT("Expresión","Cadena",&ARRAY,TAMAÑO)

A partir de una expresión regular que actúa como separador, esta función divide la cadena original en partes. Estas partes se almacenarán como elementos del vector de strings indicado como tercer parámetro (precedido del símbolo &). El cuarto parámetro es el número máximo de elementos que podrá almacenar (es decir, la longitud del array en sí).

Por ejemplo, si la cadena original es "12/04/1980" y la expresión regular es "/", la función guarda las cadenas "12", "04" y "1980" en las tres primeras posiciones del array y devuelve 3.

Es importante notar que el separador puede ser una expresión regular compleja. Es posible por ejemplo poner como separador la expresión "[/- ]", la cual dividiría la cadena allí donde hubiera un carácter "/", un carácter "-", o bien un espacio. El uso de expresiones regulares es un concepto avanzado en sí mismo, se recomienda consultar la función REGEX para obtener más información.

#### PARAMETROS:

STRING "Expresión" : Expresión regular  
 STRING "Cadena" : Cadena original  
 STRING POINTER ARRAY: Vector para guardar los resultados  
 INT TAMAÑO : Número de elementos en el vector

VALOR RETORNADO: INT: Número de elementos

Un ejemplo, que parte una cadena en trozos utilizando como separador el símbolo de espacio:

```

PROGRAM lala;
private
    string cadena="Esto es una cadena de prueba.";
    string mivector[9];
    int elementos,i;
End
BEGIN
set_mode(320,240,16);
elementos=split(" ",cadena,&mivector,9);
for(i=0;i<elementos;i++)

```

```

        write(0,100,10+10*i,4,mivector[i]);
end
loop
    frame;
    if(key(_esc)) break;end
end
end

```

### JOIN("Separador",&ARRAY,TAMAÑO)

Dado un array de cadenas, esta función junta todas ellas en una cadena final, intercalando la cadena especificada como separador entre los elementos.

Esta función permite obtener el efecto contrario a la función SPLIT .Por ejemplo, si un array contiene tres cadenas: "12", "04" y "1980", y se utiliza la cadena "/" como separador, esta función devolverá la cadena "12/04/1980".

#### PARAMETROS:

STRING "Separador" : Cadena a intercalar entre elementos

STRING POINTER ARRAY: Vector de donde obtener las cadenas a unir

INT TAMAÑO : Número de elementos en el vector

VALOR RETORNADO: STRING: Cadena generada a partir de la unión de los elementos.

Un ejemplo, que crea una cadena a partir de los valores de los elementos de un vector de strings, separándolos entre ellos con el símbolo del espacio:

```

PROGRAM lala;
private
    string cadena;
    string mivector[9]="Hola","qué","tal","yo","bien","y","tu?";
End
BEGIN
set_mode(320,240,16);
cadena=join("",&mivector,9);
write(0,100,100,4,cadena);
loop
    frame;
    if(key(_esc)) break;end
end
end

```

### Trabajar con fechas:

Fénix no tiene demasiada flexibilidad a la hora de permitirnos utilizar funciones predefinidas que trabajen con fechas. De hecho, sólo incorpora dos.

### TIME()

Esta función devuelve la fecha actual en formato entero como el número de segundos transcurridos desde el 1 de Enero de 1970.

El valor devuelto por esta función debería utilizarse con la función FTIME para transformarla a un formato

legible.

### FTIME("formato",time)

Esta función nos devuelve una cadena de texto con el formato de fecha especificado en el primer parámetro, a partir de la fecha en formato entero que recibe como segundo parámetro. La fecha en formato entero corresponde al número de segundos transcurridos desde el día 1 de Enero de 1970, por ejemplo generada por la función TIME.

Para especificar el formato se le pasa a la función una cadena donde todos los caracteres de texto se respetarán y las siguientes secuencias de caracteres se transforman acorde al siguiente listado:

**%d**: Día, dos dígitos (01-31)  
**%m**: Mes, dos dígitos (01-12)  
**%y**: Año, dos dígitos (00-99)  
**%Y**: Año, cuatro dígitos  
**%H**: Hora, dos dígitos (00-23)  
**%M**: Minuto, dos dígitos (00-59)  
**%S**: Segundo, dos dígitos (00-59)  
**%e**: Día, uno o dos dígitos (1-31)  
**%k**: Hora, uno o dos dígitos (0-23)  
**%a**: Nombre abreviado del día  
**%A**: Nombre completo del día  
**%b**: Nombre abreviado del mes  
**%B**: Nombre completo del mes  
**%C**: Centuria, dos dígitos (19-99)  
**%I**: Hora americana, dos dígitos (01-12)  
**%l**: Hora americana, uno o dos dígitos (1-12)  
**%p**: Coletilla americana de la hora (AM/PM)  
**%P**: Coletilla americana de la hora (am/pm)  
**%u**: Día de la semana (1-7)  
**%U**: Día de la semana empezando por 0 (0-6)  
**%j**: Día del año, tres dígitos (001-366)  
**%U**: Número de semana del año, dos dígitos (00-53)

#### PARÁMETROS:

STRING FORMATO : Formato en el que se presentará la fecha

INT TIME : Fecha en formato entero, obtenido normalmente a partir de la función TIME

Un ejemplo sencillo de uso de ambas funciones sería:

```
program Test_Ftime;
begin
  write(0,10,10,3,"Test ftime para Fénix...");
  write(0,160,390,4,"Pulsa ESC para Salir...");
  write(0,10,60,3,"%d - Día, dos dígitos: "+ftime("%d",time()));
  write(0,10,70,3,"%e - Día, uno o dos dígitos: "+ftime("%e",time()));
  write(0,10,80,3,"%m - Mes, dos dígitos: "+ftime("%m",time()));
  write(0,10,90,3,"%y - Año, dos dígitos: "+ftime("%y",time()));
  write(0,10,100,3,"%Y - Año, cuatro dígitos: "+ftime("%Y",time()));
  write(0,10,110,3,"%H - Hora, dos dígitos: "+ftime("%H",time()));
```

```

write(0,10,120,3,"%k - Hora, unos o dos dígitos: "+ftime("%k",time()));
write(0,10,130,3,"%M - Minuto, dos dígitos: "+ftime("%M",time()));
write(0,10,140,3,"%S - Segundo, dos dígitos: "+ftime("%S",time()));
write(0,10,150,3,"%j - Día del año, tres dígitos: "+ftime("%j",time()));
write(0,10,160,3,"%U - Numero de semana del año: "+ftime("%U",time()));
repeat
  frame;
until(key(_esc))
end

```

Y, evidentemente, para obtener una información más completa y técnica sobre estas funciones y todas las demás, no olvides dirigirte a la ayuda de referencia oficial de Fénix.

### **Funciones matemáticas:**

Fénix aporta un conjunto (limitado) de funciones matemáticas para poder realizar cálculos aritméticos básicos. Algunas son:

#### **ABS(número)**

Esta función devuelve el valor absoluto de un número dado, es decir, convierte cualquier número negativo en positivo y deja números positivos o el 0 inalterados.

PARÁMETROS: FLOAT NÚMERO : Número en coma flotante

#### **COS(número)**

Esta función devuelve el coseno de un ángulo dado en milésimas de grado (la unidad estándar para especificar ángulos en Fénix). El resultado será un número en coma flotante entre -1 y 1.

De igual forma existen en Fénix las funciones **SIN(número)**, que calcula el seno de un ángulo, y **TAN(número)**, que calcula su tangente. Ambas funciones también tienen como parámetro un número FLOAT e igualmente devuelven un número FLOAT.

PARÁMETROS: FLOAT NÚMERO : Ángulo en milésimas de grado (90000 = 90°)

#### **ACOS(número)**

Devuelve el arcocoseno de un valor. (El valor ha de ser un número decimal entre -1 y 1).

Es decir, devuelve un ángulo en milésimas de grado, entre 0 y 180000 cuyo coseno equivale al valor indicado como parámetro. Este resultado devuelto será un número en coma flotante.

De igual forma existen en Fénix las funciones **ASIN(número)**, que calcula el arcoseno de un número entre -1.0 y 1.0, y **ATAN(número)**, que calcula su arcotangente.

PARÁMETROS: FLOAT NÚMERO : Valor entre -1.0 y 1.0

### SQRT(número)

Esta función devuelve la raíz cuadrada, en coma flotante, de un número especificado como parámetro.

PARÁMETROS: FLOAT NÚMERO : Número en coma flotante

### POW(número,potencia)

Esta función devuelve el resultado, en coma flotante, de elevar un número a una potencia dada. Por ejemplo, el resultado de POW(2,3) es 8 ( $2*2*2$ ). Escribir POW(49,0.5) sería lo mismo que SQRT(49).

PARÁMETROS:

FLOAT NÚMERO : Número en coma flotante

FLOAT POTENCIA : Potencia a la que se desea elevar

### Función de redondeo

Fénix, a diferencia de la mayoría de lenguajes de programación, no posee ninguna función nativa que facilite el redondeo de números decimales. Este pequeño inconveniente lo podemos suplir creando nosotros mismos una función que realice esta operación. Como ejemplo, a continuación presento una posible solución.

El programa es muy sencillo. A la variable `num_a_redondear` se le asigna el valor que se quiere redondear, y se muestra por pantalla dicho valor, y además el resultado del redondeo. Además, para comprobar que realmente esto es así, el programa ofrece la posibilidad de, mediante las teclas CTRL o ALT, subir o bajar el número a redondear -en este caso una centésima cada vez, pero puede ser cualquier otra cantidad, claro-, para poder ver dinámicamente que el resultado redondeado se amolda a los nuevos números. Como se puede comprobar, el secreto del programa está simplemente en sumar o restar 0.5 al número a redondear. ¿Y esto por qué? Por un detalle muy importante: el número a redondear es de tipo float, pero el número redondeado lo hemos definido como de tipo entero. Así que en realidad, quien realiza el redondeo es el propio Fénix, cuando hace la conversión de float a int: nosotros sólo le hemos dado el empujón. La necesidad de sumar o restar 0.5 viene del hecho que en realidad Fénix no redondea, sino corta por abajo: un 87.6 se convierte en un 87, un 109.8 se convierte en un 109, un 54.2 se convierte en un 54... Así que para conseguir el efecto de redondeo que todos conocemos (es decir, si las décimas son 5 o superior, se redondea al número inmediatamente superior) hay que hacer este pequeño truco. Este código no obstante tiene un inconveniente, y es que no podremos redondear números para una posición decimal concreta (no podremos redondear a décimas o centésimas o decenas): sólo se puede redondear a números enteros.

Dejo al lector el ejercicio de convertir este programa en una función que admita como parámetro el número a redondear y devuelva el número redondeado, para poderla usar así en cualquier otro programa donde necesitemos esta funcionalidad.

```
program rounding;
global
int num_redondeado;
float num_a_redondear;
end
begin
set_mode(200,28,8);

num_a_redondear=87.9;
write(0,5,5,0, "Número a redondear:");
```

```

write(0,5,15,0, "Número redondeado:");
write_var(0,150,5,0, num_a_redondear);
write_var(0,150,15,0, num_redondeado);
loop
  if(num_a_redondear>=0)
    num_redondeado=num_a_redondear+0.5;
  end
  if(num_a_redondear<0)
    num_redondeado=num_a_redondear-0.5;
  end
  if(key(_control))
    num_a_redondear=num_a_redondear+0.01;
  end
  if(key(_alt))
    num_a_redondear=num_a_redondear-0.01;
  end
frame;
end
end

```

### RAND\_SEED(número)

Sabemos de un capítulo anterior que para generar números pseudoaleatorios disponemos de la función RAND. ¿Por qué se llaman números pseudoaleatorios y no aleatorios? Porque tal como funciona RAND, los números que genera no son completamente independientes unos de otros, y por tanto, no son exactamente aleatorios. Que una serie de números esté formada por números exactamente aleatorios implica que no se pueden hallar patrones de números que se repitan periódicamente.

La función RAND para funcionar lo que utiliza es un número inicial, llamado “semilla” (seed), el cual será el generador de toda la ristra de números pseudoaleatorios a obtener. Es decir, la función RAND necesita siempre ese número para poder empezar a trabajar. Y lo importante del tema está en que si se escoge como semilla un mismo número siempre, la ristra de números ¡siempre será la misma!, porque a partir de la semilla se genera el primer número; a partir del primero se genera el segundo, y así. Claro, si la semilla es la misma, la ristra también.

Para evitar esto, y hacer que cada vez que un programa que incluya la función RAND genere ristas de números completamente diferentes, Fénix hace que automáticamente el valor tomado como semilla de la función RAND sea la hora (en segundos) actual. Evidentemente, este número cada segundo cambiará, por lo que todos los programas que ejecutemos en segundos diferentes tendrán ristas de números diferentes.

No obstante, a veces nos puede ser de utilizar conseguir precisamente lo contrario: es decir, hacer que cada vez que se ejecute un programa, por ejemplo, genere siempre la misma ristra de valores. Para hacer eso, deberíamos poder fijar el valor de la semilla. Y eso es lo que hace la función RAND\_SEED. El parámetro que tiene es un entero que será la semilla a partir de la cual se generará la serie de números pseudoaleatorios. Y si hacemos que ese parámetro siempre valga lo mismo, ya tendremos asegurada una serie siempre igual.

Veámoslo con un ejemplo. Ejecuta el siguiente programa.

```

program haa;
private
  int mirand;
  int mifich;

```

```

end
begin
    if(file_exists("prueba.txt")==false)
        mifich=fopen("prueba.txt",o_write);
    else
        mifich=fopen("prueba.txt",o_readwrite);
        fseek(mifich,0,2);
    end
    while(timer[0]<=500)
        mirand=rand(1,9);
        fputs(mifich,itoa(mirand));
        frame;
    end
    fclose(mifich);
end

```

Este programa lo que hace es crear un fichero si no existe (o utilizar uno ya existente) y escribir en él –sin sobrescribir nada si hubiera algo ya- una serie de valores pseudoaleatorios, durante 5 segundos.

Ejecútalo una vez y abre el fichero de texto generado. Verás que es una serie de números sin ningún sentido. Ve al final del fichero y escribe un carácter que no sea un número. Éste será la marca que indicará el comienzo de una serie nueva. Vuelve a ejecutar el programa, y vuelve a abrir el fichero. Verás que justo después de la marca que habías escrito vuelve a aparecer una serie de números, que no tienen nada que ver con los de la serie anterior. Esto es porque RAND automáticamente coge la semilla de la hora del sistema, y la hora del sistema ha cambiado entre las dos ejecuciones del programa.

Ahora modifica el código anterior añadiendo la línea siguiente justo después del begin:

```
Rand_seed(3);
```

Borra el archivo de texto y vuelve a realizar los pasos anteriores: ejecutar una vez el programa, escribir una marca al final del fichero y volver a ejecutar el programa. Verás que ahora, como hemos definido en el programa que la semilla a utilizar cuando se ejecute RAND sea siempre la misma (en este caso, el número 3), las dos series de números, pese a ser pseudoaleatorias, ¡son las mismas!

Podrás deducir conmigo finalmente que poner RAND\_SEED(TIME()) y no poner nada sería lo mismo, pues.

Y, evidentemente, para obtener una información más completa y técnica sobre estas funciones y todas las demás, no olvides dirigirte a la ayuda de referencia oficial de Fénix.

### **Trabajar con gráficos generados dinámicamente en memoria:**

En determinadas ocasiones, no nos hará falta tener gráficos creados independientemente de nuestro juego ni usar contenedores FPG externos. Fénix es capaz de generar gráficos directamente desde código -es lo que se dice generación “dinámica” en memoria-. Aparentemente esto ya lo hemos visto en el apartado de primitivas gráficas, pero en realidad ahora es diferente: estamos hablando de cualquier tipo de gráfico/sprite, no de una primitiva. Y además, estos gráficos que mientras el código se está ejecutando permanecen en la RAM del ordenador, podemos grabarlos en un fichero, de manera que Fénix nos puede servir también como editor de dibujos rudimentario.

A continuación presento las funciones más importantes de generación y manipulación dinámica

de imágenes.

### NEW\_MAP ( ANCHO, ALTO, PROFUNDIDAD )

Esta función crea un nuevo gráfico en memoria, con el tamaño y número de bits por pixel de color especificados. El gráfico se crea dentro de la librería 0, y se le asigna un número cualquiera disponible, que no haya sido usado por ningún otro gráfico anterior.

La profundidad de color tiene el siguiente significado:

- Un gráfico de 1 bit por pixel no tiene información de color. Los pixels a 0 representan pixels transparentes, y los bits a 1 serán dibujados usando el color escogido mediante la función DRAWING\_COLOR .
- Un gráfico de 8 bits por pixel tiene 256 valores disponibles para cada pixel. Cada valor representa un índice en una tabla de colores general, la paleta de colores, que afecta a todos los gráficos de 8 bits.
- Un gráfico de 16 bits contiene información de los componentes rojo, verde y azul de cada pixel, por lo que es independiente de ninguna paleta de colores. La codificación exacta de estas componentes dentro de los 16 bits depende de la tarjeta gráfica, por lo que sólo es válido crear y usar gráficos de 16 bits si primero se establece un modo gráfico de 16 bits.

El nuevo gráfico se crea con todos los pixels a 0 (es decir, transparentes).

#### PARÁMETROS:

INT ANCHO : Ancho en pixels del nuevo gráfico

INT ALTO : Alto en pixels del nuevo gráfico

INT PROFUNDIDAD : Profundidad de color (en bits por pixel: 1, 8 ó 16)

### MAP\_PUT\_PIXEL ( LIBRERÍA, GRÁFICO, X, Y, COLOR )

Esta función permite alterar el color de un pixel determinado dentro de un gráfico. Las coordenadas dadas crecen hacia abajo y a la derecha, y el punto (0, 0) representa la esquina superior izquierda del gráfico.

En el caso de gráficos de 256 colores (8 bits), el valor de color debe ser un número de 0 a 255. En el caso de gráficos de 16 bits, el valor es una codificación de las componentes del color que varía en función de la tarjeta de vídeo y el modo gráfico. Lo normal es usar la función RGB para obtener la codificación de un color concreto, o bien usar un color obtenido por una función como MAP\_GET\_PIXEL. El color 0, tanto en 8 como en 16 bits, representa un pixel transparente.

Si se quiere realizar la misma función pero específicamente sobre el fondo de pantalla, o bien se hace uso de la librería 0 gráfico 0 dentro de map\_put\_pixel -es decir, se escribe map\_put\_pixel(0,0,x,y,color)-, o bien se puede utilizar por comodidad la función **PUT\_PIXEL(X,Y,COLOR)**, cuyos tres parámetros tienen idéntico significado que los últimos tres de map\_put\_pixel, y donde se sobreentiende que el gráfico sobre el que se pintará es el del fondo.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico dentro de la librería

INT X : Coordenada horizontal

INT Y : Coordenada vertical

INT COLOR : Color a dibujar

Con estas dos funciones ya podemos generar nuestro primer gráfico “al vuelo” dentro de Fénix. Un ejemplo podría ser éste:

```
program hola;
begin
set_mode(640,480,16);
migrafico();
loop
    frame;
end
end

process migrafico()
private
    int a,b;
end
begin
x=50;
y=50;
graph=new_map(30,30,16);
for (a=0;a<=30;a++)
    for(b=0;b<=30;b++)
        map_put_pixel(0,graph,a,b,rgb(234,54,87));
        frame;
    end
end
loop
    frame;
end
end
```

Fijarse lo que hacemos. El proceso principal lo único que hace es llamar al proceso “migráfico”, que será el gráfico que pintaremos, y mantenerse vivo con el LOOP. Al proceso “migráfico” le asignamos una posición inicial (x=50 e y=50) y un gráfico inicial que, a diferencia de como siempre, no se obtendrá de un fichero externo, sino que lo creamos en el momento con la orden *new\_map*. Este gráfico tendrá 30 píxeles de alto y 30 píxeles de ancho (será un cuadrado), y será de 16 bits.

Pero con *new\_map*, no veremos nada. Tenemos el gráfico creado pero por defecto es transparente. Hay que rellenarlo de color. Para ello utilizamos *map\_put\_pixel*. Con esta función pintamos un pixel de un gráfico determinado con un color, y para indicar sobre qué gráfico queremos pintar el pixel, le damos como segundo parámetro el valor que tiene en este momento la variable GRAPH, el cual no es más que el identificador que acaba de devolvernos *new\_map*. Por tanto, los píxeles se pintarán sobre el nuevo gráfico acabado de crear.

Para recorrer todos los píxeles que forman el cuadrado, se ha de llamar a esta función dentro de un par de bucles anidados: de esta manera el tercer y cuarto parámetro de la función, que indican qué pixel hay que colorear, valdrán desde (0,0) -esquina superior izquierda del gráfico- hasta (30,30) -esquina inferior derecha-. El color es uno cualquiera dado por la función *rgb*. Lo más interesante del asunto es el segundo parámetro, que representa el gráfico donde se va a pintar, que precisamente es el gráfico del proceso, el cual está generado en memoria y era transparente. Acordarse de que al no pertenecer a ninguna librería FPG, el primer parámetro ha de valer 0.

Si ejecutas el ejemplo, verás que el cuadrado aparece poquito a poquito, pintándose, hasta llegar a quedar pintado completamente. ¿Por qué es esto? Porque he añadido una orden `frame` dentro de los dos `for` anidados para que a cada iteración -es decir, cada vez que se pinta un pixel- se vea los resultados por pantalla. Si quisiéramos ver el cuadrado pintado ya directamente, simplemente tendríamos que quitar esa línea, ya que entonces el primer `frame` que nos encontraríamos ya sería el del interior del loop -imprescindible, por otra parte, para que el proceso siga vivo y el cuadrado se pueda seguir viendo indefinidamente-, con lo que ya habrían pasado todas las iteraciones de los `for` anteriores,

Otro ejemplo, parecido al anterior, ofrece un efecto curioso:

```
program Test_MAP_PUT_PIXEL;
begin
  set_mode(640,480,16);
  graph=new_map(640,480,16);
  repeat
    map_put_pixel(0,graph,rand(0,639),rand(0,479),rand(1,65536));
  frame;
  until(key(_esc))
end
```

Creo que viendo el resultado es fácil de entender lo que hace este código.

Un efecto interesante para rematar este pequeño ejemplo podría ser incluir dentro del bloque `repeat/until`, justo antes de la línea del `map_put_pixel`, la línea

```
angle=angle%360000-5625;
```

Fíjate lo que ocurre: estamos variando en cada iteración la orientación del gráfico del proceso principal; en concreto lo estamos girando en la dirección de las agujas del reloj 5625 milésimas de grado. Con esto, nos aparece un bonito efecto de rotación.

Recuerda que aplicamos la operación módulo (%) al ángulo simplemente para que `ANGLE` no crezca indefinidamente. Ya se comentó cuando se introdujo la variable `ANGLE` que cuando se gira una vuelta entera una imagen, su valor no se pone a 0 automáticamente sino que continúa creciendo más allá de las 360000 milésimas de grado. Si queremos que a cada vuelta, la variable `ANGLE` vuelva a valer 0, tenemos que aplicarle el módulo con la cantidad límite donde se produce el “reseteo” (360000). Esto es fácil verlo si añades un “`write_var`” mostrando lo que vale `ANGLE`, tanto si se le efectúa el módulo como si no.

Otras funciones interesantes son:

### MAP\_CLEAR ( LIBRERÍA, GRÁFICO, COLOR )

Esta función borra el contenido de un gráfico en memoria, estableciendo todos los pixels al color indicado, sin descargarlo. Es otra manera de decir que rellena de un color el gráfico completo, por lo que utilizar esta función es una manera de pintar rápidamente un gráfico creado anteriormente con `new_map`.

El significado del parámetro `color` depende de la profundidad de color del gráfico. Para un gráfico de 8 bits, representa un número de color de 0 a 255. Para un gráfico de 16 bits, es un número de 0 a 65535 con un código de color que varía en función del modo gráfico. Ya sabes que puedes utilizar la función `RGB` para obtener la codificación de un color determinado a partir de sus componentes. En cualquier caso, el color 0 representa un pixel transparente, tanto en 8 como en 16 bits.

Recuerda también que el gráfico perteneciente a la librería 0 (la del sistema) cuyo código identificador es 0 corresponde a la imagen de fondo de pantalla.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG  
INT GRÁFICO : Número de gráfico dentro de la librería  
INT COLOR : Número de color

Un ejemplo donde se puede ver el cambio repetido de color que sufre un gráfico gracias a repetidas llamadas a `map_clear` -con la ayuda de temporizadores para realizar el cambio de color de manera automática- es:

```
program Test_MAP_CLEAR;
begin
  set_mode(640,480,16);
  graph=new_map(300,200,16);
  x=320;
  y=240;
  timer=0;
  repeat
    if(timer>200)
      map_clear(0,graph,rgb(rand(0,255),rand(0,255),rand(0,255)));
      timer=0;
    end
  frame;
until(key(_esc))
end
```

#### **MAP\_GET\_PIXEL ( LIBRERÍA, GRÁFICO, X, Y )**

Esta función recupera el color de un pixel determinado dentro de un gráfico. Las coordenadas dadas crecen hacia abajo y a la derecha, y el punto (0, 0) representa la esquina superior izquierda del gráfico.

En el caso de gráficos de 256 colores (8 bits), el valor devuelto por esta función es un color de 0 a 255. En el caso de gráficos de 16 bits, el valor devuelto es una codificación de las componentes del color que depende de la tarjeta de vídeo y el modo gráfico. Puede usarse `GET_RGB` para obtener los valores aproximados de las componentes del color.

Esta función será de vital importancia cuando se trate el tema de los mapas de durezas (visto en el capítulo tutorial de RPG). En su momento se explicará pormenorizadamente su funcionalidad y diferentes usos.

Si se quiere realizar la misma función pero específicamente sobre el fondo de pantalla, o bien se hace uso de la librería 0 gráfico 0 dentro de `map_get_pixel` -es decir, se escribe `map_get_pixel(0,0,x,y)`-, o bien se puede utilizar por comodidad la función **GET\_PIXEL(X,Y)**, cuyos dos parámetros tienen idéntico significado que los últimos dos de `map_get_pixel`, y donde se sobreentiende que el gráfico sobre el que se pintará es el del fondo.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG  
INT GRÁFICO : Número de gráfico dentro de la librería  
INT X : Coordenada horizontal

## INT Y : Coordenada vertical

Como ejemplo introductorio a la funcionalidad de este comando, aquí va el siguiente ejemplo:

*/\*La idea del programa es sencilla: existe un fondo de pantalla con diferentes colores, y existe un cuadradito negro que se va moviendo intermitentemente y aleatoriamente por todo ese fondo. Cada vez que el cuadradito se para en un punto, aparece escrito el código de color que devuelve map\_get\_pixel correspondiente al color donde se ha situado el cuadradito.\*/*

```
program Test_MAP_GET_PIXEL;
```

```
global
```

```
int a,b;
```

```
int fondo;
```

```
int color;
```

```
end
```

```
begin
```

```
set_mode(640,480,16);
```

*/\*Las líneas siguientes no son más que bucles para generar un fondo formado por diferentes columnas de colores, a modo de "carta de ajuste". Como se puede ver, las columnas son de 100 píxeles de ancho y cada una se pinta de un color\*/*

```
fondo=new_map(640,480,16);
```

```
for(a=0;a<100;a++)
```

```
for(b=0;b<480;b++)
```

```
map_put_pixel(0,fondo,a,b,rgb(255,0,0));
```

```
end
```

```
end
```

```
for(a=100;a<200;a++)
```

```
for(b=0;b<480;b++)
```

```
map_put_pixel(0,fondo,a,b,rgb(0,255,0));
```

```
end
```

```
end
```

```
for(a=200;a<300;a++)
```

```
for(b=0;b<480;b++)
```

```
map_put_pixel(0,fondo,a,b,rgb(0,0,255));
```

```
end
```

```
end
```

```
for(a=300;a<400;a++)
```

```
for(b=0;b<480;b++)
```

```
map_put_pixel(0,fondo,a,b,rgb(255,255,0));
```

```
end
```

```
end
```

```
for(a=400;a<500;a++)
```

```
for(b=0;b<480;b++)
```

```
map_put_pixel(0,fondo,a,b,rgb(255,0,255));
```

```
end
```

```
end
```

```
for(a=500;a<600;a++)
```

```
for(b=0;b<480;b++)
```

```
map_put_pixel(0,fondo,a,b,rgb(0,255,255));
```

```
end
```

```
end
```

```
for(a=600;a<640;a++)
```

```
for(b=0;b<480;b++)
```

```

        map_put_pixel(0,fondo,a,b,rgb(255,255,255));
    end
end
put_screen(0,fondo);

/*Ahora genero el gráfico del proceso, el cual será un cuadradito negro que se moverá aleatoriamente
sobre el fondo de columnas de color*/
graph=new_map(10,10,16);
/*En vez de utilizar map_put_pixel para rellenar de color el nuevo gráfico, se puede ir más rápido y de
una tacada utilizando map_clear para pintar de golpe todo el gráfico -de color negro en este caso-*/
map_clear(0,graph,rgb(0,0,0));
write(0,10,10,3,"Color de la pantalla: ");
write_var(0,200,10,3,color);
/*Usamos un temporizador para situar el cuadradito en lugares distintos de la pantalla y dejarlo allí un
tiempo determinado antes de volverlo a mover otra vez.*/
timer=0;
repeat
if(timer>100)
    x=rand(0,639);
    y=rand(0,479);
    color=map_get_pixel(0,fondo,x,y);
    timer=0;
end
frame;
until(key(_esc))
end

```

Un ejemplo curioso donde se emplean *map\_put\_pixel* y *map\_get\_pixel* es el siguiente. Primero se visualiza como fondo de pantalla una imagen con código 001 perteneciente a un FPG llamado “graficos.fpg”. A continuación, se realiza un barrido horizontal donde se aplica un curioso efecto de **oscurecimiento**. Una vez acabado éste, se realiza otro barrido horizontal donde se aplica un curioso efecto de abrillantamiento.

Par lograr el oscurecimiento, lo único que se hace es convertir el color de todos aquellos píxeles de la imagen que tengan un cantidad de rojo,verde y azul menor de 15, al color negro más absoluto. Con esto, lo que estamos haciendo es oscurecer los píxeles que eran más oscuros ya de entrada, dejando inalterables los demás. Se puede jugar con el límite que queramos para incluir o excluir píxeles del oscurecimiento: a un mínimo más elevado, abrá más zonas de la imagen que se conviertan en negro. También se puede jugar con la condición del if: si en vez de haber un “<” hay un “>”, estaremos logrando el efecto contrario, también muy interesante.

Para lograr el abrillantamiento, se parte de la misma idea: se van recorriendo los píxeles de la imagen y si el color de éstos cumple una condición límite, su color se verá incrementado con el brillo que queremos definir (se puede variar su valor para comprobar el efecto que causa).

```

program imgx;
Private
    int c;
    int min=15,bri=50;
    int tt;
    int fpg;
end
begin
set_Mode(640,480,16);

```

```

fpg=Load_Fpg("graficos.fpg");
Set_Fps(1000,0);
put_Screen(fpg,1);
//Primer efecto: oscurecimiento
For (X=0;x<640;x++)
    For (y=0;y<480;y++)
        C=map_get_pixel(0,0,x,y);
        if (c<rgb(min,min,min)) c=Rgb(0,0,0); end
        map_put_Pixel(0,0,x,y,c);
    end
//La siguiente línea es para visualizar el efecto más velozmente
tt++;if(tt>5) tt=0;frame;end
end

//Segundo efecto: abrillantamiento
For (X=0;x<640;x++)
    For (y=0;y<480;y++)
        C=map_get_Pixel(0,0,x,y);
        if (C<rgb(255-bri,255-bri,255-bri)) c=c+Rgb(bri,bri,bri); end
        map_put_Pixel(0,0,x,y,c);
    end
tt++;if(tt>5) tt=0;frame;end
end
/*Si quisiéramos grabar en un fichero la imagen resultante de ambos efectos, podríamos escribir en este
lugar la línea: Save_png(0,0,"Hola.png"); La orden save_png se tratará más adelante*/
loop
frame;
end
end

```

Otro ejemplo de oscurecimiento, similar en la filosofía al anterior, es el siguiente. Necesitarás un FPG llamado "graficos.fpg" con un gráfico de identificador 001. Si ejecutas el ejemplo, primero verás la imagen tal cual; después de pulsar la tecla SPACE no verás nada (habrás quitado esta imagen del fondo) para entonces volver a pulsar la tecla SPACE y volver a ver la misma imagen de antes, pero con una zona rectangular en su interior donde aparecerá más oscurecida. El tamaño de esta porción de imagen oscurecida lo podrás cambiar (sólo) en el código fuente.

```

program imgx;
private
    int fpg;
end
begin
set_Mode(640,480,16);
fpg=Load_Fpg("graficos.fpg");
//Primero pongo la imagen original tal cual
put_Screen(fpg,1);
//Mientras no se pulse la tecla espacio no se hace nada; es decir, se visualiza la imagen de fondo y ya está
while(not key(_space)) frame; end
//Una vez que he pulsado la tecla SPACE, mientras la tenga pulsada tampoco hago nada
while(key(_space)) frame; end
//Una vez que he soltado la tecla SPACE, borro el fondo de pantalla, la cual se queda negra
clear_screen();
while(not key(_space)) frame; end

```

```

while(key(_space)) frame; end
//Después de haber pulsado otra vez SPACE, aplico la función "oscurecer" a la misma imagen de antes
oscurecer(fpg,1,30,30,500,500);
/*Y la vuelvo a mostrar como fondo de pantalla, observando las diferencias con el fondo anterior
(básicamente, la presencia de un cuadrado más oscuro)*/
put_screen(fpg,1);
loop
frame;
end
end

/*Esta función tiene 6 parámetros: el primero es el fpg donde se encuentra la imagen, (dada por el
segundo parámetro), que será oscurecida.No obstante, no tiene porqué oscurecerse toda la imagen, sino
sólo una porción rectangular de esta indicada por los cuatro parámetros siguientes:el tercero es la
coordenada X de la esquina superior izquierda de esta zona oscura, el cuarto es la coordenada Y de ese
punto, el quinto es la coordenada X de la esquina inferior derecha de esa zona oscura y el sexto es la
coordenada Y de ese punto. Hay que tener en cuenta que estas coordenadas tienen como origen la esquina
superior izquierda de la imagen, no de la pantalla.*/

function oscurecer(int fpg,int imagen,int x1,int y1,int x2,int y2)
private
    int i,j,color;
    byte r,g,b;
end
BEGIN
//Recorro la zona a oscurecer a lo largo y a lo ancho
FOR(i=x1; i<=x2; i++)
    FOR(j=y1; j<=y2; j++)
        //Obtengo el color original de cada pixel dentro de esta zona
        color=map_get_pixel(fpg,imagen,i,j);
/*Obtengo, a partir del color obtenido, sus componentes RGB individuales. La función get_rgb no la
hemos visto todavía (se verá en profundidad más adelante en este capítulo), pero básicamente sirve para
esto: a partir de un código único de color dado por map_get_pixel en este caso, obtenemos los valores
independientes de las componentes R,G y B en sendas variables (los tres últimos parámetros). Atención
con poner el & de cada uno de los nombres de estas variables.*/
        get_rgb (color, &r, &g, &b);
/*Repinto el mismo pixel del cual había obtenido el color original con un nuevo color generado a partir de
multiplicar por 0.6 (y por tanto, reducir) cada uno de los valores de las tres componentes del color
original. Con esto se logra el efecto de oscurecimiento. Si multiplicamos por un decimal menor
(0.5,0.4,etc), obtendremos cada vez valores de las componentes más pequeños y por tanto tenderemos
cada vez más a 0 -es decir, más negro-. Lo puedes probar.¿Qué pasaría si este factor fuera más grande de
1? Podríamos incluso modificar la función para que admitiera un nuevo parámetro que fuera
precisamente el valor de este factor, para tener una única función que permitiera generar diferentes tipos
de oscurecimientos. Un detalle importante: el hecho de utilizar map_put_pixel implica que la imagen se ha
modificado en memoria, pero de momento no es visible: necesitamos otra orden que haga que esta (nueva)
imagen se vea por pantalla: por eso en el código principal, después de haber llamado a "oscurecer",
volvemos a llamar a put_screen.*/
        map_put_pixel(fpg,imagen,i,j, rgb(r*0.6, g*0.6, b*0.6) );
    END
END
END

```

Si no se desea usar el gráfico por más tiempo, es recomendable liberar la memoria ocupada por

el mismo mediante la función *unload\_map/unload\_fbm*, función que tiene dos parámetros: el identificador del archivo FPG donde reside el gráfico a descargar (o 0 si se utiliza *write\_in\_map*, por ejemplo) y como segundo parámetro tiene el código numérico (el id) del gráfico a descargar.

#### UNLOAD\_MAP(librería,gráfico)/UNLOAD\_FBM(librería,gráfico)

Libera la memoria ocupada por un gráfico que puede formar parte de una librería FPG, haber sido creado independientemente por *new\_map*, o recuperado de disco por una función como *load\_png*. Los posteriores accesos al gráfico liberado son considerados un error.

Es posible utilizar esta función para quitar gráficos de un fichero FPG (lo contrario de *fpg\_add*). Los cambios realizados por esta función sólo afectan al FPG en memoria: para actualizar el fichero FPG con ellos recuerda que es preciso usar la función *save\_fpg*.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico dentro de la librería

#### WRITE\_IN\_MAP ( FUENTE, "TEXTO", ALINEACIÓN )

Esta función crea un nuevo gráfico en memoria, de la misma manera que la función *NEW\_MAP*, pero a partir de un texto.

El gráfico contendrá el texto especificado como parámetro, en el tipo de letra indicado, y tendrá el tamaño justo para contener todas las letras sin desperdiciar espacio. Formará parte de la librería del sistema, de código 0.

El parámetro de alineación indica el valor que será tomado por el centro del nuevo gráfico (punto de control cero), y puede ser uno de los siguientes valores:

- 0: el punto ocupará la esquina superior izquierda del gráfico
- 1: el punto ocupará el centro de la primera fila
- 2: el punto ocupará la esquina superior derecha
- 3: el punto ocupará el centro de la primera columna
- 4: el punto ocupará el centro exacto del gráfico
- 5: el punto ocupará el centro de la última columna
- 6: el punto ocupará la esquina inferior izquierda
- 7: el punto ocupará el centro de la fila inferior
- 8: el punto ocupará la esquina inferior derecha

El nuevo gráfico tendrá la misma profundidad de color que el tipo de letra, y pixels transparentes al igual que los propios caracteres de la fuente.

Esta función devuelve un identificador único para el nuevo gráfico creado, de forma similar por ejemplo a *load\_png*, y formará parte de la librería del sistema (FILE=0).

Si no se desea usar el gráfico por más tiempo, es recomendable liberar la memoria ocupada por el mismo mediante la función *UNLOAD\_MAP/UNLOAD\_FBM*.

La mayor utilidad de esta función consiste en poder dibujar textos con la misma variedad de efectos que los gráficos normales (escalado, transparencias, espejo, etc). Y también para poder asignar a la variable *GRAPH* de un proceso un texto, por ejemplo, con lo que un texto podrá asociarse a un proceso obteniendo así para ese texto todos los beneficios de la gestión de procesos (señales, colisiones, jerarquía, etc).

### PARÁMETROS:

INT FUENTE : Tipo de letra

STRING TEXTO : Texto a dibujar

INT ALINEACIÓN : Tipo de alineación

VALOR DE RETORNO: INT : Identificador del nuevo gráfico

Como ejemplo de utilización de *write\_in\_map*, puedes probar de ejecutar este código:

*Program ejemplo\_write\_in\_map;*

*Begin*

*set\_mode(640,480,16);*

*/\* Se crea un mapa nuevo con el texto "Esto es un ejemplo...1,2,3" con el punto de control en el centro, y se asigna como grafico del puntero del raton.\*/*

*graph = write\_in\_map(0,"Esto es un ejemplo...1,2,3",4);*

*x=320;*

*y=260;*

*Repeat*

*if(key(\_a)) angle=(angle+5000)%360000; end*

*if(key(\_s)) size=size + 10; end*

*Frame;*

*Until (key(\_esc));*

*unload\_map(0,graph);*

*End*

### **GRAPHIC\_INFO(librería,gráfico,tipo)**

Esta función permite consultar en tiempo de ejecución determinadas características de un gráfico, especialmente de su animación (si éste es un gráfico animado:GIF,APNG...), aunque también resulta útil para acceder al contenido del gráfico.

Las animaciones de los gráficos MAP son automáticas, e independientes de los frames por segundo establecidos por la función SET\_FPS .

### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico dentro de la librería

INT TIPO : Tipo de característica a consultar. Pueden ser:

G\_WIDTH :Ancho en pixels del gráfico

G\_HEIGHT :Alto en pixels del gráfico

G\_CENTER\_X :Coordenada horizontal del centro

G\_CENTER\_Y :Coordenada vertical del centro

G\_DEPTH :Profundidad de color (en bits por pixel)

G\_FRAMES Total de "versiones" del gráfico

G\_ANIMATION\_STEP Frame actual de la animación

G\_ANIMATION\_STEPS Total de frames de la animación

G\_ANIMATION\_SPEED Velocidad actual de la animación

G\_PITCH Diferencia en bytes, en memoria, entre 2 filas del gráfico

VALOR DEVUELTO: INT : Valor consultado

### MAP\_PUT ( LIBRERÍA, GRÁFICO, GRÁFICO-ORIGEN, X, Y )

Esta función dibuja un gráfico directamente sobre otro, utilizando unas coordenadas dentro del gráfico de destino, (como si éste fuese el fondo de pantalla), de manera que el primer gráfico quedará integrado parte constituyente del segundo

Una limitación de esta función es que ambos gráficos deben pertenecer a la misma librería. Habitualmente los gráficos de destino suelen ser gráficos temporales creados por NEW\_MAP , por lo que puede usarse como gráfico de origen una copia MAP\_CLONE de un gráfico de cualquier librería para paliar la limitación.

El punto especificado con los parámetros equivaldrá al lugar dentro del gráfico destino donde se dibuje el centro del gráfico origen

Si se especifica como librería el número 0 y como gráfico el número 0, el gráfico-origen se dibujará sobre el fondo de la pantalla.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico de destino dentro de la librería

INT GRÁFICO-ORIGEN : Número de gráfico a dibujar dentro de la librería

INT X : Coordenada horizontal

INT Y : Coordenada vertical

Un ejemplo sería éste (se necesitan dos archivos png: "a.png" y "b.png"). En él se "incrusta" dentro del gráfico "a.png" el gráfico "b.png", a cada frame y en distintas posiciones. Este nuevo gráfico formado por las múltiples incrustaciones de "b.png" en "a.png", si quisiéramos, lo podríamos grabar en disco, cosa que haremos cuando sepamos cómo hacerlo:

```
program Test_MAP_PUT;
begin
  set_mode(640,480,16);
  graph=load_png("a.png");
  x=320;
  y=240;
  repeat
    map_put(0,graph,load_png("b.png"),rand(0,319),rand(0,199));
  frame;
  until(key(_esc))
end
```

### MAP\_XPUT ( LIBRERÍA, GRÁFICO, GRÁFICO-ORIGEN, X, Y, ÁNGULO, TAMAÑO, FLAGS, REGIÓN )

Esta función dibuja un gráfico directamente sobre otro, utilizando unas coordenadas dentro del gráfico de destino, como si este fuese el fondo de pantalla. A diferencia de MAP\_PUT , esta función tiene toda la flexibilidad del dibujo normal de procesos, incluyendo ángulo,tamaño y recorte de una región.

Una limitación de esta función es que ambos gráficos deben perteneces a la misma librería. Esto no suele ser problema porque habitualmente los gráficos de destino suelen ser gráficos temporales creados por NEW\_MAP , por lo que puede usarse como gráfico de origen una copia MAP\_CLONE de un gráfico de cualquier librería para paliar la limitación. No obstante, se puede utilizar *map\_xputnp* si se desean utilizar

dos gráficos de diferentes librerías y obtener una funcionalidad similar.

El punto especificado con los parámetros equivaldrá al lugar donde se dibuje el centro del gráfico a dibujar.

Los flags son una suma o unión con el operador OR de los siguientes valores:

- 1: Dibuja el gráfico reflejado horizontalmente, como visto en un espejo.
- 2: Dibuja el gráfico reflejado verticalmente.
- 4: Dibuja el gráfico con transparencia.
- 128: Dibuja el gráfico sin tener en cuenta los pixels transparentes del mismo. Esta opción no es compatible con el flag 4.

La región corresponde a una limitación de coordenadas. Hay que tener en cuenta que normalmente una región se define respecto a la pantalla, y en el caso de esta función, los mismos límites se aplican a un gráfico de destino potencialmente más pequeño. Seguramente sea preciso definir nuevas regiones para usarlas específicamente con esta función.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico de destino dentro de la librería

INT GRÁFICO-ORIGEN : Número de gráfico a dibujar dentro de la librería

INT X : Coordenada horizontal

INT Y : Coordenada vertical

INT ÁNGULO : Ángulo del gráfico a dibujar en miligrados (90000 = 90°)

INT TAMAÑO : Tamaño del gráfico a dibujar, en porcentaje (100 = tamaño original)

INT FLAGS : Flags para el dibujo

INT REGION : Región de corte

Un ejemplo sería el siguiente (se necesitan dos archivos png: "a.png" y "b.png"). En él se "incrusta" dentro del gráfico "a.png" el gráfico "b.png", a cada frame y en distintas posiciones y con distinto ángulo, tamaño y flags. Este nuevo gráfico formado por las múltiples incrustaciones de "b.png" en "a.png", si quisiéramos, lo podríamos grabar en disco, cosa que haremos cuando sepamos cómo hacerlo:

```
program Test_MAP_XPUT;
global
    int f[] = 0, 1, 2, 3, 4, 5, 6, 7, 128;
end
begin
    set_mode(640, 480, 16);
    graph = load_png("a.png");
    x = 320;
    y = 240;
    repeat
        map_xput(0, graph, load_png("b.png"), rand(0, 639), rand(0, 239), rand(-pi*2, pi*2), rand(5, 100),
            rand(f[0], f[8]));
        frame;
    until(key(_esc))
end
```

**MAP\_XPUTNP(LIBRERIA, GRAFICO, LIBRERIA-ORIGEN, GRAFICO-ORIGEN, X, Y, ÁNGULO,**

## ESCALA-X, ESCALA-Y, FLAGS)

Esta función dibuja un gráfico (gráfico de origen) directamente sobre otro (gráfico de destino). Si los parámetros avanzados no son necesarios, se puede usar `map_put` o `map_xput` en su lugar. Si el ángulo es 0 y el tamaño es 100, la velocidad será mayor ya que el gráfico no necesita rotarse o escalarse.

Una gran diferencia práctica con `map_xput` es que `map_xputnp` permite (a diferencia de la primera) especificar como gráfico-origen y gráfico-destino dos gráficos pertenecientes a DIFERENTES librerías.

### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG que contiene el gráfico de destino

INT GRÁFICO : Número de gráfico de destino dentro de la librería

INT LIBRERÍA-ORIGEN : Número de librería FPG que contiene el gráfico de origen.

INT GRÁFICO-ORIGEN : Número de gráfico a dibujar dentro de la librería

INT X : Coord. horizontal del gráfico de destino donde se colocará el gráfico de origen

INT Y : Coord. vertical del gráfico de destino donde se colocará el gráfico de origen

INT ÁNGULO : Ángulo del gráfico a dibujar en miligrados (90000 = 90°)

INT ESCALA-X: Escala que define en el eje horizontal el tamaño del gráfico de origen

INT ESCALA-Y: Escala que define en el eje vertical el tamaño del gráfico de origen

INT TAMAÑO : Tamaño del gráfico a dibujar, en porcentaje (100 = tamaño original)

INT FLAGS : Flags para el dibujo

Un ejemplo de esta función podría ser:

```
program watskeburt;
Global
  int destgraph;
  int origgraph;
end
Begin
  set_mode(320,200,16);
  // El gráfico de destino será un cuadrado rojo
  destgraph=new_map(100,100,16);
  map_clear(0,destgraph,rgb(255,0,0));
  //El gráfico de origen será un cuadrado azul
  origgraph=new_map(100,100,16);
  map_clear(0,origgraph,rgb(0,0,255));
  /* Dibuja el cuadrado azul en el centro del cuadrado rojo transparentemente, en un tamaño y ángulo
  aleatorios*/
  map_xputnp(0,destgraph,0,origgraph,50,50,rand(-180000,180000),rand(50,150),rand(50,150),4);
  map_xputnp(0,destgraph,0,origgraph,50,50,rand(-180000,180000),rand(50,150),rand(50,150),4);
  // Muestra el resultado final
  map_put(0,0,destgraph,160,100);
Loop
  frame;
End
End
```

## PUT(LIBRERIA,GRAFICO,X,Y)

Esta función dibuja un gráfico en el fondo de pantalla, en las coordenadas especificadas como parámetro. El punto especificado equivaldrá al lugar donde se dibuje el centro del gráfico. Por lo tanto, añade un poco más de flexibilidad que *put\_screen*, la cual siempre coloca el centro del gráfico en el centro de la ventana del juego.

La misma funcionalidad se puede obtener con *map\_put* y sus dos primeros parámetros iguales a 0.

### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG  
INT GRÁFICO : Número de gráfico a dibujar  
INT X : Coordenada horizontal  
INT Y : Coordenada vertical

Como ejemplo de esta función, en vez de poner un fondo “estándar”, vamos a ser más originales y vamos a poner un fondo en mosaico. Se repetirá horizontal y verticalmente una imagen (de dimensiones bastante menores que la ventana) de manera que se cubra toda la extensión de éste, obteniendo ese efecto mosaico.

```
program hola;  
begin  
  set_mode(640,480,16);  
  fill_map();  
  loop  
    frame;  
  end  
end
```

```
function fill_map()  
private
```

```
  int idpng;  
  int ancho;  
  int alto;  
  int i,j;
```

```
end
```

```
begin
```

```
idpng=load_png("dibujo.png");  
ancho=graphic_info(0,idpng,g_width);  
alto=graphic_info(0,idpng,g_height);
```

*/\*El truco está en repetir la orden put a lo largo y ancho de la ventana, mediante dos bucles for: el primero la irá recorriendo "columna" a "columna" y el segundo, para cada una de esas "columnas", todas las "filas". Como las coordenadas donde la función put pinta -o sea, el tercer y cuarto parámetro- corresponden al centro del gráfico en cuestión, los valores de los dos bucles tendrán que valer las coordenadas de las diferentes posiciones del centro de ese gráfico a lo largo de toda la ventana para cubrirla entera, y sin solaparse. Por eso, el primer valor del primer bucle es "ancho/2", porque la coordenada horizontal del centro del primer gráfico a dibujar (el de la esquina superior izquierda) es esa, y por eso el primer valor del segundo bucle es "alto/2", por la misma razón. Para buscar la coordenada del siguiente centro es fácil: si se busca el siguiente centro horizontal, es sumar el ancho del gráfico a la posición del centro actual, y similarmente si buscamos el siguiente centro vertical: sumando la posición del alto del gráfico a la posición del centro actual. Ésos son los valores del paso de los dos bucles. Y el significado de la condición de salida de los fors se basa en detectar cuándo ya no hace falta seguir pintando gráficos tanto horizontalmente como verticalmente pose se ha llegado al final de la ventana. La condición viene de que si la coordenada del siguiente centro es mayor que el extremo de la ventana (más*

un último ancho/alto para forzar a que en los límites de la ventana se pinten los gráficos aunque estén recortados) ya no hace falta seguir pintando por allí.

Este sistema de hacer mosaicos en un tema posterior, en el último capítulo, se tratará más en profundidad, ya que es bastante importante para el uso de los llamados "tiles"\*/

```
for(i=ancho/2;i<800+ancho;i=i+ancho)
    for(j=alto/2;j<600+alto;j=j+alto)
        put(0,idpng,i,j);
    end
end
end
```

### **XPUT(LIBRERÍA, GRÁFICO,X,Y,ÁNGULO,TAMAÑO,FLAGS,REGION):**

Esta función dibuja un gráfico en el fondo de pantalla, en las coordenadas especificadas como parámetro, y aplicando todas las posibilidades de dibujo de un proceso (ángulo de rotación, transparencias, escalado...). El punto especificado equivaldrá al lugar donde se dibuje el centro del gráfico, y a partir de cual se rotará o cambiará de tamaño.

La misma funcionalidad se puede obtener con *map\_xput* y sus dos primeros parámetros iguales a 0

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG  
INT GRÁFICO : Número de gráfico a dibujar  
INT X : Coordenada horizontal  
INT Y : Coordenada vertical  
INT ÁNGULO : Ángulo en miligrados (90000 = 90°)  
INT TAMAÑO : Tamaño en porcentaje (100 = tamaño original)  
INT FLAGS : Flags para el dibujo  
INT REGION : Región de recorte (0 = ninguna)

Un ejemplo trivial para ver su utilidad (necesitarás un gráfico llamado "cosa.png"):

```
program lala;
private
    int idpng;
end
begin
    set_mode(640,480,16);
    idpng=load_png("cosa.png");
    xput(0,idpng,120,240,0,50,1,0);
    xput(0,idpng,520,240,90000,200,4,0);
    xput(0,idpng,60,400,180000,100,0,1);
loop
    if(key(_esc)) exit(); end
frame;
end
end
```

### **MAP\_CLONE ( LIBRERÍA, GRÁFICO )**

Esta función crea una copia en memoria de un gráfico cualquiera. El nuevo gráfico se crea como perteneciente a la librería 0, y Fénix asigna un número nuevo disponible para el mismo. El valor de retorno de esta función corresponde al número del nuevo gráfico, y debe recogerse para poder utilizarse en

cualquier operación gráficas.

Si lo que se desea es crear un gráfico nuevo dentro de una librería FPG distinta de la del sistema, entonces es preferible usar la función FPG\_ADD .

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico dentro de la librería

Un ejemplo:

```
program Test_MAP_CLONE;
global
  int orimap,clonemap,clonemap2;
end
begin
  set_mode(640,480,16);
  write(0,80,50,4,"Original ");
  orimap=new_map(32,20,16);
  map_clear(0,orimap,rgb(127,127,127));
  map_put(0,0,orimap,80,100);
  write(0,240,50,4,"Clon 1");
  clonemap=map_clone(0,orimap);
  map_put(0,0,clonemap,240,100);
  write(0,400,50,4,"Clon 2");
  clonemap2=map_clone(0,clonemap);
  map_put(0,0,clonemap2,400,100);

  repeat
    frame;
  until(key(_esc))
end
```

Una utilidad práctica de esta función es partir de un gráfico inicial, el cual se clona, y modificar el gráfico clonado como se desee (por ejemplo con las funciones de la familia *map\_put...*), de tal forma que obtengamos un nuevo gráfico, levemente diferente al original en tiempo real sin tener la necesidad de dibujarlo explícitamente y guardarlo en un FPG como un elemento más,por ejemplo. El siguiente código muestra esto:

```
program Test_MAP_CLONE;
global
  int grancuad,pequecuad,cloncuad,cloncuad2;
end
begin
  set_mode(640,480,16);
  //Genero el cuadrado grande amarillo (pero no lo muestro todavía)
  grancuad=new_map(50,50,16);
  map_clear(0,grancuad,rgb(255,255,0));
  //Genero el cuadrado pequeño rojo (pero no lo muestro todavía)
  pequecuad=new_map(10,10,16);
  map_clear(0,pequecuad,rgb(255,0,0));
  //Dibujo en pantalla el cuadrado amarillo original
  map_put(0,0,grancuad,80,100);
```

```

//Genero una copia en memoria del cuadrado amarillo...
cloncuad=map_clone(0,grancuad);
//...y sobre él pinto el cuadrado rojo...
map_put(0,cloncuad,pequecuad,20,20);
//...y muestro el resultado en pantalla
map_put(0,0,cloncuad,180,100);

//Repito el proceso a partir del primer clon, para crear un segundo
cloncuad2=map_clone(0,cloncuad);
map_put(0,cloncuad2,pequecuad,40,40);
map_put(0,0,cloncuad2,280,100);

repeat
  frame;
until(key(_esc))
end

```

Incluso podemos utilizar esta misma idea con procesos, El siguiente código es una ligera modificación del anterior:

```

program Test_MAP_CLONE;
global
  int grancuad,pequecuad,cloncuad,cloncuad2;
end
begin
  set_mode(640,480,16);
  //Genero el cuadrado grande amarillo (pero no lo muestro todavía)
  grancuad=new_map(50,50,16);
  map_clear(0,grancuad,rgb(255,255,0));
  //Genero el cuadrado pequeño rojo (pero no lo muestro todavía)
  pequecuad=new_map(10,10,16);
  map_clear(0,pequecuad,rgb(255,0,0));

  //Este proceso tendrá un gráfico asociado compuesto por incrustación del cuadrado rojo en el amarillo
  miproceso(0,grancuad,pequecuad);

  /*Podemos comprobar cómo el gráfico "grancuad" permanece inalterado, porque el map_put del
  proceso anterior lo hemos hecho sobre un clon suyo*/
  put_screen(0,grancuad);
  repeat
    frame;
  until(key(_esc))
end

Process miproceso(int fpg, int grafgrand, int grafpeque)
private
  int grafclonado;
end
Begin
grafclonado = map_clone(fpg,grafgrand);
map_put(fpg,grafclonado,grafpeque,20,20);
graph=grafclonado;
y=320;x=240;
Loop

```

```

        if(key(_left)) x=x-10; end
        if(key(_right)) x=x+10; end
    frame;
End
End

```

Finalmente, como ejercicio, intenta entender lo que ocurre cuando ejecutamos el siguiente código (necesitarás una imagen llamada “dibujo.png”):

```

program mapacome;
private
    int graforig,grafdest;
end
begin
set_mode(640,480,16);
graforig=load_png("dibujo.png");
grafdest=paintit(graforig);
x=320;y=240;
graph=graforig;
loop
    if(key(_o)) graph=graforig; end
    if(key(_d)) graph=grafdest; end
    frame;
end
end

function Paintit(int grafico)
private
    int map,map2;
end
begin
map=Map_clone(0,grafico);
map2=New_map(graphic_info(0,grafico,g_width),graphic_info(0,grafico,g_height),16);
map_xput(0,map2,map,0,0,0,200,0);
unload_map(0,map);
return(map2);
end

```

### **MAP\_BLOCK\_COPY ( LIBRERÍA, GRÁFICO, X-DESTINO, Y-DESTINO, GRÁFICO-ORIGEN, X-ORIGEN, Y-ORIGEN, ANCHO, ALTO, FLAGS )**

Esta función copia una sección de un gráfico delimitada por unas coordenadas de origen (para la esquina superior izquierda de la zona), un ancho y un alto, a un gráfico de destino, posiblemente en otras coordenadas.

Gracias al valor de flags indicado es posible determinar si los pixels transparentes en el gráfico de origen se ignoran (flags 0) o se copian también (flags 128).

La única limitación es que ambos gráficos deben formar parte de la misma librería. Si se desea realizar la operación con gráficos en librerías diferentes, la alternativa es crear una copia de uno de los dos gráficos mediante FPG\_ADD o MAP\_CLONE

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG  
 INT GRÁFICO : Número de gráfico de destino dentro de la librería  
 INT X-DESTINO : Coordenada horizontal de destino  
 INT Y-DESTINO : Coordenada vertical de destino  
 INT GRÁFICO-ORIGEN : Número de gráfico de origen dentro de la librería  
 INT X-ORIGEN : Coordenada horizontal dentro del gráfico de origen  
 INT Y-ORIGEN : Coordenada vertical dentro del gráfico de origen  
 INT ANCHO : Ancho en pixels del bloque a copiar  
 INT ALTO : Alto en pixels del bloque a copiar  
 INT FLAGS : Valor de flags

Un ejemplo autoexplicativo:

```

program prioridades;
const
    //Ancho de la imagen sobre la cual se copiará -parte de- otra imagen diferente ("fuente")
    anchoDestino=50;
    //Alto de la imagen sobre la cual se copiará -parte de- otra imagen diferente ("fuente")
    altoDestino=50;
    anchoFuente=25; //Ancho de la imagen que será copiada sobre otra imagen ("destino")
    altoFuente=25; //Alto de la imagen que será copiada sobre otra imagen ("destino")
end
process main()
begin
    set_mode(640,480,16);
    procesol();
    loop if(key(_esc)) exit();end frame; end
end

process procesol()
private
    int graf1;
end
begin
    //Imagen "destino": cuadrado de color rojo de 50x50
    graph = new_map(anchoDestino,altoDestino,16); map_clear(0,graph,rgb(255,0,0));x=320;y=240;
    //Imagen "fuente" que se copiará -al menos, parte- sobre "destino": cuadrado de color amarillo de 25x25
    graf1 = new_map(anchoFuente,altoFuente,16);map_clear(0,graf1,rgb(255,255,0));
    frame(500);
    /*Pongo "fuente" en "destino" de manera que la coordenada (0,0) de "fuente" coincida con la coordenada
    (0,0) de "destino",y copio todo el gráfico de "fuente", sin ningún flag*/
    map_block_copy(0,graph,0,0,graf1,0,0,anchoFuente,altoFuente,0);
    frame(500);
    /*Pongo "fuente" en "destino" de manera que la coordenada (0,0) de "fuente" coincida con la coordenada
    (anchoDestino/2,altoDestino/2) de "destino",y copio todo el gráfico de "fuente", sin ningún flag*/
    map_block_copy(0,graph,anchoDestino/2,altoDestino/2,graf1,0,0,anchoFuente,altoFuente,0);
    frame(500);
    /*Pongo "fuente" en "destino" de manera que la coordenada (0,0) de "fuente" coincida con la coordenada
    (anchoDestino/2,0) de "destino",pero copio sólo una parte del gráfico de "fuente" (en concreto, un
  
```

```

cuarto de la imagen original ya que el bloque está definido por (anchoFuente/2,altoFuente/2)), sin ningún
flag*/
map_block_copy(0,graph,anchoDestino/2,0,graf1,0,0,anchoFuente/2,altoFuente/2,0);
frame(500);
/*Pongo "fuente" en "destino" de manera que la coordenada (0,0) de "fuente" coincida con la coordenada
(0,altoDestino/2) de "destino",pero copio sólo una parte del gráfico de "fuente" (en concreto, un
novo de la imagen original ya que el bloque está definido por (anchoFuente/3,altoFuente/3)), con un
flag de transparencia*/
map_block_copy(0,graph,0,altoDestino/2,graf1,0,0,anchoFuente/3,altoFuente/3,4);
loop frame; end
end

```

A continuación pongo otro código de ejemplo similar al anterior, pero en donde las copias de los bloques de la imagen “fuente” se realizan de forma aleatoria, tanto en posición como en tamaño, consiguiendo un efecto curioso, al menos. Para poderlo ejecutar se necesitarán dos imágenes PNG de nombre “a.png” y “b.png”, de un tamaño de 640x480 o más:

```

Program ejemplo_map_block_copy;
const
//Tamaños de las imágenes
XMAX=640;
YMAX=480;
end

```

```

Private
int x_destino, y_destino;
int x_origen, y_origen;
int ancho, alto;
int png1, png2;
end

```

```

Begin
set_mode(640,480,16);
png1=load_png("a.png");
png2=load_png("b.png");
//Visualizamos al fondo, de forma centrada, la primera imagen.
graph=png1;
x=XMAX/2;
y=YMAX/2;

```

```

Loop
/*Definimos el punto de la segunda imagen a partir del cual,estableciendo su alto y ancho, crearemos el
bloque que posteriormente copiaremos sobre la primera imagen. Evidentemente, esta posición ha de
pertenecer a la segunda imagen,por lo que su valor ha de estar dentro del tamaño de ésta (en este caso,
640x480)*/
x_origen=rand(0,XMAX);
y_origen=rand(0,YMAX);
/*Definimos el tamaño del bloque de imagen. Con el ancho, el alto, y el punto de origen definido en las
dos líneas anteriores,ya tenemos establecido la porción de la segunda imagen que se colocará sobre la
primera. Ahora sólo falta decidir dónde*/
ancho=rand(1, 64);
alto=rand(1, 64);
/*Definimos la posición dentro de la primera imagen a parte de la cual se colocará el bloque establecido
en las líneas anteriores.Evidentemente, esta posición ha de pertenecer a la primera imagen, por lo que su

```

```

valor ha de estar dentro del tamaño de ésta (en este caso, 640x480)*/
  x_destino=rand(0, XMAX);
  y_destino=rand(0, YMAX);
/*Ésta es la orden que realiza todo el proceso explicado hasta ahora.De hecho, lo único que habíamos
hecho hasta aquí era declarar variables,que son las que se utilizan en este comando*/
  map_block_copy(0,png1, x_destino, y_destino, png2, x_origen, y_origen, ancho, alto,0);
Frame;
//angle=angle+1000;
If(key(_esc)) Break; End
End
End

```

Con la misma imagen “a.png” del ejemplo anterior podemos probar también este otro ejemplo (original de Carles)de uso de la función *map\_block\_copy*, (un poco más complejo), con el que se obtiene un logrado efecto de **ondulación**..:

```

program ondular;
begin
  set_mode(640,480,16);
  write_var(0,0,0,0,fps);
  ondular(320,240,0,alpha,load_png("a.png"),20,5000,5000);
end

process ondular(x,y,z,alpha,grafico,radio,omega,delta)
private
  int angulo,altura,desfase;
end
begin
  graph=new_map(radio*2+graphic_info(0,grafico,g_width),graphic_info(0,grafico,g_height),16);
  loop
    angulo=desfase;
    from altura=0 to graphic_info(0,grafico,g_height)
      map_block_copy(0,graph,radio+get_distx(angulo,radio),altura,grafico,0,altura,graphic_info(0,gr
afico,g_width),1,128);
      angulo=angulo+omega;
    end
    desfase=desfase+delta;
    if(angulo>360000) angulo=0; end
    if(desfase>360000) desfase=0; end
    frame;
  end
end
end

```

Con las funciones vistas hasta ahora se pueden lograr efectos visuales muy interesantes, como por ejemplo realizar **transiciones** de diferentes tipos entre imágenes.

El siguiente código de ejemplo (original de Pixel) consiste en pasar de una imagen a otra mediante una cortina horizontal. Para probarlo has de utilizar las dos imágenes PNG de los ejemplos anteriores: “a.png” y “b.png”.Una vez probado el ejemplo, si te gusta, para poderlo utilizar en tu juego o programa, deberás copiar a tu código los 2 procesos : “transicion()” y “pixel()”. Y cuando quieras utilizar el efecto, tendrás que escribir: *transicion(int grafico1,int grafico2,int velocidad)*; ¡No te olvides de hacer pruebas!

```

Program transition;
Private
    int ancho_mapa;
    int alto_mapa;
end

Begin
    set_mode(640,480,16);
    set_fps(60,0);
    //iniciamos la transición
    transicion(load_png("a.png"),load_png("b.png"),10);
/* descomentar lo de abajo si en vez de otro gráfico queremos que se vuelva negra la pantalla*/
//    transicion(load_png("./1.png"),0,10);
End

/*Estos dos procesos son los que tienes que copiar a tu juego o programa si quieres usar este efecto */
Process transicion(fixer01,fixero2,velocidad)
Private
    int ancho_mapa;
    int alto_mapa;
    int tipo=0;
end
Begin
    ancho_mapa=graphic_info(0,fixero1,g_width);
    alto_mapa=graphic_info(0,fixero1,g_height);
    While(y<alto_mapa/2+1)
        If(tipo==1) tipo=0; Else tipo=1; End
        pixel(fixer01,y,ancho_mapa,velocidad,tipo);
        pixel(fixer01,alto_mapa-y,ancho_mapa,velocidad,tipo);
        y++;
        Frame(velocidad*3);
    End
    timer[0]=0;
    While(timer[0]<300)
        Frame;
    End
    y=0; x=0;
    If(fixer02==0) fixero2=new_map(640,480,8); End
    While(y<alto_mapa/2+1)
        If(tipo==1) tipo=0; Else tipo=1; End
        pixel(fixer02,y,ancho_mapa,velocidad,tipo);
        pixel(fixer02,alto_mapa-y,ancho_mapa,velocidad,tipo);
        y++;
        Frame(velocidad*3);
    End
    While(NOT(key(_esc)))
        Frame;
    End
End

/*este proceso que no se te olvide, van juntos*/
Process pixel(fixer0,y,ancho_mapa,velocidad,tipo);
Private
    color;

```

*Begin*

```
graph=new_map(1,1,8);  
drawing_map(0,graph); //Las funciones draw_ * las veremos en el siguiente apartado  
drawing_color(254);  
draw_fcircle(0,0,1);  
If(tipo==0)
```

```
    While(x<ancho_mapa-1)  
        x++;  
        color=map_get_pixel(0,fixero,x,y);  
        map_put_pixel(0,father.graph,x,y,color);  
        Frame(velocidad);
```

*End*

*End*

```
If(tipo==1)
```

```
    x=ancho_mapa;  
    While(x>1)  
        x--;  
        color=map_get_pixel(0,fixero,x,y);  
        map_put_pixel(0,father.graph,x,y,color);  
        Frame(velocidad);
```

*End*

*End*

*End*

Otra transición que se puede conseguir es la “cascada”. Un ejemplo (original de “La momia que fuma”):

*Program efectillo;*

*Const*

*tam=5;*

*//cambiar para modificar el tamaño de los "fideos", cuanto mas grandes mejor rendimiento*

```

end
Global
  Byte empieza;
  int fondo1;
  int fondo2;
end
Private
  int xf=tam;
end
Begin
  // full_screen=1;
  set_mode(640,480,16);
  fondo1=load_png("a.png");
  fondo2=load_png("b.png");
  set_center(0,fondo2,0,0);
  put_screen(0,fondo1);
  graph=fondo2;//se puede comentar esta linea para ver como se van generando los "fideos"
  z=100;
  flags=128;//en teoria al usar este flag (prescindir de transparencia) el gráfico se dibuja más rápido
  While(xf>-690)
    xf=xf-tam;
    fideo(xf);
    Frame;
  End
  empieza=1;
End

Process fideo(cual)
  Private int vy; end

  Begin
    flags=128;
    y=240;x=-cual+(tam/2);
    z=-100;
    vy=rand(0,10);
    graph=new_map(tam,480,16);
    map_put(0,graph,fondo2,cual,0);
  Repeat
    Frame;
  Until(empieza)
  Repeat
    y=y+vy;
    vy++;
    Frame;
  Until(out_region(id,0)) //Hasta que el gráfico salga fuera de la pantalla. Consultar función out_region
  unload_map(0,graph);
End

```

Finalmente, aquí transcribo un código fuente que muestra el funcionamiento de cinco efectos: espiral, persiana horizontal, persiana vertical, cortina horizontal y cortina vertical. La idea del código es que los procesos que corresponden a cada uno de estos efectos sean inmediatamente transportables a otros códigos fuente que podamos hacer, de manera que tengamos esta funcionalidad ya lista para funcionar.

Si te fijas, para conseguir estos efectos las únicas funciones que hemos necesitado han sido

*map\_get\_pixel()* y *put\_pixel()* (y *graphic\_info()*).

Para poderlos visualizar necesitarás tener un FPG llamado “graficos.fpg” con dos gráficos en su interior, de códigos 001 y 002 (es aconsejable que el gráfico 002 sea de dimensiones bastante menores que 001, para una mayor comodidad en la visualización del efecto):

```
/* Autor: Pedro Tomás Matilla ( !Deemo )
   sp_put_ESPIRAL      ( fpg, grafico, x, y, modo );
   sp_put_PERSIANA_HORIZ ( fpg, grafico, x, y, modo );
   sp_put_PERSIANA_VERT ( fpg, grafico, x, y, modo );
   sp_put_CORTINA_HORIZ ( fpg, grafico, x, y, modo, velocidad );
   sp_put_CORTINA_VERT  ( fpg, grafico, x, y, modo, velocidad ); */
Program sp_put;

begin
  set_fps(25, 0);
  set_mode(800,600,16);
  load_fpg("graficos.fpg");
  loop
  //El while posibilita la visualización de los efectos
    sp_put_ESPIRAL      (0, 1, 50, 20, 0); timer=0; while(timer<200) frame; end
    sp_put_ESPIRAL      (0, 2, 60, 40, 1); timer=0; while(timer<200) frame; end
    sp_put_PERSIANA_HORIZ (0, 1, 50, 20, 0); timer=0; while(timer<200) frame; end
    sp_put_PERSIANA_HORIZ (0, 2, 60, 40, 1); timer=0; while(timer<200) frame; end
    sp_put_PERSIANA_VERT (0, 1, 50, 20, 0); timer=0; while(timer<200) frame; end
    sp_put_PERSIANA_VERT (0, 2, 60, 40, 1); timer=0; while(timer<200) frame; end
    sp_put_CORTINA_HORIZ (0, 1, 50, 20, 0, 10); timer=0; while(timer<200) frame; end
    sp_put_CORTINA_HORIZ (0, 2, 60, 40, 1, 10); timer=0; while(timer<200) frame; end
    sp_put_CORTINA_VERT  (0, 1, 50, 20, 0, 10); timer=0; while(timer<200) frame; end
    sp_put_CORTINA_VERT  (0, 2, 60, 40, 1, 10); timer=0; while(timer<200) frame; end
  end
end

/-- Efecto 1: - Fundido en espiral -----=oOo===== by !Deemo -<

process sp_put_ESPIRAL(fpg, grafico, x, y, modo)
private
  int alto, ancho, gx, gy, c, pixel, ini, fin, inc;
  int Espiral[49]= 0,0, 1,0, 2,0, 3,0, 4,0, 4,1, 4,2, 4,3,4,4, 3,4, 2,4, 1,4, 0,4, 0,3, 0,2, 0,1,1,1, 2,1, 3,1, 3,2,
  3,3, 2,3, 1,3, 1,2, 2,2;
end
begin
  alto= graphic_info(fpg, grafico, g_height);
  ancho= graphic_info(fpg, grafico, g_width);

  if(modo==0)
    ini=0; fin=50; inc=2;
  else
    ini=48; fin=-2; inc=-2;
  end

  for(c=ini; c<>fin; c=c+inc)
    for(gy=0; gy<alto; gy=gy+5)
      for(gx=0; gx<ancho; gx=gx+5)
```

```

        pixel= map_get_pixel(fpg, grafico, gx+Espiral[c], gy+Espiral[c+1]);
        if(pixel<>0) put_pixel(x+gx+Espiral[c], y+gy+Espiral[c+1], pixel); end
    end
end
frame;
end
end

```

*//-- Efecto 2: - Persiana horizontal -----==oOo==----- by !Deemo -<*

```

process sp_put_PERSIANA_HORIZ(fpg, grafico, x, y, modo)

```

```

private

```

```

    int alto, ancho, gx, gy, c, pixel, ini, fin, inc;

```

```

end

```

```

begin

```

```

    alto= graphic_info(fpg, grafico, g_height);

```

```

    ancho= graphic_info(fpg, grafico, g_width);

```

```

    if(modo==0)

```

```

        ini=0; fin=5; inc=1;

```

```

    else

```

```

        ini=5; fin=0; inc=-1;

```

```

    end

```

```

    for(c=ini; c<>fin; c=c+inc)

```

```

        for(gy=0; gy<alto; gy++)

```

```

            for(gx=0; gx<ancho; gx=gx+5)

```

```

                pixel= map_get_pixel(fpg, grafico, gx+c, gy);

```

```

                if(pixel<>0) put_pixel(x+gx+c, y+gy, pixel); end

```

```

            end

```

```

        end

```

```

    frame;

```

```

end

```

```

end

```

*//-- Efecto 3: - Persiana vertical -----==oOo==----- by !Deemo -<*

```

process sp_put_PERSIANA_VERT(fpg, grafico, x, y, modo)

```

```

private

```

```

    int alto, ancho, gx, gy, c, pixel, ini, fin, inc;

```

```

end

```

```

begin

```

```

    alto= graphic_info(fpg, grafico, g_height);

```

```

    ancho= graphic_info(fpg, grafico, g_width);

```

```

    if(modo==0)

```

```

        ini=0; fin=5; inc=1;

```

```

    else

```

```

        ini=5; fin=0; inc=-1;

```

```

    end

```

```

    for(c=ini; c<>fin; c=c+inc)

```

```

for(gy=0; gy<alto; gy=gy+5)
  for(gx=0; gx<ancho; gx++)
    pixel= map_get_pixel(fpg, grafico, gx, gy+c);
    if(pixel<>0) put_pixel(x+gx, y+gy+c, pixel); end
  end
end
frame;
end
end

/-- Efecto 4: - Cortina horizontal -----==oOo==----- by !Deemo -<

process sp_put_CORTINA_HORIZ(fpg, grafico, x, y, modo, velocidad)
private
  int alto, ancho, gx, gy, c, pixel, ini, fin, inc;
end
begin
  alto= graphic_info(fpg, grafico, g_height);
  ancho= graphic_info(fpg, grafico, g_width);

  if(modo==0)
    ini=0; fin=ancho+1; inc=1;
  else
    ini=ancho-1; fin=0; inc=-1;
  end

  for(gx=ini; gx<>fin; gx=gx+inc)
    for(gy=0; gy<alto; gy=gy+2)
      pixel= map_get_pixel(fpg, grafico, gx, gy);
      if(pixel<>0) put_pixel(x+gx, y+gy, pixel); end
      pixel= map_get_pixel(fpg, grafico, ancho-gx, gy+1);
      if(pixel<>0) put_pixel(x+ancho-gx, y+gy+1, pixel); end
    end
    frame(velocidad);
  end
end

/-- Efecto 5: - Cortina vertical -----==oOo==----- by !Deemo -<

process sp_put_CORTINA_VERT(fpg, grafico, x, y, modo, velocidad)
private
  int alto, ancho, gx, gy, c, pixel, ini, fin, inc;
end
begin
  alto= graphic_info(fpg, grafico, g_height);
  ancho= graphic_info(fpg, grafico, g_width);

  if(modo==0)
    ini=0; fin=alto+1; inc=1;
  else
    ini=alto-1; fin=0; inc=-1;
  end
end

```

```

for(gy=ini; gy<>fin; gy=gy+inc)
  for(gx=0; gx<ancho; gx=gx+2)
    pixel= map_get_pixel(fpg, grafico, gx, gy);
    if(pixel<>0) put_pixel(x+gx, y+gy, pixel); end
    pixel= map_get_pixel(fpg, grafico, gx+1, alto-gy);
    if(pixel<>0) put_pixel(x+gx+1, y+alto-gy, pixel); end
  end
  frame(velocidad);
end
end
end

```

Como ejemplo de lo que se puede llegar a conseguir gráficamente sólomente con el uso de las funciones que acabamos de ver, y sin la ayuda de ninguna imagen externa, a continuación pongo el código de un ping-pong (original de Pixel). Su lógica todavía no está acabada (la pelota no hace más que rebotar horizontalmente) pero lo interesante no es el juego en sí sino el acabado gráfico resultante, obtenido en tiempo real de ejecución del juego. Como valor añadido además, el código está escrito de tal manera que es totalmente escalable, ya que todos los gráficos son proporcionales a la resolución señalada en cada momento: así, si se indica una resolución más pequeña, de forma automática las palas, la bola, los marcadores, etc amoldarán su nueva posición acordemente, y si la resolución es mayor, igual. Merece mucho la pena echarle una ojeada.

*Program burrada;*

*Const*

*blanco=15;*

*azul=55;*

*rojo=24;*

*gris=6;*

*grisoscuro=3;*

*res\_x=800;*

*res\_y=640;*

*End*

*Global*

*int altura\_marcadores;*

*int filas;*

*int columnas;*

*int tamanocelda; //20 en 800x600*

*int efepeese=60;*

*int turno; //1: jugador izquierda, 2: jugador derecha*

*int graff[5];*

*int velocidad=3; //Velocidad de la bola*

*End*

*Begin*

*if(res\_x>400) set\_mode(res\_x,res\_y,16); else set\_mode(res\_x,res\_y,16,mode\_2xscale); end*

*set\_fps(efepeese,9);*

*tamanocelda=res\_x\*0.025; //800 = 20*

*altura\_marcadores=res\_y/6;*

*filas=(res\_y-altura\_marcadores)/tamanocelda;*

*columnas=res\_x/tamanocelda;*

*mouse.graph=graff[3];*

*hazgraficos();*

*pintafondo();*

*bola();*

```

palo1(res_x/10,res_y/2, graf[0]);
palo2(9*res_x/10,res_y/2, graf[1]);
senalaturno();
End

Process hazgraficos()
Begin
//palo1
graf[0]=new_map(tamanocelda,tamanocelda*4,8);
drawing_map(0,graf[0]);
drawing_color(rojo);
draw_box(0,0,tamanocelda,tamanocelda*4);
drawing_color(rojo-2);
draw_rect(0,0,tamanocelda,tamanocelda*4);
//palo2
graf[1]=new_map(tamanocelda,tamanocelda*4,8);
drawing_map(0,graf[1]);
drawing_color(azul);
draw_box(0,0,tamanocelda,tamanocelda*4);
drawing_color(azul-2);
draw_rect(0,0,tamanocelda,tamanocelda*4);
//bola
graf[2]=new_map(tamanocelda*1.5,tamanocelda*1.5,8);
drawing_map(0,graf[2]);
drawing_color(blanco);
draw_fcircle(tamanocelda/1.5,tamanocelda/1.5,tamanocelda/1.5);
drawing_color(gris);
draw_circle(tamanocelda/1.5,tamanocelda/1.5,tamanocelda/1.5);
//raton
graf[3]=new_map(tamanocelda/2,tamanocelda/2,8);
drawing_map(0,graf[3]);
drawing_color(blanco);
draw_box(0,0,1,1);
draw_box(0,0,tamanocelda/2,tamanocelda/6);
draw_box(0,0,tamanocelda/6,tamanocelda/2);
draw_box(0,0,tamanocelda/4,tamanocelda/4);
End

Process pintafondo()
Private
    int i;
End
Begin
graph=new_map(res_x,res_y,8);
drawing_map(0,graph);
drawing_color(gris);
// caja marcadores
draw_box(0,0,res_x,altura_marcadores);
drawing_color(grisoscuro);
draw_rect(0,0,res_x,altura_marcadores);
// celda fondo
drawing_color(grisoscuro);
write(0,res_x/4,altura_marcadores - 10,4,"Marcador 1");

```

```

write(0,3*res_x/4,altura_marcadores - 10,4,"Marcador 2");
//filas
from i=0 to filas

draw_line(0,altura_marcadores+(i*tamanocelda),res_x,altura_marcadores+(i*tamanocelda));
end
//columnas
from i=0 to columnas
    draw_line(tamanocelda*i,altura_marcadores+1,tamanocelda*i,res_y);
end
//linea vertical del centro
drawing_color(gris);
draw_box((res_x/2)-2,altura_marcadores,(res_x/2)+2,res_y);
//linea roja vertical
drawing_color(rojo);
draw_box(0,altura_marcadores,res_x,altura_marcadores+1);
//linea azul vertical
drawing_color(azul);
draw_box(0,res_y-2,res_x,res_y);
//y lo ponemos de fondo
put_screen(0,graph);
End

Process bola()
private
    int ancho;
    int alto;

end
Begin
graph=graf[2];
x=res_x/2;
y=((res_y-altura_marcadores)/2)+altura_marcadores;
ancho=graphic_info(file,graph,g_width);
alto=graphic_info(file,graph,g_height);
loop
    if(x<res_x/2) turno=1;end
    if(x>res_x/2) turno=2;end
    if(collision( type palo1) or collision (type palo2))
        angle=(angle+180000)%360000;
        advance(velocidad);
    end
    advance(velocidad);
    sombra();
    frame;
end
End

Process palo1(x,y,graph)
private
    int ancho;
    int alto;

end
Begin
    ancho=graphic_info(file,graph,g_width);

```

```

        alto=graphic_info(file,graph,g_height);
        loop
            if(key(_down)) y=y+2; end
            if(key(_up)) y=y-2;end
            sombra();
            frame;
        end
    End

    Process palo2(x,y,graph)
    private
        int ancho;
        int alto;
    end
    Begin
        ancho=graphic_info(file,graph,g_width);
        alto=graphic_info(file,graph,g_height);
        loop
            if(key(_a)) y=y+2; end
            if(key(_q)) y=y-2;end
            sombra();
            frame;
        end
    End

    Process sombra()
    Begin
    if(exists(father))
        file=father.file;
        graph=father.graph;
        size=father.size;
        x=father.x;
        y=father.y;
        z=father.z+1;
        from alpha=128 to 0 step -8
            frame;
        end
    else
        return;
    end
    End

    Process senalaturno()
    Private
        int color1=rojo;
        int color2=azul;
        int mapa1;
        int mapa2;
    End
    Begin
        mapa1=new_map(res_x/2,res_y-altura_marcadores,8);
        mapa2=new_map(res_x/2,res_y-altura_marcadores,8);
        y=((res_y-altura_marcadores)/2)+altura_marcadores;
        z=2;

```

```

// mapa1
drawing_map(0,mapa1);
drawing_color(color1);
draw_box(0,0,res_x/2,res_y-altura_marcadores);
// mapa2
drawing_map(0,mapa2);
drawing_color(color2);
draw_box(0,0,res_x/2,res_y-altura_marcadores);
loop
    if(turno==1)
        x=res_x/4;
        graph=mapa1;
        from alpha=0 to 64 step 4
            frame;
        end
        while(turno==1) frame; end
        from alpha=64 to 0 step -6;
            frame;
        end
    end
    if(turno==2)
        x=(res_x/4)*3;
        graph=mapa2;
        from alpha=0 to 64 step 4
            frame;
        end
        while(turno==2) frame; end
        from alpha=64 to 0 step -6;
            frame;
        end
    end
    end
    frame;
end
End

```

#### FPG\_EXISTS(librería)

Esta función devuelve 1 si una librería FPG existe con el código indicado. Puede haber hasta 1000 librerías FPG en el programa, con códigos de 0 a 999. Es posible usar esta función dentro de un bucle para enumerar todos los gráficos que existan en memoria, con la ayuda de *map\_exists*.

PARÁMETROS: INT LIBRERÍA : Número de librería FPG

#### MAP\_EXISTS(librería, gráfico)

Esta función devuelve 1 si un gráfico dado existe dentro de la librería indicada. Dado que sólo puede haber 1000 gráficos en una librería, con códigos de 0 a 999, es posible usar esta función dentro de un bucle para enumerar todos los gráficos que existan dentro de una librería FPG.

PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico dentro de la librería

### GET\_SCREEN()

Esta función crea un nuevo gráfico con el estado de pantalla actual. Es decir, realiza una captura de pantalla.

Más concretamente, crea un nuevo gráfico con el tamaño y número de bits por pixel equivalentes a la pantalla actual, y seguidamente dibuja sobre él el fondo de pantalla y todos los procesos, textos y objetos, en el mismo orden que existan. El resultado es que el nuevo gráfico contendrá una representación fiel de la última pantalla que fue mostrada al usuario.

La imagen de la captura queda almacenada en memoria, y lo que devuelve `get_screen` es su código identificador (igual que como lo hace `load_png`, por ejemplo) el cual es creado en ese momento con un valor aleatorio mayor de 999. Para poder usar a partir de entonces en nuestro código esta imagen capturada, se tendrá que usar una variable INT -llamémosle "varcaptura"- a la cual asignaremos el valor que `get_screen()` devuelve (`varcaptura=get_screen()`), para luego, por ejemplo, *poder utilizarla como imagen de un proceso asignando su `graph` a dicha `varcaptura` (`graph=varcaptura`), y haciendo también `file=0`.*

VALOR DE RETORNO: INT : Código del nuevo gráfico

El nuevo gráfico utiliza el estado actual de todos los procesos y objetos, lo cual significa que cualquier proceso que se ejecute antes que el proceso que haga `GET_SCREEN()` puede mostrar un aspecto ligeramente diferente al de la pantalla dibujada el frame anterior. Esto puede resultar útil, ya que por ejemplo es posible obtener una copia de la pantalla sin el cursor del ratón simplemente eliminando su gráfico antes de llamar a `GET_SCREEN` y volviéndolo a restaurar inmediatamente después (para que se vea normalmente en el próximo frame, sin apreciarse ningún "parpadeo").

Si se piensa realizar varias capturas de pantalla, se ha de tener en cuenta el descargar de la memoria la captura (`unload_map(0,captura)`) ya que cada `get_screen` no sobrescribe el anterior sino que lo guarda en otro sitio de la memoria con otro código aleatorio, por lo que si se realizan muchas capturas sin descargar, el programa rápidamente empezará a ir cada vez más lento ya que se estará consumiendo los recursos -la memoria- del ordenador de forma incontrolada.

Para grabar las capturas en el disco duro, no funciona hacer un `save_fpg` (función que veremos posteriormente) del `fpg 0`, ya que los graficos de indice mayor de 999 no se graban: se tendría que cambiar el código del gráfico de la captura por otro que fuera menor de 999 y no estuviera ocupado, con lo que el trabajo del programador aumenta. Donde no hay problema es utilizando la función `save_png`.

### SAVE\_PNG(LIBRERIA,GRAFICO,"fichero")

Esta función guarda en disco una imagen en formato PNG, a partir de otra imagen cargada previamente en memoria .

#### PARAMETROS:

INT LIBRERIA : Código de la librería FPG a partir de la cual se obtiene el gráfico origen – o bien 0 si se trabaja con la librería del sistema-.

INT GRAFICO : Número del gráfico dentro de ese FPG, o bien el código obtenido mediante `new_map`, `load_png` o similares.

STRING "fichero" : Ruta completa del gráfico PNG a guardar.

En combinación con otros comandos, como *map\_xput* o *map\_xputnp*, *save\_png* nos ayudará a guardar en disco imágenes generadas y manipuladas en nuestros programas que pueden mostrar efectos visuales interesantes. He aquí, por ejemplo, un código artístico. Éste guarda (después de haber cargado un gráfico desde un FPG llamado "test.fpg" y haberlo puesto como fondo) en un fichero temporal en memoria una captura del fondo hecha a cada frame con *save\_png*, y con *map\_xput*, pega este PNG creado encima de la imagen de fondo, girado 0.1 grados sobre su centro y un 1% más pequeño. Se consigue un efecto bonito.

```
PROGRAM TP_SAVE_PNG;
global
    fichero1;
    int a=0;
end
begin
    fichero1 = load_fpg("test.fpg");
    set_mode(320,240,16);
    set_fps(150,0);
    put_screen(fichero1,1);
    loop
//Se guarda en disco el gráfico actual visible en el fondo
        save_png(0,0,"tp_save_png.png");
        a=a+100;
/*Se obtiene un "clon" del PNG actualmente grabado, se reescala un -1% y se gira una cantidad "a" y
seguidamente se vuelve a incrustar en el fondo*/
        map_xput(0,0,load_png("tp_save_png.png"),160,120,a,99,7);
        frame;
    end
end
```

El fichero PNG creado no se borra cuando se cierra el programa. Aviso: no dejes funcionando mucho rato el ejemplo que ralentiza la máquina.

Un ejemplo de esto último sería el siguiente código, donde grabamos en un fichero la captura de pantalla realizada:

```
program aa;
global
    int captura;
end
begin
set_mode(320,240,16);
write(0,10,120,14,"PRUEBA DE CAPTURA");
loop
if(key(_c))
    captura=get_screen();
    save_png(0,captura,"captura.png");
end
if(key(_esc));break;end
frame;
end
end
```

Incluso podríamos utilizar funciones como *map\_xput* o *map\_xputnp* para grabar capturas de pantallas redimensionadas, movidas de orientación, semitransparentes, etc. Por ejemplo, podríamos modificar el

código anterior para obtener una fichero que guarde una captura de la mitad de tamaño que la captura original.

```
PROGRAM TP_SAVE_PNG2;
const
    RESX=320;
    RESY=240;
end
private
    int original;
    int captura;
    int nuevo;
end
begin
    set_mode(RESX,RESY,16);
    original = load_png("a.png");
    put_screen(0,original);
    loop
        if(key(_c))
            //Se obtiene la captura de pantalla
            captura=get_screen();
            /*Se genera una nueva imagen vacía de la mitad de tamaño que la pantalla original,
donde se alojará un "clon" (reescalado a la mitad de la captura)*/
            nuevo=new_map(RESX/2,RESY/2,16);
            /*Se incrusta (centrada) la captura reescalada a la mitad en la imagen vacía creada
en la línea anterior*/
            map_xput(0,nuevo,captura,RESX/4,RESY/4,0,50,0);
            //Y se graba esta nueva imagen en un fichero
            save_png(0,nuevo,"nuevo.png");
        end
        frame;
    end
end
```

Un código curioso es el siguiente, que consigue dotar de un efecto “temblor” a la imagen de un proceso. Se necesita un png llamado “proceso.png”:

```
program aa;
global
    int idpng;
end
begin
    set_mode(320,240,16);
    idpng=load_png("proceso.png");
    miproceso();
    while(!key(_esc))
        frame;
    end
end

process miproceso()
begin
    graph=idpng;
```

```

x=160;
y=120;
loop
  if(key(_e))efecto_temblor(15,x,y);end
    frame;
end
end

Process efecto_temblor(int magnitud,a,b)
private
  int mapatemblor;
end
begin
//alpha=150;
//flags=128;
while(!key(_f))
  x=a+rand(-magnitud,magnitud);
  y=b+rand(-magnitud,magnitud);
  mapatemblor=get_screen();
  graph=mapatemblor;
  frame;
  unload_map(0,mapatemblor);
end
end

```

La idea del código anterior es sencilla. Se muestra un proceso, y cuando se apreta la tecla “e”, aparece el efecto temblor. Para que desaparezca éste, se ha de apretar la tecla “f”. El proceso “efecto\_temblor” es el más interesante: recoge los valores de la x e y del proceso “miproceso” y también recibe como parámetro un número que será la magnitud del temblor. A partir de aquí, lo único que se hace para simular ese temblor es variar la x e y aleatoriamente de este nuevo proceso, a partir de la x e y originales y dependiendo de la magnitud fijada. Pero todavía no se ve nada en pantalla. Lo interesante del asunto está en que a cada movimiento del gráfico, se hace una captura de pantalla y ésta es la que se asignará al gráfico del proceso “efecto\_temblor”, por lo que en realidad en el programa estaremos viendo dos gráficos cada vez, el del “miproceso”, inamovible, y el de “efecto\_temblor”, el cual en cada iteración cambiará.

Importante recalcar que después de haber visionado por pantalla el gráfico correspondiente a una iteración del temblor, se descarga de memoria para proseguir con la siguiente captura, ya que si no el programa a los pocos segundos se ralentizaría hasta el extremo.

Lo dejo como ejercicio: ¿cómo se podría hacer para que mientras durase el efecto temblor, el gráfico del proceso “miproceso” se dejara de ver, y retornara a ser visible sólo cuando el temblor hubiera concluido?

### GRAPHIC\_SET(librería, gráfico,tipo,valor)

Esta función permite cambiar en tiempo de ejecución determinadas características de un gráfico, especialmente si éste es un gráfico animado (GIF,APNG...).

Puede usarse para escoger un frame concreto en los gráficos animados, o cambiar la velocidad de su animación, en milisegundos. Por ejemplo, una velocidad de 0 detiene la animación del gráfico. Las animaciones de los gráficos son automáticas, e independientes de los frames por segundo establecidos por la función SET\_FPS .

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG  
INT GRÁFICO : Número de gráfico dentro de la librería  
INT TIPO : Tipo de característica a cambiar. Pueden ser:  
    G\_CENTER\_X :Coordenada horizontal del centro  
    G\_CENTER\_Y :Coordenada vertical del centro  
    G\_ANIMATION\_STEP :Frame actual de la animación  
    G\_ANIMATION\_SPEED :Velocidad actual de la animación  
INT VALOR : Nuevo valor a asignar a la característica

No quisiera dejar de comentar al final de este apartado dedicado a los gráficos una posibilidad curiosa que tenemos disponible a la hora de cargar ficheros FPG. Todo el mundo sabe a estas alturas que para cargar estos contenedores hemos de escribir en nuestro código una línea semejante a

```
idfpg=load_fpg("fichero_fpg.fpg");
```

Pero existe una alternativa para hacer lo mismo, que es escribiendo esto:

```
load_fpg("fichero_fpg.fpg", &idfpg);
```

donde como vemos, el identificador cargado del FPG se le pasará a un puntero que apunta a la variable "idfpg". Pero, ¿hay alguna diferencia entre esta nueva forma de hacer la carga y la que ya sabíamos?

Pues sí, hay una diferencia sutil: esta segunda forma tiene su utilidad básicamente cuando se cargan ficheros FPG muy grandes (con muchísimos gráficos en su interior o con gráficos de dimensiones muy elevadas). Algunos ficheros FPG pueden ser tan grandes que pueden tardar demasiado tiempo en acabarse de cargarse todo completo, y como hasta que no se acaba ese proceso el programa se queda detenido en esa línea y no continúa, el juego se quedaría "congelado" durante unos instantes.

Con esta segunda manera de cargar FPGs se evita este efecto, porque los diferentes gráficos del interior del contenedor se irán cargando "en segundo plano", continuando la ejecución del programa mientras tanto. Esto es muy útil, ya hemos dicho, si el archivo es tan grande que se corre el riesgo de parecer que el programa se haya bloqueado, o si quieres ir haciendo una animación mientras se acaba de cargar entero.

Por último comentar la existencia de la variable global **WINDOW\_STATUS**. Esta variable se pone automáticamente a 0 cuando la ventana del programa está minimizada, y a 1 cuando está visible (incluso aunque esté en segundo plano, tapada por otras ventanas). En juegos que se ejecutan a ventana completa, esta variable se pone a 1 cuando la aplicación está visible, y a 0 si el usuario pasa al escritorio (por ejemplo, pulsando ALT+TAB en Windows, o el equivalente en otro sistema operativo).

### **Trabajar con gráficos FGC:**

En este apartado voy a intentar explicaros el funcionamiento de un puñado de funciones de Fénix que nos permitirán, entre otras cosas, poder programar un rudimentario "FPGEdit" que lo único que haga de momento sea crear un archivo FGC, añadir en él imágenes y grabarlo en disco.

Este ejercicio nos será muy útil porque de momento no hay ninguna aplicación disponible que sea capaz de trabajar con el nuevo formato FGC, sustituto del FPG. Ya se ha comentado que ni el "FPGEdit" ni el comando de consola "fpg.exe" son capaces de abrir, editar o guardar ficheros en este nuevo formato, por lo que si queremos empezar a trabajar con ellos, tendremos que recurrir a las funciones que a continuación se detallan.

### FPG\_NEW()

Esta función crea en memoria una nueva librería FGC (el nombre de la función puede prestarse a confusión) y devuelve su código numérico de librería. A partir de entonces, es posible añadir nuevos gráficos a esta librería mediante la función FPG\_ADD , y grabarla posteriormente a disco con SAVE\_FGC .

NOTAS Todos los gráficos añadidos a una misma librería deben tener la misma profundidad de color (1, 8 o 16 bits). Añadir un gráfico a una librería cuando ya hay en la misma algún gráfico de otro tipo se considera una condición de error.

VALOR DE RETORNO: INT : Código de la nueva librería FGC

### FPG\_ADD ( LIBRERÍA, GRÁFICO, LIBRERÍA-ORIGEN, GRÁFICO-ORIGEN )

Esta función permite modificar una librería FGC (el nombre de la función puede prestarse a confusión) en memoria, añadiendo un nuevo gráfico a partir de otro gráfico ya en memoria.

La librería de destino debe ser el código de una librería FGC obtenido mediante las funciones LOAD\_FGC o FPG\_NEW . La librería de origen debe ser el código de otra librería en memoria, aunque también puede identificar la propia librería de destino (para hacer copias de gráficos dentro de la misma) o bien la librería 0, para poder especificar gráficos obtenidos con funciones como NEW\_MAP .

FPG\_ADD crea una copia del gráfico original, que sigue siendo válido y accesible. Si ya hubiese un gráfico con el mismo código en la librería de destino, éste será eliminado de memoria y sustituido por el nuevo.

La mayor utilidad de esta función es crear nuevas librerías de gráficos para guardarlas luego con SAVE\_FGC . Otras funciones permiten modificar los parámetros en memoria de los gráficos añadidos a la librería, como MAP\_SET\_NAME .

Todos los gráficos añadidos a una misma librería deben tener la misma profundidad de color (1, 8 o 16 bits). Añadir un gráfico a una librería cuando ya hay en la misma algún gráfico de otro tipo se considera una condición de error.

#### PARÁMETROS:

INT LIBRERÍA : Código de la librería a modificar

INT GRÁFICO : Código deseado para el nuevo gráfico, de 0 a 999

INT LIBRERÍA-ORIGEN : Código de la librería origen

INTGRÁFICO-ORIGEN : Código del gráfico original

### SAVE\_FGC ( LIBRERÍA, "FICHERO" )

Esta función crea o sobrescribe un fichero en disco con el contenido de una librería FGC en memoria que puede haberse creado con la función FPG\_NEW , o recuperado de disco mediante LOAD\_FGC . Los cambios realizados en memoria con funciones como FPG\_ADD o UNLOAD\_MAP/UNLOAD\_FBM no afectan al fichero original en disco, así que es preciso utilizar esta función para sobrescribirlo si así se desea.

#### PARÁMETROS:

INT LIBRERÍA : Código de la librería a guardar

STRING FICHERO : Nombre del fichero

### MAP\_SET\_NAME ( LIBRERÍA, GRÁFICO, "NOMBRE" )

Esta función cambia el nombre de un gráfico en memoria. Todo gráfico tiene un nombre de hasta 32 caracteres, que puede cambiarse mediante esta función, o recuperarse mediante MAP\_NAME . Este nombre es especialmente útil cuando el gráfico forma parte de una librería FGC, ya que de otra manera no habría otro sistema aparte del número del gráfico para distinguir uno de otro.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FGC

INT GRÁFICO : Número de gráfico dentro de la librería

STRING NOMBRE : Nuevo nombre del gráfico

### MAP\_NAME ( LIBRERÍA, GRÁFICO )

Devuelve el nombre de un gráfico. Todo gráfico tiene un nombre de hasta 32 caracteres, que puede recuperarse mediante esta función, o cambiarse mediante MAP\_SET\_NAME . Este nombre es especialmente útil cuando el gráfico forma parte de una librería FGC, ya que de otra manera no habría otro sistema aparte del número del gráfico para distinguir uno de otro.

