Universidad Complutense de Madrid
Falcutad de Informática

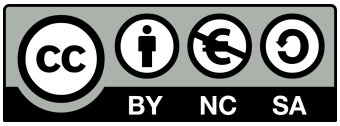
Azimuth: diseño y desarrollo de un videojuego no euclídeo

Trabajo de fin de grado
Doble Grado en Ingeniería Informática y Matemáticas

Autor:
Jose Pablo Cabeza García

Director:
Dr. Marco Antonio Gómez Martín

Septiembre 2015



Azimuth: Diseño y desarrollo de un videojuego no euclídeo

Jose Pablo Cabeza García

Resumen

El desarrollo de un videojuego no euclídeo necesita de unos fundamentos matemáticos sólidos con los que diseñar e implementar la arquitectura de un motor de juego. En esta memoria se describe la arquitectura e implementación de un motor no euclídeo sobre Unity, sin entrar en detalles del modelo matemático subyacente.

Palabras clave: geometría diferencial, geometría no euclídea, desarrollo de videojuegos, Unity, informática gráfica

Azimuth: Design and Development of a Non-Euclidean Videogame

Jose Pablo Cabeza García

Abstract

Developing a non-Euclidean videogame needs solid mathematical foundations to build and design a game engine. This report describes the architecture and implementation of a non-Euclidean game engine in Unity, without getting into details of the mathematical model.

Keywords: differential geometry, non-Euclidean geometry, videogame development, Unity, computer graphics

Índice general

Capítulo 1

Introducción

Existen muchos juegos que dicen no ser euclideos y romper las reglas del espacio, pero al final siempre se basan en motores convencionales que se comportan y mueven *como lo esperaríamos*. Estos juego estan basados en coser el espacio con bucles y eventos que no existen en el mundo real, pero que en última instancia no rompen las reglas del espacio.

Frente a esto, mis compañeros, Alejandro Aguirre Galindo, Paco Criado Gallart, y yo nos embarcamos en el proyecto de crear un juego que fuese genuinamente no eucídeo. Este proyecto comenzó hace casi dos años como un embrión de idea que podría explotarse hasta llegar al día de hoy, en el que ya hay resultados visibles.

En un primer paso se desarrolló, sobre todo por parte de Paco Criado, un modelo de cómo funciona la geometría no euclídea y de todas propiedades que tiene. Esto cristalizó en unas primeras implementaciones que se fueron refinando hasta conseguir crear una manera de modelar y proyectar el espacio eficiente y elegante.

Con este modelo matemático diseñamos una arquitectura que lo cumpliese y que además fuese lo suficiente eficiente como para poderse usar en el desarrollo de un videojuego. Esto nos llevó a enfrentarnos a distintos problemas que normalmente no se darían en un juego normal y que tuvimos que solventar. Finalmente creamos e implementamos una arquitectura con la que pudimos diseñar y jugar a algunos puzles que verdaderamente son no euclídeos.

El desarrollo del motor ha sido realizado sobre todo por Alejandro Aguirre y yo, encargándose mi compañero de la parte de representar en Unity el modelo que diseñamos y yo más de deseñar la arquitectura y cómo implementar el modelo matemático.

Capítulo 2

Nociones preliminares

2.1. Videojuegos de referencia

Previamente a desarrollar y diseñar el motor, se hizo un estudio de los videojuegos que explotaban el uso del espacio de una manera no estándar para saber lo que ya se había realizado antes y cómo se había hecho.

El concepto de espacio no euclídeo no se suele usar en los videojuegos, y cuando se usa es de manera poco rigurosa, pero hay algunos juegos que hacen uso de estructuras que no son posibles con las reglas normales del espacio, de forma que pueden considerarse en cierta manera de espacio no euclídeo, aunque el comportamiento del movimiento y la física sea estándar.

2.1.1. Antichamber

Antichamber es un juego en primera persona creado en 2013 por un desarrollador independiente sobre el motor *Unreal Engine 3*. Originalmente el juego en su publicidad afirmaba ser un juego de espacio no euclídeo, haciendo uso de perspectivas y objetos que normalmente serían imposibles en la vida real. Entre estos puzles están algunos como pasillos infinitos o pasillos donde tienes que girar más de una vuelta para volver a donde estabas.

Además de esta manera de definir el mundo que se salta las reglas usuales del espacio, hace uso de mecánicas y eventos que dependen de la orientación con la que el jugador mira y se mueve por las salas. El jugador también cuenta con diversas pistolas que va desbloqueando con las que interactuá con ciertos cubos de colores para seguir avanzando.

A pesar de que afirma no ser euclídeo, todos los movimientos y física son euclídeos localmente, es decir, dentro de una sala te mueves de la manera usual, pero es al unir distintas salas lo que da lugar a estructuras que normalmente no serían posibles.

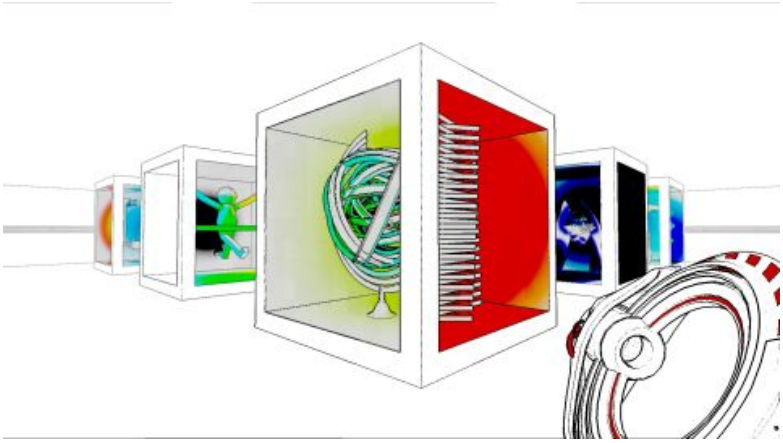


Figura 2.1: Captura de antichamber donde se pueden ver estructuras que parecen imposibles [1]

2.1.2. Gateways

Gateways es un juego 2D de plataformas al estilo tradicional, donde las mecánicas básicas son saltar en enemigos, plataformas y pinchos. Además de esto incluye una forma de poner portales que tienen distintas propiedad al atravesarlos, por ejemplo, al poner portales de distinto tamaño el jugador también modifica su tamaño, o portales que juegan con el tiempo y permiten interactuar con versiones anteriores del jugador y las acciones que ha realizado.

Lo más importante es que estos portales, y en general todo el juego, hacen uso del concepto de *línea de visión*, modificando lo que el jugador ve según donde está y lo que tiene dentro de su campo de visión.