#### PARÁMETROS:

INT LIBRERIA: Número de librería FGC

INT GRAFICO: Número de gráfico dentro de la librería

VALOR DEVUELTO: STRING : Nombre del gráfico

Vamos a hacer un ejemplo de una aplicación que cargará un archivo FPG previamente existente (llamado "antiguo.fpg") con dos imágenes de códigos 001 y 002, y guardará las mismas imágenes en un nuevo archivo FGC llamado "nuevo.fgc". Es decir, crearemos un conversor FPG->FGC. Esto sería así:

```
program aver;
private
    int idfpg1;
    int idfpg2;
end
begin
    idfpg1=load_fpg("antiguo.fpg");
    idfpg2=fpg_new();
/*Los gráficos se introducen en el nuevo archivo FGC en el mismo orden que estaban en el antiguo FPG*/
    fpg_add(idfpg2,1,idfpg1,1);
    fpg_add(idfpg2,2,idfpg1,2);
    save_fgc(idfpg2,"nuevo.fgc");
end
```

Prueba ahora de abrir el archivo "nuevo.fgc" con el FPGedit o el fpg.exe. Verás que devuelven un error porque no reconocen el formato. ¿Y cómo podemos saber si este archivo "nuevo.fgc" es correcto? Pues escribiendo un programa que sea capaz de cargar y leer este tipo de archivos.

```
program aver;
```

```

global
  int idfgc1;
end
begin
  idfgc1=load_fgc("nuevo.fgc");
  put_screen(idfgc1,1);
  proceso();
  loop
    if(key(_esc)) break; end
    frame;
  end
  unload_fgc(idfgc1);
  let_me_alone();
end

process proceso()
begin
file=idfgc1;
graph=2;
x=100;
y=100;
loop
  frame;
end
end

```

Si deseas tener a mano una herramienta práctica que te sirva de conversor de archivos FPG completos en FGC, no es demasiado difícil programársela uno mismo en Fénix. Como muestra, a continuación presento el código de un conversor FPG->FGC (original de Izubiaurre) llamado “fpg2fgc”, el cual es una aplicación que se ejecuta desde la línea de comandos de la siguiente manera:

#### **fxi fpg2fgc.dcb archivo.fpg**

Es decir, nuestro programa ha de recibir un parámetro desde la propia consola cmd.exe, el cual será el nombre del archivo FPG (incluyendo su extensión) a convertir. El archivo FPG ha de estar en la misma carpeta desde donde se ejecuta “fpg2fgc”, y el nuevo archivo FGC se grabará en ese mismo directorio.

```

program fpg2fgc;
global
int f_fpg;
string filename;
end
begin

/*Este if recoge el nombre del archivo FPG especificado como primer parámetro en la línea de
comandos de Windows. La recogida de parámetros por parte de nuestros programas de Fénix desde la
consola cmd.exe es un tema que no se ha tratado todavía, pero se estudiará posteriormente. (consultar
último capítulo). Baste sabe que para ello se utilizan dos variables predefinidas: argc y argv */
if (argc > 0) //Si se han pasado parámetros a nuestro programa desde la consola...
  filename = argv[1]; //...El primero de ellos será el nombre del archivo FPG
else
  exit();
end

```

```
f_fpg = load_fpg(filename); //Cargo el archivo FPG
filename = substr(filename,0,len(filename)-4); /*Extraigo la extensión de su nombre.La función substr se
explica más adelante en este mismo capítulo*/
save_fgc(f_fpg, filename + ".fgc"); /*Línea clave: guardo el archivo identificado por f_fpg en formato
FGC, y su nombre será el mismo que tenía anteriormente más la extensión ".FGC"*/
end
```

A partir de aquí podríamos programarnos nosotros mismos pequeñas aplicaciones que suplan en parte la carencia de no tener un programa estilo “FPGEdit” para poder trabajar con ficheros FGC. Por ejemplo, a continuación se muestra un código (original de Pixel) que sirve para dos cosas:

- 1.-Introduce múltiples imágenes PNG (han de estar situadas en el directorio actual) dentro de un FGC, creado en ese mismo momento. Los archivos PNG deberán de tener obligatoriamente como nombre el número que se desea tengan dentro del FGC (es decir: 1.png, 2.png,etc hasta 999.png)
- 2.-Extrae de un FGC múltiples imágenes PNG en una carpeta dentro del directorio actual llamada igual que el archivo FGC, y les pone de nombre en el fichero el número que tenían dentro del FGC ( es decir: 1.png, 2.png,etc hasta 999.png)

*Program hazme\_el\_fgc;*

*Global*

```
int i=1;
String default_text="";
String prompt="_ ";
int inputtext;
String entry;
int el_fgc;
int next=1;
String txt_numi;
int porcentaje;
int linux=-1;
int bites;
```

*end*

*Begin*

```
set_fps(0,9);
write(0,0,8,0,"1.Mete imágenes en un nuevo FGC");
write(0,0,16,0,"2.Saca imágenes de un FGC existente");
Loop
    If(key(_1)) While(key(_1)) Frame; End png2fgc(); Break; End
    If(key(_2)) While(key(_2)) Frame; End fgc2png(); Break; End
    Frame;
End
```

*End*

*Process png2fgc()*

*Begin*

```
delete_text(all_text);
write(0,0,0,0,"Una pequeña nota:");
write(0,0,8,0,"Suele provocar problemas en sistemas linux");
write(0,0,16,0,"(u otros que no sean windows) que la");
write(0,0,24,0,"extensión no coincida mayusculas y minusculas.");
write(0,0,32,0,"Debes pasar todas las extensiones a minusculas.");
write(0,0,40,0,"Cuando estes preparado pulsa Intro.");
```

```

While(!key(_enter)) Frame; End
delete_text(all_text);
write(0,0,0,0,"Nº de colores");
write(0,0,8,0,"(1) 8bit - 256");
write(0,0,16,0,"(2) 16bit - 65536");
While(bites==0)
    If(key(_1)) bites=8; End
    If(key(_2)) bites=16; End
    Frame;

End
set_mode(320,240,bites);
y=10;
delete_text(all_text);
entry = default_text ;
write(0,0,0,0,"Nombre del FGC a crear (sin la extensión)");
Loop
/*En las líneas siguiente utilizaremos una variable global predefinida todavía no explicada:
SCAN_CODE. No te preocupes ahora de ella, no es importante. Sólo has de saber que actúa de forma
similar -aunque no idéntica- a la función key(): es decir, sirve para detectar qué tecla se ha pulsado. Sus
valores pueden ser las mismas constantes que se usan en la función key()*/
    If(scan_code == _backspace) entry = substr (entry, 0, -2); End
    If(scan_code == _enter) Break ; End
    scan_code = 0 ; //Esta línea hace que el programa crea que no hay pulsada ninguna tecla
/*La línea siguiente es un poco complicada de entender en estos momentos, sin haber estudiado primero
las funciones de cadena y de uso del teclado, pero básicamente sirve para recoger el texto introducido a
través del mismo: recoge lo que el usuario escriba y lo almacena en la variable de tipo cadena "entry".*/
    If(ascii >= 32) entry = entry +chr(ascii) ; ascii = 0 ; End
    inputText = write (0,x,y,0,entry + prompt);
    Frame ;
    delete_text (inputText) ;
End
delete_text(all_text);
write(0,0,0,0,"Espera...");
el_fgc=fpg_new();
delete_text(all_text);
write(0,0,0,0,"Cargando...");
delete_text(all_text);
write(0,0,0,0,"Metiendo imágenes al fgc...");
write_var(0,208,0,0,porcentaje);
While(i<999)
    porcentaje=i*100/999;
    If(next==1) next=0; convierte_png2fgc(i); End
    Frame;

End
delete_text(all_text);
write(0,0,0,0,"¡Guardando!");
save_fgc(el_fgc,entry + ".fgc");
delete_text(all_text);
write(0,0,0,0,"¡Se ha finalizado la introducción!");
While(!key(_enter)) Frame; End
exit();

End
Process convierte_png2fgc(num)

```

*Begin*

```
If(file_exists(itoa(num)+".png"))  
    graph=load_png(itoa(num)+".png");  
    fpg_add(el_fgc,num,0,graph);
```

```
End  
i++;  
next=1;
```

*End*

*Process fgc2png()*

*Begin*

```
delete_text(all_text);  
set_mode(320,240,16);  
set_fps(0,0);  
y=10;  
delete_text(all_text);  
write(0,0,0,0,"¿En que sistema operativo estás?");  
write(0,0,8,0,"(1) Windows");  
write(0,0,16,0,"(2) Linux");  
While(linux==-1)  
    If(key(_1)) linux=0; End  
    If(key(_2)) linux=1; End  
    Frame;  
End  
delete_text(all_text);  
write(0,0,0,0,"Necesitas crear una carpeta que se llame como");  
write(0,0,8,0,"el fgc y darle derechos de acceso para");  
write(0,0,16,0,"que el programa pueda guardar las imágenes.");  
write(0,0,24,0,"En caso contrario, las imágenes se guardarán");  
write(0,0,32,0,"en la misma carpeta en la que esta el programa.");  
write(0,0,40,0,"Cuando estes preparado pulsa Intro.");  
While(!key(_enter)) Frame; End  
While(el_fgc<=0)  
    delete_text(all_text);  
    entry = default_text ;  
    write(0,0,0,0,"Nombre del FGC a tratar (sin la extensión)");  
    Loop  
        If(scan_code == _backspace) entry = substr (entry, 0, -2); End  
        If(scan_code == _enter) Break ; End  
        scan_code = 0 ;  
        If (ascii >= 32) entry = entry + chr(ascii);ascii = 0 ; End  
        inputText = write (0,x,y,0,entry + prompt);  
        Frame ;  
        delete_text (inputText) ;  
    End  
    delete_text(all_text);  
    write(0,0,0,0,"Espera...");  
    el_fgc=file_exists(entry + ".fgc");  
    frame;  
End  
x=160; y=120;  
delete_text(all_text);  
write(0,0,0,0,"Cargando...");  
el_fgc=load_fgc("./" + entry + ".fgc");
```

```

delete_text(all_text);
If(linux==0) mkdir(entry); End
If(linux==1) chdir(entry); End
file=el_fgc;
delete_text(all_text);
write(0,0,0,0,"Extrayendo...");
write_var(0,118,0,0, porcentaje);
While(i<999)
    porcentaje=i*100/999;
    If(next==1) next=0; graph=i; convierte_fgc2png(i); End
    Frame;
End
delete_text(all_text);
write(0,0,0,0,"¡Se ha finalizado la extracción!");
While(!key(_enter)) Frame; End
exit();

```

End

Process convierte\_fgc2png(num)

Begin

```

If(graphic_info(el_fgc,num,g_width)!=0 AND graphic_info(el_fgc,num,g_height)!=0)
    If(linux==0)
        save_png(el_fgc,num,entry+"/"+itoa(num)+".png");
    Else
        save_png(el_fgc,num,itoa(num)+".png");
    End
End

```

```

End
i++;
next=1;

```

End

Con el código anterior obtenemos una parte pequeña de la funcionalidad que nos ofrece el “FPGEdit” para archivos FPG, porque, no obstante, falta bastante todavía. Por ejemplo, para lograr tener más flexibilidad en el tratamiento de contenedores FGC, se necesitaría incorporar a Fénix alguna función que nos permitiera eliminar imágenes del FGC y seguramente también otra que las listara, entre otras ideas.

### **Dibujar primitivas gráficas:**

Por primitiva gráfica se entiende una figura simple de dibujar, como una línea, un círculo o un rectángulo. Estas formas sencillas se llaman primitivas porque a partir de ellas se pueden crear la mayoría de dibujos posibles: cualquier gráfico de hecho no es más que un conjunto más o menos complejo de círculos (y elipses) y polígonos (rectángulos, triángulos...)

Para dibujar primitivas gráficas, Fénix dispone de unas cuantas funciones específicas. Está claro que podríamos utilizar como hasta ahora cualquier editor de imágenes para dibujar los círculos y rectángulos y luego utilizar el FPG correspondiente, pero en el caso de estas figuras tan sencillas de dibujar, Fénix ofrece la posibilidad de crearlas mediante código, mejorando así la velocidad de ejecución del programa, y también la comodidad del programador, que no tiene que recurrir a dibujos externos creados previamente.

La primera función que tendremos que escribir siempre cuando queramos pintar primitivas gráficas, sea cual sea su tipo, es DRAWING\_MAP.

### **DRAWING\_MAP(librería, gráfico)**

Esta función escoge el gráfico destino donde se pintarán las primitivas gráficas (DRAW\_RECT, DRAW\_LINE, DRAW\_CIRCLE,etc) que vayamos a dibujar, ya que en Fénix cualquier primitiva gráfica siempre se ha de pintar sobre un gráfico determinado, definido mediante esta función. Es decir, no es válido llamar a ninguna de estas funciones si no se escoge antes un fondo como destino.

El gráfico (0, 0) representa el fondo de pantalla.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG  
INT GRÁFICO : Número de gráfico a dibujar

### **DRAWING\_COLOR(color)**

Esta función permite escoger el color con el que se dibujarán las primitivas gráficas (DRAW\_RECT, DRAW\_BOX, DRAW\_LINE, DRAW\_CIRCLE, DRAW\_FCIRCLE,etc) hasta nuevo cambio..

Como siempre, en el caso de modos gráficos de 256 colores (8 bits), el valor de color debe ser un número de 0 a 255. En el caso de modos gráficos de 16 bits, el valor es una codificación de las componentes del color que varía en función de la tarjeta de vídeo y el modo gráfico. Lo normal es usar la función RGB para obtener la codificación de un color concreto, o bien usar un color obtenido por una función como MAP\_GET\_PIXEL

PARÁMETROS: INT COLOR : Número de color

### **DRAWING\_ALPHA(alpha)**

Esta función permite escoger el valor de transparencia a utilizar con todas las primitivas gráficas (funciones que empiezan por DRAW), hasta nuevo cambio. Un valor de 255 indica opacidad completa (sin transparencia), mientras con un valor de 0 la primitiva gráfica sería invisible. Es válido usar cualquier valor intermedio (por ejemplo, 128 para especificar una transparencia del 50%). Por defecto el valor de transparencia es de 255 y las primitivas se dibujan sin transparencia.

PARÁMETROS: INT ALPHA : Valor de opacidad, entre 0 y 255

### **DRAW\_RECT(X1,Y1,X2,Y2)**

Un rectángulo consta de cuatro líneas (dos verticales y dos horizontales) y no está relleno. Esta función dibuja un rectángulo entre dos puntos sobre el gráfico escogido por la última llamada a DRAWING\_MAP, empleando el color elegido con la función DRAWING\_COLOR.

El primer punto representa el vértice superior izquierdo de la figura, y el segundo punto el vértice inferior derecho.

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará el rectángulo (el escogido por la última llamada a DRAWING\_MAP). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

Además, se utiliza el último valor especificado con DRAWING\_ALPHA como nivel de transparencia para el dibujo, de manera que es posible dibujar un rectángulo parcialmente transparente.

PARÁMETROS:

INT X1 : Coordenada horizontal del primer punto  
INT Y1 : Coordenada vertical del primer punto  
INT X2 : Coordenada horizontal del segundo punto  
INT Y2 : Coordenada vertical del segundo punto

Vamos a escribir un pequeño ejemplo que nos pinte en pantalla varios rectángulos de dimensiones, colores y transparencias diferentes:

```
program Test_DRAW_RECT;  
begin  
  set_mode(640,480,16);  
  drawing_map(0,0);  
  repeat  
    drawing_color(rand(0,65535));  
    drawing_alpha(rand(0,255));  
    draw_rect(rand(0,639),rand(0,479),rand(0,639),rand(0,479));  
    frame;  
  until(key(_esc))  
end
```

En este ejemplo lo que hemos hecho ha sido pintar rectángulos en el gráfico de fondo de pantalla –y como no hay ninguno definido, en realidad se pintan sobre el fondo negro-.

Pero con DRAWING\_MAP hemos dicho que se puede definir un gráfico cualquiera sobre el cual, y sólo sobre éste, se pintarán las primitivas. Para verlo, ejecuta el siguiente ejemplo, donde se ha definido un gráfico sobre el cual se pintarán unas primitivas (y que por tanto, no podrán pintarse fuera de él), de manera que éstas no se verán por todo el fondo de la pantalla sino que aparecerán visibles solamente los trozos que se pinten sobre ese gráfico determinado. Para comparar y ver la diferencia, en el mismo ejemplo también se pintan primitivas en toda la pantalla

```
program Test_DRAWING_MAP;  
global  
  int Map;  
end  
begin  
  set_mode(640,480,16);  
  /*Genero el gráfico donde se van a pintar las primitivas. Fuera de él no se pintarán. Este gráfico es un  
  cuadrado de 200x200 de color blanco*/  
  map=new_map(200,200,8);  
  map_clear(0,Map,rgb(255,255,255));  
  //Visualizo el gráfico, y lo doy una posición dentro de la pantalla  
  graph=Map;  
  x=320; y=240;  
  //Proceso que pinta una serie de primitivas gráficas dentro de el cuadrado blanco  
  Draw_Grafico();  
  /*Para comparar con el proceso anterior, también ejecutamos este proceso, el cual pinta las primitivas  
  gráficas en todo el fondo (gráfico 0,0)*/
```

```

Draw_Pantalla();
timer=0;
repeat
//Cada 2 segundos pinto una nueva primitiva tanto dentro del cuadrado blanco como fuera
if(timer>200)
  Draw_Grafico();
  Draw_Pantalla();
  timer=0;
end
frame;
until(key(_esc))
end

process Draw_Grafico()
begin
/*La línea clave: estamos diciendo que todas las primitivas que se generen a partir de ahora (hasta nuevo
aviso)sólo se pintarán sobre el gráfico especificado por "Map"*/
drawing_map(0,Map);
drawing_color(rand(0,255));
draw_rect(rand(0,639),rand(0,419),rand(0,639),rand(0,419));
end

process Draw_Pantalla()
begin
/*Todas las primitivas que se generen a partir de ahora (hasta nuevo aviso) se podrán pintar en toda la
pantalla*/
drawing_map(0,0);
drawing_color(rand(0,255));
draw_rect(rand(0,639),rand(0,419),rand(0,639),rand(0,419));
end

Creo que es fácil de entender.

```

A parte de rectángulos, también podemos dibujar más cosas:

### **DRAW\_CIRCLE(X,Y,RADIO)**

Dibuja un círculo (sin relleno) alrededor de un punto, sobre el gráfico escogido por la última llamada a DRAWING\_MAP, empleando el color elegido con DRAWING\_COLOR, y el nivel de transparencia especificado en la última llamada a DRAWING\_ALPHA.

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará el círculo (el escogido por la última llamada a DRAWING\_MAP). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

#### PARÁMETROS:

INT X : Coordenada horizontal del centro  
INT Y : Coordenada vertical del centro  
INT RADIO : Radio en pixels

### **DRAW\_LINE(X1,Y1,X2,Y2)**

Dibuja una línea entre dos puntos en el gráfico escogido por la última llamada a DRAWING\_MAP, empleando el color elegido con DRAWING\_COLOR y el nivel de transparencia especificado en la última llamada a DRAWING\_ALPHA.

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará la línea (el escogido por la última llamada a DRAWING\_MAP). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

PARÁMETROS:

INT X1 : Coordenada horizontal del primer punto  
INT Y1 : Coordenada vertical del primer punto  
INT X2 : Coordenada horizontal del segundo punto  
INT Y2 : Coordenada vertical del segundo punto

Un código curioso que se puede probar para ilustrar el funcionamiento de esta función es éste.

```
program line;
begin
  set_mode(320,240,16);
  x=10; y=10;
  loop
    if(key(_up))y=y-1;end
    if(key(_down))y=y+1;end
    if(key(_left))x=x-1;end
    if(key(_right))x=x+1;end

    delete_draw(0);
    drawing_color(rgb(0,255,0));
    draw_line(0,0,x,y);
  frame;
end
end
```

**DRAWING\_STIPPLE(valor)**

Esta función permite escoger el aspecto con el que se dibujan las líneas dibujadas posteriormente con DRAW\_LINE o DRAW\_RECT.

El parámetro es un número entero de 32 bits, en el cual cada bit a 1 representa un pixel activo en cada segmento de 32 pixels de la línea. Por defecto, el valor de stipple es (en notación hexadecimal) 0FFFFFFFh -que equivale en decimal a 4294967295- ya que todos los 32 bits están a 1 , por lo que las líneas tienen la apariencia de una línea sólida. Algunos valores útiles para este parámetro son 05555555h -que equivale a 1431655765 en decimal- para dibujar una línea de puntos o 0F0F0F0Fh -que equivale a 252645135 en decimal- para dibujar una línea de rayas.

PARÁMETROS: INT VALOR : Nuevo valor de "stipple"

**DRAW\_CURVE(X1,Y1,X2,Y2,X3,Y3,X4,Y4,nivel)**

Dibuja una curva Bézier entre dos puntos (puntos 1 y 4), usando otros dos puntos (puntos 2 y 3) como puntos de control de la curva. El nivel de calidad es un valor entero entre 1 y 16. A mayor nivel de calidad,

más precisa será la curva, pero más costoso será dibujarla. Se recomienda un nivel de calidad entre 6 y 9.

Usa el color escogido por elegido con `DRAWING_COLOR`, en el gráfico escogido por la última llamada a `DRAWING_MAP`. Además, se utiliza el último valor especificado con `DRAWING_ALPHA` como nivel de transparencia para el dibujo, de manera que es posible dibujar una curva parcialmente transparente.

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará la curva (el escogido por la última llamada a `DRAWING_MAP`). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

**PARÁMETROS:**

INT X1 : Coordenada horizontal del primer punto  
INT Y1 : Coordenada vertical del primer punto  
INT X2 : Coordenada horizontal del segundo punto  
INT Y2 : Coordenada vertical del segundo punto  
INT X3 : Coordenada horizontal del tercer punto  
INT Y3 : Coordenada vertical del tercer punto  
INT X4 : Coordenada horizontal del cuarto punto  
INT Y4 : Coordenada vertical del cuarto punto  
INT NIVEL : Nivel de calidad de la curva

**DRAW\_BOX(X1,Y1,X2,Y2)**

Dibuja un rectángulo con relleno de color entre dos puntos sobre el gráfico escogido por la última llamada a `DRAWING_MAP`, empleando el color elegido con `DRAWING_COLOR` y el nivel de transparencia especificado en la última llamada a `DRAWING_ALPHA`. Todos los puntos delimitados por esas cuatro coordenadas serán cambiados al color escogido.

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará el rectángulo (el escogido por la última llamada a `DRAWING_MAP`). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

**PARÁMETROS:**

INT X1 : Coordenada horizontal del primer punto  
INT Y1 : Coordenada vertical del primer punto  
INT X2 : Coordenada horizontal del segundo punto  
INT Y2 : Coordenada vertical del segundo punto

**DRAW\_FCIRCLE(X,Y,RADIO)**

Dibuja un círculo (con relleno de color) alrededor de un punto sobre el gráfico escogido en la última llamada a `DRAWING_MAP`, empleando el color elegido con `DRAWING_COLOR` y el nivel de transparencia especificado en la última llamada a `DRAWING_ALPHA`.

El origen de coordenadas vienen referidas siempre a la esquina superior izquierda del gráfico sobre el que se pintará el círculo (el escogido por la última llamada a `DRAWING_MAP`). Es válido especificar coordenadas fuera de los límites del gráfico, pero el dibujo resultante se recortará sobre éstos.

**PARÁMETROS:**

INT X : Coordenada horizontal del centro  
 INT Y : Coordenada vertical del centro  
 INT RADIO : Radio en pixels

Vamos a ver un ejemplo sencillo sobre el uso de todas estas funciones nuevas.

```

program Test_DRAW;
begin
  set_mode(640,480,16);
  set_fps(1,1);
  drawing_map(0,0);
  repeat
    drawing_color(rand(0,65535));
    drawing_alpha(rand(100,200));
    draw_circle(rand(0,639),rand(0,200),rand(0,200));frame;
    draw_fcircle(rand(0,639),rand(0,200),rand(0,200)); frame;
    drawing_stipple(rand(1,4294967294));
    draw_line(rand(0,639),rand(0,479),rand(0,639),rand(0,479)); frame;
    draw_curve(rand(0,639),rand(0,479),rand(0,639),rand(0,479),rand(0,639),rand(0,479),rand(0,639),r
and(0,479),9); frame;
    draw_box(rand(0,639),rand(0,479),rand(0,639),rand(0,479)); frame;
  until(key(_esc))
end
  
```

Otro ejemplo bastante parecido, donde unas cuantas primitivas cambian su color automáticamente, recorriendo todo el espectro de colores, desde el blanco (RGB=255,255,255) hasta el negro (RGB=0,0,0):

```

program Test_DRAWING_COLOR;
global
  int r,g,b,color;
end
begin
  set_mode(320,240,16);
  write(0,10,30,3,"Color Number: ");
  color=rgb(r,g,b);
  write_var(0,105,30,3,color);
  drawing_map(0,0);
  //Recorro todos los valores posibles de las distintas componentes RGB
  for(r=255;r>=0;r--)
    for(g=255;g>=0;g--)
      for(b=255;b>=0;b--)
        color=rgb(r,g,b);
        drawing_color(color);
        Draw();
        frame;
      end
    end
  end
end

process Draw()
begin
  draw_line(50,50,270,50);
  
```

```

draw_rect(50,60,100,110);
draw_box(220,60,270,110);
draw_circle(75,130,15);
draw_fcircle(245,130,15);
end

```

Otro ejemplo ingenioso; el dibujo de una casa:

```

PROGRAM ej_dibujo_primitivas;
BEGIN
set_mode(320,240,16);
set_title("Ejemplo de primitivas");
write_var(0,10,10,0,fps);
drawing_map(0,0); //Se selecciona la imagen sobre la que se dibujará (en este caso, el fondo de pantalla)
drawing_color(rgb(255,255,0));
draw_box(75,100,250,190); //paredes
drawing_color(rgb(255,0,0));
draw_rect(142,130,182,190); //puerta
drawing_color(rgb(255,100,255));
draw_triangle(75,100,162,25,250,100); //techo
drawing_color(rgb(0,0,255));
draw_fcircle(162,70,20); //ventana
drawing_color(rgb(100,0,0));
draw_circle(172,160,5); //pomo
FRAME;
WHILE(!key(_esc)) FRAME; END
END
PROCESS draw_triangle(x1,y1,x2,y2,x3,y3)
BEGIN
draw_line(x1,y1,x2,y2);
draw_line(x2,y2,x3,y3);
draw_line(x3,y3,x1,y1);
END

```

A continuación muestro otro ejemplo bastante interesante. Necesitarás un fichero FPG llamado “paleta.fpg” con dos imágenes. Una tendrá el código 001, será de 800x600 y estaría bien que tuviera el máximo de colores posible. Otra tendrá el código 002, será de 15x15 y representará la imagen del puntero del ratón (el tema de cómo introducir el uso del ratón en nuestros programas se explicará en profundidad más adelante en este mismo capítulo).

```

Program ejemplo_get_pixel;
Global
  int color;
  int fichero1;
end
Begin

  set_mode (800,600,16);
  fichero1=load_fpg("paleta.fpg");
  put_screen(0, 1);
  mouse.graph=2;
  write (0, 260, 180, 0, "Color");
  write_var(0, 260, 200, 0, color);

```

```
Loop
    delete_draw(0);
    drawing_color(color);
    draw_box(320,180,380,220);
    // Cogemos el color del punto del fondo de la pantalla
    color = map_get_pixel(0,0,mouse.x, mouse.y);
    Frame;
    If(key(_esc)) Break; End
End
End
```

Con las funciones descritas hasta ahora, lo único que podemos hacer es dibujar una primitiva asociándolo a un gráfico (aunque sea el fondo). Si este gráfico dejara de visualizarse en pantalla, todas las primitivas pintadas en él desaparecerían con él. Para hacer que una primitiva tenga una entidad propia y no necesite de ningún gráfico extra para poder ser mostrada, lo primero que hay que hacer es “activar este modo de trabajo”, con el cual lograremos tener un mayor control sobre una primitiva determinada (en concreto, podremos borrarla o moverla de forma independiente), ya que cada una de ellas dispondrá de un identificador propio y único, obtenido en la llamada a la función gráfica que la haya creado.

Para “activar este modo de trabajo”, lo primero que hay que hacer es prescindir de DRAWING\_MAP, y en su lugar utilizar la función DRAWING\_Z. Esta función, además, tiene un parámetro que indica la Z a la que se verán las primitivas que se pinten a partir de entonces.

¿Y qué ganamos con esta manera de trabajar? Pues poder utilizar dos nuevas funciones: DELETE\_DRAW y MOVE\_DRAW, las cuales no podríamos usar si trabajáramos con DRAWING\_MAP, y que sirven, respectivamente, para borrar y mover en la pantalla una primitiva concreta.

DRAWING_Z(Z)
Activa el “modo persistente” (independiente de los demás gráficos) en las funciones de dibujo de primitivas y escoge el nivel de profundidad a utilizar.
Tal como he dicho, por defecto, las funciones de dibujo actúan sobre el fondo de pantalla o el gráfico especificado en la última llamada a la función DRAWING_MAP. Sin embargo, una llamada a esta función activará un modo de funcionamiento alternativo, en el cual las primitivas gráficas (dibujadas con las funciones que empiezan por DRAW) persisten en pantalla como un gráfico independiente más, y podrán tener un identificador único.
Al igual que un proceso, estas primitivas gráficas se dibujan con una coordenada Z determinada, que permite que hayan procesos dibujados por encima o debajo de ellas. Esta función, además de activar este modo, permite escoger la coordenada Z que se aplicará a las primitivas gráficas dibujadas después de llamarla.
Para poder dibujar objetos con distinta coordenada Z, cada primitiva recuerda la coordenada Z elegida en el momento en que fue creada. Es posible, por lo tanto, hacer varias llamadas a DRAWING_Z entre llamadas a funciones de dibujo para poder cambiar el valor de coordenada Z con la que ir creando nuevas primitivas.
Las primitivas gráficas dibujadas de esta manera, tal como he dicho, podrán manipularse mediante las funciones MOVE_DRAW y DELETE_DRAW.

Para desactivar este modo y volver a hacer dibujos directamente sobre el fondo de pantalla u otro gráfico, es preciso llamar a la función DRAWING\_MAP cuando se necesite.

PARÁMETROS : INT Z : Valor de profundidad, indicativo del orden de dibujo

### DELETE\_DRAW(primitiva)

Destruye (borra) una primitiva creada con una de las funciones de primitivas (utilizan todas el prefijo DRAW\_), si ésta se ha creado después de una llamada a DRAWING\_Z.

El parámetro que identifica a la primitiva es el valor devuelto por la función llamada para crearla.

Alternativamente, puede especificarse 0 como objeto, con lo que esta función destruirá todos los objetos creados con funciones de primitivas gráficas.

PARÁMETROS: INT OBJETO : Identificador de la primitiva a eliminar

### MOVE\_DRAW(primitiva,X,Y)

Mueve una primitiva gráfica creada con una de las funciones de primitivas (utilizantodas el prefijo DRAW\_), si ésta se ha creado después de una llamada a DRAWING\_Z, a una coordenada concreta de la pantalla.

Las coordenadas especificadas hacen referencia al primero de los puntos utilizados para dibujar una línea o curva, o bien al centro del círculo.

La gráfica no será deformada, sólo desplazada: en una línea, el segundo punto será desplazado en la misma magnitud que el primero.

PARÁMETROS:

INT OBJETO : Identificador del objeto a mover

INT X : Nueva coordenada horizontal

INT Y : Nueva coordenada vertical

Para ver esto mejor, estudiemos unos ejemplos. Prueba de escribir esto:

```
program Test_DRAW;
global
    int idcirc;
end
begin
    set_mode(640,480,16);
    set_fps(1,1);
    drawing_map(0,0);
    repeat
        drawing_color(rand(0,65535));
        drawing_alpha(rand(0,255));
        idcirc=draw_circle(rand(0,639),rand(0,479),rand(0,320));
    frame;
```

```

        move_draw(idcirc,400,400);
        frame;
    until(key(_esc))
end

```

Lo que debería de hacer es pintar un círculo en una posición dada (primer frame) y después moverlo para que su centro se situara en (400,400) (segundo frame). Y esto para cada una de las iteraciones del repeat/until. Pero verás que pinta el círculo pero no lo mueve. ¿Por qué? Porque estamos utilizando DRAWING\_MAP en vez de DRAWING\_Z, y MOVE\_DRAW no funciona con DRAWING\_MAP. Ahora ejecuta el siguiente ejemplo, ya modificado:

```

program Test_DRAW;
global
    int idcirc;
end
begin
    set_mode(640,480,16);
    set_fps(1,1);
    drawing_z(1);
    repeat
        drawing_color(rand(0,65535));
        drawing_alpha(rand(0,255));
        idcirc=draw_circle(rand(0,639),rand(0,479),rand(0,320));
        frame;
        move_draw(idcirc,400,400);
        frame;
    until(key(_esc))
end

```

Ahora debería funcionar.

Y para finalizar, un ejemplo interesante:

```

program Test_DRAW;
private
    int idcircle;
end
begin
    set_mode(640,480,16);
    drawing_z(1);
    repeat
        delete_draw(idcircle);
        drawing_color(rand(0,65535));
        idcircle=draw_circle(320,240,200);
        frame;
    until(key(_esc))
end

```

## Trabajar con el teclado (aceptar cadenas de caracteres) :

Supongo que a estas alturas te habrás preguntado si existe algún comando de Fénix que permita la introducción de cadenas de caracteres desde el teclado por parte del jugador. Un ejemplo típico sería cuando en un juego se pregunta el nombre al jugador para inscribirlo en la lista de récords de puntos. O para darle un nombre al protagonista de un RPG, o incluso, en ese RPG, para poder escribir la respuesta a alguna pregunta que se le haga al personaje.

Pues bien. No existe ningún comando que haga eso. Y para poder conseguir una funcionalidad parecida, tendremos que recurrir a ciertas artimañas.

Lo que sí tiene Fenix es una variable global entera llamada *ASCII* que guarda en cada momento un número. Y ese número es el valor ASCII de la tecla pulsada en el último frame o un 0 si no se pulsó ninguna. (Si se han pulsado varias teclas durante el frame las pulsaciones quedan almacenadas en un pequeño buffer que evita la pérdida de pulsaciones).¿Pero qué es eso de los valores ASCII?¿Qué tiene que ver con lo que yo escribo con el teclado?Lo explico.

Todos los PC incorporan grabada nivel de circuitería hardware una tabla, la tabla ASCII. Por ser PC, ya tienen esa tabla incrustada en los chips. Y esa tabla simplemente es una correspondencia entre los caracteres más usuales (letras minúsculas, letras mayúsculas, dígitos, signos de puntuación, espacios en blanco,etc) y un número, el llamado código ASCII para un carácter determinado. La tabla tiene 256 códigos, desde el 0 hasta el 255, por lo que no caben muchos caracteres, sólo los más usuales en el idioma inglés. Por ejemplo, el código ASCII del carácter 'A' (letra a mayúscula) sería el número 65, del código de 'B' sería el 66, el del dígito '0' el 48, el del dígito '1' el 49, el del carácter 'a' (letra a minúscula) el 97, el de la 'b' el 98, el del espacio en blanco el 32,etc. Si quieres saber cuál es el código ASCII de cualquier carácter es muy fácil: o haces un programa que te saque toda la lista, o bien visitas <http://www.asciitable.com>, o bien en cualquier editor de textos o barra de direcciones del explorador tecleas el código que quieras con el teclado numérico del teclado mientras mantienes pulsada la tecla ALT. Y ya verás que se escribe el carácter correspondiente. Por ejemplo, algunas direcciones de páginas web tienen el carácter ~. Este carácter no aparece en ninguna tecla del teclado; por lo que el truco está en imprimirlo utilizando su código ASCII, que es el 126.

Venga, va. Porque soy bueno, aquí tienes un posible código para obtener la lista de caracteres ASCII.

```
PROGRAM ascii_table;

PRIVATE
INT cx = 0;
INT cy = 0;
INT g_char = 0;
END

BEGIN
graph = new_map( 320, 320, 8);
set_center ( 0, graph, 0, 0);
set_mode( 640, 480, 8);
FROM cy = 0 TO 15
    FROM cx = 0 TO 15
        g_char = write_in_map( 0, chr( (cy*16)+cx ), 0);
        map_put( 0, graph, g_char, cx*20, cy*20);
        unload_map( 0, g_char);
    END
END
WHILE (NOT key(_esc)) FRAME; END
```

END

¿Y por qué este lío de asignar un número con un carácter? ¿Para qué sirve? Bueno, pues porque para las máquinas les es muchísimo más fácil trabajar con números que con letras. Muchísimo. Y por tanto, se pensó que se podría trabajar con las letras como si fueran números, y continuar trabajando con ellos igualmente. Teniendo la tabla ASCII ya se tiene todo lo necesario.

Bien, volvamos a donde estábamos. Fenix tiene una variable global entera, la variable ASCII, que almacena el código ASCII del carácter pulsado en el teclado en el frame actual. Si en este último frame no se ha pulsado ninguna tecla, devuelve 0. Por tanto, la variable ASCII se actualiza en cada Frame a 0 si no se ha pulsado nada o a la tecla que se haya pulsado en ese frame concreto. Si se han pulsado varias teclas durante el frame, (que es lo normal porque dependiendo de la velocidad que hayamos configurado con *set\_fps* tendremos tiempo de sobra para pulsar más teclas), las pulsaciones quedan almacenadas en un pequeño buffer (almacén temporal de memoria) que evita la pérdida de pulsaciones.

Por lo tanto, si se quiere recoger la entrada por teclado de un conjunto de caracteres, el truco más evidente será hacer la lectura de los caracteres de uno en uno en un bucle, donde en cada iteración el valor de la variable ASCII se imprimiría por pantalla. Y el bucle finalizaría cuando se pulsara una tecla concreta, como puede ser la tecla ENTER. Por cierto, no confundir la tecla ENTER con el carácter ENTER (que también existe): el carácter ENTER es un carácter más como otro cualquiera que se “imprime” cuando se pulsa la tecla ENTER, pero no es lo mismo una tecla que un carácter (en la tecla “A” se pueden imprimir dos caracteres diferentes: la a mayúscula y la a minúscula. Por si lo quieres saber, el carácter ENTER viene representado por el número 13 (retorno de carro).

Por ejemplo, podríamos utilizar el siguiente código.

```
program hola;
private
    string cadena;
end
begin
    repeat
        if(ascii!=0)
            cadena=cadena+chr(ascii);
            write_var(0,100,100,4,cadena);
        end
        frame;
    until(key(_enter))
loop
    frame;
end
end
```

Este programa hace lo siguiente: consiste en un bucle REPEAT/UNTIL que va a acabar cuando se pulse la tecla ENTER. Mientras no ocurra esto, se van a ir recogiendo los caracteres tecleados y se irán imprimiendo a la vez en tiempo real por pantalla. Para lograr esto, la línea fundamental es *cadena=cadena+chr(ascii)*. Lo que hacemos aquí es actualizar el valor de “cadena” (la variable donde almacenamos lo que tecleamos y que mostraremos por pantalla) a partir del valor que tenía antes concatenándole (acordarse de que para concatenar –“juntar” cadenas se utiliza el símbolo +) el nuevo carácter pulsado en el frame actual.

Este último carácter pulsado viene dado por la expresión *chr(ascii)*. Esta expresión no es más que la

función Chr de Fenix, la cual lo único que hace es convertir el código numérico ASCII que se le ponga como parámetro (en este caso el valor de la variable ASCII) en su carácter correspondiente, que es lo que queremos concatenar a la cadena. Por ejemplo, Chr(97) devolvería "a".

Y seguidamente, una vez actualizada la cadena, se imprime por pantalla. Fíjate en el detalle de utilizar write\_var; se podría haber utilizado el binomio delete\_text/write, pero con write\_var es mucho más elegante.

No dejar de mencionar la existencia de la línea frame;, imprescindible entre otras cosas para que la variable ASCII pueda actualizarse. Y por último, comento el por qué del if. Si no lo pusiéramos, el programa funcionaría igualmente, pero lo que haría es que en cada frame que no se pulsara tecla alguna, concatenaría a la cadena el carácter que tiene el código ASCII 0 –es un carácter especial que no se imprime-, porque la variable ASCII en esos frames siempre vale 0. Por tanto, estaríamos alargando incesantemente "cadena" con caracteres inútiles hasta el infinito, y a la larga el programa petaría porque no podría manejar una cadena tan larga. Por tanto, el if sirve para detectar que si no hay tecla pulsada en ese frame, que no hace falta añadir ningún carácter a la cadena.

Por cierto, existe una función en Fénix, la función Asc, que hace exactamente lo contrario que Chr. Recibe un parámetro que ha de ser un carácter, y lo que devuelve es el código numérico ASCII que le corresponde. Así, Asc("a") devolvería 97.

No obstante, verás que si ejecutas este programa tiene varios problemas. Primero, que cuando se pulsa ENTER aparece un símbolo raro. Segundo, si pulsamos la tecla de retroceso no hace retroceso ninguno y además aparece otro símbolo raro. Y tercero y más importante, que cuando pulsamos una tecla, aparecen tres o cuatro veces repetido el mismo carácter. Esto es porque mantenemos sin querer la tecla pulsada más de un frame, y por tanto, en cada frame se realiza más de una iteración del bucle. Entonces, ¿qué podemos hacer? Mejoraremos el código de la siguiente manera:

```
program hola;
private
    int letra_ant;
    string cadena;
end
begin
    loop
        if (ascii!=letra_ant)
            switch (ascii)
                //para que no haga nada si no se pulsa una letra
                case 0: end

                //para que no se imprima el carácter ENTER
                case 13: break; end

                //para que la tecla de retroceso haga su tarea
                case 8 :
                    cadena=substr(cadena,0,len(cadena)-1);
                end

                //aquí podríamos escribir el código adecuado para el carácter TAB (tabulador)
                case 9: end

                //aquí podríamos escribir el código adecuado al carácter ESC (el de la tecla ESC)
```

```

                                case 27: end

//aquí, el código adecuado al carácter "inicio de línea" (el de la tecla Inicio)
                                case 18: end

//aquí, el código adecuado al carácter "final de línea" (el de la tecla Fin)
                                case 19: end

//Si no se ha pulsado ninguno de los caracteres especiales anteriores, es un carácter "normal" y por lo
//tanto, se imprime por pantalla
                                default:
                                        cadena=cadena+chr(ascii);
                                        write_var(0,100,100,4,cadena);
                                end
                                end
                                end
                                letra_ant=ascii;
                                frame;
                                end
end
end

```

Este código está bastante mejor. Lo que se hace es tener un bucle infinito del cual sólo podremos salir si apretamos el carácter 13 (ENTER). Si no apretamos nada en un frame, fíjate que el case correspondiente (Case 0) no hace nada de nada. También he incluido el esqueleto para dar la posibilidad más adelante de que el programa haga diferentes cosas para los caracteres tabulador, ESC, inicio y fin de línea, ya que si te fijas, éstos continúan mostrando signos raros en pantalla cuando se pulsan y no hacen lo que presumiblemente deberían hacer (tabular y salir, por ejemplo).

Si se pulsa la tecla de retroceso, que tiene el código 8, fíjate lo que hacemos para que realice su función. Le damos un nuevo valor a la variable "cadena", que presumiblemente será el que tenía menos el último carácter. Y para hacer eso utilizamos el comando *Substr*, ya visto anteriormente en alguna ocasión, pero que ahora vamos a comentar en profundidad.

Este comando precisamente sirve para devolver un trocito de cadena (una subcadena) a partir de una cadena más grande. Ves que este comando tiene tres parámetros: la cadena original, un número que es la posición del primer carácter que se quiere "extraer" de la cadena original por el cual comenzará la subcadena (empezando por el 0), y otro número que es el número de caracteres que se quiere que tenga la subcadena, cogiendo los caracteres a partir del primero extraído.

Es más fácil verlo con un ejemplo. Si tenemos la línea *substr("Hola qué tal",2,3)*, lo que devuelve esta línea será la cadena "la ". Porque lo que se ha hecho es, a partir de la cadena original, ir al carácter que ocupa la posición 2, y a partir de él, coger los tres caracteres siguientes. Como el primer carácter es el número 0, el carácter 2 es la ele. Y como hemos dicho que se quieren coger 3 caracteres, se coge la ele, la a y el espacio en blanco. Y ésta es la subcadena obtenida.

La posición de inicio (el segundo parámetro) puede ser mayor o igual a 0 para indicar un carácter desde la izquierda de la cadena (0 para el primero, 1 para el segundo, y así sucesivamente, como hemos visto), o bien un número negativo para especificar una posición desde el final. -1 se refiere al último carácter de la cadena, -2 al anterior, y así sucesivamente. El número de caracteres (el tercer parámetro) puede omitirse, con lo cual la función tomará todos los caracteres que haya desde la posición especificada hasta el final de la cadena.

Por lo tanto, ¿qué hace la línea *cadena=substr(cadena,0,len(cadena)-1);*? Parte de la cadena existente que se ha ido imprimiendo, y, a partir del primer carácter de esta cadena, se obtiene una subcadena con un

determinado número de caracteres. ¿Cuántos? Lo que nos dé la expresión *len(cadena)-1*. Recuerda que *len* lo que hace es devolver el número de caracteres que tiene una cadena pasada como parámetro.

Así que ya lo tenemos. Si ponemos como tercer parámetro de *substr* esto: *len(cadena)-1*, lo que estamos diciendo es que “extraeremos” para la subcadena que queremos crear un número de caracteres tal que sea el mismo que el de la cadena original menos uno. Si empezamos a “extraerlos” desde el principio (segundo parámetro igual a 0), lo que tendremos es pues, una subcadena igual a la cadena original pero con un carácter menos (el del final).

Nota: si como tercer parámetro de *substr* escribimos un número negativo quiere decir lo siguiente: -1 significa el último carácter (sería lo equivalente a *len(cadena)*), -2 el penúltimo carácter, -3 el antepenúltimo carácter, y así. Por tanto, en vez de *len(cadena)-1* podríamos haber puesto -2 con el mismo resultado.

Un ejemplo práctico donde puedes observar los efectos de este comando, si lo ejecutas, sería éste:

```
program Test_SUBSTR;
private
  string Txt="Hola Mundo!";
end
begin
  set_mode(640,480,16);
  write(0,10,30,3,"Texto = Hola Mundo!");
  write(0,10,40,3,"SUBSTR( 0, 3) = "+substr(Txt,0,3));
  write(0,10,50,3,"SUBSTR( 3, 0) = "+substr(Txt,3,0));
  write(0,10,60,3,"SUBSTR(-1, 5) = "+substr(Txt,-1,5));
  write(0,10,70,3,"SUBSTR( 5, -1) = "+substr(Txt,5,-1));
  write(0,10,80,3,"SUBSTR(-3, -1) = "+substr(Txt,-3,-1));
  write(0,10,90,3,"SUBSTR(-1, -3) = "+substr(Txt,-1,-3));
  write(0,10,100,3,"SUBSTR( 9, -1) = "+substr(Txt,9,-1));
  write(0,10,110,3,"SUBSTR(-1, 9) = "+substr(Txt,-1,9));
  write(0,10,120,3,"SUBSTR(-9, -1) = "+substr(Txt,-9,-1));
  write(0,10,130,3,"SUBSTR(-1, -9) = "+substr(Txt,-1,-9));
  repeat
    frame;
  until(key(_esc))
end
```

Y volviendo al código, ves que si no se pulsa ninguna de estos caracteres especiales, es entonces cuando ejecutamos el default, y por tanto, imprimimos por pantalla el carácter adecuado.

Por último, he de comentar el *if(ascii!=letra\_ant)*. Este if es el responsable de que no aparezcan los caracteres repetidas veces cuando pulsamos una tecla. Fíjate que en cada fotograma (antes de la línea *frame*;) está la línea *letra\_ant=ascii*; Con esta línea lo que hacemos es utilizar una variable de refuerzo para almacenar el código ASCII del carácter pulsado en el frame actual –justo antes de pasar al siguiente-. De tal manera, que cuando se vuelve a comprobar la condición del switch, como estamos ya en el siguiente fotograma (la línea *frame*; se ha acabado de ejecutar), la variable ASCII ya valdrá otro valor. Y lo que comprueba el if es si este nuevo valor de ASCII coincide con el valor del frame anterior, guardado en “*letra\_ant*”. Si es igual, no hace nada de nada y se va directamente a la siguiente iteración del Loop, y así hasta que por fin se detecte que el valor de la variable ASCII ha cambiado (porque ya no se pulsa ninguna tecla o porque se pulsa otra). El pequeño inconveniente de este código es que no detectará palabras que tengan dos caracteres iguales seguidos (“lluvia” la tomará por “luvia”, “llaves” como “laves”, etc.

Otro detalle (que es fácil modificar: lo dejo como ejercicio) es evitar dejar que el usuario pueda escribir cadenas infinitas de caracteres, como hasta ahora permite el programa. Simplemente habría que comprobar que la longitud de la cadena introducida hasta el momento fuera menor que un límite máximo fijado por nosotros.

Finalmente, sería muy interesante hacer de este código, (con las modificaciones que tú creas oportunas) una función, con los parámetros que tú consideres, de manera que cada vez que se necesite pedir al jugador que escriba algún texto no se tenga que escribir siempre la parrafada anterior sino que pudieras invocar en una sola línea a una función que contuviera este código listo para ejecutar. .

Esto último es lo que se ha pretendido con el código siguiente. La finalidad es la misma que el anterior, recoger la entrada por teclado, pero este ejemplo es más sofisticado, ya que también detecta el tiempo de pulsación de las teclas para cerciorarse de que es una pulsación "válida", no accidental (por ejemplo, para detectar que efectivamente queremos pulsar la tecla varias veces seguidas).

*Program texts;*

*Declare Function*

*String textinput();*

*End*

*Begin*

*set\_fps(60,0);*

*say("Escribiste: " + textinput() + " ");*

*Repeat*

*frame;*

*Until(key(\_esc));*

*End*

*Function String textinput()*

*Private*

*String str;*

*int t;*

*int t2;*

*byte last\_ascii;*

*int txtid;*

*Begin*

*//Muestra lo que escribes en la esquina superior izquierda*

*txtid = write\_var(0,0,0,0,str);*

*//Limpia la cadena*

*str = "";*

*//Recoge la entrada del usuario. Pulsando Enter el loop termina.*

*Loop*

*//Chequea si una tecla es presionada y si la misma tecla fue presionada en el último frame*

*if(ascii!=0&&last\_ascii==ascii)*

*//Chequea si la tecla fue presionada levemente o si ha sido presionada más de 0.25 segundos*

*if(t==0||t>fps/4)*

*//Chequea si la tecla fue presionada levemente o si ha estado presionada en los últimos 0.03 segundos*

*if(t==0||t2>fps/30)*

*t2=0;*

*switch(ascii) // Gestionar entrada*

*case 8: //Tecla retroceso (backspace)*

*str = substr(str,0,len(str)-1);*

*end*

```

        case 13: //Tecla enter
            break;
        end
        default: //Añade la tecla
            str=str+chr(ascii);
        end
    end
end
t2++;
end
t++;
else
    t = t2 = 0; // Reset
end
last_ascii = ascii;
frame;
End
//Borrar el texto utilizado
delete_text(txtid);
//Retorna la cadena tecleada
return str;
End

```

### **Trabajar con el teclado (cómo redefinirlo):**

(extraído del tutorial de Wakroo, en <http://divnet.divsite.net>)

Una de las características que más a menudo se suele olvidar pero que más necesaria considero es la opción de redefinir el teclado. Muchos juegos traen una serie de teclas por defecto que son las que todos tienen que usar para jugar. A menudo están elegidas de una manera adecuada, pero puede ser que a alguien no le guste o le resulte incómodo. La solución es dar al jugador la opción de escoger sus propias teclas para jugar. Además, resulta muy fácil de hacer.

Lo primero que hay que tener saber para redefinir el teclado es cómo funciona. Cada tecla tiene asignado por Fénix un valor numérico determinado. Ojo, no estamos hablando de la tabla ASCII, la cual es estandar para todos los PC y afecta a los caracteres (por lo que “a” y “A” tienen códigos ASCII diferentes), sino que estamos hablando de que Fénix asocia un número a cada tecla del teclado (por lo cual “a” y “A”, al estar en la misma tecla, tendrán el mismo código).

El programa lo único que hace es detectar el código que le viene y reacciona en consecuencia. Por ejemplo, el código de la tecla "a" es el 30, por lo que sería lo mismo poner `key(_a)` o `key(30)` –de hecho, “\_a” no es más que una constante predefinida por Fénix para ayudarnos a no tener que acordarnos de cuál era el código de la tecla “a”/“A”. Y aquí está la clave de todo el misterio.

Una vez que conocemos la naturaleza numérica de las teclas la solución al problema es usar variables a la hora de detectar si el jugador ha pulsado las teclas adecuadas. Si creamos una variable llamada “arriba” y le damos a esa variable el valor numérico que corresponda a la tecla que queremos, luego comprobamos si ha sido pulsada con `key(arriba)`. Si en cualquier momento variamos el valor de “arriba” se cambiará la tecla que responde a esa acción.

Ahora el problema se presenta en qué valores guardar en la variable. Se pueden poner los valores desde programación, pero eso implicaría tener que recompilar el juego cada vez que se cambie la configuración del teclado, además de que tendríamos que pasar el código a la gente. La gente que no se dedique a programar muchas veces ni se molestará en investigar cómo hacer para cambiar las cosas, por muy bueno que sea el juego. Por lo tanto, no es una buena opción. Por lo tanto, habrá que darle la opción al

usuario desde el propio programa para que haga los cambios que desee. Bueno, pues montamos el típico menú desde donde hacerlo. Pero, ¿cómo nos las arreglamos para detectar qué tecla ha elegido?

La solución más directa sería poner un IF (key(...)) variable=...; END para cada tecla que podamos elegir. En principio eso significaría muuuuchas líneas de código y sería bastante engorroso. Hay otra solución más elegante: **SCAN\_CODE**. Ésta es una variable global entera predefinida que guarda el código numérica de la última tecla que ha sido pulsada por el jugador (con buffer para cuando hay muchas y evitar su pérdida), Por lo tanto, comprobando su valor podríamos saber qué tecla ha elegido el jugador y asignarla a la variable. De esta manera, con poco código tendremos la posibilidad de elegir cualquier tecla.No confundir el uso de SCAN\_CODE con el de la función *key*: esta función nos indica simplemente si la tecla con el código X está o no pulsada en el frame actual (devuelve “true” o “false”). En cambio, SCAN\_CODE c

Y ¡jojo!, no confundir el significado de la variable SCAN\_CODE con el de la variable ASCII: la primera admite valores que representan teclas del teclado, la segunda admite valores que representan caracteres dentro de la tabla ASCII: la letra “a” y “A” se escriben con la misma tecla, por lo que SCAN\_CODE valdrá lo mismo en ambos casos, pero ASCII no, porque “a” tiene un código ASCII y “A” tiene otro. Cuidado.

Naturalmente, se pueden usar truquillos como usar arrays o structs en vez de variables sueltas para guardar los códigos de las teclas, pero eso ya es una cuestión a parte. El método de todas formas tiene una pega: el uso de SCAN\_CODE es más lento que usar la función key(). Por lo tanto, no es adecuado para usarlo en tiempo real. En el menú no importa mucho si hay que darle un par de veces para que se entere de la tecla, pero si intentas dar ese salto que te va a llevar a la salvación y el ordenador se niega a responder a nuestra órdenes con presteza... Creo que me entendéis.

Como ejemplo de lo dicho, aquí tienes un código que redefine la tecla para mover un gráfico a la derecha o a la izquierda. Para que funcione debes crear una imagen PNG llamada “raton.png”

```
Program Redefinir_teclas;
Global
  Int derecha;
  Int izquierda;
end
Begin
  set_mode (320,240,16);
  // Se define la tecla para ir a la derecha
  write(0,0,0,0,"Pulse tecla para derecha");
  Repeat
    derecha=scan_code;
    Frame;
  Until (derecha<>0)

  // Se define la tecla para ir a la izquierda
  delete_text(0);
  write(0,0,0,0,"Pulse una tecla para izquierda");
  Frame(500);
  Repeat
    izquierda=scan_code;
    Frame;
  Until (izquierda<>0)

  delete_text(0);
  write(0,0,0,0,"Pulse esc para terminar");
```

```

// Se pone el gráfico en pantalla
x=160; y=120;
graph=load_png("raton.png");

Loop
// Se mueve el gráfico
If (key(derecha)) x++; End
If (key(izquierda)) x--; End

If (key(_esc)) exit(); End
Frame;
End
End

```

Existe otra variable global predefinida relacionada con el teclado interesante, aparte de `SCAN_CODE`, llamada ***SHIFT\_STATUS***. Esta variable almacena el estado de las teclas de control; es decir: contiene el estado de las diferentes teclas de control (CTRL, SHIFT, ALT). Según su valor, sabremos si se está pulsando las siguientes teclas (además de cualquier otra tecla adicional recogida por `SCAN_CODE` o la función `key()`). Sus valores pueden ser los siguientes (y la suma de éstos para detectar pulsaciones simultáneas de varias teclas de control)

- 1: Se mantiene pulsada la tecla SHIFT (MAYUS) izquierda
- 2: Se mantiene pulsada la tecla SHIFT (MAYUS) derecha
- 4: Se mantiene pulsada la tecla CONTROL (cualquiera de ellas)
- 8: Se mantiene pulsada la tecla ALT (cualquiera de ellas)

### **Trabajar con el ratón:**

El ratón no se programa como un proceso normal, su funcionamiento se basa en una estructura, la estructura “mouse”: se consultan y asignan sus valores y no necesita ningún proceso.

Para consultar y asignar la posición del cursor del ratón tenemos las variables “**mouse.x**” y “**mouse.y**”. Estas coordenadas son respecto a la ventana y sus valores máximos dependen del modo de pantalla (800x600, 640x480,...).

También puedes ponerle al cursor del ratón el dibujo del puntero que quieras: sólo tienes que asignarle los valores adecuados a la variable “**mouse.file**” y “**mouse.graph**”. También puedes cambiarle el tamaño, como si del gráfico de un proceso se tratara, con “**mouse.size**”. Y sus flags, al igual que cualquier otro gráfico, con “**mouse.flags**” (con los mismo valores posibles). Y su orientación con “**mouse.angle**”. Y su profundidad con “**mouse.z**”. También podemos especificar la región de la pantalla donde queremos que sólo allí sea visible el puntero, con “**mouse.region**” -el tema de las regiones se tratará en este capítulo más adelante-.

Para comprobar la pulsación de los botones puedes comprobar, como si fueran teclas del teclado, las variables “**mouse.left**”, “**mouse.right**” y “**mouse.center**” para ver si se ha pulsado el botón izquierdo, derecho o central respectivamente.

Puedes hacer uso también de la función *collision* de la forma “**collision (type mouse);**” para comprobar si el gráfico de un proceso ha hecho contacto con el “punto caliente” del cursor (el punto que normalmente es la punta de la flecha, la que selecciona el objeto), algo muy útil para hacer que cambie éste con el contacto.

Como ves, tienes un control completo sobre el ratón y con un manejo sencillísimo. Un ejemplo:

```
program tutorial;
begin
    set_mode(320,240,16);
    proceso();
    mouse.file=0;
    mouse.graph=new_map(20,20,16);map_clear(0,mouse.graph,rgb(205,24,46));
    write_var(0,100,20,4,mouse.x);
    write_var(0,100,40,4,mouse.y);
    write_var(0,100,60,4,mouse.angle);
    write_var(0,100,80,4,mouse.size);
    write_var(0,100,100,4,mouse.flags);
    write_var(0,100,120,4,mouse.z);
    write_var(0,100,140,4,mouse.region);
    loop
//Por si se quiere mover el ratón con el teclado
        if(key(_x)) mouse.x=mouse.x+5;end
        if(key(_y)) mouse.y=mouse.y+5;end
//Gira la orientación del gráfico del puntero
        if(key(_a)) mouse.angle=(mouse.angle+5000)%360000;end
//Aumenta el tamaño del gráfico del puntero
        if(key(_s)) mouse.size=mouse.size+10; end
/*El while está puesto para que mientras se tenga pulsada la tecla no se haga nada, y sólo se cambie el
flag una vez se ha soltado. Esto es para evitar que, si se mantiene pulsada más de la cuenta la tecla, el flag
varíe demasiados valores de golpe*/
        if(key(_f)) while(key(_f))frame;end mouse.flags=(mouse.flags+2)%16;end
//Aun teniendo Z>0, no se pinta por debajo del proceso (!!!?)
        if(key(_z)) mouse.z++; end
//Si la región deja de ser la n°0, que representa la pantalla completa, el puntero deja de ser visible
        if(key(_r)) mouse.region++;end
        if(mouse.left==true) write(0,100,160,4,"Botón izquierdo pulsado");end
        if(mouse.right==true) write(0,100,180,4,"Botón derecho pulsado");end
        if(key(_esc)) exit(); end
        frame;
    end
end

Process proceso()
private
    int a;
end
begin
    graph=new_map(50,50,16);map_clear(0,graph,rgb(255,255,0));
    x = 160;
    y = 120;
    loop
        if(key(_up))y=y-5;end
        if(key(_down))y=y+5; end
        if(key(_left))x=x-5; end
        if(key(_right))x=x+5; end
        if(collision(type mouse)) write(0,100,200,4,"¡Colisión!");end
    frame;
end
```

end

No obstante, suele ser habitual que en vez de utilizar la estructura MOUSE “a pelo”, se utilice para controlar el puntero del ratón un proceso normal y corriente que lo único que haga sea actualizar permanentemente la posición de su gráfico a la del puntero (invisible en este caso). Con esta manera de hacer, se gana flexibilidad y versatilidad (sobre todo en la detección de colisiones) como veremos en los ejemplos siguientes.

Cabe destacar también que de este modo, los únicos campos que se utilizarán de la estructura MOUSE son “mouse.x” y “mouse.y”, para asignar su valor en cada iteración a la X e Y del proceso en cuestión, pero ni “mouse.file”, ni “mouse.graph”, ni “mouse.size”, ni “mouse.angle” ni “mouse.flags” ni “mouse.z” ni “mouse.region” tendrán utilidad, ya que en su lugar usaremos las correspondientes variables locales para ese proceso en cuestión (FILE, GRAPH, etc). Como muestra, a continuación pongo el mismo ejemplo anterior pero haciendo que el puntero del ratón sea un proceso “estándar”.

```
program tutorial;  
begin
```

```
    set_mode(320,240,16);  
    proceso();  
    raton();
```

```
/*Podemos ver que si modificamos el ANGLE,SIZE,etc del proceso "raton" -y observamos su efecto en el gráfico del puntero- los valores de la estructura mouse (cuyo gráfico de puntero es invisible) no se ven modificados para nada*/
```

```
    write_var(0,100,60,4,mouse.angle);  
    write_var(0,100,80,4,mouse.size);  
    write_var(0,100,100,4,mouse.flags);  
    write_var(0,100,120,4,mouse.z);  
    write_var(0,100,140,4,mouse.region);  
    loop  
        if(mouse.left==true) write(0,100,160,4,"Botón izquierdo pulsado");end  
        if(mouse.right==true) write(0,100,180,4,"Botón derecho pulsado");end  
        if(key(_esc)) exit(); end  
        frame;  
    end
```

```
end
```

```
process raton()
```

```
begin
```

```
    file=0;  
    graph=new_map(20,20,16);map_clear(0,graph,rgb(205,24,46));  
    loop
```

```
/*Estas dos líneas siguientes simularán que este proceso es el puntero de ratón. IMPORTANTE ponerlas dentro de un bucle, para ir actualizando las coordenadas a cada frame.*/
```

```
        x=mouse.x;  
        y=mouse.y;  
        if(key(_x)) x=x+5;end  
        if(key(_y)) y=y+5;end  
        if(key(_a)) angle=(angle+5000)%360000;end  
        if(key(_s)) size=size+10; end  
        if(key(_f)) while(key(_f))frame;end flags=(flags+2)%16;end  
        if(key(_z)) z++; end  
        if(key(_r)) region++;end  
        frame;  
    end
```

```

end

Process proceso()
private
    int a;
end
begin
    graph=new_map(50,50,16);map_clear(0,graph,rgb(255,255,0));
    x = 160;
    y = 120;
    loop
        if(key(_up))y=y-5;end
        if(key(_down))y=y+5; end
        if(key(_left))x=x-5; end
        if(key(_right))x=x+5; end
//Ahora detecto la colisión con el proceso "ratón", no con la estructura MOUSE
        if(collision(type raton)) write(0,100,200,4,"¡Colisión!");end
    frame;
end
end

```

A partir de aquí, se nos pueden ocurrir muchas aplicaciones del cursor del ratón en nuestros juegos. Por ejemplo:

¿Cómo podríamos hacer por ejemplo que un proceso cualquiera estuviera mirando constantemente al movimiento del ratón?. Pues una manera sería:

```

program prueba;
begin
    set_mode(640,480,16);
    set_fps(30,0);

    observador();
    destino();
    loop
        frame;
    end
end

process observador()

private
    int angulo;
end
begin
    x=320,y=240;
    graph = load_png("observador.png");
    loop
        angle=get_angle(get_id(type destino));
        frame;
    end
end

process destino()

```

```

begin
    graph = load_png("raton.png");
    loop
        x=mouse.x;
        y=mouse.y;
        frame;
    end
end

```

No creo necesario explicar nada: espero que se entienda todo: *get\_angle* devuelve el ángulo entre la horizontal y el proceso observado cuyo identificador se le pasa como parámetro (identificador, por cierto, que en este ejemplo lo obtenemos gracias a *get\_id*). Y el ANGLE de cualquier proceso sabemos que es el ángulo entre la línea horizontal y donde está apuntando en este momento. Por lo tanto: al igualar ambos valores estamos diciendo que el ángulo respecto a la línea horizontal al que ha de apuntar el proceso observador sea el mismo que el ángulo respecto a la línea horizontal donde está el proceso “destino”: es decir, que el proceso “observador” apunte a “destino”.

A lo mejor a ti se te hubiera ocurrido poner *angle=get\_angle(mouse)*; Sin embargo, no habría funcionado porque *mouse* no es un proceso, es una estructura.

Una alternativa a la ya escrita hubiera sido sustituir la línea *angle=get\_angle(get\_id(type destino))*; por ésta otra: *angle=fget\_angle(x,y,mouse.x,mouse.y)*; Aquí hacemos uso de una función hasta ahora desconocida: *fget\_angle*, cuya referencia la encontrarás más adelante dentro de este mismo capítulo. Básicamente, lo que hace esta función es calcular el ángulo (en milésimas de grado) formado por una línea determinada y la línea horizontal (el eje X). Los cuatro parámetros que tiene son respectivamente la coordenada X del primer punto de la línea, la coordenada Y del primer punto de esa línea, la coordenada X del segundo punto de esa línea y la coordenada Y del segundo punto de esa línea (una línea se forma por la unión de dos puntos, claro).

Bueno, volviendo a lo que teníamos entre manos, fijate como cosa curiosa que con el proceso “destino()” tenemos una manera de cambiar el icono del ratón.

Otro truco muy útil es hacer que el gráfico de un proceso se dirija allí donde se haya hecho clic con el ratón. ¿Cómo se podría hacer esto? Imaginemos que tenemos un proceso “personaje()”. Lo que necesitaremos hacer es, primero, crear dos variables globales, por ejemplo “posx” y “posy”, y seguidamente, escribir en el bucle Loop principal del proceso “personaje()” algo así:

```

Loop
    if(mouse.left==true)
        posx=mouse.x;
        posy=mouse.y;
        mouse.left=false;
    end
    if(x>posx); x=x-1; end
    if(x<posx); x=x+1; end
    if(y>posy); y=y-1; end
    if(y<posy); y=y+1; end
    frame;
end

```

También está claro, ¿no?

A continuación pongo un código muy curioso, que nos permite dibujar en la pantalla con el puntero del ratón igual que si utilizáramos alguna herramienta de dibujo a mano alzada, tipo lápiz. (Para que el programa funcione, deberás crear un archivo FPG llamado "bola.fpg", que contendrá una única imagen con código 100 y que será precisamente, una bola).

Además, este ejemplo también muestra cómo sustituir el cursor del ratón por cualquier dibujo que queramos. En concreto, esto se realiza simplemente asignando a la variable x e y del proceso cuyo dibujo sustituirá el cursor los valores mouse.x y mouse.y, respectivamente. Es decir, en realidad estamos superponiendo el gráfico que nosotros queremos sobre el cursor, y haciendo que aquél vaya siguiendo en todo momento los movimientos de éste.

*Program prueba;*

*Private*

*Int fichero1;*

*end*

*Begin*

*set\_mode(640,480,16);*

*fichero1=load\_fpg("bola.fpg");*

*write(0,10,20,0,"DIBUJA CON EL MOUSE");*

*definemouse();*

*While (NOT key(\_esc))*

*Frame;*

*End*

*fade\_off();*

*exit("");*

*End*

*//-----*

*Process dibuja(x,y)*

*Begin*

*flags=4;*

*graph=100;*

*Loop*

*Frame;*

*End*

*End*

*//-----*

*Process definemouse()*

*Begin*

*mouse.x=270;*

*mouse.y=240;*

*Loop*

*graph=100;*

*If (mouse.x>640) mouse.x=640; End*

*If (mouse.x<0) mouse.x=0; End*

*If (mouse.y>480) mouse.y=480; End*

*If (mouse.y<0) mouse.y=0; End*

*If (mouse.left) dibuja(mouse.x,mouse.y); End*

*x=mouse.x;*

*y=mouse.y;*

*Frame;*

*End*

*End*

El siguiente ejemplo es un mero efecto estético, similar al que visto anteriormente que simulaba un pincel, pero con un comportamiento diferente, más sofisticado y espectacular. Este efecto es ideal para simular gases de propulsión, lanzallamas, etc. Necesitarás una imagen de unos 10x10 píxeles llamada "a.png" para poderlo ejecutar:

```

program lanzallamas;
global
    int idpng;
end
begin
idpng=load_png("a.png");
mouse.graph=idpng;
loop
    if (mouse.right) llama(mouse.x,mouse.y); end
    if (key(_esc)) exit(); end
    frame;
end
end

process llama(x,y)
private
    byte i;
end
begin
graph=idpng;
size=rand(5,20);
angle=rand(0,360)*1000;
flags=4;
repeat
    x=x+5;
    z++;
    angle=(angle+5000)%360000;
    i=i+5;
    frame;
    size=size+7;
until(i>110)
end

```

Otro código interesante: lo que hace es alterar la sensibilidad del ratón con un par de sencillos procesos que podemos trasladar a cualquiera de nuestros programas. De esta manera, podremos hacer que el cursor del ratón vaya más ligero o que le cueste moverse (como si pesara toneladas), ofreciendo así un efecto muy resultón:

```

//AUTHOR: Tristan
program sensibilitymouse;
const
    SENSIBILITY_FAST=1;
    SENSIBILITY_SLOW=5;
end
global
//Es float por si queremos hacer sensibilidades decimales, o menores de 1 para hacer el cursor ultrarápido
float sensibility;
end

```

```

process main()
begin
  set_mode(640,480,16);
  sensibility= SENSIBILITY_FAST; //Sensibilidad inicial
  write(0,10,10,0,"Pulsa (R)ápido (L)ento");
  mouse_test();
  loop
    if (key(_R)) sensibility= SENSIBILITY_FAST; end
    if (key(_L)) sensibility= SENSIBILITY_SLOW; end
  frame;
  end
end

process mouse_test()
begin
  //Este proceso representa el puntero del ratón
  x = mouse.x;
  y = mouse.y;
  graph = create_mouse_pointer();
  loop
    /*Lo importante del ejemplo: la nueva posición del gráfico del proceso vendrá dada por el valor de
    "sensibility". Se puede ver que si ésta vale 1 (rápida),  $x=x+(mouse.x-x)/1 \rightarrow x=mouse.x$ ; pero si la
    sensibilidad vale 5 (lenta),  $x=x+(mouse.x-x)/5 \rightarrow x=5*mouse.x + 4*x/5$ . Se pueden buscar otras fórmulas
    para simular el mismo efecto.*/
    x = x +(mouse.x-x)/(sensibility);
    y = y +(mouse.y-y)/(sensibility);
  frame;
  end
end

function create_mouse_pointer()
private
  int copo;
end
begin
  copo=new_map(3,3,8);
  /*Todas las primitivas gráficas que se utilicen a partir de ahora,si no se dice lo contrario, se van a pintar
  sobre la imagen acabada de generar -inicialmente vacía de contenido gráfico-. Esto es una manera de
  decir que se rellenará de contenido gráfico esta imagen con las primitivas gráficas que usemos a partir de
  ahora.*/
  drawing_map(0,copo);
  //Dibujamos un "copo de nieve" sobre la imagen "copo"
  drawing_color(RGB(250,250,250));
  draw_line(1,1,3,3);
  draw_line(2,1,2,3);
  drawing_color(RGB(255,255,255));
  draw_line(1,2,3,2);
  draw_line(3,1,1,3);
  /*Devolvemos el identificador de la imagen creada con new_map, (y rellena de contenido gráfico con
  las primitivas sucesivas). Este identificador devuelto puede ser utilizado para cualquier cosa que
  necesite asignarse a un código de imagen, como por ejemplo, en este caso, la variable global GRAPH. De
  esta manera, la imagen del proceso mouse_test será la correspondiente al identificador devuelto, que es
  el "copo de nieve".*/

```

```
return copo;
end
```

El siguiente ejemplo es una implementación para detectar un doble click. El código muestra un cuadrado en el centro de la pantalla, sobre el cual se podrá hacer click con el cursor del ratón (y fuera del cuadrado también, claro), pero solamente cuando se haga doble click dentro del cuadrado, se notificará con un mensaje. Si se hace doble click fuera de él, o clicks simples tanto dentro como fuera, no pasará nada.

```
//DESARROLLADOR: DIEGO KUNZEVICH (DIVNETICO)
PROGRAM doble_clic;
BEGIN
set_title("Test doble clic");
set_mode(640,480,16,MODE_2XSCALE);
process_mouse();
caja_comprobacion();
END

PROCESS process_mouse()
BEGIN
mouse.graph = new_map(20,20,16);map_clear(0,mouse.graph,rgb(100,100,200));
mouse.x=320;mouse.y=240;
LOOP
if (key(_esc)) exit(); end
FRAME;
END
END

PROCESS caja_comprobacion()
private
//Vale 0 si no se ha hecho ningún click, 1 si se acaba de hacer uno y 2 si se acaba de hacer dos
int clickeo;
//Tiempo entre un primer click y el segundo
int stop;
//Tiempo después del segundo click
int time;
end
BEGIN
graph = new_map(50,50,16);map_clear(0,graph,rgb(255,0,255));
x = 300;y = 240;
LOOP
//Si es la primera vez que se clica sobre el proceso...
if ((collision(type mouse) AND (mouse.left) AND (clickeo==0))) clickeo = 1; end
/*A partir de hacer un click, se aumenta a cada frame "stop". Si "stop" supera el valor 8 (es decir, si pasan
ocho frames -esto se puede regular-), se resetea al estado inicial*/
if (clickeo==1) stop++; end
if (stop>=8) stop=0; clickeo = 0; time=0; end
/*Si es la segunda vez que se clica sobre el proceso...(y ese segundo click se ha hecho relativamente
separado del primero -porque "stop" ha de valer más de 4 y no valdrá nunca más de 8...-) */
if ((collision(type mouse) AND (mouse.left) AND (clickeo==1) AND (clickeo==1) AND (stop>=4)))
clickeo = 2; end
/*Si se ha hecho doble click, se notifica y se aumenta a cada frame "time". Si "time" supera el valor 14
(es decir, si pasan 14 frames -esto se puede regular-), se borra la notificación y se resetea al estado
inicial*/
if ((clickeo==2)) write(0,400,240,4,"Segundo clickeo"); time++; end
```

```

if (time>=14) delete_text(0); clickeo=0; time=0; stop=0; end
frame;
END
END

```

Finalmente, no está de más comentar la existencia de la variable global **MOUSE\_STATUS**, la cual vale 1 cuando el puntero del ratón está dentro de los límites de la ventana (si la aplicación se ejecuta en modo ventana) y 0 cuando sale de ella. Este valor no tiene utilidad si la aplicación se ejecuta en modo de ventana modal (usando el parámetro MODE\_MODAL en la llamada a SET\_MODE).

En este apartado veremos cómo implementar la funcionalidad **Drag&Drop** en nuestros juegos. Drag&Drop (literalmente, “Arrastrar y Soltar”) es el efecto, por todo usuario de Windows conocido, de hacer clic con algún botón del ratón sobre una imagen o icono, de arrastrar esa imagen a lo largo y ancho de la pantalla (manteniendo pulsado el botón del ratón), y finalmente de soltar el botón cuando deseemos, instante en el que la imagen permanecerá fija en esa posición donde hayamos soltado el botón. Drag&Drop es una característica que puede dotar a nuestro programa de una gran vistosidad: es muy útil sobretodo por ejemplo para juegos de rompecabezas y puzzles.

Voy a poner ahora de golpe todo el código fuente que utilizaremos en este apartado, para posteriormente irlo comentando línea a línea. Si ejecutas el ejemplo verás que es sencillo: sobre un fondo aparecerán unas cuantas “fichas” que tienen cada una de ellas una letra impresa. También tendremos un icono para el ratón, el cual podremos situar encima de una ficha cualquiera y arrastrarla y soltarla allí donde nos plazca dentro de la ventana. En el código no hay implementado la opción de detectar colisiones entre fichas, pero no sería muy difícil añadirlo, y con un poco más de programación se podría conseguir un pequeño juego de puzzle. Animo fervientemente al lector a que intente llevar a cabo este proyecto tan interesante.

Bien. El código fuente a estudiar es éste:

```

/* Autor: Moogle or Roelf Leenders */

program demostracion_dragdrop;

global
  /*Declaramos una tabla conteniendo todos los caracteres que necesitaremos para irlos arrastrando*/
  String example_text[8] = 'D','r','a','g','&','D','r','o','p';
  //Algunas variables de ayuda
  int graphic1;
  int graphic2;
  /*Estas variables contendrán coordenadas relativas y el ID del proceso que estemos arrastrando en ese momento.*/
  int relativeX;
  int relativeY;
  int dragID;
end

begin
  set_title("Ejemplo de Drag&Drop");
  set_mode(320,240,16);
  /*Cambio el color de texto para que sea más bonito que no blanco*/
  set_text_color(rgb(150,150,150));
  /*El fondo siempre puede ser accesible como el gráfico número 0 de la librería del sistema (el FPG número 0)*/

```

```
map_clear(0,0,rgb(10,10,200));
write_var(0,0,0,0,dragID);
```

*/\*A continuación vienen las rutinas de generación dinámica de los gráficos y procesos que los contienen. No es lo más importante del programa; se usa este sistema principalmente porque así es más fácil enviar el código a otra persona: sólo hay que enviarle el .prg sin necesidad de utilizar otros molestos archivos como los FPG.*

*El código de generación puede parecer bastante complicado a primera vista y realmente no incumbe a aquellos que quieran aprender exclusivamente cómo funciona las bases del Drag&Drop, así que no estará muy documentado.*

*La idea principal para crear el puntero del ratón dinámicamente con forma de flecha es definir unos criterios para que los puntos pertenezcan al gráfico flecha, y si no los cumplen, éstos serán de color negro. Chequea este funcionamiento si quieres, alterando el código como consideres oportuno para ver los posibles cambios en el dibujo.\*/*

```
//Dibujaremos una flecha que será el puntero del ratón:
mouse.graph = new_map(30,30,16);
set_center(0,mouse.graph,0,0);
for(x=0;x<30;x++)
    for(y=0;y<30;y++)
        if( (x+y =< 30 and abs(fget_angle(0,0,x,y)-fget_angle(0,0,30,30)) < 25000 )
or (abs(x-y) < 4 and x+y > 20))
            map_put_pixel(0,mouse.graph,x,y,rgb(254,254,254));
        end
    end
end
end
```

*/\*Ya que 30x30 es bastante grande para un puntero de ratón en esta resolución (320x240), muéstralo al 50% de su tamaño\*/*

```
mouse.size = 50;
```

*/\*El código siguiente crea un gráfico de 1x1 pixel para servir como puntero efectivo del ratón. En casos donde utilices un puntero de ratón y planees utilizar comandos de colisión con él, a menudo es mejor utilizar dos gráficos: uno para el puntero del ratón (típicamente una flecha) y bajo éste, un gráfico de 1x1 para chequear las colisiones. Haciendo esto evitarás errores como que puedas arrastrar cosas con la cola de la flecha, por ejemplo: el puntero efectivo del ratón será una porción mucho más pequeña y precisa de lo que en realidad se ve por pantalla. En este ejemplo uso una flecha como gráfico del puntero del ratón (en la estructura mouse) pero el programa tiene un proceso representado por pequeño gráfico que será el puntero efectivo. Dicho gráfico está coloreado blanco porque píxeles transparentes recuerda que no colisionan\*/*

```
puntero();
```

*/\*Aquí es donde los gráficos con las letras se generan y los procesos "letter()" son creados también. Uso un FOR para generar 9 procesos "letter()" -x va de 0 a 8-. Como texto uso la tabla declarada al principio en la sección global y rellena con letras.\*/*

```
for(x=0;x<9;x++)
    //Creo un nuevo gráfico
    graphic1 = new_map(25,25,16);
    /*Lo pinto de gris oscuro (no negro, porque sería transparente)*/
    map_clear(0,graphic1,rgb(20,20,20));
```

*/\*Creo otro gráfico, negro con la letra actual (posicionada dentro de la tabla como elemento x)\*/*

```
graphic2 = write_in_map(0,example_text[x],4);
```

*/\*Superpongo el gráfico negro sobre el gris oscuro, obteniendo así un dibujo que será un cuadrado gris con una letra encima\*/*

```

        map_put(0,graphic1,graphic2,12,12);
        /*Creo un nuevo proceso con este gráfico, en una posición calculada de la pantalla*/
        letter(160+(x-4)*30,120,graphic1);
    end

    loop
        if(key(_esc))exit();end
        frame;
    end
end

/* Proceso que contiene un gráfico específico en una posición específica. No mucho que comentar*/
process letter(x,y,graph)
begin
    loop
        frame;
    end
end

/*Proceso REALMENTE importante para entender el funcionamiento del Drag&Drop.*/
process puntero()
begin
    graph = new_map(1,1,16);
    map_clear(0,graph,rgb(254,254,254));

    /*Copio las coordenadas del ratón (ya que allí es donde queremos que esté el puntero "efectivo")*/
    loop
        x = mouse.x;
        y = mouse.y;
        //Si se clicca con el botón izquierdo del ratón, adelante
        if(mouse.left)
            //Gestiona la colisión de la letra (o no si no hay ninguna)
            dragID = collision(type letter);
            /*Pero si se chequea y sí existe alguna colisión, guardo las coordenadas de la posición
            relativa entre el centro de la ficha y el centro del puntero efectivo*/
            if(exists(dragID))
                relativeX = mouse.x - dragID.x;
                relativeY = mouse.y - dragID.y;
            end
        end
    loop
    /*Ahora necesitamos arrastrarlo mientras el jugador mantenga pulsado el botón izquierdo del ratón*/
    while(mouse.left)
        /*Y mientras eso ocurre, necesitamos no olvidar mantener sincronizadas las coordenadas del puntero del
        ratón con las de nuestro proceso, el puntero "efectivo", porque si no va a aparecer un punto blanco fijo
        mientras hagamos el Drag&Drop*/
        x = mouse.x;
        y = mouse.y;
        /*Aquí está el punto crucial. Si un proceso se detecta que colisiona a la vez que el clic, necesita ser movido
        cada fotograma mientras el botón del ratón permanezca pulsado. Esto quiere decir que las coordenadas
        del proceso deberían moverse con exactamente las mismas cantidades de píxeles que el ratón.Éste es el
        punto crucial del programa.*/
        if(exists(dragID))
            dragID.x = mouse.x - relativeX;
            dragID.y = mouse.y - relativeY;
        end
    end
end

```

```

                                end
                                if(key(_esc))exit();end
                                frame;
                                end //While(mouse.left)
                                frame;
                                end //Loop
end

```

La línea `map_clear(0,0,rgb(10,10,200))`; lo que hace simplemente es pintar el gráfico de fondo de pantalla de un color azul eléctrico. Si hubiéramos puesto otro gráfico en vez del (0,0), habríamos visto que ese gráfico era el que se pintaba entero de azul.

Después de esta línea vemos que aparece un `write_var` donde se imprime el valor de la variable “`dragID`”, la cual, como de momento no la hemos usado para nada, valdrá 0.

La línea `mouse.graph = new_map(30,30,16)`; lo que hace es asignar a el campo `graph` de la estructura `mouse` ( es decir, establecer el icono del puntero del ratón) a una imagen de 30x30 píxeles, todavía transparente.

Por lo tanto, el siguiente paso es hacer que esta nueva imagen que se corresponderá al icono del ratón deje de ser transparente y pase a ser, por ejemplo, una flecha blanca. Es decir, ahora se trataría de “dibujar” por código la flecha. Y es lo que el programa hace a continuación, mediante la función `map_put_pixel` dentro de los bucles FOR.

No obstante, antes de llegar allí nos encontramos con una línea que dice `set_center(0,mouse.graph,0,0)`; La línea `set_center(0,mouse.graph,0,0)`; lo que hace es establecer el centro del gráfico perteneciente al puntero del ratón –el cual como lo hemos creado con `new_map`, pertenece a la librería del sistema- en su coordenada (0,0), es decir, su esquina superior izquierda. Esto simplemente se hace para lograr tener un origen “de coordenadas” bueno para empezar los bucles FOR siguientes de forma que, tal como están diseñados, funcionen bien. La función `set_center` la comentaremos más adelante.

Y llegamos a los bucles FOR que contienen la función `map_put_pixel`. Esta función lo que nos hará, tal como está puesta en el código, es dibujarnos la imagen que se visualizará en la imagen – transparente en principio-, generada por `new_map`.

Fíjate dónde está `map_put_pixel`. Dentro de dos bucles FOR y un IF. No nos pararemos a ver qué es lo que hacen exactamente todos estos bloques porque nos apartaríamos bastante del objetivo principal: lo básico que tienes que saber es que, tal como están implementados, estos bloques logran pintar píxeles blancos de tal manera que hagan una figura de flecha apuntando hacia arriba a la izquierda (donde está el “punto caliente”). Es decir, lo que hacen estos FOR e IF es realizar cálculos más o menos complejos e ingeniosos para determinar si cada uno de los píxeles que forman parte del gráfico 30x30 que representará al puntero del ratón será blanco o será transparente. Fíjate que los dos FOR no son más que dos bucles anidados para ir coordenada por coordenada, por todas las Y de cada una de las X, y en cada coordenada se comprueba la condición que determinará qué pintar. La gracia está en que al final resulta el dibujo de la flecha. La condición del If es bastante ingeniosa: puedes cambiar algún valor de éste, a ver qué ocurre.

Ya tenemos el puntero del ratón dibujado, el cual se moverá allí donde lo mandemos. Pero todavía no hace nada más. De momento, fíjate que la línea siguiente lo que hace es disminuir a la mitad el tamaño visible del puntero del ratón con su propiedad `size`, para que sea un tamaño más normal:15x15.

Y ahora llegamos a un punto crucial: la creación del proceso “`puntero()`”. Este proceso va a representar el “punto caliente” del ratón: es decir, la zona del puntero del ratón que será la que detecte colisiones con las diferentes cartas. Esta zona haremos que sea pequeña, y situada en la punta de la flecha,

para que las colisiones solamente se detecten en ese punto y no en todo el gráfico de la flecha, cosa que quedaría feo.

Posiblemente te estés preguntando si es necesario crear un proceso como este. Tenemos una estructura *mouse* donde hemos definido el cursor, el tamaño, la posición dentro de la ventana, etc. ¿Por qué no podemos utilizar esta estructura para detectar colisiones? Pues precisamente, porque *mouse* no es ningún proceso, aunque se parezca. Es decir, en anteriores capítulos hemos visto ejemplos que podríamos aplicar aquí, del estilo:

```
Process letter()
Private
    Int id1;
End
    ...
    If(id1=collision(type mouse))
        ...
    end
...
end
```

donde, efectivamente, el ratón detecta colisiones. Pero esto no es lo que nos interesa ahora. En el código anterior, “id1” siempre valdrá el identificador del proceso que esté entre paréntesis tras el *type*. Y este “proceso” es el ratón, así que si el puntero –o más concretamente, su punto central- choca contra el proceso “letter()” determinado cualquiera, lo único que obtendremos es que “id1” pasa a valer siempre 1, porque sólo hay un puntero. Nosotros lo que queremos es al revés: obtener el identificador concreto de la carta con la que choca el puntero. ¿Para qué? Para arrastrar sólo y exclusivamente esa carta y no las demás cuando haya Drag&Drop. Es decir, nosotros necesitaríamos algo parecido a esto:

```
Process mouse()
Private
    Int id1;
End
    ...
    If(id1=collision(type letter))
        ...
    end
...
end
```

Pero esto no puede ser porque el proceso *mouse* NO EXISTE: es una estructura. Por lo tanto, nos lo tenemos que inventar. Es decir, tendremos que crear un proceso, que asuma el papel del puntero.

Este nuevo proceso tendrá asociado un gráfico, tal como he dicho, que será un cuadrado de 1x1 píxel situado en la punta de la flecha, que será precisamente la única zona que podrá detectar las colisiones, así que matamos dos pájaros de un tiro: hacemos que el puntero del ratón detecte colisiones y además que no las detecte en todo su gráfico sino sólo en su punta.

Este proceso “puntero()” es el que realmente tiene todo el peso del programa. Lo comentaremos al final, una vez acabemos con el programa principal.

El FOR que viene a continuación, en el programa principal, es sencillo de entender: simplemente realiza 9 iteraciones, donde en cada una de ellas crea un gráfico 25x25 en memoria de color oscuro, que representará una ficha. Seguidamente, con *write\_in\_map* se logra convertir -en cada iteración- cada una de las letras de la tabla en un gráfico más. Luego aparece la función *map\_put* para fusionar el

dibujo de la carta con su letra correspondiente en un sólo dibujo.

Finalmente, la última línea interna de este FOR es una llamada –repetida 9 veces- al proceso “letter()”, por lo que aparecerán 9 fichas con su letra correspondiente situadas una al lado de la otra, (ya que su variable X local cambia en cada llamada).

Fijate que el proceso “letter()” no tiene nada, y ya nuestro proceso principal tampoco. Ahora falta ver el código de nuestro proceso “puntero()”, que es que realmente va a efectuar el Drag&Drop.

El código de este proceso empieza creando el gráfico del proceso, un cuadrado 1x1 blanco, para meterse en un bucle infinito. En este bucle lo primero que se hace es obligar a que este cuadrado pequeño viaje siempre con el ratón, de manera que pase desapercibido y logre realizar su cometido de detector de colisiones. Y eso lo hará siempre hasta que se aprete el botón izquierdo del ratón. En ese momento, es cuando se detectará o no si hay colisión. Esta línea almacenará en dragID un 0 si no hay colisión (se ha pulsado el ratón en una zona sin fichas) o bien, si la hubiera, el identificador concreto de la ficha con la que se está colisionando. Seguidamente comprobamos si efectivamente dragID tiene el valor de algún identificador de un proceso existente (con la hasta ahora inédita función *exists*: también habríamos podido poner como hacíamos hasta ahora la línea del *collision* dentro del if y no usar, pues, esta función).

Si no hubiera colisión, realmente el código no hace nada: el proceso continúa siguiendo al puntero y ya está. Pero si hubiera colisión, lo primero que se hace es fijar unas cantidades (“relativeX” y “relativeY”) que representarán la distancia entre el puntero del ratón y el centro de la ficha colisionada, en el momento de apretar el botón izquierdo. Este valor constante simplemente sirve para que cuando arrastremos alguna ficha, el centro de su gráfico no se acople con la punta de la flecha –es el comportamiento normal, pero queda feo- sino que la posición relativa de su centro respecto la punta de la flecha se mantenga igual al principio del Drag&Drop correspondiente.

A continuación entramos en el while que ejecuta de hecho el Drag&Drop: mientras se mantenga pulsado el botón izquierdo, las coordenadas del centro de la ficha colisionada –identificada por dragID- serán las mismas que las de la punta de la flecha, pero corregidas por la distancia relativa que acabamos de comentar. Y ya está. Basta procurar escribir la orden Frame; en aquellos sitios que puedan hacer colgar el programa (bucles), y en principio tendría que funcionar.

### **Trabajar con sonido:**

Antes de poder usar los maravillosos efectos especiales de sonido que hemos preparado (explosiones, disparos, choques, etc) tenemos que decirle al programa que los cargue en memoria. ¿Cómo se hace eso? Con la instrucción *load\_wav*.

*Load\_wav* se encarga, como su nombre indica, de cargar en memoria un fichero de audio con formato wav (que no de reproducir nada: sólo carga). Recordarás que en la introducción de este curso comentamos que este formato contiene una grabación digital de audio sin comprimir, y que por tanto ocupaba mucho espacio en disco –y en memoria, claro-. Por lo tanto, los archivos wav son recomendables sólo cuando se quieren utilizar sonidos realistas pero relativamente cortos de duración, como puede ser una explosión, un choque... pero no para poner una canción, por ejemplo.

De hecho, para poner música en un juego –que no sonidos concretos- existe una manera distinta de trabajar, de la cual hablaremos dentro de nada.

También hay otro uso de los ficheros wav, y es el de poner un sonido que se repita constantemente a lo largo de un período de tiempo, ya que la carga del wav en memoria sólo se realiza la primera vez, y a partir de ésta se puede ir repitiendo la emisión de ese sonido sin problemas. Ahora lo veremos.

Una vez cargado el sonido, entonces se procederá a reproducirlo, con la función *play\_wav*. Podremos parar la reproducción con *stop\_wav*, pausarla con *pause\_wav* y reemprenderla con *resume\_wav* y

varias cosas más como cambiar el volumen o el balance del sonido con las funciones que veremos a continuación.

### LOAD\_WAV("fichero")

Esta función carga un fichero en formato WAV de disco y lo almacena en memoria para reproducirlo en el futuro mediante la función PLAY\_WAV . El valor que devuelve la función es un número entero que representa el recurso en memoria, o -1 si no fue posible cargar el fichero o no estaba en un formato correcto.

Si no se desea utilizar por más tiempo, puede descargarse de memoria el efecto de sonido empleando la función UNLOAD\_WAV.

PARAMETROS: STRING FICHERO : Nombre del fichero Wav

VALOR RETORNADO: INT: Identificador del sonido

### PLAY\_WAV(WAV, REPETICIONES)

Reproduce un efecto de sonido recuperado previamente de disco mediante la función LOAD\_WAV.

Es posible reproducir varias veces un mismo efecto de forma simultánea. Esta función devuelve un identificador del *canal* de sonido empleado para reproducir el efecto de sonido. Existen 16 canales de sonido disponibles para efectos, y esta función puede elegir cualquiera de ellos que esté libre. En caso de reproducir una gran cantidad de efectos de sonido simultáneamente, es posible que en un momento dado hayan ya 16 sonidos activos, en cuyo caso el nuevo sonido no será reproducido y esta función devolverá -1. El identificador de canal devuelto por esta función puede usarse con IS\_PLAYING\_WAV, SET\_WAV\_VOLUMNE , y demás funciones de sonido.

El efecto de sonido puede reproducirse una sola vez, si se pasa como parámetro de repeticiones 0 ó 1, el número concreto de repeticiones indicado, o bien indefinidamente si este parámetro es -1. Hay que tener en cuenta que si el sonido se reproduce indefinidamente, el canal quedará ocupado hasta que el sonido se detenga con STOP\_WAV .

PARAMETROS:

INT WAV : Identificador del sonido devuelto por LOAD\_WAV

INT REPETICIONES : Número de repeticiones (0 ó 1=1 vez; -1= infinitas veces)

VALOR RETORNADO: INT: Número de canal

### STOP\_WAV(CANAL)

Detiene la reproducción de cualquier sonido que esté activo a través de un canal determinado. Para saber qué canal corresponde a un efecto de sonido concreto, debes guardar el valor devuelto por PLAY\_WAV

PARAMETROS:

INT CANAL : Número de canal devuelto por PLAY\_WAV

### PAUSE\_WAV(CANAL)

Detiene temporalmente la reproducción de cualquier sonido que esté activo a través de un canal

determinado, hasta que vuelva a reanudarse la reproducción empleando RESUME\_WAV . Para saber qué canal corresponde a un efecto de sonido concreto, debes guardar el valor devuelto por PLAY\_WAV .

PARAMETROS:

INT CANAL : Número de canal devuelto por PLAY\_WAV

**RESUME\_WAV(CANAL)**

Reanuda la reproducción del sonido que estaba activo a través de un canal determinado en el momento que se detuvo la reproducción empleando PAUSE\_WAV. Para saber qué canal corresponde a un efecto de sonido concreto, debes guardar el valor devuelto por PLAY\_WAV

PARAMETROS:

INT CANAL : Número de canal devuelto por PLAY\_WAV

**IS\_PLAYING\_WAV(CANAL)**

Esta función comprueba si cualquier efecto de sonido reproducido anteriormente mediante PLAY\_WAV \_ todavía está sonando, devolviendo en este caso 1. En caso contrario esta función devolverá 0.

Hay que tener en cuenta que el canal puede ser reutilizado más tarde por otros efectos de sonido, por lo que es conveniente no seguir haciendo la comprobación una vez esta función ha devuelto 0.

PARAMETROS:

INT CANAL : Número de canal devuelto por PLAY\_WAV

**SET\_CHANNEL\_VOLUME(CANAL,VOLUMEN)**

Especifica el volumen general del canal de sonido, entre 0 (silencio) y 128 (volumen máximo).

Si el canal especificado es -1, afecta al volumen de todos los canales.

A parte de este nivel de volumen por cada canal, el volumen de cada efecto de sonido que se reproduzca podrá ser regulado independientemente por SET\_WAV\_VOLUME. De tal manera, el volumen final de un efecto de sonido vendrá dado por estas dos funciones: SET\_CHANNEL\_VOLUME establece el volumen base del canal y SET\_WAV\_VOLUME, a partir de ese volumen base, lo modifica para un efecto de sonido en concreto.

PARÁMETROS :

INT CANAL : Número de canal devuelto por PLAY\_WAV

INT VOLUMEN : Volumen deseado (0-128)

**SET\_WAV\_VOLUME(CANAL,VOLUMEN)**

Cambia el volumen de un efecto de sonido cargado anteriormente mediante LOAD\_WAV , entre 0 (silencio) y 128 (volumen original del efecto).

El volumen final de reproducción del efecto será este modificado por el valor de volumen de reproducción asignado al canal por el que esté con SET\_CHANNEL\_VOLUME..

### PARÁMETROS :

INT WAV : Identificador del efecto devuelto por LOAD\_WAV  
INT VOLUMEN : Volumen deseado (0-128)

Un ejemplo trivial del uso de esta función es poder crear un mando con los cursores del teclado que controle en tiempo real el volumen de un sonido que se está reproduciendo en este momento:

```
program Test_SET_WAV_VOLUME;
global
  int wav;
  int vol=64;
end
begin
  set_mode(320,240,16);
  write(0,10,90,3,"-> = UP Volumen...");
  write(0,10,100,3,"<- = DOWN Volumen...");
  write(0,10,110,3,"Volumen Actual... ");
  write_var(0,124,110,3,vol);
  wav=load_wav("wav.wav");
  play_wav(wav,-1);
  repeat
    if(key(_left) && vol>0) vol=vol-8; end
    if(key(_right) && vol<128) vol=vol+8; end
    set_wav_volume(wav,vol);
  frame;
  until(key(_esc))
end
```

### **SET\_DISTANCE(CANAL, DISTANCIA)**

Esta función permite alterar el sonido de un canal determinado simulando un alejamiento del mismo, a partir de una distancia al jugador. La distancia 0 deja el sonido como estaba al principio, mientras una distancia de 255 hace el sonido inaudible.

Esta función filtra el sonido de un canal de sin hacer uso de las características avanzadas de algunas tarjetas de audio. Si se desea una emulación algo más sofisticada, es mejor usar SET\_POSITION .

#### PARÁMETROS :

INT CANAL : Número de canal devuelto por PLAY\_WAV  
INT VOLUMEN : Distancia del jugador (0-255)

### **SET\_PANNING(CANAL,IZQUIERDO,DERECHO)**

Cambia el balance de un canal. Es decir, esta función permite cambiar el posicionamiento izquierdo/derecho en stereo de un sonido, simplemente ajustando el volumen de cada uno de los dos subcanales. Si se desea dejar el sonido tal cual estaba en un principio, basta con hacer un SET\_PANNING(255,255).

#### PARÁMETROS :

INT CANAL : Número de canal devuelto por PLAY\_WAV  
INT IZQUIERDO : Volumen izquierdo (0-255)  
INT DERECHO: Volumen derecho (0-255)

### SET\_POSITION(CANAL,ANGULO,DISTANCIA)

Esta función permite alterar el sonido de un canal determinado simulando un posicionamiento en 3D del mismo, a partir de un ángulo de posición y una distancia al jugador. Los parámetros (0,0) dejan el sonido como estaba al principio, mientras una distancia de 255 hace el sonido inaudible.

Esta función filtra el sonido de un canal de una forma algo lenta, sin hacer uso de las características avanzadas de algunas tarjetas de audio. Si sólo se desea una emulación más simple, es más rápido llamar a SET\_DISTANCE.

#### PARÁMETROS :

INT CANAL : Número de canal devuelto por PLAY\_WAV

INT ANGULO : Ángulo respecto al jugador, en grado (0-360)

INT DISTANCIA: Distancia del jugador (0-255)

### UNLOAD\_WAV(WAV)

Elimina un efecto de sonido de memoria. Es importante que el efecto no se esté reproduciendo en el momento de descargarlo de memoria, por lo que debes asegurarte de ello mediante la función IS\_PLAYING\_WAV.

Normalmente sólo es preciso descargar de memoria efectos de sonido continuos, que ocupan más espacio, como el motor de un coche o una lluvia de fondo.

#### PARÁMETROS :

INT WAV : Identificador del efecto devuelto por LOAD\_WAV

Un ejemplo muy simple de estas funciones de sonido lo podemos ver aquí. En este código se crea un proceso, el cual mientras esté vivo mostrará por pantalla un rectángulo de color amarillo. Y este proceso estará vivo el tiempo que dure en sonar un archivo wav (llamado "a.wav"). El usuario podrá pausar y reemprender la ejecución del sonido con la tecla "s" y "r" respectivamente.

Notar que todas las funciones excepto *play\_wav*, *set\_wav\_volume* y *unload\_wav* tienen como parámetro el identificador de canal, y no el identificador del archivo wav.

```
program aa;
global
  int idwav;
end
begin
set_mode(320,240,16);
//La carga de los sonidos se puede hacer al principio del programa, al igual que se hace con los fpgs.
idwav=load_wav("a.wav");
miproceso();
while(!key(_esc))
  frame;
end
end

process miproceso()
```

```

private
  int idcanal;
end
begin
  graph=new_map(200,100,16);
  map_clear(0,graph,rgb(255,255,0));
  x=200;
  y=150;
  idcanal=play_wav(idwav,0);
  /*Las cinco líneas siguientes lo único que hacen es alterar el sonido emitido por play_wav de manera que
  suene más alejado, con menor volumen y balanceado hacia la izquierda. Serían opcionales, en todo caso*/
  set_wav_volume(idwav,100);
  set_channel_volume(idcanal,100);
  set_panning(idcanal,100,0);
  set_distance(idcanal,100);
  set_position(idcanal,100,100);
  while(is_playing_wav(idcanal))
    if(key(_s)) pause_wav(idcanal); end
    if(key(_r)) resume_wav(idcanal); end
    frame;
  end
  unload_wav(idwav);
end

```

Al volumen del canal en vez de un valor concreto se le podría poner una variable, de manera que si todos los sonidos tienen esa variable se podrá controlar el volumen general del juego alterando el valor de esa variable. Recuerda que también existe la función *set\_wav\_volume* -no confundir con la anterior- que establece el volumen de ese sonido en sí de forma permanente, no sólo cuando se reproduce. Hay que tener cuidado con la suma de los efectos de las funciones *set\_channel\_volume* y *set\_wav\_volume*: a la hora de mezclar primero afecta el valor del sample y luego el del canal.

### **Trabajar con música:**

Los wavs están muy bien para efectos especiales cortos, como disparos, puertas, saltos y otro tipo de sonidos que no se reproducen durante mucho tiempo. Pero para temas de fondo o bandas sonoras no son adecuados, ya que ocupan mucho espacio. Para eso están las funciones *load\_song*, *play\_song*, *stop\_song*, *pause\_song*, *resume\_song* o *unload\_song* que permiten usar o bien archivos MIDI o bien archivos OGG (y también archivos WAV otra vez).

Recordarás de la introducción de este curso que los archivos MIDI (.mid) ocupan muy poco espacio porque sólo contienen una representación de las notas que componen la música, y es la tarjeta de sonido la que se encarga (con menor o mayor fidelidad según su calidad) de reproducir los instrumentos. Por lo tanto, es el formato ideal si quieres crear juegos que ocupen lo mínimo posible. Los archivos OGG en cambio, contiene una grabación digital de sonido comprimida al máximo, y están recomendados por su excelente calidad; si quieres usar música grabada en tu juego, es el formato adecuado.

Quisiera comentar la existencia de un formato de archivo de sonido del cual no hemos hablado hasta ahora, y que tampoco utilizaremos en este curso, pero que creo que como mínimo has de conocer, y que las funciones que estamos tratando son capaces también de reproducir. Se trata de los módulos. Son archivos con extensión .xm, o .it, o .s3m, o .mod, los cuales contienen una representación de las notas que componen la música (como los .mid) y, además, una digitalización de cada uno de los instrumentos usados (sonido digitalizado), así que es un formato intermedio a los midi y a los wav, tanto en calidad como en tamaño: suenan más realista que los .mid, pero ocupan más.

### LOAD\_SONG("fichero")

Recupera de disco un fichero de música, para reproducir posteriormente con PLAY\_SONG . Actualmente se soportan los siguientes formatos: OGG Vorbis (.ogg) , MIDI (.mid), módulos (.xm,.it, .s3m, .mod) y WAV (.wav)

Esta función devuelve un número entero que identifica la música, o -1 si el fichero no está en un formato reconocido. Puedes usar el número devuelto para reproducir la música mediante la función PLAY\_SONG .

PARAMETROS: STRING FICHERO : Nombre del fichero

VALOR RETORNADO: INT : Identificador de la música

### PLAY\_SONG (MUSICA, REPETICIONES)

Inicia la reproducción de un fichero de música recuperado con la función LOAD\_SONG . Dependiendo del tipo de fichero, sólo es posible reproducir un fichero de música a la vez, por lo que cualquier música que se estuviese reproduciendo debe detenerse primero con STOP\_SONG,, ya que a diferencia de *play\_wav*, *play\_song* no trabaja con canales, por lo que sólo se puede reproducir una música a la vez

El segundo parámetro indica el número de veces que debe repetirse la reproducción. Puede usarse 0 para no repetirla, o -1 para repetirla continuamente hasta que se detenga con STOP\_SONG

PARAMETROS:

INT MUSICA : Identificador de la música

INT REPETICIONES : Número de repeticiones

### STOP\_SONG()

Detiene la reproducción de cualquier fichero de música en curso inmediatamente..

Para un mejor efecto, prueba a usar la función FADE\_MUSIC\_OFF

### UNLOAD\_SONG (MUSICA)

Libera la memoria ocupada por una música cargada en memoria con LOAD\_SONG . Antes es necesario, si se está reproduciendo, detener la reproducción de la música con STOP\_SONG

PARAMETROS:

INT MUSICA : Identificador de la música

### FADE\_MUSIC\_IN (MUSICA, REPETICIONES, MILISEGUNDOS)

Esta función equivale a PLAY\_SONG , excepto que la música no empieza a reproducirse a máximo volumen sino desde el silencio, y va subiendo de volumen gradualmente hasta que, pasado el tiempo

indicado en el parámetro *milisegundos* , el volumen es completo.

El segundo parámetro indica el número de veces que debe repetirse la reproducción. Puede usarse 0 para no repetirla, o -1 para repetirla continuamente hasta que se detenga con `STOP_SONG`

Esta función es especialmente útil para reproducir efectos de sonido continuos de este tipo (como por ejemplo, sonidos ambientales) en lugar de música.

PARAMETROS:

INT MUSICA : Identificador de la música

INT REPETICIONES : Número de repeticiones

INT MILISEGUNDOS : Tiempo para llegar al volumen completo en milisegundos

### **FADE\_MUSIC\_OFF (MILISEGUNDOS)**

Esta función detiene la música que esté ejecutándose actualmente (que haya sido iniciada mediante la función `PLAY_SONG` ) bajando gradualmente el volumen hasta que se apague. El parámetro *milisegundos* indica el tiempo aproximado en el que se desea que la música haya terminado.

Obviamente justo después de llamar a esta función, la música no ha acabado inmediatamente. Para saber cuándo es seguro usar `UNLOAD_SONG` o reproducir otra música, es preciso utilizar `IS_PLAYING_SONG` para comprobar si la música ya ha terminado de apagarse.