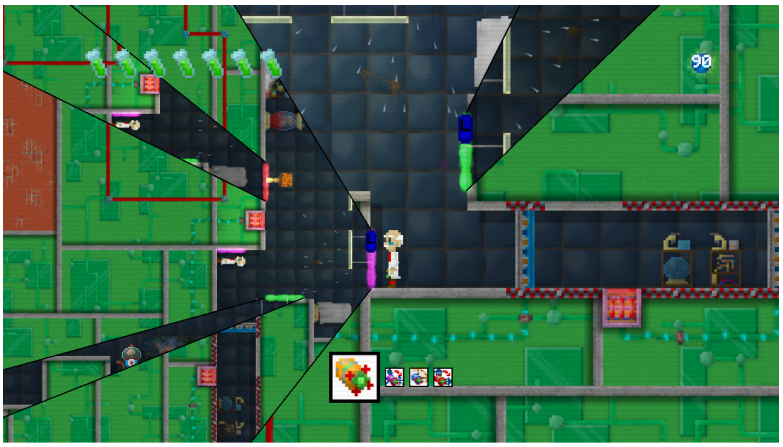


Figura 2.2: Captura de Gateways donde se ve claramente el concepto de *líneas de visión* [2]

[1] Imagen obtenida de: https://en.wikipedia.org/wiki/Antichamber#/media/File:Antichamber_screenshot.jpg
[2] Imagen obtenida de: http://www.smudgedcat.com/gateways.htm

2.1.3. Portal

Portal es un juego de puzles en primera persona creado en 2007 por *Valve Corporation*, sobre su propio motor *Source*. El juego fue muy bien recibido, considerándolo uno de los más originales de 2007, aunque fue criticado por ser demasiado corto. En 2011 se lanzó una secuela, Portal 2, que añadía nuevas mecánicas y un modo cooperativo multijugador.

El juego se basa en una serie de puzles que deben ser resueltos teletransportando al jugador y algunos objetos usando una pistola que permite crear portales entre dos superficies planas. Debido a cómo están construidos, estos portales permiten mantener algunas propiedades físicas, como el momento, entre los extremos del portal.

El comportamiento del juego es claramente euclídeo, siendo los portales los que rompen las reglas normales, pemitiendo coser distintos puntos del espacio. En nuestro motor se ha introducido un comportamiento similar, pero en 2D.

Figura 2.3: Captura de Portal que muestra el comportamiento de los portales [3]

2.2. Conceptos fundamentales

Hay una serie de conceptos que son fundamentales, sin lo cuales no se entendería el funcionamiento y el diseño del motor. Estos conceptos se introducen de manera no rigurosa, para que se pueda entender el resto del trabajo, ya que se hace un uso extensivo de ellos.

[3]Imagen obtenida de: http://tevisthompson.com/portal-2-and-point-of-view/

2.2.1. Línea de visión

Este concepto está extraído de Gateways, se basa en simular la manera de ver que tenemos en el espacio, entendiendo que vemos rayos de luz que se mueven en línea recta desde los objetos del mundo a nuestros ojos.

En nuestro caso es como si lanzáramos rayos desde el jugador y pintásemos todo lo que tocan esos rayos. Esta es la manera abstracta de modelar la visión y hemos encontrado formas eficientes de implementarlo, sobre todo tendiendo en cuenta la definición no euclídea del espacio que se usa, en la que hay objetos potencialmente pueden verse varias veces.

2.2.2. Fundamentos matemáticos

La arquitectura que hemos diseñado está basada sobre el fundamento de espacios con distintas curvaturas y como modelizarlos e interactuar con ellos de manera que se pueda formar un motor con el que diseñar y crear juegos.

El espacio está modelizado sobre una superficie en tres dimensiones que depende de la curvatura del espacio, la superficie viene dada por la ecuación

$$k(x^2 + y^2) + z^2 = 1$$

donde k corresponde a la curvatura y (x, y, z) a los puntos de la superficie. Dependiendo de los valores que tome k, la superficie se comporta de distinta manera, así:

- curvatura positiva: es un elipsoide, una superficie acotada.
- curvatura cero: un par de planos.
- curvatura negativa: un hiperboloide de dos hojas.

Para poder trabajar con las coordenadas de la superficie usamos distintas proyecciones, para lo que es necesario escoger con cuál de las hojas vamos a trabajar, por convención tomamos la hoja superior en el caso del hiperboloide y el plano.

A la hora de definir una posición en la superficie no solo usamos las coordenadas de un punto, si no que consideramos una base de $\mathbb{R}^3$ formada de tres vectores

$$\begin{pmatrix} v_{11} & v_{21} & p_1 \\ v_{12} & v_{22} & p_2 \\ v_{13} & v_{23} & p_3 \end{pmatrix}$$

el vector columna p corresponde a la posición y es un punto de la superficie (además es normal a esta), los vectores columna v_1 y v_2 corresponden a una base del espacio

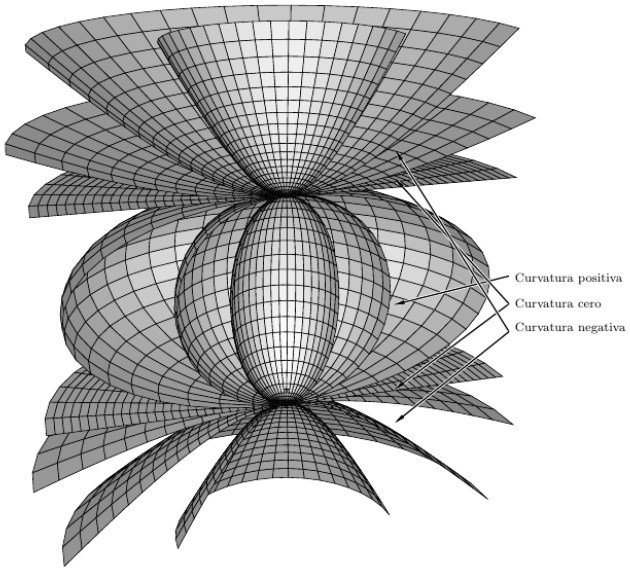


Figura 2.4: Distintas superficies según cambia la curvatura

tangente en p. Para operar sobre estas matrices, en muchos casos interpretemos los vectores por separado como un par posición, p, y orientación, (v_1, v_2).

Es necesario redefinir el concepto de recta entre dos puntos que se tiene en el espacio euclideo. Para ello definimos que la recta que une dos puntos es una geodésica de la superficie. Esto es importante al moverse en la superficie, ya que no basta con el movimiento usual, si no que hay que hacerlo con el transporte paralelo y ciertas aproximaciones numéricas que nos permiten realizarlo de manera rápida sin tener que acudir a las ecuaciones diferenciales.