PARAMETROS:

INT MILISEGUNDOS : Tiempo para llegar al final de la reproducción en milisegundos.

### **IS\_PLAYING\_SONG()**

Esta función comprueba si cualquier música reproducida anteriormente mediante `PLAY_SONG` todavía está sonando, en cuyo caso devolverá 1. En caso contrario (la música ya ha terminado y no se reproduce en modo de repetición, o bien se ha detenido con funciones como `STOP_SONG` o `FADE_MUSIC_OFF` ), esta función devolverá 0.

VALOR RETORNADO: INT : Indicador de si la canción se está reproduciendo (1) o no (0)

### **PAUSE\_SONG()**

Detiene temporalmente la reproducción de cualquier fichero de música en curso, hasta que se continúe más tarde mediante la función `RESUME_SONG`.

Esta función detiene la reproducción de música inmediatamente. Para un mejor efecto, prueba a usar la función `FADE_MUSIC_OFF`

### **RESUME\_SONG()**

Continúa la reproducción de música desde el punto donde se detuvo anteriormente mediante la función `PAUSE_SONG`

## SET\_SONG\_VOLUME(VOLUMEN)

Esta función establece el nivel general de volumen para la música entre un nivel de 0 (silencio total) y 128 (volumen original del fichero).

PARÁMETROS : INT VOLUMEN : Volumen deseado (0-128)

Un ejemplo de estas funciones de música podría ser éste.

En este código si se pulsa la tecla “a” se crea un proceso, el cual mientras esté vivo mostrará por pantalla un rectángulo de color amarillo. Y este proceso estará vivo el tiempo que dure en sonar un archivo wav (llamado “a.wav”). El usuario podrá pausar y reemprender la ejecución del sonido con la tecla “s” y “r” respectivamente.

En cambio, si se pulsa la tecla “b”, se creará otro proceso -”proceso2”, el cual mientras esté vivo mostrará por pantalla un rectángulo de color rosa. Y este proceso estará vivo el tiempo que dura en sonar -mediante un fade in- el mismo archivo wav. El usuario podrá realizar un fade out si pulsa la tecla “f”.

```
program aa;
global
  int idsong;
end
begin
  set_mode(320,240,16);
  idsong=load_song("a.wav");
  while(!key(_esc))
    if(key(_a))miproceso(); end
    if(key(_b))miproceso2();end
    frame;
  end
  unload_song(idsong);
end

process miproceso()
begin
  graph=new_map(200,100,16);
  map_clear(0,graph,rgb(255,255,0));
  x=200;
  y=150;
  play_song(idsong,0);
  set_song_volume(100);
  while(is_playing_song())
    if(key(_s)) pause_song(); end
    if(key(_r)) resume_song(); end
    frame;
  end
end

process miproceso2()
begin
  graph=new_map(200,100,16);
  map_clear(0,graph,rgb(255,0,255));
  x=300;
  y=40;
  fade_music_in(idsong,0,3);
```

```

while(is_playing_song())
  if(key(_f)) fade_music_off(3); end
  frame;
end
end

```

¿Has probado de ejecutar los dos procesos a la vez -apretando “a” y “b”? Verás, como era de esperar, que los dos procesos se ponen en marcha mostrándose los dos rectángulos, pero en cambio, la música no suena doblada, sino que cada vez que se ejecuta uno de los procesos, se para la música que estaba sonando en aquel momento y empieza la música del nuevo proceso, así que sólo se escucha esa música una sola vez cada vez, sin ningún tipo de superposición. Esto es así porque, a diferencia de las funciones de la familia “load\_wav”, “play\_wav”, etc, las funciones de la familia “load\_song”, “play\_song”, etc no funcionan con canales, por lo que sólo se puede escuchar a la vez un sonido cargado con load\_song. Si quieres, haz la prueba de ejecutar el mismo ejemplo anterior pero utilizando las funciones de la familia “load\_wav”: si utilizar los canales de forma adecuada (un canal diferente para cada proceso), verás que sí que sonarán las diferentes músicas a la vez cuando los procesos estén en marcha.

### **Trabajar con ángulos y distancias:**

Una de los conceptos más elementales que hay que dominar para lograr realizar videojuegos que sean mínimamente atractivos es el dominio de la medida de los ángulos y de las distancias existentes entre los diferentes procesos que en un momento determinado se visualicen en pantalla. Echando mano de las funciones que Fénix dispone en este campo, lograremos efectos tan básicos como conseguir que un proceso esté encarado siempre hacia otro, o que un proceso persiga constantemente a otro proceso, etc,etc

#### **FGET\_ANGLE( X1, Y1, X2, Y2 )**

Esta función devuelve, en milésimas de grado, el ángulo formado por la línea entre los dos puntos especificados, y el eje X (horizontal) de coordenadas.

Con esta función puede, fácilmente, orientar el ángulo de un proceso para que éste siempre esté "mirando" hacia un lugar concreto (como puede ser otro proceso). La aplicación más sencilla puede ser un proceso que "persiga" a otro simplemente utilizando FGET\_ANGLE seguida de ADVANCE.

#### **PARAMETROS:**

INT X1: Coordenada X del primer punto.  
 INT Y1: Coordenada Y del primer punto  
 INT X2: Coordenada X del segundo punto  
 INT Y2: Coordenada Y del segundo punto.

#### **GET\_ANGLE(ID)**

Esta función devuelve el ángulo formado por la línea que parte del centro del proceso actual y pasa por el centro del proceso cuyo identificador se especifica como parámetro.

El uso habitual de esta función consiste en hacer que un proceso "apunte" en dirección a otro: es el caso típico de monstruos y otros objetos que buscan siempre atrapar al jugador.

Para que esta función devuelva un resultado útil, asegúrate de que no has asignado valores diferentes de la

variable RESOLUTION y de que las coordenadas de ambos representan la misma zona (por ejemplo, que no estén cada uno en un área de scroll diferente).

PARAMETROS:

INT ID: Identificador de un proceso

**FGET\_DIST( X1, Y1, X2, Y2 )**

Esta función devuelve la distancia entre dos puntos, en las mismas unidades que vengan dados éstos. Es una función relativamente lenta, que devuelve un resultado exacto empleando una raíz cuadrada.

PARAMETROS:

INT X1: Coordenada X del primer punto.  
INT Y1: Coordenada Y del primer punto  
INT X2: Coordenada X del segundo punto  
INT Y2: Coordenada Y del segundo punto.

**GET\_DIST(ID)**

Esta función devuelve la distancia en línea recta entre el centro del proceso actual y el centro del proceso cuyo identificador se especifica como parámetro.

PARAMETROS:

INT ID: Identificador de un proceso

**GET\_DISTX(ANGULO, DISTANCIA)**

Esta función devuelve, en la misma unidad que se use para la distancia, el ancho del rectángulo formado por la línea que recorre esa distancia. Usando esta función junto con GET\_DISTY se puede saber qué cantidades sumar a las coordenadas X e Y de un objeto para desplazarlo en la dirección que se desee.

PARAMETROS:

INT ANGULO: Ángulo en milésimas de grado (90000= 90°)  
INT DISTANCIA : Magnitud de distancia

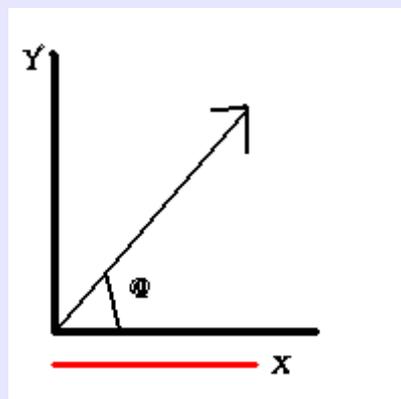


Gráfico que ilustra, dada una distancia y un ángulo, el valor "x" que devuelve la función get\_distx (lo marcado en rojo)

## GET\_DISTY(ANGULO, DISTANCIA)

Esta función devuelve, en la misma unidad que uses para la distancia, la altura del rectángulo formado por la línea que recorre esa distancia. Usando esta función junto con GET\_DISTX se puede saber qué cantidades sumar a las coordenadas X e Y de un objeto para desplazarlo en la dirección que se desee.

### PARAMETROS:

INT ANGULO: Ángulo en milésimas de grado (90000= 90°)

INT DISTANCIA : Magnitud de distancia

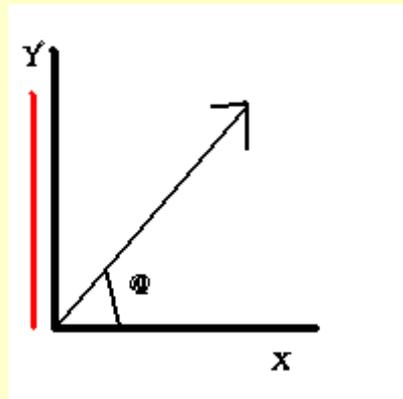


Gráfico que ilustra, dada una distancia y un ángulo, el valor “y” que devuelve la función `get_disty` (lo marcado en rojo)

A continuación vamos a ver unos cuantos ejemplos básicos que ilustran el funcionamiento de las funciones anteriores. Por ejemplo, el siguiente código va mostrando a cada segundo una línea diferente cuyos dos extremos son puntos de coordenadas aleatorias. Para cada línea que aparezca el programa nos dirá el ángulo que forma ésta respecto el eje X (con `fget_angle`) y su longitud -es decir, la distancia entre sus extremos- (con `fget_dist`).

```
program hola;
global
  int xorigen,yorigen,xfinal,yfinal;
  int distancia,angulo;
end
begin
  set_mode(640,480,16);
  set_fps(1,0);
  drawing_z(0);
  loop
    delete_text(all_text);
    xorigen=rand(10,300);
    yorigen=rand(10,300);
    xfinal=rand(10,300);
    yfinal=rand(10,300);

    delete_draw(0);
    draw_line(xorigen,yorigen,xfinal,yfinal);
  //Las cajitas hacen un poco más visibles los dos extremos de la línea
```

```

draw_box(xorigen-3,yorigen-3,xorigen+3,yorigen+3);
draw_box(xfinal-3,yfinal-3,xfinal+3,yfinal+3);

distancia=fget_dist(xorigen,yorigen,xfinal,yfinal);
write(0,450,100,4,"Distancia entre los puntos: " + distancia);
angulo=fget_angle(xorigen,yorigen,xfinal,yfinal);
write(0,450,150,4,"Angulo formado por la línea y el eje X: "+ (float)angulo/1000);
frame;
end
end

```

Para mostrar la utilidad de las funciones `get_dist` y `get_angle` es interesante estudiar el ejemplo siguiente. En él aparecen dos procesos: un cuadrado verde inmóvil y un cuadrado rojo que podremos mover con los cursores. En todo momento aparecerán por pantalla dos datos, que serán los devueltos por `get_dist` y `get_angle`, respectivamente. Es decir, la distancia a la que se encuentra en cada momento el cuadrado rojo que dominamos respecto el cuadrado verde, y el ángulo que forma la línea que une ambos cuadrados respecto la horizontal -ambas líneas también aparecen pintadas de blanco para facilitar la interpretación de dicho número-

```

program hola;
global
  int idverde;
end
begin
  set_mode(640,480,16);
  //El identificador idverde lo necesitaremos para las funciones get_dist y get_angle, ejecutadas desde rojo()
  idverde=verde();
  rojo();
  while(!key(_esc))
    frame;
  end
  let_me_alone();
end

process rojo()
private
  int angulo,distancia;
end
begin
  graph=new_map(30,30,16);
  map_clear(0,graph,rgb(255,0,0));
  x=400;
  y=100;
  drawing_z(0);
  loop
    if(key(_left)) x=x-10; end
    if(key(_right)) x=x+10; end
    if(key(_up)) y=y-10; end
    if(key(_down)) y=y+10; end

  /*Dibujo la horizontal y la línea que une los dos procesos, para mostrar visualmente el significado del
  valor devuelto por get_angle. Fijarse como aquí utilizo ya el identificador idverde para acceder a las
  coordenadas de posición de un proceso diferente al actual.*/
  delete_draw(0);

```

```

draw_line(x,y,x+100,y);
draw_line(x,y,idverde.x,idverde.y);

delete_text(0);
distancia=get_dist(idverde);
write(0,200,200,4,"La distancia al cuadrado verde es de " + distancia);
angulo=get_angle(idverde);
/*Es curioso comprobar que los valores que devuelve get_angle van de 0° a 270°, y a partir de allí de -90°
a 0° otra vez*/
write(0,200,250,4,"El ángulo que formo respecto el cuadrado verde es " + (float)angulo/1000);
frame;
end
end

process verde()
begin
graph=new_map(30,30,16);
map_clear(0,graph,rgb(0,255,0));
x=500;
y=100;
loop
frame;
end
end

```

Si queremos ver la utilidad de las funciones `get_disty` e `get_distx`, podemos aprovechar el ejemplo anterior y simplemente añadir, justo antes de la orden `frame` del `loop/end` del proceso `rojo()` las siguientes líneas:

```

distx=get_distx(angulo,distancia);
disty=get_disty(angulo,distancia);
write(0,200,300,4,"La distancia horizontal al cuadrado verde es de " + distx);
write(0,200,350,4,"La distancia vertical al cuadrado verde es de " + disty);

```

teniendo la preocupación además de declarar como variables privadas del proceso `rojo()` las variables “`distx`” y “`disty`”.

Si pruebas ahora el código, verás que obtienes dos datos más: los dos datos devueltos por las funciones `get_distx` y `get_disty` respectivamente, a partir de un ángulo y una distancia dados, los cuales hemos puesto que sean para redondear el ejemplo precisamente el ángulo y la distancia existentes entre los dos procesos del ejemplo. Fíjate que, evidentemente, si movemos el cuadrado rojo en horizontal, el valor devuelto por `get_disty` no se altera, y al revés: si movemos el cuadrado rojo en vertical, el valor devuelto por `get_distx` tampoco cambia, como ha de ser.

Otro ejemplo ilustrativo del uso de `get_angle`, `get_disx` y `get_disty` es el siguiente. En él tenemos un proceso “cuadamar” visualizado por un cuadrado amarillo el cual podremos mover con el cursor y que es perseguido incansablemente por otro proceso, “cuadroj” (visualizado por un cuadrado rojo). La idea es que cuando “cuadroj” se sale de la pantalla se vuelve a colocar en una nueva posición aleatoria dentro de la pantalla para continuar la persecución. Esta persecución, no obstante, es *sui generis*, porque “cuadroj” solamente tiene en cuenta la posición que tiene “cuadamar” en el momento de su creación -o su puesta de nuevo dentro de la pantalla-, por lo que si “cuadamar” se mueve, “cuadroj” no lo sabrá y seguirá imperturbable en la dirección que “cuadamar” tenía originalmente.

```

Program ejemplo_get_distx_get_angle;
Global
  int idcuadamar;
End
Begin
  set_mode(640,480,16);
  write(0,0,20,0,"TECLAS RIGHT,LEFT,UP y DOWN para mover cuadrado amarillo");
  idcuadamar=cuadamar();
  /*El cuarto parámetro de cuadroj hace que a la hora de crear dicho proceso, se obtenga el ángulo que en
  ese instante tiene cuadamar respecto el nuevo proceso cuadroj. Este ángulo se utilizará para dirigir
  cuadroj a la posición que ocupa cuadamar, pero como esta medición sólo se hace en este instante -cuando
  se crea cuadroj-, si posteriormente cuadamar se mueve, cuadroj no se enterará y continuará estando
  dirigido en la dirección que ocupaba cuadamar originalmente.*/
  cuadroj(x,y,get_angle(idcuadamar));
End

Process cuadroj(x,y, int angulo)
  Begin
    graph=new_map(30,30,16);map_clear(0,graph,rgb(255,0,0));
    Loop
      /*A partir del ángulo que hay entre cuadamar y cuadroj, se calcula las nuevas coordenadas de cuadroj*/
      x=x+get_distx(angulo,16);
      y=y+get_disty(angulo,16);
      /*Si cuadroj sale de los límites de la pantalla, se vuelve a colocar en unas coordenadas aleatorias otra vez
      dentro, y se vuelve a obtener la nueva orientación (el nuevo ángulo) de cuadroj respecto cuadamar (el
      cual también puede haber cambiado de posición mientras tanto) como si "cuadroj" hubiera sido creado
      de nuevo, para volver a empezar la persecución*/
      If(x>640 OR y>480 OR x<0 OR y<0)
        x=rand(0,600);y=rand(0,450);
        angulo=get_angle(idcuadamar);
      End
      If(key(_esc)) exit();End
      Frame;
    End
  End
End

Process cuadamar()
  Begin
    graph=new_map(30,30,16);map_clear(0,graph,rgb(255,255,0));
    x=160;y=100;
    Loop
      If(key(_right)) x=x+16;End
      If(key(_left)) x=x-16;End
      If(key(_up)) y=y-16;End
      If(key(_down)) y=y+16;End
      //No se puede salir de los extremos de la pantalla
      If(x>620) x=620;End
      If(x< 20) x=20;End
      If(y>460) y=460;End
      If(y< 20) y=20;End
      Frame;
    End
  End
End

```

Otro ejemplo, a lo mejor más sencillo, de estas mismas funciones es el siguiente código, donde se puede ver un cuadrado rojo que hace de “escudo vigilante” a un cuadrado amarillo.

```
Program ejemplo_get_distx;
Private
  int distancia=100; //Distancia del cuadrado rojo al amarillo
  int idcuadamar;
end
Begin
  set_mode(640,480,16);
//El programa principal representa el proceso del cuadrado rojo
  graph=new_map(30,30,16);map_clear(0,graph,rgb(255,0,0));
  idcuadamar=cuadamar();
  Loop
/*Cambiamos las coordenadas del cuadrado rojo (y su orientación). Si coges lápiz y papel verás que es
fácil deducir las fórmulas que se emplean para simular el movimiento circulatorio*/
    angle=angle+2000;
    x=idcuadamar.x+get_distx(angle,distancia);
    y=idcuadamar.y+get_disty(angle,distancia);
    If(key(_esc)) exit();End
  Frame;
End
End

Process cuadamar()
Begin
  x=250;y=240;
  graph=new_map(30,30,16);map_clear(0,graph,rgb(255,255,0));
  loop
//Aunque me mueva, el cuadrado rojo va a continuar rodeándome
    if(key(_left))x=x-10;end
    if(key(_right))x=x+10;end
    if(key(_up))y=y-10;end
    if(key(_down))y=y+10;end
  Frame;
end
End
```

### XADVANCE(ANGULO,NUMERO)

Esta función desplaza el proceso actual en la dirección especificada por el ángulo indicado como primer parámetro, el número de unidades indicado como segundo parámetro.

Hay que tener en cuenta que el desplazamiento se hace en unidades enteras, y por lo tanto tiene limitaciones de precisión. Un desplazamiento de 1 ó 2 unidades no puede reflejar las diferencias entre ángulos, por lo que no será efectivo. Si se desea desplazar un proceso en ángulos arbitrarios y sólo unos pocos pixels, será preciso utilizar la variable RESOLUTION para que las coordenadas del gráfico vengan expresadas en unidades más pequeñas que un pixel, como ocurre por defecto.

#### PARAMETROS:

INT ANGULO: Ángulo en milésimas de grado (90000= 90°)  
INT NUMERO : Número entero

Esta función se utiliza bastante para conseguir que un proceso se dirija hacia la posición que ocupa otro proceso en particular. Sabemos que Xadvance necesita un ángulo para dirigir el proceso actual; pues bien, en este caso, el ángulo que necesitamos sería el devuelto por la función `get_angle`, poniendo como parámetro de dicha función el identificador del proceso “perseguido”, ya que recordemos que esta función devuelve el ángulo que forma la línea que atraviesa ambos procesos respecto la horizontal: precisamente el ángulo que queremos. Un ejemplo de esta aplicación sería el siguiente. En él tenemos dos procesos: un cuadrado rojo que persigue a otro, un cuadrado verde inmóvil que cuando es alcanzado cambia de posición aleatoriamente para continuar siendo perseguido sin descanso por el cuadrado rojo.

```

program hola;
global
  int idverde;
end
begin
  set_mode(640,480,16);
  idverde=verde();
  rojo();
  loop
    frame;
  end
end

process rojo()
private
  int angulo;
end
begin
  graph=new_map(30,30,16);
  map_clear(0,graph,rgb(255,0,0));
  x=100;
  y=400;
  loop
    //Si yo, cuadrado rojo, colisiono con el cuadrado verde, hago que éste cambie de posición
    if(collision(idverde))
      idverde.x=rand(0,599);
      idverde.y=rand(0,399);
    end
    //Las dos líneas importantes que realizan la persecución
    angulo=fget_angle(x,y,idverde.x,idverde.y);
    xadvance(angulo,10);
    frame;
  end
end

process verde()
begin
  graph=new_map(30,30,16);
  map_clear(0,graph,rgb(0,255,0));
  x=500;
  y=100;
  loop
    frame;
  end
end

```

*end*

Fíjate, no obstante, que a diferencia de la explicación inicial, en el código no hemos utilizado la función `get_angle` para averiguar el ángulo que forman los dos procesos entre ellos respecto la horizontal, sino que hemos hecho servir la función `fget_angle`, especificando “a mano” los centros de ambos procesos. Las dos son funciones diferentes con propósitos diferentes, pero en este ejemplo podríamos usar una u otra según nos interese, y haciéndolo bien obtendríamos el mismo resultado. ¿Cómo se podría modificar el código anterior para poder utilizar en vez de `fget_angle`, el comando `get_angle`?

Existen otras posibilidades (como no podía ser de otra manera) a la hora de implementar mecanismos de **persecución de procesos**, a parte del más típico comentado en el apartado anterior cuando se hablaba de la función `Xadvance`.

En ocasiones nos puede interesar, por ejemplo, que un proceso, a partir de colisionar con otro, vaya siguiendo los pasos de éste. ¿Cómo podríamos lograr este efecto? A continuación se presenta un ejemplo que de hecho no hace ningún uso de las funciones acabadas de comentar, sino que utiliza métodos “más manuales”, pero igualmente efectivos.

La idea es simple: tenemos dos procesos: un cuadrado amarillo que será el proceso perseguido y el cual podremos mover con los cursores y un cuadrado blanco que permanecerá inmóvil hasta que el cuadrado amarillo colisione con él, momento en el cual el cuadrado blanco comenzará a perseguir al amarillo pisándole los talones para siempre.

Lo primero que hemos tenido que preguntarnos es en qué proceso escribimos el comando `collision`. ¿El cuadrado amarillo colisiona con el blanco, o viceversa? En realidad, si sólo vamos a incluir estos dos procesos, la respuesta es irrelevante porque tanto da, pero si pensamos en una posible ampliación del programa donde haya más de un proceso potencialmente a perseguir por parte del cuadrado blanco, la respuesta sólo tiene una posibilidad. Si queremos centralizar todas las colisiones que ocurran dentro de nuestro programa entre el único cuadrado blanco y los múltiples procesos a perseguir, el comando `collision` lo deberemos de escribir en el proceso correspondiente al cuadrado blanco, porque así podremos gestionar las colisiones que sufre éste con cualquiera de los otros procesos; si hiciéramos al revés, deberíamos de poner un comando `collision` en cada uno de los procesos perseguidos comprobando su respectiva colisión con el cuadrado blanco, cosa que no es nada óptimo. De momento, sin embargo, sólo incluiremos en nuestro ejemplo un proceso perseguido (el cuadrado amarillo), y posteriormente veremos cómo introducir más procesos de este tipo, acción que si lo hemos hecho bien desde el principio, será trivial.

Una vez resuelto dónde colocamos el comando `collision`, fijémosnos en el código.

```
program hola;
begin
  set_mode(640,480,16);
  perseguido1();
  perseguidor();
loop
  frame;
end
end

process perseguido1()
begin
  graph=new_map(30,30,16);map_clear(0,graph,rgb(255,255,0));
  x=300;y=300;
loop
```

```

    if(key(_left)) x=x-10;end
    if(key(_right))x=x+10;end
    if(key(_up)) y=y-10;end
    if(key(_down)) y=y+10;end
    frame;
end
end

```

```

process perseguidor()

```

```

private

```

```

    int idprota;

```

```

    int flagseguir=0;

```

```

end

```

```

begin

```

```

    graph=new_map(30,30,16);map_clear(0,graph,rgb(255,255,255));

```

```

    x=200;y=200;

```

```

    loop

```

```

        if(idprota=collision(type perseguido1))flagseguir=idprota; end

```

*/\*Los +20 y -20 son para que el perseguidor no se coloque encima del perseguido, sino que se quede al lado. Estos ifs lo que hacen es comprobar si el perseguido está a la derecha/izquierda/arriba/abajo respecto el perseguido, para moverse en consecuencia. Fijarse que el perseguidor va un poco más lento que el perseguido, para causar un efecto chulo.\*/*

```

        if (flagseguir!=0)

```

```

            if (x+20<flagseguir.x)x=x+7;end

```

```

            if (x-20>flagseguir.x)x=x-7;end

```

```

            if (y+20<flagseguir.y)y=y+7;end

```

```

            if (y-20>flagseguir.y)y=y-7;end

```

```

        end

```

```

        frame;

```

```

    end

```

```

end

```

Lo único digno de mención es el proceso perseguidor. Su punto interesante está en el uso de la variable global “flagseguir”: mientras el cuadrado amarillo no colisione con el blanco, esta variable valdrá 0 y mientras valga este valor, no ocurrirá nada. En el momento de la colisión, “flagseguir” pasa a valer el identificador del proceso con el que se ha colisionado -en este caso, el identificador del cuadrado amarillo-, y es entonces cuando el código que hay en el interior del if (ya que flagseguir ya no vale 0) se ejecuta, posibilitando así la persecución, puesto que flagseguir repite que vale el identificador del proceso que acaba de colisionar (es decir, el proceso a perseguir).

Un detalle importante: ¿por qué NO se ha programado el loop del proceso perseguidor así?

```

loop

```

```

    if(idprota=collision(type perseguido1))flagseguir=1; end

```

```

    if (flagseguir!=0)

```

```

        if (x+20<idprota.x)x=x+7;end

```

```

        if (x-20>idprota.x)x=x-7;end

```

```

        if (y+20<idprota.y)y=y+7;end

```

```

        if (y-20>idprota.y)y=y-7;end

```

```

    end

```

```

    frame;

```

```

end

```

Si pruebas el cambio verás que el programa deja de funcionar justo en el momento que se

produce una colisión. Y esto ocurre por lo siguiente: justo en ese momento activamos “flagseguir” para señalar que ha habido colisión y además el comando collision devuelve el identificador del cuadrado amarillo y , tal como hemos hecho, lo guardamos en la variable “idprota”. Pero esto último sólo ocurre mientras haya colisión entre los dos procesos: en el momento en que el proceso perseguido se distancie un poco del cuadrado blanco de manera que deje de haber colisión, el comando collision volverá a devolver 0, y flagseguir mantendrá su valor a 1 igual -se supone que esto nos interesa para seguir manifestando que ha habido colisión y por tanto ha de comenzar la persecución-. El problema viene cuando se quieren evaluar las condiciones  $if(x+20 < idprota.x)$  y similares: cuando se deja de colisionar y por tanto ha de empezar la persecución, acabamos de decir que collision retorna otra vez 0, con lo que “idprota” es 0, con lo que al intentar evaluar estos cuatro ifs problemáticos -ya que “flagseguir” recordemos que vale 1- nos encontramos con que no tenemos proceso a perseguir: el identificador del mismo es 0, o lo que es igual, ningún proceso. Así que no se puede perseguir a nadie y se produce el error.

Imaginémosnos por último que en pantalla hay más procesos susceptibles de colisionar con el cuadrado blanco y ser perseguidos por éste (un cuadrado verde, otro azul,etc). Si tenemos el código anterior, modificarlo para que se adapte a estas nuevas circunstancias es un juego de niños.

```
program hola;
begin
  set_mode(640,480,16);
  perseguido1();
  perseguido2();
  perseguidor();
  loop
    frame;
  end
end
```

```
process perseguido1()
begin
  graph=new_map(30,30,16);map_clear(0,graph,rgb(255,255,0));
  x=300;y=300;
  loop
    if(key(_left)) x=x-10;end
    if(key(_right))x=x+10;end
    if(key(_up)) y=y-10;end
    if(key(_down)) y=y+10;end
    frame;
  end
end
```

```
process perseguido2()
begin
  graph=new_map(30,30,16);map_clear(0,graph,rgb(255,0,255));
  x=100;y=100;
  loop
    if(key(_a)) x=x-10;end
    if(key(_d)) x=x+10;end
    if(key(_w)) y=y-10;end
    if(key(_s)) y=y+10;end
    frame;
  end
end
```

```

process perseguidor()
private
  int idprota;
  int flagseguir=0;
end
begin
  graph=new_map(30,30,16);map_clear(0,graph,rgb(255,255,255));
  x=200;y=200;
  loop
    if(idprota=collision(type perseguido1) or collision(type perseguido2))flagseguir=idprota; end
    /*Los +20 y -20 son para que el perseguidor no se coloque encima del perseguido, sino que se quede al
    lado.Estos ifs lo que hacen es comprobar si el perseguido está a la derecha/izquierda/arriba/abajo
    respecto el perseguido, para moverse en consecuencia. Fijarse que el perseguidor va un poco más lento
    que el perseguido, para causar un efecto chulo.*/
    if (flagseguir!=0)
      if (x+20<flagseguir.x)x=x+7;end
      if (x-20>flagseguir.x)x=x-7;end
      if (y+20<flagseguir.y)y=y+7;end
      if (y-20>flagseguir.y)y=y-7;end
    end
    frame;
  end
end
end

```

Lo único que se ha hecho (aparte de incluir un nuevo proceso perseguido: un cuadrado rosa) es incluir la posibilidad, dentro del código del proceso perseguidor, de colisionar o bien con un proceso de tipo “perseguido1” o bien “perseguido2”. Y ya está: todo lo demás permanece igual.

Comentar finalmente que, si pruebas este último código, verás que el cuadrado blanco es muy “promiscuo”. Si está persiguiendo un determinado proceso y en un momento dado colisiona con otro proceso diferente, el perseguidor abandona a su antiguo perseguido para comenzar a perseguir al que acaba de colisionar con él. ¿Se te ocurre alguna manera evitar esto, es decir, para hacer que una vez el proceso perseguidor haya empezado a perseguir a otro proceso, ya no se cambie el proceso perseguido por ningún otro más?

Otro ejemplo más sofisticado de persecución de procesos (donde sí utilizamos funciones como `get_angle`, `get_dist`, `get_distx` o `get_disty`) es el siguiente. Si lo ejecutas verás que tienes un cuadrado verde (el “protagonista”) que lo podrás mover con los cursores, y tres cuadrados rojos (los “enemigos”) que harán dos cosas: por un lado siempre estarán encarados al protagonista: es decir, su ángulo será tal que en cada movimiento que haga el protagonista, los enemigos irán actualizando su orientación para “vigilarlo” permanentemente. Este efecto, aunque no lo parezca, en realidad no es necesario para lo siguiente. Y lo siguiente es que, si el protagonista se acerca demasiado a un enemigo (ya que éste tiene definida una zona de “sensibilidad”), el enemigo se “enterará” de la presencia del protagonista y éste será perseguido con insistencia hasta que se aleje lo suficiente para salir de esa zona de “sensibilidad”, momento en el que el enemigo dejará de “notar” que hay alguien cerca y se volverá a quedar quieto. Una aplicación directa de este código la veremos en el capítulo-tutorial sobre RPG.

```

PROGRAM Persec;
BEGIN
  set_mode(640,480,16);
  protagonista(320,240);

```

```

enemigo(100,200);
enemigo(50,100);
enemigo(500,300);
loop
    frame;
    if(key(_esc)) exit();end
end
END

PROCESS protagonista(x,y)
BEGIN
graph=new_map(30,30,16);map_clear(0,graph,rgb(0,255,0));
loop
    if(key(_left)) x=x-4;end
    if(key(_right))x=x+4;end
    if(key(_up)) y=y-4;end
    if(key(_down)) y=y+4;end
    frame;
end
END

Process enemigo(x,y)
private
    int idprota;
    int angleprota;
    int move_x,move_y;
end
Begin
graph=new_map(30,30,16);map_clear(0,graph,rgb(255,0,0));
Loop
//Obtengo el identificador del proceso protagonista (sólo hay uno)
    idprota=get_id(type protagonista);
//Obtengo el ángulo entre el proceso enemigo actual y el protagonista, para...
    angleprota=get_angle(idprota);
/*...hacer que el enemigo esté siempre "mirando" al protagonista. Ésta es la línea que lo hace posible:
con un poco de geometría en un papel podrás ver de dónde sale la fórmula: sólo hay que tener claro qué
ángulo representa "angleprota" y qué ángulo es "angle".*/
    angle=angleprota-90000;

/*2ª parte: Si el protagonista se acerca demasiado al enemigo, el cual siempre lo estará vigilando, quiero
que éste sea perseguido. Para ello podría utilizar la función get_dist, pero se va a hacer de otra manera,
por componentes. Primero obtengo las componentes X e Y de la distancia al protagonista, para...*/
    move_x=get_distx(angleprota,distprota);
    move_y=get_disty(angleprota,distprota);

/*..comprobar unas condiciones determinadas con estos cuatro ifs. Estas condiciones provocan que, si las
componentes X e Y de la distancia están dentro de un rango determinado, (es decir, si el protagonista está
demasiado cerca del enemigo), éste se mueva en una determinada dirección, que es la dirección donde se
supone está el protagonista. Las condiciones de los ifs se pueden ver de dónde salen si coges un papel y
un lápiz y dibujas estas cuatro condiciones posibles. El rango -los números de las condiciones del if- se
puede cambiar para hacer que el enemigo sea más o menos sensible a la presencia del protagonista. */

/*Esta condición ocurrirá cuando el protagonista está a la izquierda del enemigo (entre 20 y 100 píxeles
horizontalmente) y a su misma coordenada vertical más/menos 100 píxeles*/

```

```

    if ((move_x<-20 and move_x>-100) and (move_y>-100 and move_y<100))
/*Notar que el desplazamiento del enemigo no lo hago en función de su ángulo, sino
en base a movimientos horizontales o verticales. Lo de las líneas anteriores que
hacían que el enemigo estuviera encarando al protagonista si éste se acercaba
demasiado es un efecto puramente estético*/
        x=x-1;
    end
/*Esta condición ocurrirá cuando el protagonista está a la derecha del enemigo (entre 20 y 100 píxeles
horizontalmente) y a su misma coordenada vertical más/menos 100 píxeles.*/
    if ((move_x>20 and move_x<100) and (move_y>-100 and move_y<100)) x=x+1; end
/*Esta condición ocurrirá cuando el protagonista está por arriba del enemigo (entre 20 y 100 píxeles
verticalmente) y a su misma coordenada horizontal más/menos 100 píxeles.*/
    if ((move_y<-20 and move_y>-100) and (move_x>-100 and move_x<100)) y=y-1; end
/*Esta condición ocurrirá cuando el protagonista está por debajo del enemigo (entre 20 y 100 píxeles
verticalmente) y a su misma coordenada horizontal más/menos 100 píxeles.*/
    if ((move_y>20 and move_y<100) and (move_x>-100 and move_x<100)) y=y+1; end
    Frame;
end //loop
END

```

En el ejemplo anterior hemos visto, entre otras cosas, cómo hacer para que un proceso encare a otro permanentemente (es decir, que un proceso varíe su ángulo según se mueva otro proceso, apuntándolo permanentemente, como por ejemplo cuando un enemigo vigila sin descanso al protagonista).

De todas maneras, tal como lo hemos hecho, es posible que no nos venga bien en determinadas circunstancias. Imagínate que en vez de cuadrados estamos trabajando con gráficos de personajes: el cuadrado verde es nuestro héroe valeroso y los cuadrados rojos en realidad son feroces orcos, con un gráfico asociado tal como éste:



Para que veas lo que pretendo decirte, cambia el código anterior para que en vez de tener cuadrados rojos, tengas el gráfico anterior o cualquier otro. Yo a este gráfico lo he llamado "orco.png". Te vuelvo a poner el código haciendo este cambio (quitando las líneas que no nos interesan: ahora no queremos perseguir al protagonista):

```

PROGRAM Orient;
Global
    int idorco,idorco2,idorco3;
End
BEGIN
set_mode(640,480,16);
idorco=load_png("orco.png");
idorco2=load_png("orco2.png");
idorco3=load_png("orco3.png");
protagonista(320,240);

```

```

enemigo(100,200);
enemigo(50,100);
enemigo(500,300);
loop
  frame;
  if(key(_esc)) exit();end
end
END

PROCESS protagonista(x,y)
BEGIN
graph=new_map(30,30,16);map_clear(0,graph,rgb(0,255,0));
loop
  if(key(_left)) x=x-4;end
  if(key(_right))x=x+4;end
  if(key(_up)) y=y-4;end
  if(key(_down)) y=y+4;end
  frame;
end
END

Process enemigo(x,y)
private
  int idprota;
  int angleprota;
end
Begin
graph=idorco;
Loop
  idprota=get_id(type protagonista);
  angleprota=get_angle(idprota);
  angle=angleprota-90000;
  frame;
end
end

```

Si pruebas este código verás lo que ocurre. El gráfico de los orcos cambia su orientación según la posición del cuadrado verde, tal como esperábamos, pero el efecto ¡queda fatal! Los orcos parecen astronautas que están flotando a expensas de nuestro protagonista. ¿Cómo podemos hacer que, en vez de esto, lo que ocurra es que si se detecta que el protagonista está a un cierto ángulo del orco, el gráfico de éste pase automáticamente a ser otro, como éste, llamado “orco2.png”:



si el héroe vaga por su derecha, o éste otro (llamado “orco3.png”):



si va por su izquierda?

Bueno, tendremos que modificar nuestro código así (las líneas nuevas están marcadas en negrita):

```
PROGRAM Orient;
Global
    int idorco,idorco2,idorco3;
End
BEGIN
set_mode(640,480,16);
idorco=load_png("orco.png");
idorco2=load_png("orco2.png");
idorco3=load_png("orco3.png");
protagonista(320,240);
enemigo(100,200);
enemigo(50,100);
enemigo(500,300);
loop
    frame;
    if(key(_esc)) exit();end
end
END

PROCESS protagonista(x,y)
BEGIN
graph=new_map(30,30,16);map_clear(0,graph,rgb(0,255,0));
loop
    if(key(_left)) x=x-4;end
    if(key(_right))x=x+4;end
    if(key(_up)) y=y-4;end
    if(key(_down)) y=y+4;end
    frame;
end
END

Process enemigo(x,y)
private
    int idprota;
    int angleprota;
    int pepe;
end
Begin
graph=idorco;
Loop
    idprota=get_id(type protagonista);
    angleprota=get_angle(idprota);
    pepe=angleprota-90000;
```

*/\*Piensa que "pepe" tiene el siguiente rango de valores: vale desde 0 en los puntos que están alineados justo en la línea vertical centrada en el enemigo pero por encima de éste, hasta 180 en los puntos de la misma línea pero por debajo del enemigo siempre que el proceso protagonista esté a la izquierda del enemigo. Si el protagonista está a la derecha del enemigo, pepe valdrá desde 0 hasta -179 yendo desde la perpendicular de encima del enemigo hasta la de debajo suyo.. Esto lo puedes ver si sólo tienes un proceso enemigo funcionando e incluyes fuera de este loop la línea write\_var(0,100,100,4,pepe); para ver los valores de pepe según la posición del protagonista, y tener así una guía para construir las condiciones de los ifs siguientes.\*/\**

```
if(pepe>45000 and pepe<135000)flags=0;graph=idorco3; end //izquierda
if(pepe>-135000 and pepe<-45000)flags=0;graph=idorco2;end //derecha
if((pepe>0 and pepe<45000) or (pepe<-1 and pepe>-45000)) flags=2; graph=idorco; end //arriba
if((pepe>135000 and pepe< 180000) or (pepe<-135000 and pepe>-179000)) flags=0;graph=idorco;
end //abajo
frame;
end
end
```

Lo que hemos hecho ha sido cambiar la antigua línea que alteraba la orientación de los enemigos (ya que cambiaba su ANGLE) por otra muy similar donde el ángulo corregido que nos da la posición relativa del protagonista respecto el enemigo lo almacenamos en una variable privada que la hemos llamado “pepe”. Así, para empezar, los enemigos no cambiarán la orientación de su gráfico, pero continuarán sabiendo dónde está el protagonista en cada momento, gracias a “pepe”.

La idea es, que según en qué zona respecto el enemigo se sitúe el protagonista (zona superior, derecha, izquierda, inferior), el enemigo cambiará su gráfico por uno acorde, dando la sensación de ojo avizor permanente, ya que los enemigos estarán permanentemente mirando al protagonista, esté donde esté..

Está claro que nos podríamos haber ahorrado el gráfico “orco3.png” jugando con los flags (al igual que hemos hecho cuando el protagonista está arriba o abajo del enemigo).

En ejemplos anteriores ya hemos podido ver que uno de los problemas típicos que tendremos que solventar a menudo en nuestros juegos es averiguar la **distancia mínima** entre diferentes procesos. Para resolver esta necesidad ya sabes que contamos con la función get\_dist y sus “parientes” get\_distx y get\_disty..

Un caso típico es el de los juegos de estrategias, donde los escuadrones tienen que decidir a qué enemigo atacar y deciden atacar al más cercano. Por eso se va a presentar a continuación algunos ejemplos de código donde hay un proceso que detecta al más cercano de todos sus “enemigos”.

Para empezar, pondremos un código que muestra por pantalla un cuadrado blanco en la esquina inferior derecha de la pantalla (nuestro proceso “prota”) y 400 cuadrados azules repartidos aleatoriamente por toda la pantalla (procesos “enemigos”). El programa lo primero que hará será obtener todas las distancias de cada enemigo con el proceso “prota” e seguidamente irá matando uno a uno todos los procesos enemigos, por orden de cercanía al “prota”. Es decir, primero morirá (y por tanto desaparecerá de la pantalla) el enemigo más cercano al cuadrado blanco, después el segundo más cercano, etc hasta llegar al último enemigo, el más lejano. El efecto resultante es como si hubiera una onda expansiva en forma de círculo que se engrandece y que arrasa los cuadrados azules.

```
program hola;
global
    int idenemigo[399];
```

```

end
private
    int i;
end
begin
    set_mode(640,480,16);
    prota();
    for(i=0;i<400;i++)
        idenemigo[i]=enemigo(rand(10,600),rand(10,400));
    end
    while(!key(_esc)) frame; end
end

process prota()
private
    int i;
    int objetivo;
    int distaux, distminima;
end
begin
    graph=new_map(5,5,16);map_clear(0,graph,rgb(255,255,255));
    x=600;y=450;
    loop
    /*Típico algoritmo para encontrar el valor mínimo de una lista de valores: se va recorriendo los valores
    del vector y si se encuentra un valor menor al que hasta ahora era el mínimo, este valor pasa a
    ser el nuevo mínimo.Se empieza por un valor mínimo muy alto para que el primer elemento sea siempre el
    mínimo y se puedan hacer las comparaciones.*/
        distminima=10000;
        for(i=0;i<400;i++)
            distaux=get_dist(idenemigo[i]);
        /*La condición de "distaux>0" es MUY importante.Ya que cada vez que se mate un enemigo, se volverá a
        realizar este algoritmo otra vez para encontrar al siguiente enemigo más cercano, el vector idenemigo[i]
        tendrá cada vez más elementos que corresponderán a identificadores de procesos ya muertos. Si
        realizamos un get_dist con un identificador no existente, get_dist nos devuelve 0, con lo que siempre
        tendríamos como distancia mínima la calculada con procesos inexistentes. Evidentemente, esto está
        fatal.*/
            if(distaux<=distminima and distaux>0)
                distminima=distaux;
                objetivo=idenemigo[i];
            end
        end
        write_var(0,100,100,3,objetivo); //Vemos el identificador del enemigo a destruir
        signal(objetivo,s_kill);
        frame;
    end //loop
end

process enemigo(x,y)
begin
    graph=new_map(5,5,16);map_clear(0,graph,rgb(0,0,255));
    loop
        frame;
    end
end

```

Vamos a jugar con este código un rato. Escribe la siguiente línea: *signal(objetivo,s\_kill)*; al final del if del proceso “prota”. ¿Qué pasa si ejecutas el programa ahora? Que los enemigos son asesinados a un ritmo mucho más vertiginoso. ¿Podrías explicar el por qué? Porque estamos eliminando todos aquellos procesos que MIENTRAS se está recorriendo el bucle sean designados “objetivo”, pero el objetivo correcto sólo se puede saber DESPUES de haber recorrido todo el bucle, que es cuando se sabrá el proceso a la menor distancia de entre todos: si se mata mientras se va recorriendo el bucle, se irán encontrando varios objetivos, uno por cada proceso que tenga una distancia menor que los anteriores, pero que no necesariamente sea la mínima total, porque no se ha acabado de recorrer el bucle entero.

Importante recalcar que las órdenes *signal* se ejecutarán inmediatamente, sin esperar a ninguna línea Frame;. Esto quiere decir que si mientras estamos dentro del bucle se ejecutan varias órdenes *signal*, una para cada objetivo encontrado diferente, todas estas órdenes *s\_kill* diferentes se ejecutarán inmediatamente, sin esperar a llegar a ningún fotograma. Es por eso que la eliminación es mucho más rápida: porque cuando por fin se llega a un fotograma, ya se han encontrado antes varios objetivos que han sido eliminados.

Ahora vamos a hacer otra cosa. Vamos a hacer que nuestro cuadradito blanco se desplace a la posición del enemigo más cercano, éste se muera y el cuadradito blanco se vuelva a desplazar al siguiente enemigo más cercano, éste se muera, y así. Para ello, lo único que tenemos que hacer es añadir las líneas en negrita al ejemplo anterior:

```

program hola;
global
    int idenemigo[399];
end
private
    int i;
end
begin
    set_mode(640,480,16);
    prota();
    for(i=0;i<400;i++)
        idenemigo[i]=enemigo(rand(10,600),rand(10,400));
    end
    while(!key(_esc))frame; end
end

process prota()
private
    int i;
        int objetivo;
        int distaux, distminima;
end
begin
    graph=new_map(5,5,16);map_clear(0,graph,rgb(255,255,255));
    x=600;y=450;

    loop
        distminima=10000;
        for(i=0;i<400;i++)
            distaux=get_dist(idenemigo[i]);
            if(distaux<distminima and distaux>0)
                distminima=distaux;

```

```

        objetivo=idenemigo[i];
    end
end
write_var(0,100,100,3,objetivo);
if(exists(objetivo)==true) x=objetivo.x;end
if(exists(objetivo)==true) y=objetivo.y;end
signal(objetivo,s_kill);
frame;
end //loop
end

process enemigo(x,y)
begin
graph=new_map(5,5,16);map_clear(0,graph,rgb(0,0,255));
loop frame; end
end

```

Si ejecutas este proceso verás que ahora ya no aparece el efecto de onda expansiva, sino que como las distancias mínimas se recalculan en cada iteración del LOOP principal, y precisamente en cada una de éstas cambiamos las coordenadas X e Y del “protagonista” por las que en ese momento tiene el objetivo actual -si existe-, lo que obtenemos es nuevas medidas de las distancias en cada nueva posición del cuadradito blanco, con lo que los enemigos que mueren son los que están más cerca del protagonista en un momento dado.

Otro código muy interesante donde se puede ver la utilidad de la detección de las distancias mínimas entre procesos, y sus orientaciones recíprocas es el siguiente.

Este ejemplo representa el combate de dos ejércitos: uno azul y otro rojo. El usuario puede generar soldados azules con el botón derecho del ratón, y los soldados rojos se generarán en posiciones aleatorias cada x tiempo. La gracia del asunto radica en que cada soldado de ambos bandos se irá aproximando a aquel soldado del otro bando (y sólo del otro bando) que tenga más cerca. En el momento que llegue a una distancia mínima, ese soldado se parará para comenzar a dirigir a su enemigo disparos de flechas sin parar. Cuando un soldado recibe un número determinado de éstas, muere y desaparece de pantalla. Además, existe el modo “formación de combate”, invocado con la tecla SPACE, el cual consiste en lo mismo pero donde los soldados de ambos bandos se sitúan ordenadamente a cada lado de la ventana en forma de escuadrón listo para la batalla.

Lo interesante del ejemplo es sobretodo el sistema de detección del enemigo más cercano (similar al de los ejemplos anteriormente vistos), la aproximación a éste y el direccionamiento hacia éste de las flechas.

```

/*Este tutorial explica cómo se puede detectar la distancia más corta entre todos
los soldados enemigos (cuadrados rojos -bando 2-) y nuestros personajes (cuadrados azules -bando 1-)*
PROGRAM tutorts;
global
int idsold;
int idobjetivo;
int grafflecha;
end
local
int bando;
int energia;

```

```

end
private
  int i,j;
  int grafsold1;
  int grafsold2;
  int modonormal=1; /*Variable que vale 1 si el juego está en modo "normal" y 0 si está en modo
"Combate en formación" (al pulsar la tecla SPACE)*/
end
BEGIN
  set_mode(640,480,16);
  set_fps(30,0);
  write(0,10,10,3,"Utilizar el ratón para poner tus personajes y testear las distancias");
  grafsold1=new_map(10,10,16);map_clear(0,grafsold1,rgb(0,0,255)); //Gráfico de tus personajes
  grafsold2=new_map(10,10,16);map_clear(0,grafsold2,rgb(255,0,0)); //Gráfico de tus enemigos
  grafflecha=new_map(5,5,16);drawing_map(0,grafflecha);draw_box(0,0,5,2);//Gráfico de la flecha
  mouse.graph=grafsold1;mouse.flags=4;
  /*Aparición de 5 soldados enemigos y 5 propios no importa dónde*/
  FROM i=0 TO 5
    soldado(rand(20,620),rand(20,380),grafsold1,1); //Soldado bando 1 (nuestros personajes)
    soldado(rand(20,620),rand(20,380),grafsold2,2); //Soldado bando 2 (nuestros enemigos)
  END
  LOOP
    //Si se clicla el botón derecho del ratón, aparecerá un soldado del bando 1 (nuestros personajes)
    IF(mouse.right)soldado(mouse.x,mouse.y,grafsold1,1); END
    /*Modo combate en formación. Se eliminan los soldados que estuvieran en este momento y se generan
nuevos soldados colocados en formación*/
    IF(key(_space))
      IF(modonormal==1) //Para no hacer un modo formación si ya hay uno funcionando
        signal(TYPE soldado,s_kill);
        FROM i=10 TO 80 STEP 10
          FROM j=10 TO 390 STEP 20
            soldado(i,j,grafsold1,1);
            soldado(640-i,j,grafsold2,2);
          END
        END
        modonormal=0; //Después de haber creado la formación, vuelvo al estado "normal", donde los
enemigos se crean en posiciones aleatorias
      END
    ELSE
      modonormal=1;
      //Aparición de los soldados del bando 2 aleatoriamente
      IF(rand(0,100)<5)soldado(rand(20,620),rand(20,380),grafsold2,2);END
    END
  FRAME;
END //Loop
END

/*Proceso de los soldados de los dos campos*/
PROCESS soldado(x,y,graph,int bando)
private
  int tiro;
  int dist;
end
BEGIN

```

```

energia=50;
LOOP
//Detección del soldado enemigo (entre sí) más cercano (el algoritmo ya es conocido)
idobjetivo=0;
dist=1000;
WHILE(idsold=get_id(TYPE soldado)) //Mira todos los procesos soldados
    IF(idsold.bando<>bando) //Si el bando del proceso soldado es diferente de este soldado...
//...detecta si la distancia entre el soldado es inferior a la distancia guardada anteriormente
        IF(get_dist(idsold)<dist)
            dist=get_dist(idsold); //Si es inferior, guarda la distancia actual
            idobjetivo=idsold; //El objetivo a aniquilar será ese soldado
        END
    END
END

//Si hay un objetivo fijado, después de haber pasado por el bucle anterior...
IF(idobjetivo<>0)
//.y si la distancia entre el soldado y el objetivo es inferior a 100, le lanzo una flecha
    IF(get_dist(idobjetivo)<200)
        tiro=tiro+1;
        /*Este if es simplemente para reducir la frecuencia con el que se lanzarán las flechas, ya que si no
se pusiera se estarían generando flechas a cada frame. La restricción por un lado es hacer que un número
aleatorio sea menor que una cantidad dada (sistema visto en el capítulo anterior del tutorial del
matamarcianos), pero además se ha de cumplir una condición más que es que la variable tiro (la cual va
aumentando su valor a cada frame en la línea anterior) ha de ser mayor de 15*/
        IF(rand(0,100)<10 AND tiro>15)
            tiro=0; //Se vuelve a resetear "tiro"
            flecha(x*10,y*10,get_angle(idobjetivo),grafflecha); /*La flecha sale dirigida en
dirección al objetivo. El 10 que se multiplica es el valor de la "resolution" del proceso flecha.*/
        END
        /*Si la distancia es mayor, en vez de lanzar una flecha, me muevo hacia él, de tal manera que llegará
un momento en que la distancia ya sí sea menor y pueda entonces pararme y proceder al disparo*/
        ELSE
            x=x+get_distx(get_angle(idobjetivo),3);
            y=y+get_disty(get_angle(idobjetivo),3);
        END
    END

    IF(energia<=0) signal(id,s_kill); END //Si no tengo energía, muero
    FRAME;
END
END

/*Las flechas*/
PROCESS flecha(x,y,angle,graph)
Private
    int vidaflecha;
end
BEGIN
    bando=father.bando; //La flecha será del bando de su padre
    resolution=10; //Para que las flechas salgan dirigidas hacia el objetivo de forma más precisa
    LOOP
        //A cada frame, la flecha avanza y aumenta su tiempo de vida
        advance(5*resolution);vidaflecha=vidaflecha+1;
    END
END

```

```

//Si la flecha choca contra un soldado...
IF(idsold=collision(TYPE soldado))
  //y si el soldado es diferente de su bando...
  IF(idsold.bando<>bando)
    //...este soldado pierde 10 puntos de energía
    idsold.energia=idsold.energia-10;signal(id,s_kill);
  END
END
//Si la flecha sale de la pantalla o ya ha volado demasiado, muere
IF(out_region(id,0)OR vidaflecha>50) signal(id,s_kill); END
FRAME;
END
END

```

En general, podemos concluir que cuando se tienen muchos procesos de un tipo, como en el caso anterior con los 400 enemigos, es recomendable por claridad y comodidad contar en nuestro programa con un proceso “controlador”, que gestione de forma centralizada la creación/manipulación/destrucción de múltiples procesos. De esta forma se gana en limpieza de código y rapidez en la depuración de errores. Por ejemplo, un proceso controlador tipo sería el siguiente:

```

process controlador ()
private
  enemigos[los-que-necesites];
end
begin
  enemigos[0]=enemigo(); //Creación de múltiples procesos y almacenamiento de su ID en un vector
  enemigos[1]=enemigo();
  ...
  enemigos[los-que-necesites]=enemigo();

//Bucle para comprobar todos los enemigos
for (cont=0;cont<=los-que-necesites;cont++)
/*Aquí podemos escribir lo que necesitamos. En este caso por ejemplo miramos si un enemigo concreto
(sólo uno cada vez) está dentro de un determinado rango de acción. Si es así, lo despertamos; si no, lo
dormimos.*/
  if (enemigos[cont].x<distancia_minima_x and enemigos[cont].y<distancia_minima_y)
    signal(enemigos[cont],s_wakeup); //Despertamos a ese enemigo
  else //De lo contrario, lo congelamos y le quitamos el gráfico
    signal(enemigos[cont],s_freeze); //Puedes usar sleep, que le quita el gráfico, si te gusta más
    enemigos[cont].graph=0;
  end
end //for
...
loop
  frame;
end
end

```

### NEAR\_ANGLE(ANGULO1,ANGULO2,MAX\_INC)

Esta función suma a ANGULO1 un número de grados dado por MAX\_INC y devuelve el nuevo ángulo, siempre y cuando el resultado no sea un ángulo mayor a ANGULO2, en cuyo caso el ángulo devuelto será exactamente ANGULO2.

Es decir, dicho de otro modo, si la distancia entre los dos ángulos es más pequeña que MAX\_INC, entonces el ángulo resultante devuelto será exactamente ANGULO2.

El valor de MAX\_INC ha de ser positivo, para que el proceso de aproximación se realice por el lado de la circunferencia donde la distancia entre los dos ángulos sea más pequeña.

Esta función sirve de ayuda para hacer que procesos puedan perseguir a otros pero con una capacidad limitada de giro en cada frame.

PARAMETROS:

INT ANGULO1: Ángulo original en milésimas de grado (90000= 90°)

INT ANGULO2 : Ángulo de destino

INT MAX\_INC: Número máximo de milésimas de grado de incremento del ángulo original

Un ejemplo trivial para probar esta función sería algo como éste:

```
program Test_NEAR_ANGLE;
global
  int ang=10000,ang2=20000,ang3;
end
begin
  set_mode(640,480,16);
  repeat
    delete_text(0);
    ang3=near_angle(ang,ang2,9999);
    write(0,100,100,4,ang);
    write(0,100,110,4,ang2);
    write(0,100,120,4,ang3);
  frame;
  until(key(_esc))
end
```

donde se puede ir cambiando el valor del tercer parámetro de near\_angle para probar los efectos.

Otro ejemplo más útil es el siguiente:

```
program Test_NEAR_ANGLE;
global
  int ang;
end
begin
  set_mode(640,480,16);
  graph=new_map(10,10,16);
  map_clear(0,graph,rgb(0,0,255));
  mouse.graph=new_map(10,10,16);
  map_clear(0,mouse.graph,rgb(255,255,0));
  repeat
    ang=fget_angle(x,y,mouse.x,mouse.y);
    angle=near_angle(angle,ang,10000);
    advance(5);
  frame;
  until(key(_esc))
end
```

¿No ves lo que pasa? Sustituye la línea donde aparece `near_angle` por ésta otra: `angle=ang` . Comprueba qué pasa. ¿Podrías explicar por qué?

Y otro:

```
program homing_missiles;
local
  int count=0; //Cada proceso target tiene su contador de impactos
end
private
  int graphic; //Para el gráfico del cursor del ratón
end
begin
  set_mode(400,600,16);
  set_fps(30,0);
  target(50,50); //4 objetivos inmóviles que recibirán el impacto de los misiles
  target(250,70);
  target(70,200);
  target(280,180);
  graphic=draw_circle(x,y,2); //El cursor del ratón es una primitiva
loop
  move_draw(graphic,mouse.x,mouse.y);
  if(mouse.left)missile(mouse.x,mouse.y);end //Los misiles partirán del cursor
frame;
end
end

process target(x,y)
private
  int dist; //Distancia de cada objetivo al cursor del ratón
end
begin
  write_var(0,x,y-5,4,count); //Muestro el número de impactos que lleva recibidos
  write_var(0,x,y+7,4,dist); //Muestro la distancia al cursor del ratón en tiempo real
  //Los objetivos son simples pixels blancos
  put_pixel(x,y,65535);
  put_pixel(x+1,y,65535);
  put_pixel(x,y+1,65535);
  put_pixel(x+1,y+1,65535);
loop
  dist=fget_dist(mouse.x,mouse.y,x,y); //Distancia al cursor del ratón
frame;
end
end

process missile(x,y)
private
  int mindist;
  int idmascercano;
  int tempdist;
  int tempid;
end
begin
  angle=1000*rand(70,110); //Ángulo inicial del misil es aleatorio
```

```

loop
  /*Borro el rastro actual del misil en el fotograma anterior (de hecho, lo que hago
  es pintarlo de negro -o sea, transparente-), antes de que avance una nueva posición */
  put_pixel(x,y,rgb(0,0,0));
  put_pixel(x+1,y,rgb(0,0,0));
  put_pixel(x,y+1,rgb(0,0,0));
  put_pixel(x+1,y+1,rgb(0,0,0));

  /*Vuelvo a usar el mismo algoritmo de siempre para detectar el objetivo más cercano -distancia
  mínima- en cada frame respecto al misil actual. Al final obtengo "idmascercano" (que representa el id de
  uno de los cuatro objetivos) y "mindist", (que es la distancia de éste respecto al misil)*/
  mindist=10000;
  idmascercano=0;
  while(tempid=get_id(type target))
    if((tempdist=fget_dist(x,y,tempid.x,tempid.y))<mindist)
      idmascercano=tempid;
      mindist=tempdist;
    end
  end

  //Si se encuentra un objetivo cercano...
  if(exists(idmascercano))
  /*...y la distancia de éste respecto al misil es menor que una dada, aumento el contador de ese objetivo, y
  mato el misil correspondiente*/
    if(mindist<8)idmascercano.count++;return;end
    //El misil varía su orientación para teledirigirse de forma espiral sobre algún objetivo
    angle=near_angle(angle,fget_angle(x,y,idmascercano.x,idmascercano.y),7500);
  end
  advance(9);

  //El misil no es más que simples pixels
  put_pixel(x,y,rgb(255,0,0));
  put_pixel(x+1,y,rgb(255,0,0));
  put_pixel(x,y+1,rgb(255,0,0));
  put_pixel(x+1,y+1,rgb(255,0,0));
  frame;
end //loop
end

```

### **Trabajar con fades y efectos de color:**

#### **FADE(R,G,B,Velocidad)**

Esta función activa una transición de colores automática, que puede usarse para oscurecer la pantalla, hacer un fundido, o colorear su contenido.

Llamar a la función simplemente activa ciertas variables internas y pone a 1 la variable global **FADING**. A partir de entonces, el motor gráfico se encargará automáticamente de colorear la pantalla cada frame.

Los valores de componentes R, G y B recibidos por esta función se dan en porcentaje. Un porcentaje de 100% dejará un color sin modificar, mientras un porcentaje de 0% eliminará por completo la componente. También es posible especificar un valor por encima de 100. Se considera que un valor de 200 en una componente, debe poner al máximo la intensidad de la misma.

Estos valores permiten hacer combinaciones flexibles, como por ejemplo un fundido al blanco

(200,200,200) o bien aumentar la intensidad de color de una o varias componentes (150,150,100). Tal vez el valor más habitual sea el fundido al negro (0,0,0) o restaurar el estado normal de los colores (100,100,100).

El valor de velocidad es un valor entero entre 1 y 64. Un valor de 64 activaría los porcentajes de color indicados de forma inmediata en el próximo frame, mientras un valor de 1 iría modificando las componentes de forma sucesiva, de forma que alcanzarían los valores especificados como parámetro en aproximadamente unos 60 frames. Un valor intermedio permite realizar la transición más rápidamente.

La variable global **FADING** estará a 1 inmediatamente después de llamar a esta función. El motor gráfico la pondrá a 0 cuando la transición alcance los colores especificados como parámetro.

En modo de 16 bits esta función modifica efectivamente los valores de todos los pixels en pantalla, realizando una costosa operación por cada uno de ellos. Esta operación es muy lenta y representa un coste considerable en la tasa de frames por segundo. Si se desea hacer un juego con efectos de colorización globales, se recomienda usar los efectos localmente en una zona reducida de la pantalla mediante blendops. Si ello no fuera posible, entonces es preferible escoger una resolución de pantalla limitada para aumentar el rendimiento. Hay que tener en cuenta que, incluso una vez acabada la transición, si las componentes finales de los colores no son 100 para las tres, el motor gráfico seguirá modificando los pixels de pantalla cada frame, por lo que el perjuicio al rendimiento no se limitará al período que dure la transición y seguirá presente una vez acabada esta.

#### PARAMETROS:

INT R: Cantidad de componente Rojo  
INT G: Cantidad de componente Verde  
INT B: Cantidad de componente Azul  
INT Velocidad: Velocidad

Un ejemplo de esta función podría ser éste (necesitarás un gráfico "d.png" para ponerlo de fondo):

```
program Test_FADE;
global
  int idpng;
  int r=100,g=100,b=100,s=3;
end
begin
  set_mode(640,480,16);
  idpng=load_png("d.png");
  put_screen(0,idpng);
  write(0,10,60,3,"1: Fundido en blanco (fade 200,200,200):");
  write(0,10,70,3,"2: Fundido en negro (fade 0,0,0):");
  write(0,10,80,3,"3: Fundido personalizado (fade %R,%G,%B):");
  write(0,20,90,3,"R: Nuevo valor de R (en porcentaje): ");
  write_var(0,250,90,3,r);
  write(0,20,100,3,"G: Nuevo valor de G (en porcentaje): ");
  write_var(0,250,100,3,g);
  write(0,20,110,3,"B: Nuevo valor de B (en porcentaje): ");
  write_var(0,250,110,3,b);
  write(0,10,140,3,"S: Nuevo valor de Velocidad: ");
  write_var(0,190,140,3,s);
  repeat
    if(key(_1))
      fade(255,255,255,s);
      while(fading==1)
        frame;
```

```

        if(key(_esc)) exit(); end
    end
end
if(key(_2))
    fade(0,0,0,s);
    while(fading==1)
        frame;
        if(key(_esc)) exit(); end
    end
end
if(key(_3))
    fade(r,g,b,s);
    while(fading==1)
        frame;
        if(key(_esc)) exit(); end
    end
end
if(key(_r)) r++; if(r>200) r=0; end end
if(key(_g)) g++; if(g>200) g=0; end end
if(key(_b)) b++; if(b>200) b=0; end end
if(key(_s)) s++; if(s>63) s=1; end end
frame;
until(key(_esc))
end

```

El funcionamiento del ejemplo es el siguiente: si se apreta la tecla 1, se realiza un fundido en blanco; si se pulsa la tecla 2, se realiza un fundido en negro; si se apreta la tecla 3, se realiza un fundido partiendo de los valores en porcentaje actuales de las tres componentes, los cuales se pueden modificar con las teclas “r”, “g” y “b” respectivamente. Para cualquiera de los tres fades posibles, se aplicará la velocidad actual de fundido, la cual se puede modificar con la tecla “s”.

Fijarse que es muy importante el bucle `while(fading==1) frame; end`, porque es el que posibilita que mientras esté durando el fundido éste se pueda visualizar en pantalla, puesto que obliga a pasar fotogramas sin hacer otra cosa hasta que el fundido se acabe. Si no se hiciera así, el fundido se estaría realizando pero hasta que no se encontrara una orden `frame` no podría mostrarse su estado en ese momento.

Fíjate también que se ha hecho que se pueda salir del programa incluso en mitad de un fundido, apretando la tecla ESC.

### FADE\_ON()

Esta función equivale exactamente a ejecutar un `FADE(100,100,100,16)`.

Se usa para restaurar la pantalla después de una transición de color como bien puede ser un fundido al negro. La pantalla se restaurará, tras llamar a esta función, en unos 4 frames.

### FADE\_OFF()

Esta función es parecida a ejecutar una orden `FADE(0,0,0,16)`. Sin embargo, a diferencia de la orden `FADE`, que vuelve inmediatamente, esta función espera a que el fundido al negro termine antes de volver.

Mientras dure este fundido al negro especial, todos los procesos estarán detenidos así como cualquier tipo de proceso interno, incluyendo la actualización de variables globales. La función tardará aproximadamente

unos cuatro frames en terminar.

Un ejemplo de esta función y la anterior podría ser éste (necesitarás un gráfico “d.png” de fondo):

```
program Test_FADE_ON;
global
  int idpng;
  int fades=1;
end
begin
  set_mode(640,480,16);
  idpng=load_png("d.png");
  write(0,10,90,3,"1: Fade on:");
  write(0,10,110,3,"2: Fade off:");
  put_screen(0,idpng);
  repeat
    if(key(_1) and fades==2) fade_on(); fades=1; end
    if(key(_2) and fades==1) fade_off(); fades=2; end
  frame;
  until(key(_esc))
end
```

### GET\_RGB( COLOR, &R,&G,&B)

Esta función permite obtener las componentes de un color de 16 bits, obtenido mediante la función RGB o bien leído de los pixels de un gráfico mediante una función como MAP\_GET\_PIXEL .

Las componentes contendrán valores entre 0 y 255. Sin embargo, debido a la diferencia de precisión entre modos gráficos y tarjetas gráficas, no es posible determinar de antemano cual es el valor exacto que se obtendrá. Por ejemplo, incluso un color blanco puro seguramente obtendrá valores como 242 o 248 en sus componentes, en lugar de 255 como sería de esperar. Hay que tener en cuenta que para almacenar los 256 valores posibles serían necesarios 8 bits de precisión. Sin embargo, en un modo de 15 ó 16 bits sólo habrán 5 ó 6 bits de precisión disponibles por cada componente.

El mismo pixel del mismo gráfico puede devolver valores de componentes diferentes en ordenadores distintos. Es importante no comparar los valores devueltos por GET\_RGB con ningún valor constante. Sin embargo, se consideran válidos los siguientes usos con los valores devueltos por GET\_RGB:

- Comprobar que una componente está a cero (== 0). Este valor se considera seguro. Es posible comprobar que un color sea el negro puro observando si sus tres componentes valen cero.
- Comparar componentes de dos valores obtenidos mediante RGB, MAP\_GET\_PIXEL, u otra función que trabaje con colores de 16 bits, durante la misma ejecución del programa.
- Comparar valores de forma aproximada (por ejemplo, comparar si una componente tiene un valor mayor a 128) cuando no sea importante obtener un resultado exacto ni el mismo en todos los ordenadores.

#### PARAMETROS:

INT Color: Un color de 16 bits

POINTER R: Variable de tipo INT que contendrá la componente roja

POINTER G: Variable de tipo INT que contendrá la componente verde

POINTER B: Variable de tipo INT que contendrá la componente azul

Un ejemplo de esta función podría ser:

```
program Test_GET_RGB;
```

```

global
  int Color;
  int r,g,b;
end
begin
  set_mode(320,240,16);
  write(0,10,85,3,"Número de color: ");
  write_var(0,125,85,3,Color);
  write(0,10,95,3,"Valor R: ");
  write_var(0,125,95,3,r);
  write(0,10,105,3,"Valor G: ");
  write_var(0,125,105,3,g);
  write(0,10,115,3,"Valor B: ");
  write_var(0,125,115,3,b);
  timer[0]=200;
  repeat
    if(timer[0]>200)
      /*Aquí se genera automáticamente, pero lo más normal es obtener
      el valor de color a partir de la función map_get_pixel(), o rgb()*/
      Color=rand(1,65535);
      get_rgb(Color,&r,&g,&b);
      timer[0]=0;
    end
  frame;
  until(key(_esc))
end

```

### BLUR(LIBRERIA,GRAFICO, MODO)

Difumina un gráfico según un modo determinado.

El modo puede ser uno de los siguientes:

- 0: Solo se tienen en cuenta los pixels situados a la izquierda y arriba de cada pixel.
- 1: 3x3. Se tienen en cuenta todos los pixels colindantes (8).
- 2: 5x5. Se tienen en cuenta los 24 pixels que rodean cada pixel.
- 3: 5x5 con mapa adicional. Igual que el anterior pero usando un mapa temporal.

El modo define la calidad y la velocidad del difuminado. Un número mayor de modo tiene más calidad, pero requiere mayor tiempo de proceso.

#### PARAMETROS:

INT LIBRERIA: Identificador de la librería cargada con load\_fpg()

INT GRAFICO: Número de gráfico dentro de esa librería

BYTE MODO: Modo de difuminación

Un ejemplo de esta función (se necesita una imagen "d.png"), que creo que no necesita explicación, es el siguiente:

```

program Test_BLUR;
global
  int png;
end
begin

```

```

set_mode(640,480,16);
png=load_png("d.png");
bola(40,110,"T -1",-1);//No se produce difuminado
bola(160,110,"T 0",0);
bola(280,110,"T 1",1);
bola(400,110,"T 2",2);
bola(520,110,"T 3",3);
repeat
  frame;
until(key(_esc))
let_me_alone();
end

```

```

process bola(x,y,string txt,int difum)
begin
  write(0,x,y-20,4,txt);
  graph=map_clone(0,png);
  blur(0,graph,difum);
  loop
    frame;
  end
end

```

### FILTER(LIBRERIA,GRAFICO,&VECTOR)

Aplica un filtro personalizado a un gráfico.

#### PARAMETROS:

INT LIBRERIA: Identificador de la librería cargada con load\_fpg()

INT GRAFICO: Número de gráfico dentro de esa librería

POINTER INT VECTOR: Vector de 10 enteros: los 9 primeros definen una matriz de convolución 3x3 y el décimo el factor de escala (normalmente la suma de las 9 casillas anteriores)

Un ejemplo de esta función (se necesita una imagen "d.png"), que creo que no necesita explicación, es el siguiente:

```

program Test_FILTER;
global
  int png;
  int filter[]=8,20,3,4,5,6,7,8,9,70;
end
begin
  set_mode(640,480,16);
  png=load_png("d.png");
  bola(100,100,"Normal",0);
  bola(400,100,"Filtrado",1);
  repeat
    frame;
  until(key(_esc))
  let_me_alone();
end

process bola(x,y,string txt,int f)
begin

```

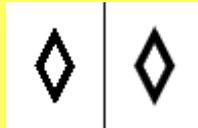
```

write(0,x,y-40,4,txt);
size=200; graph=map_clone(0,png);
if(f==1) filter(0,graph,&filter); end
loop
  frame;
end
end

```

Se pueden probar diferentes valores para la vector y observar los resultados.

Esta función, junto con BLUR(), se pueden utilizar para lograr un efecto de antialiasing en gráficos, fuentes, primitivas.... “Antialiasing” es el nombre que se le da al difuminado de los bordes de una figura, usado para evitar la visualización de feos “escalones”. En el dibujo siguiente, al rombo de la derecha se le ha aplicado algún tipo de antialiasing y al de la izquierda no.



### GRAYSCALE(LIBRERIA,GRAFICO,MODO)

Cambia un gráfico a una escala de colores especificada por el tercer parámetro, el modo.

El modo puede ser uno de los siguientes:

- 0: RGB -escala de grises-
- 1: R -escala de rojos-
- 2: G -escala de verdes-
- 3: B -escala de azules-
- 4: RG
- 5: RB
- 6: BG

Sólo funciona con gráficos de 16 bits.

#### PARAMETROS:

INT LIBRERIA: Identificador de la librería cargada con load\_fpg()

INT GRAFICO: Número de gráfico dentro de esa librería

BYTE MODO: Modo que define la escala de colores a usar

Un ejemplo de esta función (se necesita una imagen “d.png”), que creo que no necesita explicación, es el siguiente:

```

program Test_GRAYSCALE;
global
  int png;
end
begin
  set_mode(640,480,16);
  png=load_png("d.png");
  bola(10,50,"T 0",0);
  bola(110,100,"T 1",1);

```

```

bola(210,150,"T 2",2);
bola(310,200,"T 3",3);
bola(410,250,"T 4",5);
bola(510,300,"T 5",6);
bola(610,350,"T 6",7);
repeat
  frame;
until(key(_esc))
let_me_alone();
end

process bola(x,y,string txt,int modo)
begin
write(0,x,y-20,4,txt);
graph=map_clone(0,png);
grayscale(0,graph,modo);
loop
  frame;
end
end

```

### RGBSCALE(LIBRERIA,GRAFICO,R,G,B)

Cambia un gráfico a una escala de colores especificada en forma de componentes RGB. El valor de R,G y B debe estar comprendido entre 0 y 1.

#### PARAMETROS:

INT LIBRERIA: Identificador de la librería cargada con load\_fpg()  
 INT GRAFICO: Número de gráfico dentro de esa librería  
 FLOAT R: Cantidad de rojo  
 FLOAT G: Cantidad de verde  
 FLOAT B: Cantidad de azul

Un ejemplo de uso de esta función podría ser el siguiente (se necesita un gráfico "d.png"):

```

program Test_RGBSCALE;
global
float r,g,b;
int rr,rg,rb;
float rgb[]=0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0;
int idpng;
end
begin
set_mode(640,480,16);
idpng=load_png("d.png");
x=160; y=120; graph=map_clone(0,idpng); size=200;
write(0,160,70,4,"Valores R G B:");
write_var(0,110,80,4,r);
write_var(0,160,80,4,g);
write_var(0,210,80,4,b);
timer[0]=0;
repeat
  if(timer[0]>100)
    unload_map(0,graph);

```

```

graph=map_clone(0,idpng);
rr=rand(0,10);
rg=rand(0,10);
rb=rand(0,10);
r=rgb[rr];
g=rgb[rg];
b=rgb[rb];
rgbscale(0,graph,r,g,b);
timer[0]=0;
end
frame;
until(key(_esc))
end

```

En este ejemplo se muestra una imagen a la que cada segundo (más o menos) se le cambia su escala de colores de forma aleatoria. Para ello se utiliza un vector con diez valores posibles de las componentes R,G y B. Los que se aplican a la imagen de forma efectiva son seleccionados a partir estos valores utilizando aleatoriamente una posición del vector diferente cada segundo.

La utilidad más evidente de los fades está en implementarlos en los menús iniciales y las introducciones de los juegos, o para mostrar una pantalla de cambio de nivel,etc.

A continuación se presenta un ejemplo donde primeramente aparece un gráfico de fondo (junto con más cosas: un proceso, un texto,etc); al cabo de dos segundos se realiza un fadeoff a negro, eliminando a la vez todo lo demás (los procesos, los textos, el fondo definido); y al cabo de dos segundos se vuelve a hacer un fadein mostrando un gráfico de fondo diferente al original, y sin rastro de procesos,textos,etc.

Necesitarás son gráficos: "a.png" y "b.png":

```

program fades;
global
    int idpng1,idpng2;
    int hora;
    int espera=200;
end
begin
    set_mode(640,480,16);
    idpng1=load_png("a.png");
    idpng2=load_png("b.png");
    put_screen(0,idpng1); //Mostramos el primer fondo
    proceso(); //Ponemos un proceso pululando por ahí para meter más cosas
    write(0,100,100,4,"HOLAHOLAHOLAHOLAHOLA");//Además escribimos un texto cualquiera

    //Permanecemos un tiempo de espera -2 segundos- viendo todo esto
    hora = timer[0]; //Hora guarda la hora actual
    while(timer[0] < hora+espera) frame; end /*Espera es el tiempo (en centésimas de segundo) que
    queremos que el juego se quede parado mostrando el fondo, el proceso y el texto*/

    fade(0,0,0,1); while(fading==1) frame; end //Visualizamos en fadeoff completo
    let_me_alone(); //Sólo si queremos matar al resto de procesos
    delete_text(all_text); //Borramos todos los textos de pantalla
    clear_screen(); //Borramos todos los fondos que hayamos puesto

```

```

//Permanecemos un tiempo de espera con la pantalla en negro
hora = timer[0];
while(timer[0] < hora+espera) frame; end

fade(100,100,100,1); put_screen(0,idpng2); while (fading==1) frame; end /*Visualizamos el fadein
completo con un nuevo fondo definido justo después*/

repeat
    frame;
until(key(_esc))
let_me_alone();
end

process proceso()
begin
    graph=new_map(40,40,16);map_clear(0,graph,rgb(200,100,40));
    x=300;y=200;
        loop
            frame;
        end
end

```

Volver a insistir en la importancia de introducir el bucle `while(fading==1)frame;end` para lograr visualizar el difuminado mientras éste esté activo.

### **Trabajar con puntos de control:**

Los puntos de control son posiciones que puedes definir dentro de un gráfico (puedes definir hasta 999).

Uno de ellos es especial: el punto 0, que define el "centro virtual" de la imagen. Si no defines el punto 0, el centro virtual pasa a ser automáticamente el centro real de la imagen. En ocasiones viene bien redefinir ese punto 0 y ponerlo en otro sitio (por ejemplo entre los pies del personaje si haces un juego de plataformas) ya que la imagen se dibujará poniendo ese punto 0 en la posición dada por las variables locales X Y del proceso.

Otra cosa a tener en cuenta es que se escala y se rota usando de centro ese punto 0.

La ventaja de estos puntos (aparte del 0 y lo comentado) es que usar puntos de control es un modo de definir, por ejemplo, dónde quieres situar un arma en un personaje en todos sus movimientos. Luego sólo tienes que buscar el punto de control desde el código y situar el arma sin tener que meter tú las posiciones en el código a pelo.

Fénix dispone de una serie de funciones que nos facilitan por un lado la creación de puntos de control en los gráficos de nuestro juego, y por otro, nos permite la obtención de las coordenadas de un punto de control concreto de un gráfico en particular, para su posible uso en nuestro código. Estas funciones básicamente son cuatro: SET\_CENTER(),SET\_POINT(),GET\_POINT() y GET\_REAL\_POINT().

#### **SET\_CENTER ( LIBRERIA, GRAFICO, X, Y )**

Esta función permite cambiar el centro de un gráfico.

El centro es un punto dentro de los pixels del gráfico que se utiliza para:

\*Especificar el significado de la coordenada (x, y) cuando el gráfico se visualiza en pantalla. Es decir: si un gráfico se visualiza en la coordenada (10, 15) significa que se dibujará de tal manera que su centro ocupará dicho punto.

\*Especificar el punto a partir de cual se rota el gráfico -por ejemplo, variando el valor de la variable ANGLE del posible proceso asociado a ese gráfico-. El centro no se desplaza, y el resto del gráfico gira a su alrededor.

\*Especifica el punto a partir del cual se realizan las operaciones de espejo horizontal y vertical, por ejemplo mediante variable FLAGS .

Cuando un gráfico se utiliza como puntero del ratón, usando la variable MOUSE.GRAPH, indica el punto "caliente" del mismo, que se usa para comprobar dónde está marcando el ratón. En el ejemplo de Drag&Drop mostrado en el apartado sobre el ratón, en el capítulo 9, se utilizó este hecho.

Cuando un gráfico no tiene centro asignado (por ejemplo, porque se ha recuperado a partir de un fichero estandar como puede ser un PNG), se utiliza como tal su centro geométrico.

Esta función permite cambiar dicho centro, especificando unas nuevas coordenadas para el mismo. Estas coordenadas se dan siempre en pixels dentro del gráfico, donde el pixel (0, 0) representa la esquina superior izquierda del gráfico y las coordenadas crecen abajo y a la derecha.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico dentro de la librería

INT X : Nuevo valor para la coordenada horizontal del centro

INT Y : Nuevo valor para la coordenada vertical del centro

En el siguiente ejemplo se muestra un posible uso de establecer el punto central en otro lugar diferente del centro geométrico de la figura. En este caso, pintamos un cuadrado azul y situamos su centro cerca de uno de sus vértices. El programa consiste en irlo rotando ininterrumpidamente. Se podrá apreciar entonces que el eje de giro no es el centro geométrico sino el punto marcado en blanco, que representa precisamente el punto central (punto de control 0). De similar manera podríamos haber demostrado que ese punto es el eje de espejado cuando se cambia el valor de la variable FLAGS a 1 ó 2.

```
program Test_SET_CENTER;
private
//Coords del punto central (punto de control 0). Se pueden cambiar sus valores para comprobar el efecto
  int pmx=10,pmx=10;
end
begin
  set_mode(640,480,16);
  x=160;
  y=100;
  graph=new_map(50,50,16);
  map_clear(0,graph,rgb(0,0,255));
  map_put_pixel(0,graph,pmx,pmx,rgb(255,255,255)); //Pinto el punto central para que se vea
//Si queremos que el punto central se vea un poco más grande
  //drawing_map(0,graph);draw_box(pmx-2,pmx-2,pmx+2,pmx+2);
  set_center(0,graph,pmx,pmx);
  repeat
    angle=(angle+7000)%360000;
  frame;
  until(key(_esc))
```

end

### SET\_POINT ( LIBRERIA, GRAFICO, NÚMERO, X, Y )

Un gráfico cualquiera puede contener hasta 999 puntos de control de los cuales el punto de control 0 representa el centro del gráfico, y el resto están disponibles para el usuario.

Mediante esta función puede asignarse un punto de control cualquiera del gráfico. Las coordenadas de este punto se dan siempre en pixels dentro del gráfico, donde el pixel (0, 0) representa la esquina superior izquierda del gráfico y las coordenadas crecen abajo y a la derecha.

Si el gráfico no contenía puntos de control, o no contenía puntos de control suficientes, se añadirán los necesarios. Por ejemplo, si un gráfico no dispone de puntos de control y mediante esta función se añade el punto de control 20, los puntos de control 0 a 19 serán rellenados con el valor indefinido (-1, -1).

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG  
INT GRÁFICO : Número de gráfico dentro de la librería  
INT NUMERO : Número de punto de control (0 a 999)  
INT X : Nuevo valor para la coordenada horizontal del punto  
INT Y : Nuevo valor para la coordenada vertical del punto

A continuación se presenta un ejemplo en donde aparece visualizado en pantalla un cuadrado azul, en el cual podremos establecer los puntos de control que deseemos, donde deseemos. Con los cursores podemos cambiar el número de punto de control que queramos introducir (teclas "1" y "2"), y sus coordenadas dentro del cuadrado (cursores del teclado). Aparecerá un punto blanco indicando en todo momento la posición que tendría en cada instante el punto de control. Y éste se crea de forma efectiva pulsando ENTER.

```
program Test_SET_POINT;
global
  int pmx,pmy; //Coordenadas de un punto de control
  int pdc; //Número de punto de control
end
begin
  set_mode(320,240,16);
  graph=new_map(100,100,16);
  x=220;
  y=100;
  map_clear(0,graph,rgb(0,0,255));
  write(0,10,30,3,"Punto N°:");
  write_var(0,85,30,3,pdc);
  write(0,10,40,3,"Coord. X:");
  write_var(0,85,40,3,pmx);
  write(0,10,50,3,"Coord. Y:");
  write_var(0,85,50,3,pmy);
  write(0,10,70,3,"1 > N° Punto");
  write(0,10,80,3,"2 < N° Punto");
  write(0,10,90,3,"UP > Coord. Y");
  write(0,10,100,3,"DOWN < Coord. Y");
  write(0,10,110,3,"LEFT > Coord. X");
  write(0,10,120,3,"RIGHT < Coord. X");
  write(0,10,170,3,"ENTER = Set Point");
```

```

repeat
  if(key(_1) && pdc<999) pdc++; end
  if(key(_2) && pdc>0) pdc--; end
  if(key(_up) && pmy>0) pmy--; end
  if(key(_down) && pmy<99) pmy++; end
  if(key(_left) && pmx>0) pmx--; end
  if(key(_right) && pmx<99) pmx++; end
  if(key(_enter)) set_point(0,graph,pdc,pmx,pmy); end

```

```

  map_clear(0,graph,rgb(0,0,255)); //Para que no salga pintada una línea marcando el recorrido del
  cursor, y solamente aparezca el punto blanco marcando la posición actual
  map_put_pixel(0,graph,pmx,pmy,rgb(255,255,255));
  frame;
until(key(_esc))
end

```

¿Qué pasa si pulsamos ENTER cuando el número del punto de control está establecido en 0? ¿Tienes una explicación?

### GET\_POINT ( LIBRERÍA, GRÁFICO, NÚMERO, &X, &Y )

Un gráfico cualquiera puede contener un número indeterminado de puntos de control de los cuales el punto de control 0 representa el centro del gráfico, y el resto están disponibles para el usuario.

Mediante esta función puede consultarse un punto de control del gráfico. Las coordenadas obtenidas se dan siempre en pixels dentro del gráfico, donde el pixel (0, 0) representa la esquina superior izquierda del gráfico y las coordenadas crecen abajo y a la derecha.

Si el número de punto de control especificado queda fuera de rango o no está definido, la función devuelve 0 y las variables x e y no serán modificadas. En el resto de casos, x e y pasarán a contener las coordenadas del punto de control, de manera que (0, 0) representa el pixel ubicado en la esquina superior izquierda del gráfico, y las coordenadas crecen hacia abajo y a la derecha.

#### PARÁMETROS:

INT LIBRERÍA : Número de librería FPG

INT GRÁFICO : Número de gráfico dentro de la librería

INT NUMERO : Número de punto de control (0 es el centro)

POINTER INT X : Dirección de memoria de una variable de tipo entero que pasará a contener la coordenada horizontal del punto respecto la esquina superior izquierda del propio gráfico.

POINTER INT Y : Dirección de memoria de una variable de tipo entero que pasará a contener la coordenada vertical del punto respecto la esquina superior izquierda del propio gráfico.

VALOR RETORNADO: INT : 1 si el punto está definido, 0 en caso contrario

Un ejemplo bastante sencillo de entender:

```

program Test_GET_POINT;
global
  int gpx,gpy; //Coordenadas de un punto de control
  int pdc; //Número de un punto de control
  int p;
end
begin
  set_mode(320,240,16);

```

```

graph=new_map(100,100,16);
map_clear(0,graph,rgb(0,0,255));
x=160;
y=120;
write(0,10,30,3,"Get Point:");
write_var(0,85,30,3,fdc);
write(0,10,40,3,"Coord. X:");
write_var(0,85,40,3,gpx);
write(0,10,50,3,"Coord. Y:");
write_var(0,85,50,3,gpy);
set_point(0,graph,0,0,0); //Establezco las coordenadas (0,0) del gráfico como las del punto de control 0
/*Establezco de forma aleatoria (pero dentro de las dimensiones del gráfico mostrado) las coordenadas
del resto de 999 puntos de control posibles. Estas coordenadas tienen su origen en la esquina superior
izquierda del propio gráfico. */
from p=1 to 999
  set_point(0,graph,p,rand(0,99),rand(0,99));
end
//Cada décima de segundo obtendré las coordenadas de un punto de control seleccionado al azar.
//Además, visualizaré mediante un punto blanco su situación dentro del gráfico
timer[0]=0;
repeat
  if(timer[0]>100)
    fdc=rand(0,999);
    get_point(0,graph,fdc,&gpx,&gpy);
    map_clear(0,graph,rgb(0,0,255));
    map_put_pixel(0,graph,gpx,gpy,rgb(255,255,255));
    timer[0]=0;
  end
  frame;
until(key(_esc))
end

```

### GET\_REAL\_POINT ( NUMERO, &X, &Y )

Esta función permite obtener la posición *respecto la pantalla* de un punto de control del gráfico contenido por el *proceso actual*, (teniendo en cuenta las variables que indican el aspecto o posición del gráfico, como X, Y o ANGLE) .

El valor de las variables locales X e Y contienen las coordenadas horizontal y vertical respectivamente del punto de control 0 del gráfico del proceso actual. Por tanto, la orden GET\_REAL\_POINT(0,&X,&Y) sería equivalente a consultar directamente el valor de dichas variables.

El segundo y tercer parámetro de esta función pasarán a contener las coordenadas del punto de control del proceso actual, respecto la pantalla.

#### PARÁMETROS:

INT NUMERO : Número de punto de control (0 es el centro)

POINTER INT X : Dirección de memoria de una variable de tipo entero que pasará a contener la coordenada horizontal del punto respecto la esquina superior izquierda de la pantalla.

POINTER INT Y : Dirección de memoria de una variable de tipo entero que pasará a contener la coordenada vertical del punto respecto la esquina superior izquierda de la pantalla.

VALOR RETORNADO: INT : 1 si el punto está definido, 0 en caso contrario

El ejemplo siguiente es el mismo que utilizamos para mostrar el uso de `get_point()`, pero añadiéndole (y por tanto, pudiendo comparar los datos devueltos) una línea que ejecuta `get_real_point()` -y otra mostrando por pantalla lo que esta función devuelve-, pudiendo observar así la diferencia de resultado entre ambas funciones. Básicamente, la diferencia está en que a pesar de devolver datos sobre el mismo punto la primera toma el origen de coordenadas del gráfico que se indica, y la segunda toma el origen de coordenadas de la pantalla, haciendo referencia al gráfico que esté asociado al proceso actual

```
program Test_GET_REAL_POINT;
global
  int rgpx,rgpy; //Coordenadas de un punto de control respecto la pantalla (get_real_point)
  int gpx,gpy; //Coordenadas de un punto de control respecto el gráfico (get_point)
  int pdc; //Número de un punto de control
  int p;
end
begin
  set_mode(320,240,16);
  graph=new_map(100,100,16);
  map_clear(0,graph,rgb(0,0,255));
  x=160;
  y=120;
  write(0,10,30,3,"Get Point:");
  write_var(0,100,30,3,pdc);
  write(0,10,40,3,"Coord. Real X:");
  write_var(0,100,40,3,rgpx);
  write(0,10,50,3,"Coord. Real Y:");
  write_var(0,100,50,3,rgpy);
  write(0,10,60,3,"Coord. X:");
  write_var(0,100,60,3,gpx);
  write(0,10,70,3,"Coord. Y:");
  write_var(0,100,70,3,gpy);
  set_point(0,graph,0,0,0); //Establezco las coordenadas (0,0) del gráfico como las del punto de control 0
  /*Establezco de forma aleatoria (pero dentro de las dimensiones del gráfico mostrado) las coordenadas
  del resto de 999 puntos de control posibles. Estas coordenadas tienen su origen en la esquina superior
  izquierda del propio gráfico. */
  from p=1 to 999
    set_point(0,graph,p,rand(0,99),rand(0,99));
  end
  //Cada décimo de segundo obtendré las coordenadas de un punto de control seleccionado al azar.
  //Además, visualizaré mediante un punto blanco su situación dentro del gráfico
  timer[0]=0;
  repeat
    if(timer[0]>100)
      pdc=rand(0,999);
      get_point(0,graph,pdc,&gpx,&gpy);
      get_real_point(pdc,&rgpx,&rgpy);
      map_clear(0,graph,rgb(0,0,255));
      map_put_pixel(0,graph,gpx,gpy,rgb(255,255,255));
      timer[0]=0;
    end
  frame;
  until(key(_esc))
end
```

Seguramente estarás deseando encontrar alguna aplicación gráfica, sencilla y cómoda que nos permita crear, manipular y consultar puntos de control de un gráfico sin tener que recurrir a programar “a pelo” en Fénix. Pues estás de suerte, porque el FPGEdit permite todo esto y bastante más.

Si abres un archivo FPG cualquiera y seleccionas alguna de las imágenes que están contenidas en él, pulsa en el botón “Edit” de la barra de herramientas de la zona inferior. Verás que aparece un cuadro ya conocido, donde podemos ver cierta información sobre la imagen (las dimensiones, el identificador del gráfico dentro del FPG, etc). Pero no sé si te habías fijado que también hay un botón en ese cuadro, arriba a la izquierda, que pone “Puntos de control”. Si clicas allí verás como el contenido del cuadro cambia y aparecen un montón de opciones referentes a la manipulación de puntos de control.

Lo más básico que podemos hacer es añadir las coordenadas dentro de la imagen del nuevo punto de control que queremos (con las cajas de texto correspondientes, o si quieres, con la ayuda del cuadrado formado por los botones radio) y crearlo pulsando sobre el botón con el icono de la flecha apuntando a la izquierda. Como consecuencia, ese punto aparecerá en la lista de la izquierda. Para eliminar un punto, simplemente hay que seleccionarlo de allí y pulsar sobre el botón con el icono de la flecha apuntando a la derecha. En principio el número del punto de control se asigna automáticamente de forma consecutiva dependiendo de los puntos que ya haya definidos.

Puedes investigar por tu cuenta las otras opciones que aporta el programa: todas son interesantes, y sobre todo, bastante más rápidas que hacerlo uno mismo con código.

Como ejemplo práctico de utilización de los puntos de control, vamos a considerar el caso de una típica barra de energía. Imagínate que estamos programando un juego donde nuestro personaje se enfrenta a enemigos terribles que le atacan constantemente y hacen que su energía disminuya en cada investida. Un caso similar ya nos lo encontramos en el capítulo-tutorial anterior, en el del juego del matamarcianos. En aquel momento se hizo uso de las regiones. Ahora vamos a utilizar otra técnica para obtener un resultado idéntico, con un ejemplo simple.

Dibuja un bonito gráfico rectangular de unos 100x30 píxeles que nos permita visualizar la energía que tiene en cada momento nuestro personaje. La idea es que a medida que la energía disminuya, esta barra se ha de hacer más y más corta.

A continuación presento un código simple, donde se puede observar que tenemos un proceso “personaje” (que resulta ser un cuadrado inmóvil) y un proceso “barra” que es el responsable de mostrar la barra de energía de este personaje. En vez de complicar el ejemplo con procesos enemigos, se ha optado por hacer que nuestro personaje decremente su vida en una unidad cada vez que se pulse la tecla “e”. Así pues, el enemigo de nuestro personaje será esta tecla. Aquí está el programa:

```
Program Energia;  
Global  
  int vidapersonaje=100;  
End  
Begin  
  set_mode(640,480,16);  
  personaje();  
  barra();  
Loop  
  if(key(_esc)) exit();end  
  Frame;  
End  
End
```

```

Process personaje()
Begin
graph=new_map(50,50,16);map_clear(0,graph,rgb(0,255,255));
x=320;y=240;
Loop
  If(key(_e)) vidapersonaje--; end
  Frame;
End
End

process barra()
Begin
graph=new_map(100,30,16);map_clear(0,graph,rgb(0,255,0));
x=320;y=50;
Loop
  IF (vidapersonaje<size_x) size_x--; END
  Frame;
End
End

```

La línea clave está en *IF (vidapersonaje<size\_x) size\_x--; END*. Recuerda que la variable local predefinida *SIZE\_X* establece el tanto por ciento del tamaño del gráfico en su coordenada horizontal, y por defecto vale 100. Bien. Para que esta línea sea útil hemos utilizado dos premisas: 1º en cada ataque enemigo (cada pulsación de “e”) la energía disminuye una unidad; 2º la energía inicial es 100. La primera premisa es importante porque en la línea del if también disminuimos *SIZE\_X* en una unidad: para que el efecto del decrecimiento de la barra esté acorde con la energía real que tiene nuestro personaje, ambas cosas (energía y barra) han de decrecer al mismo ritmo. Y esto tiene mucho que ver también con la segunda premisa: los valores iniciales de la energía y de *SIZE\_X* son iguales, para hacer así que cuando se tenga la mitad de energía (50), la barra tenga la mitad del tamaño original (*SIZE\_X*=50) y así. Es evidente ambas cosas que podrían tener valores iniciales diferentes (y decrementos diferentes también), pero entonces se complicaría la cosa: para empezar, la condición del if se tendría que cambiar, porque si te fijas, tal como está lo que hace es precisamente suponer que “vidapersonaje” y *SIZE\_X* tienen valores que van a la par, y cuando “vidapersonaje” decrece una unidad y se queda por debajo de *SIZE\_X*, entonces esta variable es la que decrece una unidad inmediatamente después.

Bien, si has probado de ejecutar este código, verás que la barra decrece, sí, pero no como te gustaría seguramente. Decrece por los dos lados por igual (derecho e izquierdo), y lo que sería más razonable es que de un lado permaneciera fija y sólo decreciera del otro -normalmente el lado derecho-. ¿Cómo solucionamos esto? Con los puntos de control.

Más exactamente, lo que haremos será modificar la posición del punto de control 0 -es decir, el centro virtual- para que deje de estar en el centro real de la imagen y pase a estar en el extremo izquierdo de la barra (y centrado verticalmente, eso sí). Haciendo esto, conseguiremos que la barra sólo disminuya por el lado contrario, ya que la posición de su centro permanecerá inalterable. ¿Qué tenemos que hacer, pues? Escribe la siguiente línea dentro del proceso “barra”, justo después del LOOP/END:

```
set_center(0,graph,0,15);
```

Fíjate que hemos situado el punto de control 0 del gráfico dado por GRAPH en la coordenada horizontal 0 y la coordenada vertical 15 (30/2, donde 30 es el alto de la imagen). Si ahora pruebas el programa, verás que la barra ya disminuye su valor tal como queríamos (aunque verás que aparece un poco desplazada hacia la derecha, precisamente porque hemos alterado su centro virtual que ha dejado de ser su centro real)

## **Las fuentes FNT. La aplicación “FNTEdit”:**

Algo muy básico en cualquier juego que se precie es mostrar textos: menús, el título, los puntos, el jugador al que le toca, y así un largo etcétera. Y mostrarlos con tipos de letra (las fuentes) atractivos y vistosos.

Fénix no incorpora ningún editor de fuentes, por lo que tendremos que recurrir, si queremos escribir textos espectaculares, a alguna aplicación externa. El problema es que el formato de las fuentes que utiliza Fénix, el formato FNT, es exclusivo de DIV/Fénix y ningún otro programa lo utiliza. Este hecho proviene de la herencia que tiene Fénix de DIV. Por tanto, ya que este formato es tan específico y peculiar, ningún editor de fuentes “normal” que encontremos en el mercado podrá crear las fuentes con el formato usado por Fénix. Así que, ¿cómo creamos las fuentes FNT? Pues los usuarios de Windows tenemos la suerte de disponer de una aplicación, “FNT Edit”, programada por la misma persona que ha creado el “FPG Edit”, que nos soluciona el problema. “FNT Edit” es un fenomenal editor de fuentes FNT, y se puede descargar desde <http://cdiv.sourceforge.net/html/down/down.htm>, o también desde FenixWorld; el enlace actual concretamente es <http://fenixworld.se32.com/download.php?view.20>. Recuerda que también esta utilidad viene incluida dentro del Fénix Pack (<http://fenixpack.blogspot.com>)

El funcionamiento de esta aplicación es muy simple. A la hora de crear o modificar una fuente deberemos cargar de entrada alguna de las fuentes TTF, TTC o FON que tengamos preinstaladas en el sistema (éstos son los formatos de las fuentes que usamos por ejemplo en el Word y aplicaciones más comunes ya que son formatos “estándar”, no como FNT), y a partir de ahí podremos definir su tamaño, el estilo, su color, su sombreado (y su color, y su orientación...), su borde (y su color), incluir imágenes, etc. Cuando ya hayamos acabado, crearemos la fuente dándole al botón “Generar” (saldrá un cuadrado donde dejaremos la opción señalada por defecto: “Calcular paleta optimizada a los colores de la fuente”) y entonces podremos ver en la parte inferior de la ventana una muestra de lo que sería un texto escrito con dicha fuente. Si estamos contentos, guardaremos el estilo de fuente en un archivo \*.FNT. Recuerda que lo que grabamos es la fuente de letra en “abstracto”, no es ningún texto concreto ni nada.

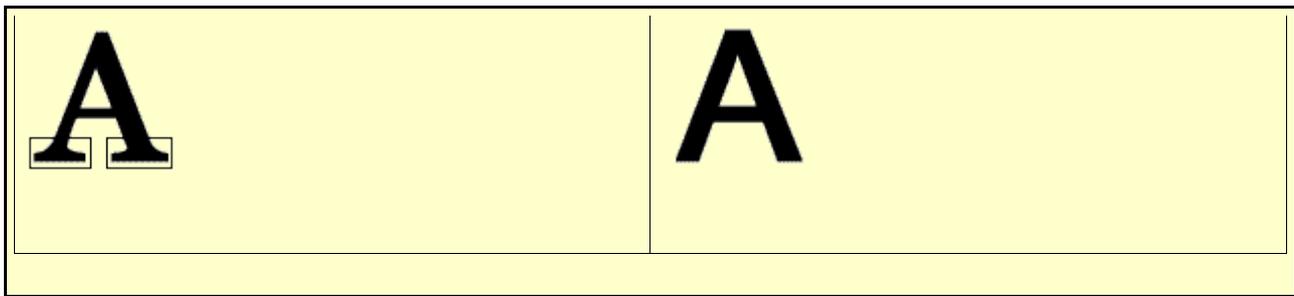
El formato estándar TTF es el formato llamado True Type, y es el más extendido en sistemas Windows y Macintosh, usado en toda clase de aplicaciones. De hecho, este formato fue desarrollado conjuntamente por Microsoft y Apple, y es por eso que está soportado de forma nativa por los sistemas operativos de ambas compañías, haciendo que cualquiera pueda crear documentos utilizando fuentes con dicho formato.

Otro formato de fuentes es el formato TTC, desarrollado por Microsoft y que es un TTF comprimido, y el formato FON, antiguo formato de fuentes usado en las primeras versiones de Windows. Para ver qué fuentes tienes instaladas en tu ordenador simplemente tienes que dirigirte a la carpeta “Fonts” que está dentro de la carpeta “Windows”, la cual estará colgando directamente de la letra de la partición donde hayas instalado Windows (normalmente, C: ).

Una clasificación general que verás frecuentemente si profundizas en el mundo de la tipografía es distinguir entre fuentes “Serif” y fuentes “Sans serif”, independientemente del formato del archivo que tengan. Las fuentes “Sans serif” son las fuentes que no usan serifs, es decir, pequeñas líneas decorativas al final de los caracteres que embellecen su forma básica. Fuentes sans-serif populares son: Helvetica, Arial, Geneva, ... Las fuentes sans-serif son más difíciles de leer, y por eso son usadas frecuentemente en textos cortos como encabezados de línea, títulos o mayúsculas. Las fuentes “Serif” son las fuentes que incorporan serifs, y las más populares son: Times Roman, Courier, New Century, etc.

**Fuente Serif**

**Fuente Sans Serif**



Si quieres conseguir fuentes directamente descargables de Internet, hay muchas web que las ofrecen gratuitamente. A continuación presento una lista, ni mucho menos exhaustiva de sitios donde podremos conseguir gran cantidad de fuentes estándares (pero no FNT, aunque eso no importa porque sabemos que a partir de una estándar podemos crear una fuente FNT con FNTEdit).

<http://fonts.tom7.com>

<http://www.free-fonts.com>

<http://www.goodfonts.org>

[http://thefreesite.com/Free\\_Fonts](http://thefreesite.com/Free_Fonts)

Desde ésta última se puede acceder, entre otras, a:

<http://www.1001freefonts.com>

<http://www.freetype.org>

Y evidentemente, el mejor sitio para encontrar fuentes siempre será

<http://www.google.com>

Y si quieres saber más sobre las fuentes TTF (historia, aplicaciones, documentación, etc), consulta:

<http://www.trueType.demon.co.uk/>

También puedes instalar y utilizar editores de fuentes “estándares” que te ofrezcan más posibilidades de diseño, para posteriormente utilizar el FNTEdit para pasarlas a formato FNT. Algunos programas editores de fuentes que puedes utilizar podrían ser (la lista no es ni mucho menos completa):

**Program FontMagic** (<http://www.mattcawley.com/fontmagic> )

**FontLab** (<http://www.fontlab.com>)

**FontCreator Program** (<http://www.high-logic.com>)

**Fontographer** (<http://www.macromedia.com>)

**Fontforge** (<http://fontforge.sourceforge.net>). Esta aplicación es libre.

Bien, una vez que ya tenemos las fuentes creadas y listas, ¿cómo las utilizamos dentro de nuestro videojuego? Como no podía ser menos, Fénix incorpora una serie de funciones que posibilitan el uso en nuestros programas de fuentes para los textos que aparecen en la pantalla. Las fundamentales serían:

### **LOAD\_FNT(“fichero”)**

Esta función carga en memoria un tipo de letra desde disco.