Este concepto también se aplica a las *líneas de visión*, que de ser rectas en el espacio euclideo pasan a ser geodésicas de la superficie. Esto da lugar a propiedades interesantes de cada una de las superficies, por ejemplo:

- El espacio esférico es acotado, de forma que si caminas en una dirección acabas volviendo al mismo lugar. Además los rayos que salen de un punto vuelven al mismo, con lo que terminas viéndote a ti mismo otra vez.
- En el espacio hiperbólico la distancia crece de manera exponencial conforme se aleja del centro, es decir, aunque se vean dos objetos a una cierta distancia en la proyección, al alejarse de ti los objetos, la distancia entre ellos es mayor (o lo que es lo mismo, los ves más próximos).

2.3. Unity y sus herramientas

A la hora de decidir qué herramienta e infraestructura usábamos tomamos en consideración distintos motores gráficos. La principal función de los motores gráficos

es facilitar el proceso de pintar, pero no da facilidades para desarrollar el resto de un posible juego, teniéndolo que hacer todo el desarrollador, el bucle principal del juego, la interacción con el usuario... Debido a esto decidimos usar un motor de juegos más completo para enfocarnos en el modelo del espacio.

Unity es una herramienta muy usada en el desarrollo de videojuegos, con una gran comunidad detrás y una documentación oficial completa de la API de desarrollo, además de muchos recursos a los que acudir a por información. Debido a la manera que aproximamos el proyecto, uno de los inconvenientes que encontramos es que hace una distinción muy clara en *API de scripts*, o API para programadores, y el entorno de ventanas visual. Esto ha hecho que no hayamos explotado del todo la potencia de Unity, ya que el desarrollo se ha realizado en los *scripts*.

2.3.1. Scenes, GameObjects y components

El desarrollo en Unity funciona por proyectos, cada proyecto está compuesto de varias `Scenes` o escenas. Las escenas contienen los objetos que van a visualizarse y con los que vas a interactuar en un mapa o en una pantalla. De esta forma podríamos tener una escena para el menú principal, otras para cada uno de los niveles y mecanismos para cargar distintas escenas.

Los objetos principales de las escenas son los `GameObjects`. El concepto básico es que son objetos que están en una posición y con una orientación dadas en el mundo que define la escena. Para extenderlo Unity usa una arquitectura de componentes, en la que el comportamiento del objeto cambia según qué se haya añadido.

Entre los componentes más importantes están el `MeshRenderer` y el `MeshFilter`. Estos componentes le dan la forma física y la apariencia al objeto, que junto con propiedades como los *Materiales*, son los encargados de pintar el objeto en la pantalla.

Los `MonoBehavior` son los encargados de dar comportamiento específico a los objetos. Se pueden añadir varios a un `GameObject` y mediante la API que ofrece Unity se ejecuta código con el que interactuás con los distintos objetos y las escenas.

A parte de éstos, hay otros componentes para el resto de funcionalidades que tendría normalmente un juego, por ejemplo, existe un `MeshCollider`, que define las colisiones de los objetos, pero debido a que nuestra física y espacio son especiales hemos tenido que reimplementarlo en los scripts.

2.3.2. Cámara

Las cámaras (`Camera` en Unity) son un tipo de dispositivo a través del cual se muestra el mundo. Puede haber un número ilimitado de cámaras, que se pueden configurar de distintas maneras para cambiar la manera en la muestran el mundo.

Una cámara para decidir que es lo que muestra o no usa dos planos de corte que forman una pirámide truncada, cómo se ve en la Figura 2.5, en inglés a esto se le conoce como *view frustum*. De esta manera, la cámara muestra lo que está dentro de la pirámide, ignorando lo que queda fuera de ella.

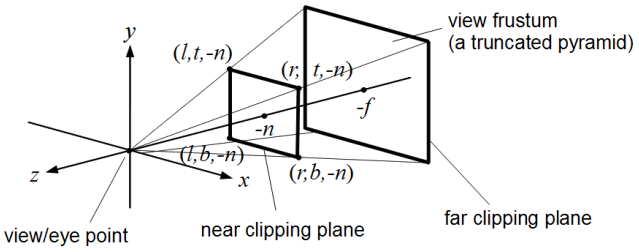


Figura 2.5: Pirámide truncada de visión

Las cámaras, además, pueden o no mantener la perspectiva. Para ello Unity permite elegir entre cámaras `Perspective`, que funcionan como en la Figura 2.5, o cámaras `Orthographic`, que eliminan la perspectiva, de forma que los objetos no cambian su tamaño con la distancia.

2.3.3. Recursos y assets

Para organizar todo el contenido Unity usa una jerarquía de carpetas donde colocar el contenido de distintos tipos de archivos. A todos estos archivos que son necesarios en el desarrollo Unity los llama *assets* y están en el subdirectorio *Assets/*, donde se realiza la mayor parte del trabajo.

Aquí puedes colocar `MonoBehavior`, *Materiales* y *shaders* en cualquier lugar, ya que Unity se encarga de cargarlos para que puedas acceder a ellos desde su entorno. Además de esto ofrece ciertas carpetas con comportamientos diferentes, las relevantes para nosotros por la manera en la que hemos estructura el código son:

- *Resources*: lugar donde almacenar recursos como texto o imágenes que se quieran cargar y usar desde los scripts.
- *Plugins*: el código o las librerías que se añaden aquí están disponible para cualquier script y es el primero que se carga.

2.3.4. MonoBehavior y API

Los `MonoBehavior` están pensados para personalizar el comportamiento de los objetos. Pueden estar escritos en *javascript* o *C#*, pero nosotros hemos elegido *C#* por se un lenguaje orientado a objetos y más pensado para desarrollos complejos.

Para implementar el comportamiento en un script se implementan métodos específicos que son llamados cuando ocurren ciertos eventos, por ejemplo `Start()` se ejecuta al cargar el script o `Update()`, que se llama en cada frame.

El problema al que nos enfrentamos es que estos scripts están pensados para cambiar el comportamiento de los objetos, pero nosotros necesitábamos reimplementar la arquitectura básica de cómo se define y pinta el mundo. Finalmente tomamos la decisión de implementarlo como un *Plugin*, de forma que el código específico de los objetos no está en un `MonoBehavior`, si no que es un manager el que se encarga de gestionarlo y el `MonoBehavior` tan solo se encarga de avisar de los eventos al manager.

2.3.5. Materiales y shaders

El paso final para pintar un objeto es el shader. Un shader es un trozo de código que normalmente se ejecuta en la tarjeta gráfica y se encarga de decidir qué y como pinta los objetos.