El FNT es un formato propio de Fénix que permite almacenar tipos de letra de formato "bitmap" (cada carácter dentro de la fuente es un gráfico compuesto de pixels), de 1, 8 o 16 bits por pixel en formato comprimido, con pixels transparentes.

Los tipos de letra tradicionales (TTF, BDF, FON) son a menudo tipos de letra escalables (es decir, en el

mismo fichero se incluye información para dibujar textos a cualquier resolución y tamaño) y en blanco y negro, mientras los tipos de letra en Fenix no son escalables (no es posible dibujar un texto a varios tamaños diferentes) pero a todo color.

LOAD\_FNT devuelve un número entero que identifica al nuevo tipo de letra. Es posible utilizar a partir de entonces este valor con cualquier función que admita un tipo de letra como parámetro, como por ejemplo WRITE

Importante: El formato FNT de 8 bits incluye también una paleta de colores, pero sólo puede haber una paleta de colores activa en todo momento, por lo que todo gráfico de 8 bits existente debe compartir la misma paleta de colores. Fenix usará automáticamente la paleta de colores del primer gráfico en recuperar de disco e ignorará la paleta de los demás. Esto quiere decir que si se cargan varios ficheros FNT de 8 bits con la función LOAD\_FNT uno detrás de otro, a pesar de que cada fuente tenga su paleta propia, sólo se utilizará la paleta del primer fichero FNT, con lo que los colores de los siguientes tipos de fuente no serán los mismos que los originales (asociados a cada paleta particular). Es por eso que se recomienda, si se van a usar fuentes de 8 bits, utilizar una paleta común para todas ellas para evitar sorpresas. Otro truco sería crear las fuentes de 1 bit y utilizar luego la función set\_text\_color().

PARÁMETROS: STRING FICHERO: Nombre del fichero

VALOR RETORNADO: INT : Identificador del nuevo tipo de letra

### UNLOAD\_FNT(FUENTE)

Libera la memoria ocupada por un tipo de letra recuperado de disco por la función LOAD\_FNT.

Se considera un error utilizar a partir de ese momento ese identificador de fuente en cualquier llamada a función, así como mantener en pantalla textos de esta fuente escritos con la función WRTE u otra equivalente.

PARÁMETROS: INT FUENTE : Identificador del tipo de letra

Es decir, con sólo utilizar LOAD\_FNT, ya dispondremos de un identificador de fuente, al igual que con LOAD\_PNG disponíamos de un identificador de Png. Y este identificador de fuente lo podremos utilizar como valor del primer parámetro de las funciones write o write\_var, las cuales hasta ahora habíamos usado poniendo en dicho parámetro el valor de 0 -la fuente del sistema-. Y ya está: así de sencillo. Es decir, que si se supone que tenemos creada una fuente llamada "a.fnt", para utilizarla en un texto de nuestro programa, tendríamos que hacer lo siguiente:

```
PROGRAM fonte;
private
  int idfuente;
end
begin
  set_mode(640,480,16);
  idfuente=load_fnt("a.fnt");
  write(idfuente,320,240,4,"Hola!");
loop
  frame;
  if(key(_esc)) break;end
end
unload_fnt(idfuente);
end
```

Fíjate que gracias a los tipos de fuentes, ahora podremos escribir textos en cualquier tamaño, por ejemplo. Una cosa que hasta ahora no podíamos hacer.

A parte de las funciones de carga y descarga de fuentes, Fénix también dispone de unas cuantas funciones más relacionadas con la tipografía, que aunque no son tan comunes, al menos son interesantes de conocer, sobre todo porque ofrecen la posibilidad de crear nuestras propias fuentes FNT sin la necesidad de utilizar el FNTEdit. Estas funciones son cuatro:

#### **FNT\_NEW(PROFUNDIDAD)**

Esta función crea un tipo de letra FNT.

El nuevo tipo de letra estará preparado para contener caracteres de la profundidad de color deseada, pero en un principio estará vacío. Será necesario utilizar la función SET\_GLYPH para añadir gráficos de caracteres al tipo de letra.

PARÁMETROS: INT PROFUNDIDAD : Profundidad de color (1,8 ó 16)

VALOR RETORNADO: INT : Identificador del nuevo tipo de letra

#### **GET\_GLYPH(FUENTE,CARACTER)**

Dado un tipo de letra, esta función crea un gráfico que contiene uno de sus caracteres.

Esta función ha sido diseñada para poder modificar este gráfico en memoria según las necesidades de cada uno y guardar luego los cambios en la propia fuente, empleando la función SET\_GLYPH .

El gráfico creado pertenecerá a la librería 0 y será una copia de la información contenida en la fuente. Esto significa que las modificaciones realizadas sobre él no afectarán directamente a la fuente: será necesario llamar a la función SET\_GLYPH para que los cambios sean realmente efectivos.

*El gráfico creado es una copia fiel del contenido en el tipo de letra, y se incluyen dos puntos de control (el 1 y el 2) para almacenar información adicional del carácter:*

- *El punto 1 contendrá el desplazamiento a efectuar sobre el punto de escritura en el momento de escribir el carácter. Normalmente ocupará la esquina superior izquierda (0, 0) indicando que no hay ningún desplazamiento.*
- *El punto 2 contendrá el avance efectivo de posición, horizontal y vertical, que se realizará al escribir el carácter. El avance vertical no se emplea actualmente.*

PARÁMETROS:

INT FUENTE : Identificador del tipo de letra

INT CARACTER: Número de carácter de la tabla ASCII (de 0 a 255)

VALOR RETORNADO: INT: Número de gráfico generado a partir del carácter de la fuente especificado

Un ejemplo de uso:

```

program prueba;
begin
set_mode(320,240,16);
graph=get_glyph(0,asc("Z"));
size=1000;x=160;y=120;flags=5;angle=90000;

while(!key(_esc)) frame; end
end

```

### SET\_GLYPH(FUENTE,CARACTER,LIBRERIA,GRAFICO)

Dado un tipo de letra, esta función permite cambiar cualquiera de sus caracteres por el gráfico especificado entre sus tercer y cuarto parámetros (que puede ser un gráfico obtenido previamente con GET\_GLYPH y modificado posteriormente).

Esto cambia efectivamente el aspecto de la fuente en memoria, y el nuevo carácter será reflejado en posteriores usos de la fuente, incluyendo cualquier texto escrito por la función WRITE u otra equivalente, que pudiera estar en pantalla en ese momento.

*Es posible incluir dos puntos de control con información adicional sobre el carácter:*

- *El punto 1 contendrá el desplazamiento a efectuar sobre el punto de escritura en el momento de escribir el carácter. Normalmente ocupará la esquina superior izquierda (0, 0) indicando que no hay ningún desplazamiento.*
- *El punto 2 contendrá el avance efectivo de posición, horizontal y vertical, que se realizará al escribir el carácter. El avance vertical no se emplea actualmente.*

*Si estos puntos de control no se incluyen, la información del carácter existente no será modificada. Normalmente esto no es deseable, así que se recomienda añadir siempre los puntos de control al gráfico mediante la función SET\_POINT*

#### PARÁMETROS:

INT FUENTE : Identificador del tipo de letra  
 INT CARACTER: Número de carácter de la tabla ASCII (de 0 a 255)  
 INT LIBRERIA: Identificador de librería FPG  
 INT GRAFICO: Número de gráfico dentro de la librería FPG

### SAVE\_FNT(FUENTE,"fichero")

Crea o sobrescribe un fichero en disco, con el nombre especificado, que contenga un tipo de letra FNT actualmente ubicado en memoria.

Este tipo de letra habrá sido recuperado de disco previamente por la función LOAD\_FNT u otra equivalente.

Se guardará cualquier cambio efectuado en memoria al tipo de letra con funciones como SET\_GLYPH . Además, el fichero se guardará en formato comprimido.

#### PARÁMETROS:

INT FUENTE : Identificador del tipo de letra  
 STRING FICHERO : Nombre del fichero

Vamos a utilizar los comandos acabados de ver para hacer un primer ejercicio de creación de fuentes FNT propias a través de código Fénix (original de SplinterGU), sin utilizar el FNTEdit u otras aplicaciones externas. Para empezar a hacer alguna prueba, lo que haremos será crearnos una fuente que contenga sólo dos símbolos, correspondientes a lo que serían los caracteres “A” y “B”, pero que ambos serán pintados como un cuadrado gris. Es decir, haremos una fuente con la que sólo se puedan escribir la “A” y la “B”, pero que además, en vez de escribir estos caracteres, se pintarán sendos cuadrados grises.

```

program GeneradorFuenteFNT;
private
  int font,map;
end
begin
  set_mode(800,600,16);
  map=new_map(10,10,16);
  font=new_fnt(16);
  //Dibujo un gráfico (un cuadrado gris)
  drawing_map(0,map);drawing_color(rgb(100,100,100));draw_box(0,0,9,9);
  /*Este gráfico corresponderá al carácter 65 de la nueva fuente (El carácter 65 de la tabla ASCII
  corresponde a la "A"). Como sólo se asigna este carácter, el resto de letras y símbolos no serán visibles
  ya que no estarán definidos para esta fuente: habría que hacer 255 set_glyph, una por cada carácter de la
  nueva fuente creada*/
  set_glyph(font,65,0,map);
  /*Sólo imprimirá un cuadrado gris correspondiente a "A". "B" no se escribirá porque no tiene asociado
  ningún símbolo de la fuente.*/
  write(font,400,300,0, "AB");
  //Ahora hago que "B" también se escriba como el mismo cuadrado gris
  set_glyph(font,66,0,map);
  //Ahora se imprimen dos cuadrados grises
  write(font,450,300,0, "AB");
  /*Guardo la fuente creada (que está en memoria), en el disco, para poderla utilizar posteriormente en
  otros programas como cualquier otra fuente. Ojo, el formato que utiliza save_fnt no es compatible con
  FNTEdit, así que no podrás manipularla dentro de esta aplicación*/
  save_fnt(font,"mifuente.fnt");
  while(!key(_esc))frame; end
end

```

Supongo que pensarás que el ejemplo de arriba no es de mucha utilidad práctica. Vamos ahora a crearnos una fuente un poco más usable, siguiendo la misma filosofía. Se escribirá primero un carácter utilizando la fuente del sistema; seguidamente, se modificará éste con *write\_in\_map()* y efectos propios de gráficos; finalmente, este carácter modificado se guardará en una nueva fuente y ésta se utilizará para volver a escribir el mismo carácter, comprobando así su nuevo aspecto.

```

program prueba;
private
  int font,graf;
end
begin
  set_mode(320,240,16);

  write(0,150,50,4,"Letra 'a' con la fuente del sistema : a");

  /*Transformo el carácter pasado por parámetro a una imagen de tamaño dado por el segundo parámetro

```

```

y con un flag dado por el tercer parámetro (además, le aplico difuminado). El valor devuelto es el código
identificador de esta imagen*/
graf=char2grafchar("a",300,1);

font=new_fnt(16);
/*Hago que el caracter correspondiente al símbolo "a" de la fuente "font" sea representado por la imagen
acabada de generar en la línea anterior*/
set_glyph(font,asc("a"),0,graf);
//Muestro el resultado
write(0,150,100,4, "Letra 'a' con la fuente acabada de generar : ");
write(font,300,100,4, "a");
/*Y creo otro archivo en disco con esta nueva fuente (que consta únicamente de un caracter por ahora),
para poderla utilizar posteriormente en otros programas de forma normal*/
save_fnt(font,"mifuentes2.fnt");

while(!key(_esc))
frame;
end
END

/*Función que transforma un carácter pasado por parámetro en una imagen, de un tamaño y flags dados,
y con difuminado*/

function char2grafchar(string letra,int nuevotam, int nuevoflag)
private
    int blend,graf,graf2;
    int alto,ancho;
end
begin
//Convierto el carácter pasado por parámetro en una imagen tal cual.
graf=write_in_map(0,letra,4);

/*Las líneas siguientes sirven para cambiar el aspecto de la imagen generada por write_in_map. No
puedo utilizar variables como SIZE,FLAGS,etc porque estoy dentro de una función y ahí éstas variables no
tienen sentido (y estoy en una función y no un proceso porque necesito devolver el identificador del gráfico
modificado con return). Tampoco existe una función específica que cambie las propiedades de un gráfico
como su tamaño o su flags si éste no pertenece a un proceso. Pero lo que sí puedo utilizar para modificar
el tamaño, flags,etc de un gráfico es función map_xput. Esta función "pega" en un gráfico destino una
copia de un gráfico origen, opcionalmente modificado en su tamaño y flags, entre otras cosas. Así que lo
que haremos será crear un gráfico destino vacío (llamado graf2), donde "pegaremos" el gráfico graf y lo
haremos cambiando su tamaño y su flag. De esta manera, obtendremos en graf2 el gráfico modificado del
carácter, y éste gráfico será el que la función retornará*/

    ancho=graphic_info(0,graf,g_width);
    alto=graphic_info(0,graf,g_height);
/*Creamos el gráfico vacío cuyo tamaño será el que tiene el gráfico original (graf) variando su tamaño
según el que se ha indicado como segundo parámetro. Éste parámetro está escrito en las mismas unidades
que la variable SIZE (es decir, en tanto por ciento), por lo que si se quiere modificar el tamaño en píxeles,
habrá que dividir por 100. Es decir, en este caso, si el 2º parámetro vale 300, el tamaño de graf2 será el
triple, por lo que tendremos que multiplicar el alto y el ancho del gráfico original por 3 (300/100).*/
    graf2=new_map(ancho*nuevotam/100,alto*nuevotam/100,16);
/*La línea clave: copiamos graf en graf2 en la coordenada central de éste (que se calcula simplemente
dividiendo por dos el ancho y alto de graf2 (ancho*nuevotam/100 y alto*nuevotam/100 respectivamente),
y ADEMÁS, lo copiamos con un nuevo tamaño y con un nuevo flag. También podríamos haberlo copiado

```

con un nuevo angle, pero no lo hemos hecho...Después de esta línea ya tenemos en graf2 el gráfico modificado del caracter original que queríamos\*/

```
map_xput(0,graf2,graf,ancho*nuevotam/200,alto*nuevotam/200,0,nuevotam,nuevoflag);
```

//Aplico también un difuminado al gráfico

```
blur(0,graf2,3);
```

/\*Las siguientes líneas comentadas sirven para aplicar a graf2 un efecto de tintado en verde y de transparencia. Para ello utilizan las llamadas tablas blendop, tema que requiere un poco más de estudio y que se explicará detalladamente en el último capítulo de este manual. No obstante, si se desea observar el efecto que producen estas líneas, se pueden descomentar y volver a ejecutar el programa\*/

```
/*blend=blendop_new();  
blendop_tint(blend,1.0,145,245,45);  
blendop_translucency(blend,0.4);  
blendop_assign(0,graf2,blend);*/
```

//Devuelvo el código del gráfico transformado

```
return graf2;
```

end

No obstante, lo más normal no es hacer lo que hemos hecho en el ejemplo anterior (crear los nuevos caracteres de la fuente “al vuelo” con *write\_in\_map*) sino que normalmente los caracteres se diseñan con un programa específico: un editor de fuentes o incluso un editor de bitmaps o vectoriales. Y lo único que hacemos para generar la fuente FNT con nuestro programa escrito en Fénix es invocar a *set\_glyph* con cada uno de los diseños de esos caracteres previamente dibujados en el programa externo.

Como último ejercicio: ¿qué hace este código?

```
PROGRAM Orient;
```

```
private
```

```
int idmif fuente;
```

```
int graf;
```

```
int i;
```

```
End
```

```
BEGIN
```

```
set_mode(640,480,16);
```

```
idmif fuente=fnt_new(16);
```

```
for(i=1;i<256;i++)
```

```
graf=get_glyph(0,i);
```

```
//...Modifico graf como quiera
```

```
set_glyph(idmif fuente,i,0,graf);
```

```
end
```

```
write(idmif fuente,320,240,4,"Hola!");
```

```
loop
```

```
frame;
```

```
if(key(_esc)) break;end
```

```
end
```

```
save_fnt(idmif fuente,"lala.fnt"); //Da un error de "gr_font_save: fuente corrupta" (?)
```

```
unload_fnt(idmif fuente);
```

```
end
```

## **Trabajar con regiones:**

Una “región de pantalla” no es más que una “ventana” invisible dentro de la ventana del juego, una zona invisible de la pantalla que puede contener una serie de gráficos, o no.

De hecho, de forma predefinida, toda la ventana del juego es ya una región, la denominada región cero. Fenix nos permite definir hasta 31 regiones, de la 1 a la 31, del tamaño que nos de la gana, siempre que no se salgan de la zona de pantalla; la región cero no se puede definir por razones obvias: ya está definida.

Sus utilidades son muchas, por ejemplo, para saber si un gráfico está fuera o dentro de una zona de la pantalla, para mover el fondo, para definir ventanas para multijugador a pantalla partida, etc, pero la que más utilidad le vamos a dar es que podemos meter un gráfico dentro de una región, de tal manera que si éste se sale de ella, desaparece como si fuera el borde de la pantalla.

Sus utilidades son muchas, especialmente en juegos en los que no toda la zona visible corresponde al área de juego (como por ejemplo se usan zonas de pantalla destinadas al uso de marcadores). Además, las regiones también se pueden usar para:

- Definir el área ocupada por una región de scroll (en seguida veremos qué son y cómo funcionan).
- Definir un área de "clipping", de manera que los procesos que estén marcados como pertenecientes a la misma sean recortados por sus límites. Y en relación a esto, definir que si un gráfico sale de una región, desaparezca como si fuera el borde de pantalla, o viceversa
- Definir una zona de pantalla donde nos interesará saber fácilmente cuándo un proceso entra o sale.
- Definir ventanas multijugador a pantalla partida.
- Etc

Para poder trabajar con zonas que no sean la 0, necesitaremos crearlas antes. Y para ello utilizaremos la función DEFINE\_REGION.

### **DEFINE\_REGION(REGION,X,Y,ANCHO,ALTO)**

Esta función permite definir los límites de una región. Existen 32 regiones disponibles, de las cuales la región 0 corresponde siempre a la pantalla completa y no puede cambiarse.

Las coordenadas se dan siempre en pixels, donde (0, 0) corresponde a la esquina superior izquierda de pantalla. Sin embargo, por sí mismo definir una región no tiene ningún efecto hasta que la región se utiliza.

Para ello, pronto veremos que puedes asignar el número de región a la variable REGION de un proceso, usarla con la función START\_SCROLL, o emplear también la función OUT\_REGION con la nueva región definida.

#### **PARÁMETROS:**

INT REGION: Número de región, de 1 a 31

INT X : Coordenada horizontal de la esquina superior izquierda de la región

INT Y : Coordenada vertical de la esquina superior izquierda de la región

INT ANCHO : Ancho en píxeles de la región

INT ALTO : Alto en píxeles de la región

Tal como acabamos de decir en el cuadro anterior, definir una región no tiene ningún efecto

aparente sobre nuestro juego. Falta un paso más: utilizarla. Y hay varias maneras para eso. En el siguiente apartado ya hablaremos de los scrolls y de la aplicación que tienen las regiones a éstos, pero si no trabajamos con scrolls, el uso de las regiones suele ser el siguiente: normalmente se usan para establecer que uno o más procesos determinados sólo serán visibles dentro de una región en concreto. Y para lograr esto, lo que hay que hacer es asignar a la variable local **REGION** de ese/esos proceso/s el número de región definido previamente con DEFINE\_REGION donde queremos que el proceso sea visible. Las partes del gráfico del proceso que queden fuera de la zona definida por la región se recortarán y no serán mostradas. El valor por defecto de REGION para todos los procesos es 0, que equivale a la pantalla completa.

A continuación veremos un ejemplo donde se puede comprobar el efecto del uso de la variable REGION. El código consiste en la modificación mediante los cursores del teclado tanto del origen de la esquina superior izquierda como de la altura y anchura de una determinada región numerada como 1.

La manera de visualizar este cambio es asociar dicha región 1 en este caso al código principal del programa -podría haber sido un proceso cualquiera-. Como acabamos de decir que al asociar un proceso a una región, estamos haciendo que el gráfico de ese proceso sólo sea visible dentro de esa región, si modificamos la posición y tamaño de la región, modificaremos la porción de gráfico visible en cada momento, que es lo que tendría que ocurrir si lo pruebas (necesitarás un gráfico llamado "a.png").

```

program Test_DEFINE_REGION;
global
  int xr=0,yr=0; //Esquina superior izquierda de la región
  int wr,hr; // Anchura y altura de la región
end
begin
  set_mode(640,480,16);
  graph=load_png("a.png");
  x=320;
  y=240;
  region=1; //Sólo se verá el gráfico del código dentro de la región 1
  write(0,10,50,3,"LEFT : Disminuye anchura región");
  write(0,10,60,3,"RIGHT: Aumenta anchura región");
  write(0,10,70,3,"UP : Disminuye altura región");
  write(0,10,80,3,"RIGHT: Aumenta altura región");
  write(0,10,100,3,"Anchura región: ");
  write_var(0,120,100,3,wr);
  write(0,10,110,3,"Altura región: ");
  write_var(0,120,110,3,hr);
  write(0,10,120,3,"X región: ");
  write_var(0,120,120,3,xr);
  write(0,10,130,3,"Y región: ");
  write_var(0,120,130,3,yr);
  repeat
    if(key(_left) && xr>0) xr=xr-5; end
    if(key(_right) && xr<320) xr=xr+5; end
    if(key(_up) && yr>0) yr=yr-5; end
    if(key(_down) && yr<240) yr=yr+5; end
  /*Modifico la anchura y altura de la región acorde a la posición de su esquina superior izquierda. Esta
  fórmula se podría cambiar por otra */
  wr=640-(xr*2);
  hr=480-(yr*2);
  define_region(1,xr,yr,wr,hr); //Defino la región 1
  frame;
  until(key(_esc))

```

end

Finalmente, otra función importante respecto el tema de las regiones es OUT\_REGION.

### OUT\_REGION(ID,REGION)

Esta función devuelve 1 si un proceso cuyo identificador es su primer parámetro está fuera de la región indicada como segundo parámetro. En otras palabras, permite comprobar si un proceso ha sido dibujado dentro o fuera de una región en pantalla.

La función devuelve 1 si el proceso se encuentra completamente fuera de los límites de la región, y 0 si alguna parte o todo el proceso toca dentro de la misma.

Antes de llamar a esta función, es preciso haber establecido los límites de la región mediante DEFINE\_REGION, a no ser que se utilice la región 0, que corresponde siempre a toda la pantalla y ofrece una forma sencilla de comprobar si un proceso ha salido fuera de pantalla.

Si quieres comprobar si el proceso actual ha salido fuera de una región, recuerda que se puede utilizar la variable local ID para obtener su identificador.

#### PARÁMETROS:

INT ID: Identificador de un proceso

INT REGION : Número de región, de 0 a 31.

Un ejemplo de la función anterior podría ser el siguiente. En él tenemos cuatro procesos, cuyos identificadores se guardan cada uno en un elemento de la matriz idb[]. Además, tenemos definida una región (región 1), visualizada gracias a un cuadrado de diferente color. Los gráficos de dos de los procesos estarán entrando y saliendo continuamente de dicha región, y los gráficos de los otros dos procesos estarán entrando y saliendo continuamente de la pantalla (región 0).

El programa muestra cuatro textos que se actualizan constantemente y muestran el estado de cada uno de los cuatro procesos respecto si están fuera o dentro de la región 1 (dos procesos) o de la región 0 (los otros dos). Y esto se sabe gracias a lo que devuelve OUT\_REGION. El estado -dentro o fuera- de cada elemento de idb[] se almacena en un elemento de la matriz out[[]].

```
program Test_OUT_REGION;
```

```
global
```

```
int idb[3];
```

```
int out[3];
```

```
end
```

```
begin
```

```
set_mode(320,240,16);
```

```
/*Pinto un rectángulo visible con las mismas dimensiones y posición que la región 1 que creo a continuación, para hacer ésta visualizable.*/
```

```
graph=new_map(120,100,16);map_clear(0,graph,rgb(80,20,20));
```

```
x=220;y=75;z=1;
```

```
/*Creo la región de forma real. Notar que el ancho y alto es igual al del rectángulo pintado en la línea anterior, y la posición de su esquina superior izquierda (160,25) viene dada por un simple cálculo a partir del centro del rectángulo pintado (220,75) y su tamaño (120,100).*/
```

```
define_region(1,160,25,120,100);
```

```
write(0,10,60,3,"Out Region 0 Bola 1:");
```

```
write_var(0,147,60,3,out[0]);
```

```
write(0,10,70,3,"Out Region 0 Bola 2: ");
```

```

write_var(0,147,70,3,out[1]);
write(0,10,80,3,"Out Region 1 Bola 3: ");
write_var(0,147,80,3,out[2]);
write(0,10,90,3,"Out Region 1 Bola 4: ");
write_var(0,147,90,3,out[3]);

idb[0]=bola(300,160,100,0,1,1);
idb[1]=bola(300,160,100,0,2,1);
idb[2]=bola(270,105,50,1,3,1);
idb[3]=bola(270,105,50,1,4,1);

/*Out[0] valdrà 1 si el proceso idb[0] sale fuera de la pantalla.
Out[1] valdrà 1 si el proceso idb[1] sale fuera de la pantalla.
Out[2] valdrà 1 si el proceso idb[2] sale fuera de la región 1.
Out[3] valdrà 1 si el proceso idb[3] sale fuera de la región 1.*/
repeat
  out[0]=out_region(idb[0],0);
  out[1]=out_region(idb[1],0);
  out[2]=out_region(idb[2],1);
  out[3]=out_region(idb[3],1);
frame;
until(key(_esc))
let_me_alone();
end

/*Al pasar por parámetro la variable local region, estamos indicando a la hora de crear el proceso, en
qué región queremos que éste sea visible*/
process Bola(x,y,size,region,t,m)
begin
  graph=new_map(8,8,16);map_clear(0,graph,rgb(100,100,100));
  loop
  /*Las variables "t" y "m" simplemente son para controlar el movimiento de los diferentes procesos. No
  tienen mayor importancia.*/
  switch(t)
  case 1:
    if(x>-50 && m==1) x=x-5; else m=2; end
    if(x<370 && m==2) x=x+5; else m=1; end
  end
  case 2:
    if(y>-50 && m==1) y=y-5; else m=2; end
    if(y<250 && m==2) y=y+5; else m=1; end
  end
  case 3:
    if(x>130 && m==1) x=x-2; else m=2; end
    if(x<310 && m==2) x=x+2; else m=1; end
  end
  case 4:
    if(y>0 && m==1) y=y-2; else m=2; end
    if(y<150 && m==2) y=y+2; else m=1; end
  end
  end
frame;
end
end

```

## **Trabajar con scrolls y Modo7:**

Un scroll es un fondo más grande que la pantalla, y que se mueve. Los scrolls permiten mostrar en pantalla una zona de juego más grande que ésta, de manera que sólo es posible ver una parte del total.

Podemos usar los “scrolls” para muchas cosas: para dibujar por ejemplo un paisaje de fondo que sigue a nuestro héroe en un juego de plataformas, para hacer un mundo que se mueve bajo los pies de nuestro protagonista en un juego de rol, para hacer planetas y nieblas que van apareciendo en el universo cuando nuestra nave lo va surcando de punta a punta....

Fénix nos permite usar hasta 10 scrolls a la vez. Esto puede ser útil para mostrar dos zonas de scroll distintas, pero el uso más común es mostrar dos ventanas de movimiento dentro de la misma zona (por ejemplo para partidas a dos jugadores).

Cada uno de estos scrolls dispone además de no uno, sino dos fondos a distinta profundidad que podremos mover a distintas velocidades. El fondo de segundo plano se verá a través de los píxeles transparentes del gráfico del fondo de primer plano.

Podemos clasificar los scrolls en dos tipos básicos: los scrolls automáticos, y los scrolls que persiguen a un proceso. Antes de comenzar a codificar debemos tener claro qué tipo de scroll queremos implementar.

Un scroll automático es aquél que, independientemente de los movimientos o posiciones de los procesos en pantalla, va a ir desplazando el fondo de manera automática en una o unas direcciones determinadas, a modo de fondo siempre animado. Este movimiento de scroll lo tendremos que programar “a mano” para hacer que se mueva de esta forma “mecánica” tal como queramos.

El otro tipo de scroll, el que persigue a un proceso, consiste básicamente en hacer que el fondo se desplace junto con el movimiento de un determinado proceso: así, si el proceso está quieto, el fondo está quieto, y cuando el proceso se mueve en una dirección, el scroll se moverá en dicha dirección. Éste es el típico efecto que se utiliza en los juegos de rol a vista de pájaro (también llamados RPG, veremos un tutorial enseguida) cuando se quiere mostrar el desplazamiento de nuestro protagonista a través de un territorio.

Para iniciar un scroll (del tipo que sea) hay que usar la función *start\_scroll*, en sustitución de *put\_screen*, (si es que estábamos usándola hasta ahora). Es decir, *start\_scroll* hace básicamente lo mismo que *put\_screen* : pinta una imagen de fondo (bueno, en realidad dos imágenes para dos fondos a distinta profundidad, ahora lo veremos), con la gran diferencia que esta/s imagen/es de fondo son capaces de moverse. *Put\_screen* pinta un fondo fijo, *Start\_scroll* pinta un fondo móvil.

### **START\_SCROLL(NUMERO,FICHERO,GRAFICO,FONDO,REGION,FLAGS)**

#### **PARAMETROS:**

INT NUMERO : Número identificador de scroll (0 a 9)

INT FICHERO : Número de fichero FPG que contiene los gráficos GRAFICO y FONDO (o 0 si los gráficos de scroll se han cargado individualmente, por ejemplo con *load\_png*)

INT GRAFICO : Número de gráfico para el primer plano del scroll. Es obligatorio especificar un gráfico válido.

INT FONDO : Número de gráfico para el fondo del scroll, 0 para ninguno.

INT REGION : Número de región donde el scroll funcionará. Esto es útil si divides la pantalla y quieres que cada parte sea un scroll para un jugador distinto, o para hacer un pequeño mapa scrollable a modo de zona pequeña dentro de la ventana principal, por ejemplo. Si vas a usar toda la pantalla, ya sabes que el número que tienes que indicar es el cero.

INT FLAGS : Si este parámetro vale 0, el scroll se moverá hasta que el margen del gráfico del primer plano del fondo –el 2º parámetro- coincida con uno de los límites de la región/pantalla, momento en el que el scroll se parará, ya que se ha llegado al final de la imagen. Hay que tener en cuenta que este valor de 0 en este parámetro sólo tiene sentido en el caso de los scrolls “perseguidores” de procesos, porque con los scrolls automáticos precisamente lo que interesa es evitar esto, es decir, lo que interesa es conseguir que el scroll se repita a modo de mosaico indefinidamente de forma cíclica. Para ello, este parámetro puede valer cualquiera de los siguientes valores:

1	El gráfico de primer plano se repite horizontalmente.
2	El gráfico de primer plano se repite verticalmente.
3 (2+1)	El gráfico de primer plano se repite tanto horizontal como verticalmente.
4	El gráfico de fondo se repite horizontalmente.
5 (4+1)	El gráfico de fondo se repite horizontalmente y el de primer plano también.
6 (4+2)	El gráfico de fondo se repite horizontalmente y el de primer plano verticalmente
7 (4+2+1)	El gráfico de fondo se repite horizontalmente y el de primer plano en ambas direcciones
8	El gráfico de fondo se repite verticalmente.
9 (8+1)	El gráfico de fondo se repite verticalmente y el de primer plano horizontalmente
10 (8+2)	El gráfico de fondo se repite verticalmente y el de primer plano también.
11 (8+2+1)	El gráfico de fondo se repite verticalmente y el de primer plano en ambas direcciones
12 (8+4)	El gráfico de fondo se repite tanto horizontal como verticalmente.
13 (8+4+1)	El gráfico de fondo se repite en ambas direcciones y el de primer plano horizontalmente.
14 (8+4+2)	El gráfico de fondo se repite en ambas direcciones y el de primer plano verticalmente.
15 (8+4+2+1)	El gráfico de fondo se repite en ambas direcciones y el de primer plano también.

Una vez iniciado el scroll, no hay que olvidarse de cerrarlo antes de acabar la ejecución del programa para que ésta se realice limpiamente, usando *stop\_scroll* pasándole el identificador de scroll que queremos parar.

### STOP\_SCROLL (NUMERO)

Detiene un scroll activado previamente mediante la función *START\_SCROLL* . Debe indicarse el mismo identificador de scroll especificado al inicializarla. Ese scroll no será visible a partir del próximo frame (como si hubiéramos hecho un *clear\_screen* para un *put\_screen*).

#### PARAMETROS:

INT NUMERO : Número identificador de scroll (0 a 9)

Vamos a ir conociendo cómo funciona el mecanismo de scroll a partir de un ejemplo que iremos ampliando a medida que vayamos aprendiendo más aspectos del mismo.

Para este ejemplo y todos los demás de este apartado, necesitarás tener dos imágenes que serán el fondo de primer plano y de segundo plano de nuestro scroll. Como el ejemplo lo haremos en una resolución de 640x480, estas imágenes han de ser más grandes que estas dimensiones para que se note algo; por ejemplo, de 1000x1000. Además, dichas imágenes tendrán que ser algo variadas en su dibujo para que se note más el scroll, e importante, la imagen que será la de primer plano ha de tener bastantes zonas transparentes (es decir, pintada de negro real -RGB=0,0,0-) para que se puedan ver en esas zonas las porciones de la imagen de segundo plano que coincidan. Estas dos imágenes las incluiremos en un FPG llamado "scroll.fpg": el primer plano con código 001 y el segundo con 002.

Además, en nuestro ejemplo crearemos un proceso que se podrá mover en las cuatro direcciones mediante el cursor. Su gráfico puede ser cualquiera, por ejemplo de unos 40x40 píxeles, y lo llamaremos "grafproc.png".

Empezamos con el siguiente código inicial:

```
program hola;
private
    int idfpg;
end
begin
    set_mode(640,480,16);
    idfpg=load_fpg("scroll1.fpg");
    start_scroll(0,idfpg,1,2,0,0);
    procesol();
loop
    if(key(_q)) stop_scroll(0); end
    if(key(_esc)) exit();end
    frame;
end
end

process procesol()
begin
    graph=load_png("grafproc.png");
    x=320;y=240;
    loop
        if(key(_up)) y=y-5;end
        if(key(_down))y=y+5;end
        if(key(_left))x=x-5;end
        if(key(_right))x=x+5;end
        frame;
    end
end
```

Lo que hacemos aquí es poner un fondo de scroll mediante *start\_scroll*, y el proceso que se moverá encima de él. Fíjate que en las zonas transparentes del fondo de primer plano se tiene que ver lo que asoma del fondo de segundo plano.

Los valores de *start\_scroll* son claro: creamos el scroll nº 0, a partir de ningún FPG; el gráfico de primer plano será "scroll1.png" y el de segundo "scroll2.png"; la región donde será visible el scroll es la región 0 (toda la pantalla) y no habrá mosaico de scroll (es decir, cuando los procesos se desplacen hasta el extremo de la imagen que forma el scroll no podrán ir más allá).

Fíjate también que si apretamos la tecla “q”, desaparece el scroll, gracias a *stop\_scroll*. (Normalmente, esta orden se escribe justo antes de acabar el programa, pero la hemos puesto aquí para demostrar su acción). A lo mejor no lo notas al instante, pero intentas mover el proceso verás que no puedes, porque el scroll por donde debería de moverse ya no existe, y se queda bloquead.

No obstante, tenemos un problema: de entrada el fondo no se mueve, ni siquiera cuando movemos el proceso fuera de los límites de la pantalla. El fondo está tan quieto como lo estaba con *put\_screen*. ¿Qué nos falta? Decidir si vamos a hacer un scroll automático o “perseguidor” de un proceso.

Un pequeño inciso. Fíjate que las imágenes de primer plano y segundo plano del scroll las hemos cargado dentro de un archivo FPG. Podrías pensar que sería lo mismo haberlas cargado individualmente con *load\_png*, y luego haber escrito algo así como *start\_scroll(0,0,idpng1,idpng2,0,0)* para poner el scroll. Si lo haces de esta manera, verás con disgusto que las zonas que en teoría tenían que ser transparentes del gráfico de primer plano no lo son: son negras. Esto es porque al cargar Fénix los PNG de forma independiente, toma el color negro tal como es: negro, y para que las imágenes tengan zonas transparentes, ha de ser la propia imagen PNG la que incorpore las zonas transparentes tal cual -dicho técnicamente: ha de llevar declarado el canal alpha en su creación-. En cambio, al incluir un PNG en una librería FPG el PNG se transforma al formato interno de Fénix (FBM) y como consecuencia el color transparente de Fénix sí es tomado en cuenta.

Para empezar, en nuestro ejemplo vamos a implementar el scroll de tipo automático (luego ya veremos el otro). Para ello, has de saber lo siguiente:

Al igual que pasaba con el dispositivo ratón, una vez creado el scroll mediante *start\_scroll*, es posible ajustar diversos aspectos del mismo, incluso a cada frame, modificando una estructura global predefinida: en este caso la estructura llamada SCROLL. De esta manera, no es necesario usar un proceso para trabajar con scrolls, aunque si recomendable por el orden.

Más exactamente, como podemos definir 10 zonas de scroll, en realidad, de lo que disponemos es de una tabla de estructuras, donde **SCROLL[0]** será la estructura que contenga los datos de configuración para el scroll de nº 0, **SCROLL[1]** será la estructura correspondiente al scroll nº1, etc.

En concreto, para mover el scroll, ¿qué hay que hacer, pues?. La estructura **SCROLL[n]** tiene muchos campos, pero para mover el scroll vamos a usar los campos “**x0**”, “**y0**”, “**x1**” e “**y1**”. “**x0**” controla la coordenada x del primer plano del scroll y “**x1**” controla la coordenada x del segundo plano; de igual manera, “**y0**” controla la coordenada y del primer plano del scroll y “**y1**” la coordenada y del segundo plano, respecto la pantalla. Sólomente hay que establecer el valor que queremos a estos campos para desplazar el scroll a ese nuevo valor, igual que haríamos con las variables X e Y de cualquier gráfico de proceso.

Hay otro campo de la estructura SCROLL que puede que te interese, y es la variable “**ratio**” (usada así, por ejemplo: *scroll[0].ratio*). En ella se guarda la velocidad del segundo plano del fondo respecto al primer plano en porcentaje; así, si vale 100, el fondo se moverá a la misma velocidad que el plano principal, si vale 50, el fondo irá a la mitad de la velocidad y si vale 200 irá el doble de rápido (y si vale 33 irá a un tercio, etc.).

Vamos a implementar este tipo de scroll en el ejemplo anterior. Para ello, modificaremos el código añadiendo la siguiente línea dentro del bucle Loop/End del programa principal, justo antes de la orden *frame*::

```
scroll[0].y0=scroll[0].y0 - 10;
```

Lo que hace esta línea, en teoría, es modificar la posición del gráfico del primer plano del scroll de manera

que en cada frame se mueva 10 píxeles para arriba automáticamente. Pero si lo volvemos a probar, no pasa nada de nada...

Esto ocurre porque se nos ha olvidado cambiar el valor del último parámetro de *start\_scroll*, el parámetro de flags. Recordemos que si vale 0 no se va a poder ver ningún mosaico de imágenes de scroll, que es lo que necesitamos para crear un scroll de tipo automático, así que tendremos que cambiar ese valor. ¿Por cuál? Ya que con la línea que acabamos de añadir lo que estamos haciendo es mover el gráfico de primer plano del fondo verticalmente (de abajo a arriba), nos interesará generar un mosaico vertical para ese gráfico, así que valores apropiados podrían ser el 2,3,6,7,10,11,14 o 15. Si probamos con el 2 mismo, veremos que, efectivamente, ahora el gráfico de primer plano se mueve (y el de segundo plano se queda quieto).

Y de hecho, ocurrirá el mismo efecto pongamos el número que pongamos de la lista anterior como último parámetro del *start\_scroll*, ya que, si por ejemplo pones ahora allí un 10, puedes comprobar que el gráfico de segundo plano no se moverá, ya que este número sólo indica que ese gráfico “puede” crear un mosaico vertical, pero para ello se ha de mover en esa dirección, y eso lo tenemos que decir explícitamente añadiendo una línea más dentro del bucle Loop/End del programa principal, justo antes de la orden frame. Por ejemplo:

```
scroll[0].y1=scroll[0].y1 - 5;
```

Verás que haciendo esto, ahora los dos gráficos se mueven de abajo a arriba, aunque el de más al fondo lo hace la mitad de rápido.

Podrías haber obtenido exactamente el mismo efecto que el que acabamos de conseguir si en vez de escribir la línea anterior, escribes esto:

```
scroll[0].ratio=200;
```

Esta línea lo que hace es mover el gráfico de segundo plano en la misma dirección en que lo haga el de primer plano, (sea cual sea), a una velocidad relativa a éste dada por el porcentaje escrito: en este caso, “200” quiere decir el doble de lento, (50 querría decir el doble de rápido, y así). Compruébalo.

Finalmente, y si pruebas ahora de poner un 3 como último parámetro de *start\_scroll*, ¿qué pasa?. Primero: volvemos a dejar sin movimiento el gráfico de segundo plano porque aunque ahora tengamos escrita una línea que mueve dicho gráfico -bien usando “*y1*”, bien usando “*ratio*”-, éste no tiene ya la posibilidad de hacer mosaico (es como si hubiéramos escrito un “0” de valor para ese fondo en *start\_scroll*). Y además, con el valor “3”, estamos concediendo la posibilidad de que se produzca un mosaico horizontal, pero para verlo necesitaremos mover horizontalmente el gráfico de primer plano, añadiendo esta línea:

```
scroll[0].x0= scroll[0].x0 - 20;
```

Si escribimos esta nueva línea, verás que el scroll se mueve en diagonal hacia arriba a la izquierda: esto es evidentemente porque estamos moviéndolo arriba con “*y0*” y a la izquierda con “*x0*”, y los efectos se suman.

Te recomendaría que probaras diferentes valores para el último parámetro de *start\_scroll* y comprobaras su efecto en el movimiento del fondo: es la mejor manera de aprender.

Un dato curioso respecto al valor de este último parámetro de *start\_scroll*: si escribes un valor que posibilite el mosaico horizontal y/o vertical de alguno de sus dos gráficos, y ese gráfico tiene unas dimensiones menores que el ancho y alto de la pantalla, lo que observarás es que con sólo escribir *start\_scroll*, se produce un efecto de mosaico de forma automática en el fondo de nuestro juego, sin tener

que programarlo “a mano” ,como cuando escribimos el ejemplo expuesto en la explicación de la función *put*.

Ya tenemos creado nuestro scroll “automático”. Ahora vamos a por el otro tipo, el scroll “perseguidor”. Volvemos a partir del código inicial que teníamos:

```
program hola;
private
    int idfpg;
end
begin
set_mode(640,480,16);
idfpg=load_fpg("scroll1.fpg");
start_scroll(0,idfpg,1,2,0,0);
procesol();
loop
    if(key(_q)) stop_scroll(0); end
    if(key(_esc)) exit();end
    frame;
end
end

process procesol()
begin
    graph=load_png("grafproc.png");
    x=320;y=240;
    loop
        if(key(_up)) y=y-5;end
        if(key(_down))y=y+5;end
        if(key(_left))x=x-5;end
        if(key(_right))x=x+5;end
        frame;
    end
end
```

y añadiremos otro proceso más -aquí lo dibujaremos con el mismo gráfico que el otro, es igual-, para notar mejor los efectos de este scroll (acuérdate de llamarlo desde el programa principal):

```
process proceso2()
begin
    graph=load_png("grafproc.png");
    x=220;y=200;
    loop
        if(key(_a)) angle=angle+5000;end
        if(key(_d)) angle=angle-5000;end
        if(key(_s))advance(5);end
        frame;
    end
end
```

Lo primero que hay que hacer es decir cuál va a ser el proceso que será perseguido por el scroll. Es decir, cuál será el proceso “cámara”. Para ello, hay que asignar al campo “**camera**” del scroll correspondiente el identificador del proceso que queremos que persiga. Esto se puede hacer desde el código principal, pero lo

más habitual es hacerlo al principio del código del propio proceso, así:

```
scroll[0].camera=id;
```

donde ID ya sabemos que es una variable predefinida local que devuelve el identificador de proceso del proceso actual. Prueba de añadir esta línea justo después del BEGIN de “proceso1”, por ejemplo, y prueba de mover este proceso con los cursores.

Parece que va, pero no del todo bien. Por ejemplo, si el proceso sale de los límites de la pantalla, desaparece, y lo chulo sería que se fuera viendo cómo éste se va desplazando por el scroll sin salirse nunca de la pantalla. Otra cosa que no queda bien es que “proceso2” no participa del movimiento del scroll: se queda fijo, con lo que no es realista que nos desplazemos: los procesos también deberían de moverse al igual que el scroll para dar una sensación de distancia y posicionamiento fijo real dentro de un mapa scrolleable. ¿Cómo solucionamos estos dos problemas?

Bueno, en realidad estos dos efectos son consecuencia de un sólo problema. Por defecto, los procesos no son dibujados en ningún scroll, por lo que por defecto no responden a la lógica de éstos; es como si el scroll fuera por un lado, y los procesos por otro, sin enterarse de la existencia de los primeros. Hay que decir explícitamente que tal proceso o tal otro queremos que forme parte del scroll y así se comportará como esperamos. Y esto se hace con la variable local predefinida **CTYPE**.

Si queremos que un proceso forme parte de un scroll y se mueva acorde con él, es preciso que la variable local del proceso **CTYPE** sea igual a la constante predefinida **C\_SCROLL** (con valor real 1).

Importante: hecho este cambio, las variables de posición X e Y del proceso se considerarán relativas a la esquina superior izquierda del gráfico de primer plano del scroll y no a la de la pantalla.

Si queremos que un proceso deje de formar parte de un scroll (su valor por defecto), esta variable ha de ser igual a la constante predefinida **C\_SCREEN** (con valor real 0).

Otra cosa: por defecto, los procesos con **CTYPE=1** serán visibles en todos los scrolls existentes en nuestro juego. Para escoger un scroll concreto (o más) en los que un proceso es visible, hay que asignar a su variable local predefinida **CNUMBER** la suma de una o más de las constantes predefinidas **C\_0**, **C\_1**... hasta **C\_9**, para indicar qué números de scrolls son aquellos en los que el proceso será visible. Así, con **CNUMBER = C\_0 + C\_2** se hace que un proceso sea visible en las zonas de scroll 0 y 2, pero no en la 1 ni en ninguna otra (si existe). Los valores reales de estas constantes predefinidas, por si interesa, son:

- 1 : **C\_0**
- 2 : **C\_1**
- 4 : **C\_2**
- 8 : **C\_3**
- 16 : **C\_4**
- 32 : **C\_5**
- 64 : **C\_6**
- 128 : **C\_7**
- 256 : **C\_8**
- 512 : **C\_9**

El valor por defecto de **CNUMBER** es 0, que indica, como hemos dicho, que los procesos son visibles en todos los scrolls existentes.

Esta variable sólo tiene sentido si se inicializa también la variable local CTYPE.

Una vez que ya conocemos la existencia de estas dos nuevas variables predefinidas, vamos a usarlas.

Si queremos que un proceso sea visible y pertenezca a nuestro scroll, lo que tendremos que escribir, al principio del código de cada uno de estos procesos, las líneas:

```
ctype=c_scroll;  
cnumber=0;
```

aunque ésta última no es necesaria porque su valor por defecto ya es el hemos puesto: si sólo quisiéramos que los procesos se vieran en el scroll 0 (que de hecho, es el único que hay), también podríamos haber puesto `cnumber=c_0`. Puedes probar otros valores para ver lo que pasa.

Así pues, si has escrito ambas líneas en “proceso1” y “proceso2”, ya tendremos nuestro scroll “perseguidor” perfectamente funcional. Y ya está.

Si has seguido los pasos que se han ido marcando, al final tendrás que tener un código como éste:

```
program hola;  
private  
    int idfpg;  
end  
begin  
set_mode(640,480,16);  
idfpg=load_fpg("scroll.fpg");  
start_scroll(0,idfpg,1,2,0,3);  
proceso1();  
proceso2();  
loop  
    if(key(_q)) stop_scroll(0); end  
    if(key(_esc)) exit();end  
frame;  
end  
end  
  
process proceso1()  
begin  
graph=load_png("grafproc.png");  
x=320;y=240;  
ctype=c_scroll;  
cnumber=c_0;  
scroll[0].camera=id;  
/*Estas dos líneas muestran que las coordenadas X e Y del proceso son relativas a la esquina superior  
izquierda del gráfico del primer plano del scroll (y no de la pantalla como hasta ahora) porque este  
proceso pertenece al scroll gracias a la línea ctype=c_scroll. Si se observan valores negativos, es porque  
se ha salido fuera de los límites del gráfico a causa de la repetición de éste en mosaico.*/  
write_var(0,20,10,4,x);  
write_var(0,20,20,4,y);  
loop  
    if(key(_up)) y=y-5;end  
    if(key(_down))y=y+5;end
```

```

                if(key(_left))x=x-5;end
                if(key(_right))x=x+5;end
                frame;
            end
        end

        process proceso2()
        begin
            graph=load_png("grafproc.png");
            x=220;y=200;
            ctype=c_scroll;
            cnumber=c_0;
            /*Estas dos líneas muestran que las coordenadas X e Y del proceso son relativas a la esquina superior
            izquierda del gráfico del primer plano del scroll (y no de la pantalla como hasta ahora) porque este
            proceso pertenece al scroll gracias a la línea ctype=c_scroll. Si se observan valores negativos, es porque
            se ha salido fuera de los límites del gráfico a causa de la repetición de éste en mosaico.*/
            write_var(0,20,30,4,x);
            write_var(0,20,40,4,y);
            loop
                if(key(_a)) angle=angle+5000;end
                if(key(_d)) angle=angle-5000;end
                if(key(_s))advance(5);end
                frame;
            end
        end
    end
end

```

A partir del código anterior, podemos irlo ampliando y modificarlo según nuestros intereses. Por ejemplo, un ejercicio: intenta hacer que cuando “proceso1” colisione con “proceso2”, éste último cambie su posición a otras coordenadas (x,y) aleatorias. Es muy sencillo.

Ahora vamos a ver un par más de ejemplos de diferentes scrolls, para tener más seguridad en este tema:

El siguiente ejemplo muestra un scroll con sólo un gráfico (el de primer plano), el cual se irá moviendo autónomamente e irá cambiando su dirección de movimiento automáticamente cada cinco segundos de tal manera: dirección noroeste,norte,noreste,este,sureste,sur,suroeste,oeste. Además, si se apreta la tecla Enter, el scroll se parará durante tres segundos, para reanudarse automáticamente en acabar éstos.

```

program Test_STOP_SCROLL;
global
    int ixS,iyS; //Cantidad horizontal y vertical que se moverá el scroll en cada frame, respectivamente
    int DirS=1; /*Según el valor que tenga, el valor de ixS e iyS será diferente, y por tanto el movimiento del
    scroll será en una dirección u otra*/
    int fpg;
end
begin
    set_mode(640,480,16);
    fpg=load_fpg("scroll.fpg");
    write(0,160,100,4,"ENTER = Detiene Scroll durante 3 segundos:");
    start_scroll(fpg,0,1,0,0,3); //Comienza el programa con el scroll iniciado
end

```

```

repeat
/*Si pulso ENTER y el scroll está funcionando, lo paro, y pongo en marcha el temporizador que a los tres segundos lo reemprenderá*/
  if(key(_enter))
    stop_scroll(0);
    timer[0]=0;
/*Durante tres segundos justo después de haber parado el scroll, no pasa nada, con lo que el scroll continúa parado. Pasado este tiempo, lo volvemos a iniciar.*/
  while(timer[0]<=300) frame; end
  start_scroll(fpg,0,1,0,0,3);
end

/*Sumo 1 al valor de esta variable, hasta un máximo de 8, que son las combinaciones que se han definido en este ejemplo para mostrar el movimiento del scroll en las 8 direcciones principales de las dos dimensiones*/
DirS=(DirS+1)%8;
switch(DirS)
  case 1: ixS=3; iyS=0; end
  case 2: ixS=3; iyS=3; end
  case 3: ixS=0; iyS=3; end
  case 4: ixS=-3; iyS=3; end
  case 5: ixS=-3; iyS=0; end
  case 6: ixS=-3; iyS=-3; end
  case 7: ixS=0; iyS=-3; end
  case 8: ixS=3; iyS=-3; end
end

/*Ponemos en marcha otro temporizador que hará moverse en una determinada dirección, determinada por ixS y iyS (determinados a su vez por DirS), al scroll durante 5 segundos. Al cabo de los cuales, durante otros 5 segundos se moverá en otra dirección, y así*/
timer[1]=0;
while(timer[1]<=500)
  scroll.x0=scroll.x0+ixS; scroll.y0=scroll.y0+iyS;
  if(key(_esc)) exit(); end
  if(key(_enter)) break; end
  frame;
end
until(key(_esc))
end

```

Otro ejemplo muy interesante es el que viene a continuación. En él se puede observar el uso de un scroll delimitado dentro de una región definida, de tamaño menor que la pantalla. Con este ejemplo se puede ver cómo se podría implementar un scroll sólo en una zona concreta de la pantalla en vez de en toda ella, algo muy útil para multitud de juegos (carreras, mapas en miniatura, etc).

Para poderlo ejecutar, necesitarás crear un gráfico mayor que el tamaño de la pantalla y tendrás que guardarlo dentro del FPG que estamos usando en estos ejemplos, "scroll.fpg" con el código 003. La idea es que esta nueva imagen sea el fondo fijo puesto con `put_screen` visible en toda la pantalla excepto en una zona de ésta, que corresponderá a una región definida con `DEFINE_REGION` -con el número 1- dentro de la cual se habrá implementado un scroll con movimiento automático vertical del gráfico de primer plano (código 001) y con el gráfico de segundo plano quieto (código 002).

```
PROGRAM prueba2;
```

```

CONST
ANCHOPANT = 640;
ALTOPANT = 480;
ANCHOSCROLL = 300;
ALTOSCROLL = 300;
END
GLOBAL
  int fpgId;
END
BEGIN
fpgId = load_fpg("scroll.fpg");
set_mode(ANCHOPANT,ALTOPANT,16);
put_screen(fpgId,3);
define_region(1,(ANCHOPANT-ANCHOSCROLL)/2,(ALTOPANT-ALTOSCROLL)/2,
ANCHOSCROLL,ALTOSCROLL);
scroller();
loop
    frame;
end
END

PROCESS scroller()
BEGIN
x = ANCHOSCROLL/2;
y = ALTOSCROLL/2;
start_scroll(0, fpgId, 1, 2, 1, 2);
scroll[0].camera = ID;
loop
    y = y + 5;
    frame;
end
END

```

Este código tiene varias cosas interesantes. Lo primero es que definimos una región con el código 1, cuya esquina superior izquierda es un punto tal que el centro de la región coincide con el centro de la pantalla (es fácil verlo si haces un pequeño diagrama). Seguidamente, después de definir la región, llamamos a un proceso “scroller”, que se encargará de poner en marcha el scroll. Y esto lo hace de una manera peculiar: fijarse primero que el quinto parámetro de *start\_scroll* es “1”, lo cual indica que el scroll sólo será visible en esa región, pero lo más interesante está en la línea siguiente: *scroll[0].camera=ID*. Lo que hacemos aquí, ya lo debes saber, es hacer que el scroll se mueva siempre que se mueva el proceso actual: si éste no se mueve, el scroll no se moverá (es lo que llamábamos un scroll “perseguidor”). Pues aprovechamos precisamente este hecho para escribir dentro del LOOP/END un incremento perpetuo en la posición vertical del proceso actual. Como dicho proceso no parará de moverse de forma indefinida, el scroll lo seguirá de forma indefinida, con lo que observaremos el efecto del movimiento continuo del scroll. Fíjate que no hemos definido ningún graph para este proceso, por lo que el proceso es invisible, pero a pesar de eso, su posición vertical varía de igual manera, que es lo que nos interesa para lo que pretendemos.

A parte de los campos que ya hemos visto de la estructura SCROLL (x0,y1,radio,camera...), existen unos cuantos campos más que merece la pena conocer porque son muy interesantes y nos pueden ayudar a conseguir efectos realmente logrados en nuestros juegos.

Uno de estos campos es el campo “z”, que como puedes intuir, indica la profundidad del scroll. Por defecto su valor es 512. Piensa que aunque los scrolls tengan gráfico de primer plano y de segundo, su Z es única, por lo que el nombre de “primer plano” y “segundo plano” no corresponde a la realidad estricta:

ambos gráficos poseen a efectos prácticos la misma profundidad.

Si hacemos que la Z de un proceso sea mayor que la del scroll, éste tapaná al gráfico del proceso, ya que estaremos dibujando el fondo movable encima. Aplicación: si los gráficos del scroll poseen amplias áreas transparentes (por ejemplo, representan el cielo salpicado de nubes), con una Z de scroll pequeña se logrará que éste pase por encima de los gráficos de los procesos con Z mayor. Además, aparte de poder jugar entre las profundidades del scroll y la de los procesos también se puede jugar con las profundidades de diferentes scrolls (SCROLL[0], SCROLL[1]...) entre ellos, logrando efectos realmente muy conseguidos.

Un ejemplo muy sencillo:

```
program hola;
private
    int idfpg;
end
begin
    set_mode(640,480,16);
    idfpg=load_fpg("scroll.fpg");
    start_scroll(0,idfpg,1,0,0,2);
    scroll[0].z=-1;
    procesol();
loop
    scroll[0].y0=scroll[0].y0 + 1;
    frame;
end
end

process procesol()
begin
    graph=load_png("grafproc.png");
    x=320;y=240;
loop
    frame;
end
end
```

Ya que la Z de los procesos por defecto es 0, sólo hemos tenido que modificar la Z del scroll poniéndola a -1 para que el scroll se pinte por encima del gráfico del proceso. Si el gráfico del scroll tuviera muchas zonas transparentes, se podría comprobar el efecto comentado de las “nubes del cielo”: si no, el proceso no se verá porque quedará oculto bajo el scroll.

Jugando con distintas profundidades para múltiples scrolls y/o personajes, se pueden obtener efectos curiosos. Por ejemplo, en el siguiente código tenemos dos scrolls automáticos independientes, de un sólo gráfico cada uno (o sea, no tienen gráfico de segundo plano), los cuales están situados a distintas profundidades y van a velocidades diferentes. Al usar dos scrolls pero sólo de un plano, el rendimiento es más o menos el mismo que si fuese uno solo de dos planos.

Además, tenemos tres procesos cuyas profundidades se pueden observar que son: por encima de ambos scrolls, en medio de ellos y por debajo de ambos scrolls. Dependiendo de las zonas transparentes que tengas en los gráficos de los dos scrolls, verás o no según qué procesos en cada momento.

```
Program scroll1;
private
    int idfpg;
```

```

end
Begin
set_mode (800,600,16);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 1);
scroll[0].z= 10;
start_scroll(1, idfpg, 2, 0, 0, 1);
scroll[1].z= 20;

proceso(120,140,1);
proceso(360,240,15);
proceso(620,340,25);
loop
    scroll[0].x0=scroll[0].x0 +5;
    scroll[1].x0=scroll[0].x0 * 2; //El scroll 1 va el doble de rápido que scroll 0
    frame;
    if(key(_esc)) exit();end
end
end

process proceso(x,y,z)
begin
    graph=load_png("grafproc.png");
    loop
        frame;
    end
end

```

Podemos hacer el mismo ejemplo pero haciendo que un scroll sea “perseguidor” de un proceso y el otro de otro proceso, por ejemplo. En el código siguiente, al igual que el anterior, hay dos scrolls de un sólo fondo, a diferentes z, y tres procesos cuya z es: encima de los dos scrolls, entre uno y otro y debajo de los dos. La novedad está en que uno de esos procesos es la cámara de uno de los scrolls, y otro proceso es la cámara del otro. Pruébalo.

```

Program scroll1;
private
    int idfpg;
end
Begin
set_mode (800,600,16);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 1);
scroll[0].z= 10;
start_scroll(1, idfpg, 2, 0, 0, 1);
scroll[1].z= 20;

scroll[0].camera=proceso1(120,140,1);
scroll[1].camera=proceso2(360,240,15);
proceso3(620,340,25);
loop
    frame;
    if(key(_esc)) exit();end
end
end

```

```

process proceso1(x,y,z)
begin
  ctype=c_scroll;
  graph=load_png("grafproc.png");
  loop
    if(key(_up))advance(5);end
    if(key(_left)) angle=angle+5000;end
    if(key(_right)) angle=angle-5000;end
    frame;
  end
end

process proceso2(x,y,z)
begin
  ctype=c_scroll;
  graph=load_png("grafproc.png");
  loop
    if(key(_s))advance(5);end
    if(key(_a)) angle=angle+5000;end
    if(key(_d)) angle=angle-5000;end
    frame;
  end
end

process proceso3(x,y,z)
begin
  ctype=c_scroll;
  graph=load_png("grafproc.png");
  loop
    frame;
  end
end

```

Pero, ¡ojo!, si pruebas el código anterior, verás que ocurre una cosa muy rara: ¡los procesos se ven duplicados! Parece muy extraño pero es lógico: si te fijas, no le hemos dicho a ningún proceso en qué scroll queremos que sea visible (con CNUMBER), con lo que por defecto los procesos serán visibles en todos los scrolls que haya. Así que, lo que tienes que hacer es lo siguiente: en “proceso1” tienes que añadir al principio la línea *cnumber=c\_0*; en “proceso2” tienes que añadir la línea *cnumber=c\_1*; y en “proceso3” *cnumber=c\_0*; ó *cnumber=c\_1*; tanto da. Lo que hacemos con esto es decir que “proceso1” sólo se verá en el scroll 0, del cual es cámara -y como tiene una Z menor, estará por encima y se podrá ver-, que “proceso2” sólo se verá en el scroll 1 de la misma manera, y que “proceso3” se ha de ver o en el scroll 0 o en el 1 pero que esto es indiferente porque como su Z es mayor, no se va a ver en ninguno. Pruébalo ahora y verás que obtienes el efecto deseado.

Existen más campos interesantes de la estructura SCROLL. Uno de ellos es “**follow**”. Este campo es recomendable usarlo con scrolls que sólo tengan el gráfico de primer plano, (ninguno de segundo plano, para evitar posibles inconsistencias). Para explicar su significado, partamos de este código, ya visto antes:

```

Program scroll1;
private
  int idfpg;
end

```

```

Begin
set_mode (800,600,16);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 1);
scroll[0].z= 10;
start_scroll(1, idfpg, 2, 0, 0, 1);
scroll[1].z= 20;
loop
  scroll[0].x0=scroll[0].x0 +5;
  scroll[1].x0=scroll[0].x0 * 2; //El scroll 1 va el doble de rápido que scroll 0
  frame;
  if(key(_esc)) exit();end
end
end

```

Aquí tenemos dos scrolls a distintas profundidades, que se mueven de manera que uno está siguiendo al otro a distinta velocidad. Pues bien, el campo “follow” sirve precisamente para esto de forma no “tan manual”: es decir, sirve para hacer que un scroll determinado siga automáticamente -“follow” quiere decir “seguir” en inglés- a otro scroll “maestro”, cuyo número será el valor que asignaremos a este campo. Es decir, que si escribimos `scroll[1].follow=0;`, lo que estaremos diciendo es que el scroll nº 1 seguirá el movimiento que haga el scroll 0, sea en la dirección que sea.

Así pues, podríamos sustituir la línea del código anterior `scroll[1].x0=scroll[0].x0 * 2;` por esta nueva que acabamos de descubrir: `scroll[1].follow=0;` Pero, ¿y la velocidad relativa? En la línea antigua decíamos que el scroll nº1 fuera el doble de rápido que el scroll nº 0. ¿Cómo se especifica ahora eso?

Pues con el mismo campo “**ratio**” que ya conocemos. Si te acuerdas cuando explicamos este campo, dijimos que lo que significaba era el tanto por ciento de velocidad relativa que el segundo plano de un scroll tenía respecto el primer plano. Pues bien, en el momento que un scroll persiga a otro, su campo “ratio” deja de significar eso para significar el tanto por ciento de velocidad relativa que tiene el scroll perseguidor del scroll perseguido. Haz una prueba: añade antes de la orden `frame;` del ejemplo anterior esta línea:

```

scroll[1].ratio=200;

```

Verás que el scroll 1 va el doble de rápido que el 0. Si en cambio, escribiéras `scroll[1].ratio=10`, verías que va diez veces más lento.

Otro ejemplo ilustrativo donde se puede apreciar el uso de “follow” más el uso de regiones lo podemos encontrar en este código. En él tenemos definidas dos regiones (más la pantalla), y en cada una de ellas definimos un scroll diferente, que además se moverán a diferentes velocidades relativas.

Este efecto es muy interesante si se quiere implementar por ejemplo un paisaje donde a primer plano (en la parte inferior de la pantalla) se puedan ver los elementos más cercanos moviéndose con rapidez, a medio plano (en la parte media de la pantalla, en otra región) se puedan ver los elementos un poco más alejados moviéndose más lentamente y en el plano de fondo (en una parte más superior de la pantalla, en otra región) se visualicen los elementos más alejados y más lentos. Como remate en este supuesto, se podría incluir en la parte más superior de la pantalla un fondo quieto con `put_screen`. El resultado es muy convincente.

```

program prueba;
private
  int idfpg;
end

```

```

begin
set_mode(640,480,16);
idfpg=load_fpg("scroll.fpg");

//La región 1 estará en la parte superior de la pantalla, y la 2 en la inferior
define_region(1,0,0,640,224);
define_region(2,0,300,640,180);

//Dibujo dos rectángulos delimitando visualmente las dos regiones
draw_rect(0,300,640,480);
draw_rect(0,0,640,224);

//Prueba a cambiar los valores del último parámetro de start_scroll por 1, a ver qué pasa
start_scroll(1,idfpg,1,3,1,0);
/*El scroll 0 sólo se verá fuera de las dos regiones, en medio de la pantalla. Si se quiere
que se vea en las otras regiones, se tendrá que disminuir su Z (p.ej: scroll[0].z=-1;)*
start_scroll(0,idfpg,1,0,0,0);
start_scroll(2,idfpg,1,2,2,0);

//Sitúo en distintas coordenadas iniciales el gráfico de scroll
scroll[0].x0=250;
scroll[2].x0=420;
scroll[1].x0=550;

scroll[2].follow=0;
scroll[1].follow=0;
scroll[2].ratio=50;
scroll[1].ratio=250;

mariposo();

repeat
if(key(_left))scroll[0].x0=scroll[0].x0+4;end
if(key(_right))scroll[0].x0=scroll[0].x0-4;end
frame;
until(key(_esc))
exit();
end

process mariposo()
begin
x=320;y=250;
graph=load_png("grafproc.png");
/*Con esta línea, las coordenadas X e Y se referirán al scroll y no a la pantalla. Se ha de tener en cuenta
que si el scroll también se mueve (ya que usamos la misma tecla para el movimiento de las dos cosas), esto
hará que el proceso vaya a una velocidad bastante elevada, lógicamente, porque hay que sumar los 5
píxeles de rigor a la nueva posición del scroll, la cual también está cambiando a cada momento*/
ctype=c_scroll;
//Hacemos que el proceso sólo sea visible en la franja del medio de la pantalla (el scroll 0),
cnumber=c_0;
loop
if(key(_left)) x=x-5;end
if(key(_right)) x=x+5;end
if(key(_up)) y=y-5;end

```

```
if(key(_down)) y=y+5;end
frame;
end
end
```

Otros campos muy interesantes de la estructura SCROLL son “**flags1**” y “**flags2**”. Su significado y sus posibles valores son los mismos que la variable local predefinida FLAGS, pero referidos al gráfico de primer plano y al de segundo plano de un scroll, respectivamente. Con estos campos puedes obtener efectos muy logrados de transparencias o efecto espejo, por ejemplo.

La estructura SCROLL también dispone de los campos “**region1**” y “**region2**”, que representan la llamada “región de bloqueo” (el primero para el gráfico de primer plano y el segundo para el gráfico de segundo plano del scroll). Es decir, que si hacemos por ejemplo scroll[0].region1=2, lo que estaremos haciendo es asignar la región número 2 como región de bloqueo del gráfico de primer plano del scroll 0. Y una región de bloqueo es simplemente una zona donde el scroll no se moverá mientras el proceso cámara permanezca dentro de ella. Por defecto, el campo region1 vale -1, que quiere decir que no hay ninguna región de bloqueo.

En el uso de scrolls también disponemos de una función más (relativamente avanzada) que no hemos comentado hasta ahora:

### MOVE\_SCROLL(NUMERO)

La función requiere como parámetro el número de scroll de 0 a 9 que se indicó en la función START\_SCROLL como primer parámetro cuando se inició el scroll.

Esta función se utiliza cuando el scroll se controla automáticamente, por haber definido el campo camera de la estructura scroll correspondiente con el identificador de un proceso.

El propósito es forzar a que se actualicen los valores (x0, y0, x1 y y1) de dicha estructura; si no se utiliza esta función estos valores no se actualizarán hasta la próxima imagen del juego. Es decir, cuando un proceso necesita conocer antes de la próxima imagen el valor de las coordenadas de un scroll (normalmente para colocarse él en una posición acorde al movimiento del fondo) se debe hacer esto:

1. Se inicia el scroll con START\_SCROLL.
2. Se crea el proceso que se utilizará como cámara y se pone su código identificador en el campo camera de la estructura scroll.
3. A este proceso se le debe poner una prioridad muy alta, para que se ejecute antes que el resto de los procesos (poniendo en su variable local priority un valor entero positivo como, por ejemplo, 100).
4. Justo antes de la sentencia FRAME del bucle del proceso usado como cámara se llamará a la función MOVE\_SCROLL.

De esta forma se garantizará que este proceso se ejecute el primero y, justo al finalizar, actualice los valores (x0, y0, x1 y y1) de la estructura scroll, de forma que el resto de los procesos puedan utilizar estas variables ya actualizadas.

El uso más generalizado de esta función es cuando en una ventana de scroll se quieren tener más de dos planos de fondo y, para ello, se crean una serie de procesos que simulen un tercer o cuarto plano, situando sus coordenadas en función de la posición exacta del scroll en cada imagen.

#### PARAMETROS:

INT NUMERO : Número identificador de scroll (0 a 9)

Un ejemplo:

```
program Test_MOVE_SCROLL;
```

```

global
int DirS;
int fpg;
int ixS,iyS;
end
begin
set_mode(640,480,16);
fpg=load_fpg("scroll.fpg");
start_scroll(0,fpg,1,0,0,3);
/*La cámara del scroll será el programa principal, el cual variará su X e Y más adelante y forzará así a
mover el scroll mediante move_scroll para seguirlo.*/
scroll[0].camera=id;
DirS=7;
repeat
switch(DirS)
case 1: ixS=3; iyS=0; end
case 2: ixS=3; iyS=3; end
case 3: ixS=0; iyS=3; end
case 4: ixS=-3; iyS=3; end
case 5: ixS=-3; iyS=0; end
case 6: ixS=-3; iyS=-3; end
case 7: ixS=0; iyS=-3; end
case 8: ixS=3; iyS=-3; end
end
timer[1]=0;
while(timer[1]<=50)
x=x+ixS; y=y+iyS;
move_scroll(0);
frame;
if(key(_esc)) exit(); end
if(key(_enter)) break; end
end
DirS=(DirS+1)%8;
until(key(_esc))
end

```

Finalmente, no hemos hablado de la utilización del ratón con los scrolls. En concreto, en los ejemplos anteriores siempre hemos usado el teclado para mover nuestro personaje, pero, ¿qué pasa si a éste lo queremos dirigir mediante el ratón? Pues que tenemos que tener cuidado con las coordenadas.

Para empezar, prueba este ejemplo (necesitarás un nuevo gráfico de unos 10x10 llamado "raton.png" que representará el cursor del ratón):

```

Program scroll1;
private
int idfpg;
end
Begin
set_mode (800,600,16);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 3);
scroll[0].camera=procesol(160,300) ;

```

```

grafrat();
loop
  frame;
  if(key(_esc)) exit();end
end
end

process procesol(x,y)
begin
  ctype=c_scroll;
  cnumber=c_0;
  graph=load_png("grafproc.png");
  write_var(0,100,100,4,x);
  write_var(0,100,120,4,y);
  write_var(0,100,140,4,mouse.x);
  write_var(0,100,160,4,mouse.y);
  loop
    if(key(_left)) x=x-5; end
    if(key(_right)) x=x+5; end
    if(key(_up)) y=y-5; end
    if(key(_down)) y=y+5; end
    frame;
  end
end

```

```

process grafrat()
begin
  graph=load_png("raton.png");
  loop
    x=mouse.x;
    y=mouse.y;
    frame;
  end
end

```

Es fácil ver que las coordenadas del proceso y las del ratón van completamente a su aire y no coinciden en nada. A estas alturas ya deberías saber por qué: porque el proceso que representa el cursor del ratón no pertenece a ningún scroll y por tanto sus coordenadas son relativas a la pantalla. Así que lo único que tenemos que hacer es añadir, antes del loop de “grafrat”, las líneas *ctype=c\_scroll;* y *cnumber=c\_0;*.

Si vuelves a probar el código, ahora verás que las coordenadas coinciden, por lo que parece que estamos cerca de la solución: ahora sólo tenemos que cambiar la manera en que se mueva “procesol” para que cada vez que se clique con el botón derecho, ese proceso se dirija al punto donde se ha hecho el click. Es decir, tener algo como esto:

```

Program scroll1;
private
  int idfpg;
end
Begin
  set_mode (800,600,16);
  idfpg=load_fpg("scroll.fpg");
  start_scroll(0, idfpg, 1, 0, 0, 3);
  scroll[0].camera=procesol(160,300) ;

```

```

grafrat();
loop
  frame;
  if(key(_esc)) exit();end
end
end

process procesol(x,y)
begin
  ctype=c_scroll;
  cnumber=c_0;
  graph=load_png("grafproc.png");
  write_var(0,100,100,4,x);
  write_var(0,100,120,4,y);
  write_var(0,100,140,4,mouse.x);
  write_var(0,100,160,4,mouse.y);
  loop
    if(mouse.left)
      x=mouse.x;
      y=mouse.y;
    end
    frame;
  end
end

```

```

process graftrat()
begin
  graph=load_png("raton.png");
  ctype=c_scroll;
  cnumber=c_0;
  loop
    x=mouse.x;
    y=mouse.y;
    frame;
  end
end

```

Aparentemente va bien, pero ¡ojo!, si el cursor se mueve más allá de las coordenadas de la resolución de la pantalla (en este caso, 800x600), éste deja de responder: ¡no podemos mover nuestro personaje más allá de las dimensiones de la pantalla, aunque el gráfico de scroll sea más grande! ¿Cómo solucionamos esto? Así (las modificaciones están en negrita):

```

Program scroll1;
global
  int idgraftrat;
end
private
  int idfpg;
end
Begin
set_mode (800,600,16);
idfpg=load_fpg("scroll.fpg");
start_scroll(0, idfpg, 1, 0, 0, 3);
idgraftrat=graftrat();

```

```

scroll[0].camera=procesol(160,300) ;
loop
  frame;
  if(key(_esc)) exit();end
end
end

```

```

process procesol(x,y)
begin
  ctype=c_scroll;
  cnumber=c_0;
  graph=load_png("grafproc.png");
  write_var(0,100,100,4,x);
  write_var(0,100,120,4,y);
  write_var(0,100,140,4,mouse.x);
  write_var(0,100,160,4,mouse.y);
  write_var(0,100,180,4,idgrafrat.x);
  write_var(0,100,200,4,idgrafrat.y);
  loop
    if(mouse.left)
      x=idgrafrat.x;
      y=idgrafrat.y;
    end
    frame;
  end
end

```

```

process grafrat()
begin
  graph=load_png("raton.png");
  ctype=c_scroll;
  cnumber=c_0;
  loop
    x=scroll[0].x1+mouse.x;
    y=scroll[0].y1+mouse.y;
    frame;
  end
end

```

Hemos hecho que “procesol” siga no al ratón en sí sino al proceso que representa gráficamente su cursor, y dicho proceso tendrá unas coordenadas relativas al scroll dadas, en todo momento, por la suma de la posición del cursor del ratón respecto la pantalla más la posición en ese momento del scroll respecto la pantalla (es una simple traslación de coordenadas).

Otro ejemplo similar al anterior sería el que viene: aquí tenemos un proceso que está permanentemente orientado con su ANGLE mirando al cursor del ratón, y si apretamos las teclas de los cursores, si irá moviendo lentamente hacia la posición que ocupe en ese momento dicho cursor. No necesitas ningún gráfico externo.

```

PROGRAM test;
CONST
    screenx = 1024;
    screeny = 768;
END

```

```

GLOBAL
    int vizierid;
    int mapscroll;

END
BEGIN
SET_MODE(screenx,screeny,16);
mapscroll=new_map(200,200,16);map_clear(0,mapscroll,rgb(155,30,234));
START_SCROLL(0,0,mapscroll,0,0,3);
character();
vizierid = vizier();
LOOP
    IF (key(_esc)) exit(); fade_off(); END
    FRAME;
END
END

```

```

PROCESS vizier()
BEGIN
graph=new_map(20,20,16);map_clear(0,graph,rgb(55,55,34));
ctype=c_scroll;
LOOP
    x=scroll[0].x1+mouse.x;
    y=scroll[0].y1+mouse.y;
    FRAME;
END
END

```

```

PROCESS character()
BEGIN
x=screenx/2;
y=screeny/2;
graph=new_map(50,50,16);map_clear(0,graph,rgb(155,155,134));
ctype=c_scroll;
scroll.camera=id;
LOOP
    angle = get_angle(vizierid);
    IF (key(_up)) xadvance(angle,4); END
    IF (key(_down)) xadvance(angle,-4);END
    IF (key(_left)) xadvance(angle+90000,4); END
    IF (key(_right))xadvance(angle-90000,4); END
    FRAME;
END
END

```

Ya para acabar, podríamos haber modificado el código anterior para hacer que el proceso “character” cambie su orientación siguiendo la posición del cursor del ratón de otra manera diferente (es cuestión de gustos). En vez de escribir la línea `angle = get_angle(vizierid)`; podríamos haber puesto:

```

difmouse = mouse.x-bmousex;
angle=angle+difmouse*1000;
bmousex = mouse.x;

```

donde difmouse y bmousex son variables privadas del proceso “character”. Pruébalo.

¿Y qué pasa si en cambio pones...?

```
difmouse = screenx/2-mouse.x;  
angle =angle+ difmouse*1000;
```

### El Modo7

El Modo7 es un tipo muy particular de scroll en el cual el plano de éste no es paralelo a la pantalla sino que está abatido, dando una sensación de profundidad que puede simular un efecto 3D.

Una limitación importante que sufre es que todos los gráficos relacionados con el Modo7 (tantos los gráficos de fondo scrolleables como los gráficos que se moverán sobre éstos) han de ser de 8 bits -y si se utiliza un contenedor FPG, éste ha de ser también de tipo 8 bits (en el FPGEdit se puede decidir eso en el momento de su creación). Recuérdalo.

Pero lo mejor es que veas un ejemplo para entenderlo. Para ello, nos tendremos que crear unos nuevos gráficos de 8 bits y meterlos en un fpg de 8 bits también llamado "m7.fpg". En concreto, dos gráficos: uno con código 001 de 640x480 que representará el fondo de scroll, y otro de código 002 y 30x30 que representará un objeto (una nave) colocada en un lugar concreto dentro de este scroll.

La idea es que aparecerá un scroll abatido, en el cual nos podremos mover como si tuviéramos una cámara en primera persona. También aparecerá un proceso inmóvil en un sitio concreto dentro de ese scroll.

```
program FenixEjemplo7;
```

```
global
```

```
    int id_nave;  
    int id_camara;  
    int idfpg;
```

```
end
```

```
begin
```

```
    set_mode(640,480,8);  
    idfpg=load_fpg("m7.fpg");
```

```
    //Definimos la región (centrada y más pequeña que la pantalla) sobre la que haremos el m7  
    define_region(1,0,0,640,480);
```

```
    //Este proceso lo situaremos dentro del scroll como un elemento más del paisaje
```

```
    id_nave=nave();
```

```
    //Este proceso es el que moveremos por el modo 7 y usaremos como visión
```

```
    id_camara=camara();
```

```
    /*Creamos el modo 7, con los parámetros:
```

```
    número           : 0 en nuestro caso. Puede haber hasta 10 scrolls Modo7 (de 0 a 9)
```

```
    fpg              : librería de la que cogemos el gráfico del scroll m7
```

```
    gráfico_interior : número del gráfico para el fondo del scroll m7
```

```
    gráfico_exterior : este gráfico bordeará al anterior, 0 para ninguno
```

```
    región           : región de pantalla donde se visualizará el m7, en nuestro caso la
```

```
1, que hemos definido antes, un poco más pequeña que la pantalla
```

```
    horizonte        : altura del horizonte (en pixels)*/
```

```
    start_mode7(0,idfpg,1,0,1,100);
```

```
    //La cámara del scroll Modo7 será el proceso "camara()"
```

```
    m7[0].camera=id_camara;
```

```

        loop
            frame;
            if(key(_esc)) break; end
        end
        let_me_alone();
end

process nave()
begin
    /*La variable local ctype indica el sistema de coordenadas del proceso.En este caso, está
"dentro del scroll" de ahí que le pongamos c_scroll para indicarlo*/
    ctype=c_m7;
    file=idfpg;
    graph=2;
    x=300; y=275;
    loop
        frame;
    end
end

process camara()
begin
    /*La variable local ctype indica el sistema de coordenadas del proceso.En este caso, está
"dentro del MODO7" de ahí que le pongamos c_m7 para indicarlo*/
    ctype=c_m7;
    x=150; y=100;
    loop
        if(key(_left)) angle=angle+5000; end
        if(key(_right)) angle=angle-5000; end
        if(key(_up)) advance(5); end
        if(key(_down))advance(-3); end
        frame;
    end
end
end

```

Podemos enumerar las diferencias que encontramos entre los scrolls ya conocidos y éste:

La función que inicializa el scroll pasa de ser **START\_SCROLL** a **START\_MODE7**.

El significado de los parámetros de ésta última son los mismos que el de la primera, excepto el 4º y el 6º.

\*En **START\_MODE7**, el cuarto parámetro también es un código de gráfico, pero a diferencia de **START\_SCROLL**, este gráfico no sirve para dibujar un segundo plano del fondo sino que sirve para dibujar un borde exterior infinito al gráfico principal (dado por el tercer parámetro).Es decir, el 4º parámetro sirve para extender el scroll de forma indefinida en todas direcciones, en forma de mosaico infinito. Prueba, en el ejemplo anterior, de asignar un 1 o un 2 a este parámetro y mira lo que pasa.

\*En **START\_MODE7**, el sexto parámetro sirve para definir la altura del horizonte, en píxeles.Es decir, indicará a cuántos puntos desde la parte superior de la ventana se quiere situar la línea del horizonte. Si la cámara se sitúa por encima del plano abatido, entonces no se mostrará nada por encima de la línea del horizonte (ese hueco se suele rellenar con otra ventana de scroll o de modo 7); en cambio si la cámara se sitúa por debajo del plano, entonces no se mostrará nada por debajo de la línea del horizonte. Según esté más o menos elevada esta línea, dará la sensación de que la cámara está mirando más "hacia el cielo" o

más “hacia el suelo”.

Las 10 estructuras posibles que almacenan la información de los 10 scrolls posibles, en vez de llamarse SCROLL[] se llaman M7[].

En el ejemplo se puede ver que esta estructura M7 también tiene un campo “camera”, cuyo significado es idéntico al que ya conocíamos, pero que en el caso del Modo7 es imprescindible establecer.

Para hacer que un proceso determinado pertenezca a un scroll Modo7, utilizamos también la variable CTYPE al igual que los scrolls ya conocidos, pero en el caso le asignamos el valor `c_m7`.

La variable CNUMBER tiene el mismo significado y valores posibles que siempre.

Campos específicos de la estructura M7[] -que no hemos visto en el ejemplo anterior- que no están presentes en la estructura SCROLL ya vista son:

\*”**height**”\*: Establece la altura vertical (en píxeles) del proceso que hace de cámara del scroll respecto al gráfico de fondo.

\*”**distance**”\*: Establece la distancia horizontal en píxeles de la cámara visual respecto al proceso que hace de cámara. Este proceso se situará siempre delante de la cámara real (a la distancia especificada), y será ésta la que será manejada por el usuario y seguirá los movimientos de ésta. Si sólo se quiere una cámara sin visualizar un proceso delante, ya que es obligatorio tener un proceso cámara, se puede hacer que su GRAPH valga 0 para que sea invisible.

\*”**camera.X**”\*: Coordenada X del proceso cámara dentro del gráfico de fondo del scroll

\*”**camera.Y**”\*: Coordenada Y del proceso cámara dentro del gráfico de fondo del scroll

\*”**focus**”\*: Focal de la visión

\*”**color**”\*: Número de color dentro de la paleta que será el color de fondo del Modo7 si no se define ningún gráfico como mosaico en el 5º parámetro de START\_MODE7.

\*”**horizon**”\*: Su significado es el mismo que el 6º parámetro de START\_MODE7, pero este campo nos permitirá, una vez generado el scroll, modificar la altura de la línea del horizonte allí donde nos convenga, dando una sensación de movimiento de cámara vertical.

Así mismo, campos que conocemos de SCROLL[] pero que no están presentes en M7[] son: “x0”, “x1”, “y0”, “y1”, “ratio”, “follow”, “flags1”, “flags2”, “region1”, “region2”.

El campo “z”, en cambio, está en los dos tipos de scroll con el mismo significado.

Veamos otro ejemplo similar al anterior, pero donde tendremos nuestra nave enfrente de la cámara siguiendo todos nuestros movimientos, y donde repetiremos hasta el infinito nuestro gráfico de fondo.

*PROGRAM Carse;*

*GLOBAL*

*int idy;*

*int idx;*

*int fpg;*

*int i;*

*end*

*BEGIN*

*fpg=LOAD\_FPG("m7.fpg");*

*set\_fps(30,2);*

*set\_mode(640,480);*

*start\_mode7(0,fpg,1,1,0,0);*

*M7[0].CAMERA=id; //El programa principal hará de cámara*

*M7[0].HEIGHT=25; //A cierta altura del fondo*

*M7[0].DISTANCE=150; //A cierta distancia del fondo original*

```

M7[0].CAMERA.X=50; //En una coordenada X concreta dentro del fondo original
M7[0].COLOR=166;
navecamara(320,240);
//Pongo otras 60 naves inmóviles en coordenadas al azar, incluyendo en el fondo repetido en mosaico
while (i<60)
    navenormal(rand(0,1000), rand(0,1000));
    i++;
end

LOOP
    IF (KEY(_esc)) exit(); end
    IF (KEY(_left)) angle=angle+5000;end //Muevo la cámara a la izquierda
    IF (KEY(_right)) angle=angle-5000;end
    IF (KEY(_up)) ADVANCE(10);end
    IF (KEY(_down)) ADVANCE(-4);end

//Si llego a una determinada coordenada Y o X, la cámara no puede avanzar más.
If(Y=>1000) Y=999; END
If(Y=<20) Y=21; END
If(X=>1000) X=999; END
If(X=<20) X=21; END

/*Estas dos líneas sirven para pasar la coordenada de la cámara a variables globales que luego usaremos
para establecer a su vez esas coordenadas a los procesos que queremos que la sigan*/
    idx=x;
    idy=y;
    frame;
END
END

PROCESS navenormal(x,y)
BEGIN
ctype=C_M7;
graph=2;
size=200;
LOOP
    frame;
END
END

Process navecamara(x,y)
Begin
ctype=C_M7;
graph=2;
Loop
//La posición de nuestra nave será la misma que la de la cámara.
    x=idx;
    y=idy;
    Frame;
END
END

```

A partir del ejemplo anterior, por ejemplo, podríamos implementar un sistema que detectara colisiones

entre los ejemplos “navecamara” y “navenormal”...

Piensa que si quieres poner un gráfico de fondo en la parte superior de la pantalla, a modo de “cielo”, lo puedes hacer con un simple `put_screen` o un `start_scroll` estándar. También podrías poner el gráfico de un proceso, pero entonces tienes que hacer es procurar que el proceso que muestre ese gráfico no pertenezca al scroll Modo7 (es decir, que no tenga la línea `ctype=c_m7`) y que tenga una *Z* superior a la del scroll Modo7, el valor de la cual es de 256..

Como último ejemplo, aquí presento un código que intenta simular la implementación de un scroll Modo7 “a pico y pala”, sin utilizar las `START_MODE7`, únicamente utilizando funciones básicas de distancias y dibujo de Fénix:

```
Program Modo7_en_fenix_para_fenix;
```

```
global
```

```
    int fpg;
```

```
end
```

```
begin
```

```
    Set_Mode(800,600,8);
```

```
    fpg= load_fpg("m7.fpg");
```

```
    AmosPaYa();
```

```
    loop
```

```
        if (key(_esc)) exit();end
```

```
    frame;
```

```
end
```

```
end
```

```
Process AmosPaYa()
```

```
private
```

```
    int dist,dx,dy;
```

```
    int x7,y7,z7=50,a7;
```

```
    int i;
```

```
end
```

```
begin
```

```
loop
```

```
    if (key(_up)) y7=y7+3; end
```

```
    if (key(_down))y7=y7-3;end
```

```
    if (key(_a)) z7=z7+1; end
```

```
    if (key(_z)) z7=z7-1;end
```

```
    if (key(_left))a7=a7+3000;end
```

```
    if (key(_right))a7=a7-3000;end
```

```
    dist=fget_dist(0,0,x7,y7);
```

```
    dx=get_distx(a7,dist);
```

```
    dy=get_disty(a7,dist);
```

```
    //aqui se pinta el grafico usando rotacion + zoom
```

```
    for (i=200;i<590;i=i+2);
```

```
        define_region(10,0,i,800,2);
```

```
    /*La línea clave está aquí: pinto el gráfico de código 001 del Fpg según los valores de las coordenadas x7,y7,z7 y a7 en aquél momento*/
```

```
        xput(fpg,1,400+dx,500+dy,a7,(((i-100)*2)*z7)/100,0,10);
```

```
    end
```

```
    frame;
```

```
end
```

```
end
```

## CAPÍTULO 10: TUTORIAL PARA UN RPG BÁSICO

(extraído del tutorial de Michael Sexton, en <http://www.div-arena.co.uk>)

Un RPG (Rol Playing Game) es un tipo de juego que parte de una localización concreta (un mundo más o menos fantástico) y de uno o varios personajes principales, -cada uno de ellos con cualidades concretas y diferentes (más fuerte, más ágil, etc),- que se encargarán de realizar diferentes misiones para lograr al fin encadenar una historia en la que se obtiene una recompensa final. Durante el camino se enfrentarán a enemigos y conocerán amigos, podrán incrementar sus cualidades o decrementarlas, utilizarán objetos que hayan recogido antes para realizar diferentes acciones, viajarán por el escenario explorando mundos nuevos, etc. Su nombre viene de que cada personaje en realidad es un Rol, un papel, diferenciado de los demás personajes, con sus propias características.

En este sencillo tutorial, lo que haremos será construir un personaje que se mueva por un escenario (un campo verde) en las cuatro direcciones cardinales. Dentro de él, situaremos dos casas, en las cuales el personaje podrá entrar y donde encontrará en una de ellas un personaje con el que podrá conversar, y en otra un objeto, una llave, que podrá recoger. También se incluirán enemigos que puedan acabar con la vida de nuestro amigo. Y ya está.

### La pantalla de inicio:

La pantalla de inicio del juego consistirá en un menú donde el jugador podrá elegir entre diferentes opciones: comenzar un juego nuevo y salir del juego, básicamente.

La resolución que escogeremos para la ventana de nuestro juego va a ser 320x200 píxeles; la puedes cambiar como quieras, pero ten en cuenta que las dimensiones de algunas imágenes también las tendrás que cambiar en consecuencia.

Crea los siguientes gráficos: un cursor (para el ratón) apuntando hacia la derecha de 40x40, y un fondo para la pantalla de inicio de 320x200. Mételos en un gráfico FPG llamado "rpg.fpg", con el código 005 y el 605 respectivamente.

Crea además dos tipos de fuente, con el FNTEdit. Una fuente para el título del juego llamada "titlescr.fnt" y otra para las palabras del menú inicial "Jugar" y "Salir" llamada "select.fnt".

El cuerpo esquelético de nuestro programa será algo así:

```
Program RPG;
Global
    Int file1;
    Int select_fnt;
    Int title_fnt;
    Int level=1;
End
Begin
    set_mode(320,200,16);
    set_fps(60,1);
    file1=load_fpg("rpg.fpg");
    select_fnt=load_fnt("select.fnt");
    title_fnt=load_fnt("titlescr.fnt");
    Loop
        Switch(level)
            Case 1:
```



```

    put_screen(file1,605);
    write(title_fnt,150,50,4,"Pon el título de tu juego aquí");
write(select_fnt,150,140,4,"Jugar");
    write(select_fnt,150,170,4,"Salir");
    Loop
        frame;
    End
End

```

Fíjate que las órdenes write no las he puesto dentro del LOOP, porque si no nos habría saltado el famoso error de “Hay demasiados textos en pantalla”.

Ok, ahora vamos a hacer el cursor de selección de las opciones del menú. Como es un gráfico, que además tendrá que controlar el jugador, crearemos un proceso para él. Lo primero será modificar el programa principal para que llame al proceso “cursor()”. Tendremos que hacer dos modificaciones: crear una nueva variable global llamada “selection”, con lo que la lista de variables globales hasta ahora sería:

```

Global
    Int file1;
    Int select_fnt;
    Int title_fnt;
    Int level=1;
    Int selection;
End

```

y modificar de la siguiente manera el Case 1:

Case 1:

```

    put_screen(file1,605);
    write(title_fnt,150,50,4,"Pon el título de tu juego aquí");
write(select_fnt,150,140,4,"Jugar");
    write(select_fnt,150,170,4,"Salir");

    cursor();

    Loop
        if(selection==1) level=2; break; end;
        if(selection==2) fade_off();exit("Gracias por jugar");end
    frame;
    End

    fade_off();
    delete_text(0); /*Si no borramos los textos, saldrán en los otros niveles*/
    clear_screen(); /*Si no borramos antes el fondo, cuando queramos poner otra
        imagen de fondo –en el nivel 2- nos va a dar error*/

    let_me_alone();
    fade_on();
End

```

Lo que hemos hecho aquí es crear el cursor y meternos en el LOOP a la espera de si el valor de la variable global “selection” llega a modificarse –de hecho, lo modifica el propio proceso cursor- a 1 o a 2. Esta variable representa la selección que hemos hecho con el cursor: 1 si queremos jugar y 2 si queremos salir, pero eso lo veremos ahora mejor en el código del proceso “cursor()”.

Fíjate de momento que si elegimos jugar, ponemos el valor 2 a la variable “level” y salimos del Switch. Es decir, salimos del switch diciéndole que cuando volvamos a entrar en él –porque el Switch está dentro de un LOOP- queremos ir al Case 2 (variable “level”).

Si elegimos salir, hacemos *con fade\_off()* un borrado progresivo de la pantalla: es un efecto estético muy eficaz; y luego acabamos la ejecución del juego mostrando un texto. Las líneas que hay después del LOOP/END, evidentemente sólo se ejecutarán si hemos elegido Jugar, porque con Salir directamente ya se acaba todo. Estas líneas lo que hacen es hacer un fade off también, como transición al segundo nivel, borrar todos los textos que hubiera en pantalla, quitar la imagen de fondo puesta con *put\_screen* mediante la función nueva *clear\_screen*, matar todos los procesos que hubiera corriendo excepto el actual (es decir, en este caso matar el proceso “cursor()” solamente porque no hemos creado ningún más todavía) y poner un fade on con la función nueva *fade\_on()* para acabar la transición hacia el segundo nivel.

Seguidamente, escribiremos el código del proceso “cursor()” al final del programa principal:

```

PROCESS cursor()
PRIVATE
  Int c;
END
BEGIN
  file=file1;
  graph=5;
  x=70;
  y=140;
  c=3; //Necesaria para el efecto de tembleque

  selection=0; //Ponemos un valor inicial neutral para “selection”

  WHILE (NOT key(_enter) AND selection==0)
    IF (key(_up)) y=140; END
    IF (key(_down)) y=170; END
    IF (key(_esc)) selection=2; END /*Sale fuera del bucle como si se hubiera escogido la opción de salir del juego */
    //Efecto tembleque horizontal
    x=x+c;
    IF (x>=80) c=-3; END
    IF (x<=70) c=3; END

  FRAME;
END

//Una vez que se ha apretado el enter, se comprueba dónde estaba el cursor
SWITCH (y);
  CASE 140: //Coord Y donde hemos escrito la palabra “Jugar” con el write
    selection=1;
  END
  CASE 170: //Coord Y donde hemos escrito la palabra “Salir” con el write
    selection=2;
  END
END
END

```

La verdad es que no deberías molestarte mucho en entender este código. Lo único que hace es que podamos mover el cursor arriba y abajo para señalar las dos opciones disponibles jugando con las

coordenadas de las variables locales X e Y, y que éste tenga un efecto de que parezca que tiembla horizontalmente. Cuando apretemos ENTER se parará y asignará el valor correspondiente de la opción elegida (1 o 2) a la variable global “selection”, que, como hemos visto, será usada en el Case 1 para saber, en el momento que cambie de valor, a qué nivel se va.

Ahora mismo tu programa debería ir al nivel 2 (nuestro mundo) cuando apretemos ENTER estando el cursor en la opción de Jugar, o debería de acabarse con un bonito fade off si se apreta ESC o se elige la opción Salir. Antes de testear tu programa, sin embargo, modifiquemos rápidamente el Case 2 para que cuando estés en el mundo puedas regresar a la pantalla de inicio apretando la tecla ESC:

Case 2:

```
Loop
    if(key(_esc))
        //Generar retardo
        while(key(_esc)) frame; end

        //Salir
        level=1;
        break;
    end
frame;
End
fade_off();
delete_text(0);
clear_screen();
let_me_alone();
fade_on();
```

End

Así, ahora, cuando estés en el mundo, si apretas ESC la computadora esperará un momento y entonces volverá a la pantalla de selección inicial. Pero, ¿por qué se pone ese WHILE dentro del IF que no hace nada? Pues para que mientras tengas pulsada la tecla ESC vayan pasando fotogramas (idénticos entre ellos) indefinidamente, creando así un estado de “pausa” hasta que dejes de pulsar la tecla ESC. Éste es un truco que se hace para que tengas la oportunidad de dejar de apretar la tecla ESC antes de poder hacer cualquier otra cosa como moverte a otro nivel: la computadora espera a que dejes de pulsar para continuar trabajando. Si no, lo que pasaría es que la computadora volvería enseguida a la pantalla inicial y posiblemente continuarías teniendo apretada la tecla ESC (el ordenador es más rápido que tu dedo), con lo que como en el nivel 1 si apretas ESC sales del juego, de golpe habrías acabado la ejecución del programa sin poder siquiera pararte en la pantalla inicial de selección.

El mismo código que hay en el Case 2 escríbelo en el Case 3 y el Case 4, el interior de las dos casas, para volver a la pantalla inicial. Deja el Case 5 sin tocar, ya que es la ventana de selección de ítems del juego.

Y la pantalla de selección ya está lista.

### **Creando el mundo y nuestro protagonista moviéndose en él:**

Primero haremos los gráficos:

-Haz un gráfico de 3000x3000 píxeles que sea un gran cuadrado verde, con algunas manchas marrones que representen montañas, o manchas verde oscuro que sean bosque, o lo que se te ocurra. Se pretende que sea el campo por el que nuestro personaje caminará. Guarda este gráfico en el

fichero FPG con código 300.

-Dibuja el protagonista, el chico/a que caminará por nuestro mundo. Este juego tendrá lo que se llama vista isométrica, así vamos a dibujar el personaje mirando hacia el norte, el sur y el este. No es necesario dibujarlo mirando al oeste porque haremos que el gráfico del este cambie de orientación, como en un espejo. Así que necesitarás tres gráficos para el chico. Asegúrate de que todos sean de 20x20 píxeles. Guarda los gráficos en el FPG así: el que mira al sur –hacia tí- con código 1, el que mira al norte con código 6 y el que mira al este –hacia la derecha- con código 10.

-Por cada dirección a la que mira nuestro personaje, necesitaremos crear tres dibujos similares para animar la caminata del protagonista en esa dirección cuando se mueva. Así que crea tres dibujos más por cada dirección ligeramente diferentes entre ellos, y guárdalos de tal manera que tengas cuatro imágenes mirando al sur con los códigos 1,2,3 y 4, cuatro imágenes mirando al norte con los códigos 6,7,8 y 9 y cuatro mirando al este con los códigos 10,11,12 y 13.

Ya tenemos los gráficos necesarios para este apartado.

Primero añadiremos dos variables más a la lista de variables globales del proceso principal, “MyChar\_Position\_X” y “MyChar\_Position\_Y”, y ya puestos las inicializaremos. Estas dos variables representarán la coordenada X e Y del personaje principal dentro del escenario:

*Global*

```
Int file1;  
Int select_fnt;  
Int title_fnt;  
int level=1;  
int selection;  
int MyChar_Position_X=1516;  
int MyChar_Position_Y=1549;
```

*End*

Y modificaremos el Case 2 del proceso principal, añadiendo las líneas que aparecen en negrita:

*Case 2:*

```
MyCharacter(MyChar_Position_X,MyChar_Position_Y);  
start_scroll(0,file1,300,0,0,0);  
Loop  
    if(key(_esc))  
        while(key(_esc)) frame; end  
        level=1;  
        break;  
    end  
frame;  
End  
fade_off();  
delete_text(0);  
clear_screen();  
let_me_alone();  
fade_on();
```

*End*

Fíjate lo que hemos hecho. Hemos creado un proceso MyCharacter, que será nuestro protagonista, con dos parámetros, que son su X e Y inicial. Además, hemos iniciado un scroll, (aunque

recuerda que al utilizar *start\_scroll* lo único que haces es decir que unas imágenes determinadas se usarán posteriormente para que hagan scroll, pero si no se hace nada más y sólo se escribe *start\_scroll*, no se moverá nada. De hecho, *start\_scroll* se puede usar como sustituto de la función *put\_screen* porque hasta que al scroll no se le dé ninguna consigna posteriormente, *start\_scroll* lo único que hará será poner la imagen que se le diga como parámetro de imagen de fondo).

Según los valores de los parámetros que hemos escrito en *start\_scroll*, éste será el scroll número 0; los gráficos del scroll se cogerán del archivo referenciado por "file1"; el gráfico en concreto del primer plano será el gráfico 300 (el mapa verde); como gráfico de segundo plano no habrá ninguno; el scroll existirá en una región que es la número 0 (es decir, la pantalla completa); y el número 0 como valor del último parámetro indica que el scroll no estará formado por ningún mosaico: es decir, que cuando se llegue a los límites del cuadrado verde no existirá nada más. Recuerda que si ese parámetro valiera 1, por ejemplo, indicaría que el scroll estaría formado por un mosaico horizontal de fondo: es decir, que cuando por la izquierda o la derecha el personaje llegara al límite de la imagen del cuadrado verde, éste se repetiría otra vez como en un mosaico. Puedes consultar el significado de los otros valores posibles en el apartado correspondiente del capítulo 9 de este manual.

¿Y por qué queremos crear un scroll? Porque la idea es que nuestro personaje se vaya moviendo por el escenario de tal manera que el mundo se mueva bajo sus pies según la dirección del movimiento del personaje. De hecho, la idea es que nuestro protagonista nunca deje de estar en el centro de la pantalla, y lo que se mueva es el escenario, dando el efecto de que viaje a través del mundo. O sea, queremos crear un scroll donde la cámara (a la que éste seguirá) será nuestro personaje.

Así pues, creamos el siguiente proceso para nuestro personaje:

*Process MyCharacter (x,y)*

*BEGIN*

```
ctype=c_scroll;
scroll[0].camera=id;
graph=1;
angle=0;
```

*Loop*

```
if (key(_right) and not key(_up) and not key(_down))
    x=x+2;
    if (graph<=10 or graph>=13) graph=10; end;
    graph=graph+1;
    flags=0;
```

*end*

```
if (key(_left) and not key(_up) and not key(_down))
    x=x-2;
    if (graph<=10 or graph>=13) graph=10; end;
    graph=graph+1;
    flags=1;
```

*end*

```
if (key(_up) and not key(_left) and not key(_right))
    y=y-2;
    if (graph<=6 or graph>=9) graph=6; end;
    graph=graph+1;
```

*end*

```
if (key(_down) and not key(_left) and not key(_right))
    y=y+2;
    if (graph>=3); graph=1; end
```

```

                                graph=graph+1;
                                end
                                frame;
                                End //Loop
End

```

Aquí hay muchas cosas que no hemos visto todavía, y hay que explicarlas con calma. Lo primero que vemos es la línea:

```
ctype=c_scroll;
```

que como sabes, indica cómo se tendrán que interpretar los valores de las coordenadas X e Y del gráfico del proceso en cuestión: si su valor es “c\_scroll”, en vez de ser la esquina superior izquierda de la ventana el origen de coordenadas a partir del cual se mide la X e Y del proceso, lo que se está diciendo al proceso MyCharacter es que los valores de sus variables X e Y se van a tomar respecto la esquina superior izquierda del primer gráfico del scroll (el cuadrado verde). En otras palabras: los movimientos de MyCharacter serán relativos respecto el scroll, no respecto la pantalla.

Y eso, ¿por qué? Porque será mucho más cómodo para nosotros programar teniendo en cuenta la posición de nuestro personaje respecto el mapa que no respecto a la pantalla. Por tanto, es mucho más inteligente tomar como referencia un origen de coordenadas del propio mapa para saber en qué sitio está nuestro personaje en cada momento, ya que todos los objetos que pongamos en el mapa (casas,etc) estarán colocados en unas coordenadas concretas dentro de él, así que si tomamos las coordenadas de nuestro personaje referenciadas al mapa, será mucho más fácil comparar coordenadas entre esos objetos y el protagonista y ver si éste choca contra una casa o no, por ejemplo.

Una puntualización que conviene recordar: CTYPE no es una variable de cadena, sino entera: el hecho de que pueda valer “c\_screen” (su valor por defecto) o “c\_scroll” viene de que en realidad estos dos valores son constantes enteras, también predefinidas: “c\_screen” es igual a 0 y “c\_scroll” es igual a 1.

Hay que decir que esta línea NO es opcional, porque es necesaria ponerla para que la línea que viene a continuación pueda funcionar bien. Y la línea que viene a continuación es muy importante:

```
scroll[0].camera=id;
```

Ésta es la línea que hace moverse al scroll, recuerda. Lo que hace es relacionar el movimiento del scroll con el movimiento de nuestro personaje, tal como nosotros queríamos. El efecto que veremos será que nuestro personaje va a quedarse inmóvil en el centro de nuestro juego y lo que irá moviéndose es el suelo, persiguiendo al personaje como una lapa.

El valor “id” es el código identificador del proceso MyCharacter. ¿Por qué? Porque recordad que ID es una variable local que contiene eso precisamente, el código identificador generado automáticamente por Fenix en la creación de ese proceso concreto. Así que lo que se está diciendo en el proceso MyCharacter es que el scroll coja como proceso a “perseguir” a ese mismo, al proceso MyCharacter.

Posteriormente a estas dos líneas, lo más digno a comentar son los diferentes ifs del movimiento del personaje. Hay cuatro ifs para las cuatro teclas del cursor. En cada uno se cambia las coordenadas X o Y del personaje (recordemos que respecto el mapa), y además, a cada pulsación del cursor se establece un nuevo gráfico dentro del rango adecuado para que de la impresión de animación al caminar. Es decir, que si hemos hecho que los gráficos del personaje mirando al este tengan los códigos del 10 al 13, a cada pulsación de tecla el gráfico irá cambiando: 10,11,12,13...y cuando llegue al final –gracias al ese if de una línea que hay- se volverá otra vez al 10, y así.

¿Para qué sirven las líneas *flags=0;* del primer if y *flags=1;* del segundo?. Recuerda que si la

variable predefinida local FLAGS vale 0, muestra el gráfico tal cual; y si vale 1 muestra el gráfico espejado horizontalmente. Es decir, si asignamos el valor de 1 a FLAGS lo que estamos haciendo es dibujar el gráfico del proceso “mirando hacia el lado contrario” del original, que es justo lo que queríamos para convertir nuestro personaje que originariamente mira al este en nuestro personaje mirando al oeste.

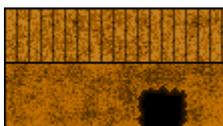
Y ya está. Deberías tener nuestro personaje moviéndose a través del cuadrado verde. Lo podrás ver mejor si añades al cuadrado verde manchas (montañas, ríos...).

### **El mapa de durezas:**

Un mapa de durezas es un dibujo especial que se superpone al dibujo del mapa (el cuadrado verde) – y por tanto, tiene las mismas dimensiones- pero que permanece invisible en pantalla. Este dibujo normalmente tiene dos colores: un color para casi todo el mapa y luego otro diferente que sirve para remarcar zonas especiales. Estas zonas especiales tendrán que coincidir con lugares concretos del mapa visible, como una casa, una montaña, etc; por lo que controlar las coordenadas de los objetos dentro del mapa es crucial. Y estas zonas especiales dentro del mapa de durezas sirven normalmente para indicar las zonas del mapa visible por donde el personaje no podrá caminar: emulan una barrera física. Quedaría muy feo dibujar una montaña y que nuestro personaje la traspasara como hace hasta ahora sin más. Por tanto, el mapa de durezas es un sistema para indicar que, por ejemplo, la montaña dibujada en el mapa visible corresponde a una zona especial en el mapa de durezas y por tanto es imposible atravesarla. La técnica para conseguir esto es sencilla: si en un momento determinado el ordenador detecta que el personaje tiene unas coordenadas X o Y que coinciden dentro del mapa de durezas con una zona especial (porque detecta que en ese sitio el mapa de durezas ha cambiado de color), entonces evita que dichas coordenadas X o Y varíen, y lo que verá el jugador simplemente es que al intentar cruzar un río, por ejemplo, el protagonista no lo puede pasar. Seguramente lo entenderás mejor cuando hagamos el código.

Como de costumbre, primero haremos los gráficos.

-En el mapa verde, dibuja una casa en vista isométrica (es decir, mostrando la fachada y sin perspectiva). Asegúrate que la coordenada de la esquina inferior izquierda de la casa sea (1440,1540) y que la coordenada superior derecha sea (1550,1480). También deberías dibujarle una puerta a tu casa. Graba el nuevo mapa con la casa dentro del FPG, con el mismo código 300. Como orientación, tu casa debería tener esta pinta:



-Ahora abre el mapa con la casa con tu editor de imágenes favorito. Repinta de un único color muy concreto y específico (por ejemplo, rojo) la casa entera, sin ningún reborde, y deja la puerta fuera, sin “sobrepintar”. El resto del mapa, si quieres, puedes hacer que sea de un mismo color uniforme sin nada más, negro por ejemplo. Tendría que quedarte, pues, un mapa enteramente negro (por decir un color) excepto un cuadrado rojo con una muesca, que sería la puerta. Guarda este nuevo mapa (que es el mapa de durezas), con el código 102.



Es muy importante la elección de un color fijo para el relleno de la casa, ya que la idea del uso del mapa de durezas es comprobar en cada momento si las coordenadas de nuestro personaje concuerdan

con las coordenadas donde en el mapa de durezas aparece ese color determinado, y reaccionar en consecuencia. Es decir, que haremos comparaciones de colores: “si en esta posición del mapa de durezas – donde está el personaje- está el color X, entonces quiere decir que se ha encontrado con la casa”. Por lo tanto, hay que tener perfectamente claro y codificado cuál va a ser el color que especifique la presencia de los objetos en el mapa, por que si no las durezas no funcionarán. Entonces, surge el problema de elegir el color. El color concreto no importa, puedes poner el que tú quieras, pero tendrá que ser siempre ése.

La función clave que realizará el proceso de comprobación del cambio de color en el mapa de durezas es *map\_get\_pixel*. Ya sabes que esta función lo que hace simplemente es “leer” el color que tiene un píxel concreto de una imagen en concreto. Cuando digo “leer” un color me refiero a que devolverá un código numérico que representará inequívocamente un color determinado. En el caso de gráficos de 16 bits, el valor devuelto es una codificación de las componentes del color (rojo, verde, azul) que depende de la tarjeta de vídeo y del modo gráfico, por tanto, ese código dependerá de cada ordenador donde se ejecute el juego (de hecho, el color podrá cambiar ligeramente de ordenador a ordenador). Así que en principio el valor devuelto por *map\_get\_pixel* no será fijo y esto podría ser un gran problema, pero veremos ahora cómo se puede solucionar.

Un truco para que con gráficos de 16 bits se puedan hacer mapas de durezas sin problemas es el siguiente: crear un gráfico auxiliar que sólo contenga puntos con los colores de referencia que hayas usado en el mapa de durezas. Al iniciar el programa (o el juego o cuando haga falta) leer de ese mapa cada punto y guardar el valor en una variable, que luego se usará para la comparación. De esta forma en vez de usar un valor absoluto sacado de tu ordenador concreto con *map\_get\_pixel* contra el mapa de durezas y compararlo a palo seco, se usaría el que se obtiene del ordenador en el que se está ejecutando el juego mediante el uso de *map\_get\_pixel* contra el gráfico auxiliar, de forma que si el color no es exactamente el mismo dará igual porque el programa funcionará de todas formas. Es como tener una pequeña biblioteca de píxeles en posiciones conocidas con colores conocidos, para usar con todos los mapas y gráficos que necesites.

Este truco es la mejor manera de no tener problemas. De todas maneras, en este tutorial no nos complicaremos más y como tampoco tenemos tantos objetos en nuestro mapa, no usaremos este gráfico auxiliar. Lo que haremos simplemente es recoger el código de color que identifica un obstáculo directamente del mapa de durezas con *map\_get\_pixel*, meterlo en una variable y luego, cuando nos interese, realizar la comparación de ese valor recogido –que he dicho cambiará de ordenador a ordenador según su tarjeta gráfica y demás- con el valor que devolverá (otra vez) *map\_get\_pixel* en la posición donde esté el personaje en ese momento. Cuando veamos el código lo entenderás mejor, seguramente.

Recuerda finalmente que esta función tiene cuatro parámetros: el código del archivo FPG cargado de donde se sacará el gráfico a usar, el código de ese gráfico dentro del FPG, la coordenada horizontal (X) del píxel del que se leerá su color y la coordenada vertical (Y) del mismo. Es importante recordar que los valores del 3º y 4º parámetro tienen como origen de coordenadas el propio gráfico, siendo su esquina superior izquierda el punto (0,0).

Bien, volvamos al juego. Una vez creados los gráficos, lo primero será añadir unas cuantas variables globales, todas enteras, al programa principal. A partir de ahora en el tutorial sólo se listarán los nombres de las nuevas variables que se añadan, que son:

```
despega_x  
despega_y  
obstacle  
colorObstaculo
```

Seguidamente, modificaremos el Case 2 del programa principal de la siguiente manera (los cambios son los marcados en negrita):

Case 2:

```
MyCharacter(MyChar_Position_X,MyChar_Position_Y);  
start_scroll(0,file1,300,0,0,1);  
colorObstaculo=map_get_pixel(file1,102,1500,1500);
```

Loop

```
IF(map_get_pixel(file1,102,Son.x+despega_x,Son.y+despega_y)==colorObstaculo)  
    obstacle=true;  
    ELSE  
    obstacle=false;  
    END  
  
    if(key(_esc))  
        while(key(_esc)) frame; end  
        level=1;  
        break;  
  
    end  
frame;  
  
End  
fade_off();  
delete_text(0);  
clear_screen();  
let_me_alone();  
fade_on();  
End
```

¿Qué hemos hecho aquí? Con el primer *map\_get\_pixel* lo que hacemos es obtener un código de color (el cual almacenaremos en la variable global “colorObstaculo”), que es el código que tendrá el píxel (1500,1500) de nuestro mapa de durezas. Este punto corresponde a un punto cualquiera del interior del cuadrado –rojo, en principio- que representa el obstáculo casa. Podríamos haber cogido cualquier otro punto que esté dentro del cuadrado rojo, porque lo que interesa es el color, que es el mismo en todos esos puntos.

Y la siguiente línea es la clave: comparamos el color de referencia obtenido con el anterior *map\_get\_pixel* con el color del píxel que corresponde en ese momento al centro del gráfico del protagonista. ¿Y eso cómo se sabe? De momento olvídate de las variables “despega\_x” y “despega\_y”: ¿qué es lo que tenemos como 3º y 4º parámetro del *map\_get\_pixel*? *Son.X* y *Son.Y*. Acuérdate que *Son* es una variable predefinida que referencia al último proceso hijo del proceso actual. Estamos en el proceso principal, y fíjate que tres líneas antes, dentro del Case 2, hemos creado el proceso *MyCharacter*. Es decir, que *Son* hará referencia a nuestro personaje. Y con *Son.X* y *Son.Y* estamos accediendo a las variables locales *X* e *Y* del proceso *Son*, es decir, del proceso *MyCharacter*. Por lo que este *map\_get\_pixel* lo que hace es lo dicho: coger el color del píxel que es el centro del personaje.

Esto se podría haber hecho de muchas otras maneras diferentes. Creo que esta es la más fácil. También sería posible, en vez de utilizar *Son*, haber hecho que en el momento de crear el personaje en el proceso principal, se recogiera el identificador de ese proceso en una variable, y utilizar esta variable en vez de *Son* en el *map\_get\_pixel*. La ventaja que tiene esto es que esa variable siempre contendrá el identificador de nuestro personaje, porque *Son* sólo lo tendrá hasta que el proceso principal cree un nuevo hijo –que se supone que no lo hará, pero a lo mejor en nuevas versiones del juego...- con lo que si aparece un nuevo hijo, *Son* dejará de referirse al protagonista y no funcionará nada. En seguida explicaré para qué sirven las variables “despega\_x” y “despega\_y”.

Cuando el color coincida, quiere decir que el personaje se ha chocado contra la casa, y en ese momento, lo único que hacemos es poner a “true” la variable “obstacle”. La idea es que el valor de esta variable global se compruebe de forma continua en el proceso *MyCharacter*, y en el momento que cambie a

“true”, el movimiento de nuestro protagonista quede interrumpido en la dirección adecuada, a pesar de pulsar las teclas del cursor.

Fijarse que el IF lo escribimos dentro un bucle infinito para que constantemente estemos recogiendo el nuevo valor del color presente por donde pisa nuestro protagonista, pero en cambio, el primer *map\_get\_pixel* lo escribimos fuera porque es fácil ver que sólo necesitamos asignar una vez el valor que queremos a la variable “colorObstaculo”.

Bueno, pero esto todavía no está acabado. Ahora, para que funcione, tendremos que modificar el proceso MyCharacter para que responda a los obstáculos que se encuentre (las novedades del código están en negrita):

```
Process MyCharacter (x,y)
BEGIN
```

```
  ctype=c_scroll;
  scroll[0].camera=id;
  graph=1;
  angle=0;
```

```
  Loop
```

```
    despega_x=0;
    despega_y=0;
```

```
    if(key(_right)) despega_x=2; end;
    if(key(_left)) despega_x=-2; end;
    if(key(_up)) despega_y=-2; end;
    if(key(_down)) despega_y=2; end;
```

```
    if (key(_right) and not key(_up) and not key(_down) and obstacle==false)
      x=x+2;
      if (graph<=10 or graph>=13) graph=10; end;
      graph=graph+1;
      flags=0;
```

```
    end
```

```
    if (key(_left) and not key(_up) and not key(_down) and obstacle==false)
      x=x-2;
      if (graph<=10 or graph>=13) graph=10; end;
      graph=graph+1;
      flags=1;
```

```
    end
```

```
    if (key(_up) and not key(_left) and not key(_right) and obstacle==false)
      y=y-2;
      if (graph<=6 or graph>=9) graph=6; end;
      graph=graph+1;
```

```
    end
```

```
    if (key(_down) and not key(_left) and not key(_right) and obstacle==false)
      y=y+2;
      if (graph>=3); graph=1; end
      graph=graph+1;
```

```
    end
```

```
  frame;
```

*End //Loop*

*End*

Lo más importante y fácil de entender es la modificación que se ha hecho a los cuatro ifs del movimientos del personaje. Simplemente se dice que sólo se moverá en la dirección especificada por el cursor si la variable “obstacle” es false. Si no, no. Es lo que he dicho antes. Si el personaje choca contra la casa, en el proceso principal se pone “obstacle” a true y por tanto, en el proceso personaje, aunque tecleemos el cursor, éste no se moverá.

¿Y para qué sirven las variables “despega\_x” y “despega\_y” que vuelven a aparecer aquí? Quítalas un momento del *map\_get\_pixel* del proceso principal (dejando el Son.x y el Son.y). Ejecuta el programa. Verás que parece que funciona bien cuando el protagonista choca contra la casa, pero una vez chocado, nos es imposible moverlo de allí: se ha quedado pegado, incrustado. ¿Por qué? Porque, tal como lo hemos hecho, al chocar contra la casa “obstacle” pasa a valer “true”, y por ello, los ifs del movimiento dejan de responder a las teclas del cursor, ya que “obstacle” no es false, y ya no podemos hacer nada. ¿Cómo podríamos hacer para que una vez que el personaje choque contra la casa, podamos seguir moviéndolo en las direcciones permitidas? Una alternativa sería esperar un plazo de tiempo después del choque y cambiar automáticamente el valor de “obstacle” de true a false otra vez como si no hubiera pasado nada. Pero la solución más fácil creo que es la que ha hecho aquí.

Recuerda que la comprobación de choque entre el personaje y la casa (el segundo *map\_get\_pixel*) se realiza constantemente. Bueno, pues la idea es que en realidad la comprobación del choque no se haga utilizando el punto central del personaje sino otro punto del personaje diferente según las circunstancias. Si pulsamos el cursor derecho, ese punto estará 2 píxeles a la derecha del centro, si pulsamos el cursor izquierdo, ese punto estará 2 píxeles a la izquierda del centro, si pulsamos arriba ese punto estará 2 píxeles arriba del centro y si pulsamos abajo, 2 píxeles debajo del centro. ¿Y por qué? Piensa, es un sistema muy ingenioso. Supongamos que nos acercamos a la casa desde su izquierda, es decir, pulsando el cursor derecho. Cuando el punto que está a 2 píxeles a la derecha del centro del personaje choque con la casa, según el Case 2 “obstacle” valdrá true y por tanto, tal como está en el proceso MyCharacter no nos podremos mover en ninguna dirección. Sin embargo, si pulsamos el cursor izquierdo, para “despegarnos” de la casa, vemos que la comprobación continúa de *map\_get\_pixel* en el Case 2 ahora se realizará con el punto que está a 2 píxeles a la izquierda del centro. Es evidente que este nuevo punto no estará chocando a la casa, porque el que choca es el que está a la derecha del centro. Por lo tanto, en ese momento “obstacle” volverá a valer false, y podremos ejecutar los ifs del movimiento otra vez. Y este sistema es extrapolable a las demás direcciones. ¿Y por qué 2 píxeles? Porque hemos hecho que el personaje se mueva de 2 píxeles en 2 píxeles (eso se puede cambiar, claro). Así, la detección del choque siempre se realizará siempre “un paso” (un movimiento) antes de efectuarlo realmente. No es la única solución, pero posiblemente sea la más sencilla de implementar.

Y ya está. A partir de ahora simplemente tendrás que dibujar con el color que hayas elegido aquellas partes del mapa donde no quieras que el protagonista pueda ir. Chequea que tu programa funcione: en estos momentos el protagonista no podría atravesar la casa. Si ves que el protagonista la atraviesa sólo un poquito, tendrás que ajustar la ubicación de la mancha correspondiente en el mapa de durezas. Requiere cierta práctica con el tema de las coordenadas, pero al final lo harás cuidadosamente.

### **Entrar y salir de la casa:**

Primero, los gráficos:

-Dibuja un gráfico de 320x200 que represente el interior de la casa, como tú quieras. Asegúrate de que la puerta esté en la parte superior de la imagen. Guardala dentro del FPG con código 401. Tiene que quedar algo similar –o mejor- a esto:



-Modifica tu mapa de durezas. Pinta ahora el hueco que dejaste sin pintar cuando rellenaste de rojo toda la casa (es decir, el hueco de la puerta de la casa) de otro color diferente al de la casa, por ejemplo, amarillo. Yo he dibujado la puerta de tal manera que el punto (1525,1535) está dentro de ella; si tú no lo tienes igual tendrás que cambiar estas coordenadas en el código que venga a continuación. Guarda el nuevo mapa de durezas otra vez dentro del FPG con el mismo código 102, claro.

Primero de todo, no te olvides de añadir dos variables globales enteras más a la lista: "house1" y "colorUmbral1". E inicializa "house1" con valor 0.

Tal como comenté hace tiempo, el Case 3 del programa principal será el nivel del interior de esta casa (en seguida haremos una segunda casa, que será el Case 4). Por lo tanto, lo primero que hay que hacer es decirle al Case 2 cuando saldremos de él para ir al Case 3. En el Case 2 tendremos que añadir lo siguiente (las novedades están en negrita, y hay código que no se muestra porque es irrelevante: se ausencia se denota por puntos suspensivos):

Case 2:

```

...
colorUmbral1=map_get_pixel(file1,102,1525,1535);
Loop
IF(map_get_pixel(file1,102,Son.x+despega_x,Son.y+despega_y)==colorObstaculo) ...
  END
IF (map_get_pixel(file1,102,Son.x+despega_x,Son.y+despega_y)==colorUmbral1)
  house1=true;
  END

  if(key(_esc))
    ...
  end
  if(house1==true)
    house1=false;
    level=3;
    MyChar_Position_X=160;
    MyChar_Position_Y=60;
    break;
  end
  frame;
End
...
//clear_screen();Con stop_scroll(0) ya no hace falta esta línea
stop_scroll(0);
End

```

Fíjate que el proceso que hemos escrito ahora es muy similar al de antes: utilizamos una vez el *map\_get\_pixel* para recoger en una variable (“colorUmbral”) el valor numérico del color presente en el mapa de durezas que representa la puerta, y una vez dentro del Loop, constantemente vigilarémos que nuestro personaje entre o no en contacto con dicha zona. En el momento que entra, fíjate que ponemos a 1 (“true”) la variable “house1”, y esto lo único que implica es que al final de esa misma iteración se ejecutará el último if. Lo que hace este if es, primero, volver a poner “house1” a 0 –la “resetea”- para la siguiente vez que se ejecute el Case 2 se vuelva al valor inicial de esa variable, se especifica con “level” el nuevo nivel al que queremos ir, ponemos unas nuevas coordenadas para nuestro protagonista (ya que en el interior de la casa la he dibujado arriba a la izquierda de la pantalla, y por tanto el personaje, como representa que aparecerá por esa puerta, se visualizará en un lugar de la pantalla completamente diferente de donde estaba dentro del mapa verde), y finalmente salimos del bucle infinito, para poder escapar del Case 2.

Pero fíjate que hemos añadido, (justo antes de ejecutar otra vez en el Switch el cual volverá a hacer la comprobación de la variable “level” y entrará pues en el Case 3) una última orden muy importante: *stop\_scroll*, con la que detendremos el scroll del mapa verde, pues si no se parara, al entrar en el nivel 3 se seguiría viendo el mapa verde porque el scroll –y por tanto, su imagen asociada: el mundo exterior- se seguiría ejecutando. Se puede entender que *stop\_scroll* es una función equivalente a *clear\_screen*, sólo que la primera afecta a los fondos escrolleables que han sido creados mediante *start\_scroll*, y la segunda a los fondos estáticos creados con *put\_screen*.

Antes de modificar el Case 3 para poder ver algo (si ejecutas ahora el programa verás que cuando entres en la casa no verás nada de nada porque no hemos ningún código visible en dicho Case), vamos a hacer otra cosa antes. Vamos a crear otro proceso llamado *MyCharacter\_in\_House*. ¿Por qué? Porque el proceso que hemos creado para mostrar nuestro protagonista en el mundo verde (“MyCharacter()”) no lo podemos utilizar para mostrarlo dentro de la casa debido a un pequeño detalle: la casa no tiene scrolling. Por eso tendremos que crear un proceso ligeramente diferente, y usarlo dentro de las casas y espacios sin scrolling.

Primero, añadiremos en el proceso principal dos nuevas variables globales enteras más, “increase\_x” e “increase\_y”, y luego escribimos esto:

```
Process MyCharacter_in_House(x,y)
```

```
BEGIN
```

```
  graph=1;
  angle=0;
```

```
  Loop
```

```
    despega_x=0;
    despega_y=0;
```

```
    if(key(_right)) despega_x=2; end;
    if(key(_left)) despega_x=-2; end;
    if(key(_up)) despega_y=-2; end;
    if(key(_down)) despega_y=2; end;
```

```
    if (key(_right) and obstacle==false)
      x=x+2;
      if (graph<=10 or graph>=13) graph=10; end;
      graph=graph+1;
      flags=0;
```

```
  end
```

```

        if (key(_left) and obstacle==false)
            x=x-2;
            if (graph<=10 or graph>=13) graph=10; end;
            graph=graph+1;
            flags=1;
        end

        if (key(_up) and obstacle==false)
            y=y-2;
            if (graph<=6 or graph>=9) graph=6; end;
            graph=graph+1;
        end

        if (key(_down) and obstacle==false)
            y=y+2;
            if (graph>=3); graph=1; end
            graph=graph+1;
        end

        frame;
    End //Loop
End

```

Fíjate que este proceso es muy parecido a MyCharacter, pero ligeramente diferente. Básicamente, el cambio que hay es el haber eliminado las dos líneas donde se establecían los valores para CTYPE y Scroll[0].Camera respectivamente.

A más a más, vamos a crear un nuevo mapa de durezas para el interior de la casa. Abre el dibujo 401 y pinta el umbral de la puerta que dará otra vez al mundo exterior del mismo color que tenga la puerta de la casa en el mapa de durezas que hicimos antes para el mundo exterior –en nuestro caso,el color amarillo, que viene referenciado por nuestra variable “colorUmbral1”- , y vuelve a pintar del mismo color que tenga la casa en el mapa de durezas del mundo exterior,- en nuestro caso, el color rojo, que viene referenciado por nuestra variable “colorObstaculo”- todas aquellas partes de la casa donde no quieras que tu personaje se quiera mover. Fíjate que hemos seguido un criterio: en nuestro ejemplo las partes rojas de todos los mapas de durezas que hagamos serán zonas prohibidas y las zonas amarillas serán zonas de transición a otros niveles. Guarda este nuevo mapa de durezas en el FPG con el código 106.

Ahora sí que podemos modificar el Case 3 del proceso principal (las novedades están en **negrita**):

Case 3:

```

put_screen(file1,401);
MyCharacter_in_House(MyChar_Position_X,MyChar_Position_Y);

    Loop
IF (map_get_pixel(file1,106,son.x+despega_x, son.y+despega_y)==colorObstaculo)
        obstacle=true;
    ELSE
        obstacle=false;
    END

IF (map_get_pixel(file1,106,son.x+despega_x,son.y+despega_y)==colorUmbral1)
        house1=true;
    ELSE

```

```

        house1=false;
    END
    if(house1==true)
        house1=false;
        level=2;
        MyChar_Position_x=1519;
        MyChar_Position_y=1546;
        break;
    end
        if(key(_esc))
            while(key(_esc)) frame; end
            level=1;
            break;
        end
    frame;
End
fade_off();
delete_text(0);
clear_screen();
let_me_alone();
fade_on();
End

```

Se parece bastante al Case 2: dibujamos un fondo, dibujamos el personaje, y a partir de entonces, comprobamos continuamente que el personaje no choca con ninguna zona prohibida. (Si chocara, pondría la variable “obstacle” a true, con lo que automáticamente los ifs de movimiento del proceso “MyCharacter\_in\_House()” no se ejecutaría). De igual manera, se mantiene el mismo truco de “despega\_x” y “despega\_y”. Y también se comprueba que el personaje no choque con la zona del umbral de la puerta, en cuyo caso, sale del bucle y retorna al nivel 2. Fíjate que en el final no hemos puesto la función *stop\_scroll* porque no hay ningún scroll que parar, pero sí que hemos puesto la función *clear\_screen*, que borra del fondo la imagen anteriormente dibujada con *put\_screen*. Comprueba el funcionamiento de tu programa. Deberías de ser capaz de entrar y salir de la casa volviendo al mundo exterior.

### **Conversaciones entre personajes:**

Éste será posiblemente el apartado más duro de este tutorial, porque la clave de un buen RPG está en conseguir conversaciones interesantes de las que el personaje pueda sacar algún provecho para su aventura. Haremos dos nuevos personajes en este RPG, uno le dará al protagonista una espada después de que el protagonista haya hablado previamente con otro carácter y le haya dicho una respuesta correcta.

Primero, los gráficos:

-Crea tres dibujos de 20x20 de un carácter mirando al norte, este y sur (similar a nuestro protagonista, pero sin tener que preocuparnos de la animación de caminar). Grábalos dentro del FPG así: el gráfico que mira al sur con el código 900, el que mira al norte con código 901 y el que mira al este con código 902. Como anteriormente, no es necesario hacer un gráfico mirando al oeste porque podemos decir simplemente que la computadora haga el reflejo del gráfico que mira al este. Este dibujo representará un mago.

-Haz lo mismo para el otro personaje pero guarda sus dibujos así: el que mira al sur con el código 911, el que mira al norte con el código 912 y el que mira al este con el código 913. Este dibujo será Bob.

-Crea un dibujo de 20x20 de un cursor apuntando hacia la derecha, y gráballo con el código 14.

-Crea, finalmente, una nueva fuente con el FNTEdit, tan pequeña como sea posible (8x8 ya vale) y grábala como "talk\_fon.fnt".

Bien. Primero crearemos los dos nuevos procesos que controlarán a los dos nuevos caracteres que acabamos de hacer. Primero, el mago:

```
Process Wizard(x,y)
```

```
Private
```

```
    Int id2;  
    Int idangle;  
    Int angletoChar;
```

```
End
```

```
Begin
```

```
    graph=900;  
    id2=get_id(TYPE MyCharacter_in_House);
```

```
    Loop
```

```
        idangle=get_angle(id2);  
        angletoChar=idangle-90000;
```

```
        if(angletoChar>-45000 and angletoChar<45000) graph=901; end  
        if(angletoChar>45000 and angletoChar>0) graph=902; flags=1; end  
        if(angletoChar<-45000 and angletoChar>-135000) graph=902; flags=0; end  
        if(angletoChar<-135000 and angletoChar>-195000) graph=900; end  
        frame;
```

```
    End
```

```
End
```

Y después, Bob:

```
Process Bob(x,y)
```

```
Private
```

```
    Int id2;  
    Int idangle;  
    Int angletoChar;
```

```
End
```

```
Begin
```

```
    graph=911;  
    id2=get_id(TYPE MyCharacter_in_House);
```

```
    Loop
```

```
        idangle=get_angle(id2);  
        angletoChar=idangle-90000;
```

```
        if(angletoChar>-45000 and angletoChar<45000) graph=912; end  
        if(angletoChar>45000 and angletoChar>0) graph=913; flags=1; end  
        if(angletoChar<-45000 and angletoChar>-135000) graph=913; flags=0; end  
        if(angletoChar<-135000 and angletoChar>-195000) graph=911; end  
        frame;
```

```
    End
```

*End*

En ambos procesos, el LOOP, con sus Ifs internos, para elegir qué gráfico en concreto se mostrará para ese nuevo personaje, dependiendo de la orientación que tengan respecto nuestro protagonista, de tal manera que siempre estarán de frente a él. Es decir, que cuando nuestro protagonista se mueva, tanto el mago como Bob corregirán su orientación para estar permanentemente mirándonos de frente. La clave está en dos funciones ya conocidas: *get\_id* y *get\_angle*.

Recuerda que *Get\_id* devuelve el identificador del primer proceso que encuentre que se corresponda con tipo especificado. Si no encontrase ninguno, esta función devolverá 0. En las siguientes llamadas a esta función que se realicen utilizando el mismo parámetro, *get\_id* devolverá sucesivamente los identificadores de los siguientes procesos de ese tipo que se encuentren en memoria, hasta que ya no quede ninguno, en cuyo caso devolverá 0. También sería posible llamar a *get\_id* con un parámetro de tipo de proceso 0. En este caso, *get\_id* devolverá los identificadores de todos los procesos en memoria, de cualquier tipo, siempre que no estén muertos. Esta función es muy práctica si a la hora de crear los procesos no recogimos en variables los identificadores que devolvían.

Y ya sabes que *Get\_angle* devuelve el ángulo formado por la línea que parte del centro del proceso actual y pasa por el centro del proceso cuyo identificador se especifica como parámetro, y la línea horizontal. El uso habitual de esta función consiste en hacer que un proceso "apunte" en dirección a otro: es el caso típico de monstruos y otros objetos que buscan siempre atrapar al jugador. Para que esta función devuelva un resultado útil, asegurarse de que las coordenadas de ambos representan la misma zona (por ejemplo, que no estén cada uno en un área de scroll diferente).

¿Para qué usamos estas funciones en el código anterior? Lo que primero hacen es conseguir el identificador del proceso de tipo *MyCharacter\_in\_House* (como sólo hay uno, no habrá confusión posible) y guardarlo en la variable privada "id2". Y con este identificador en el bolsillo, en cada iteración del Loop se comprobará el ángulo formado por la línea horizontal y la línea que en ese momento vaya del centro del gráfico del proceso mago/Bob al centro del gráfico del proceso *MyCharacter\_in\_House*. Después se hace una pequeña corrección, y el ángulo resultante será el que decida qué gráfico del mago/Bob (sur, norte, este, oeste) será el que se visualice en ese momento.

Bueno. Ahora nos falta incluir estos dos nuevos procesos en el interior de la casa. Por lo tanto, en el Case 3, JUSTO ANTES de la creación del proceso *MyCharacter\_in\_House*, creamos los dos procesos:

```
Wizard(250,105);  
Bob(130,100);
```

¿Por qué justo antes? Porque si te acuerdas, con las funciones *map\_get\_pixel* utilizamos la variable SON, que guarda el identificador del último proceso hijo del proceso actual. Como ese proceso hijo nos interesa que sea el *MyCharacter\_in\_House*, debemos dejar este proceso como el creado inmediatamente anterior a la aparición de *map\_get\_pixel*.

Bien. Cuando ejecutes el programa y entres en la casa deberías ver los dos nuevos personajes en ella y éstos permanecerán siempre de frente mirando a la dirección donde esté el protagonista. Pero no. ¿Por qué? Porque te he enredado. Si después de la línea *get\_id* en los procesos mago/Bob pones un write para ver el contenido de "id2", vale 0, y eso quiere decir que no se pilla bien el identificador de *MyCharacter\_in\_House*, y por tanto, ya no funciona todo lo que sigue. Y no pilla bien el identificador de *MyCharacter\_in\_House* porque tal como lo hemos hecho, en el Case 3 hemos creado el proceso mago/Bob antes de *MyCharacter\_in\_House*, y por tanto, cuando el mago/Bob comienza a ejecutarse encuentra un proceso que todavía no existe. Por tanto, la solución será poner en el Case 3 la creación del mago/Bob después de la de *MyCharacter\_in\_House*, cosa que nos obligará, en el Case 3, a cambiar las líneas donde aparecía la variable son (como parámetro de los *map\_get\_pixel*) por otra cosa. Esto es relativamente fácil. Lo único que tenemos que hacer es almacenar en una variable global el identificador del proceso *MyCharacter\_in\_House* cuando éste se crea, y utilizar esta variable global en vez de Son. Es mas, de esta

manera incluso no sería necesario utilizar la función *get\_id* en los procesos mago/Bob porque ya tendríamos esa variable global que nos serviría. Por tanto, crea una nueva variable global llamada “idCharHouse” y haz todos estos cambios que te he comentado. Y por fin, funcionará.

Aprovecharemos este razonamiento que hemos hecho ahora y nos curaremos en salud. Cuando se crea el proceso MyCharacter en el Case 2 haremos también que se recoja su identificador en una variable global, “idChar”, de tal manera que ahí también nos olvidemos de utilizar la variable Son. En estos momentos no sería imprescindible hacer esto, pero por si las moscas, hazlo ya.

Lo que falta es decirle a la computadora qué es lo que ha de pasar cuando nuestro protagonista se acerque a los demás personajes. Seguiremos el mismo sistema de siempre: modificaremos el mapa de durezas de la casa para identificar dos nuevas zonas, que serán el radio de acción de los dos personajes inmóviles. Cada una de estas zonas las pintaremos de un color diferentes a los ya existentes: propongo azul para el mago y verde para Bob.

Y , previa inclusión de cuatro nuevas variables globales enteras, “colorMago”, “colorBob”, ”near\_wizard” y “near\_bob”, añadiremos las siguientes líneas en el Case 3:

Case 3:

```

...
//Recogemos en colorMago el color de la zona del mago
colorMago= map_get_pixel(file1,106,250,105);
//Recogemos en colorBob el color de la zona de Bob
colorBob= map_get_pixel(file1,106,130,100);
Loop
...
IF (map_get_pixel(file1,106, idCharHouse.x+despega_x, idCharHouse.y+despega_y)==colorMago)
    near_wizard=true;
ELSE
    near_wizard =false;
END

IF(map_get_pixel(file1,106, idCharHouse.x+despega_x, idCharHouse.y+despega_y)==colorBob)
    near_bob=true;
ELSE
    near_bob=false;
END

...
End

...
End

```

En seguida veremos qué consecuencias tiene el que “near\_wizard” o “near\_bob” valgan true o false.

Ahora lo que haremos será crear un nuevo proceso que pueda manejar el texto que aparecerá en pantalla cuando uno de los personajes hable. Primero, añade a la lista las variables globales enteras : “talk\_font”, “talking”, “yes”, “no”, ”MyChar\_has\_KeyOne”, “talkboblines”, ”talkwizardlines” y ”asked\_bob\_for\_key” , y luego crea el siguiente proceso:

*Process TalkLines()*

*Begin*

*Loop*

```
talkwizardlines==1) if(key(_space) and near_wizard==true and asked_bob_for_key==false and
    talking=true;
    Write(talk_font,160,180,4,"¿Qué quieres? Fuera de aquí");
    //Retardo
    frame(5000);
    talking=false;
    delete_text(0);
end

talkwizardlines==1) if(key(_space) and near_wizard==true and asked_bob_for_key==true and
    talking=true;
    Write(talk_font,160,180,4,"Ah, Bob dice que quieres la llave. Bien,
    aquí la tienes");
    Frame(5000);
    talking=false;
    delete_text(0);
    MyChar_has_KeyOne=true;
    talkwizardlines=2;
end

talkwizardlines==2) if(key(_space) and near_wizard==true and asked_bob_for_key==true and
    talking=true;
    Write(talk_font,160,180,4,"Ya tienes la llave, así que fuera");
    Frame(5000);
    talking=false;
    delete_text(0);
end

talkboblins==1) if(key(_space) and near_bob==true and asked_bob_for_key==false and
    talking=true;
    Write(talk_font,160,150,4,"¿Quieres la llave?");
    Frame(5000);
    Write(talk_font,150,170,4,"Si");
    Write(talk_font,150,190,4,"No");
    talk_cursor();
    repeat
        frame;
    until(yes==true or no==true)
    talking=false;
    delete_text(0);
end

if(near_bob==true and yes==true)
    talking=true;
    Write(talk_font,160,180,4,"Pregunta eso al mago");
    Frame(5000);
    talking=false;
    delete_text(0);
```

```

        asked_bob_for_key=true;
        talkboblins=3;
        yes=false;
    end

    if(near_bob==true and no==true)
        talking=true;
        Write(talk_font,160,180,4,"Bueno, está bien...");
        Frame(5000);
        talking=false;
        delete_text(0);
        talkboblins=1;
        no=false;
    end

    if(key(_space) and near_bob==true and asked_bob_for_key==true and
talkboblins==3)
        talking=true;
        Write(talk_font,160,180,4,"No hay nada más que me hayan
programado para decir...");
        Frame(5000);
        talking=false;
        delete_text(0);
    end

    frame;
End //Loop
End

```

Ya ves que este proceso lo que hace básicamente es comprobar si se cumplen unas determinadas condiciones, dependiendo de las cuales el mago y Bob dirán una frase u otra, e incluso tendremos la opción de responder nosotros a una pregunta que nos hagan. La idea es que primero se hable con Bob y luego el mago nos dé una llave. Si se va al mago sin haber hablado con Bob y haberle respondido correctamente, no nos dará nada. El primer if, por ejemplo, comprueba si se ha pulsado la tecla SPACE (es el truco para que aparezca el texto que dice el mago: si quieres esta condición se puede quitar y así el texto aparecerá automáticamente cuando nos acerquemos a él), comprueba si efectivamente estamos cerca de él, si ya hemos hablado con Bob para poder conseguir la llave (en este caso, no todavía) y si es la primera vez que hablamos con el mago. La variable "talkwizardlines" sirve para controlar qué frase deberá decir el mago dependiendo de si ya se ha hablado con él previamente o no; digamos que es una variable que si vale 1, por ejemplo, el mago sólo dirá unas determinadas frases disponibles, si vale 2 las frases anteriores ya no se volverán a decir nunca más y se pasa "a otro nivel" de frases, y así. Seguramente se podría haber hecho de una manera más óptima, te lo aseguro. Si se cumplen todas esas condiciones, aparecerá una frase concreta que dice el mago, se espera un cierto retardo para que haya tiempo de leer la frase y, una vez acabado el tiempo, la frase desaparece. El retardo viene dado por la inclusión del parámetro numérico en la orden Frame; recuerda que este parámetro fijaba cuántos fotogramas podían pasar sin que se mostrara para el proceso en cuestión ningún cambio en pantalla. En este caso, poniendo un valor como 5000, lo que estamos haciendo es esperar un tiempo igual a 50 frames para que el código del proceso pueda continuar ejecutándose (durante esos 50 frames, evidentemente, los otros procesos que pudieran existir habrán ejecutado su código de forma completamente normal). Finalmente, el significado de la variable "talking" la veremos más adelante (fíjate que se pone a true antes de mostrar la frase y cambia a false cuando se ha dejado de mostrar).

Todos los ifs son más o menos iguales, excepto el cuarto If, que es cuando Bob pregunta al protagonista si quiere o no la llave, y el protagonista ha de decidir si responde "Sí" o "No". En ese If aparece la llamada a un proceso que no hemos visto todavía, "talk\_cursor()", y se introduce dentro de un bucle hasta

que demos una respuesta.

Hagamos, antes de acabar, unas pequeñas modificaciones necesarias en diferentes partes del programa. Primero, al final de todo del Case 1, escribe las siguientes líneas:

```
MyChar_has_KeyOne=false;
near_bob=false;
near_wizard=false;
talkboblins=1;
talkwizardlines=1;
asked_bob_for_key=false;
```

Esto es porque cuando en algún momento se vuelva al menú inicial – o al poner en marcha el juego-, los valores de estas variables se reseteen y valgan su valor inicial por defecto otra vez.

Y otra cosa que tendremos que cambiar es añadir al principio del programa principal, donde se cargan las fuentes, la línea que carga nuestra nueva fuente para los diálogos.

```
talk_font=load_fnt("talk_fon.fnt");
```

Y otra cosa que hay que hacer, es evidentemente, llamar al proceso "talklines()" en algún momento para que se pueda ejecutar. Coincidirás conmigo que el lugar más natural es al inicio del Case 3. Justo después de haber creado el proceso MyCharacter\_in\_House, Wizard y Bob, seguidamente añade:

```
talklines();
```

Finalmente, crearemos el código para el proceso "talk\_cursor" que vimos que se creaba cuando Bob nos preguntaba si queríamos la llave y nuestro protagonista se veía obligado a responder. Fíjate que básicamente es el mismo proceso que el del cursor de la pantalla inicial; sólo cambia el gráfico y las coordenadas. En este caso, lo que apuntará el cursor será a las dos posibles respuestas que podamos dar: "Sí" o "No".

```
PROCESS talk_cursor()
```

```
PRIVATE
```

```
  Int c;
```

```
END
```

```
BEGIN
```

```
  file=file1;
```

```
  graph=14;
```

```
  x=70;
```

```
  y=170;
```

```
  c=3;
```

```
  yes=false;
```

```
  no=false;
```

```
  WHILE (NOT key(_space) AND NOT key(_control) AND NOT key(_enter) AND no==false and yes==false)
```

```
    IF (key(_up)) y=170; END
```

```

    IF (key(_down)) y=190; END
    x=x+c;
    IF (x>=80) c=-3; END
    IF (x<=70) c=3; END
    FRAME;
    IF (key(_esc)) no=true; y=190; END
END

SWITCH (y);
    CASE 170: yes=true; END
    CASE 190: no=true; END
END
END

```

Y por último, explico el significado de la variable “talking”. Es buena idea hacer que nuestro personaje no se pueda mover mientras está conversando con los personajes. Así que el valor que contiene esta variable denota si en ese momento el personaje está hablando o no, y por tanto, podríamos utilizarla para inmovilizar al personaje en consecuencia. Lo único que tenemos que hacer para eso es modificar el proceso MyCharacter\_in\_House forzando a que los movimientos los haga (a más a más de las restricciones anteriores) siempre que “talking” valga false. Es decir, tienes que modificar lo siguiente:

```

Process MyCharacter_in_House(x,y)
...

if(key(_right) and talking==false) despega_x=2; end
if(key(_left) and talking==false) despega_x=-2; end
if(key(_up) and talking==false) despega_y=-2; end
if(key(_down) and talking==false) despega_y=2; end

if (key(_right) and talking==false and obstacle==false)
...
end

if (key(_left) and talking==false and obstacle==false)
...
end

if (key(_up) and talking==false and obstacle==false)
...
end

if (key(_down) and talking==false and obstacle==false)
...
end

...
End

```

Ahora, por fin, podrás tener la conversación soñada con los dos personajes. Esta conversación es sólo de ejemplo: modificala como creas oportuno.

### **Entrar y salir de la segunda casa:**

Lo único que vamos hacer ahora es añadir una nueva casa en el mapa verde, y hacer, en el Case 4, que podamos entrar en ella, siempre y cuando tengamos la llave que nos ha dado el mago.

Primero de todo, los gráficos:

-Dibuja otra casa en el mapa verde. Yo dibujaré esta segunda casa en la esquina superior izquierda del mapa verde, en el extremo. Vuelve a guardar el mapa con el mismo código, el 300.

-Modifica también el mapa de durezas de nuestro mundo. Ten encuenta una cosa muy importante: el color de la segunda casa en el mapa de durezas puede seguir siendo el mismo rojo (porque ambas casas son obstáculos igual), pero ahora el color que indica el umbral de la puerta para esta segunda casa ha de ser diferente del amarillo –el color que usamos en la primera casa-, porque si fuera igual, tal como hemos escrito el código, el personaje entendería que entra en la primera casa, y no en la segunda. Cuidado con esto. Así pues, elige otro color que no sea el amarillo –yo elijo el naranja- para indicar en el mapa de durezas que la puerta de entrada a la segunda casa es diferente. El código de ese color lo guardaremos cuando utilicemos el *map\_get\_pixel* en la variable global “colorUmbral2”, así que también tendrás que crear esta nueva variable. Vuélvolo a guardar con el mismo código de antes, el 102.

-Para el interior de la casa puedes utilizar el mismo dibujo que el de la primera casa, o bien otro dibujo. Nosotros no queremos trabajar mucho y usaremos el mismo dibujo, así que no tocamos nada y para la segunda casa seguiremos usando el dibujo número 401. Si tú quieres hacer otro dibujo, tendrás que usar otro código y modificar el programa.

-Crea un nuevo mapa de durezas para la segunda casa. Si usamos el mismo gráfico 401 para visualizar el interior de la casa, el mapa de durezas nuevo para esta segunda casa tendrá que tener la misma disposición espacial para los obstáculos, pintados en rojo. Lo que ha de cambiar respecto el mapa de durezas de la primera casa es el color con el que se denotará el umbral de la puerta de salida hacia el mundo, por el mismo motivo de antes. Si mantuviéramos el amarillo, el personaje saldría de la segunda casa pensando que sale de la primera y aparecería en el mundo verde al lado de ésta. Por tanto, hagamos que el color del umbral sea ahora otro. Mejor que sea el mismo que hemos utilizado en el mapa de durezas del mundo para indicar el umbral de entrada; o sea, naranja, y así podremos continuar utilizando la variable “color5”. Guarda este nuevo mapa de durezas con el código 103.

Añade la variable global entera “house2”.

Lo primero será modificar el Case 2 para que podamos entrar desde el mundo verde a la segunda casa. Justo antes del Loop del Case 2, escribe esto:

```
//Recoge el color de un punto del umbral de la puerta en el mapa de durezas
colorUmbral2=map_get_pixel(file1,102,125,55);
```

y justo antes de la línea *if(key(\_esc))* escribe esto:

```
IF      (MyChar_has_KeyOne==true      and      map_get_pixel(file1,102,      idChar.x+despega_x,
idChar.y+despega_y)==colorUmbral2)
        house2=true;
ELSE
        house2=false;
END
```

Fíjate en la condición que se pone: el personaje ha de tener la llave para poder entrar. Y fíjate también lo que te he dicho antes: si hubiéramos dejado “colorUmbral1” como color del umbral de la casa,

inmediatamente antes de este if que acabamos de poner está el if que comprueba si el personaje ha chocado o no con el umbral de la primera casa: si hubiéramos puesto el mismo color de umbral, el personaje no podría distinguir en qué casa entra.

Y en el mismo Case 2, justo después del bloque *if(house1==true)/end* –justo antes del último *frame*- escribe esto:

```

if(house2==true)
    house2=false;
    level=4;
    MyChar_Position_X=160;
    MyChar_Position_Y=60;
    break;
end

```

Evidentemente, ahora tendremos que modificar el Case 4 para que podamos entrar y salir de la segunda casa. El Case 4 quedará así, pues:

Case 4:

```

    put_screen(file1,401);
    idCharHouse=MyCharacter_in_House(MyChar_Position_X,MyChar_Position_Y);

```

Loop

```

IF(map_get_pixel(file1,103,idCharHouse.x+despega_x, idCharHouse.y+despega_y)==colorObstaculo)
    obstacle=true;
ELSE
    obstacle=false;
END

```

```

IF(map_get_pixel(file1,103,idCharHouse.x+despega_x,
idCharHouse.y+despega_y)==colorUmbral2)
    house2=true;
ELSE
    house2=false;
END

```

```

    if(house2==true)
        house2=false;
        level=2;
        MyChar_Position_X=300;
        MyChar_Position_Y=300;
        break;
    end

```

```

    if(key(_esc))
        while(key(_esc)) frame; end
        level=1;
        break;
    end

```

```

end
frame;

```

*End //Loop*

```
fade_off();  
delete_text(0);  
clear_screen();  
let_me_alone();  
fade_on();  
End
```

Un comentario final: en este tutorial ya no volveremos a retocar más el mapa de durezas, así que los cinco colores que tiene son los que tendrá. Aprovecho ahora, pues, para comentarte que sería una buena idea recopilar las cinco líneas esparcidas por todo el código fuente donde definíamos el color de cada una de estas partes (las líneas del tipo *colorX=map\_get\_pixel(...)*) y escribirlas juntas una detrás de otra, al inicio del programa principal, antes del bloque Loop/End. De esta manera, mantendrás una coherencia en el código porque sabrás que todas las asignaciones de colores para las distintas zonas del mapa de durezas se realizan al principio del juego, en el mismo sitio del código; y justo después de haber cargado el FPG es el lugar ideal. El código mejorará en claridad y comodidad. Y de hecho, incluso podrías hacer otra cosa más, que es crear un nuevo gráfico que sirviera simplemente como tabla de muestras de colores, y que a la hora de asignar los valores a la variables “colorX” se cogieran de allí en vez del mapa de durezas. Con esto ganarías facilidad a la hora de cambiar colores y/o posiciones de durezas dentro del mapa.

### **La pantalla del inventario:**

En este apartado vamos a darle a nuestro personaje una espada y hacer que la pueda usar cuando pulsemos ENTER.

La idea (no tiene por qué ser la mejor) es que en cualquier momento del juego, si el jugador pulsa la tecla “I”, ha de aparecer en la parte superior de la pantalla una zona en forma de barra que representará el inventario del protagonista. Este inventario sólo tendrá un hueco para incluir –o no- la espada; ya verás que es fácil hacer que pueda tener más huecos para incluir más elementos, pero lo tendrás que hacer por tu cuenta. La barra aparecerá por encima del gráfico del juego que estuviera visualizándose en ese momento (o sea, el personaje en el mundo verde o dentro de alguna casa), sin borrarlo pero dejándolo congelado, hasta que el jugador vuelva a pulsar “I”, momento en el cual la barra superior desaparecerá, dejando otra vez toda la pantalla para nuestro personaje y dotándolo otra vez de movimiento y vida.

Vamos a hacer que nuestro personaje desde el principio del juego tiene la espada ya en su inventario. Y vamos a hacer que para que la pueda usar, el jugador tenga que forzar la aparición de la barra del inventario y seleccionar la espada de allí. Haciendo esto, el gráfico de nuestro personaje cambiará en el mundo y se le verá manejando la espada, y el inventario en ese momento tendrá que aparecer vacío. A partir de entonces, si se vuelve a apretar “I”, se podrá seguir jugando con la espada “desenfundada”. Cuando el jugador desee, pulsará “I” otra vez y podrá devolver la espada al inventario haciendo que el protagonista “enfunde” la espada.

Primero, los gráficos:

-Haz un fondo blanco para la barra del inventario, de 320x40 píxeles. Guardala con el código 216.

-Crea una imagen de 20x20 píxeles y dibuja un cuadrado gris vacío en ella. Guárdala con el código 217. Este cuadrado gris representará el –único- hueco del que dispondrá nuestro inventario; es decir, representa la falta del único elemento que podremos incluir (la espada). De hecho, en el interior de este cuadrado podrá aparecer el ítem espada, o no, dependiendo de si en ese momento el protagonista la está utilizando. Si quisieras que en el inventario pudieran haber más elementos (“items”), tendrías que modificar el código que verás próximamente para añadir más imágenes de este cuadrado gris, denotando así que hay sitio para más objetos dentro de él.

-Crea una imagen idéntica a la anterior, pero con una espada dentro. Guárdala con el código 218. Éste será el gráfico que se mostrará en sustitución del anterior si la espada viene incluida dentro del inventario del personaje (es decir, si el hueco no está vacío).

-Crea una imagen idéntica a la anterior, pero con el cuadrado de color azul. Guárdala con el código 221. Éste será el gráfico que se muestre cuando apretemos la tecla ENTER en la pantalla del inventario, y denotará que hemos elegido el objeto del interior del hueco –la espada- para cogerla y usarla en el juego

-Crea cinco imágenes de nuestro protagonista mirando hacia el sur (20x20 también), empuñando la espada. En cada imagen la orientación de la espada ha de ser diferente, dando la sensación en conjunto de estar moviendo la espada de derecha a izquierda y viceversa. Guarda estas imágenes con los códigos de 700 a 704.

-Haz lo mismo que en el punto anterior pero con el personaje mirando al norte. Guarda estas imágenes con los códigos del 705 al 709. Y si quieres también, aunque yo no lo haré, puedes hacer lo mismo pero mirando hacia el este.

Lo siguiente que haremos, antes de continuar, es añadir a la lista una cuantas variables globales enteras más, que usaremos a lo largo de todo este apartado:

```
to_item_select_screen
last_level
direction
kill_mode
sword_selected
```

Y también añadiremos una variable global entera más, pero que deberá ser inicializada a true: “sword\_in\_inventory”; porque recuerda, y esto es muy importante, que si cuando creamos una variable entera, no la inicializamos (como las cinco anteriores), Fénix automáticamente la inicializará a 0, o lo que es lo mismo, a falso. Acuérdate de esto si no quieres tener sorpresas

Una vez creados los gráficos y declaradas (e inicializadas en su caso) las variables globales necesarias, empecemos con el código a escribir. Lo primero que haremos es poner en el Case 2, Case 3 y Case 4 que para ir al Case 5 ( y mostrar así la pantalla del inventario por encima de las otras) habrá que pulsar la tecla “I”.

Modificaremos los Cases así. En el Case 2:

*Case 2:*

```
if (to_item_select_screen==false)
    idChar=MyCharacter(MyChar_Position_X,MyChar_Position_Y);
    start_scroll(0,file1,300,0,0,0);
    color=map_get_pixel(file1,102,1500,1500);
    colorUmbral1=map_get_pixel(file1,102,1525,1535);
    color5=map_get_pixel(file1,102,125,55);
end

to_item_select_screen=false;

Loop
```

```

...
    if (house2==true)
        house2=false;

        level=4;
        MyChar_Position_X=160;
        MyChar_Position_Y=60;
        break;
    end

    if (key(_i))
        level=5;
        break;
    end
end
frame;
End

```

```

if (level<>5)
    fade_off();
    delete_text(0);
    let_me_alone();
    stop_scroll(0);
    fade_on();
end

last_level=2;

```

End

En el Case 3, haremos los mismos cambios:

Case 3:

```

if (to_item_select_screen==false)
    put_screen(file1,401);
    idCharHouse=MyCharacter_in_House(MyChar_Position_X,MyChar_Position_Y);
    Wizard(250,105);
    Bob(130,100);
    talklines();
    colorMago= map_get_pixel(file1,106,250,105);
    colorBob= map_get_pixel(file1,106,130,100);
end

to_item_select_screen=false;

```

Loop

```

...
    if (house1==true)
        house1=false;

        level=2;
        MyChar_Position_x=1516;
        MyChar_Position_y=1549;
        break;
    end
end

```

```

if (key(_i))
    level=5;
    break;
end

if(key(_esc))
    while(key(_esc)) frame; end
    level=1;
    break;
end

frame;
End

```

```

if (level<>5)
    fade_off();
    delete_text(0);
    clear_screen();
    let_me_alone();
    fade_on();
end

last_level=3;

```

**End**

Y en el Case 4, igual:

**Case 4:**

```

if (to_item_select_screen==false)
    put_screen(file1,401);

    idCharHouse=MyCharacter_in_House(MyChar_Position_X,MyChar_Position_Y);
end

to_item_select_screen=false;

Loop
    ...
    if(house2==true)
        house2=false;

        level=2;
        MyChar_Position_X=170;
        MyChar_Position_Y=100;
        break;
    end

    if (key(_i))
        level=5;
        break;
    end

```

```

        if(key(_esc))
            while(key(_esc)) frame; end
            level=1;
            break;
        end

        frame;
    End //Loop

    if (level<>5)
        fade_off();
        delete_text(0);
        clear_screen();
        let_me_alone();
        fade_on();
    end

    last_level=4;
End

```

End

Antes de explicar los cambios que hemos introducido en los tres Case anteriores, que son muy sutiles e importantes, voy a escribir a continuación cómo ha de ser el código del Case 5, para que puedas ver de una sola vez todas las partes del código de los diferentes Cases que están relacionadas entre sí. Así que el Case 5 ha de ser así:

Case 5:

```

delete_text(0);/*Si en la pantalla queda algún diálogo visible,que se borre*/
to_item_select_screen=true;

item_select_screen_background();
signal(idChar,s_freeze);
signal(idCharHouse, s_freeze);

if (sword_in_inventory==true)
    item_select_cursor();
else
    item_select_cursor_empty();
end

frame(2000);

Loop
    if (key(_i))
        level=last_level;
        break;
    end
    frame;
End

signal(type item_select_screen_background,s_kill);
signal(type item_select_cursor,s_kill);
signal(type item_select_cursor_empty,s_kill);

```

```
signal(idChar,s_wakeup);
signal(idCharHouse,s_wakeup);
```

```
frame(2000);
```

End

Comencemos. ¿Qué hemos modificado en los Case 2,3 y 4? Lo que es más evidente de entender es que se ha incluido la posibilidad dentro del Loop –para que se vaya comprobando constantemente- de que se pulse la tecla “I” para salir del bucle e ir al Case 5.

Ahora miremos el código del Case 5. Fíjate que para salir de este nivel también se tendrá que pulsar la tecla “I”. Cuando ocurra esto, se le asigna a la variable “level” –la que se usa en la condición del switch para saber a qué nivel se va- el valor de otra variable, “last\_level”. ¿De dónde ha aparecido esta nueva variable? Fíjate que otra de las modificaciones de los Case 2,3 y 4 es que justo antes de salir de ellos guardamos en esta variable el nivel donde estamos (puede ser el nivel 2,3 o 4). De esta manera, cuando queramos salir del Case 5, “last\_level” valdrá el nivel donde estábamos antes de entrar en la pantalla del inventario, y por tanto, al salir del inventario sabremos a qué nivel tenemos que ir.

También salta a la vista los varios retardos que hemos introducido en el Case 5, para que el dedo tenga tiempo de soltar las teclas (“I”,etc)de forma adecuada.

En los Case 2,3 y 4 hay dos modificaciones más. Al principio, la inclusión de la creación de fondos y la de los distintos procesos dentro del bloque *if (to\_item\_select\_screen==false)*; y al final, la inclusión de las últimas líneas de los Case dentro del bloque *if(level<>5)*. Estas modificaciones requieren de un estudio profundo para saber su razón de ser.

Primero miremos para qué sirve la variable “to\_item\_select\_screen”, inicializada en principio a false. Vemos que esta variable toma el valor true nada más entrar en el Case 5, siempre. Y que una vez salido de éste, continúa manteniendo su valor. Por tanto, esta variable no es más que una marca, una señal que indica que si vale true se ha pasado por el Case 5 al menos una vez. Si valiera false, en principio podemos pensar que no se ha entrado en la pantalla del inventario todavía. ¿Y para qué sirve saber si se ha entrado o no en el inventario?

Recordemos lo que queríamos hacer: poner la pantalla del inventario en forma de barra por encima de la pantalla que estuviera visible en ese momento, en la parte superior de la ventana. Fíjate qué es lo que ponemos dentro del if: básicamente la creación de los procesos (personaje, personaje\_en\_casa, mago,bob...). Si no se ha entrado en el inventario, se crean normalmente, pero si hemos entrado, no se crearán. ¿Por qué? ¿Porque ya están creados? Piensa un poco. Cuando estemos en la ventana del inventario, los procesos que estuvieran en ese momento visible continuarán estándolo, tal como hemos dicho. Así pues, cuando salgamos del Case 5 porque hemos “cerrado” la pantalla del inventario y queramos seguir jugando allí donde estábamos, lo que hará nuestro programa es volver al Case adecuado (2,3 o 4) y volver a ejecutar su interior. Si no pusiéramos este if, por tanto, se volverían a crear otra vez los procesos que no han sido destruidos, por lo que tendríamos dos personajes, o dos personajes\_en\_casa, dos magos, etc... Y así, cada vez que entráramos en el inventario, crearíamos nuevos procesos. Prueba de quitar el if y verás lo que te comento más claramente.

Una vez evitado el escollo de crear nuevos procesos que no deben crearse, fíjate que hemos vuelto a poner en los tres Cases el valor de “to\_item\_select\_screen” a false, para volver a empezar como si no hubiera pasado nada. Si no, la variable “to\_item\_select\_screen” valdría siempre true y nos podría dar problemas.

¿Y para qué sirve el otro if, el del final? Fíjate qué es lo que hemos incluido en este if: el fade\_off, el borrado de cualquier texto que hubiera, el borrado de la imagen de fondo –ya sea estática mediante *clear\_screen* o escrolleable mediante *stop\_scroll*- el fade on y, muy importante, la función

*let\_me\_alone*. Esta función es de vital importancia para entender lo que estamos haciendo, y hasta ahora no le habíamos prestado ninguna atención y ni tan sólo sabíamos por qué la poníamos. Ya es hora de remediarlo.

Esta función está en los tres Cases:(2,3 y 4). La tarea de este comando es matar todos los procesos excepto el proceso que lo llama, que en nuestro caso es el programa principal. Piensa un poco: si estuviéramos ejecutando el código del Case 2, ¿qué procesos habría activos en ese momento? Aparte del proceso principal, es evidente que los procesos que hubiéramos creado en el propio Case 2, es decir, “MyCharacter()”. Por lo tanto, si salimos del Case2, con *let\_me\_alone* mataríamos el proceso MyCharacter. Haciendo un análisis similar podríamos ver que saliendo del Case 3 mataríamos los procesos MyCharacter\_in\_House(), ”Wizard()”, ”Bob()” y “Talklines()”; y saliendo del Case 4 mataríamos el proceso “MyCharacter\_in\_House()”. ¿Y por qué hacemos esto? Está claro: si entramos en un nivel, no nos interesa para nada tener los procesos de otro nivel funcionando: si estamos en el mundo verde no queremos para nada el mago, ¿verdad? Y además, haciendo esto nos facilitamos las cosas porque como habremos destruido al salir los procesos, al regresar a ese nivel los volvemos a crear y ya está. De hecho, podrías probar el quitar esta línea y verás que cada vez que volvamos a un nivel donde ya hubiéramos estado antes nos aparecería un nuevo personaje, o personaje\_en\_casa, o mago,etc...porque no habríamos matado el anterior.

Pues bien, la explicación del bloque *if(level<>5)* está clara. Queremos que ocurra el proceso que he descrito en el párrafo anterior siempre que nos movamos entre los Case 2,3 y 4 (que entremos a o que salgamos de la casa 1 o 2 al mapa verde y viceversa), pero no cuando desde cualquier de estos niveles vayamos al nivel 5. ¿Por qué? Porque como hemos dicho que pondremos una barra superior en la ventana pero que la pantalla seguirá viéndose tal como estaba, no podremos matar los procesos “MyCharacter()” y compañía, porque si no lo que se vería sería la barra, sí, pero sobre un fondo negro. Por tanto, siempre que vayamos a la pantalla del inventario, no mataremos los procesos existentes ya que queremos que continúen en la pantalla.

No obstante, lo que sí que haremos será que estos procesos, aunque visibles, no se puedan mover ni puedan responder a ninguna acción del jugador, mientras la ventana del inventario esté activada (quedaría muy feo): haremos un “pause” del juego. ¿Y ésto cómo lo hacemos? Podríamos utilizar el mismo sistema que hicimos para las conversaciones entre el protagonista y el mago y bob: hacer que mientras el valor de una variable (en aquel caso, “talking”) sea uno determinado, las condiciones de los ifs del movimiento hagan que el personaje no se pueda mover. No obstante, ahora usaremos otra técnica un poco más sofisticada. Lo que haremos será que cuando entremos en el Case 5 (la pantalla del inventario activada), congelemos los procesos susceptibles de moverse, es decir, “MyCharacter()” y “MyCharacter\_in\_House()”. Acuérdate que congelar significa que el gráfico de ese proceso permanecerá en pantalla pero que su código no se ejecutará y por tanto permanecerá impasible, hasta que “se despierte”. Y acuérdate que congelar y despertar lo podíamos hacer con la orden *signal*. Acuérdate de que como primer parámetro de esta función podíamos poner o bien el identificador concreto de un proceso, o bien indicar los procesos que fueran de un tipo determinado. Pues bien, si te fijas, eso es lo que he hecho en el Case 5: congelar los dos procesos, y luego, antes de salir del Case 5, despertarlos otra vez.

Un detalle, el proceso “MyCharacter()” solamente estará activo en el Case2 y “MyCharacter\_in\_House()” lo estará en el Case 3 y el 4. Dentro del Case 5 podría haber escrito un if que comprobara el valor de “last\_level” y si éste era igual a 2, hacer que se congelara sólo MyCharacter, y si éste era igual a 3 o 4, hacer que se congelara sólo MyCharacter\_in\_House. No obstante, Fenix no da ningún error cuando se le dice que se envíe una señal a un proceso inexistente...así que nos despreocupamos de este hecho.

Falta todavía por repasar qué es lo que hace en definitiva el Case 5, aparte de congelar y descongelar los procesos existentes en el nivel de donde se procede. Lo primero que ejecuta es un proceso creado por nosotros también llamado “*item\_select\_screen\_background()*”. Su código es el siguiente:

```
Process Item_Select_Screen_Background()
```

```
begin
```

```
    graph=216;
```

```
    X=160; //Fondo centrado
```

```
    z=-5;
```

```
    loop
```

```
        frame;
```

```
    end
```

```
end
```

Puedes comprobar fácilmente que lo único que hace este proceso –de hecho, la razón de su existencia- es poner un gráfico, que será una barra visible en la parte superior de la ventana, la cual representará la pantalla del inventario. Como queremos que esta barra se vea encima del gráfico de fondo visible hasta ahora (el mapa verde, alguna casa), hemos de especificarle un valor de la variable Z adecuado. Fíjate que este proceso es infinito (el bucle no tiene ninguna condición de finalización), por lo que en principio esta barra, una vez que se vea, se verá hasta la finalización del juego. Para que esto no ocurra, fíjate que en Case 5, después de que el jugador haya pulsado “I” para hacer desaparecer el inventario de su vista, se le envía una señal S\_KILL a este proceso para matarlo.

Seguidamente nos encontramos con un bloque if/else que comprueba si el valor de la variable “sword\_in\_inventory” es true o no. Si lo es, se ejecutará otro proceso creado por nosotros, “item\_select\_cursor()”, y si no lo es, se ejecutará el “item\_select\_cursor\_empty()”. Acuérdate de que la variable “sword\_in\_inventory” la inicializamos a true por defecto. Esta variable indicará si la espada está (true) o no (false) en el inventario. Es decir, que si vale true, deberá aparecer en la barra del inventario un gráfico indicando que la espada está disponible para poder cogerla, y si vale false, la barra del inventario aparecerá vacía, y –esto lo veremos en seguida- nuestro protagonista deberá aparecer empuñando la espada. Como puedes ver, hemos dicho a la computadora que tienes ya la espada de entrada, pero puedes poner esto a falso, y entonces hacer algo en el juego para que sea verdadero (en otras palabras, puedes conseguir la espada en una tienda o similar).

Así pues, si la espada está en el inventario, se ejecuta “item\_select\_cursor()”. Si no, “item\_select\_cursor\_empty”. El primer proceso es simplemente una manera para mostrar el gráfico de la espada dentro de la barra del inventario (evidentemente, si la espada no estuviera, no deberíamos mostrar su correspondiente gráfico en el inventario, por lo que no deberíamos ejecutar este proceso). Una cosa muy importante que debes tener presente es que como en este tutorial sólo tenemos un ítem en el inventario, el código de este proceso es bastante trivial, pero si quieres tener más elementos, deberás de jugar con las coordenadas de cada ítem en la barra, y utilizar los cursores para poder seleccionar el ítem adecuado, comprobando si está o no está en ese momento... Por lo tanto, el código de este proceso puede complicarse un poco. El código de este proceso es el siguiente:

```
Process Item_select_cursor()
```

```
Begin
```

```
    graph=218;
```

```
    x=10;
```

```
    y=10;
```

```
    z=-15;
```

```
    loop
```

```
        if(key(_enter) and graph=218)
```

```
            graph=221;
```

```
            sword_selected=true;
```

```
            frame(1000);
```

```
        end
```

```

        if(key(_enter) and graph=221)
            graph=218;
            sword_selected=false;
            frame(1000);
        end
    end
    frame;
end
End

```

Puedes comprobar que si se apreta la tecla ENTER, el cuadrado que rodea al gráfico de la espada (el cursor de selección, que en este caso siempre seleccionará al único ítem existente) cambia de apariencia. Esta tecla servirá pues para seleccionar un ítem y poder utilizarlo en el juego. En seguida lo veremos.

El otro proceso lo único que hace es mostrar el mismo cuadrado gris que antes incluía la imagen de la espada, pero ahora gris. Y no podemos interaccionar nada con él: la barra del inventario se muestra desierta e inútil. Aquí está su código.

```
Process Item_select_cursor_empty()
```

```

Begin
    graph=217;
    x=10;
    y=10;
    z=-15;
    loop
        frame;
    end
end

```

Si ejecutas en estos momentos el juego, verás que aparece la barra de inventario y podemos apretar ENTER, pero si salimos del inventario no ha pasado nada de nada. Tenemos que hacer que si se selecciona la espada del inventario, al volver al juego nuestro protagonista la empuñe y la pueda mover de un lado a otro, por ejemplo. Manos a la obra.

Vamos a hacer que cuando seleccionemos con la tecla ENTER en el inventario la espada, y salgamos de la pantalla del inventario, dentro del mundo o de las casas el protagonista pueda mover la espada cuando se pulse por ejemplo la tecla “e”. Habrá que modificar dos procesos: “MyCharacter()” y “MyCharacter\_in\_House()”, de esta manera:

```
Process MyCharacter (x,y)
```

```
BEGIN
```

```

...
Loop

```

```

...
if(key(_right)) despega_x=2; direction=2;end
if(key(_left)) despega_x=-2; direction=4;end
if(key(_up)) despega_y=-2; direction=1;end
if(key(_down)) despega_y=2; direction=3;end

```

```

...

```

```

if (sword_selected==true and key(_e) and direction==1 and kill_mode==false)

```

```

        graph=705;
        kill_mode=true;

    repeat
        graph=graph+1;
        frame;
    until (graph==709)

        graph=6;

    else
        kill_mode=false;
    end

    if (sword_selected==true and key(_e) and direction==3 and kill_mode==false)
        graph=700;
        kill_mode=true;

    repeat
        graph=graph+1;
        frame;
    until (graph==704)

        graph=1;

    else
        kill_mode=false;
    end

    frame;
End //Loop
End

```

Para el proceso “MyCharacter\_in\_House()” se deberán hacer exactamente las mismas modificaciones.

No hay mucho que comentar: siempre que se pulse la tecla “e” y se haya escogido la espada del inventario, aparecerá la animación corta de nuestro personaje empuñando la espada, orientado hacia una dirección u otra. Para saber la orientación de la animación (si son los gráficos mirando al norte, al sur, etc) se utiliza la variable “direction”. En este código no hemos escrito las animaciones para las orientaciones este y oeste: tendrías que escribir dos ifs similares a los mostrados, cambiando los códigos de los gráficos adecuados en el FPG.

Finalmente, un detalle que no podemos escapar es resetear el valor de algunas variables para que si el jugador desea volver a empezar el juego habiéndolo empezado ya (es decir, volver al Case 1 desde cualquier otro Case), que el jugador no se encuentre empezando el juego con la espada empuñada, por ejemplo. Así que añade las siguientes líneas al final del Case 1:

```

to_item_select_screen=false;
last_level=0;
sword_in_inventory=true;
kill_mode=false;
sword_selected=false;

```

Un último detalle para que trabajes...Fíjate que si ejecutas el juego y seleccionas la espada del inventario, la podrás usar en el juego, pero si vuelves al inventario, el dibujo de la espada todavía continúa allí como si no la hubiéramos cogido. Una solución sería poner esto

```
If(sword_selected==true)
    Sword_in_inventory=false;
End
```

en el Case 5, justo antes de las señales S\_KILL. No obstante, queda pendiente un hecho:si en este momento nuestro personaje empuña la espada,¿cómo podemos devolver ésta al inventario otra vez? Queda como ejercicio.

Por cierto, la variable “kill\_mode” se usará próximamente, cuando añadamos enemigos.

### **Guardar partidas en el disco duro y cargarlas posteriormente:**

Lo primero que tienes que decidir es cuáles serán las variables cuyos valores necesitas guardar. La idea es que apretando CTRL+S o algo parecido se guarden dichos valores para posteriormente cargarlos apretando CTRL+L o similar y proseguir el juego con los valores almacenados de estas variables. Aquí os pongo la lista de las variables que vamos a guardar:

```
level
MyChar_Position_X
MyChar_Position_Y
MyChar_has_KeyOne
talkboblins
talkwizardlines
asked_bob_for_key
sword_selected
sword_in_inventory
```

Guardaremos estos valores en una tabla. En la lista hay 9 variables, pero a lo mejor querrás salvar más variables más adelante, así que crearemos una tabla llamada “savedata[20]” con 21 elementos, incluyendo el 0.

Así que añade una variable global entera más a la lista, que será dicha tabla:

**Int savedata[20];**

Y ahora tendremos que modificar el Case 1 de la siguiente manera:

*Case 1:*

```
...
Loop
    if(selection==1) level=2; break; end;
    if(selection==2) fade_off();exit("Gracias por jugar");end
```

```

        if(key(_control) and key(_s))
            savedata[0]=level;
            savedata[1]=MyChar_Position_X;
            savedata[2]=MyChar_Position_Y;
            savedata[3]=MyChar_has_KeyOne;
            savedata[4]=talkboblins;
            savedata[5]=talkwizardlines;
            savedata[6]=asked_bob_for_key;
            savedata[7]=sword_selected;
            savedata[8]=sword_in_inventory;

            save("player.dat",savedata);
            end
        frame;
    end

...
End

```

Lo único nuevo aquí es el uso de la función ya conocida *Save*. Este comando sirve para guardar el contenido de una variable (o estructura, o tabla) en un determinado archivo, el cual se puede crear en ese momento, o utilizarse uno ya existente, borrándose lo que contuviera antes –así que ojo-. El primer parámetro de *Save* es precisamente el nombre completo (ruta más nombre) del archivo donde vamos a grabar los datos; si no se pone ruta el archivo se creará (o buscará si ya existe) en la misma carpeta donde esté el DCB. La extensión de este archivo puede ser cualquiera. Y el segundo parámetro es el nombre de aquella variable, estructura o tabla que se quiere guardar.

Para ver si funciona, vamos a hacer que nuestro juego pueda cargar los valores de las variables previamente guardadas. Para ello, tendremos que utilizar la función *Load*. Esta función tiene los mismos dos parámetros: la ruta completa de un archivo y el nombre de una variable/tabla/estructura, pero en este caso lo que hace es “leer” el archivo especificado y rellenar con los datos leídos la variable/tabla/estructura que se le haya dicho. Comprenderás que es importante, que carguemos con *Load* las mismas variables/tablas/estructuras que fueron usadas a la hora de grabar, porque si no la información leída no tendrá ningún sentido. En nuestro juego haremos que se carguen los valores pulsando CTRL+L, así que, en el Case 1, justo después del bloque *if(key(\_control) and key(\_l))/end* –antes del Frame;- añadiremos lo siguiente:

```

if (key(_control) and key(_l))

    load("player.dat", savedata);

    level=savedata[0];
    MyChar_Position_X=savedata[1];
    MyChar_Position_Y=savedata[2];
    MyChar_has_KeyOne=savedata[3];
    talkboblins=savedata[4];
    talkwizardlines=savedata[5];
    asked_bob_for_key=savedata[6];
    sword_selected=savedata[7];
    sword_in_inventory=savedata[8];
    break;
end

```

Fíjate que una vez cargados en memoria, en la tabla “savedata” los valores leídos del archivo

“player.dat”, lo que hay que hacer es asignarle el valor de cada elemento recogido a la variable que le pertoca para que pueda ser usada en el programa con el nuevo valor. Y ya está.

Ahora, lo que hay que hacer es escribir estos dos bloques de ifs seguidos, el *if(key(\_control) and key(\_s))/end* y el *if(key(\_control) and key(\_l))/end* en los lugares oportunos de los Case 2, 3 y 4 (en el Case 5, evidentemente, no hace falta).

Coincidirás conmigo en los sitios idóneos: en el Case 2 hay que escribirlo justo antes del Frame; del Loop; en el Case 3 antes del bloque *if(key(\_esc))/end* que está al final del Loop; y en el Case 4, igualmente, justo antes del *if(key(\_esc))/end* que está al final del Loop.

Testea tu programa. Por ejemplo, consigue la llave del mago y salva pulsando CTRL+S y entonces reinicia el juego. Pulsa CTRL+L y deberías de tener la llave y poder entrar en la segunda casa.

Lo más normal es que quieras hacer en tus juegos del estilo del que estamos haciendo un inventario que pueda incluir más de un sólo ítem. En este caso, la idea general sería la misma que la que hemos comentado en los párrafos anteriores, y sólo tienes que tener en cuenta unos detalles adicionales.

Por ejemplo, para gestionar los diferentes elementos del inventario, lo más práctico es crear un tipo de datos personalizado “ítem”, en el cual básicamente se guardará el tipo de ítem que es, el número de este ítem que hay actualmente en el inventario, etc. Haciéndolo así, el inventario como tal simplemente podría ser una tabla de estos ítems incluida dentro de una estructura que contendría éste y otros datos relevantes posibles relativos a nuestro personaje. De esta manera, podríamos controlar por ejemplo, para ese personaje, si tal elemento o tal otro de esa tabla (es decir, tal o cual ítem) está presente o no, consultando para ello el valor de dicho elemento: si vale 0 no hay ítem, y si vale otra cosa, entonces se podría saber qué ítem es consultando su campo identificador correspondiente. Es decir, algo parecido a esto:

```
type item
  int id=0; // identificador del objeto
  byte numero=0; // numero de unidades que tienes del objeto
end
```

Y...

```
struct player
  ...
  item inventario[X][Y];
end
```

donde X sería el número de columnas que tendrá el inventario e Y el número de filas del inventario

Esto por lo que respecta a la gestión lógica del inventario. Otro tema es cómo se visualizaría ese inventario en la pantalla. ¿Cómo hacer que cada objeto que coja el personaje, aparezca en el inventario cada en distintas coordenadas como si estuviesen ordenados? Siguiendo la filosofía de mostrar un cuadrado por cada ítem (vacío o no), lo que se podría hacer en el caso de tener varios ítems es lo siguiente:

Habría que obtener las coordenadas del primer cuadro del inventario, y a partir de ahí sumar a cada objeto la diferencia de X. Una vez se haya llegado al extremo derecho de la ventana del inventario, se tendría que resetear la posición horizontal como al principio y sumar la diferencia de Y.

Me explico: imagina por ejemplo que el centro del cuadro de la espada es X=50 e Y=150, y que cada cuadro siguiente está 50 píxeles hacia la derecha y los de abajo están 30 píxeles por debajo. Si lo hacemos así, el primer ítem del inventario iría en (50,150), pero el segundo en (100,150), el tercero en (150,150), etc. Para colocar el objeto en el inventario, tendrías que saber cuántos hay (para conocer qué

posicion le toca) y utilizar alguna fórmula parecida a ésta:

```
x=50+(50*idobjeto);  
y=150;
```

donde decimos que la posición horizontal del ítem viene dado por su identificador (1,2,3...), colocándose así de forma ordenada respetando los tamaños y posiciones de los cuadros. Cuando se colocan tantos objetos que el siguiente debe ir debajo, y a hemos dicho que entonces la X tendría que resetearse y aumentar la Y. ¿Pero cómo se sabe cuándo ocurre esto? En las siguiente dos líneas solucionamos este problema y lo tenemos todo hecho de un golpe:

```
x=50+(50*(idobjeto%4));  
y=150+(30*(idobjeto/4));
```

Lo que hacemos aquí es utilizar la operación de módulo y aprovechar que la división de dos enteros devuelve un entero truncado, para dibujar un inventario que conste de cuatro cuadros por fila. Compruébalo “a mano” observando qué valores coge X e Y cuando idobjeto vale 1,2,3,4,5,etc.

### **Añadiendo enemigos, medida de vida y Game Over:**

Éste es el último apartado. Lo que haremos aquí es crear un proceso que muestre cuánta vida tiene el protagonista y que cuando se le acabe la vida haga volver automáticamente al jugador a la pantalla inicial (Case 1).

Primero los gráficos:

-Dibuja una flecha roja apuntando hacia arriba. Se pretende que sea un enemigo; lo puedes hacer mejor más adelante. Dibuja cuatro imágenes más de esta flecha roja, que serán las que se mostrarán cuando el enemigo muera. Guarda el enemigo “sano” con el código 600, y los otros cuatro con los códigos del 601 al 604.

-Dibuja 15 imágenes de 80x30 del medidor de vida. El primero debería representar la vida completa y el último debería de estar totalmente vacío. Puedes usar una barra o algo parecido a corazones, lo que tú quieras. Guárdalos entro del FPG con los códigos 200 al 214 (repito: el 200 sería el gráfico de máxima vida y el 214 el de nada de vida).

Ahora, a programar. Vamos a hacer un enemigo muy simple, el cual se mantiene siempre mirando hacia nuestro protagonista (como el mago y Bob) y además, que caminará detrás del personaje si éste se le acerca demasiado. Añade el siguiente proceso al final del programa.

```
Process Enemy_Pointer(x,y)
```

```
private
```

```
id2; //Esta variable contendrá el código identificador del protagonista  
idangle; //Ésta contendrá el ángulo entre el enemigo y el protagonista  
iddist; //Y ésta lo mismo que la anterior pero para la distancia  
hurt=3; //Vida del enemigo  
counter_kill=16; //Contador que retrasa la pérdida de vida del enemigo  
move_x;  
move_y;
```

```
end
```

*begin*

*z=1;*

*ctype=c\_scroll;//Estos enemigos se verán dentro del mundo verde*

*graph=600;*

*loop*

*id2=get\_id(TYPE MyCharacter);*

*iddist=get\_dist(id2); /\*Esta función , como recordarás, lo que hace es devolver la distancia en píxeles entre el centro del gráfico del proceso identificado por “id2” –según la línea anterior, nuestro protagonista- respecto al centro del gráfico del proceso que realiza la llamada a get\_dist, o sea, “enemy\_pointer”. Resultado: almacena en “iddist” la distancia entre el protagonista y este enemigo.\*/*

*if (iddist=<81)*

*idangle=get\_angle(id2);*

*else*

*angle=0;*

*end*

*angle=idangle-90000; /\*Esta línea es la que hace que el enemigo esté siempre*

*“mirando” al protagonista\*/*

*/\*Recuerda que las funciones get\_distx y get\_disty lo que hacen es devolver, en las mismas unidades que uses para la distancia, el ancho y el alto respectivamente del rectángulo formado por la línea que recorre esa distancia (es decir: la línea sería la diagonal de ese rectángulo y el ancho y alto del rectángulo serían los catetos de cualquiera de los dos triángulos rectángulo que se formarían).*

*Usando estas funciones puedes saber qué cantidades sumar a las coordenadas X e Y de un objeto para desplazarlo en la dirección que desees. Repasa el apartado correspondiente del capítulo donde se explicaron dichas funciones..\*/*

*move\_x=get\_distx(idangle,iddist);*

*move\_y=get\_disty(idangle,iddist);*

*/\*Los cuatro ifs de a continuación simplemente le dicen al enemigo que se mueva hacia nuestro personaje si éste se acerca demasiado al enemigo.\*/*

*if ((move\_x<-20 and move\_x>-100) and (move\_y>-100 and move\_y<100))*

*x=x-1;*

*end*

*if ((move\_x>20 and move\_x<100) and (move\_y>-100 and move\_y<100))*

*x=x+1;*

*end*

*if ((move\_y<-20 and move\_y>-100) and (move\_x>-100 and move\_x<100))*

*y=y-1;*

*end*

*if ((move\_y>20 and move\_y<100) and (move\_x>-100 and move\_x<100))*

*y=y+1;*

*end*

*/\*Las siguientes líneas están para que el protagonista no pueda matar al enemigo tan rápidamente con la espada\*/*

*counter\_kill=counter\_kill-1;*

*if (counter\_kill==0)*

*counter\_kill=16;*

*end*

*/\*Si nuestro protagonista usa la espada, quitará alguna vida al enemigo. Fíjate que esto sólo lo podrá hacer*

```

si "kill_mode" vale true.¿Te acuerdas de cuándo le asignamos ese valor?*/
    if (collision (type MyCharacter) and kill_mode==true and counter_kill==1)
        hurt=hurt-1;
        x=x+get_distx(idangle,-10);
        y=y+get_disty(idangle,-10);
    end

    if (hurt==0); //Si el enemigo muere, aparece la animación pertinente
        repeat
            graph=graph+1;
            frame;
        until (graph>604)
        break;
    end

    frame;
end //Loop
end

```

Cuando hayas añadido este proceso tendrás que hacer otro para mostrar el medidor de vida, allí donde los valores especificados de sus variables locales X e Y digan (yo lo pondré en la parte inferior de la pantalla).Añade el siguiente proceso al final del programa:

```

Process Health_Meter(x,y)
Begin
    loop
        If (health==14) graph=200;end
        If (health==13) graph=201;end
        If (health==12) graph=202;end
        If (health==11) graph=203;end
        If (health==10) graph=204;end
        If (health==9) graph=205;end
        If (health==8) graph=206;end
        If (health==7) graph=207;end
        If (health==6) graph=208;end
        If (health==5) graph=209;end
        If (health==4) graph=210;end
        If (health==3) graph=211;end
        If (health==2) graph=212;end
        If (health==1) graph=213;end
        If (health<=0) graph=214;end
        frame;
    end
end

```

Como puedes comprobar, utilizo una variable llamada "health" que controla este proceso, por lo que habrá que añadirla también a la lista de variables globales enteras del programa.

Ahora nos falta llamar a este proceso en los Case 2,3 y 4.

Así pues, en el Case 2 escribiremos, justo después de iniciar el scroll con *start\_scroll* (dentro del bloque *if(to\_item\_select\_screen==false)/end*):

```
Health_Meter(40,170);
```

En el Case 3 escribiremos la misma línea anterior, justo después de haber creado el proceso "talklines()". Y en el Case 4 la escribiremos justo después de la llamada al proceso "MyCharacter\_in\_House()".

Después de esto, lo que falta es modificar los dos procesos de nuestro protagonista, "MyCharacter()" y "MyCharacter\_in\_House()", para decirle que tendrá que perder vida si choca contra un enemigo.

Justo antes del Frame; del Loop, en ambos procesos, escribe esto:

```
atraso=atraso+1;
if (atraso==50)
    atraso=0;
end
if (collision(type Enemy_Pointer) and atraso==0)
    health=health-1;
end
```

Una nueva variable ha sido usada, "atraso", así que tendrás que declararla en la lista de variables globales enteras. Fíjate que lo que hemos hecho ha sido simplemente hacer que cada vez que ocurra una colisión con un proceso de tipo enemigo, nuestro protagonista disminuya en 1 su energía. ¿Y qué pinta la variable "atraso"? Bueno, es una variable que si te fijas sirve para que cuando un enemigo choque con el protagonista no le quite en un santiamén la energía que tiene, porque normalmente los choques no son instantáneos, sino que el choque se produce durante bastantes frames seguidos, por lo que si en cada frame se le quitara un punto de vida, por cada leve roce con un enemigo nuestro protagonista moriría. El truco que se ha hecho servir es hacer que solamente una vez de cada 50 –cuando "atraso" valga 0, y eso ocurrirá cada 50 frames- se produzca la disminución de energía. Esto querrá decir que en cada choque, la disminución de energía será más pausada y el jugador tendrá más posibilidades de reaccionar y salir del choque sin perder tanta energía.

Además, deberemos añadir una última línea en el Case 1, diciéndole que resetee la variable "health" cuando se reinicie el juego:

```
health=14;
```

Y tendremos que cambiar los Case 2,3 y 4 para volver a la pantalla inicial cuando el personaje muera. Para ello, en los tres Cases, justo después del bloque *if(key(\_i))/end*, escribiremos:

```
If (Health==0)
    write (select_fnt,150,100,4,"Game Over:");
    let_me_alone();
    fade(100,0,0,5);
    level=1;

    frame(5000);

    break;
```

*end*

Aquí hemos utilizado la función *fade*. Recuerda que esta función activa una transición de colores automática, que puede usarse para oscurecer la pantalla, hacer un fundido, o colorear su contenido. Tiene cuatro parámetros: los tres primeros representan un porcentaje de las componentes primarias del color: R, G y B y el cuarto parámetro es la velocidad: que es un valor entero entre 1 y 64. En el caso concreto de nuestro juego, realizamos un fundido a rojo mate de forma lenta. (este efecto sange mostrará visualmente que hemos muerto)

¡Y ya está!¿Ya está? ¡Pero los enemigos no salen por ningún lado!Claro, los tenemos que crear. Modifica el Case 2 –el mapa verde- para que tus enemigos se muestren. Yo he creado tres y los he puesto en el centro del mundo, así que tendrás que ir a buscarlos si te quieres encontrar con ellos. Así pues, justo después de crear nuestro protagonista llamando al proceso “MyCharacter()” –dentro del bloque *if(to\_item\_select\_screen)/end*, crearemos tres enemigos, así:

```
enemy_pointer(1600,1400);  
enemy_pointer(1750,1300);  
enemy_pointer(1200,1750);
```

Ten en cuenta que si lo hacemos así, cada vez que entremos en una casa y volvamos a salir, se volverán a crear estos tres enemigos otra vez –volvemos al Case 2-. Cambia como tu veas si no quieres este comportamiento.

Chequea tu programa ahora. Deberías encontrar algunos enemigos en la pantalla; los podrás matar con tu ataque de espada, pero igualmente, ellos podrán lentamente quitarte la vida, si les tocas.

Ya hemos acabado nuestro pequeño RPG. Si deseas saber más, dirígete al último capítulo de este manual, donde encontrarás más recursos para aplicar en este tipo de juegos, y referencias a códigos fuente de juegos ya terminados para que los puedas estudiar. También puedes consultar las interesantísimas páginas (en inglés) [http://gpwiki.org/index.php/RPG\\_CHAR](http://gpwiki.org/index.php/RPG_CHAR) ó [http://gpwiki.org/index.php/RPG\\_Map](http://gpwiki.org/index.php/RPG_Map)

### **Concepto y utilidad de los “tiles”:**

Seguramente habrás apreciado que en el inicio del juego parece que éste se demora un poco: tarda en salir la pantalla inicial. Esto es por una razón clara: necesita cargar en memoria todo el archivo FPG de nuestro juego, el cual contiene entre otras cosas dos grandes mapas de 3000x3000 (el mapa visible y el de durezas). Evidentemente, el tener grandes mapas ocupa muchos recursos al ordenador, porque éste ha de estar permanentemente controlando que el espacio de memoria reservado a estas grandes imágenes permanezca disponible.

En seguida verás que si tenemos más mapas de estas dimensiones, el tamaño del archivo FPG llegará a ser inabarcable y nuestro juego se resentirá en velocidad y posibles cuelgues por no poder llegar a gestionar de golpe tantos píxeles cargados en memoria.

Podríamos pensar en no cargar todos los gráficos al principio del juego. Es decir, hacer diferentes FPGs y cargarlos según fuera necesario. La idea es buena, pero el problema con los mapas grandes los seguiríamos teniendo, y además, en el momento de cargarlos –en mitad del juego- se notaría más el “parón”.

Una posibles solución podría ser, por ejemplo, reducir el tamaño del mapa de durezas a la mitad ( o a la cuarta parte, etc) en sus dos coordenadas. Se esta manera, tendríamos un mapa de, por ejemplo 1500x1500, con lo que ganaríamos algo. Evidentemente, si hacemos esto, entonces en todas las líneas de

nuestro código donde utilicemos la función `map_get_pixel` para comprobar durezas, las coordenadas pasadas como parámetros deberían de dividirse por dos ( o por cuatro,etc).

La mejor solución para estos problemas son los “tiles” (“baldosa” en inglés). Un escenario “tileado” es un escenario donde se usan muchas piezas pequeñas que encajan entre sí en vez de una única imagen grande. Por ejemplo, en vez de hacer un gráfico enorme de campo verde con dos árboles se prepara un cuadrado verde y un cuadrado con un árbol y con eso se monta la imagen en tiempo de ejecución. De esta forma ahorras mucha memoria, ya que necesitas tener menos tamaño total de gráficos cargados, y haces el juego más ligero en lo que a uso de CPU se refiere, ya que no hay nada fuera de pantalla (se pinta sólo lo que queda dentro de la pantalla). Además, si el decorado a “tilear” es un poco repetitivo, se puede ahorrar mucha memoria –piensa en un juego de plataformas 2D- ya que si te fijas verás que los decorados son una repetición continua de unos pocos tipos diferentes de baldosas: es mucho más económico dibujar 20 baldosas distintas y decirle al programa dónde colocarlas que guardar enormes imágenes de cada fase, que ocuparían mucho. A cambio es un poco más complicado de programar, ya que hay que controlar todas esas pequeñas piezas en vez de una grande, pero eso pasa siempre que se optimiza algo.

Una utilidad evidente para los “tiles” es emplearlos para simular un scroll, por ejemplo, en un RPG. Es decir, se construyen pequeños "cuadrados" que juntos formarán el fondo e se mueven según convenga, simulando el scroll. Así se puede conseguir mas rendimiento que con las funciones de scroll de Fenix, que aunque éstas son mas sencillas de programar.

No obstante, es éste un tema relativamente avanzado (y complejo) que requeriría por sí mismo un capítulo entero exclusivamente dedicado a éste. Es por esto que no profundizaremos más en este tutorial básico, y reservamos la explicación de su uso y programación a un apartado del último capítulo de este manual. Remito al lector allí si desea aprender más sobre “tiles”.

### **Includes:**

Otra cosa que habrás descubierto en este tutorial es que cuando los códigos fuente empiezan a sobrepasar cierta longitud, se hacen mucho menos manejables: se pueden cometer más errores, los cambios son engorrosos, se pierde la referencia de lo que se está haciendo...Que el código sea más o menos corto no depende de nosotros: es el que es. Pero lo que sí depende de nosotros es la claridad y pulcritud con la que lo escribamos.

Existe una manera interesante de estructurar y clasificar el código fuente de un juego cuando éste empieza a tener una longitud considerable. Se trata de dividir el código fuente entre varios ficheros de texto diferentes, de manera que cada uno de estos ficheros contenga código sobre partes diferenciadas del total. Por ejemplo, se podría utilizar un fichero para escribir el código del programa principal, otro para escribir los procesos relacionados con el protagonista, otro para escribir los procesos relacionados con los enemigos,etc.De esta manera, se tienen separados los trozos de código por significado o funcionalidad, haciendo mucho más comodo el trabajo de desarrollo y depuración del código. Este proceso es lo que se llama modularización del código.

Y este sistema tiene otra ventaja: podemos reunir en un archivo una serie de procesos o funciones (y secciones con las variables globales,locales,privadas...que utilicen) que sepamos que vamos a reutilizar en varios proyectos, de tal manera que no tengamos que reescribir código ya escrito anteriormente y sólo tengamos que hacer una referencia a ese archivo que contiene el código que nos interesa. Es evidente, pues, que podríamos crearnos nuestras propios almacenes de código listo para ser utilizados desde los más diversos proyectos. Con el ahorro de tiempo y trabajo que eso supone.

Y ¿cómo hacemos referencia dentro de un código fuente “A” a otro archivo el cual contendrá el código “B” que queremos utilizar? Con el comando *include*.

Este comando inserta el contenido de un fichero de código fuente especificado dentro del actual, allí donde el comando esté situado. Tiene un único parámetro que es la ruta del fichero de código fuente a

incluir. Dicho de otra manera: este comando hace que el compilador lea el contenido textual de otro fichero (cuyo nombre se indica entre comillas justo después de la palabra INCLUDE) ,lo copie y lo pegue literalmente en la posición donde se encuentra el comando, de tal manera que parezca que se haya escrito todo en el mismo archivo.

El fichero incluido puede tener cualquier extensión, pero se recomienda “PRG”.

Este comando puede usarse en cualquier lugar del código.

En realidad, técnicamente INCLUDE no es un comando, es una directiva. Esto a nuestro nivel no nos importa salvo en un par de detalles muy importantes:

- 1º) El parámetro que tiene NO ha de ir entre paréntesis
- 2º) El punto y coma al final (;) es opcional: puede llevarlo o no.

Conociendo la existencia de este comando, ahora podríamos estructurar el código de nuestros juegos más extensos de forma más ordenada y coherente. Incluso podríamos llegar al extremo de hacer que el fichero Prg principal del juego constara simplemente de una serie de INCLUDES con todos los ficheros secundarios, y el proceso principal.

Como ejemplo muy sencillo de utilización de la orden *include*, aquí teneis dos códigos fuentes. El primero será el que se incluirá dentro del segundo. El primero definirá la acción que realiza una función creada por nosotros, y el segundo la utilizará. Con este ejemplo se pretende mostrar que si quisiéramos, podríamos crear otro programa que hiciera uso también de la función definida en el código primero, de tal manera que el código de ésta sólo hubiera que escribirlo una sólo vez y pudiera ser utilizado múltiples veces.

Código primero, guardado en el archivo “hola.prg”:

```
Function mifuncion( int pepe, int pepa)
Begin
    Return pepe*pepa;
End
```

Código segundo, guardado en el archivo “miprograma.prg”:

```
Program codigo_q_usa_funcion_hola;
Include “hola.prg”;
Private
    Int mivar;
end
Begin
    Mivar=mifuncion(3,4);
    Write(0,100,100,4,mivar);
Loop
    Frame;
end
End
```

Cuando queramos ejecutar “miprograma.prg”, no se notará que el cálculo de la multiplicación lo realiza otro fichero aparte. Evidentemente, el fichero “hola.prg” no podrá ser ejecutado sólo por sí mismo.

Otro ejemplo donde se demuestra que INCLUDE se puede escribir en cualquier sitio del programa y leerse varias veces si es necesario, puede ser el siguiente:

Código primero, guardado en el archivo “hola.prg”:

```
if(i<5)
    c = i+b*b;
elseif(i<7)
    c = i+b*a;
else
    c = i+a*a;
end
```

Código segundo, guardado en el archivo “miprograma.prg”:

```
Program codigo_q_usa_funcion_hola;
Private
/*Las variables a,b y c se hacen servir en "hola.prg". También se podrían haber declarado en una sección
Private dentro de éste.*/
    Int i,a=2,b=3,c;
end
Begin
    for(i=0;i<10;i++)
        include "hola.prg"
        write(0,100,10+10*i,4,c);
    end
Loop
    Frame;
end
End
```

Quiero recalcar un detalle. No es lo mismo incluir (con *include*) que importar (con *import*). Con *include* insertamos código fuente dentro de otro código fuente: copiamos texto dentro de otro texto. En cambio, con *import* (ya veremos esta orden cuando hablemos de las DLL) lo que hacemos es incrustar código binario ya precompilado –la DLL- dentro del código binario generado por nuestro juego, que no es lo mismo.

Y una cosa importante: los archivos a incluir no deben comenzar nunca por la línea *program...*: esta línea no ha de existir: este archivo simplemente constará de los cuerpos de los diferentes procesos o funciones. Es lógico: sólo puede haber una línea *program* por cada programa. Además, si pretendemos que un archivo incluido pueda ser usado por muchos PRG diferentes, no tiene sentido hacer que un aquél pertenezca a un único program: por eso no se escribe ninguno.

Con esto que hemos comentado, no sería mala idea coger ahora el código de nuestro RPG –que sobrepasa las 1100 líneas- e irlo diseccionando en diferentes INC según el significado o ámbito de los diferentes procesos, para hacer más cómoda y rápida la lectura de su código, facilitando así la búsqueda de errores y su mantenimiento.

## CAPÍTULO 11: TUTORIAL PARA UN JUEGO DE PLATAFORMAS

(extraído de <http://www.flamingbird.com>)

Un juego de plataforma típico es un juego 2D donde el jugador maneja un personaje que debe ir avanzando por la pantalla horizontalmente (normalmente de izquierda a derecha) a través de un escenario, el cual se moverá en scroll horizontal de forma conveniente, y donde aparecen diferentes obstáculos que el protagonista ha de sortear, saltándolos por ejemplo, si no quiere perder vida o puntos. También se pueden encontrar premios por el camino, y en general está compuesto por varios niveles, a los que se llega después de haber completado el anterior. Esto juegos se llaman así porque los escenarios suelen constar de las llamadas plataformas, que son superficies elevadas respecto el suelo por donde el protagonista puede caminar, previo salto, para sortear un enemigo o conseguir un premio. La gracia del juego es ir cambiando de plataformas según aparezcan en el escenario y según la necesidad. Un ejemplo podría ser el Mario Bros.

En este tutorial haremos un juego de plataformas de un solo nivel, pero es fácilmente ampliable. Primero de todo dibuja el fondo scrolleable (que incluya el dibujo de las plataformas) sobre el cual nuestro personaje caminará. El fondo puede ser como tú quieras: con nubes, montañas, edificios, etc, aunque ten en cuenta que como en nuestro caso haremos el juego con una resolución de 640x480, la altura de la imagen del escenario será 480, ya que no se va a mover verticalmente; en cambio, la anchura de la imagen conviene que sea más o menos elevada (bastante mayor que 640 píxeles) para dar la sensación de que el escenario es largo y diferente. Guárdalo en un archivo FPG con el código 20.

En este tutorial no lo haremos, pero en vez de utilizar un único gráfico de scroll que contiene el dibujo de fondo y el dibujo de las plataformas, podrías utilizar dos gráficos de scrolls diferentes: uno sería el gráfico de segundo plano del scroll (el 4º parámetro de *start\_scroll()*) que representaría un paisaje más o menos bonito, y el otro sería el gráfico del primer plano del scroll (el 3º parámetro de *start\_scroll()*), el cual contendría únicamente el dibujo de las plataformas, y el resto sería una gran zona transparente.

Si lo hacemos de esta manera, podemos tener más libertad: podríamos cambiar el paisaje sin cambiar las plataformas y viceversa, o (con el uso adecuado del último parámetro de *start\_scroll()*) podríamos dejar el paisaje inmóvil mientras se estuviera haciendo el scroll en los dibujos de las plataformas -creando un efecto interesante-, etc.

Dibuja también el mapa de durezas correspondiente al escenario. Para ello, crea una imagen de igual tamaño que el escenario (para no hacerla de nuevo, puedes copiarla con otro nombre), y pinta de un color concreto (nosotros usaremos el rojo) las partes cuyas coordenadas coincidan con las zonas del escenario donde el personaje podrá caminar; procura dibujar plataformas y precipicios. Después pinta de otro color (negro, por ejemplo) todo el resto de la imagen entera. Guárdalo en el FPG con el código 27.

Y dibuja también nuestro personaje, de perfil, mirando hacia la derecha, de unos 70x40 píxeles. Guárdalo en el FPG con el código 001. Además, tal como hicimos con el tutorial del RPG, dibujaremos varias imágenes extra del personaje ligeramente diferentes para simular el efecto de caminar. En concreto, crearemos dos imágenes más, con los códigos 002 y 003.