Para esto hay dos elementos básicos, por una parte una *textura* y por parte parte un *mesh*:

- La *textura* es una imagen que se va a pintar sobre la superficie que define el objeto.
- El *mesh* es una malla de vértices y triángulos que definen la forma que va a tener el objeto, además de cómo se asocian estos vértices a la textura.

Unity usa de lenguaje para los shaders *Cg*, un lenguaje desarrollado por Nvidia para poder programar el funcionamiento de la tarjeta gráfica a alto nivel. El shader tiene dos etapas: el *vertex shader* y el *pixel shader*.

En el *vertex shader* se trabaja sobre el *mesh* y se transforman sus vértices para llevarlos al lugar donde deberían estar en el mundo según la posición del objeto y la cámara y después proyectarlos en la pantalla. Debido a que la geometría que usamos no es la usual, todo el proceso de transformar los vértices de la malla es distinto, ya que nosotros hacemos nuestra propia proyección sobre la superficie de azimuth.

En el *fragment shader* se decide qué color va a tener cada uno de los pixeles, teniendo también la posibilidad de descartar vértices que no se quieran pintar. Para esto se usa la asociación de cada vértice de la malla con puntos en la textura.

Para acceder a los shaders Unity usa *Materiales*. Un material define propiedades de un objeto, como el brillo o la opacidad, y contiene un shader, que usando estas propiedades, decide como pintar los objetos. Unity tiene una serie de shaders básico por defecto y una API para poder crear tus propios materiales y shaders y comunicarse entre ellos.

Capítulo 3

Trabajo previo

Desde un primer momento se tuvo claro que íbamos a trabajar en espacios caracterizados por su curvatura, pero antes de llegar a diseñar e implementar el motor pasamos por distintas fases, en las que fuimos progresivamente depurando los modelos que teníamos del mundo, y la manera en la que visualizamos el espacio no euclídeo, así como un juego basado en él.

3.1. Primeros diseños no euclídeos

Cuando se nos ocurrió la idea de diseñar un juego que no fuese euclídeo tuvimos distintas aproximaciones de cómo podría modelarse y comportarse el espacio, y en función de ellas nos embarcamos en pensar cómo se podría jugar, para ver si hacer un videojuego era factible y que aportaría de originalidad a la experiencia de juego esta concepción del espacio.

Uno de los primeros modelos se basaba en *burbujas de geometría*, que funcionaban de manera similar a una lente que aumenta o disminuye el espacio provando una deformación en él. Sobre este modelo diseñamos distintas mecánicas que luego se descartaron ya que se observó que no era un modelo factible.

También trabajamos en distintas formas de definir la física o mecánicas que en lugar de interactuar con el modelo de la física, trabajasen sobre la representación de la pantalla, es decir, el jugador interactúa de manera euclídea con la versión deformada del mundo.

En estos primeros diseños se trataba la geometría como algo local, que el jugador podía poner y quitar, como una mecánica más, lo que después paso a ser algo global integramente parte del mundo. Todos estos pasos adelante y atrás no fueron un fracaso, ya que nos llevaron a tener más claro qué modelo era el que buscábamos y a encaminarnos a él.

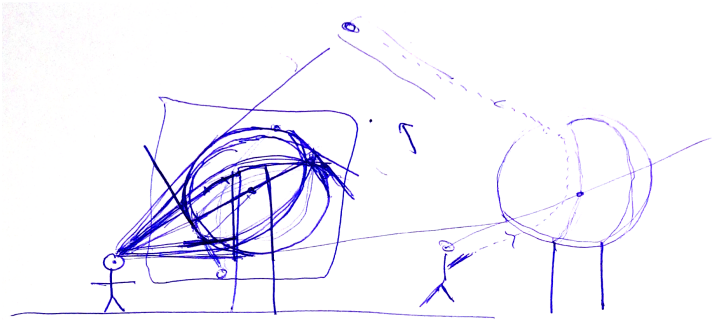


Figura 3.1: Mecánica en la que una pelota rebota sobre la versión deformada del mundo

3.2. Primera implementación

La primera aproximación al modelo de la geometría no euclídea era muy distinta a la que se usa actualmente, se intentó resolver directamente las ecuaciones diferenciales con las que representábamos la geometría, dando lugar a fórmulas intratables, que además eran distintas para distintos valores de la curvatura.

A parte de que las fórmulas tenían un coste computacional grande, para implementar el concepto de *línea de visión* se calculaba por cada píxel la línea explícitamente, dando lugar a un coste aun mayor.

Para probar esto usamos un código en java que dada una imagen, generaba una nueva versión con la curvatura dada, en la Figura 3.2 se muestra una las funciones. Esto lo usamos para crear animaciones imagen a imagen.

```
public static double[] formula(double x0, double y0, double p0, double p1, double R) {
    double t = Math.hypot(p0, p1); double theta = Math.atan2(p1, p0);
    double res[]=new double[2];
    double sT=Math.sin(theta);
    double cT=Math.cos(theta);
    double st=Math.sin(t);
    double ct=Math.cos(t);
    double a=x0*x0+y0*y0;
    double denom= (a+1)+(1-a)*ct-2*x0*cT*st-2*y0*sT*st;
    res[0]= 2*x0*ct +(1-x0*x0+y0*y0)*cT*st-2*x0*y0*sT*st;
    res[1]= 2*y0*ct+ (1-y0*y0+x0*x0)*sT*st-2*x0*y0*cT*st;
    res[0]*=R/denom; res[1]*=R/denom;
    return res;
}
```

Figura 3.2: Función que para cada píxel calcula su nueva posición (geometría esférica)

Estas primeras aproximaciones fueron un fracaso, ya que cada animación podía tardar alrededor de cinco minutos para tan solo unos segundo de animación. De todas formas nos permitieron hacernos una idea más concreta de cómo se ve y funcionan las distintas curvaturas, además de comprobar que necesitábamos una mejor aproximación.

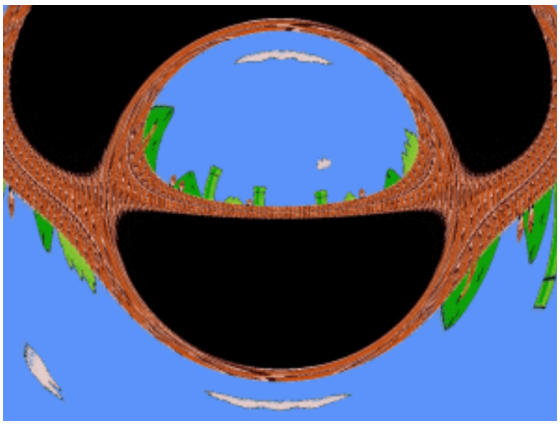


Figura 3.3: Parte de una animación de una proyección esférica

3.3. Unity y Shaders

El modelo final que se ha usado para modelar el espacio no euclídeo da una definición global para las propiedades geométricas del espacio. De esta manera, las proyecciones usadas son más uniformes y su implementación en un shader más sencilla y eficiente.