El código del programa principal podría empezar siendo algo parecido a esto:

```
program plataforma;  
global  
    int idescenario;  
end  
private  
end  
begin  
    set_mode(640,480,16);
```

```

set_fps(60,1);
set_title("Mi juego de plataformas");
idescenario=load_fpg("plat.fpg");
start_scroll(0,0,20,0,0,1);
loop
    frame;
end
end

```

Vamos ahora a crear nuestro personaje, en el extremo izquierdo de la pantalla, encima del suelo. Para ello, después de la línea del *start\_scroll* escribimos

```
Player();
```

Y crearemos el proceso “player()”, el cual nos mostrará el personaje, de momento sin animación. Aprovechando, también haremos que éste tenga la posibilidad de saltar.

```
Process player ()
```

```
BEGIN
```

```

ctype=c_scroll;
scroll.camera=id; /*Esto ya sabes que bloquea la cámara para que siga a este proceso por
pantalla. Es todo lo que hay que hacer para realizar el scrolling automáticamente*/

```

```
x=600;
```

```
y=330; /*Cambia los valores de las coord iniciales según tus necesidades*/
```

```
Graph=1;
```

```
LOOP
```

```
    If (key (_right))
```

```
        x=x+5;
```

```
        flags=0;
```

```
    End
```

```
    If (key (_left))
```

```
        x=x-5;
```

```
        flags=1;
```

```
    End
```

```
    FRAME;
```

```
End
```

```
END
```

Ahora vamos a añadir al proceso “player()” una sección de declaraciones de variables privadas (enteras). En concreto, inicializaremos las siguientes variables así:

```
Private
```

```
    Int xx=6;
```

```
    Int yy=12;
```

```
    Int iy=1;
```

```
end
```

Estas variables serán usadas para hacer que nuestro personaje salte. Añadiremos ahora pues la tecla de salto y la capacidad de saltar hacia la derecha. Justo antes de la línea del *Frame*; del proceso “Player()” escribimos:

```
If (key(_s) AND FLAGS==0)
```

```
    REPEAT
```

```
        y=y-(yy-iy);
```

```
        x=x+xx;
```

```
        FRAME;
```

```
        iy=iy+1;
```

```
    UNTIL (map_get_pixel(0,27,x,y)==rgb(255,0,0))
```

```
    iy=1;
```

```
End
```

Esto hace que el jugador salte hacia la derecha de forma realista cuando se pulse la tecla “s”. Fíjate que el truco del realismo del salto está en que al principio la coordenada Y del proceso decrece muy rápido –y la X también-, pero poco a poco la coordenada Y va decreciendo a cada frame menos y menos hasta llegar un punto donde ya no decrece y comienza a crecer poco a poco a más velocidad: para verlo mejor observa y estudia qué valores tiene Y en cada iteración del repeat/until en función de lo que vale la variable YY –que no cambia- y lo que vale IY –que aumenta a cada iteración). El personaje continuará cayendo hasta que alcance una zona (en el mapa de durezas) que tenga el color rojo puro. Se supone que el suelo en el mapa de durezas lo habrás pintado de rojo, tal como hemos dicho ¿no?, aunque en el juego se vea un precioso prado verde...

Para hacer que nuestro personaje salte hacia la izquierda, tendremos que añadir otro if antes de la línea del frame; del proceso “player()”:

```
if(key(_s) and FLAGS==1)
```

```
    REPEAT
```

```
        y=y-(yy-iy);
```

```
        x=x-xx;
```

```
        FRAME;
```

```
        Iy=iy+1;
```

```
    UNTIL (map_get_pixel(0,27,x,y)==rgb(255,0,0))
```

```
    iy=1;
```

```
end
```

La única cosa que ha cambiado es que estamos restando valores a X en vez de sumarlos. Y ya está.

Incluso podrías hacer que el jugador tuviese la posibilidad de pulsar otra tecla con un salto megaultra que pudiera llegar más alto, o más lejos. Simplemente habría que cambiar los valores de las variables “yy”, “xx” e “iy”. Si has dibujado plataformas flotantes, asegúrate que nuestro personaje puede alcanzarlas con la potencia de salto que le hemos asignado.

Ya es hora de incorporar la animación del caminar. Utilizaremos el mismo método que usamos por ejemplo en el tutorial del matamarcianos para generar las explosiones (puede haber otros, pero creo que este es el más fácil). Añade a la sección de declaraciones privadas del proceso “player()” la variable entera “cont” iniciándola a 0, y cambia los bloques *if(\_right)/end* y *if(\_left)/end* del proceso “player()” por estos otros:

```
if(key(_right))
```

```
    x=x+5;
```

```
    flags=0;
```

```
    for(cont=3;cont>=1;cont=cont-1)
```

```
        graph=cont;
```

```

        frame;
    end
end
Y
if(key(_left))
    x=x-5;
    flags=1;
    for(cont=3;cont>=1;cont=cont-1)
        graph=cont;
        frame;
    end
end
end

```

Quizás te haya sorprendido que el FOR es decreciente. Esto lo he hecho así para que la última imagen que se vea al salir de la última iteración (y por tanto, la imagen que volverá a ser la de nuestro personaje quieto) sea siempre la misma, la número 1.

Un detalle importante es poner la orden FRAME dentro del for, porque si no el bucle se realizaría entero pero no se vería nada porque sólo se mostraría en pantalla el resultado final (la imagen 1) en el FRAME del final del proceso. Si queremos “hacer un seguimiento” iteración a iteración de los cambios en la variable GRAPH, tenemos que forzar a que el fotograma se lance contra la pantalla a cada momento.

Ahora lo que haremos será añadir un precipicio por donde el jugador podrá caer si no va con cuidado. En este caso haremos el precipicio directamente en el suelo. El funcionamiento para precipicios situados en plataformas elevadas es muy parecido: sólo hay que cambiar la condición que define cuándo nuestro personaje para de caer: si desaparece por debajo de la ventana de juego (el supuesto que haremos) o si cae en una plataforma inferior (supuesto que no hemos desarrollado pero fácil de ver: simplemente hay que utilizar la función *map\_get\_pixel* para ver si en la caída el personaje choca con una plataforma dada).

Para hacer que el personaje caiga si camina por el suelo, tendremos que eliminar alguna porción del suelo de color rojo del mapa de durezas y pintarlo como el resto, negro.

Una vez hecho esto, verás que nuestro personaje, si salta sobre el precipicio, cae infinitamente y nos quedamos con el bonito escenario en pantalla sin poder hacer nada de nada. Lo que nosotros queremos es que una vez que caiga, nuestro personaje muera. Para eso, tendremos que añadir el código necesario en el proceso principal, porque aunque podríamos decir a un proceso que se matara a sí mismo (con la orden *signal(id,s\_kill);*), es mejor hacerlo desde el proceso principal para tener este asunto centralizado y así evitar posibles errores posteriores derivados de despistes (querer utilizar un proceso cuando se ha suicidado antes, y cosas así). Por tanto,añadiremos la siguiente porción de código dentro del bloque Loop/End del proceso principal, justo antes del Frame;

```

if(son.y>=500) /*La máxima coordenada Y es 480, según nuestra resolución. Añadimos unos píxeles más
para que el personaje logre desaparecer de la pantalla por completo. Sería buena idea hacer una
constante que fuera la resolución vertical y así poner en este if una condición dependiente de esa
constante: sería mucho más fácil cambiar los valores si hubiera que hacerlo alguna vez.*/
    signal(son,s_kill);
    player();
end

```

Fíjate que lo que hacemos es, en cada iteración del programa principal, comprobar si el proceso hijo (“player()” es el único hijo que hay del programa principal y por eso no hay confusión: si

hubiera más hijos tendría que usar el identificador generado en su llamada) ha superado el valor 500 de su variable local Y. Si es el caso, se le lanza una señal de matarlo, y seguidamente, se le vuelve a crear, volviéndose a ejecutar el código del proceso “player()”, el cual sitúa el jugador otra vez en las coordenadas X=600 e Y=330 para volver a empezar. Tal como está el juego ahora, haber escrito *signal(son,s\_kill)* o *let\_me\_alone()* sería equivalente, porque sólo hay un proceso más aparte del principal.

Supongo que te habrás dado cuenta de un gran fallo: nuestro personaje sólo cae en el precipicio si viene de un salto, pero si va caminando, ¡pasa por encima de él como si nada! Esto hay que solucionarlo.

Vuelve al mapa de durezas. Ahora vas a repintar el hueco del precipicio (que había pasado de ser de color rojo a negro, como todo el resto del mapa) de otro color, el azul puro, por ejemplo. Y en el bucle Loop/End del proceso “player()”, justo antes del Frame; añadimos lo siguiente:

```
if(map_get_pixel(0,27,x,y)==rgb(0,0,255))
```

```
    REPEAT
```

```
        y=y+5;
```

```
        frame;
```

```
    UNTIL (y>=500)
```

```
End
```

Lo que hace esto es, si el personaje toca la zona azul del mapa de durezas, le aumentará el valor de su variable Y para hacerlo caer hasta que desaparezca por debajo de la ventana. Y ya que cuando ocurre eso, ya has añadido la orden de enviar la señal de matarle, ya lo tenemos todo. Fíjate que hemos puesto la orden Frame; también dentro de este If: esto es para que se visualice la caída lentamente; si no se hubiera puesto, el Repeat/Until se haría de golpe y veríamos nuestro personaje de repente ya abajo, efecto que no queda nada bien.

Si has hecho unas cuantas plataformas y has probado de hacer que nuestro personaje salte y llegue a alguna de ellas, verás que una vez haya salido de la plataforma por un extremo, el personaje continuará caminando como si la plataforma no hubiera acabado, ¡y no caerá! Bueno. El truco está en poner, en el mapa de durezas, en cada extremo de la plataforma un pequeño cuadrado de otro color, por ejemplo el verde puro, de manera que quede un rectángulo rojo acabado por ambos extremos por cuadrados verdes –como una varita mágica multicolor-. Y escribir lo siguiente en el bucle Loop/End del proceso “player()”, justo antes del Frame;

```
if(map_get_pixel(0,27,x,y)==rgb(0,255,0))
```

```
    REPEAT
```

```
        y=y+5;
```

```
        frame;
```

```
    UNTIL (map_get_pixel(0,27,x,y)==rgb(255,0,0))
```

```
End
```

La idea es repetir el truco que hemos hecho antes para que nuestro personaje cayera por el precipicio del suelo, pero ahora, la condición de final de la caída no es que salga por debajo de la pantalla sino que su nuestro personaje se tope con una zona roja en el mapa de durezas, que puede ser el suelo u otra plataforma inferior. En ese momento, la caída finalizará.

Todavía falta un detalle más, y es ver qué es lo que tenemos que hacer para que cuando nuestro personaje está caminando por una plataforma inferior y salta, si choca de cabeza con la parte inferior de una plataforma más elevada rebote y caiga otra vez para abajo. Tal como tenemos ahora el juego, cuando choca contra una plataforma en mitad del salto, se mantiene en esa plataforma, y eso no debe ocurrir. ¿Qué tendríamos que hacer? Pues lo mismo de siempre: crear una zona de un color determinado en el mapa de

durezas que abarcara toda la zona inferior de las plataformas flotantes, y utilizar *map\_get\_pixel* para detectar si la parte superior del gráfico del personaje -es decir, un punto dado por la diferencia entre su coordenada Y actual y la mitad de la altura total del gráfico del personaje- colisiona con ésta. Si lo hace, el personaje debería de caer igual que lo hace cuando se lanza por un extremo de cualquier plataforma o por el precipicio. Se deja como ejercicio, aunque si no se te ocurre, échale un vistazo al código del final.

Lo que queda por añadir son los posibles premios y/o enemigos que puedan surgir a lo largo del camino de nuestro protagonista. Asociado a ellos también deberíamos contabilizar por ejemplo los puntos que pueda llegar a ganar nuestro protagonista si logra alcanzar los premios, o la energía que pueda llegar a perder si no sortea bien los obstáculos. Se podría poner un límite de puntos para alcanzar el fin del nivel, o por el contrario, se podría obligar a un Game Over si nuestro protagonista se queda sin energía. Todos estos detalles se parecen bastante a los que ya vimos con el tutorial del matamarcianos y, si te acuerdas de aquel tutorial, no te serán demasiado difíciles de programar.

La gran diferencia respecto el tutorial del matamarcianos vendría de que los premios y los enemigos no deben aparecer en pantalla tan aleatoriamente como en dicho tutorial. Ambos tendrán que estar siempre sobre alguna plataforma: al igual que nuestro protagonista, no pueden moverse por el aire, (al menos a priori). Lo único realmente aleatorio en los premios y enemigos es el intervalo de sus apariciones, ya que sus posiciones, repito, siempre estarán ligadas a unas coordenadas X e Y concretas, las que corresponden a la anchura y altura de las plataformas existentes en ese momento en pantalla (sus posiciones dentro de los límites de éstas sí que podrán ser aleatorias, pero sólo dentro de dichos límites). Los enemigos, a diferencia de los premios, se podrán mover -de forma aleatoria, o acercándose en dirección al personaje principal, o como queramos-, pero siempre respetando también los límites que imponen a sus coordenadas X e Y las plataformas existentes en el juego. No es excesivamente complejo llevar a cabo en el código fuente todo este conjunto de consideraciones, aunque tampoco es trivial. Se puede llegar de múltiples formas, y todas serán buenas si funcionan.

Si no te ha quedado clara la explicación anterior, no te preocupes: puedes ver una implementación de esta idea en el código del final del cuadro siguiente.

A continuación muestro el código comentado completo del juego de plataformas que hemos ido haciendo hasta ahora, incluyendo apariciones de premios y enemigos, y recuento de puntos y game over.

Para incluir los gráficos de los premios en el FPG hemos hecho servir los códigos 009 y 010 -hay dos gráficos de premios posibles, pues-. Además, cuando el personaje choque contra un premio, éste, después de darle puntos, desaparecerá, no sin antes cambiar su gráfico por el de un bonito gráfico de colores vivos, identificado con el código 011 en el FPG.

Para incluir los gráficos de los enemigos en el FPG hemos hecho servir los códigos 006, 007 y 008 -hay tres gráficos de enemigos posibles, pues-. Además, cuando el personaje choque contra un enemigo, éste, después de quitarle vida, desaparecerá, no sin antes cambiar su gráfico por el de una horrible explosión formada por gráficos de identificadores 012, 013, 014, 015 y 016.

También se ha añadido por pura estética una pequeña introducción, y necesitarás algún que otro archivo de sonido y de fuentes FNT. Si no los tienes a mano, comenta las líneas que hacen uso de sonido y asigna el valor de 0 a las variables "idfuentes" y "idfuentes2".

```
program plat;  
global  
    int idescenario;  
    int idplayer;  
    int idfuente;  
    int idfuente2;
```

```

    int idbandaso;
    int idding;
    int idcrash;
    int idtrumpet;
    int puntos;
    int vida;
end
begin
    set_title("Megaplataforma yeahh!!");
    set_mode(640,480,16);
    set_fps(60,1);
    //Sólo se ve la primera fuente que se carga. Las demás no (?)
    idfuente2=load_fnt("fuentes/fuente2.fnt");
    idfuente=load_fnt("fuentes/fuente.fnt");
    idescenario=load_fpg("plat.fpg");
    idbandaso=load_song("sonidos/mkend.mid");
    idding=load_wav("sonidos/ding.wav");
    idcrash=load_wav("sonidos/crash.wav");
    idtrumpet=load_wav("sonidos/trumpet1.wav");

    intro();
end

process intro()
begin
    x=320;
    y=240;
    file=idescenario;
    graph=19;
    play_wav(idtrumpet,0);
    from z=0 to 49; frame;end
    fade(0,0,0,2);
    while(fading) frame;end
    fade_on();
    juego();
end

process juego()
begin
    write(idfuente,500,30,4,"Puntos:");
    write(idfuente,500,80,4,"Vida:");
    start_scroll(0,idescenario,1,0,0,1);
    play_song(idbandaso,-1);
    idplayer=player();
    loop
        /*En cada frame tengo un 10% de probabilidad de crear un malo y un 30% de crear un premio. Ambos tendrán una posición X e Y aleatoria que puede ser cualquiera de dentro del scroll, un gráfico aleatorio entre unos cuantos disponibles y un tamaño aleatorio también -entre la mitad y el doble que el original-. En párrafos anteriores en este capítulo se ha comentado que los malos y los premios sólo pueden aparecer (a priori) sobre las plataformas existentes en el juego, y aquí los estamos creando en cualquier parte. Bueno, tal como veremos luego cuando veamos el código de los procesos "malo" y "premio", es una vez creado el malo o premio, cuando se comprueba si su X e Y corresponden a las coordenadas válidas. Si es así, se continúa ejecutando la instancia del malo o premio correspondiente; si no, se mata la instancia.*/

```

```

        if(rand(1,100)<10)malo(rand(0,2000),rand(0,480),rand(6,8),rand(50,150));end
        if(rand(1,100)<30)premio(rand(0,2000),rand(0,480),rand(9,10),rand(50,150));end

        /*If necesario para volver a comenzar otra vez desde el principio cuando el personaje haya
        desaparecido por debajo de la pantalla al caer por un precipicio*/
        if(idplayer.y>=500)
            //signal(idplayer,s_kill);
            let_me_alone();
            idplayer=player();
        end
    end
end

process player()
private
    int xx=6;
    int xxx=12;
    int yy=20;
    int yyy=10;
    int iy=1;
    int cont=0;
    int obstacle=0;
    int avanzadillader;
    int avanzadillaizq;
    int idpuntos;
    int idvida;
end
begin
    ctype=c_scroll;
    scroll[0].camera=id;
    x=600;
    y=280;
    graph=3;
    puntos=0;
    vida=10;
    loop
        idpuntos=write_var(idfuente,600,30,4,puntos);
        idvida=write_var(idfuente,600,80,4,vida);
        /*En cada fotograma actualizo los valores de estas dos variables, que servirán para hacer
        un map_get_pixel de 20 píxeles más allá de donde está el personaje (avanzadillader si el personaje va
        hacia la derecha y viceversa) para comprobar que se choca contra un obstaculo y por tanto se ha de
        parar. Para desatascarse, al cambiar el flag la avanzadilla cambia y por tanto, en vez de comprobar el
        pixel de 20 más allá, se comprueba el pixel de 20 más acá y el personaje se podrá desatascar*/
        avanzadillader=x+20;
        avanzadillaizq=x-20;
        /*Separo este if del siguiente porque lo que me interesa es que cuando choque contra algo, y
        por tanto, obstacle será 1, el flags se cambie igualmente. Y esto es importante para los elseifs que vienen a
        continuación, donde dependiendo del valor de flags, se utilizará avanzadillader o avanzadillaizq para
        poder desatascar el personaje*/
        if(key(_right))
            flags=0;
        end
        /*Si se mueve hacia la derecha y no hay obstáculo, avanza, con la animación

```

```
correspondiente*/
```

```
    if(key(_right) and obstacle==0)
        x=x+10;
        if(x>=2000) x=x-2000;end
        for(cont=5;cont>2;cont=cont-1)
            graph=cont;
            frame;
        end
    end
    /*Lo mismo que antes, para la izquierda*/
    if(key(_left))
        flags=1;
    end
    /*Lo mismo que antes, para la izquierda*/
    if(key(_left) and obstacle==0)
        x=x-10;
        if(x<=0)x=x+2000;end
        for(cont=5;cont>2;cont=cont-1)
            graph=cont;
            frame;
        end
    end
end
```

*/\*Bloque que comprueba si la parte superior de la cabeza del personaje (y-40), y por tanto, el personaje en sí, choca contra una plataforma que es más baja que él. Si choca, se pone la variable obstacle a 1, y por tanto impedirá en los ifs anteriores que el personaje se pueda mover. Notar el uso de las avanzadillas en vez de la x, para poder desatascar el personaje, y también notar el uso de elseifs excluyentes entre sí, por que si hacen con ifs/else se pueden solapar unos con otros!!\*/*

```
if(flags==0 and map_get_pixel(idescenario,2,avanzadillader,y-40)==rgb(255,0,0))
    obstacle=1;
elseif (flags==0 and map_get_pixel(idescenario,2,avanzadillader,y-40)!=rgb(255,0,0))
    obstacle=0;
elseif(flags==1 and map_get_pixel(idescenario,2,avanzadillaizq,y-40)==rgb(255,0,0))
    obstacle=1;
elseif (flags==1 and map_get_pixel(idescenario,2,avanzadillaizq,y-40)!=rgb(255,0,0))
    obstacle=0;
end
```

```
/*Salto hacia la derecha*/
```

```
if(key(_s) and flags==0 and obstacle==0)
    repeat
        /*El movimiento del salto propiamente dicho*/
        y=y-(yy-iy);
        x=x+xx;
        /*Para no salirse del mapa de durezas*/
        if(x>=2000) x=x-2000;end
```

*/\*Cuando la cabeza del personaje rebote con la base de una plataforma, éste ha de caer. El algoritmo es el mismo que el de la caída libre desde una plataforma superior a otra inferior (ver más abajo)\*/*

```
        if(map_get_pixel(idescenario,2,x,y-40)==rgb(255,255,0))
            while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
                y=y+5;
                frame;
```

```

end
break;

end
frame;
//Para que se vaya desacelerando el salto y el personaje vuelva a bajar
iy=iy+1;
/*Falta hacer que mientras esté saltando y choque contra una plataforma, se detenga el salto!!!*/
until(map_get_pixel(idescenario,2,x,y+45)==rgb(255,0,0))
/*or map_get_pixel(idescenario,2,avanzadillader,y)==rgb(255,0,0)*/
iy=1;
end

/*Salto hacia la izquierda. Igual que el de la derecha excepto en que la
coordenada x disminuye en vez de aumentar.Por lo demás, todo igual.*/
if(key(_s) and flags==1 and obstacle==0)
repeat
y=y-(yy-iy);
x=x-xx;
if(x<=0)x=x+2000;end
if(map_get_pixel(idescenario,2,x,y-
40)==rgb(255,255,0))
while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
y=y+5;
frame;
end
break;
end
frame;
iy=iy+1;
until(map_get_pixel(idescenario,2,x,y+45)==rgb(255,0,0))
iy=1;
end

/*Salto hacia la derecha*/
if(key(_d) and flags==0 and obstacle==0)
repeat
/*El movimiento del salto propiamente dicho*/
y=y-(yyy-iy);
x=x+xxx;
/*Para no salirse del mapa de durezas*/
if(x>=2000) x=x-2000;end
/*Cuando la cabeza del personaje rebote con la base de una plataforma, éste ha de caer. El
algoritmo es el mismo que el de la caída libre desde una plataforma superior a otra inferior (ver más
abajo)*/
if(map_get_pixel(idescenario,2,x,y-
40)==rgb(255,255,0))
while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
y=y+5;
frame;
end
break;
end
frame;
//Para que se vaya desacelerando el salto y el personaje vuelva a bajar

```

```

                                iy=iy+1;
/*Falta hacer que mientras esté saltando y choque contra una plataforma, se detenga el salto!!!*/
                                until(map_get_pixel(idescenario,2,x,y+45)==rgb(255,0,0))
/*or map_get_pixel(idescenario,2,avanzadillader,y)==rgb(255,0,0)*/
                                iy=1;
                                end

                                /*Salto hacia la izquierda. Igual que el de la derecha excepto en que la
coordenada x disminuye en vez de aumentar.Por lo demás, todo igual.*/
                                if(key(_d) and flags==1 and obstacle==0)
                                        repeat
                                                y=y-(yyy-iy);
                                                x=x-xxx;
                                                if(x<=0)x=x+2000;end
                                                if(map_get_pixel(idescenario,2,x,y-
40)==rgb(255,255,0))
                                                        while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
                                                                y=y+5;
                                                                frame;
                                                                end
                                                                break;
                                                                end
                                                                frame;
                                                                iy=iy+1;
                                until(map_get_pixel(idescenario,2,x,y+45)==rgb(255,0,0))
                                iy=1;
                                end

/*Cuando se encuentra un precipicio, el personaje cae y desaparece -y muere-*/
                                if(map_get_pixel(idescenario,2,x,y+45)==rgb(0,0,255))
                                        while(y<500)
                                                y=y+5;
                                                frame;
                                        end
                                end

/*Cuando el personaje se encuentra en una plataforma y ésta se acaba, ha de
caer hasta la plataforma de debajo*/
                                if(map_get_pixel(idescenario,2,x,y+45)==rgb(0,255,0))
                                        while(map_get_pixel(idescenario,2,x,y+45)!=rgb(255,0,0))
                                                y=y+5;
                                                frame;
                                        end
                                end

                                if(puntos==30)
                                        write(idfuente2,320,280,4,"¡MUY BIEN!");
                                        write(idfuente2,320,320,4,"Pulse ESC para salir");
                                        stop_scroll(0);
                                        put_screen(idescenario,17);
                                        while (not key(_esc))
                                                frame;
                                        end
                                exit();

```

```

        end

        if(vida==0)
            write(idfuente,320,240,4,"GAME OVER!");
            write(idfuente,320,280,4,"Pulse ESC para salir");
            stop_scroll(0);
            put_screen(idescenario,18);
            while (not key(_esc))
                frame;
            end
            exit();
        end

        frame;
        delete_text(idpuntos);
        delete_text(idvida);
    end
end

process malo(x,y,graph,size)
private
    int cont;
    int angleplay;
    int distplay;
    int move_x;
    int yasalio=0;
end
begin
cstype=c_scroll;
repeat
    /*Cada paréntesis que hay entre los "or" indica la posición del suelo superior de cada una de las
diferentes plataformas pintadas en nuestro fondo de scroll. Se comprueba que la posición del malo recién
creado (tanto horizontal como vertical) esté en alguna zona del scroll correspondiente al ras de suelo de
alguna de las plataformas existentes. Esto se hace mirando los valores de los extremos horizontales
izquierdo y derecho y una franja estrecha alrededor del extremo vertical superior de las plataformas.
Esta línea de código la tendrás que cambiar para que se adecúe al dibujo que hayas hecho tu del mapa de
plataformas. No se ha contemplado la posibilidad de que puedan existir enemigos fuera de estas zonas
(volando en el cielo, por ejemplo).*/
    if((x>0 and x<530 and y>410 and y<430)or(x>125 and x<375 and y>165 and y<185) or
(x>465 and x<715 and y>290 and y<310) or (x>1280 and x<1500 and y>290 and y<310) or (x>1470
and x<1710 and y>210 and y<230) or (x>1740 and x<1970 and y>390 and y<410) or (x>625 and
x<1315 and y>415 and y<435))
        /*Las siguientes líneas sirven para detectar la presencia cercana del personaje, en cuyo
caso los enemigos irían tras él. Son exactamente las mismas líneas utilizadas en el tutorial del RPG, en un
capítulo anterior. Lo que faltaría por hacer es que los enemigos puedan moverse entre plataformas,por
ejemplo.*/
        angleplay=get_angle(idplayer);
        distplay=get_dist(idplayer);
        move_x=get_distx(angleplay,distplay);
        if(move_x<-20 and move_x>-100) x=x-1;yasalio=1; end
        if(move_x>20 and move_x<100) x=x+1;yasalio=1;end
        frame;
    else

```

```

        if(yasalio==0)signal(id,s_kill);
        /*Importante este else, porque si se da el caso de que ya ha salido y se ha
llegado al extremo de una plataforma (es decir,cuando se haga este else, porque es cuando yasalio=1 y
cuando la x y la y son diferentes de las del primer if, que son los que indican las posiciones de las
plataformas), el programa se quedaría colgado al no haber un frame;*/
        else frame; end
    end
until(collision (idplayer))
/*Con sólo un choque contra al jugador, el malo muere, pero decrece en una unidad la vida del personaje,
indistintamente del tipo de gráfico de malo que sea (una aspecto para mejorar...)*
play_wav(idcrash,0);
vida=vida-1;
cont=12;
for(cont=12;cont<=16;cont=cont+1)
    graph=cont;
    frame;
end
end

process premio(x,y,graph,size)
private
    int cont;
end
begin
ctype=c_scroll;
repeat
    /*La condición de este if es idéntica a la de los malos, ya que sólo podrán aparecer sobre las plataformas
existentes, al igual que ellos*/
    if((x>0 and x<530 and y>410 and y<430)or(x>125 and x<375 and y>165 and y<185) or
(x>465 and x<715 and y>290 and y<310) or (x>1280 and x<1500 and y>290 and y<310) or (x>1470
and x<1710 and y>210 and y<230) or (x>1740 and x<1970 and y>390 and y<410) or (x>625 and
x<1315 and y>415 and y<435))
        frame;
    else
        signal(id,s_kill);
    end
until(collision (idplayer))
play_wav(idding,0);
/*Con sólo un choque contra al jugador, el premio desaparece, pero aumenta en una unidad los puntos del
personaje, indistintamente del tipo de gráfico de premio que sea (una aspecto para mejorar...).*/
puntos=puntos+1;
graph=11;
for(cont=1;cont<=50;cont=cont+1)
    y=y-10;
    frame;
end
end

```

Ya tenemos acabado nuestro juego de plataformas básico, pero es posible que quieras mejorarlo. Una de las posibles mejoras podría ser por ejemplo el introducir más niveles, los cuales incorporarían otras plataformas diferentes. A lo mejor te puede apetecer crear plataformas de forma irregular, con pequeñas cuestas o bajadas. Si es éste el caso, el de incorporar pendientes a nuestro juego, tendrás que utilizar sabiamente el mapa de durezas para hacer que el personaje baje de forma correcta por una bajada y suba de forma correcta por una subida: no es tan fácil como en el juego precedente, aunque

tampoco es difícil si piensas un poco.

Es tan simple como utilizar 2 colores para crear el "suelo" (verde #00ff00 y amarillo #ffff00, por poner un ejemplo). El primer color sería la línea sobre la que camina el personaje, y evitaría que este caiga hacia abajo. Tendrías que usar un if más o menos así para ello:

```
if (color != rgb_verde)
    y=y+caida;
end
```

El segundo color, situado por debajo del primero, "empujaría" al personaje hacia arriba hasta que deje de tocar ese color (dejándolo justo sobre la línea verde, que es por la que camina el personaje). Tendrías que usar un while más o menos así para ello:

```
while (color == rgb_amarillo)
    y--;
end
```

Con esto, subiría cuestras de forma impecable... y bajarlas, bueno, dependería de cómo apliques la gravedad y sobre todo, los cambios de gráfico... podría "caerse" constantemente al bajar una cuesta, o podría quedar también impecable: es cuestión de planearlo bien, nada más.

Por cierto, si te estás preguntando cómo añadir varios niveles a un juego de plataformas...¿te acuerdas del truco que utilizamos en el tutorial del RPG para pasar del mapa general al interior de una casa o al inventario? Exacto: con un Switch que iba controlando en qué nivel estábamos en cada momento y ejecutaba el bucle correspondiente hasta que salíamos de éste. Pues con los niveles de los juegos de plataformas podemos utilizar el mismo sistema. Ésa sería una manera fácil.

Otro consejo, para posibles futuros juegos de plataformas que te animes a hacer:

Uno de los fallos que muchos programadores tienen a la hora de hacer un juego tipo plataformas es la cantidad de memoria y procesador que requieren éstos. Esto se debe a que una vez que se crea una fase, si esta tiene 800 monedas por ejemplo y 500 enemigos, todos cargados a la vez, aunque no estén en pantalla, están gastando furiosamente memoria. El proceso que se presenta a continuación, "pon\_monedas()" -original de Francisco Bermúdez- evita este problema, haciendo que a medida que el personaje se vaya moviendo, las monedas y demás objetos van apareciendo, es decir: se crean un poco antes de que vayan a aparecer, y cuando los dejamos atrás, mueren. Además, si nuestro personaje eliminara alguno de esos procesos, ya sea recogiendo monedas o matando a los enemigos, dicho proceso ya no aparecerá más (como debe ser, claro).

Antes de ver el código de dicho proceso hay que saber lo siguiente:

- Las monedas (o bichos, u objetos, o lo que sea) a crear ya no se crearan como tradicionalmente: simplemente hay que llamar una sola vez al proceso pon\_monedas.

- En GLOBAL del programa hay que definir dos matrices:

A) viva[Nro de monedas]; (Para indicar si cada moneda esta viva 1 o muerta 0)

B) id\_coin[Nro de monedas]; (Identificador de cada moneda)

Donde "Nro de monedas" es el número de monedas máximo que pueda aparecer en cualquier fase del juego

- En LOCAL del programa hay que definir la siguiente variable:

Numero; (Que sera el numero de cada moneda)

Ahora pasamos a ver el proceso suponiendo que el juego esta a una resolucion de 320x240, las monedas han sido llamadas "coin", el personaje que las debe recoger es "personaje", y tenemos por ejemplo, 200 monedas como maximo.

Por cierto, un detalle interesante del código que verás a continuación y que no quería dejar pasar es que fijate que para situar las monedas (en este caso), en vez de utilizar el método bastante rudimentario que utilizamos en el tutorial anterior de depender de las posiciones de las plataformas, aquí se utiliza otro método más profesional: puntos del control dentro del mapa de scroll. Es decir, las monedas no las crearemos poniendo nosotros las coordenadas, sino colocando puntos de control en el mapa donde aparecerán dichas monedas. El proceso se encargara de quitarlas o ponerlas cuando le corresponda.

```
process pon_monedas() // Puede ser "pon_monedas", "pon_bichos"... :)
private
  int numero_de_moneda[200]; // Una tabla que indica si cada moneda esta puesta o no
  int contador; // Contador de las monedas a crear-matar-modificar
end
begin
from contador=1 TO 200 // Este bucle crea TODAS las monedas
  get_point(0,1,contador,&x,&y); //Obtiene las coord de los puntos de control de un supuesto mapa nro 1
  if (x<>0 AND y<>0) // Cuando las coordenadas son 0 quiere decir que no hay mas monedas
    id_coin[contador]=coin(x,y,contador); // Crea la moneda con las coords y su n° correspondiente
    numero_de_moneda[contador]=1; // Se indica que dicha moneda esta puesta (1)
    viva[contador]=1; // Se indica que dicha moneda esta viva (1)
  end
end
loop // Este bucle permanecera durante toda la fase actual del juego
  from contador=1 to 200 // Este bucle estudiara cada moneda
    get_point(1,1,contador,&x,&y); //Obtiene las coords de cada moneda(los pntos de control del mapa)

    /* A continuacion se hacen dos comprobaciones, la primera consiste en averiguar si una moneda que
    no esta en pantalla y que sigue viva, esta lista para ser puesta, y en la segunda comprobacion se averigua
    si una moneda que esta viva y puesta debe ser quitada de pantalla para ahorrar memoria.*/

    /* 1ro Si en las coordenadas hay o habia moneda y se encuentra cerca del personaje y no esta puesta en
    escena, pero esta viva*/
    if (x<>0 AND y<>0 AND x<id_personaje.x+320 AND x>id_personaje.x-320 AND
        y<id_personaje.y+240 AND y>id_personaje.y-240 AND numero_de_moneda[contador]==0 AND
        viva[contador]==1)
      // Creamos dicha moneda
      id_coin[contador]=coin(x,y,contador);
      // Y ahora indicamos que esta puesta
      numero_de_moneda[contador]=1;
    end

    /* 2do Si en las coordenadas hay o habia moneda y se encuentra lejos del personaje y esta puesta en
    escena y viva*/
    if (x<>0 AND y<>0 AND (x>id_personaje.x+320 OR x<id_personaje.x-320 OR y>id_personaje.y+240
    OR y<id_personaje.y-240) AND numero_de_moneda[contador]==1 AND viva[contador]==1)
      // Quitamos la moneda para ahorrar memoria
      signal(id_coin[contador],s_kill);
      // E indicamos que no esta puesta (aunque si viva)
      numero_de_moneda[contador]=0;
```

```

end
end //From
frame;
end //Loop
end

```

Ademas de todo esto, hay que tener en cuenta que cuando el personaje toque una moneda, ésta de morir definitivamente, cambiando su variable "viva" a 0. Deberíamos incluir en el proceso del personaje algo como esto:

```

IF (id2=collision(TYPE coin)) // Si toca una moneda
    viva[id2.numero]=0; // le quita la vida
    signal(id2,s_kill); // y la quita de pantalla
END

```

En tu afán por mejorar y ampliar tu juego de plataformas, se te puede ocurrir introducir plataformas móviles. Esto complica considerablemente la programación, pero ofrece un acabado del juego muy superior al estándar. Básicamente, si quieres implementar un sistema de plataformas móviles, lo que tendrías que hacer es que el bucle del personaje principal fuera similar a lo siguiente (hablando abstractamente):

*/\*'avance' es una variable local que se refiere a la dirección hacia la que el personaje mira (-1 para izquierda y +1 para la derecha),  
'inc\_x' es una variable privada que se refiere al incremento horizontal oportuno en el movimiento del personaje(según esté agachado, levantado, saltando) en valor absoluto.  
'impulso' es una variable privada que cuanto más grande sea más impulso vertical tendrá el protagonista, y por tanto, más arriba.  
'velocidad' es una variable local que indica la velocidad de desplazamiento de una plataforma  
'en\_plataforma' es una variable privada al personaje que adopta valores 'si' o 'no' (1 ó 0) e indica si estoy en una plataforma o no  
'id2' lo uso para recoger las colisiones con plataformas\*/*

Loop

```

// ACTUALIZAR x
If(en_plataforma) // ¿Estoy en una plataforma?
    x=x+id2.velocidad;
End
x=x+inc_x*avance; // ahora actualizo: le sumo a x el incremento en la dirección adecuada

// ACTUALIZAR y (algoritmo típico de salto)
If(impulso<>0) y=y-impulso;impulso--; End

// Calcula la gravedad, mi gravedad va aparte, más adelante te la enseño.
id_gravedad=gravity();

// COLISION CONTRA UNA PLATAFORMA
id2=collision(Type plataforma); // Cojo el ID de la plataforma
If(id2)
    // Si esta sobre ella y callendo
    If(y<id2.y+2)
        en_plataforma=si; saltando=no;
    Else
        en_plataforma=no;
    End
End

```

```

        End
    Else
        en_plataforma=no;
    End

    /* Se mantiene junto a la plataforma, ajusto automáticamente la 'y' del personaje en
    plataformas verticales*/
    If(en_plataforma==si) y=id2.y; End
    Frame; // Refresco
End

Y el resto de procesos necesarios...:

Process gravity()
Begin
ctype=c_scroll;

/* Si el proceso afectado por la gravedad se encuentra en una plataforma, queda afectado por otros
factores determinados en su propio bucle principal. En caso contrario... */
If(father.en_plataforma<>1)
    // El color 255 (rojo) es el suelo
    If(suelo(father.x,father.y)<>255)
        father.saltando=1;
    Else
        father.saltando=0;
        gravedad=1;
    End

    // La caída se controla punto a punto
    If(father.saltando==1)
        For (yy=father.y; yy<=father.y+gravedad; yy++)
            If(suelo(father.x,yy)==11) father.y=yy; Break; End
        End
    End

    // El límite de la gravedad es 75
    If(father.saltando==1)
        If(gravedad < 75); gravedad+=8.5; Else; gravedad=75;End
        // Se actualiza la posición del objeto afectado
        father.y+=gravedad;
    End
End
End

// El desplazamiento sigue el movimiento armónico simple (como el del muelle)
Process plataforma(file,graph,x,y,tipo,desp,velocidad)
Begin
ctype=c_scroll;
z=id_personaje.z+1;
cx=x;
cy=y;

// Plataforma Horizontal
If(tipo==0) xx=x-desp; yy=x+desp;End

```

```
// Plataforma Vertical
If(tipo==1) xx=y-desp; yy=y+desp; End

Loop
  // Desplazamiento horizontal
  If(tipo==0)
    If(x<xx OR x>yy) velocidad=-velocidad; End
    x+=velocidad;
  End

  // Desplazamiento vertical
  if(tipo==1)
    If(y<xx OR y>yy) velocidad=-velocidad; End
    y+=velocidad;
  End
  Frame;
End
End
```

## **CAPÍTULO 12: TUTORIAL PARA EL JUEGO “SANTA ATTACK”**

(extraído del tutorial de Wakroo, en <http://divnet.divsite.net>)

En este capítulo voy a comentar por encima cómo se ha programado el juego “Santa Attack” (de Wakroo) siguiendo la estructura del PRG; y al final del capítulo tendrás el código fuente completo para poderlo seguir sin problemas.

Este juego consiste básicamente en manejar una nave con inercia (Santa Klaus), la cual puede disparar regalos, que deberán impactar –si el jugador quiere conseguir puntos y ganar la partida- en las chimeneas de las casas que aparecen en la parte inferior de la pantalla, las cuales se van moviéndose en un scroll horizontal infinito, apareciendo por la derecha de la pantalla y desapareciendo por la izquierda.

### **La intro del juego:**

Una vez que se han declarado las constantes y las globales (en este programa no hay locales) viene el main (proceso principal). En él se inicializa el modo gráfico a 640x480, 16 bits y se cargan los FPG, FNT y WAV que se usan en el juego. Después se llama al proceso intro(), que es el encargado de poner las dos imágenes de presentación. El proceso es muy sencillo: se pone el gráfico 101 (la idea es que sea el logotipo del juego) en el centro, se espera 50 frames, se hace un fundido lento a negro, se cambia la imagen por la 102 (la idea es que sea el logotipo de la empresa desarrolladora), se vuelve a encender la pantalla, se espera otros 50 frames, se hace un fundido lento a negro se cambia la resolución a 800x600, se ponen estrellas en el fondo, se llama al proceso juego() y se vuelve a encender la pantalla.

A ver, poco a poco. Fíjate en que primero se ha inicializado la resolución a 640x480 y luego se ha puesto a 800x600. Esto se hizo para ahorrar memoria, porque las dos imágenes del principio en general pueden ocupar bastante, y poniéndolas a una resolución menor no ocupan tanto. Con esto se consigue obtener un fichero ejecutable de un tamaño menor y más manejable. El resto del proceso se ha preparado para que el programa se comporte de una manera secuencial. Hace una cosa y no sigue hasta que ha terminado ésta. Esto es muy útil para las presentaciones de los juegos, ya que se pueden preparar todas las cosas, se dejan durante un tiempo en pantalla y luego se preparan las siguientes cosas. Con esto se pueden hacer escenas cinemáticas a base de imágenes fijas o pequeñas animaciones.

Te habrás dado cuenta de que en el while hemos usado otra vez la variable global predefinida *FADING*. Recuerda que esta variable informa del estado de la operación de fundido *fade* en curso para cualquier proceso del juego: valdrá 0 –falso- si no hay fundidos en activo y 1 –verdadero- si todavía hay un fundido en marcha.

En este caso no se ha dado la posibilidad de saltarse la intro porque es corta. De todas formas, es recomendable dar la posibilidad para intros largas, además de ser muy fácil de hacer. Se podría haber creado otro proceso o haber usado el main (yo me quedo con esta opción). Se hace un LOOP y dentro se pone la condición para saltarse la intro (pulsar la tecla escape o cualquier tecla, por ejemplo) y su correspondiente FRAME; (para que no se bloquee el programa). Entonces se hace una transición bonita y, una vez acabada, se hace un *let\_me\_alone()*. De esta manera el proceso se queda solo, es decir, se carga todos los procesos que estuvieran pululando por ahí para hacer la intro. Luego se llama al proceso encargado de lanzar el juego o el menú, lo que corresponda. Esta misma técnica se puede usar al final de la intro para cargarse el proceso que está controlando si el usuario se la quiere saltar o no. Recomiendo usar la técnica de salir del bucle con un *break*; y poner estas instrucciones al final.

### **Inicio del juego:**

Finalmente están las estrellas del fondo. Fíjate que usamos la función: *put\_pixel* para pinta con un color aleatorio los píxeles del fondo de la ventana. Usar *put\_pixel* es una manera sencilla de conseguir un fondo de estrellas sencillo pero efectivo y sin aumentar el tamaño del juego, ya que no hace falta un mapa. Y

esto es todo lo que tiene la intro. Pasemos ahora al proceso que controla el juego.

Todo el control del juego recae sobre el proceso `juego()`. Es él el encargado de llamar a todos los demás procesos necesarios. Esto es útil en juegos más complejos para, por ejemplo, pausar el juego (`signal(TYPE juego,s_freeze_tree)`, que congela el proceso `juego` y todos sus descendientes). También hace posible arrancar el juego desde distintas partes del código y, si se usa algún parámetro, configurarlo con facilidad (en qué fase se quiere empezar, por ejemplo). En este caso todas esas opciones no se han usado porque no hacían falta, pero bueno, ahí están.

Primero se llama a los procesos `santa(x,y,graph)`, `luna(x,y,graph)` y `crea_casas()`. Estos son todos los procesos necesarios para el juego y tienen sus propias secciones. También se escribe en la esquina superior izquierda el valor de la variable `puntos`. Después de esto tenemos un bucle LOOP. Lo primero que hay es una comprobación de la variable `puntos`. Si es mayor que la constante `puntos_victoria` entonces se ha ganado. Se pone un texto en pantalla y se espera 100 frames. Luego se sale del bucle. Si no se cumple esa condición se comprueba si `velCasas` es mayor que `velCasas_victoria` para ver si se ha perdido. Se pone un texto, se espera 100 frames y luego sale del bucle. El último IF comprueba si se ha pulsado el escape y, si es así, sale del bucle. A continuación del bucle hay dos sentencias. La primera es `let_me_alone()`, que mata todos los procesos menos el que la ejecuta. La otra es `exit("")`, que termina el programa. En principio la primera sentencia no debería ser necesaria, ya que se supone que el `exit` lo hará todo, pero bueno, por si acaso lo pongo.

Fijaos en un pequeño detalle bastante obvio. Dentro del bucle, cuando se dan las condiciones necesarias, no he puesto las sentencias para acabar el programa, sino que he roto el bucle y las he puesto a continuación. De esta manera se ahorran líneas de código. Imaginad que fueran 8 sentencias las que hay que hacer para terminar esta zona (porque hay que volver al menú o hacer lo que sea), por lo que tendríamos que poner las 8 sentencias en cada ocasión que queramos terminarlo -que son 3-, lo que hace un total de 24 sentencias... en contra de las 11 que se usan de esta forma (los 3 breaks y las 8 sentencias que están fuera). La otra gran ventaja es la facilidad para cambiarlo si en el futuro se nos ocurre algo mejor. Si se decide que en vez de salir hay que llamar otra vez al proceso `juego()` habría que modificarlo en cada sitio donde estuviera puesto, con el riesgo de olvidarse alguno. De esta manera con modificar uno bastaría. Pasemos ahora a la nave protagonista del juego.

### **La nave:**

Los procesos relacionados con la nave son 3: `santa(x,y,graph)`, `fuego(x,y,angle,graph)` y `retro(x,y,angle)`. Se tratarán por separado excepto en aquellos aspectos en los que se solapen. Empecemos por `santa`. Este proceso es el que pone en pantalla el trineo a reacción y su movimiento. Al principio crea un `regalo()`, que será tratado en otra sección. Luego entra en el bucle que lo controla. Empieza sumando las variables "velx" a X y "vely" a Y, pero divididas entre 10. Puesto que quería que la nave tuviera inercia, creé dos variables para guardar la velocidad horizontal y la vertical de ese momento, que cada turno se suma a las variables X,Y. Además, se le suma la gravedad a la velocidad vertical (hay que recordar que la variable Y aumenta según se baja). Como se han usado enteros (X,Y lo son, de todas formas) esto le da más precisión a los movimientos y permite, por ejemplo, sumar "medio pixel" a `velx`. Luego, al dividir, se desprecian las fracciones, en realidad no afectará al movimiento, pero la próxima vez que se le sume medio pixel ya harán un pixel entero y eso sí afecta. Este mismo efecto se podría hacer con la variable local `RESOLUTION=10`, que hace que todas las magnitudes estén multiplicadas por 10 (`x=1000` sería la coordenada 100 de pantalla), pero de la manera que lo he hecho es más simple.

A continuación se controla si el jugador quiere rotar la nave y se obliga al valor de `angle` a estar entre 0 y 359999 (en milésimas de grado). Esto se hace para que la rutina de los regalos funcione correctamente, ya que `angle` puede valer 45000 o 405000, que la rotación del gráfico será la misma. Con este ángulo se calculan las nuevas velocidades cuando se activan los propulsores. El sistema es sencillo. Recuerda que mediante las funciones `get_distx`, `get_disty` se puede obtener la proyección de una distancia dada con un ángulo dado sobre los ejes x,y. Con esto se consigue que el impulso sea de una cantidad fija y en cualquier

dirección. En la función hay que poner el ángulo de la dirección en la que se quiere hacer la propulsión. Por lo tanto, para el propulsor trasero es, simplemente, el angle de la nave (hacia delante) y para los inferiores el  $\text{angle}+90000$  (hacia arriba). El último parámetro es la magnitud que hay que proyectar. Puesto que los propulsores inferiores tienen dos posibles fuerzas no es lo mismo darle a espacio o a la tecla arriba, aunque la fuerza se hará en la misma dirección. Además, como no tiene que ser posible activar los propulsores a potencia media y fuerte a la vez primero se comprueba la potencia fuerte con un IF y dentro de su ELSE se comprueba la potencia media. Con la tecla alt el procedimiento es el mismo pero para el propulsor delantero. Finalmente se hace que la nave no pueda salir de la pantalla.

### **Los propulsores:**

Pasemos ahora a los procesos fuego (para los propulsores inferiores) y retro (para el trasero), que son prácticamente iguales. Primero reproduce un sonido y luego asigna  $z=\text{father.z}+2$ . Con esto se asegura que el gráfico estará por detrás del de la nave. Luego hay un par de funciones *xadvance*, que lo que hacen es mover el gráfico hasta donde tiene que estar, partiendo de las coordenadas del padre (los parámetros). Recuerda que esta función desplaza el proceso actual en la dirección especificada por el ángulo indicado como primer parámetro, el número de unidades indicado como segundo parámetro.

Ahora hay que fijarse en el FPG. Las posiciones 2, 3 y 4 las ocupan el mismo gráfico (los fuegos inferiores) pero de distintos tamaños verticales. Recuerda que en Fenix existe la posibilidad de escalar horizontal o verticalmente un gráfico mediante las variables locales predefinidas *SIZE\_X* y *SIZE\_Y*, indicándoles un porcentaje (asignarles el valor 100 equivaldría a no escalar nada), pero en este ejemplo no las usaremos, para que veas una manera alternativa. Lo que haremos será dibujar 3 gráficos con los distintos tamaños (de menor a mayor).

Volvamos al código. El valor que se le ha asignado a GRAPH es el valor del gráfico "normal". En este momento se coge el gráfico anterior. Los motivos los entenderéis al final. Luego se hace un FRAME. A continuación se reduce en 1 la Z. Aún así, todavía queda por debajo de la nave, ya que antes se ha aumentado en 2 con respecto a su Z. Luego se aumenta en 1 el gráfico para volver a tener el original, se cogen la X, Y, ANGLE del padre y se vuelve a poner el gráfico en su sitio. Así, aunque el padre se mueva los fuegos le seguirán. Se hace otro FRAME. Luego se vuelve a coger el gráfico anterior y se aumenta Z otra vez. Se ajusta la posición con respecto a la nave y se hace FRAME.

¿Cuál es la razón de todos estos cambios en GRAPH y Z? Muy sencillo. Cada vez que se pulsa espacio o la tecla arriba se hace una llamada al proceso  $\text{fuego}(x,y,\text{angle},\text{graph})$  (la diferencia es el gráfico: 3 en el de potencia media y 4 en el de fuerte). Si os fijáis en el proceso fuego veréis que tiene 3 FRAMES pero ningún bucle. Por lo tanto, estará en pantalla 3 FRAMES y luego desaparecerá. En el primer FRAME se pone el gráfico anterior, que es más pequeño que el original, y se pone como Z la del padre +2. En el siguiente FRAME se pone el gráfico original (más grande) y se disminuye en 1 la z. Con esto se consigue que quede por debajo del gráfico del padre pero por encima de cualquier fuego que esté en el primer FRAME. Finalmente en el tercero se vuelve a poner el gráfico más pequeño y la Z de antes para que quede por detrás de cualquier fuego que esté en el segundo paso.

Todo este lío se ha hecho para conseguir el efecto de que el fuego "crece y disminuye". Si os fijáis un poco en el juego el fuego no sale completamente desde el principio, sino que primero sale más pequeño y luego a tope. A la hora de apagarse pasa lo mismo pero al revés. El problema que tiene este sistema es que, en realidad, el fuego va tarde. Suponiendo que se pulsa la tecla durante un FRAME, en ése sale un fuego pequeño, en el siguiente más grande y en el último uno pequeño. Pero el efecto de la impulsión sólo se dará en el primer FRAME. De todas formas, como es muy poco probable que alguien consiga una impulsión de un solo FRAME y era la forma más sencilla de darle un toque bonito decidí que no importaba ese retraso. Queda el proceso  $\text{retro}(x,y,\text{angle})$ , pero como funciona prácticamente igual que el proceso  $\text{fuego}(x,y,\text{angle},\text{graph})$  no hace falta comentar nada más. Sigamos adelante con los regalos.

### **Los regalos:**

El objetivo del juego es "bombardear" las chimeneas con regalos. Por lo tanto, el comportamiento de este proceso es vital para el desarrollo del juego, aunque, en realidad, esta parte está hecha de una forma un poco chapucera.

Lo primero sobre lo que quiero llamar la atención es sobre los bucles anidados que he usado. El regalo puede estar en 3 momentos distintos:

- Salir del depósito hasta su posición al final del tubo de lanzamiento.
- En espera de ser "disparado" al final del tubo de lanzamiento.
- Cayendo después de ser "disparado".

Puesto que el comportamiento es distinto en cada momento el código debe ser distinto. Por lo tanto, creé un bucle general con otros 3 bucles dentro, uno para cada momento. Cuando hacía falta rompía el LOOP y dejaba que fuera al siguiente. Al salir del último volvía al primero al estar todos en un gran círculo sin fin. Así era mucho más sencillo programar cada parte.

Nada más empezar el bucle principal se elige uno de los gráficos al azar. Como pasará por aquí en cada vuelta cambiará el gráfico después de haber hecho un ciclo completo. A continuación se entra en el momento de "carga del regalo". El gráfico sale poco a poco de la tobera hasta ponerse en su sitio. Para eso en cada vuelta se cogen las coordenadas y ángulo del padre, se mueve hacia atrás y se gira hasta ponerse en línea con la tobera. Luego se avanza según cuánto valga la variable privada *aux* y se aumenta *aux* en 2. Con esto se consigue que a cada FRAME el paquete esté un poco más fuera, por lo que da la sensación de que se está poniendo en posición. Finalmente, si  $aux=24$  se rompe el bucle porque ya estará en su sitio.

Pasemos ahora al segundo momento. Éste simplemente coge las coordenadas y el ángulo del padre y mueve el gráfico hasta ponerlo en su sitio, al final del tubo de lanzamiento. Luego espera hasta que se pulse la tecla control, que es la que "disparará" el regalo. Finalmente se ajusta el ángulo dentro del intervalo 0 - 359999 y viene el bucle que hace que el regalo caiga. Como no quería complicarme la vida lo hice muy sencillo. Se comprueba que el ángulo está entre 90000 y 270000 (mitad izquierda). En ese caso se aumenta el ángulo, lo que lo acerca a la vertical hacia abajo. Si no es así, se comprueba que la diferencia entre el ángulo y 90000 es mayor que el giro que se le da. Si es así estará en la mitad derecha pero aún no habrá llegado a la vertical, por lo que se disminuye el ángulo (lo gira hacia el otro lado). Si ya está lo suficientemente cerca se le asigna 90000 y ya está. Puede parecer que el regalo nunca cogerá la vertical si gira en el sentido contrario a las agujas del reloj, lo cual puede ser cierto dependiendo del ángulo de lanzamiento. Pero como el ángulo de giro es pequeño y en un ciclo girará hacia un lado y en el siguiente al otro no se nota la diferencia.

Para terminar se comprueba la colisión con la chimenea. Se guarda el ID del proceso con el que se ha chocado en *id2* y se usa para acceder a sus valores de *x,y*. Con esto se determina si ha impactado desde arriba y por el centro para contarlos como punto. Si no, se pone una pequeña animación de "explosión". Si no se ha chocado contra una chimenea se comprueba si se ha chocado contra una casa para poner la animación.

A lo mejor os estáis preguntando porqué he creado un bucle sin fin y he controlado el lanzamiento desde el proceso regalo y no desde santa. La razón es muy sencilla. Con el código del regalo de manera secuencial y haciendo una única llamada es imposible que haya dos regalos pululando por la pantalla a la vez. El regalo se pondrá en posición, esperará hasta que se lance, bajará, se pondrá en posición, esperará hasta que se lance, bajará,... Así para siempre jamás y sin necesidad de ninguna variable ni código que compruebe si hay algún regalo para volver a llamar al proceso, ya que es único. Con esto ya hemos dejado atrás la parte más "complicada" del juego. Ahora pasemos a las casas y chimeneas.

### **Las casas y las chimeneas:**

El primer proceso de este bloque es `crea_casas()`. Su función es muy sencilla: crea 3 casas para que vayan pasando por la pantalla y luego entra en un bucle infinito que va modificando la velocidad a la que pasan dependiendo del número de casas que hayan pasado ya. Decidí simplemente hacer que la velocidad fuera el número de casas dividido por un número (definido en la constante `cantCasas` y +1 para que no pueda ser 0). De esta manera al principio las casas irán despacio, pero pasadas `cantCasas` casas empezarán a ir más rápido. Cuando hayan vuelto a pasas `cantCasas` volverá a acelerar, pero esta vez sucederá antes porque pasan más rápido. Por lo tanto, la velocidad no aumenta de una manera lineal. Se podrían haber usado otros métodos para conseguir este efecto o haberlo hecho de otra manera, pero éste me gustó.

Luego está el proceso `casa`. Primero crea una chimenea propia y exclusiva y luego se asegura de quedar por encima de cualquier casa que se haya creado antes y con la que esté colisionando. Como se crean con coordenadas al azar (dentro de unos límites) esto es algo no sólo posible, sino incluso probable. A continuación entra en un LOOP que mueve la casa y comprueba si ha salido por el lado izquierdo. En ese caso crea un nuevo proceso `casa(x,y)` y sale del bucle para dar por terminado el proceso.

La chimenea recibe dos parámetros que determinan dónde estará colocada con respecto a las coordenadas de la casa. Luego entra en el LOOP donde sigue a la casa que lo ha creado (su padre) y comprueba a ver si ya no existe. En ese caso sale del bucle para terminar también con él mismo.

Y por último, no hace falta comentar lo que hace el proceso `luna`, ¿verdad?

### **Código fuente completo:**

Antes de nada, para poder ejecutar el juego necesitarás los siguientes recursos:

-Cinco archivos WAV llamados “`buiu.wav`”, “`victoria.wav`”, “`fallo.wav`”, “`hit.wav`” y “`motor.wav`”. Su uso, se puede ver claramente en el código, es opcional. Si no los puedes obtener, comenta las líneas del código donde aparezcan.

-Un archivo FNT llamado “`fuentes.fnt`”.

-Un archivo FPG llamado “`juego.fpg`”, que incluirá todas las imágenes del juego. Éstas imágenes tendrán los códigos siguientes:

- 001: La nave espacial de Santa
- 005: El propulsor de la nave
- 011: Regalo
- 012: Regalo
- 013: Regalo
- 014: Regalo
- 015: Regalo
- 021: Luna
- 026: Explosión
- 027: Explosión
- 028: Explosión
- 029: Explosión
- 051: Casa
- 071: Chimenea
- 101: Logotipo del juego
- 102: Logotipo del autor

Y ahora, por fin, se muestra el código fuente completo, con comentarios, del juego:

```
/******\
|| Programa: Santa Attack ||
|| Autor: Wakroo ||
|| Grupo: Emerald Works ||
\*****/

Program Santa_Attack;

// Declaración de constantes
Const
  grav=2; // Gravedad

  // Control de la velocidad de las casas
  cantCasas=5;

  // Velocidad de giro de los regalos al hacer
  // el arco hasta bajar en vertical
  ang_regalo=5000;

  // Número de regalos que hay que superar para ganar
  puntos_victoria=40;
  // Velocidad de las casas en las se pierde
  velCasas_victoria=5;

  // Fuerza de los propulsores
  prop_fuerte=10;
  prop_media=5;
End

Global
  Int velCasas=2; // Velocidad de las casas
  Int numCasas=0; // Número de casas
  Int puntos=0; // Número de aciertos
  Int fuente; // Variable para cargar la fuente de los textos

  // Sonidos
  int buiu;
  int hit;
  int motor;
  int fallo;
  int victoria;
End

Begin
  // Inicialización del modo gráfico a 640x480, 16 bits
  set_mode(640,480,16);

  // Carga de los recursos
  load_fpg("juego.fpg");
```

```

fuente=load_fnt("fuente.fnt");

buiu=load_wav("buiu.wav");
hit=load_wav("hit.wav");
motor=load_wav("motor.wav");
fallo=load_wav("fallo.wav");
victoria=load_wav("victoria.wav");

// Llamada al proceso intro()
intro();
End

/*****
| Proceso: intro()
| Función: Pone las pantallas iniciales
| Parámetros:
| *****/

Process intro()
Begin
// Posición y gráfico en pantalla
x=320; y=240; graph=101;

// Esperar durante 50 frames
From z=0 To 49; Frame; End

// Fundido lento a negro (apagar)
fade(0,0,0,2);
// Esperar durante el fundido
While (fading) Frame; End

// Cambiar el gráfico
graph=102;
// Encender
fade_on();

// Esperar 50 frames
From z=0 To 49; Frame; End

// Fundido lento a negro (apagar)
fade(0,0,0,2);
// Esperar durante el fundido
While(fading) Frame; End

// Cambiar la resolución a 800x600
set_mode(800,600);
// Poner puntos aleatorios en el fondo (estrellas)
From z=0 To 9999; put_pixel(rand(1,799),rand(1,599),rand(65300,65535)); End

// Llamada al proceso juego()
juego();

// Encender la pantalla
fade_on();

```

End

```
/******|
| Proceso: juego() |
| Función: Arranca el juego y controla la salida |
| Parámetros: |
|******/
```

*Process juego();*

*Begin*

*// Llamadas a los procesos santa(x,y,graph),*

*// luna(x,y,graph), crea\_casas()*

*santa(320,240,1);*

*luna(150,150,21);*

*crea\_casas();*

*// Se escriben los puntos que tiene el jugador*

*// con la fuente del sistema*

*write(0,10,10,0,"Puntos");*

*write\_var(0,60,10,0,puntos);*

*Loop*

*// Se comprueba si se ha ganado*

*If (puntos>puntos\_victoria)*

*play\_wav(victoria,0);*

*write(fuente,400,300,4,"¡Has repartido todos los regalos!");*

*From z=0 To 100; Frame; End*

*Break; // Se sale del loop*

*Else*

*// Se comprueba si se ha perdido*

*If (velCasas>velCasas\_victoria)*

*play\_wav(fallo,0);*

*write(fuente,400,300,4,"Has fallado");*

*From z=0 To 100; Frame; End*

*Break; // Se sale del loop*

*End*

*End*

*// Si se pulsa escape sale del loop*

*If (key(\_esc))*

*Break;*

*End;*

*Frame;*

*End*

*// Mata a todos los procesos menos éste*

*let\_me\_alone();*

*// Sale del programa*

*exit("");*

End

```
/******|
| Proceso: santa(x,y,graph) |
```

```

| Función: Protagonista del juego |
| Parámetros: |
| x: posición horizontal |
| y: posición vertical |
| graph: gráfico |
| ***** |

Process santa(x,y,graph)
Private
  Int velx = 0; // Velocidad horizontal
  Int vely = -grav; // Velocidad vertical
End

Begin
  regalo();

  Loop
    // Se aumentan x,y con velx,vely / 10
    // (para tener más precisión)
    // NOTA: también se podría haber usado RESOLUTION
    x=velx+velx/10; y=vely+vely/10;
    vely=vely+grav; // Se tiene en cuenta la gravedad

    // Se rota la nave
    If (key(_left)) angle=angle+7500; End
    If (key(_right)) angle=angle-7500; End

    // Se ajusta el ángulo para que el valor esté siempre
    // entre 0 y 359999 (en milésimas de grado)
    If (angle>359999) angle-=360000; End
    If (angle<0) angle+=360000; End

    // Propulsores inferiores a toda potencia
    If (key(_space))
      // Se modifican las velocidades horizontal y vertical
      // según el ángulo de la nave
      velx=velx+get_distx(angle+90000,prop_fuerte);
      vely=vely+get_disty(angle+90000,prop_fuerte);
      // Se llama al proceso fuego(x,y,angle,graph)
      fuego(x,y,angle,4);
    Else
      // Propulsores inferiores a media potencia
      If (key(_up))
        velx=velx+get_distx(angle+90000,prop_media);
        vely=vely+get_disty(angle+90000,prop_media);
        fuego(x,y,angle,3);
      End
    End

    // Propulsor trasero
    If (key(_alt))
      velx=velx+get_distx(angle,prop_media);
      vely=vely+get_disty(angle,prop_media);
      retro(x,y,angle);
    End
  End
End

```

```

End

// Que la nave no pueda salir de la pantalla
If (y<0) y=0; vely=0; End
If (y>600) y=600; vely=0; End
If (x<0) x=0; velx=0; End
If (x>800) x=800; velx=0; End

Frame;
End
End

```

```

/*****
| Proceso: fuego(x,y,angle,graph)      |
| Función: Propulsores inferiores     |
| Parámetros:                         |
| x: posición horizontal              |
| y: posición vertical                |
| angle: ángulo                       |
| graph: gráfico                      |
| *****/

```

```

Process fuego(x,y,angle,graph)
Begin
// Reproduce un sonido
play_wav(motor,0);

// Se pone por debajo del padre
z = father.z + 2;
// Se coloca en posición
xadvance(father.angle-90000,18);
xadvance(father.angle,1);
// Se usa el gráfico anterior
graph=graph-1;
Frame;

// Se pone por debajo del padre
// pero por encima de otros fuegos
z=z+1;
// Se usa el gráfico original
graph=graph+1;
// Se pone en posición
angle=father.angle;
x=father.x; y=father.y;
xadvance(father.angle-90000,18);
xadvance(father.angle,1);
Frame;

// Se usa el gráfico anterior
graph=graph-1;
// Se pone por debajo del padre
// y otros fuegos
z=z+1;
// Se pone en posición

```

```

angle=father.angle;
x=father.x; y=father.y;
xadvance(father.angle-90000,18);
xadvance(father.angle,1);
Frame;
End

```

```

/*****\
| Proceso: retro(x,y,angle)
| Función: Propulsor trasero
| Parámetros:
| x: posición horizontal
| y: posición vertical
| angle: ángulo
| *****/

```

```

Process retro(x,y,angle)
Begin
play_wav(motor,0);

z = father.z + 2;
xadvance(father.angle+90000,5);
advance(-42);
graph=5;
Frame;

```

```

Z=z-1;
Graph=graph+1;
angle=father.angle;
x=father.x; y=father.y;
xadvance(father.angle+90000,5);
advance(-42);
Frame;

```

```

Graph=graph-1;
Z=z+1;
angle=father.angle;
x=father.x; y=father.y;
xadvance(father.angle+90000,5);
advance(-42);
Frame;

```

```
End
```

```

/*****\
| Proceso: regalo()
| Función: Regalos con los que "bombardear"
| Parámetros:
| *****/

```

```

Process regalo()
Private
Int id2;
Int aux;
End

```

```

Begin
Loop
// Gráfico aleatorio
graph=rand(11,15);
aux=0;

// Bucle para que el regalo se ponga en posición
Loop
// Se pone el regalo en las coordenadas y el ángulo
// de la nave (father = ID del que lo ha llamado)
x=father.x; y=father.y;
angle=father.angle;
// z un punto mayor que el padre para que esté por debajo
z=father.z+1;

// Se pone en su sitio (la tobera)
advance(-20);
angle+=210000;
advance(aux);

// Se aumenta aux para la siguiente vuelta
aux=aux+2;

// Cuando está en su sitio se sale del loop
If (aux==24) Break; End

Frame;
End

// Bucle para que el regalo espere en posición
Loop
x=father.x; y=father.y;
angle=father.angle; z=father.z+1;

advance(-20);
angle=angle+210000;
advance(24);

// Se "dispara" el regalo
If (key(_control)) Break; End

Frame;
End

z=75;

// Se ajusta el ángulo
If (angle>359999) angle=angle-360000; End

// Bucle para que el regalo baje en vertical
Loop
advance(10);

```

```

// Se corrige el ángulo hasta llegar a la vertical
// dependiendo de la dirección
If (angle>90000 AND angle<270000)
    angle=angle+ang_regalo;
Else
    If ((angle+90000)>ang_regalo)
        angle=angle-ang_regalo;
    Else
        angle=-90000;
    End
End
End

// Comprueba si ha chocado con la chimenea
If (id2=collision(Type chimenea))
    // Comprueba si ha entrado en la chimenea
    If (x>id2.x-9 AND x<id2.x+9 AND y<id2.y-10)
        Puntos=puntos+1;
        play_wav(buiu,0);
        Break;
    Else // Rompe el paquete
        play_wav(hit,0);
        // Animación de la "explosión"
        From graph=26 To 29; Frame; End
        Break;
    End
Else
    // Comprueba si ha chocado con una casa o se
    // ha salido de la pantalla por abajo
    If (y>610 OR collision(Type casa))
        play_wav(hit,0);
        From graph=26 To 29; Frame; End
        Break;
    End
End
End

Frame;
End
End
End

/*****|
| Proceso: crea_casas() |
| Función: Proceso que crea las casas |
| Parámetros: |
| *****/

Process crea_casas()
Begin
    // Crea 3 casas iniciales
    casa(960,rand(495,605));
    casa(rand(1260,1360),rand(495,605));
    casa(rand(1660,1760),rand(495,605));

Loop

```

```

// Determina la velocidad de las casas dependiendo
// del número de casas que han pasado
velCasas=(numCasas/cantCasas)+1;

```

```

Frame;
End
End

```

```

/*****\
| Proceso: casa(x,y) |
| Función: Casa |
| Parámetros: |
| x: posición horizontal |
| y: posición vertical |
| *****/

```

```

Process casa(x,y)

```

```

Begin

```

```

graph=51;
z=100;

```

```

// Crea su chimenea
chimenea(rand(-130,130),-100);

```

```

// Se pone por encima de cualquier casa
// con la que colisione
If (collision(Type casa)) z=z+1; End

```

```

Loop

```

```

// Se mueve la casa según la velocidad actual
x=x-velCasas;

```

```

// Crea una nueva casa y termina el proceso
If (x<-160)
casa(rand(960,1060),rand(495,605));
Break;
End

```

```

Frame;
End

```

```

// Aumenta el número de casas que han pasado
numCasas=numCasas+1;

```

```

End

```

```

/*****\
| Proceso: chimenea(desplx,desply) |
| Función: Chimenea |
| Parámetros: |
| desplx: posición horizontal con |
| respectó al centro de la casa |
| desply: posición vertical |
| *****/

```

```

| respecto al centro de la casa |
|*****|

Process chimenea(int desplx,int desply)
Begin
  graph=71;
  z=father.z-50;

  Loop
    // Se asignan las coordenadas del padre
    // más el desplazamiento
    x=father.x+desplx;
    y=father.y+desply;

    // Si no está el padre, adiós a la chimenea
    If (NOT father) Break; End

  Frame;
End
End

|*****|
| Proceso: luna(x,y,graph) |
| Función: Luna del fondo |
| Parámetros: |
| x: posición horizontal |
| y: posición vertical |
| graph: gráfico |
|*****|

Process luna(x,y,graph)
Begin
  z=500;
  Loop Frame; End
End

```

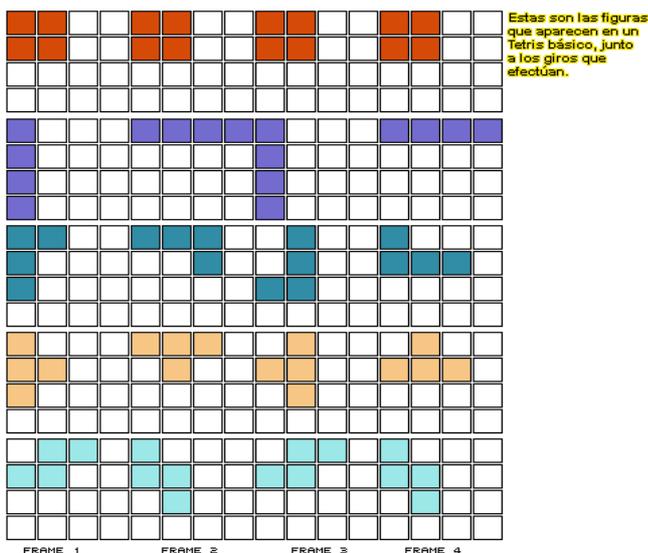
## CAPÍTULO 13: TUTORIAL DE UN TETRIS

(extraído de Macedonia Magazine, en <http://usuarios.lycos.es/macedoniamagazine>)

### Antes de empezar...:

Antes de comenzar a realizar un juego debemos de proceder a analizar todas las dificultades que nos podemos encontrar a la hora de codificar. La mecánica del Tetris es bien sencilla. El jugador ha de colocar las piezas que le van apareciendo por la parte superior de la pantalla dentro de una tabla de  $m$  filas y  $n$  columnas. Se ha de intentar que las piezas que van cayendo se sitúen de tal forma que formen una línea con las que ya se encuentren en la tabla (con los fragmentos de otras piezas). En caso de que lo logremos, las porciones de las piezas que intervienen en la formación de la misma línea desaparecerán haciendo que todas las que se encuentren de esa línea, para arriba, desciendan una fila hacia abajo (y, por lo tanto, dejen más filas para que el jugador pueda manipular las piezas futuras). En el caso de que el jugador no logre hacer una línea la pieza quedará ubicada íntegramente, con lo que limitará el espacio de juego, haciendo que aumente la dificultad para colocar la pieza que venga a continuación.

Otro de los aspectos que no podemos pasar por alto, es el referido a la forma y orientación de las piezas que manejamos en el juego. Si nos fijamos, hay un total de 5 piezas básicas que van cambiando de color de forma aleatoria. Aquí están todas las piezas del Tetris básico:



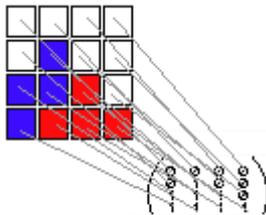
Todas giran hacia la izquierda cada vez que pulsamos la tecla de giro, es decir, no hay piezas que giren hacia la izquierda y piezas que lo hagan a la derecha. La pregunta que nos deberemos de hacer entonces es cómo poder representar de forma eficiente las piezas y los giros, teniendo siempre presente las colisiones que se puedan producir (contra otras piezas o contra los límites del área del juego, esto es, la tabla en donde vamos alojando las piezas).

De la observación del juego podemos sacar varias conclusiones. Para empezar que vamos a necesitar almacenar en algún sitio las piezas que vayamos colocando en la tabla del juego. Por otro lado, deberemos de implementar un método para almacenar las figuras y conocer su forma. También habrá que hablar de las colisiones que se puedan producir en nuestro juego. Estas colisiones podrán deberse a que hemos chocado contra algún límite del área de juego o que hemos colisionado con alguna otra pieza. Una vez que sepamos cómo detectar una colisión (contra límite y contra pieza), será el momento de pensar cómo "adivinar" que hemos hecho una línea. Por último, habrá que implementar un temporizador o similar que nos

ayude a actualizar de forma más o menos rápida (según la dificultad) el estado de las piezas que vayan cayendo. Todo esto será lo que veremos a continuación.

### **Representación del área de juego:**

La parte más importante de todas es la referida a la representación del área del juego (supongo que todo el mundo que está leyendo este artículo ha jugado alguna vez al Tetris). Como todos recordaréis, en el Tetris vamos alojando las piezas en una tabla según éstas van cayendo de la parte superior de la pantalla a la base de la misma, con una determinada velocidad. La tabla será rectangular con mayor altura que anchura. Dicho esto, la forma más fácil de implementar la tabla sería con una tabla de m filas y n columnas.



RELACION  
AREA DE JUEGO - MATRIZ

Así pues, un área como el que sale en la figura perfectamente puede implementarse con una matriz (tabla bidimensional) de m filas y n columnas. Ahora la siguiente pregunta es ¿qué guardo en cada posición de la matriz?. Para responder a esta pregunta, convendría estudiar también la forma en la que vamos a implementar las piezas, pues el área del juego, esto es, la matriz, no hace otra cosa que guardar las piezas que el jugador ha ido utilizando, y utiliza, durante la partida. La aproximación más básica nos dice que hay zonas del área de juego que están ocupadas y otras que no lo están. Para poder representar esto, bastaría con tener un valor 0 en las posiciones de la matriz libres y un 1 en las ocupadas. Sin embargo, como vamos a utilizar piezas de distintos colores (o, incluso, de distintas propiedades), convendría generalizar el valor 1 a valores distintos de 0, de tal forma que valores como el 2, el 3, el 4 o 14, también indicaran que esa zona está ocupada. Así, si a cada código distinto de 0 le asignamos un color, podríamos identificar cada zona de la matriz ocupada por una porción de pieza de un determinado color. Por ejemplo, veamos la figura siguiente:

0	0	0	0
0	2	0	0
2	2	1	0
2	1	1	1

ASIGNACION DE DISTINTOS CODIGOS  
PARA DISTINTOS COLORES DE PIEZA

Como podemos ver, un código 0 en el área de juego, esto es, en la matriz o array, representa casilla libre (que no hay pieza o, en otras palabras, que puede ocuparse) mientras que todos aquellos valores distintos de 0 representan una porción de pieza de un determinado color. Esta forma de representar el área de juego, ya nos puede permitir pensar en la forma de almacenamiento que llevarán nuestras piezas o figuras.

## Representación de las figuras:

Hasta ahora hemos quedado en que una matriz (implementada a gusto de cada cual) va a ser la estructura de datos que represente, internamente, el área de juego. También hemos fijado que la matriz va a contener distintos códigos que se pueden agrupar en dos conjuntos:

1. Los que representan casilla vacía, esto es, el conjunto formado por el código 0.
2. Los que representan presencia de pieza, esto es, los formados por códigos distintos de 0.

El siguiente paso es pensar en la forma de almacenar y representar las figuras. Conviene, sin embargo, tener presente qué tipo de figuras existen en un Tetris básico así como los giros o diferentes estados que pueden tener. En la figura [A] vemos un área de juego de un Tetris con rejilla (para constatar un par de puntos que nos van a ayudar a razonar el modo de representación de piezas).

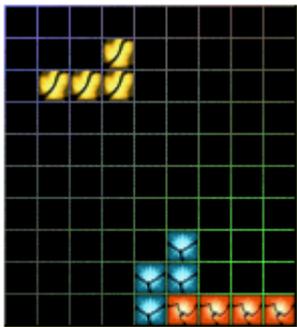


Figura [A].  
PORCIÓN DE UN ÁREA DE  
JUEGO DE UN TETRIS CON  
REJILLA

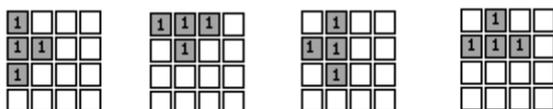
Observando la figura [A], uno se da cuenta de los siguientes detalles:

1. Todas las piezas están formadas por cuadrados.
2. Todas las piezas tienen el mismo número de cuadrados.

El que todas las piezas estén formadas por cuadrados, nos permite ajustar de forma muy elegante los sprites (o sea, imágenes en movimiento) que vamos a utilizar. Si todas las piezas están formadas por cuadrados, bastará con tener un cuadrado, es decir, una porción de pieza, por cada color posible de pieza. En otras palabras, si en nuestro Tetris las piezas pueden tener color rojo, azul, verde y amarillo, bastará con tener un cuadrado de color rojo, azul, verde y amarillo ya que la unión de varios de ellos (de cuatro), nos formará una pieza del juego. Así de simple y claro. ¿Esto que nos evita?, pues nos evita tener que almacenar un gráfico por cada pieza en cada una de sus posiciones posibles o estados. Pensemos que cada pieza tiene 4 estados de giro, si tenemos un total de 5 piezas distintas tendríamos que dibujar  $4 \cdot 5 = 20$  estados diferentes (20 sprites –imágenes- diferentes). Mediante este método construimos las piezas durante el transcurso de juego, ahorrando espacio en disco y memoria para gráficos (además los cálculos serán mínimos). Y, sobre todo, dando una mayor flexibilidad al programa de cara a modificaciones o inserción de nuevas piezas.

Sin embargo, y tal como podíamos ver cuando se mostraron todas las figuras que intervienen en un Tetris básico (primera figura de este apartado), es necesario implementar una serie de posiciones diferentes para cada pieza. Aún más, es necesario, para este sistema de almacenamiento, implementar algún

modo de saber cómo se han de disponer los cuadrados para formar las piezas. En otras palabras, ¿cómo sabemos cuál es la pieza que hay que poner en pantalla cuando se realiza un giro?. Para resolver este problema, deberemos de almacenar tantas plantillas como posiciones de giro haya por pieza. Así, si cualquier pieza tiene 4 posibles posiciones de giro, deberemos de almacenar 4 plantillas. ¿Cómo podemos construir esas plantillas?. Bien, bastará con disponer de una tabla bidimensional (una matriz) de p filas y q columnas, donde p = anchura de la pieza y q = altura de la pieza, que contenga un "1" en donde haya que poner cuadrado de la figura y un "0" donde no haga falta. Obviamente, habrá que disponer de una matriz de estas dimensiones por cada plantilla de cada una de las piezas. Cada vez que la pieza pega un giro, pasamos a la siguiente plantilla que toque y construimos la pieza en base a la plantilla que nos encontremos.



### **Evolución del juego:**

Hasta ahora, ya hemos analizado una posible forma de almacenar y mantener las piezas. Está claro que ahora llega el turno de meternos más a fondo con los que es el "runtime" del juego, esto es, el bucle de ejecución.

Durante el transcurso de la partida al Tetris, disponemos de un tiempo límite para realizar todas las acciones pertinentes sobre la pieza que estamos manipulando. Dependiendo del nivel en el que nos encontremos, el tiempo será mayor o menor y, por lo tanto, variará la dificultad del mismo. Al ir pasando los segundos, la pieza irá cayendo una posición tras otra hacia la parte final de la tabla. Esto continuará hasta que la pieza toque el fondo del área de juego o bien, hasta que colisione con otra y le sea imposible seguir bajando.

Podemos imaginarnos que en una partida al Tetris, el tiempo se encuentra "ranurado". Es decir, hay celdas de tiempo y, en cada una de ellas, se realizan una serie de acciones. Pasado el tiempo que dura una celda, se baja una posición la pieza y volvemos a realizar las acciones que se pueden llevar dentro de esa celda de tiempo. Cada pieza dispondrá de su propio espacio ranurado de tiempo hasta que le es imposible descender más por el área de juego. En ese momento, la pieza actual deja de existir y aparece otra con su nuevo espacio ranurado de tiempo.

Está claro, que hay que implementar algún tipo de temporizador para el juego. Un temporizador permite al programador establecer señales de alarma que, pasado un intervalo de tiempo (milisegundos o segundos), se activan ejecutando la función que tengan encomendada. En nuestro caso, no sería otra que disminuir en una posición la componente "y" de la pieza y redibujarla en pantalla. Una vez hecho esto, se volverán a ejecutar las operaciones comunes a una celda de tiempo hasta que, de nuevo, vuelva a vencer el temporizador. Dependiendo de la dificultad del juego, podremos bajar la pieza cada 1 segundo, 2, 3... El uso de los temporizadores es muy interesante de cara al desarrollo de videojuegos, y bien merecerían un apartado a parte.

## Colisiones:

Uno de los aspectos más importantes del análisis, es el referido a las colisiones que puedan sufrir nuestras piezas. Debemos de tener en cuenta que las colisiones deben de separarse en dos grupos:

- Las que se producen cuando chocamos con los límites del área del juego.
- Las que se producen cuando chocamos con fragmentos de otras piezas del juego.

Vamos a proceder a su análisis a continuación.

### Colisiones entre los límites del área de juego:

Las piezas pueden colisionar contra los límites del área de juego, siempre que intentemos moverlas más allá de los límites izquierdo, derecho y base de nuestra "matriz" de juego. Es decir, si el jugador comienza a pulsar de forma continuada a la tecla derecha, llegará un momento en el que la pieza chocará contra la parte lateral derecha. Esto mismo sucederá en la parte lateral izquierda si el jugador comienza a pulsar continuamente la tecla de movimiento a la izquierda. ¿Y en la base? Pues esta colisión ocurrirá cuando el usuario haga que su pieza colisione con el fondo de la matriz. Esto producirá que el tiempo de vida de la pieza que estamos manejando termine y que salga otra por la parte superior. He aquí un dibujo explicativo de cada tipo de colisión.



Pero no sólo debemos de pensar que una colisión se produce cuando se intenta mover en sentido horizontal (izquierda - derecha) o vertical (hacia abajo); también puede darse el caso de que la colisión se produzca cuando vayamos a realizar un giro de una pieza. Sí, puede ocurrir que estemos en un lateral de la pantalla y que por la naturaleza de la pieza que estamos utilizando (por ejemplo, suponed que estamos usando una barra en horizontal), el realizar un giro supone que nuestra pieza va a ocupar hacia el lateral por el que no podemos movernos más, una colisión. En este caso, deberemos de detectar también la colisión y no permitir que nuestra pieza realice el giro. En la figura siguiente, se ve cómo la barra en horizontal no puede realizar un giro estando en el tope del lateral derecho. Más aún, puesto que tiene una longitud de 4 casillas, no podrá realizar un giro estando en posición vertical, a menos que esté a 3 casillas de distancia del lateral. En caso contrario, la barra en horizontal se saldría del área de juego.



Según estos datos, deberemos de contener en todo momento las posiciones (x,y) de inicio de nuestras piezas para que, siempre que el usuario vaya a realizar un movimiento, calculemos por adelantado las nuevas posiciones de nuestras piezas, entrando en juego las alturas y anchuras propias (que variarán dependiendo del frame en el que nos encontremos. Por ejemplo, la barra en horizontal tendrá una anchura de 4 casillas y una altura de 1 casilla y, la barra en vertical tendrá una altura de 4 casillas y una anchura de 1 casilla. Todo esto debería de figurar ligado a cada una de las plantillas de la figura).

Una sencilla forma de ver el proceso de cálculo de colisión podría ser:

Sea (x, y) la posición de nuestra figura. Sea un movimiento a la derecha un incremento de la x y un movimiento a la izquierda un decremento. Sea un movimiento hacia abajo un incremento de la y (no hay movimientos hacia arriba). Sea anchura el ancho de nuestra pieza menos 1 (La barra horizontal tendría ancho 0). Sea altura la altura de nuestra pieza menos 1 (La barra en horizontal tendría alto 3). Sea limiteDerecho la posición x más a la derecha del área de juego (la posición x mayor). Sea limiteIzquierdo la posición x más a la izquierda del área de juego (suponemos que 0). Sea limiteBase la posición y más al fondo del área de juego (la posición y mayor)

**Colisión con límite derecho:** Si el usuario pulsa la tecla de movimiento hacia la derecha, deberemos de hacer la comprobación:

```
if ( (x+1)+anchura <= limiteDerecho ) { permitimos movimiento }
else { no permitimos movimiento }
```

**Colisión con límite izquierdo:** Si el usuario pulsa la tecla de movimiento hacia la izquierda, deberemos de hacer la comprobación:

```
if ( (x-1) >= limiteIzquierdo ) { permitimos movimiento }
else { no permitimos movimiento }
```

**Colisión con límite de base:** Si el usuario pulsa la tecla de movimiento hacia la abajo o la pieza toca la base al llegar por si sólo.

```
if ( (y+1)+ alto <= limiteDerecho ) { permitimos movimiento }
else { no permitimos movimiento }
```

Como se puede observar, el proceso es bastante simple en lo básico. ¿Cómo haríamos para detectar una colisión al realizar un giro?. Bien sencillo. No habrá que escribir un código especial, bastará con obtener la anchura y la altura del "frame" o plantilla que viene a continuación de la pieza actual, y aplicarle exactamente las mismas comprobaciones que antes. Lo único que hacemos, es sustituir "virtualmente" la pieza actual por la siguiente y comprobar si un movimiento de ésta última produciría algún tipo de colisión.

## Colisiones entre piezas:

Las colisiones entre piezas añaden algo más de dificultad a la hora de hacer giros, ya que la pieza actual puede chocar con los fragmentos que hayan en su camino de giro y por ello no poder realizar el cambio de posición. Por lo demás, también deberemos de realizar una comprobación de colisión cuando nuestra pieza se mueva hacia la izquierda, derecha o hacia abajo. Debemos de tener en cuenta que se nos puede presentar una situación como la siguiente:



En este ejemplo, vemos cómo una pieza que intenta moverse hacia la derecha no puede por el sencillo motivo de que hay fragmentos de otras piezas que le impiden el movimiento. Poder controlar esto es muy fácil, pues tan sólo deberemos de comprobar, internamente, si al mover una posición nuestra figura (a la izquierda, derecha o hacia abajo), ocupa alguna zona que ya estaba previamente ocupada. En este caso, no podremos permitir el movimiento. Si este tipo de colisión se da, por ejemplo, cuando el temporizador asignado a la figura que estamos manejando vence y hace que ésta tenga que bajar una posición, significará que no podrá bajar y que, por lo tanto, se ha acabado su tiempo de vida en el juego y tiene que salir otra por la parte superior.

El otro problema que podía darse al manejar una pieza, era el referido al de girarla. Si nuestra figura va a ser girada en un determinado sentido y encuentra fragmentos de otras figuras, ésta no podrá girar. ¿Cómo podemos solucionar este problema?. Bien, lo primero que deberemos de hacer es tomar las propiedades del frame siguiente al actual, más concretamente la anchura y la altura. Una vez hecho esto, deberemos de discutir dichos valores:

**Si la anchura y la altura son iguales.** En este caso, nos encontramos con que la figura realmente ocupa la misma zona al realizar un giro. Esto ocurrirá si nuestra figura es un cuadrado y, por tanto, no haremos nada.

**Si la altura es mayor que la anchura.** Lo que deberemos de hacer es recorrer la porción de la tabla del área de juego que va desde la posición  $(x,y)$  hasta la posición  $(x+altura-1,y+altura-1)$ . Esto es así, porque esa es la porción que se supone "barrería" nuestra pieza al girar. Así pues, bastará saber si en esa área hay alguna posición del área de juego distinta a 0 (es decir, que contiene algún fragmento de pieza) ya que, en ese caso, no podríamos realizar el giro.

**Si la altura es menor que la anchura.** Es idéntico que en el caso anterior, salvo que en lugar de sumar altura, sumaríamos anchura. Recorreríamos la porción de área de juego que iría de  $(x,y)$  a  $(x+anchura-1,y+anchura-1)$ .

Una vez estudiadas las colisiones, convendría hacerse la pregunta ¿y cuál de las dos compruebo antes?. Es obvio que, en cada movimiento de pieza, hay que realizar una comparación de colisión en el escenario. Lo ideal sería tratar ambas colisiones por separado. Si al encontrarnos una colisión en los límites

del área de juego, ya no realizaríamos comprobación de colisión con los fragmentos de piezas y viceversa.

### Cómo averiguar si hemos hecho línea:

Siempre que la vida de una pieza ha terminado, esto es, en el mismo instante que vence la ranura de tiempo y no puede bajar una posición porque hay colisión (contra la base del área de juego o contra otros fragmentos de piezas), hay que comprobar si ha hecho línea. Una pieza hará línea cuando alguno de sus fragmentos logren hacer que la fila en la que se han colocado todas las casillas estén ocupadas. Para verlo más claro, esta figura:



Aquí podemos ver cómo la barra que acabamos de colocar hace que dos de sus casillas hagan línea (la casilla 1 y 3, comenzando a contar de arriba a abajo). ¿Cómo detectar esto?. Bien, antes de explicar cómo detectar línea, convendría que pensáramos mejor lo que pasa cuando una pieza hace línea. Puntos a tener en cuenta:

- Las únicas filas que podrán hacer línea serán aquellas en las que la nueva pieza se haya alojado.
- Cuando hacemos línea, debemos de tener en cuenta que baja la altura de los montones de fragmentos de piezas situados en el área de juego. Si nuestra pieza hace n líneas, habrá que bajar todos los fragmentos que se encuentran desde la fila más baja (más cercana a la base del área de juego) en donde nuestra pieza ha hecho línea, n posiciones.
- Obviamente, deberemos de efectuar actualizaciones gráficas, de puntuación, de pase de nivel, etc.

Una vez definidos estos tres puntos básicos, podríamos establecer los siguientes pasos:

**Comprobar si hemos hecho línea:** Para ello deberemos de examinar todas las filas del área de juego en las que nuestra pieza se ha establecido, recorriéndolas. Si en algún momento encontramos que hay un valor igual a 0, significará que ha quedado un hueco sin ocupar y, por lo tanto, que no se ha hecho línea. Aquí yo recomendaría que se hiciera un recorrido derecha - izquierda o izquierda - derecha, es decir, que se evitara recorrer una fila de forma secuencial (todo el rato a la derecha o todo el rato a la izquierda). En lugar de ello, una alternancia extremo derecho - extremo izquierdo aumentaría en gran medida la efectividad de los cálculos pues, de no existir línea, habría más posibilidad de saberlo antes. Se guardará un contador que indique el número máximo de filas que han hecho línea y un buffer (es decir, una variable temporal) que nos indique las filas que han hecho línea.

**Si hay alguna línea, procedemos:** En el caso de que el contador de líneas hechas sea distinto de 0, significará que se ha hecho línea y que debemos de bajar la altura de los

"montones de fragmentos". Básicamente, el proceso será una recolocación de las filas del área de juego. El montón bajará una altura igual al número de filas que han hecho línea. En líneas generales, el mecanismo consistirá en evaluar todas las filas desde la última que ha hecho línea hacia arriba, marcando, en un principio, todas las filas como "desocupadas". Una vez hecho esto, se evaluarán las filas que no hacen línea (porque son las que tienen las piezas que van a ser recolocadas) y se irán situando en las filas marcadas como vacías, hasta que no quede ninguna.

**Actualizamos estructuras internas:** Como es lógico, una vez que se hace línea habrá que evaluar unas cuantas cosas como son los puntos que recibe el jugador, si ha pasado de fase o si tiene que salir otra pieza, etc.

### En resumen:

Lo expuesto aquí no es más que un análisis bastante acelerado de todo lo que sería el mecanismo interno de cara a la realización de un Tetris básico. Algunos temas no se han enfocado más a fondo por el simple hecho de que no se pretendía hacer de este apartado un listado en donde copiar la solución para hacer un Tetris. Creo que la mayor parte de las bases de este juego están expuestas. Ahora bien, todo aquel que quiera hacer un Tetris, que se olvide de agarrarse a este documento únicamente pues un juego tan trivial como el Tetris, esconde unos cuantos secretos más que sólo se desvelan cuando se toma en serio el proyecto. Así que, si quieres hacer un Tetris léete este documento una vez, obtén una idea general y luego ponte a escribir tu propia documentación.

A continuación tienes un ejemplo bien comentado de código escrito en Fénix donde se implementa un juego de tetris básico, para que puedas probarlo y estudiarlo a fondo. Para poder ejecutarlo necesitarás un fpg llamado "tetris.fpg" que constará de 8 gráficos, todos ellos cuadrados de diferentes colores de un tamaño de 20x20 píxeles cuyos códigos irán del 001 al 007, y el 010..

Estos gráficos son las casillas con las que se podrán formar (agrupándolas de manera adecuada) los distintos bloques (piezas) del tetris. En concreto, hay definidos tres niveles de dificultad, y los gráficos con códigos del 1 al 4 serán las casillas que formarán los bloques que aparecerán cuando la dificultad del juego esté definida como fácil. Los gráficos 5 y 6 servirán para generar bloques que sólo aparecerán cuando la dificultad sea media (además de los que ya aparecían en la dificultad fácil), y el gráfico 7 será una casilla que formará un bloque que sólo aparecerá cuando la dificultad sea máxima (además del resto de bloques anteriores). El gráfico 10 se utilizará para rellenar de color las casillas vacías del terreno de juego, así como también borrar éste cuando el jugador haya perdido una partida, para volver a empezar.

Además, el gráfico 1 también se utilizará para reseguir el contorno del campo de juego, y de la zona donde se ve un preview del bloque que viene a continuación del actual.

En el ejemplo no se incorporan efectos de sonido o de banda sonora, pero está claro que añadir esta clase de recursos mejoraría sustancialmente la experiencia del jugador: una melodía mientras la introducción está visible, y otra mientras dura el juego; o bien una serie de sonidos característicos cuando se hace línea completa o se sufre un game over pueden ser incluidas fácilmente dentro del código (con las funciones correspondientes `load_wav`, `load_song` y `play_wav`, `play_song` entre otras), mejorando el resultado final muchísimo.

El juego también tiene la funcionalidad de guardar en un archivo de texto llamado "scores.dat" la máxima puntuación conseguida hasta el momento para cada una de las tres dificultades, con lo que si en una partida se logra una puntuación superior, se notificará por pantalla y se guardará sobrescribiendo la anterior correspondiente a la misma dificultad.

Por otro lado, quisiera comentar por encima la jerarquías de procesos utilizada en este código. El

programa principal básicamente lo que hace es llamar al proceso "menú" para seguidamente morir. El proceso "menú", a su vez, invoca a dos procesos: "render" y "game", antes de morir él también. El proceso "render" está dedicado básicamente al dibujo (al "renderizado") de los bloques de tetris por pantalla en sus diversas posiciones y orientaciones -y desde él se puede volver a resucitar el proceso "menú", también-, mientras que el proceso "game" se encarga de la operativa del juego (responder a las teclas del jugador, comprobar que se haga una línea completa horizontal, comprobar que no se llene el campo de juego hasta arriba, etc). Para ello, este proceso "game" se ayuda de un proceso hijo auxiliar llamado "detect", que se encargará, mientras su padre esté vivo, de detectar si se ha logrado hacer una línea horizontal completa o no, para reaccionar en consecuencia (eliminar esa línea del campo de juego, aumentar la puntuación del jugador, etc). Y también se ayuda de una función llamada "kannaar" que detecta simplemente si un bloque puede caber o no en una determinada posición del campo de juego. El proceso "game" tiene un tercer hijo, que es el proceso "gameover", el cual, como su nombre indica, aparece cuando el juego se ha terminado (es decir, justo antes de morir "game" y "render"). Su tarea principal es resucitar a ambos procesos, "game" y "render" otra vez, para volver a empezar. Es decir, si hiciéramos un esquema (la "X" indica que el proceso muere), sería así:

```
Principal --> Menu --X-->Render
      |
      |---X--->Game --> Detect
          |
          |--X -->GameOver
          |--->Kannaar (función)
```

//By Moogle (Roelf Leenders)

Program tetris;

const

*/\*Tamaño del campo de juego, definido en casillas lógicas. Hay 10 casillas en horizontal y 20 en vertical, y cada una de ellas podrá ser ocupada por uno de los gráficos 20x20 que conformarán alguno de las piezas\*/*

ANCHO = 10;

ALTO = 20;

end

global

*int difficulty=0; /\*Dificultad actual (existen 3:baja,media,alta). Sirve para definir el tipo de piezas que pueden aparecer, y para definir el FPS del juego\*/*

*int graad[2]; //Almacena para cada dificultad el número de gráfico máximo de bloque que puede aparecer*

*int BLOKSPEED = 15; //Velocidad de bajada del bloque. Se modificará dependiendo del nivel del jugador.*

*int level=1; //Nivel del jugador (a mayor nivel, más rápido es el descenso)*

*int ScoreVolleRij = 100; //Puntuación lograda por completar una línea horizontal completa*

*int ScoreZomaar = 2; //Puntuación básica lograda por hacer cualquier movimiento*

*int score=0; //Puntuación total actual del jugador*

*int highrecords[2]; //Máximas puntuaciones guardadas para las 3 dificultades del juego*

*int blok[10][5][5]; //Define la forma sensible de hasta 10 posibles piezas, de un tamaño máximo de 5x5*

*int nu=1; /\*Tipo del bloque actual (en el juego irá valiendo alguna posición de la 1ª dimensión de la matriz blok, entre 0 y 9) \*/*

*int next=0; /\*Tipo del bloque siguiente (en el juego irá valiendo alguna posición de la primera dimensión de la matriz blok, entre 0 y 9) \*/*

*int nux=ANCHO/2; //Posición x del bloque actual*

*int muy=-1; //Posición y del bloque actual*

*int kant=0; //Modo en el que está rotado el bloque actual*

```

int steen[ALTO][ANCHO]; /*Define qué casilla lógica (no gráfica) del campo de juego está ocupada o
no.*/
int fpg; //Cargar el fpg
int n; //Para los piezas

//Defino las teclas:
int _A = _control;
int _B = _alt;
int _SELECT = _space;
int _R = _tab;
int _L = _backspace;
int _START = _enter;
end

begin
set_text_color(rgb(0,0,255));
set_mode(800,600,16);
fpg = load_fpg("tetris.fpg");
fill_map();//Proceso que pinta el fondo con un gradiente

//Cargo los récords desde un archivo. Si éste no existe, lo crea antes
if(!file_exists("scores.dat")); save("scores.dat",highrecords); end
load("scores.dat",highrecords);

//Creo los piezas lógicas:
n = 0;

// OO
// OO

//      H V
blok[n][0][0] = 0;
blok[n][0][1] = 0;
blok[n][0][2] = 0;
blok[n][0][3] = 0;
blok[n][0][4] = 0;

blok[n][1][0] = 0;
blok[n][1][1] = 0;
blok[n][1][2] = 1;
blok[n][1][3] = 1;
blok[n][1][4] = 0;

blok[n][2][0] = 0;
blok[n][2][1] = 0;
blok[n][2][2] = 1;
blok[n][2][3] = 1;
blok[n][2][4] = 0;

blok[n][3][0] = 0;
blok[n][3][1] = 0;
blok[n][3][2] = 0;
blok[n][3][3] = 0;
blok[n][3][4] = 0;

```

```

blok[n][4][0] = 0;
blok[n][4][1] = 0;
blok[n][4][2] = 0;
blok[n][4][3] = 0;
blok[n][4][4] = 0;

n++;

// O
// O
// OO

//      H V
blok[n][0][0] = 0;
blok[n][0][1] = 0;
blok[n][0][2] = 0;
blok[n][0][3] = 0;
blok[n][0][4] = 0;

blok[n][1][0] = 0;
blok[n][1][1] = 0;
blok[n][1][2] = 0;
blok[n][1][3] = 0;
blok[n][1][4] = 0;

blok[n][2][0] = 0;
blok[n][2][1] = 1;
blok[n][2][2] = 1;
blok[n][2][3] = 1;
blok[n][2][4] = 0;

blok[n][3][0] = 0;
blok[n][3][1] = 0;
blok[n][3][2] = 0;
blok[n][3][3] = 1;
blok[n][3][4] = 0;

blok[n][4][0] = 0;
blok[n][4][1] = 0;
blok[n][4][2] = 0;
blok[n][4][3] = 0;
blok[n][4][4] = 0;

n++;

// O
// OOO

//      H V
blok[n][0][0] = 0;
blok[n][0][1] = 0;
blok[n][0][2] = 0;
blok[n][0][3] = 0;

```

```

blok[n][0][4] = 0;

blok[n][1][0] = 0;
blok[n][1][1] = 0;
blok[n][1][2] = 1;
blok[n][1][3] = 0;
blok[n][1][4] = 0;

blok[n][2][0] = 0;
blok[n][2][1] = 1;
blok[n][2][2] = 1;
blok[n][2][3] = 1;
blok[n][2][4] = 0;

blok[n][3][0] = 0;
blok[n][3][1] = 0;
blok[n][3][2] = 0;
blok[n][3][3] = 0;
blok[n][3][4] = 0;

blok[n][4][0] = 0;
blok[n][4][1] = 0;
blok[n][4][2] = 0;
blok[n][4][3] = 0;
blok[n][4][4] = 0;

n++;

// OOOO

//      H V
blok[n][0][0] = 0;
blok[n][0][1] = 0;
blok[n][0][2] = 1;
blok[n][0][3] = 0;
blok[n][0][4] = 0;

blok[n][1][0] = 0;
blok[n][1][1] = 0;
blok[n][1][2] = 1;
blok[n][1][3] = 0;
blok[n][1][4] = 0;

blok[n][2][0] = 0;
blok[n][2][1] = 0;
blok[n][2][2] = 1;
blok[n][2][3] = 0;
blok[n][2][4] = 0;

blok[n][3][0] = 0;
blok[n][3][1] = 0;
blok[n][3][2] = 1;
blok[n][3][3] = 0;
blok[n][3][4] = 0;

```

```

blok[n][4][0] = 0;
blok[n][4][1] = 0;
blok[n][4][2] = 0;
blok[n][4][3] = 0;
blok[n][4][4] = 0;

n++;
// OO
// OO

//      H V
blok[n][0][0] = 0;
blok[n][0][1] = 0;
blok[n][0][2] = 0;
blok[n][0][3] = 0;
blok[n][0][4] = 0;

blok[n][1][0] = 0;
blok[n][1][1] = 0;
blok[n][1][2] = 1;
blok[n][1][3] = 0;
blok[n][1][4] = 0;

blok[n][2][0] = 0;
blok[n][2][1] = 1;
blok[n][2][2] = 1;
blok[n][2][3] = 0;
blok[n][2][4] = 0;

blok[n][3][0] = 0;
blok[n][3][1] = 1;
blok[n][3][2] = 0;
blok[n][3][3] = 0;
blok[n][3][4] = 0;

blok[n][4][0] = 0;
blok[n][4][1] = 0;
blok[n][4][2] = 0;
blok[n][4][3] = 0;
blok[n][4][4] = 0;

//Fin de la dificultad fácil. Los siguientes piezas son para la dificultad media
graad[0] = n;

n++;

// OO
// OOO

//      H V
blok[n][0][0] = 0;
blok[n][0][1] = 0;
blok[n][0][2] = 0;

```

```

blok[n][0][3] = 0;
blok[n][0][4] = 0;

blok[n][1][0] = 0;
blok[n][1][1] = 1;
blok[n][1][2] = 0;
blok[n][1][3] = 1;
blok[n][1][4] = 0;

blok[n][2][0] = 0;
blok[n][2][1] = 1;
blok[n][2][2] = 1;
blok[n][2][3] = 1;
blok[n][2][4] = 0;

blok[n][3][0] = 0;
blok[n][3][1] = 0;
blok[n][3][2] = 0;
blok[n][3][3] = 0;
blok[n][3][4] = 0;

blok[n][4][0] = 0;
blok[n][4][1] = 0;
blok[n][4][2] = 0;
blok[n][4][3] = 0;
blok[n][4][4] = 0;

n++;

//  O
//  OOO
//  O

//      H V
blok[n][0][0] = 0;
blok[n][0][1] = 0;
blok[n][0][2] = 0;
blok[n][0][3] = 0;
blok[n][0][4] = 0;

blok[n][1][0] = 0;
blok[n][1][1] = 0;
blok[n][1][2] = 1;
blok[n][1][3] = 0;
blok[n][1][4] = 0;

blok[n][2][0] = 0;
blok[n][2][1] = 1;
blok[n][2][2] = 1;
blok[n][2][3] = 1;
blok[n][2][4] = 0;

blok[n][3][0] = 0;
blok[n][3][1] = 0;

```

```

blok[n][3][2] = 1;
blok[n][3][3] = 0;
blok[n][3][4] = 0;

blok[n][4][0] = 0;
blok[n][4][1] = 0;
blok[n][4][2] = 0;
blok[n][4][3] = 0;
blok[n][4][4] = 0;

//Fin de la dificultad media. Los piezas siguientes son para la dificultad máxima.
graad[1] = n;

n++;

// O
// O
// O

//      H V
blok[n][0][0] = 0;
blok[n][0][1] = 0;
blok[n][0][2] = 0;
blok[n][0][3] = 0;
blok[n][0][4] = 0;

blok[n][1][0] = 0;
blok[n][1][1] = 1;
blok[n][1][2] = 0;
blok[n][1][3] = 0;
blok[n][1][4] = 0;

blok[n][2][0] = 0;
blok[n][2][1] = 0;
blok[n][2][2] = 1;
blok[n][2][3] = 0;
blok[n][2][4] = 0;

blok[n][3][0] = 0;
blok[n][3][1] = 0;
blok[n][3][2] = 0;
blok[n][3][3] = 1;
blok[n][3][4] = 0;

blok[n][4][0] = 0;
blok[n][4][1] = 0;
blok[n][4][2] = 0;
blok[n][4][3] = 0;
blok[n][4][4] = 0;

//Fin de la dificultad máxima
graad[2] = n;

//Inicializo...