Antes de decidir si usar o no Unity y de diseñar la arquitectura del motor, se crearon una serie de pruebas para comprobar que se podría *pintar* lo que queríamos. Para ello diseñamos un primer shader, pero dado que el modelo matemático aun no estaba totalmente refinado, todos los cálculos relacionados con la geometría estaban en él, lo que involucraba una gran cantidad de cálculos por cada vértice de la malla.

Este shader transformaba una textura plana (el mesh tenía que ser un plano) y realizaba una serie de proyecciones hasta dar el resultado final. Con esto creamos escenas en las que nos movíamos y observamos cómo se deformaba el mundo al movernos. Esto nos permitió ver que la aproximación de los shaders era factible, aunque aun tenía margen de mejora. En la versión final del shader, la mayor parte de los cálculos relacionados con la geometría se sacaron a matrices, de manera que el cálculo de la matriz sirve para todos los vértices de la malla.

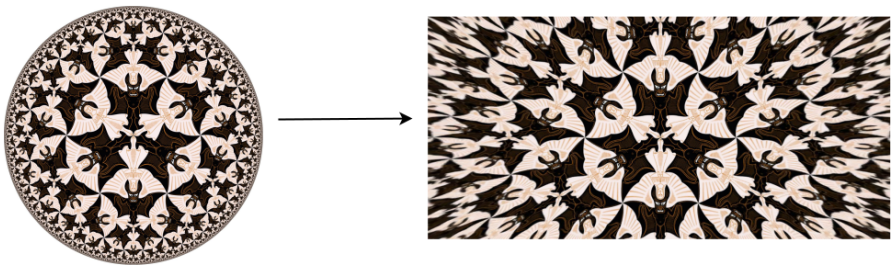


Figura 3.4: Prueba con el grabado *Límite Circular IV* de M.C. Scher

Capítulo 4

Visión general de la arquitectura

Para dar soporte a los conceptos presentados antes es necesario que la arquitectura del motor pueda modelar superficies complejas con distintas curvaturas y que se pueda trabajar y representar en la pantalla de manera eficiente.

En general, la arquitectura del motor es bastante independiente de Unity, ya que la mayor parte de las funcionalidades y necesidades del motor están encapsuladas dentro de él. Esto ha supuesto un problema posteriormente, dado que al estar poco integrado con Unity, el desarrollo de videojuegos usando el motor es más complejo por no usar la interfaz estándar de Unity.

4.1. Partes de la arquitectura

El motor está divido en distintas partes según la lógica que encapsulan. En la Figura 4.1 se muestran estas partes, y las interacciones entre ellas. Si se excluye Unity del diagrama, las capas se comportan como un MVC.

4.1.1. AzimuthWorld

El `AzimuthWorld` o *mundo azimuth* es donde se define el mundo de manera canónica, de forma que es en este nivel en el que ocurren las interacciones entre los objetos del mundo, que posteriormente se representará visualmente.

La representación del `AzimuthWorld` consiste en un *grafo de parches*, además de lógica y elementos necesarios para interactuar con él. Este grafo de parches define la estructura del mundo a trozos (un trozo del mundo es un parche), además de la manera de unir unos trozos con otros (enlaces). Los objetos están posicionados en los parches, navegando a través de los enlaces de un parche a otro.

Sobre los objetos, `AzimuthObject`, se ha construido un sistema de scripts y listeners que permiten reaccionar a los cambios en los objetos en las distintas partes. Además,

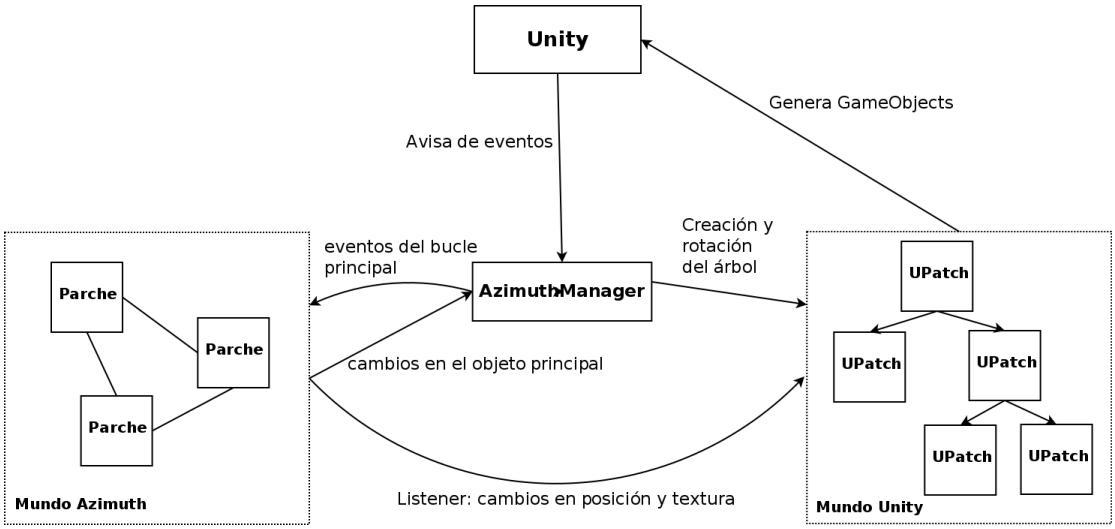


Figura 4.1: Esquema general de las interacciones entre las distintas partes

de cara a un desarrollador que use el motor, proporciona una manera de implementar reacciones y comportamientos específicos para los objetos.

4.1.2. UnityWorld

Para poder representar visualmente el *mundo azimuth* está el `UnityWorld` o *mundo unity*. En esta capa se transforma el *mundo azimuth* a una representación que se pueda mostrar en pantalla, ya que hay que solucionar problemas como objetos que se muestran varias veces en la pantalla.

La parte más importante es el *árbol de visibilidad*, donde se decide qué se va a mostrar del `AzimuthWorld`. Al generar el árbol de visibilidad es necesario un objeto desde el que parten las *líneas de visión* para decidir lo que es visible o no. Este objeto se llama *objeto principal* (`main_object` en el código) y normalmente será el jugador (como en Gateways), aunque esto puede cambiar dependiendo del diseño concreto del escenario.

4.1.3. AzimuthManager

La conexión entre estas capas se hace a través de un controlador, el `AzimuthManager`, que se encarga de la comunicación entre ambas usando un sistema de *listeners*, además de manejar los distintos eventos. También tiene la tarea de crear y rotar el árbol de visibilidad, ya que es necesario una visión global del *mundo azimuth* y *mundo unity* para ello.

El `AzimuthManager` también es un singleton global que permite acceder a los objetos del *mundo azimuth* e interactuar con ellos, de forma que desde cualquier parte del

motor se puede tener acceso a estas funcionalidades. Entre estas funcionalidades se incluye controlar y gestionar el *grafo de parches*.

4.1.4. Unity

Unity funcionaría como una capa más con la que se interactúa, que, como se muestra en la Figura 4.1, es en última instancia el que se encarga de la comunicación con el usuario, de pintar el mundo, del bucle principal y los eventos. Unity avisa de los eventos que ocurren al manager para que a su vez los comunica al *mundo azimuth*. Además, el *mundo unity* al transformar el *mundo azimuth* lo hace de manera que Unity lo pueda entender y pintar.

La parte final del proceso de pintar los objetos son los shaders. El objetivo final del motor es pasarle los datos necesarios para que pinten el mundo tal y como lo queremos, para lo que en el `UnityWorld` se crean `GameObject` de Unity, a través de los cuales acceder al shader y sus propiedades.

Capítulo 5

Modelo del mundo

Por la forma de modelar y proyectar el mundo que hemos usado, los escenarios complejos o muy grandes no se pueden definir de manera global, si no que tienen que separarse en distintas partes, que las hemos llamado *parches*. Por ejemplo, en el caso de una esfera no puede definirse un parche que cubra toda su superficie, así que se parte la esfera en distintos trozos, de forma que al juntarlos se comporten y representen en la pantalla como si fuesen una esfera completa.

La estructura que soporta esta definición es el *grafo de parches*, que contiene los parches y como están unidos entre ellos. Este grafo representa el mundo de una escena y además contiene los objetos con los que se va a interactuar.

Sobre el *grafo de parches* se define una API de *listeners* y *scripts* para especificar el comportamiento de los objetos y poder extender el mundo con definiciones propias de estos.

5.1. Grafo de parches

Para definir este grafo y modelar el mundo definimos los conceptos de:

- *Parche*: un trozo del mundo de tamaño arbitrario, funciona de nodo del grafo.
- *Enlace* o *Costura*: cumple la función de *coser* parches entre sí, es decir, se comporta como una arista del grafo y permite pasar de un parche a otro.
- *Objeto*: los elementos situados en los parches que contienen la lógica del mundo.

Basado en estos conceptos se crean las clases `Patch`, `Link` y `AzimuthObject` respectivamente. Estas clases son la base que define el mundo, que después se extenderá para ofrecer interacciones y comportamientos complejos.

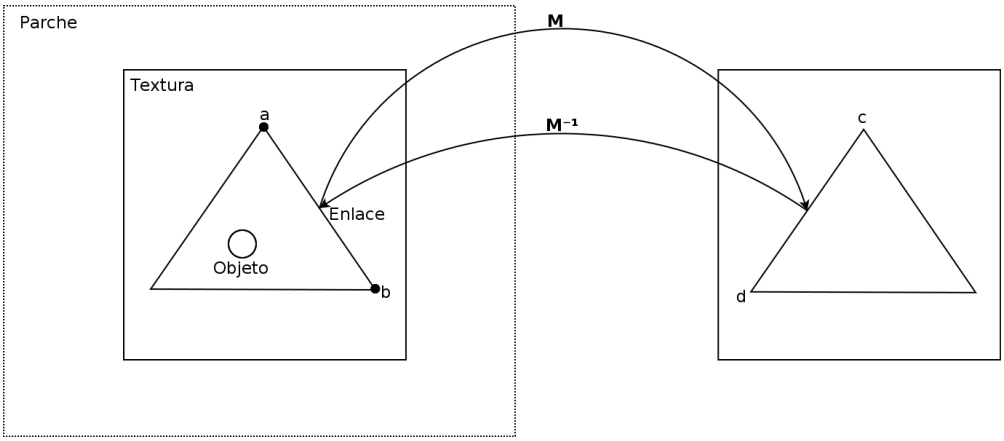


Figura 5.1: Esquema de cómo estaría organizado un parche

5.1.1. Parche

El parche define un trozo del mundo que contiene los enlaces que apuntan a otros parches y lo *cosen* con el resto del mundo, además de los objetos que en ese momento viven y se mueven en él.

Un parche tiene un componente visual, que correspondería a cómo y qué se ve en el fondo sobre el que están los objetos. De esta forma un parche contiene una *textura* y un *mesh* que se usaran posteriormente en el *mundo unity* para pintarlos.

Para permitir trabajar con el grafo, también contienen lógica que permite a los objetos moverse de un parche a otro (`addObject`, `removeObject`) y saber qué enlaces se han cruzado dentro de un parche (`crossedLink`).

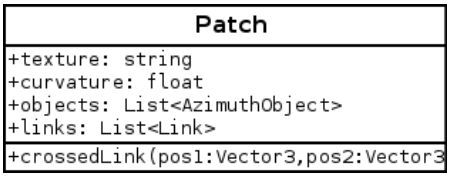


Figura 5.2: Diagrama general de un parche

5.1.2. Enlace

Un enlace representa una unión entre dos parches de forma que los objetos pueden atravesarlo para moverse por el mundo. Los enlaces también pueden visualizarse como los extremos de un portal que nos transportan de un parche a otro. Un enlace está representado por:

- un parche, `patch`, al que pertenece el enlace.

- un segmento formado por dos extremos, `left` y `right`, con coordenadas sobre el parche.
- un enlace destino, `dest`, al que apunta el enlace.

De esta forma, los enlaces no apuntan directamente al parche siguiente, si no que apuntan al enlace con el que están asociados en el siguiente parche.

El *grafo de parches* es un grafo bidireccional, pero por lo anterior, cada arista bidireccional está modelada como un par de aristas unidireccionales. Además el grafo puede potencialmente contener ciclos lo que hace que esos parches se puedan ver múltiples veces en la pantalla (por la *línea de visión*).