```

```

score = 0;
level = 0;
//...y pongo en marcha el menú inicial
Menu();
end

//-----

/*Menú básico: se ofrece la posibilidad de elegir dificultad (baja,media,alta) y de ver los correspondientes
récords */
Process Menu()
private
int i;
String graden[2] = "Tirada","Normal","Muerte!!!";
int text,tex2;
end
begin
//Se vacía el campo de juego
for(x=0;x<ALTO;x++)
for(y=0;y<ANCHO;y++)
steen[x][y] = 0;
end
end
fill_map();
difficulty = 1; //Se establece la dificultad a fácil por defecto
graph = write_in_map("TETRIS",4); //Creo un gráfico con la palabra TETRIS escrita en él
x = 400;y = 200;
//Escribo el resto de texto y variables
write(0,400,350,1,"Pulsa ENTER para Empezar");
write(0,400,370,1,"SPACE para Salir");
text = write(0,400,420,1,"Dificultad: " +graden[difficulty-1]);
write(0,370,440,1,"Record:");
tex2 = write_var(0,420,440,1,highrecords[difficulty-1]);
write(0,700,580,4,"By O.T.A ");
//Ya que TETRIS es el gráfico de este proceso, se colocará como título por encima de las opciones.

loop
//Efecto para simular que el título está rotando. Muy curioso!!
i=i+7000;
size_x = 200;
size_y = abs(get_distx(i,200));
//Se refleja verticalmente la imagen para que la rotación parezca más realista
if(get_distx(i,200)<0) flags = 2; else flags = 0; end
/*Se cambia de dificultad (teclas izquierda y derecha), y acorde a ésta se muestran diferentes valores en
el menú*/
if(key(_left) and difficulty>1)
difficulty--;
delete_text(text);
delete_text(tex2);
text = write(0,400,420,1,"Dificultad: " +graden[difficulty-1]);
tex2 = write_var(0,420,440,1,highrecords[difficulty-1]);
while(key(_left))frame;end //Hasta que no se suelte la tecla no se hará nada más
end
if(key(_right) and difficulty<3)

```

```

difficulty++;
delete_text(text);
delete_text(tex2);
text = write(0,400,420,1,"Dificultad: " + graden[difficulty-1]);
tex2 = write_var(0,420,440,1,highrecords[difficulty-1]);
while(key(_right))frame;end
end
if(key(__START))break; end //Salir del loop si el juego comienza
if(key(__SELECT) or key(_esc))exit(); end //Acabar la ejecución si se ha seleccionado la tecla SELECT
frame;
end
/*Limito los frames por segundo en relación a la dificultad escogida, para hacer el juego más rápido a
dificultades altas y más lento a dificultades bajas*/
set_fps(20+difficulty*7,0);
delete_text(all_text);
Render(); //Proceso que dibuja ("renderiza") el campo de juego en pantalla. No funciona bien.
//Escribo los textos del juego
write_var(0,560,400,4,score);
write(0,560,390,4,"Puntuación:");
write(0,560,430,4,"Nivel:");
write_var(0,560,440,4,level);
//Y comienza la lógica del juego (por fin!)
game();
end

//-----

/*Redibuja cada fotograma en la pantalla el nivel del jugador, el bloque actual y la previsualización del
siguiente bloque. No funciona bien porque no borra los piezas cuando éstos se han movido de posición,
con lo que aparece un rastro de piezas que hace que no se pueda ver lo que se está haciendo. Queda
pendiente para arreglar.*/
process Render()
private
  int i,j,k,l,block;
  int a;
end
begin
loop
//Si se pulsa SELECT mientras se está jugando, se para todo y se sale al menú inicial
if(key(__SELECT))
  while(key(__SELECT))frame;end //Hasta que no se suelte la tecla no se hace nada
  clear_screen();
  delete_text(all_text);
  score = 0;
  difficulty = 0;
  let_me_alone(); //Deja sólo el proceso actual
  menu(); //Pone en marcha el menú otra vez
  return; /*Ahora hay dos procesos, el actual (que ya no interesa), y el menú. Una manera de acabar con
el proceso actual -un suicidio- es con esta línea.*/
end

//Dibujo el campo de juego
for(i = 0;i < ALTO+2;i++)

```

```

for(j = 0;j < ANCHO+2;j++)
//De entrada, la casilla lógica está vacía
    block = 0;
//Si la casilla pertenece al reborde, la pinto con el gráfico 001
    if(i == 0 or i == ALTO+1 or j == 0 or j == ANCHO+1)
        block = 1;
/*Al principio del juego, todos los elementos de steen[][] valen 0 (es decir, no hay ninguna casilla lógica
ocupada).A medida que se vaya jugando, block irá valiendo en tiempo real un número entre 0 y 7 que
indicará el gráfico del FPG que se ha de pintar en la casilla lógica*/
    else
        block = steen[i-1][j-1];
    end
//Las casillas que permanecen vacías(al principio, todas menos las del borde)se pintan con el gráfico 010
    if(block == 0)block = 10; end
/*Ahora sí: pinto el terreno de juego.Esto lo haré a cada frame mientras el proceso Render esté
funcionando. En cada frame se redibujarán los bordes, las casillas vacías y las ocupadas, que vienen
controladas por steen y que se reflejan visiblemente en block*/
/*El 200 y el 100 son las coordenadas x,y de la esquina superior izquierda del campo, y el 20 es el tamaño
de los gráficos del FPG*/
    put(fpg,block,200+20*j,100+20*i);
    end
end

/*Dibujo la pieza actual dentro del terreno de juego, actualizándolo en cada frame mientras "Render" esté
vivo.El algoritmo recorre el bloque, -definido por 'nu'- a lo largo de las 5x5 casillas lógicas de la matriz
blok. Si el valor en cada casilla es 0 (eso no sólo depende del tipo de bloque, sino también de su
orientación actual),no se pintará nada en la posición que ocupa; si el valor es 1,pintará en esa casilla uno
de los gráficos 20x20 del fpg*/
for(i = 0;i < 5;i++)
    for(j = 0;j < 5;j++)
//kant=0: la figura no ha sido rotada
//kant=1: la figura ha sido rotada 90° en dirección horario respecto su posición original
//kant=2: la figura ha sido rotada 180° en dirección horario respecto su posición original
//kant=3: la figura ha sido rotada 270° en dirección horario respecto su posición original
        switch(kant)
            case 0: k = i; l = j; end
            case 1: k = 4-j; l = i; end
            case 2: k = 4-i; l = 4-j; end
            case 3: k = j; l = 4-i; end
        end
        block = blok[nu][i][j];
//Si existe pieza a pintar en esta casilla, la pinto
        if(block != 0)
            put(fpg,nu,140+20*(mux+k),100+20*(muy+l));
        end
    end
end

/*Dibujo los límites del cuadrado preview donde se visualiza el bloque siguiente, y el propio preview, a
cada frame.El límite de 5+2 viene de que el número de filas y columnas del cuadrado preview coinciden
con las medidas de las 2ª y 3ª dimensiones de la matriz block, con el objetivo de que el cuadro preview no
sea más que una plasmación gráfica de la matriz lógica block.*/
for(i = 0;i < 5+2;i++)

```

```

    for(j = 0;j < 5+2;j++)
//De entrada, las casillas lógicas están vacías
    block = 0;
//Si la casilla pertenece al reborde, la pinto con el gráfico 001
    if(i == 0 or i == 5+1 or j == 0 or j == 5+1)
        block = 1;
/*Si la casilla correspondiente de la matriz block dice que hay pieza, se establece qué gráfico le
corresponde del FPG*/
    else
        if(block[next][i-1][j-1]==1) block = next; end
    end
//Las casillas que permanecen vacías (al principio, todas menos las del borde)se pintan con el gráfico 010
    if(block == 0)block=10;end
    put(fpg,block,500+20*j,100+20*i);
    end
end

frame;
end //loop
end

//-----
/*Lógica del juego: gestiona el descenso y rotación de los piezas,el choque de los piezas con el
fondo,gestiona el game over...Todos estos cálculos se verán reflejados en el proceso "Render"*/
Process game()
private
    int neer=1;//Si vale cero el bloque actual está listo para bajar una línea
    int i,j,k,l,li,r,d,ct; //Variables auxiliares
end
begin
Detect(); //Lanza el proceso que autodetecta si las líneas horizontales están llenas
/*Determina qué pieza se muestran en pantalla (nu=pieza actual, next= pieza siguiente). Curiosamente (?)
siempre se empieza con la misma pieza*/
nu = rand(1,graad[difficulty-1]);
next = rand(1,graad[difficulty-1]);
/*Asigno la posición y la rotación inicial de las piezas. Y también la puntuación y el nivel del jugador.
Esto es necesario hacerlo debido a que este proceso puede ser llamado no solamente al principio de
ejecutarse el juego sino después del proceso gameover.*/
nux = ANCHO / 2; nuy = -1; kant = 0; score = 0; level = 1;
//Hace descender el bloque actual hasta que aparezca completo dentro del campo de juego
while(kannaar(nu,kant,nux,nuy)==0) nuy++; end
//Listo! Ya se puede empezar a jugar
loop
    level = 1+(score / 5000); //El nivel del jugador está basado en la puntuación
    if(level > 10) level = 10;end //No se puede tener un nivel mayor de 10
    BLOKSPEED = 16 - level; //Las piezas descienden más rápido para niveles mayores.

/*Es curioso el sistema de detección de las key (_left,_right,_down,__A): cuando se detecta su pulsación,
se aumenta el valor de cuatro variables (li,r,d,ct, respectivamente) -haciendo que valga diferente de 0-, y
entonces se comprueba si su valor es 1 -más otros condicionantes- para realizar la acción deseada
(moverse a la izquierda, derecha,abajo,rotar, respectivamente). Es un sistema para evitar que si se
mantiene pulsada mucho rato una tecla el juego se vuelva loco moviendo la ficha sin parar: de esta
manera se ha de soltar y volver a pulsar la tecla si se quiere repetir el efecto. Lo genial está en que

```

mientras la tecla esté pulsada, la variable irá aumentando su valor hasta el infinito, pero el if siguiente sólo se ejecutará cuando esa variable ha valido 1: es decir, que ya se puede tener pulsada la tecla todo el tiempo del mundo que el if sólo se ejecuta en el primer instante de la pulsación, cuando la variable deja de valer 0 a valer 1. Luego, cuando ya vale 2, el if que provoca el movimiento/rotación de la pieza ya no se ejecuta más. Puedes comprobar que si por ejemplo se cambia la sentencia de li++,r++,d++ o ct++ por li=1,r=1,d=1 o ct=1, la cosa no va porque mientras tengamos pulsada la tecla correspondiente, la pieza se irá moviendo/rotando sin parar, cosa que no interesa.\*/

```
//El usuario puede mover el bloque a la izquierda, derecha, abajo -más rápido- o rotar el bloque...
if(key(_left))li++; else li = 0; end
//...y se reaccionará a los deseos de mover el bloque por parte del usuario siempre que sea posible
if(li == 1 or ((li%(BLOKSPEED/5))==BLOKSPEED/5-1) and li > 5))
  //Si no se detecta posible colisión por la izquierda
  if(kannaar(nu,kant,--nux,nuy)==0) nux++; end
end
if(key(_right))r++; else r = 0; end
if(r == 1 or ((r%(BLOKSPEED/5))==BLOKSPEED/5-1) and r > 5))
  //Si no se detecta posible colisión por la derecha
  if(kannaar(nu,kant,++nux,nuy)==0) nux--; end
end
if(key(_down) or key(_B)) d++; else d = 0; end
if(d == 1 or ((d%(BLOKSPEED/5))==BLOKSPEED/5-1) and d > 5))
  //Si no se detecta posible colisión por abajo
  if(kannaar(nu,kant,nux,++nuy)==0) nuy--; end
end
if(key(_A)) ct++; else ct=0; end
/*Se comprueba si hay colisión simulando que la figura está rotada para verlo, sin rotarla de verdad
hasta que se ve que se puede.*/
if(ct==1 and kannaar(nu,(kant+1)%4,nux,nuy))
  kant = (kant+1)%4;
end

neer--; //Decrementa el tiempo (en frames) antes de que el bloque actual pueda bajar una línea
if(neer == 0) //Si es hora de bajar una línea...
  //Detecto si la posición vertical justo inferior está ocupada. Si es así...
  if(kannaar(nu,kant,nux,nuy+1)==0)
/*no puede descender una línea, se asigna y -podría ser cualquier otra variable- a true para que el
proceso sepa que es el turno del siguiente bloque*/
  y = 1;
else
  //Si puede, baja y añade una puntuación básica a la puntuación total
  nuy++; score = score + ScoreZomaar;
end
neer = BLOKSPEED; //El tiempo para el próximo descenso de un bloque es BLOKSPEED
else //No es hora de bajar una línea, así que no hay nuevo bloque hasta que sea la hora
  y = 0;
end

//Si y es true es hora de un nuevo bloque
if(y==1)
  for(i=0;i<5;i++)
    for(j=0;j<5;j++)
      switch(kant)
        case 0: k = i; l = j; end
        case 1: k = 4-j; l = i; end
```

```

                                case 2: k = 4-i;l = 4-j; end
                                case 3: k = j; l = 4-i; end
                                end
/*Por cada casilla que tenga la pieza actual, marco las casillas lógicas que ocupa dentro del campo de
juego como definitivamente ocupadas y pinto su gráfico determinado allí fijo.*/
    if(blok[nu][i][j] > 0) steen[l+nuy-1][k+nux-4] = nu; end
    end
end
nu = next;//Nuevo bloque!. El bloque siguiente 'next' es asignado al bloque actual 'nu'
next = rand(1,graad[difficulty-1]); //Y se escoge un nuevo tipo de bloque para 'next'
//Se reinician las variables que manejaban las órdenes de movimiento de piezas por parte del usuario
li = -300; r = -300; d = -300;
//Se reinicia la posición y rotación del nuevo bloque
nux = ANCHO/2; nuy = -1; kant = 0;
//Se desciende hasta que el fondo esté lleno
while(kannaar(nu,kant,nux,nuy)==0)
    nuy++;
    /*Si el descenso tarda demasiado, significa que el nuevo bloque no puede colocarse.
    El campo de juego está lleno y el jugador está game over*/
    if(nuy >= 4) gameover(); return; end
end
end // if y
frame;
end //loop
end

//-----
/*Devuelve si un tipo de bloque (su número de bloque viene dado en 'blokje' con rotación 'gekant')
puede caber o no (devuelve true o false) en una posición dentro del campo de juego.*/
Function KanNaar(int blokje,int gekant,x,y)
private
    int i,j,k,l;
end
begin
//Recorro las casillas de la matriz lógica blok
for(i=0;i<5;i++)
    for(j=0;j<5;j++)
        switch(gekant)
            case 0: k = i; l = j; end
            case 1: k = 4-j; l = i;end
            case 2: k = 4-i; l = 4-j; end
            case 3: k = j; l = 4-i; end
        end
        if(blok[blokje][i][j]==1) //Si esta casilla tiene una pieza
            //Línea importantísima que se ha de entender con lápiz y papel
            k = x+k-2; l = y+l-2;
            //No cabe porque se choca contra los límites
            if(k < 2 or k>ANCHO+1 or l<1 or l>ALTO-2)
                return false;
            else
                /*Si no pasa lo anterior, pero se choca contra casillas contiguas que están ocupadas por otras
                piezas,tampoco cabe*/.

```

```

    if(steen[l+1][k-2]>0) return false; end
    end
    end
    end
end
return true;
end

//-----

//Detecta las líneas horizontales completas y actúa en consecuencia
Process Detect()
private
    int i,j,k;
    int rij[ALTO];
end
begin
loop
//Controla cuántas posiciones se han cogido para cada regla horizontal
for(i = ALTO-1;i >= 0;i--)
    rij[i] = 0;
    for(j = 0;j < ANCHO;j++)
        if(steen[i][j] > 0)rij[i]++; end
    end
    //Si la cantidad de piezas en una línea es igual a la cantidad de posiciones, ¡línea detectada!
    if(rij[i] == ANCHO)
        score=score +ScoreVolleRij; //Se añaden los puntos por una línea a la puntuación del jugador
        //Mueve el resto de las líneas una posición para abajo
        for(k = i;k>0;k--)
            for(j = 0;j < ANCHO;j++)
                steen[k][j] = steen[k-1][j];
            end
        end
        i++;
    end
end //Primer for
frame;
end //Loop
end

//-----

//Proceso para gestionar el game over del jugador
Process gameover()
private
    int t;
end
begin
/*Si la puntuación alcanzada es mayor que los récords pertenecientes a la dificultad actual, se cambian
los records y se guardan en el fichero*/
if(score > highrecords[difficulty-1])
    highrecords[difficulty-1] = score;
    t = write(0,400,250,4,"¡Nuevo record!");
end
end

```

```

    save("scores.dat",highrecords);
end
x = write(0,400,300,4,"Game Over");
y = write(0,400,320,4,"Pulsa ENTER para volver a comenzar");
let_me_alone(); //Acaba con todos los procesos excepto el actual
render(); //Y reinicia el proceso Render
loop
//Se vuelve a comenzar después de un bonito efecto
if(key(__START))
//Se borran sólo los textos del Game Over
delete_text(x); delete_text(y);
//Si hay récords, se borran el texto de nuevo récord, también
if(t != 0) delete_text(t); end
//Se hace el efecto de rellenar el campo de juego por completo...
for(x=ALTO;x>=0;x--)
    for(y=0;y<ANCHO;y++) steen[x][y] = 1; end
    frame;
end
//...y seguidamente vacía para poder volver a jugar.
for(x=0;x<ALTO;x++)
    for(y=0;y<ANCHO;y++) steen[x][y] = 0; end
    frame;
end
//Y se empieza a jugar
game();
return;
end
frame;
end
end

//-----

//Proceso que pinta el fondo con un gradiente. Muy interesante
Process fill_map()
private
    int color;
end
begin
for(y=0;y<600;y++)
    color = 16+(y*16/600); //Gradiente vertical
    for(x=0;x<800;x++)
        put_pixel(x,y,color);
    end
end
end
end

```

## CAPÍTULO 14: ARTICULOS, CODIGOS Y TRUCOS VARIADOS

Este capítulo pretende ser una miscelanea, un cajón de sastre donde pueden tener cabida los más variados temas sobre la programación de Fénix: trucos, ideas, artículos, sugerencias, etc. No pretende tener ninguna coherencia interna como los capítulos anteriores: simplemente que aporte una referencia rápida de consulta para aquel aspecto o duda concreta que nos pudiera surgir a la hora de programar. Tampoco pretende ser un resumen exhaustivo: la mayoría de información recogida y recopilada aquí se ha extraído del foro de Fenix (<http://forum.divsite.net>), por lo que en cualquier caso es ese lugar el más idóneo para preguntar las dudas y problemas, y donde se aportarán las soluciones más eficaces y actualizadas.

### **Forzar a los procesos a realizar trayectorias concretas:**

Para ello se han de utilizar las funciones `PATH_FIND`, `PATH_WALL` y `PATH_GETXY`. El tema del seguimiento de caminos es un tema complejo, y para entenderlo en toda su amplitud, potencia y utilidad son necesarios ejemplos demasiado largos y espesos para lo que pretende este manual. Aquí sólo se tomará nota de la existencia de dichas funciones y se mostrará el ejemplo más elemental de su uso..

#### **PATH\_FIND (INT fichero, INT gráfico, INT x1, INT y1, INT x2, INT y2, INT opciones)**

A partir de un gráfico de **8 bits** especificado con los parámetros `fichero` y `gráfico`, indica si es posible encontrar algún camino que parta desde el punto en  $(x1, y1)$  y que concluya en el punto  $(x2, y2)$  evitando cualquier obstáculo que pueda encontrar a su paso. No se asegura que el camino encontrado sea óptimo al 100%, sin embargo se acercará razonablemente.

El gráfico de 8 bits contendrá normalmente una representación en miniatura del mapa o de la superficie del juego -en miniatura porque es tontería malgastar recursos del ordenador manipulando una representación al mismo tamaño que el mapa real-, donde cada punto del gráfico representará una casilla o un área que puede o no estar ocupada con un obstáculo. Cada punto del gráfico de 8 bits contiene un color de 0 a 255. En lugar de representar colores, indican para `path_find()` si la casilla en cuestión puede transparse (valor 0) o no (valor 1 o mayor). La función devuelve 1 (CIERTO) si se encontró un camino, o 0 si no es posible la comunicación entre los puntos de inicio y fin.

Para ir obteniendo sucesivamente los puntos intermedios de que consta el camino es preciso emplear la función `path_getxy()`.

Para configurar las rutinas de búsquedas de caminos mediante la función `path_wall()`, de forma que los valores superiores a 0 (hasta un valor máximo, indicado como parámetro a `path_wall()`) indiquen igualmente caminos transpasables, pero con un coste superior, indicado por el propio valor del punto. Así, las rutinas elegirán a veces un recorrido más largo, pero que pase por puntos de menor coste. Esta forma de operar puede tener gran utilidad en juegos en los que el mapa contiene casillas donde el de movimiento difiere; empleando esta facilidad, es posible hacer que los procesos escojan preferentemente moverse a través de carreteras y caminos antes que bosque a través, excepto cuando no hay ninguna carretera propicia para hacer el trayecto.

Los valores del parámetro de opciones pueden ser los siguientes: 0 para una búsqueda normal, o bien la suma de una o más de las siguientes constantes predefinidas:

- \*PF\_NODIAG: No permite el movimiento en diagonal (si no se especifica esta opción, sí se permite por defecto)
- \*PF\_REVERSE: Devuelve los puntos en orden inverso

### PATH\_WALL (INT valor)

Modifica el valor de "pared" empleado por la función path\_find(). Este valor de pared será un número que indica la posición mínima del color dentro de la paleta (perteneciente al gráfico de 8 bits que representa el camino) a partir del cual un punto del gráfico es intraspasable.

Es decir, si por ejemplo el parámetro vale 5, todos los píxeles del gráfico del camino pintados con los colores que ocupan las posiciones 0,1,2,3 y 4 de la paleta serán traspasables, y el resto no.

Hay que indicar que el orden de los colores traspasables influye en la facilidad de "traspase". En el ejemplo anterior, el color 1 es el más fácil de traspasar porque no ofrece apenas resistencia para los móviles; en cambio, el color 4 será más dificultoso, y en principio no será la opción escogida para crear un camino, a no ser que no haya otra opción. Es decir, la posición de los colores en la paleta se interpreta como el "coste" de movimiento de un punto dado, hasta llegar a la posición marcada por el parámetro, la cual marca la barrera infranqueable. Más concretamente: cada posición en la paleta requiere el doble de pasos que la anterior, hasta llegar a la posición infranqueable (1ª posición, 1 paso necesario; 2ª posición, 2 pasos necesarios, etc)

Por defecto, el valor de este parámetro está a 1, lo que significa que sólo los puntos a 0 serán navegados por el camino a resolver. Por cierto, recuerda que en gráficos de 8 bits, el color que ocupa la posición 0 de la paleta es considerado como el color transparente.

Si a la función se le pasa como parámetro un valor menor a 0, éste no será modificado.

### PATH\_GETXY (&POSICIONX, &POSICIONY)

Esta función actualiza dos variables cuyas direcciones de memoria se le pasan como parámetro, con el contenido del siguiente punto del camino encontrado por la última llamada válida a path\_getxy.

La función devuelve 0 si no quedan mas puntos en el camino (si ya se ha llegado al final), y en ese caso, no modifica las variables. La primera llamada actualizará x e y con los valores del punto de inicio del camino.

El tipo POINTER se utiliza en este caso para lograr que una función pueda devolver más de un valor a la vez: en este caso, devolverá dos valores que se guardaran en las respectivas variables cuyos nombres precedidos de & hallamos colocado como primer y segundo parámetro, y que serán actualizados en cada llamamiento a path\_getxy.

#### PARAMETROS:

POINTER X: Dirección de memoria de la variable entera POSICIONX donde se almacenará la coordenada horizontal del siguiente punto del camino.

POINTER Y: Dirección de memoria de la variable entera POSICIONY donde se almacenará la coordenada vertical del siguiente punto del camino.

El siguiente ejemplo movería el proceso actual a lo largo del camino encontrado por path\_find() (suponiendo que el mapa de búsqueda es de una cuarta parte del tamaño de la pantalla):

```
WHILE (path_getxy(&a, &b))
```

```
  x =a* 4;
```

```
  y =b* 4;
```

```
  FRAME;
```

```
END
```

Como ejemplo de su uso, aquí tienes el siguiente código, de HardJob. Para hacerlo funcionar necesitarás dos imágenes PNG llamadas pf\_map.png (de 200x150 píxeles y obligatoriamente de 8 bits) y pf\_2xmap.png (de píxeles 400x300 –el doble que la anterior-). El primer gráfico representará el mapa invisible que especificará por dónde podrá pasar el móvil y por donde no, y el segundo será la imagen que se muestre en pantalla.

La idea es similar a utilizar un mapa de durezas, pero en este caso, pf\_map.png (el “mapa de trayectorias”) constará tan sólo de un fondo negro y de diversas líneas blancas que harán la función de muros de un laberinto, delimitando así un cierto camino a seguir, ya que no será posible traspasarlas. La imagen visible pn\_2xmap.png, evidentemente, tendrá que mostrar algo acorde a ese mapa de trayectorias.

```

program PATH_FINDING;
global
    int mapa;
    int i
    int j;
    int p;
    int o;
    int camino;
end
begin
    set_mode(400,300,16);
    set_fps(40,0);

    //cargamos el mapa de pathfinding
    mapa = load_png("pf_map.png");
    //y mostramos el fondo
    put_screen(0,load_png("pf_2xmap.png"));

    //este sera el proceso que seguira el camino
    p = prota(80,290);
    //hacia este objetivo
    o = objetivo(310,30);

    write(0,200,280,4,"Pulsa una tecla para buscar el camino.");
    //Mientras no pulse ninguna tecla el programa no hace nada
    while(!scan_code) frame; end
    delete_text(0);

    //buscamos un camino, impidiendo movimientos en diagonal:
    camino = path_find(0,mapa,p.x/2,p.y/2,o.x/2,o.y/2,PF_NODIAG);
    //establecemos el color 4 como el minimo número no traspasable
    //de la paleta del mapa de pathfinding:
    path_wall(4);

    if(camino == 0) //si path_find devolvio 0, no hay camino posible
        delete_text(0);
        write(0,200,280,4,"No se encontro ningun camino valido.");
        frame;
    else //si se encuentra un camino,
        while(path_getxy(&i,&j)) //leer el siguiente punto del camino
            p.x=i*2; //multiplicar el punto por 2...
            p.y=j*2; /*ya que el mapa de pathfinding es la mitad

```

```

                                de grande que el grafico del fondo.*/
                                delete_text(0);
                                frame;
                                end
                                end
                                delete_text(0);
                                write(0,200,280,4,"El proceso ha llegado al destino.");
                                write(0,200,290,4,"Pulsa una tecla para salir...");
                                while(!scan_code) frame; end
                                exit();
end

process prota(x,y)
begin
    z=-1;
    graph = new_map(8,8,16);
    drawing_map(0,graph);
    drawing_color(rgb(255,255,0));
    draw_fcircle(3,3,3);
    loop
        frame;
    end
end

process objetivo(x,y)
begin
    z=0;
    graph=new_map(16,16,16);
    drawing_map(0,graph);
    drawing_color(rgb(0,255,255));
    draw_box(0,0,15,15);
    loop
        frame;
    end
end
end

```

### **Ordenar vectores y estructuras:**

En muchas ocasiones usaremos vectores simples, los valores de cuyos elementos (numéricos o de cadena) no estarán ordenados, y tendremos la necesidad de ordenar dichos elementos de menor a mayor en el caso de vectores numéricos o de la “a” a la “z” en el caso de vectores de cadenas. Es decir, que el valor del elemento vector[0] sea menor que el de vector[1], y éste que el de vector[2], y así.

En muchas ocasiones también usaremos vectores de estructuras, y desearemos ordenar los registros según uno de sus campos, típicamente numérico, de tal manera que según el valor de dicho campo los registros aparezcan dispuestos correlativamente.

Para lograr ambos obtenivos, Fénix dispone de una familia de funciones especializada en ordenaciones. Son las funciones SORT, KSORT y QUICKSORT, cada una con sus particularidades, como ahora veremos.

## SORT(VECTOR)

Ordena los elementos de un vector -tabla unidimensional- de datos simples. Los elementos del vector pueden ser de cualquier tipo simple válido: entero, decimal, cadenas...La ordenación se efectuará numéricamente o alfabéticamente según el caso.

Esta función tiene un segundo parámetro opcional de tipo entero que indica el número de elementos del vector (a partir del primero) que se quieren ordenar, por si no se quisiera ordenar todo el vector entero

Esta función devolverá 0 si ha habido algún error.

Si se desea ordenar los registros de un vector de estructuras a partir de los valores de los elementos de uno de sus campos, se recomienda usar KSORT. Para una ordenación más avanzada, consultar la función QUICKSORT.

### PARAMETROS:

XXX VECTOR: Matriz unidimensional cuyos elementos (de cualquier tipo) se ordenarán

INT NUMELEM: Número de elementos del vector que se ordenarán -opcional-

Un ejemplo sencillo que ordena un vector de enteros:

```
program hola;
const
  NUMELEM=18;
end
global
  int i;
  int vector1[NUMELEM-1]=1,5,3,7,2,4,5,3;
end
begin
  set_mode(640,480,16);
  write(0,10,10,3,"Vector1 sin ordenar");
  for(i=0;i<NUMELEM;i++)
    write(0,10,20+15*i,4,vector1[i]);
  end

  sort(vector1);

  write(0,150,10,3,"Vector1 ordenado");
  for(i=0;i<NUMELEM;i++)
    write(0,150,20+15*i,4,vector1[i]);
  end

  while(!key(_esc))
    frame;
```

```
end  
end
```

Fíjate que los elementos vacíos, al rellenar automáticamente con 0, se ordenan siempre al principio de todo, con lo que los valores del vector aparecen ordenados hacia el final. ¿Cómo podríamos ordenar solamente los elementos con contenido, obviando aquellos que están vacíos? Precisamente la función `sort` tiene un segundo parámetro opcional, que si se especifica, indica el número de elementos que deseas ordenar (si no se pone, se sobreentiende que son todos los elementos del vector). Ya que sabemos que los elementos con contenido son ocho, prueba a modificar del ejemplo la línea donde aparece la función `sort` por esta otra:

```
sort(vector1,8);
```

Al ejecutar esta variante podrás comprobar como ahora ha ordenado los primeros ocho elementos que se ha encontrado en el vector desordenado -los únicos que tenían valor- y los ha colocado, ordenados ya, en los ocho primeros puestos del vector ordenado, dejando el resto de casillas hasta el final llenas de 0.

Parece que está todo solucionado, pero surge un problema. En este ejemplo sabemos cuántos elementos llenos tenemos en el vector desordenado, pero, normalmente esa información no la tendremos. ¿Cómo podemos hacer entonces para ordenar sólo los elementos con contenido sin saber cuántos hay? A continuación voy a proponer una solución, pero no será una solución válida para todos los ámbitos.

Me explico: esta solución presupone que todos los elementos iguales a 0 son los elementos vacíos; en principio esto parece correcto, pero es muy frecuente que haya elementos “lentos” cuyo contenido sea precisamente 0, ya que el cero es un valor tan válido como otro cualquiera, por lo que estos elementos no se tendrían en cuenta a la hora de ordenar ya que serían considerados elementos vacíos, con el error que esto conlleva. Avisado estás.

Bien, la solución que propongo es sencilla : nos generamos nosotros mismos una función que, pasándole como parámetros el vector en cuestión y el número de elementos totales, nos devolverá el número de elementos llenos -los elementos con valores diferentes de 0-. A esta función la llamaremos “count”.

Fíjate que he comentado que esta función recibe como parámetro un vector. Recuerda que pasar vectores como parámetros de procesos/funciones no es tan sencillo como pasar cualquier otro tipo de dato, y se requiere un cuidado a la hora de escribir la notación correcta. Si tienes dudas, consulta el apartado correspondiente en un capítulo anterior.

```
program hola;  
const  
  NUMELEM=18;  
end  
global  
  int i, numero;  
  int vector1[NUMELEM-1]=1,5,3,7,2,4,5,3;  
end  
begin  
  set_mode(640,480,16);  
  write(0,10,10,3,"Vector1 sin ordenar");  
  for(i=0;i<NUMELEM;i++)  
    write(0,10,20+15*i,4,vector1[i]);  
  end  
  
  numero=count(&vector1,NUMELEM);  
  sort(vector1,numero);  
  
  write(0,150,10,3,"Vector1 ordenado");
```

```

for(i=0;i<NUMELEM;i++)
    write(0,150,20+15*i,4,vector1[i]);
end

while(!key(_esc))
    frame;
end
end

function count(int pointer matriz, int numero)
private
    int llenos=0;
    int j;
end
begin
    for(j=0;j<numero;j++)
        if(matriz[j]!=0)llenos++; end
    end
    return llenos;
end
end

```

También podemos comprobar que la función SORT funciona con vectores de cadenas:

```

program hola;
const
    NUMELEM=4;
end
global
    int i, numero;
    string vector1[NUMELEM-1]="Hola qué tal", "Yo muy bien", "Me alegro", "Gracias.";
end
begin
    set_mode(640,480,16);
    write(0,10,10,3,"Vector1 sin ordenar");
    for(i=0;i<NUMELEM;i++)
        write(0,10,20+15*i,4,vector1[i]);
    end

    sort(vector1);

    write(0,150,10,3,"Vector1 ordenado");
    for(i=0;i<NUMELEM;i++)
        write(0,150,20+15*i,4,vector1[i]);
    end

    while(!key(_esc))
        frame;
    end
end
end

```

### KSORT(VECTOR,CAMPO)

Ordena los registros de un vector de estructuras, a partir de los valores de uno de sus campos, especificado

éste como segundo parámetro (es lo que se llama la variable de ordenación).

Esta función tiene un tercer parámetro opcional de tipo entero que indica el número de elementos del vector (a partir del primero) que se quieren ordenar, por si no se quisiera ordenar todo el vector entero

Esta función devolverá 0 si ha habido algún error.

Si se desea ordenar un vector de datos simples (numéricos o de cadena) , se puede utilizar SORT. Para una ordenación más avanzada, consultar la función QUICKSORT.

### PARAMETROS

XXX VECTOR: Vector de estructuras cuyos registros se ordenarán

XXX CAMPO: Nombre del campo de la estructura cuyos valores definirán el orden de los registros.

INT NUMELEM: Número de elementos del vector que se ordenarán -opcional-

Un ejemplo de su utilización podría ser éste:

*Program sorting;*

*Type \_jugador*

*String nombre;*

*int puntuacion;*

*End*

*Const*

*maxjugadores = 5;*

*end*

*Global*

*\_jugador jugador[maxjugadores-1];*

*int i;*

*end*

*Begin*

*set\_mode(640,480,16);*

*// Rellenamos la estructura con algunos registros*

*jugador[0].nombre = "Jugador3";*

*jugador[1].nombre = "Jugador2";*

*jugador[2].nombre = "Jugador5";*

*jugador[3].nombre = "Jugador4";*

*jugador[4].nombre = "Jugador1";*

*jugador[0].puntuacion = 70;*

*jugador[1].puntuacion = 30;*

*jugador[2].puntuacion = 80;*

*jugador[3].puntuacion = 90;*

*jugador[4].puntuacion = 50;*

*//Mostramos el vector desordenado*

*Write(0,100,10,4,"Vector desordenado");*

*for(i=0; i<maxjugadores; i++)*

```

    write(0,100,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
end

//Ordeno el vector utilizando el campo nombre como variable de ordenación
ksort(jugador,jugador[0].nombre,maxjugadores);

//Mostramos el vector ordenado por el nombre
Write(0,300,10,4,"Vector ordenado por el nombre");
for(i=0; i<maxjugadores; i++)
    write(0,300,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
end

//Ahora ordeno el vector utilizando el campo puntuación como variable de ordenación
ksort(jugador,jugador[0].puntuación,maxjugadores);

//Mostramos el vector ordenado por puntuación
Write(0,500,10,4,"Vector ordenado por puntuación");
for(i=0; i<maxjugadores; i++)
    write(0,500,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
end

Repeat
    frame;
Until(key(_esc))

End

```

Fijarse que hemos utilizado en la función KSORT el tercer parámetro, opcional, para establecer el número de elementos que deseamos ordenar.

Como curiosidad comentar que la función SORT también es capaz de ordenar vectores de estructuras, pero siempre las ordenará siempre por su primer campo, ya que no se puede definir cuál es el campo que queremos que haga de variable de ordenación y por defecto siempre elige el primero sin posibilidad de cambiarlo. Para probar esto, puede añadir las siguientes líneas al ejemplo anterior, justo antes del bucle repeat/until del final:

```

// Volvemos a ordenar por nombre, ya que nombre es el primer campo de cada registro
sort(jugador);

// Mostramos el vector ordenado por nombre
Write(0,300,190,4,"Vector ordenado por el nombre");
for(i=0; i<maxjugadores; i++)
    write(0,300,200 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
end

```

Vemos que, efectivamente, ordena por nombre, pero con SORT es imposible en este ejemplo ordenar por puntuación, ya que esta campo no es el primero en los registros.

Otro programa (original de SplinterGU) que pone en práctica el uso de la función KSORT podría ser éste, fácil de entender:

```

program test002;

type struct1

```

```

string strng;
int intgr;
end;

type struct2
int intgr;
string strng;
end;

global
struct1 st[6] = "s10",10,"s5",5,"s210",210,"s3",3,"s33",33,"s12",12,"s3",3;
struct2 st2[6] = 10,"s10",5,"s5",210,"s210",3,"s3",33,"s33",12,"s12",3,"s3";
struct1 stbak1[6];
struct2 stbak2[6];
int i;
end

begin
/*Ordenamos el vector st con el campo String (el primero de la estructura) como variable de ordenación.
Lo hacemos dos veces: una ordenando todos los elementos del vector y otra sólo los cuatro primeros*/
write(0,0,0,0,"Orden por String (primer campo)");

stbak1 = st; /*Guardamos una copia de st en un vector auxiliar stbak1, para mantener la (des)ordenación
de los datos original*/
write(0,0,10,0,"Toda la lista");
ksort(st, st[0].strng);
/*Fijarse que para averiguar el número de elementos del vector se divide el tamaño total del vector por el
tamaño de uno de sus elementos. */
for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,0,(2+i)*10,0,st[i].intgr+" , "+st[i].strng);
end

st=stbak1; //Volvemos a desordenar st tal como estaba al principio del programa
write(0,160,10,0,"Los primeros 4");
ksort(st, st[0].strng, 4);
for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,160,(2+i)*10,0,st[i].intgr+" , "+st[i].strng);
end

/*Ordenamos el vector st con el campo Int (el segundo de la estructura) como variable de ordenación.
Lo hacemos dos veces: una ordenando todos los elementos del vector y otra sólo los cuatro primeros*/
write(0,0,100,0,"Orden por Int (segundo campo)");

st=stbak1;
write(0,0,110,0,"Toda la lista");
ksort(st, st[0].intgr);
for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,0,100+(2+i)*10,0,st[i].intgr+" , "+st[i].strng);
end

st=stbak1;
write(0,160,110,0,"Los primeros 4");
ksort(st, st[0].intgr, 4);

```

```

for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,160,100+(2+i)*10,0,st[i].intgr+ " , "+st[i].strng);
end

write(0,0,190,0,"Presione ESC para el siguiente test");
while(!key(_ESC)) frame; end//Hasta que no se pulse ESC no se hace nada
while(key(_ESC)) frame; end //Mientras se esté pulsando tampoco
delete_text(0);

/*A continuación realizo las mismas cuatro operaciones (ordenar por el primer campo todos los
elementos, ordenar por el primer campo los 4 primeros elementos, ordenar por el segundo campo todos
los elementos y ordenar por el segundo campo los 4 primeros elementos) pero efectuándolas sobre el
vector st2*/
write(0,0,0,0,"Orden por Int (primer campo)");

stbak2=st2;
write(0,0,10,0,"Toda la lista");
ksort(st2, st2[0].intgr);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,0,(2+i)*10,0,st2[i].intgr+ " , "+st2[i].strng);
end

st2 = stbak2;
write(0,160,10,0,"Los primeros 4");
ksort(st2, st2[0].intgr, 4);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,160,(2+i)*10,0,st2[i].intgr+ " , "+st2[i].strng);
end

write(0,0,100,0,"Orden por String (segundo campo)");

st2 = stbak2;
write(0,0,110,0,"Toda la lista");
ksort(st2, st2[0].strng);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,0,100+(2+i)*10,0,st2[i].intgr+ " , "+st2[i].strng);
end

st2 = stbak2;
write(0,160,110,0,"Los primeros 4");
ksort(st2, st2[0].strng, 4);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,160,100+(2+i)*10,0,st2[i].intgr+ " , "+st2[i].strng);
end

while(!key(_ESC)) frame; end

end

```

### **QUICKSORT(VECTOR,TAMAÑO ELEM,NUMELEM,OFFSET,TAMAÑO DATO,TIPO DATO)**

Ordena un vector -tanto simples como de estructuras- utilizando el algoritmo matemático de ordenamiento QuickSort.

Esta función es muy útil para ordenar vectores de tipos definidos por el usuario (datos TYPE), en donde uno de sus campos actúe como la variable de ordenamiento (es decir, que sus valores sean los que se usarán para ordenar todos los diferentes datos de ese tipo). No obstante, una de las limitaciones de esta función es que la variable de ordenamiento no puede ser de tipo cadena.

Para vectores simples, puede usarse la función SORT. Para vectores de estructuras, puede usarse KSORT.

Esta función devolverá 0 si ha habido algún error.

#### PARAMETROS

POINTER VECTOR: Puntero al primer elemento del vector que será ordenado.

INT TAMAÑOELEMENTO: Tamaño de un elemento/registro del vector, en bytes.

INT NUMELEMENTO: Número de elementos totales del vector

INT OFFSET : Número de bytes respecto al inicio de un elemento donde se sitúa el campo que ejerce de variable de ordenamiento.

BYTE TAMAÑO DATO : Tamaño en bytes del campo que ejerce de variable de ordenamiento. Por ejemplo, float=4, int=4, word=2, byte=1.

BYTE TIPO DATO: El tipo del campo que ejerce de variable de ordenamiento (0=entero, 1=decimal)

```
Program sorting;
Type _jugador
  String nombre;
  int puntuacion;
End
Const
  maxjugadores = 5;
end
Global
  _jugador jugador[maxjugadores-1];
  int i;
end
Begin
  set_mode(640,480,16);
  // Rellenamos la estructura con algunos registros
  jugador[0].nombre = "Jugador3";
  jugador[1].nombre = "Jugador2";
  jugador[2].nombre = "Jugador5";
  jugador[3].nombre = "Jugador4";
  jugador[4].nombre = "Jugador1";

  jugador[0].puntuacion = 70;
  jugador[1].puntuacion = 30;
  jugador[2].puntuacion = 80;
  jugador[3].puntuacion = 90;
  jugador[4].puntuacion = 50;

  //Mostramos el vector desordenado
  Write(0,100,10,4,"Vector desordenado");
```

```

for(i=0; i<maxjugadores; i++)
    write(0,100,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
end

```

*/\*IMPORTANTE: La función quicksort() no puede ser usada para ordenar Strings, (ya que una cadena en Fénix es en realidad un puntero a la cadena real, con lo que quicksort() ordenaría las direcciones de memoria). Esto implica que en este ejemplo no se pueda ordenar por el campo nombre.\*/\**

```

// Ordeno por el campo puntuacion.
quicksort(&jugador[0],sizeof(_jugador),maxjugadores,sizeof(String),sizeof(int),0);

```

```

//Mostramos el vector ordenado
Write(0,300,10,4,"Vector ordenado");
for(i=0; i<maxjugadores; i++)
    write(0,300,20 +10*i ,4, jugador[i].nombre + " - " + jugador[i].puntuacion);
end

```

```

Repeat
    frame;
Until(key(_esc))

```

End

Fijarse en los valores que tienen los parámetros de la función QUICKSORT. El primero hemos dicho que es un puntero al primer elemento del vector que queremos que se ordene: como queremos ordenar todo el vector, el primer elemento que queremos que se ordene será el primer elemento del vector, así que su puntero será &jugador[0]. El segundo parámetro ha de valer el tamaño en bytes de un registro del vector; como este vector es de elementos de tipo \_jugador, para averiguar el tamaño en bytes de un elemento de este tipo hacemos uso de la función sizeof así: sizeof(\_jugador). El tercer elemento es el número de elementos totales del vector: maxjugadores. El cuarto es el número de bytes que hay entre el principio de cada registro y la posición dentro de éste que ocupa la variable de ordenamiento. Como bien se comenta en el código, la función QUICKSORT es incapaz de utilizar variables de ordenamiento que sean de tipo cadena, así que el campo “nombre” del tipo \_jugador no lo podremos utilizar para ordenar, pero sí el campo “puntuacion”. Este campo es el segundo de la estructura, y por lo tanto hay una serie de bytes entre el principio de ésta y este campo. En concreto, esta serie de bytes viene dada por lo que ocupen el conjunto de campos que haya antes de “puntuacion”, y en nuestro caso, sólo es uno, “nombre”, de tipo cadena. Así pues, para saber cuanto ocupa este campo “nombre” que nos dará la diferencia de bytes entre el principio de la estructura y la posición del campo “puntuacion”, usaremos como antes la función sizeof así: sizeof(String). El quinto elemento es el tamaño en bytes del campo que será la variable de ordenación. El campo “puntuacion” es un Int, así que pondremos sizeof(int). Y el sexto elemento indica el tipo de campo que es la variable de ordenación; como “puntuacion” es entera, pondremos un 0. (Este último parámetro sirve para distinguir los Int de los Float, ya que ambos ocupan 4 bytes en memoria y no se pueden distinguir utilizando el quinto parámetro).

Otro programa (original de SplinterGU) que pone en práctica el uso de la función QUICKSORT podría ser éste, fácil de entender:

```

program test001;

```

```

type struct1
    int intgr;
    string strng;
end;

```

```

type struct2
  string strng;
  int intgr;
end;

global
int array[13]=33,42,12,3,92,34,11,99,66,44,42,55,1,20;
float flt[]=33.0, 42.2, 12.21, 3;
int i;

struct1 st[6]=10,"s10",5,"s5",210,"s210",3,"s3",33,"s33",12,"s12",3,"s3";
struct2 st2[6]="s10",10,"s5",5,"s210",210,"s3",3,"s33",33,"s12",12,"s3",3;
end

begin
//Ordenamos el vector de enteros.
/*Fijarse en los valores de los parámetros de la función quicksort(): para averiguar el número de
elementos del vector se divide el tamaño total del vector por el tamaño de uno de sus elementos. El offset
es 0 porque es un vector simple. Los demás valores son fácilmente deducibles.*/
quicksort(&array[0], sizeof(array[0]), sizeof(array)/sizeof(array[0]), 0, sizeof(array[0]), 0);
write(0,0,0,0,"Orden array enteros");
for(i=0;i<sizeof(array)/sizeof(array[0]);i++)
write(0,0,10+i*10,0,array[ i]);
end

//Ordenamos el vector de decimales
quicksort(&flt[0], sizeof(flt[0]), sizeof(flt)/sizeof(flt[0]), 0, sizeof(flt[0]), 1);
write(0,160,0,0,"Orden array floats");
for(i=0;i<sizeof(flt)/sizeof(flt[0]);i++)
write(0,160,10+i*10,0,flt[ i]);
end

write(0,0,190,0,"Presione ESC para el siguiente test");
while(!key(_ESC)) frame; end //Hasta que no se pulse ESC no se hace nada
while(key(_ESC)) frame; end //Mientras se esté pulsando tampoco
delete_text(0);

//Ordenamos el vector st, que tiene el campo entero el primero de cada registro
write(0,0,0,0,"Orden por Int (offset 0)");
quicksort(&st, sizeof(st[0]), sizeof(st)/sizeof(st[0]), 0, sizeof(st[0].intgr), 0);
for(i=0;i<sizeof(st)/sizeof(st[0]);i++)
write(0,0,10+i*10,0,st[ i ].intgr+" ", "+st[ i ].strng);
end

/*Ordenamos el vector st2, que tiene el campo entero el segundo de cada registro,
después de un campo de tipo cadena. Recordemos que quicksort() no es capaz de utilizar
variables de ordenamiento que sean de tipo cadena.*/
write(0,160,0,0,"Orden por Int (offset 4)");
quicksort(&st2, sizeof(st2[0]), sizeof(st2)/sizeof(st2[0]), 4, sizeof(st2[0].intgr), 0);
for(i=0;i<sizeof(st2)/sizeof(st2[0]);i++)
write(0,160,10+i*10,0,st2[ i ].intgr+" ", "+st2[ i ].strng);
end

while(!key(_ESC)) frame; end

```

end

### Creación de combos de teclas:

Si has jugado alguna vez a un juego de luchas tipo “StreetFighter” o “MortalKombat”, seguro que sabrás que existen en esta clase de juegos determinadas combinaciones de teclas que el jugador ha de pulsar rápidamente para lograr realizar una acción especial, como un patada más potente, o recargar energía, etc. Estas combinaciones de teclas son los llamados combos, y la gracia de estos juegos recae en parte en que el jugador sepa realizar los combos más rápido y mejor que su contrincante, para tener más posibilidades de ganarle. Normalmente los combos son combinaciones de teclas más o menos difíciles (más difíciles será realizarlas si el jugador obtiene más beneficio) que se deberán realizar en un tiempo máximo, ya que si el jugador se demora demasiado intentando ejecutar el combo, éste no se considerará realizado correctamente.

En este apartado veremos un código de ejemplo de implementación de un combo. En concreto, el combo será pulsar una detrás de otra seis teclas : `_up, _down, _up, _down, _right, _right, _down`. Y entre tecla y tecla la pulsación no deberá demorarse más de 3 decimas o si no se considerará que el combo se ha deshecho.

Para ejecutar el código tendrás que crearte un archivo FPG llamado “wecombo.fpg” con un único PNG en su interior, el cual será una bola de 40x40, con código 001.

Mientras no logremos ejecutar el combo, en pantalla aparecerá una bola que se moverá con los cursores. También aparece en pantalla un marcador que enseña si se ha pulsado alguna de las teclas que forman el combo –aparece el código numérico de la tecla concreta: por ejemplo, la tecla `_up` se corresponde con el número 72-. Si se logra conseguir un combo, verás que en el marcador, evidentemente, aparecerán que todas las teclas del combo han sido pulsadas, y se mostrará el efecto del combo, que es reventar la bola en bolitas pequeñas que salen disparadas por todos los ángulos.

*/\*Ejemplo de combo. Se utiliza un patron de teclas en un array que representa la combinaciøn del combo. Se utiliza otro array (tabla) donde se guardan las teclas que se pulsan, funciona como una cola, de manera que las nuevas teclas eliminan a las más antiguas. Se controla el tiempo entre las teclas, si es más de 30 centésimas se borra el array.\*/*

*Program combo;*

*Global*

*int tiempo; //guarda el tiempo entre teclas*

*int cola[6]; //aquí se guardan las teclas*

*int combopatron[6]=\_up,\_down,\_up,\_down,\_right,\_right,\_down; // el patrøn de teclas para el combo*

*int comboflag=0; // cuando valga 1, habrá combo*

*End*

*Local*

*Int n;*

*End*

*Begin*

*load\_fpg("wecombo.fpg");*

```

define_region(1,0,0,320,200);
While(fading) Frame; End;
write(0,20,10,0,"arr, aba, arr, aba, der, der, aba");
For(n=0;n<7;++n)
    write_var(0,20+n*30,30,0,cola[n]);
End

juego();
While(get_id(Type juego));
    Frame;
End
delete_text(0);
let_me_alone();
unload_fpg(0);
End

/* Mete un dato en la cola. Lo hace al final,pero se podría hacer al revés y que entrara en el primero.
   Compara el patrón con la cola para ver si hay combo. Devuelve 1 en ese caso y pone a 1 la variable
   "comboflag". Devuelve cero en caso contrario. Bastaría con la variable o con el valor devuelto pero
   asi tienes mas posibilades para procesar el combo.*/

Process coladato(dato)
Begin
    For(n=0;n<6;++n)//muevo los datos adelante para hacer un hueco
        cola[n]=cola[n+1];
    End
    cola[6]=dato; // meto el dato nuevo
    For(n=0;n<7;++n) // comparo
        If(cola[n]!=combopatron[n]) Return(0); End /*había alguno distinto*/
    End
    comboflag=1; //Combo!!
End

// controla el tiempo entre teclas y borra la cola si es mucho
Process controltiempo()
Begin
    If((timer[9]-tiempo)>30) // 30 centesimas
        For(n=0;n<7;++n) cola[n]=0; End
    End
    tiempo=timer[9]; // nuevo tiempo
End

Process juego()
Private
    Int arriba, abajo, derecha, izquierda;
end
Begin
    graph=1;
    x=160;
    y=100;
    Frame;

```

```

Loop
  If(key(_esc)) break; End

  If(key(_up) && y>20)
    y=y-3;
    If(!arriba)
      controltiempo();
      If(coladato(_up)) haz_combo(x,y); End
      arriba=1;
    End
  End
  If(!key(_up)) arriba=0; End

  If(key(_down) && y<180)
    y=y+3;
    If(!abajo)
      controltiempo();
      If(coladato(_down)) haz_combo(x,y); End
      abajo=1;
    End
  End
  If(!key(_down)) abajo=0; End

  If(key(_right) && x<300)
    x=x+5;
    If(!derecha)
      controltiempo();
      If(coladato(_right)) haz_combo(x,y); End
      derecha=1;
    End
  End
  If(!key(_right)) derecha=0; End

  If(key(_left) && x>20)
    x=x-5;
    If(!izquierda)
      controltiempo();
      If(coladato(_left)) haz_combo(x,y); End
      izquierda=1;
    End
  End
  If(!key(_left)) izquierda=0; End

  Frame;

End
End

//Realiza el efecto del combo, que es crear 10 bolitas pequeñas
Process haz_combo(x,y)
Begin
  For(n=0;n<10;n=n+1)
    bolita(x,y,rand(0,360));
  End
End

```

```

    comboflag=0;
End

/*Muestra el efecto visual del combo, que es que las bolitas salgan disparadas en direcciones diferentes*/
Process bolita(x,y,angle)
Begin
    angle=angle*1000;
    graph=1;
    size=50;
    While(!out_region(id,1))
        advance(5);
    Frame;
End
End

```

### **Trabajar con tablas Blendop:**

“Blendop” proviene del apócope “Blending Operation”. Es una operación de mezcla de colores (transparencias de porcentaje variable, entintado, iluminación, escala de grises, etc) que se aplica a un gráfico antes de ser dibujado, (por lo que este efecto suele ser lento). Se suele aplicar a un gráfico para modificarlo de forma permanente, para evitar el problema de la velocidad.

La idea del funcionamiento es la siguiente: primero se crea lo que se llama un tabla blendop, con la función `blendop_new()`; la cual devuelve un identificador de dicha tabla. Dicha tabla tendrá que contener en su interior un conjunto de operaciones (de “efectos”) que se definirán a continuación con las funciones `blendop_tint` -para efectos de entintado-, `blendop_translucency` -para transparencias-, `blendop_grayscale` -para escala de grises-, `blendop_intensity` -para el brillo-, etc, las cuales hacen uso del identificador de la tabla blendop previamente devuelto por `blendop_new`. En este momento tendremos una tabla blendop que contiene una serie de efectos en cadena listos para aplicar a un/unos gráfico/s determinados. Falta decir cuáles, y podremos observar entonces el resultado. Hay diferentes maneras de asignar una tabla blendop a un gráfico: o bien con la variable local “Blendop” (si estamos hablando del gráfico de un proceso), o bien con las funciones `blendop_apply` o `blendop_assign` si estamos hablando de un gráfico “estándar”. La diferencia entre `blendop_apply` y `blendop_assign` está en que mediante la primera los efectos se aplican al gráfico de forma permanente de forma que éste queda alterado incluso después de haber descargado la tabla blendop de memoria (con `blendop_free`), y mediante la segunda el gráfico sólo permanecerá alterado con los efectos de la tabla blendop asignada mientras ésta esté cargada en memoria.

#### **BLENDOP\_NEW()**

Crea una nueva tabla blendop en memoria. La tabla blendop permite crear efectos de colores y mezclas al dibujar un gráfico, incluyendo cosas tales como colorizar una superficie, o emplear un porcentaje variable de transparencia.

Una tabla blendop puede usarse de tres maneras diferentes, como ya hemos dicho:

- Activarla en un proceso, asignando el identificador devuelto por esta función a la variable local BLENDOP. Esto permite dibujar cualquier proceso con el efecto de color deseado. El rendimiento es el mismo que dibujar un gráfico transparente.
- Asignarla a un gráfico, empleando la función BLENDOP\_ASSIGN. Cualquier uso posterior de ese gráfico empleará la tabla blendop para dibujarlo, (aunque la variable local BLENDOP tiene preferencia si está presente -y los efectos no se acumularán-.)
- Aplicarla a un gráfico, empleando la función BLENDOP\_APPLY. Los puntos del gráfico serán modificados de acuerdo con la tabla. Esto es útil para colorizar gráficos y demás efectos parecidos.

Una tabla blendop ocupa una cantidad considerable de memoria (256K) por lo que se aconseja no excederse en su uso. Por ejemplo, un programa que cree 20 tablas blendop necesitará unos 5M de RAM sólo para almacenarlas. Si una tabla sólo se va a usar temporalmente, se recomienda emplear BLENDOP\_FREE para liberarla.

Una tabla blendop recién creada no realiza ninguna operación, por lo que usarla para dibujar no alterará el aspecto del gráfico. Es necesario especificar las características del efecto deseado empleando cualquiera de las funciones BLEND\_XXX. La mayoría de estas funciones son acumulables sobre una misma tabla, por lo que es posible especificar un efecto de coloreado con BLENDOP\_TINT y seguidamente un porcentaje de transparencia con BLENDOP\_TRANSLUCENTY , por ejemplo.

VALOR DE RETORNO: INT : Identificador de la nueva tabla BLENDOP

### **BLENDOP\_TINT ( BLENDOP, CANTIDAD, R, G, B )**

Aplica un color a una tabla blendop. Dada una tabla blendop creada con BLENDOP\_NEW , esta función realiza una mezcla entre el color cuyas componentes recibe como parámetros, y los colores del gráfico (todo ello en el gráfico origen de la operación). El valor de cantidad especificado como segundo parámetro indica la cantidad del nuevo color que se fusionará en cada pixel. El resultado será que cada color del gráfico se transforma según la fórmula  $(CANTIDAD * NUEVO + (1.0 - CANTIDAD) * VIEJO)$ .

Esta es una función lenta, que no es recomendable llamar periódicamente en el programa, al igual que ocurre con la mayoría de funciones BLENDOP.

#### PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP  
FLOAT CANTIDAD: Porcentaje relativo al nuevo color (0.0 a 1.0)  
INT R : Componente Roja del color (de 0 a 255)  
INT G : Componente Verde del color (de 0 a 255)  
INT B : Componente Azul del color (de 0 a 255)

### **BLENDOP\_INTENSITY ( BLENDOP, INTENSIDAD )**

Aplica transparencia a una tabla blendop. Dada una tabla blendop creada con BLENDOP\_NEW, esta función aplica un factor de intensidad a todos los colores del gráfico de origen, es decir, multiplica todas sus componentes por el valor especificado como segundo parámetro. Un valor de 0.5 oscurecería todos los colores a un 50%, mientras un valor de 2.0 aumentaría la intensidad de los colores al doble aproximadamente.

Esta es una función lenta, que no es recomendable llamar periódicamente en el programa, al igual que ocurre con la mayoría de funciones BLENDOP.

#### PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP  
FLOAT INTENSIDAD: Cantidad de color

### **BLENDOP\_TRANSLUCENCY ( BLENDOP, OPACIDAD )**

Aplica transparencia a una tabla blendop. Dada una tabla blendop creada con BLENDOP\_NEW, esta función aplica un valor de transparencia según el parámetro de opacidad. 1.0 equivale a un dibujo completamente opaco (tal como es creada la tabla), mientras 0.0 dibujaría un gráfico completamente transparente. Este efecto es semejante al proporcionado por la variable local ALPHA, aunque mucho más preciso.

Esta es una función lenta, que no es recomendable llamar periódicamente en el programa, al igual que ocurre con la mayoría de funciones BLENDOP.

#### PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP

FLOAT OPACIDAD: Cantidad de opacidad, entre 0.0 y 1.0

### **BLENDOP\_GRAYSCALE ( BLENDOP, METODO )**

Aplica conversión a escala de grises a una tabla blendop. Dada una tabla blendop creada con BLENDOP\_NEW, esta función aplica una fórmula al gráfico original que convierte todos los colores en su equivalente en escala de grises. Para realizar esta operación se soportan tres métodos de conversión diferentes:

1: Luminancia. El color resultante tiene una intensidad de  $0.3 * R + 0.59 * G + 0.11 * B$ .

Normalmente, este es el método que ofrece una mayor nitidez en imágenes fotográficas.

2: Desaturación. El color resultante es la media de las tres componentes R, G y B. Recomendable para gráficos oscuros o coloreados a mano con cierta variedad de color.

3: Máximo. El color resultante es el máximo entre las tres componentes. Recomendable para gráficos de líneas con pocos colores de diferencias marcadas.

#### PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP

INT METODO: Método a aplicar (1,2 ó 3)

### **BLENDOP\_IDENTITY ( BLENDOP )**

Reinicia una tabla blendop. Dada una tabla blendop creada con BLENDOP\_NEW, esta función la reinicia, es decir, la devuelve a sus valores originales (consistentes en no modificar el gráfico al dibujarlo).

En ocasiones puede ser útil cambiar las características de una tabla blendop, sin emplear los efectos ya acumulados. Esta función la devuelve a su estado inicial, tal como fue creada.

#### PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP

### **BLENDOP\_SWAP ( BLENDOP )**

Intercambia el efecto de una operación en una tabla blendop. Dada una tabla blendop creada con BLENDOP\_NEW, esta función intercambia las operaciones realizadas con el gráfico de origen y las realizadas con el de destino. Las funciones BLENDOP\_TINT, BLENDOP\_GRAYSCALE y BLENDOP\_INTENSITY modifican sólo el gráfico original. Utilizando esta función, es posible hacer que

las operaciones acumuladas alteren el gráfico destino.

Dicho en palabras menos técnicas: esta función altera el orden de la tabla blendop de manera que los efectos se apliquen sobre el fondo en lugar de sobre el gráfico, permitiendo efectos de tipo "lente".

Es preciso entender que todos los pixels a 0 en un gráfico son ignorados por las rutinas de dibujo: una tabla blendop que haga modificaciones sobre el gráfico de destino sólo afectará a los pixels en pantalla que haya debajo de pixels que no estén a cero en el gráfico a dibujar.

Una tabla blendop recién creada tiene una opacidad del 0% para el gráfico de destino. Si se intercambia con esta función sin haber llamado a `BLENDOP_TRANSLUCENCY`, el efecto resultante será que los pixels del gráfico a dibujar no serán visibles de por sí, pero los pixels que haya bajo todos ellos serán modificados según las operaciones que hubiese en la tabla blendop acumuladas antes de llamar a `BLENDOP_SWAP`. Si se desea un gráfico parcialmente visible, que haga algún tipo de efecto (como oscurecer o entintar) sobre la zona de pantalla que haya bajo él, es preciso llamar a `BLENDOP_TRANSLUCENCY` antes de `BLENDOP_SWAP`.

Esta es una función lenta, que no es recomendable llamar periódicamente en el programa, al igual que ocurre con la mayoría de funciones `BLENDOP`.

PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP

**BLENDOP\_FREE ( BLENDOP )**

Destruye una tabla blendop Esta función libera la memoria ocupada por una tabla blendop. Es responsabilidad del programador asegurarse de que la tabla blendop no se encuentre asignada a ningún gráfico ni a ninguna variable BLENDOP de ningún proceso en el momento de destruirla. Lo contrario se considera una condición de error grave y puede dar resultados impredecibles.

PARÁMETROS:

INT BLENDOP : Identificador de una tabla BLENDOP

**BLENDOP\_ASSIGN ( LIBRERÍA, GRÁFICO, BLENDOP )**

Asigna una tabla blendop a un gráfico Dada una tabla blendop creada con `BLENDOP_NEW`, esta función la asigna a un gráfico. Cualquier dibujo posterior del gráfico en pantalla, que no sea en un proceso que contenga su propia tabla blendop en la variable local `BLENDOP`, utilizará la tabla blendop para dibujarlo.

Si se especifica 0 como identificador blendop, cualquier tabla blendop asignada al gráfico dejará de usarse.

No es válido liberar una tabla blendop de memoria (con `BLENDOP_FREE`) cuando aún existan gráficos que la tengan asignada.

PARÁMETROS:

INT LIBRERIA: Número de librería FPG

INT GRAFICO : Número de gráfico dentro de la librería  
INT BLENDOP : Identificador de una tabla BLENDOP

### BLENDOP\_APPLY ( LIBRERÍA, GRÁFICO, BLENDOP )

Aplica una tabla blendop a los pixels de un gráfico. Esta función modifica los pixels de un gráfico, aplicando las operaciones que la tabla de blendop indica que deben hacerse sobre el gráfico original. La transparencia se aplicará como oscurecimiento (mezclando el mapa con el negro).

El gráfico quedará modificado en memoria sin necesidad de seguir manteniendo la tabla blendop en memoria y sin hacerle ninguna referencia a la misma.

#### PARÁMETROS:

INT LIBRERIA: Número de librería FPG  
INT GRAFICO : Número de gráfico dentro de la librería  
INT BLENDOP : Identificador de una tabla BLENDOP

Este tema es mucho más sencillo de lo que parece. Con el siguiente ejemplo se verá perfectamente cómo funciona todo. Simplemente consiste en mostrar un gráfico -generado a partir de *write\_in\_map*- asociado al proceso "bola" de cinco maneras distintas. La manera que se muestra en el centro es el gráfico sin habersele aplicado ningún tipo de tabla blendop. La de la esquina superior izquierda es el producto de haber aplicado al gráfico inicial un efecto de tinte, el cual se puede variar cambiando los valores de las variables c,r,g y b. La de la esquina superior derecha es el producto de haber aplicado al gráfico inicial un efecto de transparencia, el cual se puede variar cambiando los valores de la variable c. La de la esquina inferior izquierda es el producto de haber aplicado al gráfico inicial un efecto de escalado de grises, el cual se puede variar cambiando los valores de la variable m. La de la esquina inferior derecha es el producto de haber aplicado al gráfico inicial el efecto de brillo, el cual se puede variar cambiando los valores de la variable c.

En este ejemplo no se pone en práctica, pero evidentemente, a una misma tabla blendop se puede aplicar más de un efecto (tinte, transparencia,etc) mediante las funciones correspondientes, para lograr un resultado sumativo cuando se le asigne a un gráfico.

```
Program Test_BLENDOPS;  
Global  
//CAMBIAR ESTOS VALORES PARA OBSERVAR OTROS EFECTOS  
Float c=0.5; //Cantidad de tintado, de transparencia y de brillo (de 0.0 a 1.0)  
byte m=1; //Método de escalado de grises (1,2 ó 3)  
byte r=0; //Cantidad de rojo para el tinte  
byte g=255; //Cantidad de verde para el tinte  
byte b=0; //Cantidad de azul para el tinte  
end  
Private  
int ble;  
End  
Begin  
set_mode(640,480,16);  
//Ponemos un fondo de color  
fondo();
```

```

//Antes de aplicar ninguna tabla blendop
bola(320,240,0);

//Aplicamos un tinte determinado según el valor de c,r,g y b
write(0,60,20,4,"Tinte");
ble=blendop_new();
blendop_tint(ble,c,r,g,b);
//blendop_swap(ble);
bola(60,50,ble);

//Aplicamos una transparencia determinada según el valor de c
write(0,550,20,4,"Transparencia");
ble=blendop_new();
blendop_translucency(ble,c);
bola(550,50,ble);

//Aplicamos un escalado de grises según el método determinado por m
write(0,60,430,4,"Escala de grises");
ble=blendop_new();
blendop_grayscale(ble,m);
bola(60,450,ble);

//Aplicamos un brillo determinado según el valor de c
write(0,550,430,4,"Brillo");
ble=blendop_new();
blendop_intensity(ble,c);
bola(550,450,ble);

Repeat
  Frame;
Until(key(_esc))
let_me_alone();
blendop_free(ble);
End

Process bola(x,y, int ble)
Begin
  graph = write_in_map(0,"ABCD",4);
  blendop_assign(0,graph,ble);
  /*Otra manera de hacer lo mismo que la línea anterior es utilizar la variable local BLENDOP, así:
  blendop=ble;
  La otra alternativa, blendop_apply no sé por qué hace petar el programa.*/
  Loop
    Frame;
  End
End

Process fondo()
begin
graph=new_map(640,480,16);
map_clear(0,graph,rgb(200,30,100));
x=320;
y=240;

```

```

z=1;//Para estar por debajo del proceso bola
loop
frame;
    end
end

```

Un uso de las tablas blendop está en el ámbito de generación de sombras. Si se dispone de un gráfico ya visible, a partir de él se puede generar una sombra suya sin tener que disponer de otro gráfico diferente: con las tablas blendop se puede lograr un efecto suficiente. Por ejemplo, en el código siguiente tenemos un proceso -cuyo gráfico viene dado por el fichero "a.png"-, el cual él mismo genera su propia sombra invocando al proceso "sombra()". Este proceso lo único que hace es copiar una serie de valores de variables locales del padre (mismo graph, mismo angle, mismo size, misma xetc) y alterar levemente otra serie (flags para ver la sombra bocaabajo, z para ver la sombra por debajo del gráfico original, y para verse en la parte inferior...). Lo realmente interesante es que además, al gráfico perteneciente a este proceso se le aplica una tabla blendop que contiene un conjunto de efectos en cadena como son un escalado de grises, un nivel de transparencia medio, un brillo importante y un entintado ténue. El resultado puede variarse a nuestro gusto dependiendo del valor de los parámetros que pongamos a las diferentes funciones blendop.

```

program ejemplo;
begin
set_mode(640,480,16);
proceso();
loop
    frame;
end
end

process proceso()
begin
graph=load_png("a.png");
size=50;
x=320;
y=0;
sombra();
    loop
        frame;
    end
end

//Pensado para ser llamado desde el proceso del que va a ser sombra
process sombra()
private
int sombra;
end
begin
sombra=blendop_new();
blendop_grayscale(sombra,1);
blendop_translucency(sombra,0.5);
blendop_intensity(sombra,0.9);
blendop_tint(sombra, 0.7, 100, 100, 100);
blendop=sombra;

graph=father.graph;

```

```

angle=father.angle;
//size_y=75; //Lo ponemos un poco mas "achaparrado" que el proceso original
size=father.size;
flags=father.flags+2; //para que este boca abajo
z=father.z+1;
x=father.x;
y=father.y+250; //si el grafico tiene su centro virtual en la base no haría falta sumar nada
repeat
  frame;
until(key(_esc))
end

```

Otro efecto interesante parecido al anterior es el de por ejemplo oscurecer una imagen determinada. Para ejecutar el ejemplo siguiente necesitarás dos pngs: "a.png" y "b.png": el primero será un gráfico de fondo que no se alterará y el segundo será el gráfico que pulsando "o" se oscurecerá y pulsando "q" volverá a su apariencia original (ya sabes que se puede alterar el oscurecimiento cambiando los valores de los parámetros de blendop\_tint: haz pruebas)

```

program oscurecer;
global
  int idpng1,idpng2;
  int blen;
end
begin
  set_mode(640,480,16);
  idpng1=load_png("a.png");
  idpng2=load_png("d.png");
  put_screen(0,idpng1);
  blen=blendop_new();
  blendop_tint(blen,0.5,0,0,0);
  proceso(idpng2,320,240);
  repeat
    frame;
  until(key(_esc))
  let_me_alone();
end

process proceso(graph,x,y)
begin
  loop
    if(key(_o)) /*También funciona blendop_assign(0,graph,blen);*/ blendop=blen; end
    if(key(_q)) blendop=0; end
  frame;
end
end

```

El siguiente ejemplo incluye la utilización de tablas blendop para generar sombras, pero además se demuestra el uso de regiones. La idea es que el usuario puede manejar un personaje a derecha e izquierda. Este personaje estará oculto en la noche -es decir, se verá negro-, siempre que no entre en una región definida que representa un muro o pared. En ese momento, el personaje se verá tal cual es -mostrando todos sus colores-, y ahora la sombra se verá reflejada en la "pared" a un tamaño mayor, como si al personaje le estuviera iluminando un gran foco.

```

program cosas_de_sombras_y_luces;
global
    int bl;
end
begin
set_mode(640,480,16);
set_fps(50,0);
define_region(1,380,0,260,395); //La "pared"

drawing_map(0,0);
//Pinto un rectángulo de un color para el fondo de la pantalla (representa la "noche")
drawing_color(rgb(100,150,100));
draw_box(0,0,640,480);
//Pinto un rectángulo de otro color del tamaño de la región 1 para poder visualizarla.
/*El orden con el que se escriben en el código las primitivas es importante: la última primitiva se pintará
por encima de las anteriores.*/
drawing_color(rgb(100,0,100));
draw_box(380,0,640,395);

//Esta tabla blendop será usada para tinter de negro las dos sombras que generará nuestro personaje
bl=blendop_new();
blendop_tint(bl,1,4,4,4);

personaje();
end

process personaje()
begin
graph=dibujopersonaje();
x=100;y=350;
sombra1();
sombra2();
repeat
switch (scan_code)
    case _left: x=x-10; end
    case _right:x=x+10; end
end
frame;
until(key(_esc))
/*Si la tabla blendop se utilizara sólo en un proceso, se podría poner en éste una cláusula ONEXIT con la
liberación de la tabla*/
blendop_free(bl);
exit();
end

process sombra1()
begin
graph=father.graph;
//Aplicamos la tabla blendop a la sombra, con lo que su gráfico aparecerá todo de color casi negro
blendop=bl;
//La sombra aparecerá por encima del personaje, para que cuando sombra1 esté visible, lo tape
z=father.z-1;
loop

```

```

//Aquí es donde se aplica de forma práctica el priority.
x=father.x;
y=father.y;
//Si está dentro de la región 1, la sombra desaparece. Si está fuera, vuelve a aparecer
if(out_region(id,1)==0) alpha=0; else alpha=255; end
frame;
end
end

process sombra2()
begin
size=200;
z=20;
region=1;
blendop=bl;
graph=father.graph;
loop
x=father.x;
y=father.y-50;
/*La transparencia de la sombra debería disminuir a medida que X crece.
alpha=x-380;
frame;
end
end

//Obtengo el dibujo del personaje (primitivas gráficas: círculos y líneas)
function dibujopersonaje()
private
int midibujos;
end
begin
midibujos=new_map(100,100,16);
drawing_map(0,midibujos);
drawing_color(rgb(255,0,0));
draw_line(0,0,100,100);
drawing_color(rgb(0,255,0));
draw_line(0,100,100,0);
drawing_color(rgb(0,0,255));
draw_line(50,0,50,100);
drawing_color(rgb(255,255,0));
draw_line(0,50,100,50);
drawing_color(rgb(0,0,255));
draw_fcircle(50,50,30);
drawing_color(rgb(0,255,0));
draw_fcircle(50,50,20);
drawing_color(rgb(255,0,0));
draw_fcircle(50,50,10);
return midibujos;
end

```

Por último, vamos a ver ejemplos de tres funciones blendop, cuyo uso sea tal vez el más difíciles de comprender: BLENDOP\_IDENTITY, BLENDOP\_SWAP y BLENDOP\_FREE.

Prueba este ejemplo (necesitarás una imagen llamada "dibujo.png"):

```

program hola;
begin
    set_mode(640,480,16);
    fill_map();
    loop if(key(_esc)) exit(); end frame; end
end

Process fill_map()
private
    int idpng;
    int i;
    int ble;
end
begin
idpng=load_png("dibujo.png");
blur(0,idpng,0); //Un efecto extra
ble=blendop_new();
graph=idpng;x=320;y=240;
loop
    blendop_identity(ble);
    blendop_tint(ble,0.1,rand(0,255),rand(0,255),rand(0,255));
    blendop_assign(0,idpng,ble);
    frame(1000);
end
end

```

Verás que lo que ocurre es que la imagen del proceso va cambiando su tinte de forma consecutiva e infinita. ¿Por qué?.

En el loop del proceso fill\_map(), una vez ya creada antes la tabla blendop, lo primero que hago es aplicar un identity. En la primera iteración no tiene sentido "limpiar" la tabla blendop porque de entrada ya se crea "limpia", pero ahora veremos que en sucesivas iteraciones esta es necesaria. Seguidamente, aplico un tinte a la tabla blendop, y asigno ésta al gráfico en memoria que de hecho corresponde al gráfico del proceso. Justo después viene el fotograma, con lo que se verá el gráfico tinteado. A la siguiente iteración es cuando la función identity toma todo su sentido, porque limpia el tinte previo de la tabla blendop y la deja tal como si estuviera acabada de crear. Una vez hecho esto, vuelvo a aplicar otro tinte sobre la tabla "virgen", lo vuelvo a aplicar al gráfico y lo vuelvo a mostrar.

¿Qué pasaría si no estuviera la línea con la función identity? Que a cada iteración se iría añadiendo un tinte sobre otro en la tabla blendop. Y el efecto resultante sería que en unos pocos frames, el gráfico se vería de un color marronizo homogéneo, ya que a cada tinte que se aplica se va homogeneizando los distintos colores de la imagen: si se aplican muchos tintes seguidos se llega a la homogeneización total. Pruébalo.

Otro ejemplo diferente para mostrar la utilidad de esta función es el siguiente: en él se mostrará el mismo dibujo "dibujo.png" sobre el que se ha asignado una tabla blendop formada por un tinte de color aleatorio. Cuando el usuario pulse ENTER, se mostrará el gráfico sin tinte ya que se "limpiará" la tabla. Pasados tres segundos, se volverá a tinteado la tabla blendop y el gráfico se volverá a ver afectado por ese cambio de tinte.

Fíjate en el detalle de haber usado map\_clone para poder aplicar la tabla blendop a un gráfico que no es el gráfico originalmente cargado desde el archivo. Esto se suele hacer para preservar dicho gráfico de

posibles modificaciones y mantenerlo siempre tal cual, haciendo que cualquier modificación recaiga sobre clones suyos.

```
program Test_BLENDOP_IDENTITY;
global
  int bldop;
  int idpng;
  int bid;
end
begin
  set_mode(320,200,16);
  idpng=load_png("dibujo.png");
  write(0,160,70,4,"ENTER = IDENTITY Tabla BLENDOP");
  graph=map_clone(0,idpng);x=160;y=100;
  bldop=blendop_new();
  blendop_tint(bldop,0.55,0,255,0);
  blendop_assign(0,graph,bldop);
  repeat
    if(key(_enter) && !bid)
      blendop_identity(bldop);
      bid=1;
      timer=0;
    end
    if(timer>=300 && bid)
      blendop_tint(bldop,0.50,rand(0,255),rand(0,255),rand(0,255));
      bid=0;
    end
  frame;
  until(key(_esc))
end
```

Para ver el efecto que causa la función `blendop_swap()` sobre un gráfico, veremos el siguiente código, en el cual necesitarás un gráfico llamado "dibujo.png", y además, otro gráfico llamado "fondo.png" el cual será el fondo de pantalla y sería conveniente que fuera variado (que no sea un color homogéneo, que tenga elementos distintivos) para poder apreciar el efecto mejor. He aquí el ejemplo: pulsando enter aplicaremos y desaplicaremos alternativamente el efecto swap a la tabla blendop, y en consecuencia, a su gráfico asociado.

```
program Test_BLENDOP_SWAP;
global
  int bldop;
  int idpng;
  int bswap;
  int fondo;
end
begin
  set_mode(320,240,16);
  put_screen(0,load_png("fondo.png"));

  idpng=load_png("dibujo.png");
  graph=map_clone(0,idpng); x=160; y=100;
  bldop=blendop_new();
  blendop_tint(bldop,0.55,0,255,0);
  blendop_assign(0,graph,bldop);
```

```

repeat
  if(key(_enter) && bswap==0)
    while(key(_enter)) frame; end
    blendop_swap(bldop);
    bswap=1;
  end
  if(key(_enter) && bswap==1)
    while(key(_enter)) frame; end
    blendop_swap(bldop);
    bswap=0;
  end
frame;
until(key(_esc))
end

```

El siguiente código es un ejemplo de uso de la función `blendop_free()`. Si lo ejecutas, aparentemente funciona muy similar al ejemplo que hemos visto antes con la función `blendop_identity()`, pero no te engañes: en aquel ejemplo no destruíamos la tabla `blendop`, simplemente la limpiábamos cada vez; en cambio, en este ejemplo, cada vez que pulsamos ENTER estamos destruyendo completamente de la memoria o generando la tabla de nuevo completamente desde cero.

```

program Test_BLENDOP_FREE;
global
  int bldop;
  int idpng;
  int bfree;
end
begin
  set_mode(320,200,16);
  idpng=load_png("dibujo.png");
  write(0,160,70,4,"ENTER: Borra/Crea la Tabla BLENDOP:");
  graph=map_clone(0,idpng); x=160; y=100;
  bldop=blendop_new();
  blendop_tint(bldop,0.55,0,255,0);
  blendop_assign(0,graph,bldop);
  repeat
    if(key(_enter) && bfree==0)
      while(key(_enter)) frame;end
    /*Línea importante: retiro la asignación que tenía el gráfico con la tabla blendop, antes de destruir ésta.
    Si no hago esto, el blendop_free dará error porque para destruir una tabla blendop antes se han de
    desasociar
    de cualquier gráfico o proceso ligado a ella.*/
    blendop_assign(0,graph,0);
    blendop_free(bldop);
    bfree=1;
  end
  if(key(_enter) && bfree==1)
    while(key(_enter)) frame;end
  /*La tabla bldop ahora mismo no existe. La tengo que volver a crear,configurar el tinte y asignárselo al
  gráfico, como al principio del programa*/
  bldop=blendop_new();
  blendop_tint(bldop,0.55,0,255,0);
  blendop_assign(0,graph,bldop);
  bfree=0;

```

```

end
frame;
until(key(_esc))
end

```

**Utilizar los operadores binarios:**

A parte de los operadores aritméticos ya conocidos (+,-,\*,/,%), Fénix dispone de una serie de operadores avanzados para trabajar con números conocidos genéricamente como operadores a nivel de bit. Son seis: BAND (&), BOR (|) ,BXOR (^) , BNOT (~), desplazamiento a la derecha (>>) y desplazamiento a la izquierda (<<).

Se llaman así, "a nivel de bit" porque actúan bit a bit. Es decir, trabajan con la representación binaria de los números y operan con cada bit que la forman, siguiendo unas determinadas reglas. Lógicamente, estas reglas sólo están definidas para realizar operaciones sólo con 1 y con 0. Por ejemplo, una operación como ésta: 34 BAND 7 en realidad no tiene mucho sentido: lo más lógico sería escribir esta operación en representación binaria:

```

      100010
BAND  111
-----
      ??????

```

Las reglas de las seis operaciones son las siguientes:	
<b>BAND</b> (Ambos operandos deben ser 1 para que el resultado sea 1)  $0 \& 0 = 0$ $0 \& 1 = 0$ $1 \& 0 = 0$ $1 \& 1 = 1$	<b>BOR</b> (Si cualquiera de los dos operandos es 1, el resultado es 1)  $0   0 = 0$ $0   1 = 1$ $1   0 = 1$ $1   1 = 1$
<b>BXOR</b> (BOR exclusivo: si un elemento o el otro, nunca ambos juntos, son 1, el resultado es 1)  $0 \wedge 0 = 0$ $0 \wedge 1 = 1$ $1 \wedge 0 = 1$ $1 \wedge 1 = 0$	<b>BNOT</b> (Negado)  $\sim 0 = 1$ $\sim 1 = 0$
<b>Desplazamiento a la derecha</b> (Mueve hacia la derecha los bits que forman el primer operando un número dado por el segundo operando)  Por ejemplo, $01000101 \gg 3$ ( $69 \gg 3$ ) moverá el número 01000101 tres posiciones hacia la derecha, (rellenando los huecos que aparezcan por la	<b>Desplazamiento a la izquierda</b> (Mueve hacia la izquierda los bits que forman el primer operando un número dado por el segundo operando)  Por ejemplo, $01000101 \ll 3$ ( $69 \ll 3$ ) moverá el número 01000101 tres posiciones hacia la izquierda, (rellenando los huecos que aparezcan por la derecha

izquierda con 0 y destruyendo el valor de los bits que sobrepasan el extremo derecho), resultando el número 00001000. O sea, el 8.

con 0 y destruyendo el valor de los bits que sobrepasan el extremo izquierdo, el cual en el caso de los INT está en 32 posiciones, en el caso de los BYTE está en 8, etc), resultando el número 01000101000. O sea, el 552.

Así pues, sabiendo esto, la operación de antes,  $34 \& 7$  da como resultado 2 (10 en binario).

Podemos comprobar cómo funcionan efectivamente estos operadores, con un sencillito programa:

```
program prueba;
begin
    set_mode(640,480,16);
    write(0,100,100,4,1 band 2);
    write(0,100,110,4,1 bor 2);
    write(0,100,120,4,1 bxor 2);
    write(0,100,130,4,~1);
    write(0,100,130,4,69>>3);
    write(0,100,140,4,69<<3);

    while(!key(_esc))
        frame;
end
```

Obtenemos como resultado 0,3,3,-2,8,552. ¿Tiene sentido? Los dos últimos valores ya sabemos que sí. Veamos los cuatro primeros:

```
1 = 0001
& 2 = 0010
-----
0000, o sea, 0
```

```
1 = 0001
| 2 = 0010
-----
0011, o sea, 3
```

```
1 = 0001
^ 2 = 0010
-----
0011, o sea, 3
```

$\sim 1 = \sim \dots 0001 = \dots 1110$ . Al cambiar el bit de más a la izquierda por un 1, se entra en el terreno de los números negativos...

Un uso muy bueno para los operadores a nivel de bits es para hacer flags: supongamos que tenemos que controlar que un protagonista tiene o no un determinado objeto. Podríamos hacer un array de... cinco elementos de tipo byte, por ejemplo, que valiera cada uno 1 si tiene el objeto o 0 si no lo tiene. Sabemos que cada byte tiene 256 valores posibles, usando sólo dos desperdiciamos 254 posibilidades. Mala solución.

Pero ya sabes que cada byte se compone de 8 bits, y éstos pueden tener los dos valores que precisamente necesitamos. Así, en un solo byte podemos obtener información de hasta ocho valores binarios.

Por ejemplo, si a un byte le hacemos un BAND con el valor 00000001, podemos descartar los bits que no nos interesan y conocer el valor del único bit que queremos saber. Es decir: sea un byte xxxxxxxx y le hacemos un BAND con ese valor 00000001 (que llamaremos máscara), los primeros 7 bits siempre darán cero, valga lo que valga nuestro primer dato, pero el último valdrá cero si el último bit es un cero, o uno si el último bit es un uno. De esta forma, si usando nuestra máscara nos esta operación nos da un valor cero, es que el protagonista no tiene el objeto, y si nos da un uno, sí lo tiene.

### Opciones de depuración:

Los debuggers (o depuradores) son un software específico utilizado por los programadores para buscar errores de varios tipos (léxicos, sintácticos, lógicos...) dentro de un código fuente. Son capaces de ejecutar el programa paso a paso, ejecutar sólo unas determinadas líneas repetidamente, inspeccionar variables, etc, en busca de dichos errores.

Depurar un programa sirve, pues, para detectar errores en tiempo de ejecución. Depurar un programa es como ejecutarlo normal pero con la posibilidad de parar su ejecución en cualquier momento que deseemos. Es entonces, al haber parado el juego, cuando tenemos acceso a los valores que en ese preciso momento tienen las diferentes variables del programa, información vital para detectar el posible error, ya que podremos observar en tiempo real cuándo dejan de valer aquello que se supone que deberían de valer, y poderlo corregir.

Para poder realizar este proceso de depuración, lo que hay que hacer, primero de todo, es compilar nuestro código fuente de una determinada manera. Hay que utilizar FXC.EXE con el parámetro “-g”, de esta manera:

#### **FXC -g mijuego.prg**

Con esta opción, FXC almacenará en el archivo DCB información adicional innecesaria para ejecutarlo, como los nombres de las variables o la posición de las instrucciones en los ficheros de los códigos fuente, pero que es imprescindible para poder depurar cuando ejecutemos el programa.

Si utilizar el FlameBird, podrás hacer lo mismo si vas al menú “Edit”->”Preferences” y en la pestaña “Compilation” dejas marcada la opción “Compile in debug mode”.

Para depurar un programa, lo único que hay que hacer es ejecutarlo como siempre con FXI. En el momento que quieras, pulsa **ALT+C**. En ese momento, el juego se detendrá por completo (estará "congelado") y aparecerá una ventana que representa la consola desde la que podrás utilizar diferentes comandos especiales que te servirán de ayuda en la búsqueda del error díscolo: existen comandos para comprobar los valores que tienen las variables en ese mismo instante, también para mandar señales concretas a procesos concretos y observar lo que ocurre, etc, etc, ahora lo iremos viendo. Ojo, no pulses ALT+C mientras se está ejecutando un juego que no ha sido compilado con la opción -g, ya que lo terminarás fulminantemente.

Fíjate que nada más abrir la consola de depuración, ya obtienes algunos datos interesantes, como la versión de Fénix sobre la que está ejecutándose el programa, y la versión de la librería SDL en la que está basado esa versión de Fénix.

Para salir de la consola y continuar con el juego, basta con volver a pulsar **ALT+C**.

Una vez ya dentro de la consola, podemos observar qué valores en el momento de la "congelación" tienen las variables que deseemos. Para ello disponemos de una serie de comandos de consola que nos permitirán obtener esta información.

GLOBALS	Muestra un listado del tipo y nombre de todas las variables globales del programa (incluyendo las variables globales predefinidas de Fénix), junto con sus valores actuales
LOCALS nombreProceso	Muestra un listado del tipo y nombre de todas las variables locales del programa (incluyendo las variables locales predefinidas de Fénix), junto con sus valores por defecto (esto último si se especifica el nombre de un proceso, y siempre que exista como mínimo una instancia de éste funcionando -activa, dormida o congelada- en este momento).
PUBLICS nombreProceso	Muestra un listado del tipo y nombre de todas las variables públicas del proceso especificado, junto con sus valores actuales, siempre que exista como mínimo una instancia de ese proceso funcionando (activa, dormida o congelada) en este momento.
PRIVATEs nombreProceso	Muestra un listado del tipo y nombre de todas las variables privadas del proceso especificado, junto con sus valores actuales, siempre que exista como mínimo una instancia de ese proceso funcionando (activa, dormida o congelada) en este momento.
nombreProceso.nombreVariable	Muestra el valor de la variable privada/pública/local especificada para un proceso dado. El programa principal es considerado como un proceso más, llamado MAIN. No es posible mostrar el valor de una variable global concreta: para ello hay que recurrir al comando "globals" general.

Vamos a practicar esto último. Para ello haremos servir un código bastante simple como el siguiente. Lo compilamos en modo depuración, lo ejecutamos y antes de que "miproceso" acabe de escribir los números en pantalla, pulsamos ALT+C:

```

program hola;
global
    int miglobal;
end
process main()
begin
    set_mode(800,600,16);
    set_fps(74,1);
    miproceso();
    miglobal=65; //Cambio el valor inicial por defecto por otro
loop
    frame;
end
end

process miproceso()
private
    int miprivada;
end
begin

```

```
//A pesar de cambiar el valor de esta variable local, LOCALS muestra el valor por defecto (0) siempre
x=400;
while(i<40)
    write(0,100,10+5*i,4,i);
    i++;
    frame;
end
end
```

Puedes probar de observar los valores actuales de las variables globales del programa (ya sean las predefinidas o bien nuestra "miglobal", que aparece al final del listado) con el comando 'GLOBALS'. También puedes observar los valores actuales de las variables privadas de "miproseso" con 'PRIVATES miproseso'. O más específicamente, puedes obtener el valor de "miprivada" de esta manera 'miproseso.miprivada'.

También tienes la posibilidad de cambiar dentro de la consola los valores actuales de la variable privada/pública/local que desees, para el proceso que quieras, de esta manera:

```
nombreProceso.nombreVariable = valor
```

Es importante notar en ambos casos el espacio en blanco existente antes y después del símbolo =. Los valores numéricos pueden ser reconocidos como hexadecimales si su último dígito es el carácter "h", binario si su último dígito es el carácter "b" o decimal (sin ningún carácter final específico). Los valores de cadena se pueden especificar tanto entre comillas simples como dobles. No se pueden asignar, no obstante, nuevos valores a variables globales.

Prueba por ejemplo, mientras tienes la consola en marcha, de asignar un nuevo valor a la variable "miprivada" de "miproseso" (es decir, escribe 'miproseso.miprivada = 30', por ejemplo. Verás que, si ahora cierras la consola y continúa la ejecución del programa -pulsas ALT+C otra vez, recuerda-, se producirá un salto en los números visualizados en pantalla, ya que se ha alterado el valor que iba teniendo la variable "miprivada" dentro del bucle.

Existen otro conjunto de comandos de la consola de depuración relacionados con la manipulación de procesos:

INSTANCES	Lista los identificadores de proceso de todas las instancias existentes en el instante actual (ya estén activas, dormidas -con la marca "[S]"- o congeladas -con la marca "[F]"-), además de su correspondiente nombre de proceso.
RUN nombreProceso [params opcionales]	Ejecuta una nueva instancia del proceso especificado
KILL identificadorProceso	Elimina una instancia concreta del proceso especificado
WAKEUP identificadorProceso	Despierta una instancia concreta del proceso especificado
SLEEP identificadorProceso	Duerme una instancia concreta del proceso especificado
FREEZE identificadorProceso	Congela una instancia concreta del proceso especificado
KILLALL nombreProceso	Elimina todas las instancias del proceso especificado
WAKEUPALL nombreProceso	Despierta todas las instancias del proceso especificado
SLEEPALL nombreProceso	Duerme todas las instancias del proceso especificado
FREEZEALL nombreProceso	Congela todas las instancias del proceso especificado

Prueba de abrir la consola una vez "proceso1" haya acabado su ejecución (es decir, cuando hayan acabado de imprimirse los números en pantalla). En ese momento, escribe el comando 'INSTANCES'. Verás que sólo está funcionando el programa principal (proceso "main") con un determinado código identificador. Si ahora ejecutas 'RUN miproceso', lo que estarás haciendo es que cuando salgas de la consola de depuración, se ejecute otra vez desde el principio el código del "miproceso". Es posible que no te des cuenta que se está volviendo a ejecutar ya que imprimirá los números encima de los ya existentes en la pantalla, pero esto lo puedes cambiar fácilmente por ejemplo haciendo que el segundo parámetro del write sea un valor aleatorio y no 100: de esta manera, en cada ejecución de "miproceso" los números aparecerán en posiciones horizontales diferentes: es una idea. De todas maneras, siempre podrás volver a la consola en esta nueva ejecución de "miproceso" y comprobar que efectivamente, está funcionando si consultas otra vez lo que nos devuelve 'INSTANCES'.

De igual manera, puedes utilizar los demás comandos que envían señales a procesos concretos o a familias enteras, para que, al salir de la consola, éstas sean efectivas y se pueda comprobar su efecto en ese instante concreto. Por ejemplo, si añades justo antes del bucle LOOP/END del proceso principal una nueva línea que sea "miproceso();", estarás creando dos instancias de este proceso. Si ejecutas el programa y entras en la consola, haciendo 'INSTANCES' verás que aparecen efectivamente dos instancias de "miproceso" con diferentes identificadores. Si ahora escribes por ejemplo 'SLEEP idProceso', cuando salgas de la consola habrás dormido la instancia que hayas decidido (y por tanto, esa instancia no seguirá imprimiendo números en pantalla: lo podrás ver mejor si usas una coordenada horizontal aleatoria para el write). Si cuando ya se han acabado de imprimir los números correspondientes a la instancia que no se ha dormido, y vuelves a entrar en la consola otra vez, podrás ver mediante 'INSTANCES' que efectivamente, existe todavía una instancia de "miproceso" dormida, y podrás despertarla usando 'WAKEUP idProceso'. Cuando salgas de la consola de nuevo, esa instancia reemprenderá su ejecución y acabará de mostrar los números en pantalla que le quedaban.

Otro ejemplo sencillo: si mientras están ejecutándose ambas instancias de "miproceso", entras en la consola y escribes 'KILLALL miproceso', cuando salgas de ella se dejarán de escribir los números en la pantalla, ya que todas las instancias de "miproceso" habrán muerto.

Supongo que estarás pensando que no es una buena manera entrar en la consola durante la ejecución del juego pulsando ALT+C cuando se nos ocurra, porque la mayoría de las veces nos interesará entrar en la consola en un fotograma preciso del programa, para estudiar en ese instante concretísimo el estado de las variables y de los procesos. Con ALT+C no sabemos en realidad en qué momento preciso de la ejecución estamos, ya que depende de cuándo hayamos pulsado estas teclas estaremos en una línea o en otra, hecho totalmente inaceptable si queremos realizar una depuración exhaustiva y precisa.

La manera de entrar en la consola de depuración llegado un punto concreto de nuestro programa es mediante el uso del comando DEBUG; (sin paréntesis) de Fénix. Escribiendo este comando en cualquier sitio de nuestro código, la consola de depuración puede ser invocada inmediatamente (sin esperar a ningún frame) cuando se llegue a ese sitio. A partir de entonces, tendremos disponible la consola justo en el momento que la necesitamos.

Puedes probar de insertar la orden DEBUG; en cualquier punto del código del ejemplo, para comprobar su efecto. Por ejemplo, si escribimos lo siguiente dentro del bucle WHILE/END de "miproceso":

```
if(miprivada==14) debug; end
```

observa lo que pasa. Se abre automáticamente la consola en el momento que nosotros hemos dicho en el código (de hecho, sale impresa la línea que la ha invocado dentro de éste). Puedes comprobar que esto es así si haces por ejemplo un 'PRIVATEs miproces' y compruebas que efectivamente "miprivada" vale 14.

Una vez que has entrado en la consola de debug, es posible que quieras ejecutar el código fuente línea a línea individualmente, a partir del punto del juego en el que éste se ha quedado congelado, para ir, por ejemplo, rastreando a cada línea los valores que van adquiriendo determinadas variables.

Para lograr esto, dentro de la consola dispones del comando TRACE. Este comando, cada vez que se escriba, ejecutará una única línea del proceso donde actualmente se ha parado el programa. (mostrando dicha línea en la consola, para saber cuál es), y volverá el control a la consola, para poder seguir utilizándola en este nuevo punto del programa.

Recuerda que la ejecución de los códigos de los procesos es consecutiva, así que si se utiliza varias veces seguidas esta orden, cuando se acabe de ejecutar las líneas de un proceso, se continuará por las líneas del siguiente.

Otro comando interesante es DOFRAME, el cual continúa la ejecución de todos los procesos hasta el siguiente frame, y retorna a la consola de debug, con lo que podremos observar de nuevo el estado de nuestro programa después del pase de un fotograma completo en todos los procesos.

Otro aspecto que se puede manipular con la consola de debug es la gestión de breakpoints. Un breakpoint es una línea determinada del código fuente en donde la ejecución se para cada vez que se llega ahí, para poder examinar en ese punto el estado del programa (variables, procesos, etc). Éste es un tema relativamente avanzada que no trataremos en este manual

BREAK	Lista los breakpoints definidos.
BREAK nombreProceso	Agrega un breakpoint en la ejecución del proceso especificado (se pueden también especificar funciones e instancias)
DELETE nombreProceso	Elimina el breakpoint creado con "BREAK nombreProceso". Los breakpoints sobre una instancia determinada se eliminan cuando ésta muere

Otros comandos/variables interesantes de la consola de depuración son:

CONTINUE	Sale de la consola de depuración y continúa la ejecución del programa (similar a ALT+C)
QUIT	Sale de la consola de depuración y termina la ejecución del programa
DEBUG	Variable que igualada a 0 desactiva el log del debug e igualada a 1 lo activa. El log del debug contiene información de bajo nivel que se almacena (si así lo marca el valor de esta variable) en el archivo "stdout.txt".
FILES	Muestra la cantidad de archivos abiertos (en uso)
STRINGS	Muestra todas las cadenas existentes en memoria
CONSOLE_LINES	Variable cuyo valor representa el número de líneas que mostrará la consola de debug (útil para agrandar el tamaño de ésta y poder así mostrar más información)

CONSOLE_COLUMNS	Variable cuyo valor representa el número de caracteres por línea que mostrará la consola de debug (útil para agrandar el tamaño de ésta y poder así mostrar más información)
SHOW_COLOR	Variable cuyo valor permitido es cualquier color RGB
VAR	Muestra un listado con el nombre y valor de las variables internas de la consola (DEBUG, CONSOLE_LINES,etc)
SHOW expresion	Evalúa una expresión aritmética y devuelve el resultado

Para más información, utiliza el comando HELP de la consola de depuración

Si así lo deseamos para estar más cómodos, se puede deslizar el contenido de la ventana de debug a los costados, con CTRL+<Flechas de direccion>.

Para avanzar el contenido de dicha ventana se puede utilizar la tecla "AvPag", y para retroceder, la tecla "RePag".

Antes hemos comentado que desde nuestro código fuente en Fénix podíamos invocar a la consola de debug haciendo uso del comando de Fénix DEBUG;. Fénix tiene además otro comando relacionado con la depuración, que es el comando SAY.

Con el comando SAY("texto") podemos mostrar un texto determinado por la consola de depuración, el cual es el único parámetro que tiene. Este texto también se escribirá además en el archivo "stdout.txt".

Puedes probarla si lo escribes en cualquier lugar del código fuente de ejemplo, siempre antes de cualquier llamada a la consola (ya sea mediante el comando DEBUG; o por ALT+C). Puedes escribir el número de SAY que quieras: todos ellos mostrarán su mensaje a la vez en el momento que la consola aparezca visible ( y en el archivo "stdout.txt").

La principal utilidad de esta función es escribir "chivatos", pequeñas inserciones de código que muestran en momentos clave el estado de variables importantes, o simplemente informan acerca de la ejecución de un trozo de código. Esto permite obtener información útil sobre el desarrollo del programa, de cara a detectar la causa de bugs molestos.

Además de todo lo hablado hasta ahora, si has compilado en modo debug, mientras el juego se está ejecutando (es decir, en tiempo de ejecución) puedes hacer cosas curiosas como a cambiar a pantalla completa con **ALT+F** o a modo ventana de nuevo con **ALT+W** ; sacar una captura de la pantalla con **ALT+G** (la captura se guardará en la misma carpeta desde donde se está ejecutando el programa, y su nombre será "shotXXXX.png", donde las X indican un número consecutivo por cada captura que se haga, empezando por el 0000); o también salir del programa en cualquier momento con **ALT+X** (aunque esto es una salida forzada y puede dejar residuos en la RAM).

### **Introducir parámetros a nuestro programa por línea de comandos:**

Ya sabes que podemos ejecutar nuestros programas invocando al intérprete desde la consola cmd.exe de Windows, así: *fxi.exe miPrograma.dcb*. Lo que tal vez no sepas es que, si pones en marcha tus programas de esta manera, mediante el intérprete de comandos, dispones de una posibilidad bastante interesante: pasar parámetros a tu programa en el momento de iniciar su ejecución (como si fueran un comando estándar más de Windows). Estos parámetros iniciales pasados por consola podrán ser recogidos por el programa y usados en consecuencia.