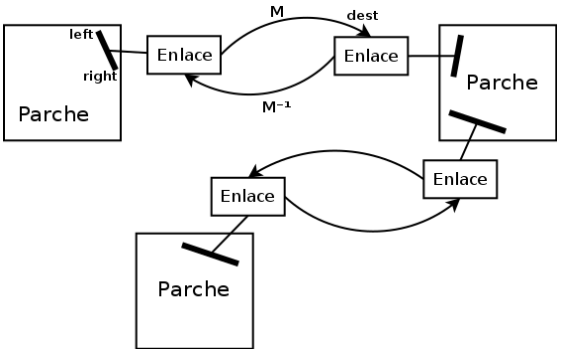


Figura 5.3: Grafo de parches, con los enlaces como aristas

El enlace tiene la función importante de pasar del entorno de coordenadas de un parche al de otro. Para hacerlo se usa una transformación afín representada como una matriz que permite saber la nueva posición y rotación al cambiar de parche. Esto también es importante a la hora de pintar los parches, ya que la *línea de visión* es una recta, que al atravesar un enlace *se gira* según la transformación afín del enlace.

5.1.3. Objetos

Los objetos, `AzimuthObject` en la arquitectura, son los elementos básicos con los que interacciona el mundo y el usuario. Estos objetos están pensados para simular un `GameObject` de unity en el sentido de que tienen distintos componentes visuales y scripts que modifican la manera en la que se comportan.

En un principio un `AzimuthObject` tiene un parche al que pertenece y una posición, una matriz tres por tres, correspondiente a su posición en la superficie de azimuth dentro del parche al que pertenece. A parte tiene un `PhysicalObject` que define propiedades respecto las colisiones y el movimiento del objeto.

Por debajo de un `AzimuthObject` tenemos el `AzimuthRenderedObject`, que incluye los elementos necesarios para poder pintarlo y que serán con los que se interactúe en el mundo unity, como el mesh o la textura.

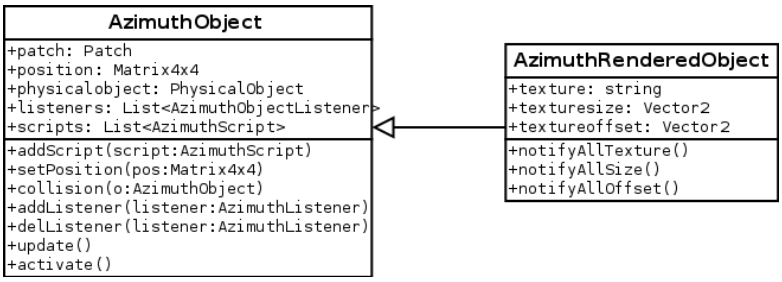


Figura 5.4: Elementos básicos de un *AzimuthObject* y *AzimuthRenderedObject*

Para extender los objetos y que un desarrollador implemente los suyos propios se heredaría de alguno de estos y se implementarían los métodos correspondientes, como el `update()`, o se añadirían `AzimuthScript` que ejecutasen ese comportamiento, por ejemplo, en la Figura 5.5 se crea un interruptor que cambie su color al activarse.

```
public class Switch : AzimuthRenderedObject {
    private bool isOn;
    private int cooldown = 0;

    public Switch(string n , Patch p, Matrix4x4 pos, UMeshModel m)
        :base(n,p,pos,m,"textures/button_red"){
        isOn = false;
    }

    public override void activate(){
        base.activate();

        if (isOn){
            texture = "textures/button_red";
            notifyAllTexture();
            isOn = false;
        } else{
            texture = "textures/button_green";
            notifyAllTexture();
            isOn = true;
        }
    }
}
```

Figura 5.5: Implementación de un interruptor que cambia al activarse

5.2. Listeners

Para que las distintas partes de la arquitectura interactúen entre sí definimos un `AzimuthObjectListener`, que define una serie de eventos a los que pueden reaccionar las clases que lo implementan, en la Figura 5.6 se muestran los eventos que hay disponibles.

Dependiendo de en qué capa de la arquitectura, se implementará uno u otro de los eventos que ofrece la interfaz, para que cuando este evento ocurra en un `AzimuthObject`, se llame al método correspondiente de sus *listeners*.

```
public abstract class AzimuthObjectListener {
    public virtual void notifyPosition(Matrix4x4 position, Matrix4x4 oldposition) {}
    public virtual void notifyPatchChange(Matrix4x4 pos, Patch p, Link l) {}
    public virtual void notifyTexture(string texture) {}
    public virtual void notifyActivate(AzimuthObject o) {}
    public virtual void notifyOffset(Vector2 offset){}
    public virtual void notifySize(Vector2 size){}
    public virtual void notifyDeath(){}
    public virtual void notifyCollision(AzimuthObject o){}
    public virtual void notifyObjectDelete(AzimuthObject o) {}
}
```

Figura 5.6: Interfaz del `AzimuthObjectListener`

Este sistema de *listeners* se usa internamente en la arquitectura para comunicar distintas partes, por ejemplo, avisar al árbol de visibilidad de movimientos en los objetos. Pero el programador puede definir libremente sus propios `AzimuthObjectListener` y registrarlos en los objetos para tener comportamientos específicos a eventos.

5.3. Scripts

También se define la clase `AzimuthScript`. Esta clase simula a una menor escala el comportamiento de los `MonoBehavior` de Unity, de forma que el manager global se encarga de avisar cuando ocurre alguno de los evento importantes en el ciclo de un juego, en la Figura 5.7 se muestran la interfaz de un `AzimuthScript`.

Además heredan de `AzimuthObjectListener` y se registran como tales en el objeto al que pertenecen, de esta forma también se comunica a los scripts de los eventos que ocurren los objetos y pueden implementar comportamientos específicos frente a ellos.

```
public class AzimuthScript :AzimuthObjectListener {
    public AzimuthObject getAObject();
    public virtual void start(){}
    public virtual void update(){}
    public virtual void ongui() {}
}
```

Figura 5.7: Interfaz del `AzimuthScript`

A través del método `getAObject()`, que devuelve el objeto al que pertenecen, además de a través del manager global, el script tiene acceso a toda la la arquitectura para poder interactuar con todas las capas.

5.4. Interacción física

Toda la interacción física está encapsulada dentro del motor motor físico, de forma que el objeto no tiene que preocuparse por los detalles de cómo funciona. De esta forma, para integrar un objeto con el motor físico, tan solo es necesario definir qué propiedades físicas tiene el objeto, lo que se traduce en darle un *PhysicalObject*.