Si queremos hacer uso de esta posibilidad, para invocar nuestros programas tendremos que escribir algo por ejemplo como: *fxi miPrograma.dcb ValorParam1 ValorParam2*, donde en este caso habríamos pasado dos valores "iniciales" que serían recogidos en nuestro programa para poder trabajar con ellos. Fíjate que los diferentes valores se separan por espacios en blanco.

Para poder recoger y utilizar estos valores dentro de nuestro código, Fénix dispone de dos variables globales predefinidas:

**ARGC** : Esta variable entera contiene el número total de parámetros recibido por el programa, incluido el propio nombre del programa.

**ARGV**: Este vector contiene el valor de los parámetros recibido por el programa, incluido el propio nombre del programa, que se corresponde a `argv[0]`. Fénix guarda internamente espacio suficiente para 32 parámetros, pero el número real de parámetros recibidos se obtiene a partir de **ARGC**.

Los parámetros pueden ser enteros, decimales, cadenas...Deberá ser el programador el que al recoger los datos asigne convenientemente el tipo de datos pertinente a cada parámetro recogido.

Veamos un ejemplo de uso.

```
Program laos;
private
  int i;
end
Begin
set_mode(320,240,16);
write(0,300,200,4,argc);
if (argc > 0) //Si se han pasado parámetros a nuestro programa desde la consola...
  for(i=0;i<=argc;i++)
    write(0,100,10*i+10,4,argv[i]); //Escribo los valores pasados uno detrás de otro
  end
else //Si no...
  write(0,100,100,4,"¡No me has pasado nada!");
end
Loop
  if(key(_esc)) exit();end
Frame;
End
End
```

Llamaremos al archivo que contiene el código "pepe.prg". Para probar este ejemplo, deberás ejecutarlo desde la consola de Windows. Si escribes por ejemplo simplemente *fxi.exe pepe.dcb* verás lo que ocurre. ¿Y qué pasa si escribes *fxi.exe pepe.dcb 1 hola que tal 354.6 -4.7* ?

También podrías utilizar **ARGC** y **ARGV** sin necesidad de emplear la consola de comandos. Si seleccionas tu programa DCB dentro del explorador de Windows y seleccionas a más a más otros archivos (con la tecla **MAYUS**), y arrastras el DCB al archivo "fxi.exe", el vector **ARGV** de llenará de las rutas absolutas de los archivos seleccionados. Lo puedes comprobar con el código anterior.

## Cómo desarrollar un formato propio de imagen, de uso exclusivo en nuestros programas:

A continuación se presenta un código fuente que, aún no siendo excesivamente útil en el día a día, no deja de ser muy interesante. En él se hace uso de dos funciones, "imgtxt()" y "txt2img()" el uso de las cuales se puede trasladar a cualquier otro programa sin problemas.

La primera función se encarga de, a partir de una imagen especificada como parámetro (incluida o no dentro de un FPG), codificar su información gráfica -básicamente, el color de cada pixel- en formato texto, y grabar éste en un archivo de texto especificado también como parámetro. Es decir, "transforma" una imagen en texto. La segunda función hace el proceso inverso: a partir de un archivo de texto que contiene la información generada anteriormente por "img2txt()", reconstruye la imagen tal como era la original. Es decir, "transforma" un texto en imagen.

Una posible utilidad de este código sería ocultar los recursos gráficos usados en nuestro juego de personas indeseadas. Utilizando estas funciones propias, las imágenes podrán distribuirse con (casi) total confianza por todo el mundo en un simple formato de texto sin temor a posibles "hurtos", ya que sólo aquellos que utilicen la función correcta de decodificación (es decir, "txt2img()") podrán reconstruir la imagen y la podrán visualizar. De hecho, esta idea (aunque mejorada) es utilizada hoy en día por los juegos profesionales.

Una mejora evidente del código presentado es utilizar una codificación binaria en vez de textual, ya que el texto ocupa muchísimo espacio, con lo que unas pocas fotografías pueden aumentar de manera considerable el peso de nuestro juego, pero si entramos en el terreno de las codificaciones binarias, de hecho estaremos entrando en la creación de un nuevo formato de imagen, tema bastante complejo en sí.

El programa lo que hace es, de entrada, mostrar como fondo una imagen (en este caso "a.png"), y dos opciones. Si se pulsa "i", se generará en el directorio actual un archivo de texto "mapa.txt" que contendrá la codificación de la imagen. Si se pulsa "c" (¡hacerlo siempre después de "i"!), se recogerá la información guardada de "mapa.txt", se borrará el fondo de pantalla que había anteriormente y se mostrará como nuevo fondo la imagen generada a partir de la información obtenida de "mapa.txt", para comprobar que la nueva imagen es idéntica a la original.

```
//A partir de una idea original de Phreak
program hola;
process main()
private
    int fpg, idimagen;
    string archtexto;
    int fondoimportado; /*Recogerá el identificador devuelto por la función txt2img(), el cual
representa una imagen que se visualizará como nuevo fondo de pantalla*/
    int idtexto;
end
begin
    set_mode(320,240,16);
    set_fps(0,0); //Para ir lo más rápido posible
    /*Estaría bien que los siguientes tres valores se pasaran como parámetros por la línea de comandos,
usando las variables globales predefinidas ARGV y ARGC*/
    archtexto=".\\mapa.txt";
    fpg=0;
    idimagen=load_png("a.png");

    write(0,150,20,4,"1º.-Pulse 'i' para codificar la imagen a texto");
    write(0,150,40,4,"2º.-Pulse 'c' para decodificar el texto a imagen");
    put_screen(fpg,idimagen);
    loop
```

```

    if (key(_i))
        //Aviso que la función está en marcha.
        idtexto=write(0,150,100,4,"Procesando...");frame;
        img2txt(archtexto,fpg,idimagen);
        //Aviso de que ya acabó
        delete_text(idtexto);
    end
    if (key(_c))
        //Borro la imagen mostrada anteriormente para poner la cargada a partir del archivo
        clear_screen();
        idtexto=write(0,150,100,4,"Procesando...");frame;
        /*El identificador de la imagen generada por la función cuyo contenido seha rellenado a partir de los
        datos almacenados en el archivo de texto se retorna y se asigna a "fondoimportado". A partir de aquí, se
        pueden hacer muchas cosas: en este ejemplo se utiliza "fondoimportado" para ponerlo de nuevo fondo de
        pantalla, y así comprobar que es la misma imagen que había al principio.*/
        fondoimportado=txt2img(archtexto);
        delete_text(idtexto);
        put_screen(0,fondoimportado);
    end
    if(key(_esc)) exit(); end
    frame;
end

end

function img2txt(string archtexto,int fpg, int idimagen)
private
    int salida;

//Los campos de esta estructura son los que se van a guardar en el archivo
    struct infimg
        int x,y; //Posición de un pixel
        int color; //Su color RGB
    end

//Variables auxiliares
    int i,j;

    int ancho,alto; //De la imagen
end

BEGIN
//Creo el archivo de texto (o lo sobrescribo)
salida=fopen(archtexto,O_write);
//Obtengo el ancho y alto de la imagen, para recorrer los bucles anidados convenientemente...
ancho=graphic_info(fpg,idimagen,g_width);
alto=graphic_info(fpg,idimagen,g_height);

//...y escribirlo al comienzo del archivo de texto, como información útil a la hora de leerlo posteriormente
fwrite(salida,ancho);    fwrite(salida,alto);

//Recorro pixel a pixel la imagen, en sus dos dimensiones
from i = 1 to ancho
    from j = 1 to alto

```

```

/*Asigno los valores que me interesan a los campos correspondientes de la estructura que se escribirá en
el archivo*/
        infimg.x=i;infimg.y=j;
/*Aquí obtengo el color de cada pixel, codificado en un único número mezcla de las tres componentes
RGB.Esto tiene un problema, y es que como la codificación única RGB depende de la tarjeta gráfica de la
máquina, la información de color guardada en el archivo es posible que en otro ordenador diferente se
visualice cromáticamente de forma diferente. Para evitar esto, se podría guardar en la estructura en vez
de un único número, el valor de las tres componentes RGB por separado. Para ello, en vez de tener un
campo INT color, tendríamos que tener tres campos BYTE llamados por ejemplo r,g y b. Y a la hora de
obtener el color de la imagen actual, obtener sus componentes con ayuda de la función get_rgb, de la
siguiente manera get_rgb(color,&(infimg.r),&(infimg.g),&(infimg.b));*/
        infimg.color=map_get_pixel(fp,imagen,i,j);
//Y lo hago: escribo esos valores en el archivo de texto
        fwrite(salida,infimg);

        end
end
fclose(salida);
end

function txt2img(string archtexto)
private
    int entrada;

/*Esta estructura tiene los mismo campos (evidentemente) que la que se usó para escribir el archivo*/
    struct infimg
        int x,y;
        int color;
    end

/*Imagen donde se volcará la información del fichero de texto,y que será mostrada como fondo de
pantalla en el programa principal*/
    int newimagen;

    int ancho,alto; //De la imagen

end
BEGIN

Entrada=fopen(archtexto,O_read);

/*Primero leo el ancho y alto de la imagen, que son los dos datos INT que se guardaron al principio del
archivo de texto cuando se generó.Estos datos los necesito para crear la nueva imagen con new_map con
el tamaño igual al de la imagen original.*/
fread(entrada,ancho);fread(entrada,alto);

/*Genero una imagen vacía, que se rellenará del contenido gráfico obtenido a partir del archivo de texto*/
newimagen=new_map(ancho,alto,16);

while(!feof(entrada))
//Leo los datos de un pixel
        fread(entrada,infimg);
/*Y los traduzco gráficamente: su posición y su color. Relleno de contenido pixel a pixel la imagen
generada vacía anteriormente.Luego la mostro como fondo de pantalla (esto último se podría

```

```
cambiar...)*/  
    map_put_pixel(0,newimagen,infimg.x,infimg.y,infimg.color);  
end  
fclose(entrada);  
//Devuelvo el identificador de la imagen generada a partir del archivo de texto  
return (newimagen);  
end
```

### **Cómo controlar muchos individuos:**

(extraído del tutorial de Wakroo, en <http://divnet.divsite.net>)

En este apartado comentaremos una manera (hay miles) de solucionar el asunto de controlar distintos individuos de manera general, sin recurrir a complicados mecanismos de inteligencia artificial.

Se puede hacer que cada proceso se encargue de detectar si tiene que hacer algo o no, pero puede resultar un poco complicado, especialmente si hay varias maneras de darle órdenes a cada individuo. Si tenemos, además, distintos tipos de individuos con su propio proceso tendremos que repetir el código en cada proceso, con todo lo que esto conlleva: posibles errores al duplicar el código, tener que cambiar todos los procesos afectados si se hace algún cambio, código más largo y menos legible,...

¿Y qué solución propongo yo? Nada del otro mundo, la verdad. Algo tan sencillo como tener una estructura general donde guardar las distintas órdenes que recibirán los individuos. En esa estructura podemos guardar datos como la ID del proceso (muy útil en muchos casos, aunque sólo sea para destruirlo), sus coordenadas (para saber dónde está), las órdenes que tiene que cumplir (codificadas en un número: 1-> moverse ; 2-> disparar ; 3-> cavar trincheras ; ...), coordenadas de las órdenes que tiene que cumplir (a dónde se tiene que mover, qué posición tiene que bombardear,...), a qué individuo tiene que atacar (su id, su posición en la estructura,...), cuánta energía le queda,... Cualquier dato que necesite nuestro individuo. De esta manera, cada proceso no tiene más que mirar en la tabla para saber qué tiene que hacer, si le queda mucha o poca energía,...

En otro proceso, podremos poner todas las rutinas de control de individuos. Simplemente sabiendo el índice en la tabla del individuo al que le queremos dar órdenes podemos controlar todos los aspectos de ese individuo: le podemos decir que vaya a una posición concreta del mapa (la del ratón,...), cambiar el estado del individuo (que no se mueva ni ataque hasta que haya sido liberado, bajo los efectos de un ataque que le hace moverse más despacio,...), reducir su energía (exposición a rayos cósmicos mortales,...),... De esta manera tendremos todo el control centralizado en un único proceso, por lo que facilitaremos el desarrollo de nuevas aspectos del control, evitaremos muchos errores y abriremos más posibilidades. Por ejemplo, sería mucho más fácil hacer alguna especie de animación previa al principio de la fase (o en medio o donde queramos), ya que bastaría con ir modificando los datos de la estructura y los individuos responderán obedientemente. Y si no hemos activado el proceso encargado de darle el control al jugador no tendremos que preocuparnos por que el jugador intente mover las tropas o hacer cosas raras. Además, si creamos un nuevo tipo de tropa que se mueve dando saltos, ataca insultando al enemigo o hace cualquier otra cosa extraña no tenemos que tocar el sistema de control, simplemente tenemos que programar el proceso de manera que el individuo responda a las órdenes de la manera adecuada.

Otra posibilidad es dar órdenes a varios individuos a la vez. Se pueden guardar sus posiciones en la estructura en un pequeño array e ir variando a la vez todas las posiciones de la tabla que hagan falta. Así nos dará igual si los individuos que hemos elegido son del mismo tipo, de distinto tipo, si se pueden mover o no, todos mirarán las órdenes y su estado y actuarán en consecuencia. Todo esto unido a que

cualquier proceso puede acceder en cualquier momento a estos valores para realizar cualquier tipo de comprobación o ajuste lo convierten, en mi opinión, en un buen sistema.

De todas formas el sistema tiene sus pegadas. La primera es que si queremos eliminar un individuo y luego poder meter otros nuevos tendremos que andar recolocando la lista, es decir, mover todos los que estén por debajo una posición hacia arriba. Eso es fácil de hacer, pero entonces los procesos no estarán mirando la tabla en la posición adecuada. Habría que corregir eso. La manera más sencilla es hacer que el proceso compruebe si su ID y la de la posición de la tabla que está mirando son la misma y pasar a mirar en la posición superior de ahí en adelante si no lo es. Es muy importante realizar esta comprobación y ajuste antes de mirar qué órdenes tiene que cumplir el individuo en cuestión, es decir, que sea lo primero que haga el proceso dentro del LOOP. Si no lo hacemos así puede ser que los datos de la tabla ya se hayan recolocado para tapar el hueco pero el proceso mire en la posición equivocada (en realidad estará mirando las órdenes y los datos del siguiente individuo en la lista).

Otro problema importante es que el número de individuos estará limitado por el número de posiciones que le hayamos puesto a la estructura, a no ser que vayamos aumentándola de manera dinámica. Esto se puede solucionar poniendo un número de posiciones lo suficientemente alto (muchos juegos comerciales tienen límite de número de unidades: en el Starcraft son 200). Lo malo es que si hay pocos individuos que controlar entonces estaremos desaprovechando bastante memoria, pero no es una grave pega hoy en día. Si lo hacemos de manera dinámica usaremos la memoria de una manera mucho más eficiente, pero complicaremos el código bastante.

He hecho un pequeño programa en el que se puede controlar a 4 individuos. Se selecciona cada uno presionando su número en el teclado. Entonces dejará de estar semitransparente e irá a cualquier posición que pinchemos con el ratón (botón izquierdo). Si mientras se está moviendo pinchamos en otro sitio irá a ese nuevo sitio, aunque no haya llegado todavía al que tenía que ir (se podría haber implementado un sistema de puntos de ruta). Si mientras se está moviendo seleccionamos otro individuo le podremos dar órdenes y se moverá, pero sin afectar al movimiento del individuo original, que irá hasta donde tiene que ir. No se ha implementado la posibilidad de elección múltiple para no complicar el programa. Para que funcione deberás tener dos imágenes PNG llamadas "bicho.png" y "raton.png", que representarán, respectivamente a los individuos y al puntero del ratón.

Y esto es todo en principio. La base de la idea es ésta y a partir de ahí lo único que tienes que hacer es particularizarla para tu caso.

```
PROGRAM ejemplo_manejo_individuos;
GLOBAL
  Int grafico;      // Gráfico de los bichos
  Int seleccionado; // Qué individuo hemos seleccionado
  struct bichos[3]; //4 posiciones para 4 bichos(empezando de 0)
                   int x_dest;
                   int y_dest; // Posición a la que se tienen que mover
END
END

BEGIN
  set_mode(640,480,16);

  // Se carga el gráfico de los individuos y del ratón
  grafico=load_png("bicho.png");
```

```

mouse.graph=load_png("raton.png");

// Se llama a los procesos de los individuos
bicho(0,0,grafico,0);
bicho(0,0,grafico,1);
bicho(0,0,grafico,2);
bicho(0,0,grafico,3);

write(0,10,0,0,"Pulsa 1-4 para seleccionar el individuo");
write(0,10,10,0,"Pulsa el ratón para hacer que un individuo se mueva");

LOOP
// Se selecciona el individuo correspondiente
IF (key(_1)) seleccionado=0; END
IF (key(_2)) seleccionado=1; END
IF (key(_3)) seleccionado=2; END
IF (key(_4)) seleccionado=3; END

// Se pone el destino al pinchar con el ratón
IF (mouse.left)
    bichos[seleccionado].x_dest=mouse.x;
    bichos[seleccionado].y_dest=mouse.y;
END

IF (key(_esc)) exit(); END
FRAME;
END
END

// Proceso que controla a los individuos
// cual: variable que indica en qué posición de la struct tiene que mirar
PROCESS bicho(x,y,graph,cual);
BEGIN
LOOP
// El individuo se mueve hacia su destino (si hace falta)
IF (bichos[cual].x_dest>x) x++; END
IF (bichos[cual].x_dest<x) x--; END
IF (bichos[cual].y_dest>y) y++; END
IF (bichos[cual].y_dest<y) y--; END

// Si no está seleccionado se pone en semitransparente
flags=4;
IF (seleccionado==cual) flags=0; END

FRAME;
END
END

```

### Miscelánea de algunas funciones más de Fénix:

#### MINIMIZE()

Como indica su nombre, esta función minimiza la ventana de juego y la sitúa en la barra del sistema

#### MOVE\_WINDOW(POSX, POSY)

Sitúa la esquina superior izquierda de la ventana del juego en las coordenadas (POSX,POSY) de la pantalla, tomando como origen de éstas la esquina superior izquierda de la misma.

##### PARAMETROS:

INT POSX: Coordenada horizontal de la esquina superior izquierda de la ventana respecto a la de la pantalla.

INT POSY: Coordenada vertical de la esquina superior izquierda de la ventana respecto a la de la pantalla.

También es interesante conocer la existencia de la variable global predefinida **FOCUS\_STATUS**. Cuando el juego funciona en una ventana y ésta está en primer plano, el valor de esta variable pasa a valer 1 automáticamente. En cambio, cuando la aplicación está en segundo plano o minimizada, este valor se pone a 0. Para que un juego esté bien integrado en el sistema operativo, se recomienda consultar esta variable cada frame y poner el juego en pausa cuando no está a 1, especialmente para los juegos que disponen de la opción de funcionar en una ventana.

Un ejemplo de ambas funciones:

```
program test004;
begin
  set_mode(320,240,16);
  write(0,0,0,0,"ESC para minimizar la ventana");
  while(!key(_esc))
    move_window(rand(0,640),rand(0,480));
    frame; //Necesario
  end
  delete_text(0);
  while(key(_ESC)) frame; end //No pasa nada hasta que suelte otra vez la tecla ESC
  minimize();
  while(!key(_ESC)) frame; end
end
```

#### MEMORY\_FREE()

Devuelve la cantidad de memoria RAM libre existente en ese momento, en bytes.

#### MEMORY\_TOTAL()

Devuelve la cantidad de memoria RAM total de la máquina, en bytes.

Un ejemplo de ambas funciones:

```
program Test_MEMORY_FREE;
private
  int memBYTE;
  int memMB;
end
begin
  write(0,10,90,3,"Memoria libre en BYTES:");
  write(0,10,110,3,"Memoria libre en MBs: ");
  write_var(0,160,90,3,memBYTE);
  write_var(0,160,110,3,memMB);
  repeat
    memBYTE=memory_free();
    memMB=memory_free()/(1024*1024);
    frame;
  until(key(_esc))
end

program Test_MEMORY_TOTAL;
private
  int memBYTE;
  int memMB;
end
begin
  write(0,10,90,3,"Memoria total en BYTES:");
  write(0,10,110,3,"Memoria total en MBs: ");
  memBYTE=memory_total();
  memMB=memory_total()/(1024*1024);
  write(0,160,90,3,memBYTE);
  write(0,160,110,3,memMB);
  repeat
    frame;
  until(key(_esc))
end
```

Estas funciones son muy útiles para controlar en todo momento el uso que hace nuestro juego de los recursos de la máquina (en concreto, la memoria) y poder descubrir si en algún momento de su ejecución se está cargando demasiado la máquina y agotando la memoria. Es una manera de evitar que nuestros juegos se relenticen demasiado y mantener la máquina en óptimas condiciones permanentemente. Por ejemplo, es muy interesante observar el valor que devuelve MEMORY\_FREE justo después de haber descargado un FPG, o un FNT o un sonido con la correspondiente función unload\_\*: se puede comprobar que se aumenta sensiblemente la cantidad de memoria libre, posibilitando que nuestro juego pueda funcionar con un poco más de fluidez.

### MAP\_BUFFER(LIBRERIA,GRAFICO)

Esta función avanzada devuelve un puntero al buffer del gráfico que se especifica, esto es, un puntero a la zona de memoria donde se guardan los pixels reales que forman la imagen. Cualquier manipulación del contenido de esta zona de memoria afecta directamente al contenido de la imagen.

Algunos usos interesantes de esta función, junto con MEMSET o MEMSETW , permiten realizar efectos visuales interesantes como borrados o rellenados con bandas de color

El puntero devuelto es un puntero a BYTE para gráficos de 8 bits y un puntero a WORD para gráficos de 16 bits.

#### PARAMETROS:

INT LIBRERIA : Número de librería FPG (los gráficos cargados con funciones de tipo LOAD\_XXX o los creados con MAP\_NEW pertenecen a la librería 0).

INT GRAFICO: Número del gráfico dentro de la librería

VALOR RETORNADO: POINTER : Puntero al buffer de píxels del gráfico

### MEMSET (&MEM,VALOR,BYTES)

Esta función avanzada rellena una zona de memoria de tamaño BYTES cuyo comienzo está situado en MEM con el valor indicado por VALOR.

Puede ser muy útil para modificar gráficos de 8 bits en combinación con MAP\_BUFFER.

#### PARAMETROS:

POINTER MEM : Puntero a una zona de memoria

BYTE VALOR : Valor que se desea asignar a todos los bytes de esa zona

INT BYTES : Tamaño en bytes de la zona de memoria

### MEMSETW(&MEM,VALOR,WORDS)

Esta función avanzada rellena una zona de memoria de tamaño WORDS cuyo comienzo está situado en MEM con el valor indicado por VALOR.

Su mayor utilidad consiste modificar gráficos de 16 bits con un color devuelto por RGB en combinación con MAP\_BUFFER.

Hay que tener presente que el tamaño total en bytes de la zona que se rellena es WORDS \* 2 ya que el tamaño de cada WORD es de 2 BYTES.

#### PARAMETROS:

POINTER MEM : Puntero a una zona de memoria

BYTE VALOR : Valor que se desea asignar a todos los bytes de esa zona

INT WORDS : Tamaño en words (1 word = 2 bytes) de la zona de memoria

Un ejemplo de estas funciones es el siguiente. Necesitarás un gráfico llamado "dibujo.png" (preferentemente de unos 50x50 de tamaño). Si ejecutas el código verás que este gráfico estará parcialmente cubierto de un color uniforme: esa zona del gráfico corresponde a la zona que ocupa éste en la memoria y que ha sido rellena de un mismo valor todos sus words (en este caso, 157). El tamaño de

esta zona vendrá dado por los 1200 words especificados -que corresponde más o menos a la mitad de una imagen de 50x50-.

```
program hola;
private
    int idpng;
    /*Es "word pointer" porque trabajamos con un gráfico de 16 bits. Con 8 bits sería "byte pointer" y
    utilizaríamos la función memset().*/
    word pointer pmap;
end
begin
    idpng=load_png("dibujo.png");
    set_mode(640,480,16);
    pmap=map_buffer(0,idpng); //Obtengo la posición en memoria del inicio de la imagen
    memsetw(pmap,157,1200); /*Y modifico a partir de allí el valor de 1200 words, poniéndolos
    todos a 157.*/
    graph=idpng;x=320;y=240; //Muestro el gráfico, ya modificado en memoria, por pantalla
    while(!key(_esc))frame;end
end
```

### GETENV("NombreVariable")

Esta función obtiene el valor actual de la variable de entorno del sistema que se le ha pasado como parámetro, y entre comillas.

**PARAMETROS:** STRING NombreVariable : Nombre de la variable de entorno del sistema cuyo valor se desea conocer.

Una variable de entorno es una variable que tiene definida el sistema operativo en sí (es decir, Windows o Linux, no ningún programa en particular), y que pueden contener información acerca del sistema, como el número de procesadores, la ruta del sistema operativo, la ruta de las carpetas temporales o de los perfiles de usuario, etc. Estas variables del sistema pueden ser accedidas desde cualquier programa de terceros, que es de hecho para lo que sirve la función GETENV de Fénix .

En Windows, puedes observar la lista de variables de entorno definidas actualmene y sus valores si vas a Panel de Control->Rendimiento y mantenimiento->Sistema->Opciones avanzadas->Variables de entorno. Desde allí también podrás modificar los valores de las variables que desees, e incluso podrás crear o destruir variables. En la consola de comandos puedes hacer algo parecido con el comando "set".

Ejemplos de variables de entorno típicas que ya están predefinidas de serie en un sistema Windows pueden ser PATH (que vale las rutas -separadas por ;- que el sistema incorpora dentro de la lista de rutas de búsqueda de ejecutables), TEMP (que vale la ruta de la carpeta temporal del sistema), etc.

Si usas Linux, existe este mismo comando, "set", para gestionar las variables de entorno de este sistema. En Linux existen otras variables de entorno típicas, como PATH (con el mismo significado que en Windows), HOME (que vale la ruta de la carpeta personal del usuario actual), etc.

De todas maneras, si no sabes lo que es una variable de entorno, seguramente no necesitarás utilizar esta función.

Un ejemplo de su uso podría ser el siguiente código:

```

BEGIN
set_mode(640, 480, 16);
write(0,10,100,0, "El valor de la variable PATH es : ");
write(0,10,120,0, getenv("PATH"));
write(0,10,140,0, "El valor de la variable TEMP es : ");
write(0,10,160,0, getenv("TEMP"));
write(0,10,180,0, "El valor de la variable OS es: ");
write(0,10,200,0, getenv("OS"));
write(0,10,220,0, "El valor de la variable WINDIR -sólo existente en Windows- es : ");
write(0,10,240,0, getenv("WINDIR"));
while(!key(_esc)) FRAME; End;
END

```

Un ejemplo práctico donde se demuestra la utilidad de este comando lo podemos encontrar en el siguiente código. En él se genera (gracias a la función "crear\_jerarquia") una jerarquía de carpetas dentro del perfil del usuario actual para guardar allí un archivo con posible contenido a preservar en nuestro juego (puntos, récords, etc). La función "crear\_jerarquia" devuelve 0 si ha conseguido crear los directorios y -1 si algo ha fallado. La gracia del ejemplo está en que el lugar donde se guarda dicho archivo será diferente según el usuario que tenga iniciada la sesión y esté ejecutando el programa, con lo que disponemos así de un sistema para guardar datos personales de cada jugador en distintas carpetas, las cuales se llamarán igual pero estarán situadas cada una en el perfil de cada usuario por separado.

Además, este programa funciona tanto en Windows como en Linux. Incluso en Windows, funciona como debiera aunque le mandemos a un directorio de otra unidad. distinta a la unidad desde la que estamos ejecutando el juego.

```

// WINXP/VISTA and Linux compatible
// AUTHOR: Josebita, PiXeL and Devilish
program save_game_directory;
Private
string savegamedir = "";
//Atención a las barras!!. Todas han de ser así (/) y hay que añadir barra al final
string developerpath = "/DevilishGames/SonoroTV/";
string savegamepath = "save.dat";
int fd=0;
end
Begin
set_mode(640, 480, 16);
/*FXI_OS es una variable global de Fénix que no hemos visto hasta ahora. Indica el sistema operativo en el que está funcionando el intérprete de Fénix (es decir, el sistema operativo del usuario). En este if estamos comprobando que su valor sea igual a la constante OS_WIN32, el cual denota toda la familia de Windows de 32 bits.*/
if (fxi_os == OS_WIN32) //Si estamos en Windows...
/*...generamos la ruta absoluta que crearemos posteriormente en disco, a partir de la ruta que obtenemos del valor de la variable APPDATA, concatenándole la ruta dada por developerpath. La variable APPDATA es una variable de entorno que ha de existir por defecto siempre en cualquier Windows 2000/XP/Vista*/
savegamedir = getenv("APPDATA") + developerpath; // ¡¡¡Atención a las barras!!!
else //Si estamos en Linux...
/*...generamos la ruta absoluta que crearemos posteriormente en disco, a partir de la ruta que obtenemos del valor de la variable HOME, concatenándole la ruta dada por developerpath. La variable HOME es una variable de entorno que ha de existir por defecto siempre en Linux*/

```

```

savegamedir = getenv("HOME") + "." + developerpath;
end

if(savegamedir == developerpath) // La variable APPDATA no está definida, por alguna razón
savegamedir = cd();
say("APPDATA no definida"); //Mensaje de debug
end

if(crear_jerarquia(savegamedir) == -1) // La creación del directorio ha fallado, tenemos dos opciones:
savegamedir = cd(); // Intentarlo a la desesperada en el directorio actual (MAAL!!)
write(0, 0, 0, 0, "Save directory error.."); //Avisar al usuario y salir:no intentar guardar el fichero(BIEN)
else
/*Ahora que sabemos que hemos creado bien la ruta o que ya existía ya podemos crear la ruta entera al
fichero de partida guardada*/
savegamepath = savegamedir + savegamepath;
// Y ya podemos trabajar con él normalmente
fd = fopen(savegamepath, O_WRITE);
if (fd) write(0, 0, 10, 0, "Write OK"); /*Aquí podría escribir algo dentro del archivo...*/
else write(0, 0, 10, 0, "Write Error");
end
fclose(fd);
write(0, 0, 10, 0, "Write OK");
write(0, 0, 20, 0, ""+savegamepath);
End
while(!key(_esc)) FRAME; End
exit();
end

```

```

Function crear_jerarquia(string nuevo_directorio)
Private
string directorio_actual="";
string rutas_parciales[10]; // Sólo acepta la creación de un máximo de 10 directorios
int i=0;
int i_max=0;
End
Begin
directorio_actual = cd(); // Recuperamos el directorio actual de trabajo, para volver luego a él
if(chdir(nuevo_directorio) == 0) // El directorio ya existe!
cd(directorio_actual);
return 0;
end
i_max = split("[\V]", nuevo_directorio, &rutas_parciales, 10);
chdir("");
while (i<i_max)
while(rutas_parciales[ i ] == "") // Se salta partes en blanco
if(i++ >=i_max)
cd(directorio_actual);
return 0;
end
end
if(chdir(rutas_parciales[ i ]) == -1)
if(mkdir(rutas_parciales[ i ]) == -1) // Error al intentar crear el directorio
cd(directorio_actual);

```

```

    return -1;
end
chdir(rutas_parciales[ i ]);
end
i++;
end
chdir(directorio_actual);
return 0;
End

```

### **Códigos de un picaladrillos y un matamarcianos sin utilizar ninguna imagen externa:**

A continuación presento un código comentado de un picaladrillos y otro de un matamarcianos, donde los gráficos de todos los procesos son generados internamente en el código fuente, con lo que no necesitaremos ningún fichero externo para poder ejecutar el juego.

Estos códigos nos pueden servir como demostración rápida al público advenedizo de lo que se puede llegar a hacer con Fénix en una pocas líneas, de forma sencilla y rápida.

```

//Este código viene "de serie" con la distribución 0.92a de Fénix
program Juego;
global
    int XSize=200,YSize=400;
    int xIncrement=5,yIncrement=5;
end
private
    int n,a;
end
begin
    set_mode(XSize,YSize,16);
    Barra();
    Bola();
/*Dibujó los ladrillos. Se pintarán 19 filas y, si la anchura de la pantalla es 200, se pintarán 7 ladrillos
por columna */
    from n=10 to XSize-10 step 30
        from a=20 to 200 step 10
            Bloque(n,a);
        end
    end
    repeat frame; until (key(_esc))
    exit();
end

PROCESS Barra()
BEGIN
    graph=new_map(50,10,16);
    map_clear(0,graph,rgb(100,100,100));
/*La posición inicial de la raqueta dependerá de las dimensiones de la pantalla. En todo caso, aparecerá
centrada horizontalmente y cerca de su límite inferior*/
    y=YSize-40;
    x=XSize/2;
    LOOP
        if (key(_left))

```

```

        if (x>0) x=x-5; end
    elseif (key(_right))
        if (x<XSize) x+=5; end
    end
    /*Si hay colisión con la bola, invierto el incremento del movimiento de ésta en la dirección vertical*/
    if (Collision(type Bola)) yIncrement=-yIncrement; end
    frame;

    END
END

PROCESS Bola()
BEGIN
    graph=new_map(10,10,16);
    drawing_map(0,graph);
    drawing_color(rgb(255,0,0));
    draw_fcircle(5,5,5);

    /*La posición inicial de la raqueta dependerá de las dimensiones de la pantalla. En todo caso, aparecerá
    centrada horizontalmente y cerca de su límite inferior, por encima de la raqueta.*/
    x=XSize/2;y=ySize-50;
    LOOP
        if (x>=XSize OR x<=0)
            //Invierto el incremento del movimiento de la bola en la dirección horizontal
            xIncrement=-xIncrement;
        elseif (y<=0)
            yIncrement=-yIncrement;
        end

        /*Si la bola desaparece por el límite inferior de la pantalla, se acabó el juego, ya que (después de ejecutar
        el proceso "Perdedor", entro en un bucle infinito sin salida, con lo que la bola deja de ser funcional.*/
        if (y>=YSize)
            Perdedor();
            loop frame; end

        end

        //Movimiento de la bola
        x=x+xIncrement;
        y=y+yIncrement;

        frame;

    END
END

PROCESS Bloque(INT x,y)
BEGIN
    z=100;
    /*Los ladrillos serán de 30x10. Su tamaño es fijo, pero el número de ellos dependerá de las dimensiones
    de la pantalla*/
    graph=new_map(30,10,16);
    map_clear(0,graph,rgb(rand(0,255),rand(0,255),rand(0,255)));

    LOOP
        /*Si hay colisión con la bola, invierto el incremento del movimiento de ésta en la dirección vertical, y
        termino inmediatamente la ejecución de la instancia actual, con lo que el ladrillo desaparecerá de la

```

```

pantalla.*/
                if (Collision(TYPE Bola))
                    yIncrement=-yIncrement;
                    return;
                end
            frame;
        END
    END

/*Escribo 200 veces en posiciones aleatorias y con colores y profundidades aleatorias también el texto
(convertido en imagen)"LOOSERRRR"*/
PROCESS Perdedor()
PRIVATE
    INT aux;
BEGIN
    FROM x=1 TO 200
        text_z=-400
        set_text_color(rgb(rand(0,255),rand(0,255),rand(0,255)));
        aux=write_in_map(0,"LOOSERRRR",0);
        put(0,aux,rand(0,XSize),rand(0,YSize));
        unload_map(0,aux);
    END
END

```

```

Program shoot_em_up;
global
    int tiro=1;//Si vale 1, el jugador puede usar el disparo especial.Si vale 0,no.
    int score=0;
end
begin
    set_fps(30,0);
    set_title("Marcianitos");
    set_mode (640,480,16);
    nave(400,500);
    /*Inicialmente salen cinco enemigos.En el código posterior se puede ver que cuando un enemigo
    desaparece de la pantalla a causa del impacto de un misil de nuestra nave, inmediatamente se genera un
    nuevo enemigo en la parte superior de la pantalla -y con una x aleatoria-. Además, si el enemigo no muere
    pero llega al extremo inferior de la pantalla, también ocurre lo mismo: se genera un nuevo enemigo en el
    límite superior, con lo que siempre habrá en pantalla 5 enemigos de forma permanente*/
    enemigo(rand(5,635),-10);
    enemigo(rand(5,635),-10);
    enemigo(rand(5,635),-10);
    enemigo(rand(5,635),-10);
    enemigo(rand(5,635),-10);
    write(0,450,460,3,"SCORE :");
    write_var(0,600,460,3,score);
    loop
        frame;
    end
end

process nave(x,y)
private
    int point;

```

```

/*xpos y ypos definen el punto desde donde saldrán los misiles, gracias a la función get_real_point*/
int xpos,
int ypos;
end
begin
graph=new_map(7,7,16);map_clear(0,graph,rgb(255,255,0));
size=80;
z=10;//Ponemos Z>0 para que los misiles se pinten por debajo
loop
if(key(_up))y=y-10;end
if(key(_down))y=y+10;end
if(key(_right))x=x+10;end
if(key(_left))x=x-10;end
if(y<10)y=10;end
if(y>450)y=450;end
if(x<20)x=20;end
if(x>620)x=620;end
//Pulsando SPACE disparamos un tipo de misil
if(key(_space))
/*Obtenemos del proceso actual (nave) la coordenada X e Y del punto de control 0 (es decir, el centro de
la nave)...*/
get_real_point(0,&xpos,&ypos);
//...para posicionar el misil en esa coordenada
misil(xpos,ypos);
/*Igualmente hacemos para situar allí donde está la nave una explosión producida por la propulsión de
nuestros misiles*/
explosion(xpos,ypos,20);
end
/*Pulsando CONTROL disparamos otro tipo de misil, formado por tres misiles en paralelo*/
if(key(_control))
/*Si tiro==0 no podemos disparar. Tiro valdrá 0 justo después de haber lanzado uno de estos misiles
triples, de manera que si se continúa pulsando la tecla CTRL, no se podrá seguir disparando. Se tendrá
que dejar de pulsar la tecla CTRL, para que a la siguiente vez que se pulse, se ejecute el else (tiro=1), y en
el siguiente frame, se pueda volver a disparar -sólo una vez, otra vez, y así-. Como se puede ver, este
sistema es para deshabilitar la posibilidad de hacer disparos infinitos con la tecla CTRL siempre
pulsada.*/
if(tiro!=0)
get_real_point(0,&xpos,&ypos);
misil(xpos-10,ypos);
misil(xpos,ypos);
misil(xpos+10,ypos);
//Por cada misil lanzado hay una explosión de propulsión
explosion(xpos-10,ypos,20);
explosion(xpos,ypos,20);
explosion(xpos+10,ypos,20);
tiro=0;
end
else
/*Si se ejecuta este else es que se ha pulsado CTRL de nuevo, después de haber disparado ya antes y haber
soltado la tecla, con lo que hacemos que se pueda volver a disparar*/
tiro=1;
end
frame;
end

```

```

end

process misil(x,y)
begin
  graph=new_map(7,7,16);map_clear(0,graph,rgb(0,255,0));
  size=70;
  loop
  /*Si se sale fuera de la pantalla, se acaba este proceso. */
    if(out_region(id,0))break;end
    y=y-10;
    frame;
  end
end

process explosion(x,y,size)
private
  int i;
end
begin
  /*Durante 100 fotogramas (cantidad que se puede regular para hacer que la explosión dure más o menos)
  se verá una explosión consistente en un cuadradito que se moverá levemente de forma aleatoria a partir de
  su posición inicial, la cual vendrá dada por desde dónde se invoque a este proceso: las explosiones pueden
  provenir de la eliminación de un enemigo o bien de la propulsión de los misiles de la nave protagonista.*/
  for(i=0;i<=100;i++)
    graph=new_map(7,7,16);map_clear(0,graph,rgb(rand(100,200),rand(0,250),0));
    x=x+rand(-2,2);
    y=y+rand(-2,2);
    frame;
  end
end

process enemigo(x,y)
private
  int tiro_enemigo=0;
  int point,xpos,ypos;
end
begin
  graph=new_map(7,7,16);map_clear(0,graph,rgb(255,0,255));
  size=80;
  loop
    y=y+5;
  /*Si se choca contra el misil de la nave, genero una explosión, aumento el score, genero un nuevo enemigo
  en la parte superior de la pantalla relevando al presente, y éste lo mato con el break.*/
    if(collision(type misil))
      get_real_point(0,&xpos,&ypos);
      explosion(xpos,ypos,100);
      score=score+10;
      enemigo(rand(5,635),-50);
      break;
    end
  /*De igual manera, si el enemigo alcanza el extremo inferior de la pantalla, genero un nuevo enemigo en
  la parte superior de la pantallarelevando al presente, el cual lo mato con el break;*/
  if(y>680)

```

```

    enemigo(rand(5,635),-50);
    break;
end

/*Sistema temporizador para los misiles de los enemigos. Éstos se crearán automáticamente cada vez que
la variable tiro_enemigo llegue a valer 50. Esta variable aumenta en uno su valor a cada fotograma, y
cuando llega a 50, después de haberse generado el misil enemigo, se vuelve a resetear a 0 para volver a
empezar la cuenta. Si se quiere que los enemigosdisparen más frecuentemente, no hay más que rebajar el
límite de 50 en el if por otro número más pequeño: ese número indicara cada cuántos fotogramas los
enemigos dispararán*/
    tiro_enemigo++;
    if(tiro_enemigo>50)
        get_real_point(0,&xpos,&ypos);
        misil_enemigo(xpos,ypos);
        tiro_enemigo=0;
    end
frame;
end
end

/*Exactamente igual que el proceso misil, pero con otro color y el sentido vertical del movimiento al
contrario. (De hecho, como sólo son dos pequeñas diferencias, se podría haber escrito un solo proceso
que sirviera para generar misiles propios y enemigos, con el uso de parámetrosadecuados).*/
process misil_enemigo(x,y)
private
    int xpos;
    int ypos;
end
begin
    graph=new_map(7,7,16);map_clear(0,graph,rgb(255,0,255));
    size=70;
    loop
        if(out_region(id,0))break;end
        y=y+10;
        frame;
    end
end
end

```

Si te fijas en el código, verás que hemos utilizado la función `get_real_point()` para obtener el punto de control 0 de los procesos pertinentes. Ya sabemos que el punto de control 0 corresponde al centro del gráfico del proceso, con lo que haciendo servir las variables locales X e Y ya no hace falta utilizar esta función. No obstante, se ha mantenido para mostrar otra posibilidad de trabajo, y para dejar abierta una posible mejora que consistiría en crear más puntos de control en los gráficos y utilizarlo mediante `get_real_point` (por ejemplo, crear un punto en la parte delantera de la nave y los enemigos, desde donde podrían salir los disparos, por ejemplo).

Si juegas, verás que el protagonista no muere nunca. En otras palabras: los disparos de los enemigos (ni éstos propiamente) afectan en lo más mínimo a la salud de nuestra nave. Se deja como ejercicio solucionar esto.

### **Código para crear explosiones un poco más espectaculares:**

A continuación veremos un código bastante ingenioso para lograr, a partir de un único gráfico

con forma de llamarada amarillo-naranja, un efecto de explosión completa, que, si bien no es nada del otro mundo, puede causar cierto asombro. Para ello dispondremos de dicho gráfico en el interior de un FPG llamado "explos.fpg", con código 001.

El proceso "explosion()", si te gusta, lo puedes utilizar en tus propios juegos para mostrar el efecto visto.

```

Program ejemplo_expl;
Global
    Int fichero1;
end
Begin
    set_mode(640,480,16);
    fichero1=load_fpg("explos.fpg");
    Loop
        If(key(_esc)) exit();End
        explosion(260,220,size);
        Frame;
    End
End

Process explosion(x,y,size)
Begin
    graph=331;
    set_fps(24,0);
    Repeat
        If (size>50 AND rand(0,40)<size/8) /* Crea otras explosiones*/
            x=x+rand(-25,45);
            y=y+rand(-30,15);
        End
        size=size-4; // Cambian de tamaño reduciendose
        Frame;
    Until (size<5) // Continua hasta que sean demasiado pequeñas
End

```

**Códigos para generar llamas de fuego (azul):**

Los dos códigos son muy parecidos (ambos ofrecen un efecto similar) aunque son a la vez igual de complejos:

```

PROGRAM fuegol; //TXABI '98
GLOBAL
    int media;
    int tx1;
    int tx2;
    int i;
end
Begin
set_mode(320,200,16);
set_fps(24,0);
while(scan_code=0) //Mientras no se pulse ninguna tecla...
    FOR(i=0; i<10 ; i++)
        tx1=rand(50,270);

```

```

tx2=rand(50,270);
put_pixel(tx1,196,rand(200,255));
put_pixel(tx2,196,0);
put_pixel(tx1+1,196,rand(200,255));
put_pixel(tx2+1,196,0);
put_pixel(tx1-1,196,rand(200,255));
put_pixel(tx2-1,196,0);
put_pixel(tx1+2,196,rand(200,255));
put_pixel(tx2+2,196,0);
END

FOR(x=45; x<275 ; x=x+2) //Fundido hacia la derecha...
FOR(y=140 ; y<196 ; y=y+2)
media=get_pixel(x+2,y) + get_pixel(x-2,y) + get_pixel(x+2,y+2) + get_pixel(x-2,y+2);
media=media>>2;
IF(media>0)media=media-2;END
IF(media>=0)
put_pixel(x,y,media);
put_pixel(x,y+1,media);
put_pixel(x-1,y,media);
put_pixel(x-1,y+1,media);
END
END
END
//FRAME;

FOR(x=275 ; x>45 ; x=x-2) //Fundido hacia la izquierda...
FOR(y=140 ; y<196; y=y+2)
media=get_pixel(x+2,y) + get_pixel(x-2,y) + get_pixel(x+2,y+2) + get_pixel(x-2,y+2);
media=media>>2;
IF(media>0)media=media-2;END
IF(media>=0)
put_pixel(x,y,media);
put_pixel(x,y+1,media);
put_pixel(x-1,y,media);
put_pixel(x-1,y+1,media);
END
END
END
FRAME;
END
EnD

```

```

//Author: Oasistudios
Program fire;
Global
int x1;
int y1;
int media;
int si;
end
Begin
set_mode (320,240,16);
set_fps(24,0);

```

```

Loop
  for(x1=0;x1<320;x1=x1+2)
    si=rand(25,255);
    put_pixel(x1,239,si);
    put_pixel(x1+1,239,si);
    put_pixel(x1,238,si);
    put_pixel(x1+1,238,si);
  end

  for (y1=0;y1<240;y1=y1+2)
    for (x1=2;x1<318;x1=x1+2)
      media=get_pixel(x1,y1);
      if(media >0)media--; end

      if (y1>=210)
        media=media+get_pixel(x1-2,y1);
        media=media+get_pixel(x1+2,y1);
        media=media+get_pixel(x1,y1+2);
        media=media>>2;
      end
      if(media !=0) media--; end

      put_pixel(x1,y1-1,media);
      put_pixel(x1,y1-2,media);
      put_pixel(x1+1,y1-1,media);
      put_pixel(x1+1,y1-2,media);
      put_pixel(x1-1,y1-1,media);
      put_pixel(x1-1,y1-2,media);
    end
  end
end
If(key(_esc)) exit(); End
frame;
End
end

```

### **Código que ondula una imagen:**

Dicha imagen la llamaremos "dibujo.png".

```

program waveywave;
const
  x_res=320;
  y_res=240;
end
local
  bool killed=false;
end
private
  int box;
end
begin
  set_mode(x_res,y_res,16);
  set_fps(60,0);
  put_screen(0,load_png("dibujo.png"));

```

```

//Se pueden cambiar el valor de los diferentes parámetros para observar qué efectos produce
waves(50,50,8,255,100,100,15,3000,3,x_res,y_res);

//Este cuadrado delimitará visualmente la zona que sufre ondulaciones, y se puede mover con los cursores
box=draw_rect(0,0,100,100);
while(!key(_esc))
    if(key(_left) and son.x>50)son.x=son.x-5;end;
    if(key(_right) and son.x<x_res-50)son.x=son.x+5;end;
    if(key(_up) and son.y>50)son.y=son.y-5;end;
    if(key(_down) and son.y<y_res-50)son.y=son.y+5;end;
    move_draw(box,son.x-50,son.y-50);
    frame;
end
son.killed=true;
end

process waves(x,y,z,alpha,int anchura,int altura,int amp,int freq,int spd,int x_res,int y_res)
private
    int pointer gfx;
    int ani=0,count,source;
end
begin
    graph=new_map(anchura,altura,16);
    gfx=alloc(sizeof(gfx)*altura);
    source=new_map(x_res+2*amp,y_res,16);
    map_block_copy(0,source,amp,0,0,0,x_res,y_res,0);
    for(count=0; count<altura; count++)
        gfx[count]=new_map(anchura+2*amp,1,16);
    end
    while(!killed)
        ani=ani+spd;
        map_clear(0,graph,0);
        for(count=0;count<altura;count++)
            map_clear(0,gfx[count],0);
            map_block_copy(0,gfx[count],0,0,source,x-anchura/2,y-altura/2+count,anchura+2*amp,1,0);
            map_put(0,graph,gfx[count],(anchura/2)+get_distx((ani+y+count)*freq,amp),count);
        end
        frame;
    end
    unload_map(0,source);
    for(count=0; count<altura; count++) unload_map(0,gfx[count]); end
end

```

Este código es ciertamente complicado de entender ya que en él se mezclan punteros, gestión dinámica de memoria, copia de bloques de imágenes, etc. Pero como recurso a tener en cuenta, es interesante ya que con sólo incorporar a nuestro proyecto el proceso "waves()" ya dispondremos de su funcionalidad.

**El juego del “disparaletas”:**  
(código original de Carles)

No se trata ahora de llenar este documento de códigos curiosos y/o interesantes, porque sería un libro inabarcable. Los límites los impone sólo la imaginación de cada uno. Pero sí que me gustaría presentar este pequeño juego como ejemplo de que con una idea realmente simple (y un código no muy difícil) se puede lograr un entretenimiento ameno. La intención es mostrar que no tenemos por qué ceñirnos siempre a los moldes tradicionales (carreras de coches, juegos de lucha, juegos tradicionales de mesa, plataformas, etc) sino que un juego original puede surgir a partir de cualquier buena idea.

El juego consiste en un menú inicial, donde se configuran los parámetros del juego, para seguidamente “disparar letras”. Es decir, en la pantalla saldrán aleatoriamente letras -y en posiciones aleatorias también- y el jugador tendrá que pulsar la tecla correspondiente dentro de un tiempo de reacción fijado. Si logra ser lo suficientemente rápido y acertar la letra, ésta explotará. Al final del juego se establece el recuento total de letras acertadas. Así de simple.

```
PROGRAM cazaletas;
GLOBAL
  int intentos=1;
  int aciertos=0;
  int tiempo=5;
  int oportunidades=0;
end
BEGIN
  set_mode(320,240,16,mode_2xscale);
  menu();
END

process menu()
private
  int opcion=1;
  int id_texto[3];
  int gatillo;
end
begin
  let_me_alone();
  delete_text(0);
  Set_text_color(rgb(255,255,255));
  write(0,160,10,4,"Jugador, bienvenido a cazaletas");
  write(0,160,30,4,"Instrucciones: durante el juego pulsa la tecla");
  write(0,160,40,4,"de la letra que aparece en la pantalla.");
  write(0,160,50,4,"Selecciona el número de intentos o el tiempo");
  write(0,160,60,4,"con las teclas derecha e izquierda y pulsa");
  write(0,160,70,4,"enter para empezar. Buena suerte.");
  write(0,150,90,4,"intentos:");
  write_var(0,185,90,4,intentos);
  write(0,150,100,4,"tiempo:");
  write_var(0,185,100,4,tiempo);
  write(0,150,120,4,"aciertos:");
  write_var(0,185,120,4,aciertos);
  loop
/*Sistema muy ingenioso para generar un menú con la opción seleccionada realzada en otro color.
Simplemente se trata de, antes de escribir el texto de la opción elegida, cambiarle el color con la función
```

*set\_text\_color* y escribir seguidamente dicha opción de nuevo en pantalla con el nuevo color. La variable *gatillo* sirve para controlar el pulsado de las teclas y evitar que una sola pulsación demasiado larga el programa la entienda como muchas pulsaciones: mientras *gatillo=0* se reconocerán las pulsaciones de teclas y cuando valga 1 no (de hecho, siempre, justo después de pulsar una tecla *gatillo* pasa a valer 1 por eso). Solamente volverá a valer 0 en el momento que la variable del sistema *scan\_code* valga 0, es decir, que no se detecte ninguna pulsación actual de ninguna tecla, y por tanto, se pueda volver a aceptar otra vez nuevas pulsaciones.\*/

```

        if(opcion==1)
            Set_text_color(rgb(255,0,0));
        else
            Set_text_color(rgb(0,255,0));
        end
        id_texto[0]=write(0,160,150,4,"modo reflejos");

    if(opcion==2)
        Set_text_color(rgb(255,0,0));
    else
        Set_text_color(rgb(0,255,0));
    end
    id_texto[1]=write(0,160,160,4,"modo time attack");

        if(opcion==3)
            Set_text_color(rgb(255,0,0));
        else
            Set_text_color(rgb(0,255,0));
        end
        id_texto[2]=write(0,160,170,4,"salir");
        frame;
        delete_text(id_texto[0]);
        delete_text(id_texto[1]);
        delete_text(id_texto[2]);

```

*/\*Según en la opción del menú donde estemos, y el cursor (derecho o izquierdo) que pulsemos, así afectará a la configuración del juego (número de intentos y tiempo). Fijarse que el nº de intentos sólo puede ir entre 1 y 10, y el tiempo entre 5 y 30.\*/*

```

    IF(key(_right) and intentos<10 and opcion==1 and gatillo==0) intentos++; gatillo=1; END
    IF(key(_left) and intentos>1 and opcion==1 and gatillo==0) intentos--; gatillo=1; END
    IF(key(_right) and tiempo<30 and opcion==2 and gatillo==0) tiempo=tiempo+5; gatillo=1; END
    IF(key(_left) and tiempo>5 and opcion==2 and gatillo==0) tiempo=tiempo-5; gatillo=1; END
    if(key(_down) and gatillo==0 and opcion<3) opcion++; gatillo=1; end
    if(key(_up) and gatillo==0 and opcion>1) opcion--; gatillo=1; end
    if(gatillo==1 and scan_code==0) gatillo=0; end

```

*/\*Entramos para jugar\*/*

```

    if(key(_enter))
        if(opcion==1) cazar(1); end
        if(opcion==2) cazar(2); end
        if(opcion==3) exit(0,""); end
    end
end //loop
end

```

*PROCESS cazar(modos)*

```

private
    int time;
    int id_letra;
end
BEGIN
    let_me_alone();
    delete_text(0);
    Set_text_color(rgb(255,255,255));
    aciertos=0;
    oportunidades=0;
    /*Las líneas que siguen sirven para causar un efecto de cuenta atrás antes de empezar el juego: salen una
    detrás de otra las palabras "preparado", "3", "2" y "1".*/
    timer[1]=0;
    time=0;
    write(0,160,120,4,"preparado");
    while(timer[1]<100) frame; end
    timer[1]=0;
    delete_text(0);
    write(0,160,120,4,"3");
    while(timer[1]<100) frame; end
    timer[1]=0;
    delete_text(0);
    write(0,160,120,4,"2");
    while(timer[1]<100) frame; end
    timer[1]=0;
    delete_text(0);
    write(0,160,120,4,"1");
    while(timer[1]<100) frame; end
    timer[1]=0;
    delete_text(0);
    /*Aquí comienza el juego de verdad*/
    if(modos==1)
        LOOP
            time=-25*oportunidades*oportunidades+525*oportunidades+100;
            IF(timer[1]>=time)
                signal(id_letra,s_kill);
                frame;
                delete_text(0);
                id_letra=letras(rand(1,26));
            oportunidades++;
            END
            IF(oportunidades==intentos+1)
                menu();
                delete_text(0);
                break;
            END
        frame;
    end //loop
end //if
if(modos==2)
    id_letra=letras(rand(1,26));
    write_var(0,160,5,4,time);
loop
    time=tiempo-timer[1]/100;

```

```

        if( not exists(id_letra))
            id_letra=letras(rand(1,26));
        end
        if(timer[1]>tiempo*100) menu(); end
        frame;
    end
end // if
end

/*El código de este proceso es evidente*/
PROCESS letras(letra)
PRIVATE
    int pulsar;
    int id_letra;
end
BEGIN
    x=rand(10,310);
    y=rand(10,230);
    SWITCH (letra)
        case 1: id_letra=write(0,x,y,4,"a"); pulsar=_a; END
        case 2: id_letra=write(0,x,y,4,"b"); pulsar=_b; END
        case 3: id_letra=write(0,x,y,4,"c"); pulsar=_c; END
        case 4: id_letra=write(0,x,y,4,"d"); pulsar=_d; END
        case 5: id_letra=write(0,x,y,4,"e"); pulsar=_e; END
        case 6: id_letra=write(0,x,y,4,"f"); pulsar=_f; END
        case 7: id_letra=write(0,x,y,4,"g"); pulsar=_g; END
        case 8: id_letra=write(0,x,y,4,"h"); pulsar=_h; END
        case 9: id_letra=write(0,x,y,4,"i"); pulsar=_i; END
        case 10: id_letra=write(0,x,y,4,"j"); pulsar=_j; END
        case 11: id_letra=write(0,x,y,4,"k"); pulsar=_k; END
        case 12: id_letra=write(0,x,y,4,"l"); pulsar=_l; END
        case 13: id_letra=write(0,x,y,4,"m"); pulsar=_m; END
        case 14: id_letra=write(0,x,y,4,"n"); pulsar=_n; END
        case 15: id_letra=write(0,x,y,4,"o"); pulsar=_o; END
        case 16: id_letra=write(0,x,y,4,"p"); pulsar=_p; END
        case 17: id_letra=write(0,x,y,4,"q"); pulsar=_q; END
        case 18: id_letra=write(0,x,y,4,"r"); pulsar=_r; END
        case 19: id_letra=write(0,x,y,4,"s"); pulsar=_s; END
        case 20: id_letra=write(0,x,y,4,"t"); pulsar=_t; END
        case 21: id_letra=write(0,x,y,4,"u"); pulsar=_u; END
        case 22: id_letra=write(0,x,y,4,"v"); pulsar=_v; END
        case 23: id_letra=write(0,x,y,4,"w"); pulsar=_w; END
        case 24: id_letra=write(0,x,y,4,"x"); pulsar=_x; END
        case 25: id_letra=write(0,x,y,4,"y"); pulsar=_y; END
        case 26: id_letra=write(0,x,y,4,"z"); pulsar=_z; END
    END
    /*Mientras no se pulse la tecla adecuada, estaremos en un bucle infinito*/
    loop
        IF(key(pulsar))
            aciertos++;
            explosion(x,y);
            break;
        end
    end
end

```

```

    frame;
end
delete_text(id_letra);
END

/*La explosión son dos círculos concéntricos que van creciendo de tamaño hasta desaparecer*/
PROCESS explosion(x,y)
BEGIN
    graph=new_map(11,11,8);
    drawing_map(0,graph);
    drawing_color(rand(1,255));
    draw_circle(5,5,5);
    drawing_color(rand(1,255));
    draw_circle(5,5,2);
    REPEAT
        size=size+10;
    frame;
    UNTIL(size>200)
END

```

### **Simulación de la escritura de una antigua máquina de escribir:**

```

program prueba;
const
    NUMLINEAS=5;
end
private
    string texto= "Debería estar prohibido
haber vivido y no haber amado.
Por eso, tírame un beso,
que he sido preso de nuestro encierro.
Jugar con fuego.";
end
begin
    set_mode(640,480,16);
    escribe_texto(texto);
    loop
        frame;
    end
end

process escribe_texto(string strin)
private
    string lineas[NUMLINEAS-1];
    int lineaactual=0;
    string caracter;
    int i;
end
begin
    /*Este bucle sirve en general para rellenar en cada elemento del vector lineas[] una cadena de caracteres

```

que corresponde a cada una de las líneas del texto. Cada una de estas líneas se tratará posteriormente en el programa como si también fuera una especie de vector de caracteres.\*/

```
i=0;
while (i<=len(strin))
    character=strin[i];
/*En Windows el salto de línea viene dado por dos caracteres, chr(10) y chr(13). Si se detecta el primero,
lo que hago es recoger el segundo sin añadir nada al vector líneas[], cambiar de línea y entonces seguir
con
la siguiente iteración normal.*/
    if(character==chr(10))
        i++;character=strin[i];
        líneaactual++;
    else
/*Aquí es donde realmente añado el siguiente carácter a la línea que corresponda, como si cada elemento
del vector líneas (es decir, líneas[0], líneas[1]...) fuera a su vez un vector de caracteres.*/
        líneas[líneaactual]=líneas[líneaactual]+character;
    end
    i++;
End

//Para cada línea
for(líneaactual=0; líneaactual<NUMLINEAS; líneaactual++)
    //Para cada carácter de esa línea
    For(i=0; i<=len(líneas[líneaactual]); i++)
        //El switch determina la posición "y" del texto
        switch(líneaactual)
/*Imprimo un carácter cada vez. Trato cada carácter como si fuera un elemento "char" dentro de los
vectores de elementos "char" que representa que son líneas[0], líneas[1], líneas[2]... aunque en realidad
en vez de vectores de caracteres son cadenas; pero los conceptos son fácilmente intercambiables*/
            case 0: write(0,10+14*i,340,0,líneas[líneaactual][i]); end
            case 1: write(0,10+14*i,370,0,líneas[líneaactual][i]); end
            case 2: write(0,10+14*i,400,0,líneas[líneaactual][i]); end
            case 3: write(0,10+14*i,430,0,líneas[líneaactual][i]); end
            case 4: write(0,10+14*i,460,0,líneas[líneaactual][i]); end
        end
    frame;
End

End
//No pasa nada hasta que apreto la tecla CTRL...
while(!key(_control)) frame; end
//...momento en el que borro todo el texto
delete_text(all_text);
loop
    frame;
end
end
```

### **Pequeño tutorial teórico para el juego de la serpiente:**

Lo primero que hay que hacer cuando se afronta la resolución de cualquier problema mediante un programa de ordenador (y hay que tener en cuenta que un juego también es un problema de programación), es analizarlo con detenimiento. El análisis es la fase más importante y compleja de la resolución de cualquier problema. Un problema bien analizado será mucho más fácil de programar, y

mucho más rápido de codificar y de ejecutar.

Así que lo primero será establecer las reglas del juego. Las reglas del juego de la Serpiente son sencillas:

1. Juguemos en un laberinto de dos dimensiones y de un tamaño adecuado para que quepa en la pantalla definida por nosotros (no habrá scroll).
2. El mapa del laberinto dependerá del nivel de juego, siendo en los niveles más bajos una única sala rodeada de un muro, y apareciendo más obstáculos a medida que el nivel vaya subiendo.
3. El personaje es una serpiente que se moverá por el laberinto según las órdenes del jugador. Este sólo dispone de las teclas del cursor para modificar la dirección en que se mueve la serpiente, y ésta permanecerá moviéndose en la misma dirección mientras el jugador no decida cambiarla. La serpiente nunca podrá detenerse.
4. Siempre existirá en pantalla un objeto comestible. El objetivo del juego es hacer comer a la serpiente tantos de estos objetos como sea posible. Cada vez que la serpiente se coma uno de estos objetos, automáticamente aparecerá uno nuevo. Para hacer el juego más atractivo, a intervalos irregulares de tiempo aparecerán otros objetos comestibles de vida limitada, pero que proporcionan más puntos.
5. Cada vez que la serpiente se come un objeto crecerá.
6. Si la serpiente choca con una de las paredes o con su propio cuerpo muere.

Como hemos comentado, el objetivo es comer tantos objetos comestibles como sea posible. La serpiente crece con cada comida, por lo tanto, la dificultad del juego aumenta a medida que crece la serpiente, puesto que será más difícil encontrar un camino hasta la siguiente comida sin chocar con el propio cuerpo de la serpiente.



Vamos a pensar ahora qué procesos a priori necesitaremos. Está claro por ejemplo que tendremos que crear una clase para "Serpiente" con los datos que definan sus propiedades y con el código necesario para manipularla, pero hay más. Los principales procesos que podríamos definir serían los siguientes:

- Código principal: declaraciones de variables y código para iniciar y manejar el juego completo
- Serpiente: datos y código para manejar la serpiente (tratamiento de sus coordenadas, tratamiento de las direcciones de su movimiento, tratamiento de las secciones de la serpiente...) Se puede dividir en más procesos diferentes más pequeños.
- Laberinto: datos y código para manejar el tablero.
- Tanteo: puntuaciones.

- Comida: manipulación de objetos comestibles.
- Extra: manipulación de objetos comestibles extras de vida limitada.

Si diseccionamos las acciones/eventos que pueden ocurrir en nuestro juego que nos interesará controlar, podrían ser:

- La serpiente puede preguntar al laberinto si el contenido de la celda a la que se va a mover está libre o no.
- La serpiente avanzará una posición en función de la temporización y de las entradas del jugador.
- La serpiente también preguntará si el objeto comida si ocupa una celda concreta antes de ocuparla, y por lo tanto comer su contenido.
- La serpiente será la encargada de colocar un nuevo objeto comida cada vez que se coma el actual.
- El objeto Extra también se sitúa en pantalla, pero en este caso no será la serpiente la encargada de colocarlo, sino el propio juego, ya que no estará siempre en pantalla.
- La comida y el extra consultarán con el laberinto si una casilla está libre, para reponer el objeto comida o extra cuando sea necesario. Las casillas libres serán las que no estén ocupadas por muros, la comida o por la propia serpiente.
- También tendremos que mantener un tanteo, e incrementarlo cada vez que la serpiente coma.

Todo esto es una estructura lógica, poco a poco veremos como implementar este diseño dentro de nuestro programa.

En el código principal almacenaremos los datos globales del juego. Incluiremos aquí los siguientes datos:

- Una variable que indique si el juego está o no detenido.
- Una variable que indique el número de laberintos disponibles. Este dato se puede leer desde un fichero de texto externo, lo que permitirá añadir más laberintos al juego, sin necesidad de compilarlo de nuevo.

Hay varios valores que se pueden también almacenar en un fichero (o en registro de Windows, con la DLL adecuada) para que estén disponibles cada vez que se ejecute el programa, y no se pierdan al cerrarlo.

- El último laberinto seleccionado.
- El último valor de velocidad seleccionado.
- Un array con las puntuaciones máximas obtenidas para cada laberinto.

El código principal puede construir un menú inicial en función del fichero de texto de configuración que contiene el número de laberintos disponibles, y los nombres de cada uno se usarían para cada ítem del menú.

Básicamente, su tarea principal será invocar a todos los demás procesos que componen el juego: laberinto, comida, extra y tanteo.

De forma adicional, este código también puede responder a la orden del usuario cuando desee elegir un laberinto diferente: esto implica actualizar el menú. O también puede responder a la orden del usuario de selección de velocidad (dada por una variable global).

El proceso más sencillo que tendremos será el proceso “Tanteo”, que almacenará los puntos acumulados por el jugador y los actualizará, sumando o restando más puntos, mostrando el valor actual por pantalla.

También se puede añadir contador que podría servir para aumentar el nivel.

Respecto a la comida, sólo nos interesa guardar su posición en el laberinto.

Sólo crearemos un proceso de este tipo: en lugar de destruirlo y volver a crearlo cada vez que la serpiente se lo coma, usaremos siempre el mismo, y modificaremos la posición cada vez.

Cuando una comida sea destruida por la serpiente, automáticamente se volverá a crear, en otras coordenadas.

Respecto los procesos “Extra”, además de la posición en el laberinto, nos interesa saber si está o no activo -es decir, visible- ( y en este caso, lo que falta por activarse, ya que, a diferencia de la comida “normal”, no aparecerán automáticamente justo después de dejar de ser activos). También el tipo de extra que es

De igual manera que las comidas “normales”, sólo crearemos un proceso de este tipo: en lugar de destruirlo y volver a crearlo cada vez, usaremos siempre el mismo modificando su posición cada vez. Pero estas comidas tienen otra diferencia importante con las normales: su periodo de vida es limitado, y el valor en puntos (y en crecimiento) que aporta depende de cuanto vida le quede, cuanto más pronto se coman, más puntos aportan.

En muchos aspectos se comportan como las comidas normales: simplemente se ha de obtener una coordenada libre para ella del laberinto. Pero a partir de ese momento el tiempo que le queda por permanecer en el laberinto empieza a disminuir. Además, mientras no hay ningún extra presente, también tendremos que disminuir otro contador que indica el tiempo que queda para que aparezca otra comida extra.

Para eso usaremos el mismo temporizador que para el movimiento de la serpiente: así, además de actualizar la posición de la serpiente, decrementaremos el tiempo que le queda al extra para aparecer, si no está presente, o para desaparecer, si lo está.

Además, mientras el tiempo corra, podemos hacer que el gráfico de la comida extra cambie, mostrando así que el tiempo se acaba. No dispondremos de gráficos para todos los estados intermedios del temporizador durante la presencia del extra en pantalla, de modo que sólo cambiaremos el gráfico cada cuatro estados, por ejemplo. Si creamos 16 gráficos, eso proporcionará un total de 64 estados. Cada estado corresponderá con un movimiento de la serpiente.

Tampoco queremos que los extras aparezcan demasiado a menudo, su función es no dejar al jugador organizarse demasiado, de modo que el juego no se convierta en algo aburrido. Los extras introducen nuevos objetivos que además tienen más preferencia, ya que dan más puntos y hay un tiempo limitado para conseguirlos. Eso obliga a cambiar las trayectorias lógicas por otras más cortas, y aumenta la dificultad. A veces puede ser mejor dejar pasar un extra que intentar conseguirlo, y por ese motivo no debemos hacer que sean muy frecuentes. Si el jugador sabe que pronto habrá otro extra, la prioridad por conseguirlo disminuirá.

Los tiempo para que aparezcan nuevos extras serán aleatorios, pero siempre estarán entre un máximo no demasiado alto y un mínimo no demasiado bajo. En nuestro caso entre 150 y 200 estados, pero eso puede cambiarse fácilmente.

En el proceso “Laberinto” almacenaremos diferentes datos sobre el tablero.

Concretamente un array con los valores de cada casilla indicando si hay en ella muro, comida,

extra o serpiente. Puede que te sorprenda que sea el proceso "Laberinto" el que detecte qué casillas está ocupando la serpiente en cada momento. Posiblemente habrás pensado inicialmente en hacer que sea la serpiente quien diga si ocupa o no una casilla concreta, pero de ese modo estaría obligada a consultar las coordenadas de todas sus secciones (cabeza, cuerpo horizontal, cuerpo vertical, ángulo, cola...), y esto también se usaría tanto para obtener una casilla libre para situar la comida, como para verificar si la serpiente ha chocado consigo misma. Esta forma de trabajar tiene consecuencias no deseadas, ya que esas consultas dependerán de la longitud de la serpiente, y pueden afectar a la velocidad con la que se desarrolla el juego. De modo que haremos que esa tarea la desarrolle el proceso "Laberinto". Cada vez que actualicemos la posición de la "Serpiente", añadiendo o eliminando secciones, lo notificaremos al proceso "Laberinto". Lo mismo harán los procesos "Comida" y "Extra", y será el "Laberinto" el que se encargará de mantener esa información en una matriz. Los procesos "Serpiente", "Comida" y "Extra" consultarán con "Laberinto" para saber el contenido de una casilla, y como esta consulta se hará a través de una matriz, el tiempo de respuesta será constante y rápido.

En este proceso también utilizaremos una variable entera con el número de casillas libres que quedan. Ese último dato se usará para obtener una posición libre del tablero para situar una comida o un extra.

También incluimos un array para almacenar la forma del laberinto y mostrarla por pantalla

Y una variable para almacenar el número del laberinto actualmente cargado.

Finalmente, se ha de implementar la funcionalidad que permita consultar el contenido de una celda concreta del laberinto, otra para cambiar el contenido actual de una casilla, y otra para obtener las coordenadas aleatoriamente de una casilla que no esté ocupada. El algoritmo de esto último sería el siguiente:

- Obtenermos un valor aleatorio 'v' entre 0 y el número de casillas libres.
- Hacemos un barrido a lo largo del laberinto ignorando las casillas ocupadas, hasta encontrar la libre que hace el número 'v'.
- Devolvemos el valor de esa casilla.

Además, se necesitará cargar un laberinto nuevo cuando el usuario lo elija.

Finalmente llegamos al proceso más complicado: "Serpiente".

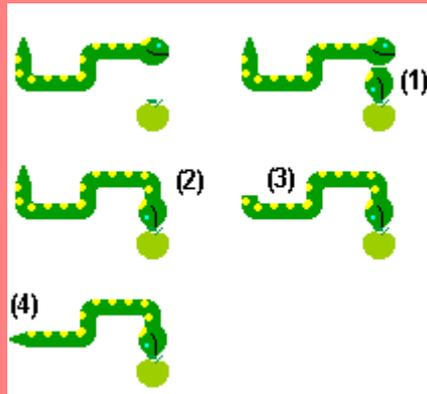
Para la serpiente necesitamos mantener varios valores importantes, que nos permiten saber en cada instante dónde se encuentra cada una de sus partes, en qué dirección se mueve, si está creciendo, y qué longitud le queda por crecer, etc.

Nuestra serpiente estará "cuantificada", es decir, cada vez que se mueva lo hará en una unidad, y cada vez que crezca también lo hará en una unidad. La serpiente también tendrá siempre un número entero de unidades.

Para cada sección de la serpiente guardaremos un conjunto de datos en una estructura:

- Cada vez que se mueva grabaremos en la estructura la siguiente posición de la cabeza de la serpiente.
- Si la serpiente no está creciendo, en cada movimiento eliminaremos una sección del extremo de la cola de la serpiente, con lo que grabaremos la nueva posición final de ésta en la estructura.
- Si la serpiente está creciendo, sencillamente no grabamos ninguna nueva posición final, de modo que realmente crecerá.

Analizando el comportamiento de la serpiente veremos que en cada movimiento no es necesario volver a pintarla toda completa. Supongamos que tenemos la serpiente como en el gráfico siguiente, y el jugador pulsa la tecla "abajo" para dirigirse a la comida.



Sólo necesitamos pintar la cabeza en su nueva posición (1), borrarla de la posición anterior, sustituyéndola por el dibujo de una sección del cuerpo (2) y, si es necesario, borrar la cola (3) y sustituir la que ahora es la nueva sección de cola por el gráfico correspondiente (4).

Esto nos será muy útil, ya que el tiempo necesario para pintar la serpiente en cada movimiento es independiente de su longitud, y por lo tanto predecible, y prácticamente constante.

Resumiendo, para toda la serpiente, guardaremos los siguientes valores:

- Dirección actual del movimiento.
- Dirección "deseada" por el usuario.
- Unidades que tiene que "Crecer" todavía.
- "Coordenadas" en el laberinto.
- Una estructura con las posiciones de las secciones y su tipo (cabeza, cola, sección recta vertical u horizontal, o sección de ángulo) de la serpiente.

El código del proceso "Serpiente" no ha de hacer demasiadas cosas. Ha de:

1. Verificar si el movimiento elegido por el usuario es legal o no. Recordemos que la serpiente no puede invertir su dirección.
2. Modificar el tipo de sección de la cabeza actual, que pasará de ser una sección de "cabeza" a ser sección de cuerpo, y dependiendo de la dirección actual de movimiento, podrá ser una sección recta o de ángulo.
3. Calcular la próxima posición de la cabeza. Para ello tomará la posición actual de la cabeza, y

dependiendo de la dirección de movimiento, actualizará el valor de la coordenada x o y según corresponda.

4. Comprobar si ha habido una colisión, ya sea con un muro o con la propia serpiente.
5. Insertar una nueva sección de cabeza en la estructura de secciones.
6. Comprobar si hemos llegado a una posición con comida. En ese caso tendremos que incrementar el valor que todavía tiene que crecer la serpiente, colocar una nueva comida en el laberinto.
7. Si la serpiente está creciendo, decrementamos el número de unidades que tiene que crecer todavía e incrementamos el contador. En caso contrario, eliminamos la última sección de la estructura de secciones, y actualizamos el valor de la actual última sección como sección de "cola".

Cada actualización, borrado o inserción de sección en la estructura implica una actualización de dicha sección en pantalla y en el laberinto, claro.

Sería recomendable implementar también un sistema que cree una serpiente nueva cada vez que se muera la actual, dando comienzo a una partida nueva.

Para gestionar el proceso de movimiento, redireccionamiento y repintado del gráfico de la serpiente, partiremos de un laberinto que puede ser vacío y que el usuario podrá elegir en función de la dificultad, asignaremos a la serpiente una posición de salida y colocaremos comida en un punto del laberinto determinado. Este será el único momento en el que la serpiente podrá estar parada, hasta que el usuario pulse una tecla.

A partir de ese momento, y a intervalos regulares de tiempo, que dependerán también del nivel de juego, se actualizará la posición de la serpiente.

Podemos crear un temporizador que actualice la posición, por ejemplo, cada 200 milisegundos, independientemente de lo que el usuario decida. En nuestro caso, el intervalo del temporizador dependerá del valor de velocidad elegido.

El algoritmo de actualización será más o menos este:

[Inicio]

<Serpiente en movimiento?>

Sí:

<Choque?>

Sí:

[Detener serpiente]

No:

<Es un objeto comestible?>

Sí:

[Incrementar unidades pendientes de crecer]

[Crear nuevo objeto comestible]

[Avanzar Serpiente]

[Salir]

Otra tarea que debemos atender es la encargada de leer el teclado y calcular la nueva dirección de movimiento. Para ese cálculo debemos tener en cuenta que no es posible invertir la dirección de movimiento de la serpiente de golpe. Es decir, si la serpiente se mueve hacia arriba o hacia abajo, los movimientos posibles sólo serán hacia izquierda o derecha, o bien permanecer en la dirección actual. Lo mismo si la serpiente se mueve hacia la derecha o izquierda, los movimientos posibles serán arriba, abajo o

permanecer en la dirección actual.

Esto nos obliga a tener dos variables para determinar el movimiento, una con el valor previo y otra con el próximo. Cada vez que se actualice la posición de la serpiente, copiaremos el valor actual en el previo, pero sólo si el valor actual es válido. Según eso, podemos usar una variable para almacenar la última opción del usuario, que incluso puede cambiar varias veces entre movimiento y movimiento, de modo que sólo nos quedamos con la última opción.

Esto complica un poco la rutina de calcular la próxima posición de la cabeza, ya que se debe comprobar si la dirección deseada es legal o no. Sin embargo, facilita el cálculo del tipo de sección que corresponde a la cabeza anterior, ya que disponemos de las direcciones anterior y actual.

---

Hemos hablado algo más arriba de las secciones de la serpiente y de que pueden ser de varios tipos. Vamos a concretar algo más este tema. Nuestra serpiente sólo puede desplazarse en cuatro direcciones, por lo tanto los giros siempre serán de noventa grados. Esto limita la secciones posibles a cuatro en ángulo, una vertical, una horizontal, cuatro posibles cabezas y cuatro colas. En total catorce secciones diferentes.

Veremos que los cálculos se simplifican mucho si optamos por codificar las direcciones y las secciones usando una lógica binaria. Por ejemplo, para cada dirección usamos un bit diferente, el primero para arriba, el segundo para la derecha, el tercero para abajo y el cuarto para la izquierda. Es decir, los valores serían 1, 2, 4 y 8, respectivamente.

O expresado en binario:

1: 0001  
2: 0010  
4: 0100  
8: 1000

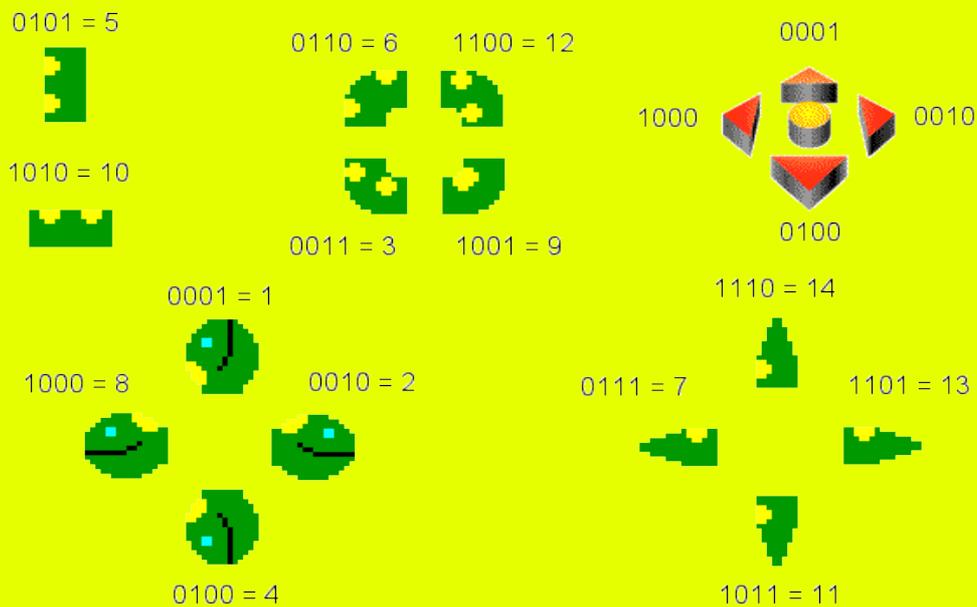
De este modo, para codificar una sección vertical usaremos los bits correspondientes a las direcciones arriba y abajo, es decir  $1+4=5$ , para una sección horizontal los bits de las direcciones izquierda y derecha,  $2+8=10$ .

1: 0001  
4: 0100  
5: 0101  
  
2: 0010  
8: 1000  
10: 1010

Para el resto de las secciones haremos igual, la sección izquierda y abajo tendría el valor  $4+8=12$ , etcétera.

Cada sección curva tendrá un nombre dependiendo de la sección de circunferencia a que

corresponda, la primera será de izquierda-abajo, la segunda la de arriba-izquierda, la tercera la de arriba-derecha y la cuarta la de abajo-derecha. Las llamaremos seccion1, seccion2, seccion3 y seccion4, respectivamente.



Para las dos secciones restantes: cabeza y cola, usaremos los demás valores posibles mediante suma de direcciones, por ejemplo, 1, 2, 4 y 8 para la cabeza. Para la cabeza mantendremos la misma lógica de direcciones. Para la cola usaremos los valores 7, 11, 13 y 14, usando lógica inversa, es decir, codificando la dirección con ceros en lugar de con unos.

Para las direcciones definiremos un vector siguiendo la misma lógica.

Así, si tenemos que determinar el tipo de una sección, sólo tenemos que sumar los valores de la dirección de procedencia y de la nueva dirección. Si la serpiente venía de abajo y gira a la izquierda, sumaremos los valores de abajo e izquierda:

$4+8=12$ , es decir, la Seccion1.

Pero cuidado, porque en este cálculo hemos usado la dirección de procedencia, y no la previa, que en este ejemplo sería arriba. Por lo tanto, para obtener el valor de una sección con los datos que tenemos hay que realizar algunos cálculos:

El valor de dirección contrario al previo más el valor de dirección actual.

Para calcular el valor contrario de una dirección usaremos dos rotaciones de bits, por ejemplo, para la dirección arriba, que tiene el valor 0001, la dirección contraria es  $0001 \ll 2 = 0100$ , es decir: abajo. Cuando se trate de valores de dirección 1 ó 2 rotaremos a la izquierda, cuando sean 4 u 8, a la derecha.

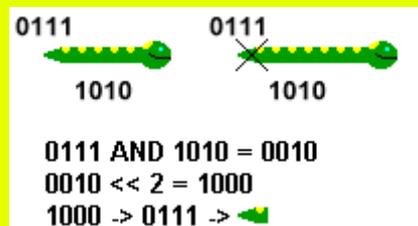
El cálculo de las nuevas secciones de la cola es algo más complicado. Por ejemplo,

supongamos que la serpiente termina con una sección recta y una de cola hacia abajo. En un momento dado eliminamos la sección de cola, y la última sección de la serpiente es ahora recta, por lo tanto tendremos que calcular el nuevo valor de la sección de cola basándonos en el valor actual de la última sección y el de la sección anterior a esa.

La cola y la cabeza sólo codifican una dirección, y cualquier otra sección codifica dos. Lo primero que haremos será aplicar una función and de bits entre los dos valores. Esto nos da la dirección que tienen en común las dos secciones.

De ese valor obtenemos la dirección contraria aplicando el algoritmo que hemos visto antes, y el resultado lo complementamos a 1, es decir, sustituiremos los unos por ceros y los ceros por unos, para obtener el valor de la sección de cola correspondiente a la dirección obtenida. Esto es tan sencillo como restar el valor de 15.

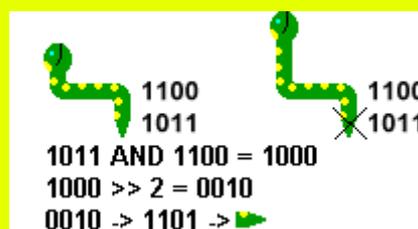
Veamos un ejemplo:



El valor de la sección de cola que vamos a eliminar es 0111. Si invertimos los bits de ese valor obtenemos 1000, que es el valor de la dirección izquierda.

El valor de la sección que pasará a ser cola es 1010, que codifica las direcciones izquierda y derecha. Aplicando la función AND obtenemos el valor 0010, es decir la dirección derecha. El valor contrario es 1000, de nuevo izquierda. Y el complementario es 0111, que es el valor de cola adecuado.

En el ejemplo anterior parece que nos complicamos la vida en exceso, veamos otro ejemplo:



Ahora, el valor de la sección de cola que vamos a eliminar es 1011. Si invertimos los bits de ese valor obtenemos 0100, que es el valor de la dirección abajo.

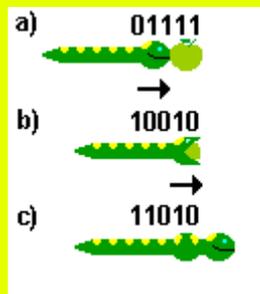
El valor de la sección que pasará a ser cola es 1100, que codifica las direcciones izquierda y abajo. Aplicando la función AND obtenemos el valor 1000, es decir la dirección izquierda. El valor contrario es 0010, o sea derecha. Y el complementario es 1101, que es el valor de cola adecuado.

Podríamos haber hecho esto mismo con tablas que codificaran todos los casos posibles, pero considero que siempre es mejor calcular los valores que memorizarlos. En este caso se trata de operaciones sencillas: sumas, restas y operaciones lógicas, y se trata de cálculos muy rápidos.

Pero además, nos interesa añadir algunos refinamientos gráficos para dar más vistosidad al juego. Cuando la serpiente coma algo, podemos hacer que se note un abultamiento en la coordenada en la que estaba la comida. Esto no requiere demasiados cálculos.

En lo que se refiere a los gráficos, bastará con añadir un quinto bit. La presencia de ese bit indica que en esa sección hay abultamiento, o en el caso de la cabeza, que la boca está abierta. La ausencia del bit indica que es una sección normal.

Por lo tanto, cuando calculemos una nueva sección tendremos en cuenta que el quinto bit se "hereda", tanto cuando sea la segunda sección como cuando calculemos la cola, que son los únicos casos en los que modificaremos los códigos de una sección. Por ejemplo:



En la transición de (a) a (b), para calcular la sección correspondiente tomamos el valor de la dirección, 2, y como se trata de una casilla con comida le añadimos el quinto bit, es decir 10010.

En la transición de (b) a (c), la dirección sigue siendo derecha, y el valor previo de la sección es 10010. Ya vimos que se toma el valor de la dirección contraria a la previa más el valor de dirección actual.

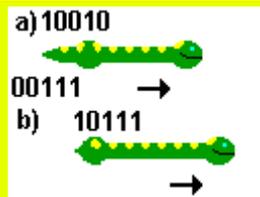
Previamente recordamos el valor del quinto bit del valor de la sección actual, eso se hace comparando ese valor con 16 ó 0x10.

El valor de dirección previa coincide con los cuatro bits de la derecha del valor previo de la sección, es decir  $10010 \text{ AND } 01111 = 00010$ .

El valor contrario es 01000, y esto sumado con el valor de dirección es:  $01000 + 0010 = 01010$ .

Por último, añadimos el valor del quinto bit que hemos memorizado  $10000 + 01010 = 11010$ .

Otro ejemplo:



En la transición de (a) a (b), para calcular la sección correspondiente tomamos el valor de la penúltima sección y el de la sección de cola. La sección de cola se elimina, y para la otra calculamos un nuevo valor.

El valor de la sección de cola que vamos a eliminar es 00111. Si invertimos los bits de ese valor, prescindiendo del quinto, que no nos interesa de momento, obtenemos 1000, que es el valor de la dirección izquierda.

El valor de la sección que pasará a ser cola es 11010, que codifica las direcciones izquierda y derecha, y además indica que se trata de una sección ancha. Aplicando la función AND obtenemos el valor 00010, es decir la dirección derecha. El valor contrario es 01000, de nuevo izquierda. Y el complementario es 00111, que es el valor de cola adecuado.

Como previamente el valor de esa sección tenía el quinto bit a uno, conservamos ese bit, y se lo añadimos al valor calculado,  $10000 \mid 00111 = 10111$ .

Por último, comentar que la parte del diseño gráfico es una de las más complicadas para el programador, ya que la mayoría de juegos son creados por especialistas y muy rara vez por los mismos programadores.

Afortunadamente nuestro juego no es muy complicado, y el repertorio de gráficos es muy limitado. De modo que podemos atrevernos a diseñar unos gráficos sencillos, aunque puedan resultar toscos.

Necesitamos los gráficos de las cuatro secciones circulares, la vertical, la horizontal y cuatro versiones de cabeza y otras cuatro de cola, según las cuatro orientaciones posibles.

Además, debemos añadir un gráfico para la sección vacía, otro para la comida, y otro para los muros del laberinto.

Para dar más vistosidad al juego, se puede añadir versiones de todas las secciones de la serpiente que simulen que hay comida en su interior. Esto no complicará mucho el programa, y le añadirá vistosidad. En cuanto a la codificación, simplemente usaremos un quinto bit para indicar la presencia de comida en el interior de la serpiente, o lo que es lo mismo, sumaremos 16. Por ejemplo, si la sección de cabeza apuntando arriba es la número 1, la sección equivalente con comida es la 17. Expresado en binario sería: 00001 y 10001. Lo mismo pasa con todas las secciones.

Además, esos números son los mismos que el número de orden dentro del mapa de bits, de modo que como cada gráfico es de 16x16 bits, para obtener la coordenada x de una sección cualquiera, bastará

con multiplicar su valor por 16.



También tendremos que diseñar gráficos para los extras. Podríamos usar una fila para cada extra, y 16 versiones de cada uno dependiendo del tiempo. Esto nos permite añadir fácilmente más extras, si queremos.

Es muy sencillo obtener las coordenadas de un gráfico cualquiera, la coordenada x corresponde con el tiempo, siempre multiplicando por 16, ya que los gráficos siguen siendo de 16x16 bits. La coordenada y depende del gráfico elegido, también multiplicando por 16.



Además, añadiremos gráficos para los dígitos del contador. Seguiremos usando un único mapa de bits para todos los dígitos, también de 16x16 bits.



## **EPÍLOGO**

(extraído de la carta abierta de Emilio de Paz, <http://www.alcachofasoft.com>)

A los desarrolladores independientes:

¿Por qué hacemos videojuegos?

No se trata de “Hacer el mejor motor del mercado”. Ni se trata de “Estamos haciendo un juego que va a vender un montón”. No se trata de “Sorprender a los demás con la idea que a nadie se le ha ocurrido”. Aquí se trata de “¿Qué hay que usar con la polea para subir el cubo?”. Aquí hablamos de “¡Mierda! ¡Me han matado! Voy a intentarlo otra vez, ahora no me va a pasar”.

En muchas ocasiones, nosotros mismos, los creadores de videojuegos, no somos conscientes de lo que estamos haciendo. El proceso de desarrollo nos abarca y, estamos tan inmersos en él, que quizá no nos damos cuenta del aspecto global. ¿Por qué hacemos videojuegos? ¿De dónde sale ese impulso de CREAR?

Miremos hacia atrás, a nuestra infancia, adolescencia o juventud, y pensemos en los primeros juegos que cayeron en nuestras manos. Los cargamos en nuestros viejos ordenadores y nos pusimos a jugar con ellos... ¿Qué sentíamos? No era “Qué bien optimizada está la memoria RAM”. No era “¿Cómo habrán conseguido el naranja en un Spectrum?”. Era el paso a un mundo mágico. Era Alicia a través del espejo. Es el mundo de las sensaciones, de las emociones. De vernos atrapados por un juego. Sin más. Sin “El personaje tiene más polígonos que el de Tomb Raider”. Sin “Las sombras molan un mazo porque se arrojan a tiempo real”. Sin “Vamos a distribuirlo en Francia, Alemania y Djibuti”.

Sí, los videojuegos son un medio, cada día más cercano al cine. Nos cuentan una historia, nos atrapan en su mundo con sus gráficos, su sonido, su ambientación... Pero hay algo que ningún otro medio tiene, al menos en tan gran medida: la INTERACCIÓN. Y es que en aquellos videojuegos, sin importar el tipo de historia, o el género del juego... NOSOTROS éramos el protagonista. Y la historia podía terminar bien o mal, todo dependía de nosotros. No había un final pre-fijado, no éramos simples espectadores: vivíamos una aventura. Era JUGAR, con mayúsculas.

¿Cuántas veces en estos años nos hemos sentido un científico metido a Rambo, un aprendiz de pirata o de mago, un piloto de carreras, o incluso una gran boca amarilla que cuando se come unos puntos se encabrona y se zampa al primer fantasma que pilla? ¿Y acaso no hemos sentido alegría cuando hemos triunfado? ¿Tristeza cuando hemos fracasado? Júbilo, Odio, Rencor, Frustración, Humor, Amor... Cuando éramos “profanos”, cuando no teníamos ni idea de lo que era un píxel, un polígono o un puntero, un juego era para nosotros, conscientemente o no, una fuente de EMOCIONES.

¿Por qué hacemos videojuegos? ¿De dónde sale ese impulso de CREAR? En el fondo todos sabemos la respuesta: queremos hacer sentir al resto de la gente las emociones que nosotros vivimos cuando jugamos a un videojuego. Porque, hora es ya de decirlo, la creación de videojuegos es un ARTE, y por lo tanto nosotros somos unos ARTISTAS. Y como artistas que somos llevamos a la gente, en forma de CD, emociones fantásticas y maravillosas.

¿Qué es la “adicción”, la “jugabilidad”? ¿Acaso el que una persona no quiera dejar de jugar a nuestro juego, o no deje de pensar en el puzzle que le hemos puesto, no evidencia que esas sensaciones le llegan de la forma más directa posible? No es “ver a un piloto de carreras” es SER un piloto de carreras. No es “¿encajarán las piezas?” es ENCAJAR las piezas. No es “A ver cómo sale el bueno de ésta” es ¡¡¿Cómo SALGO de ésta?!!

Y nuestra profesión por tanto, de programadores, diseñadores, músicos, grafistas, directores... de ARTISTAS al fin y al cabo, es transmitir esas emociones. No es, como muchos creen, “Soy un freaky que quiere demostrar que es el más listo”. No es, como nosotros quisiéramos, “Voy a hacerme rico con este negocio tan moderno”.

Hacemos felices a los jugadores creando emociones para ellos. Lo que nosotros hacemos es algo MUY IMPORTANTE. Somos ARTISTAS, haciendo un ARTE. Un arte nuevo, que tiene un mercado que crece, que tiene un futuro espléndido, que muy poca gente es capaz de realizar...

Desde aquí mi más sincero agradecimiento a todos los creadores por haberme EMOCIONADO con sus juegos. El cine ha evolucionado mucho desde los hermanos Lumière (1895, año que se inventó el cinematógrafo). Nuestro arte nació hace poco más de 30 años. Tenemos mucho camino por delante y será un placer recorrerlo juntos.

## **ÍNDICE**

Agradecimientos.....	3
A quién está dirigido este texto.....	3

### **CAPÍTULO 0: ANTES DE EMPEZAR**

#### **Conceptos básicos de programación**

¿Qué es programar?.....	4
Los lenguajes de programación.....	4
Editores, intérpretes y compiladores.....	8
Las librerías.....	9
Las librerías gráficas.....	11
Lenguajes de programación de videojuegos.....	15
Pero...¿qué es eso de “software libre”?.....	17
Algunos recursos web sobre programación de videojuegos.....	20

#### **Conceptos básicos de multimedia**

Gráficos	
Profundidad de color.....	21
Resolución.....	22
Peso.....	22
Formato.....	22
Gráficos vectoriales.....	22
Mapa de bits.....	23
Vídeo	
Procesamiento del video digital.....	26
Formatos del video digital.....	26
Códecs.....	27
3D	
¿Qué es animar?.....	28
Proceso de animación 3D.....	29
La aceleración hardware.....	30
Sonido	
El proceso de digitalización.....	31
Digitalización de alta y baja calidad .....	32
Formatos de almacenamiento.....	33
Ficheros.....	33
¿Qué es MIDI?.....	34
MIDI y sonido digital.....	34
Aparatos MIDI.....	34
Las tarjetas de sonido.....	35
Lista de algunos programas útiles (para Windows).....	36

### **CAPITULO 1: PRIMEROS PASOS CON FENIX**

¿Qué es Fénix? Propósito y características.....	45
Historia de Fénix.....	46
Críticas y carencias de Fénix 0.92a.....	46
Obtención, instalación y funcionamiento de Fénix.....	48
La versión 0.93preview9 de Fénix.....	49
Tu primer programa en Fénix.....	50
El IDE FlameBird2.....	52
Otros programas importantes: FPGEdit y FNTEdit.....	53
El FenixPack.....	53

Sobre el uso de la ayuda de Fénix.....	54
Recursos web sobre Fénix.....	54
La distribución de tus programas.....	56
Y de regalo, tu segundo programa en Fénix.....	59

## CAPITULO 2: EMPEZANDO A PROGRAMAR CON FENIX

### Explicación paso a paso de “Mi 1r programa en Fénix”

Palabra reservada “Program”.....	60
Comentarios.....	60
Bloque “Private/End”.....	61
Línea “int mivar1”.....	62
Bloque “Begin/End”.....	65
Línea “mivar1=10;”.....	65
Bloque “While/End”.....	66
Orden “delete_text”.....	67
Línea “mivar1=mivar1+2;”.....	68
Orden “write”.....	69
Orden “frame”.....	71
Funcionamiento global del programa.....	72
Notas finales.....	74

### Explicación paso a paso de “Mi 2º programa en Fénix”

Bloque “Loop/End”. Sentencias BREAK y CONTINUE.....	75
“mivar1=rand(1,10);”. Valores de retorno de las funciones.....	78
Bloque “If/Else/End”. Condiciones posibles.....	81
Funcionamiento global del programa.....	83

### Otros bloques de control de flujo: Switch, For, From, Repeat

Bloque “Switch/End”.....	84
Bloque “For/End”.....	85
Bloque “Repeat/Until”.....	89
Bloque “From/End”.....	91
Profundizando en la escritura de textos en pantalla.....	92
Algunos ejemplos más de códigos fuente.....	96

## CAPITULO 3: INCLUSIÓN DE GRÁFICOS

Los modos de pantalla.....	103
Configuración de los frames per second.....	105
Concepto de FPG. La aplicación “FPGEdit” y la utilidad fpg.exe.....	108
Carga de la imagen de fondo.....	110
Descarga de imágenes.....	113
Iconos.Set_icon. Editores de iconos y recursos.....	116
Los nuevos formatos: FBM,FGC y FPL.....	118
Uso de paletas de colores.....	119

## CAPITULO 4: DATOS Y PROCESOS

Concepto de proceso.....	121
Las variables locales predefinidas FILE,GRAPH,X e Y.....	122
Creación de un proceso.....	123
Variables globales,locales y privadas.....	127
Ámbito de las variables.....	130

Constantes.....	136
Parámetros de un proceso.....	137
Código identificador de un proceso. La variable local predef. ID y “Get_id()”.....	139
Jerarquía de procesos.....	143
Señales entre procesos. Comando “Signal()”.....	145
Variables públicas y sentencia “Declare”.....	150
Los comandos “Let_me_alone()” y “Exit()”.....	152
El comando “Exists()”.....	156
La cláusula ONEXIT.....	157
Sobre la ejecución paralela de los procesos. La variable local predef. PRIORITY...158	
El parámetro de la orden Frame.....	162
Concepto y creación de una.....	163
La sentencia #DEFINE.....	170

## **CAPITULO 5: TABLAS Y ESTRUCTURAS**

Tablas.....	173
Estructuras y tablas de estructuras.....	179
La función “Sizeof()”.....	182
Los tipos definidos por el usuario.TYPE.....	184
Copia y comparación entre 2 tablas o estructuras.Gestión básica de memoria.....	186
Pasar un vector o un TDU como parámetro de un proceso/función.....	191

## **CAPITULO 6: TUTORIAL PARA EL JUEGO DEL LABERINTO**

Imágenes y transparencias.....	195
El juego básico. Collision().....	196
Añadiendo giros al movimiento del personaje Advance() y ANGLE.....	199
Inclusión de múltiples enemigos diferentes. La variable predef. Local SIZE.....	200
Añadiendo explosiones.....	203
ANGLE,SIZE y otras var. locales predef.:SIZE_X,SIZE_Y,ALPHA y FLAGS.....	205

## **CAPITULO 7: TUTORIAL DE UN PING-PONG.....212**

Variante: El juego del picaladrillos.....	226
---	-----

## **CAPITULO 8: TUTORIAL PARA UN MATAMARCIANOS**

Punto de partida.....	229
Añadiendo los disparos de nuestra nave.La variable local predefinida Z.....	230
Añadiendo los enemigos.....	236
Eliminando procesos,matando enemigos.....	239
Añadiendo explosiones.....	245
Añadiendo energía enemiga.....	246
Añadiendo energía nuestra y su correspondiente barra gráfica. Uso de regiones.....	247
Disparo mejorado y disparo más retardado.....	251
Disparos de los enemigos.....	254
Añadiendo los puntos. Introducción al uso de fuentes FNT.....	254
Introducción al uso del ratón.....	256
Introducción al uso de scrolls de fondo.....	257
Introducción al uso de bandas sonoras y efectos de sonido.....	260

## CAPITULO 9: FUNCIONES Y ESTRUCTURAS BÁSICAS DEL LENGUAJE

Trabajar con temporizadores.....	261
Trabajar con ficheros.....	267
Cómo grabar partidas y reproducirlas posteriormente.....	283
Trabajar con cadenas.....	289
Trabajar con fechas.....	298
Funciones matemáticas .....	300
Trabajar con gráficos generados dinámicamente en memoria.....	303
Trabajar con gráficos FGC.....	340
Dibujar primitivas gráficas.....	347
Trabajar con el teclado (aceptar cadenas de caracteres).....	357
Trabajar con el teclado (cómo redefinirlo).....	365
Trabajar con el ratón.....	366
Trabajar con sonido.....	380
Trabajar con música.....	385
Trabajar con ángulos y distancias.....	389
Trabajar con fades y efectos de color.....	414
Trabajar con puntos de control.....	423
Las fuentes FNT. La aplicación “FNTEdit”.....	431
Trabajar con regiones.....	439
Trabajar con scrolls y Modo7.....	443

## CAPITULO 10:TUTORIAL PARA UN RPG BÁSICO

La pantalla de inicio.....	470
Creando el mundo y nuestro protagonista moviéndose en él.....	475
El mapa de durezas.....	478
Entrar y salir de la casa.....	482
Conversaciones entre personajes.....	486
Entrar y salir de la segunda casa.....	494
La pantalla del inventario.....	496
Guardar partidas en el disco duro y cargarlas posteriormente.....	506
Añadiendo enemigos, medida de vida y GameOver.....	509
Concepto y utilidad de los “tiles”.....	513
Includes.....	514

## CAPITULO 11:TUTORIAL PARA UN JUEGO DE PLATAFORMAS.....517

## CAPITULO 12: TUTORIAL PARA EL JUEGO “SANTA ATTACK”

La intro del juego.....	535
Inicio del juego.....	535
La nave.....	536
Los propulsores.....	537
Los regalos.....	538
Las casas y las chimeneas.....	539
Código fuente completo.....	539

## CAPITULO 13: TUTORIAL PARA UN TETRIS

Antes de empezar.....	550
-----------------------	-----

Representación del área del juego.....	551
Representación de las figuras.....	552
Evolución del juego.....	553
Colisiones entre límites del área de juego.....	554
Colisiones entre piezas.....	556
Cómo averiguar si hemos hecho línea.....	557
En resumen.....	558
Código fuente de ejemplo totalmente funcional.....	558

## **CAPITULO 14:ARTICULOS,CODIGOS Y TRUCOS VARIADOS**

Forzar a los procesos a realizar trayectorias concretas.....	574
Ordenar vectores y estructuras.....	577
Creación de combos de teclas.....	588
Trabajar con tablas Blendop.....	591
Utilizar los operadores binarios.....	604
Opciones de depuración.....	606
Introducir valores a nuestro programa por línea de comandos.....	611
Cómo desarrollar nuestro propio formato de imagen.....	613
Cómo controlar muchos individuos.....	616
Miscelánea de algunas funciones más de Fénix.....	619
Códigos de un picaladrillos y un matamarcianos sin usar ninguna imagen externa.....	625
Código para crear explosiones un poco más espectaculares.....	631
Código para generar llamas de fuego (azul).....	631
Código que ondula una imagen.....	633
El juego del “disparaletas”.....	635
Simulación de la escritura de una antigua máquina de escribir.....	639
Pequeño tutorial teórico para el juego de la serpiente.....	640

<b>EPILOGO.....</b>	<b>653</b>
---------------------	------------