Además, un objeto y sus listeners pueden reaccionar a una colisión. Ante una colisión, se llama al método `collision(obj)` del `AzimuthObject`, donde se puede implementar lógica de cómo reaccionaría el objeto (por ejemplo, una bala se desintegraría al chocar). También hay un evento, `notifyCollision()`, que también indica a los listeners que ha ocurrido una colisión. En la Figura 5.8, se da un ejemplo de cómo se haría un jugador con forma de colisión esférica.

```
public Player(string n, Patch p, Matrix4x4 pos, UMeshModel m, string t)
    : base (n, p, pos, m, t){
    physicalobject = new MovableObject(this, new SphereShape(0.1f), new TangibleBody());
}
```

Figura 5.8: Constructor de un personaje cuya forma de colisión es una esfera

Capítulo 6

Controladores y módulos globales

Para encapsular funcionalidades necesarias a lo largo de las distintas partes de la arquitectura se han creado una serie de módulos y controladores con visibilidad global.

El `GeometricEngine` y el `PhysicalEngine` se encargan de los cálculos matemáticos y la física respectivamente, teniendo ambos una interacción mínima con el resto de elementos, ya que funcionan como librerías a las que llamar. En cambio, el `AzimuthManager` se encarga de controlar las distintas partes de la arquitectura y por lo tanto está muy integrado con ellas.

6.1. AzimuthManager

El `AzimuthManager` es el controlador principal del motor, que se encarga de interactuar con el mundo azimuth y el mundo unity. Funciona como un *sigleton* global que es accesible desde cualquier punto de la API para manejar las distintas partes de la arquitectura.

La primera tarea que tiene es la de ser el punto de entrada al resto de la arquitectura. Para iniciarlo se le pasa un mapa, donde se definen que objetos, parches y enlaces hay en el mundo y se genera el *grafo de parches* del `AzimuthWorld`. Para poder gestionarlo, el manager también ofrece métodos con los que poder interactuar con el *grafo de parches* (`getObject`, `addAzimuthObject`, `removeAzimuthObject`).

El `AzimuthManager` es encargado de controlar y crear el `UnityWorld`, para ello define el `main_object`, que es el objeto desde donde van a partir las *líneas de visión* para poder construir el *árbol de visibilidad*. Para esto se usan los métodos `drawPatchTree` y `drawTreeFromLink` que son los encargados de recorrer el *grafo de parches* de manera recursiva y generar el árbol.

Para poder reaccionar a cambios en el `main_object`, el manager se registra como un listener e implementa los eventos `notifyPatchChangeMain` y `notifyPosition`.

Ante el evento de cambiar de parche, es necesario rotar el árbol, para lo que se usa el método `rotateTree`. También crea o destruye hojas hasta una profundidad máxima, usando los métodos `drawNewLeaves` y `depthBomb` respectivamente. Además, el `AzimuthManager` se encarga de propagar los cambios en la posición del `main_object` a lo largo del árbol y actualizar las distintas matrices.

Por último, el manager contiene la lógica relacionada con los eventos del *bucle principal*, para lo que hace uso de los eventos de Unity `Start`, `Update` y `OnGUI` y los propaga a todos los `AzimuthObject`. En cada frame también comprueba eventos como la activación de los objetos y llama al motor físico.

6.1.1. Carga de mapas

El `AzimuthManager` se crea con los datos del mundo, el problema es que hace falta una gran cantidad de datos relacionados con posiciones de los objetos, vértices de los enlaces, además de otros datos. Teniendo en cuenta que a su vez son matrices o vectores, la cantidad de elementos se hace inmanejable desde la interfaz de Unity rápidamente.

Por ello, los datos del mapa se almacenan en un archivo json que se traduce en el `AzimuthManager` a una instancia concreta del mundo azimuth. Estos archivos json están almacenados como *Resources* en unity y hay que crearlos a mano.

6.2. Motor geométrico

Consiste en una librería, `GemetricEngine`, que encapsula todos los cálculos complejos necesarios en el resto de la arquitectura, de forma que en el resto no haga falta meterse en detalle de cómo está implementado.

Esta librería puede descomponerse en dos partes, por un lado las distintas funciones base de la geometría, como el producto escalar, vectorial, coseno... que se necesitan redefinir para tener en cuenta la curvatura, y los cálculos complejos que hacen uso de estas funciones, sobre todo destacan:

- `Matrix4x4 move(Matrix4x4 pos, float x, float y, float k)` : calcula la nueva posición en la superficie de curvatura `k`, para ver en qué dirección y cuando te mueves se usa el vector (x, y), que está coordenadas de la base dada por los vectores $v_1; v_2$ (base del espacio tangente) de tu matriz de posición y es una aproximación del transporte paralelo.
- `float distNikolai(Matrix4x4 A, Matrix4x4 B, float k)` : Calcula la distancia entre dos posiciones de la superfice de curvatura `k`.

```
{
  "patches": [
    {
      "mesh": {
        "type": "implicit",
        "points": 1,
        "scale": 10,
        "curvature": 0
      },

      "texture": "textures/metal-textureC",
      "links": [
        {"left": [1,1,1], "right": [1,-1,1], "dest": [1,0]}
      ]
    },
    {
      "mesh": {
        "type": "implicit",
        "points": 1,
        "scale": 10,
        "curvature": 0
      },

      "texture": "textures/wood-textureB",
      "links": [
        {"left": [-1,-1,1], "right": [-1,1,1], "dest": [0,0]}
      ]
    },
  ],
  "objects": [
    {
      "mesh": {
        "type": "implicit",
        "points": 10,
        "scale": 1,
        "curvature": 0
      },
      "texture": "textures/scientist",
      "position": [[1,0,0],[0,1,0],[0,0,1]],
      "patch": 0,
      "name": "nikolai",
      "objectType": "Player",
      "script": "NikolaiScript"
    },
  ]
}
```

Figura 6.1: Ejemplo de mapa json

- `Matrix4x4 computeLinkMatrix(...)`: calcula la matriz de paso de un parche a otro. Los parámetro de esta función son cuatro `Vector3`, (`A`, `B`, `C`, `D`), donde las coordenadas del enlace en el parche origen son (`A`, `B`) y las del parche destino (`C`, `D`).

- `Matrix4x4 computeFragmentMatrix(...)`: Calcula la matriz del fragment shader. Al igual que en `computeLinkMatrix`, los primeros argumentos son las coordenadas de los enlaces. Además necesita las matrices anteriores del árbol de visibilidad.

6.3. Motor físico

El motor físico, `PhysicalEngine`, es el que se encarga de la interacción entre los objetos a nivel físico, es decir, las acciones de mover y colisionar los objetos. En un motor físico para hacer esto se tienen en cuenta muchos factores como la velocidad, la aceleración, el vector normal a la colisión, la distancia de penetración... lo cual está definido en la geometría euclídea y no en el modelo que usamos. Debido a esto las interacciones físicas con las que cuenta el motor actual son muy simples.

Para ver qué objetos colisionan o no con otros hay un *bucle físico*, que se encarga de avanzar todos los objetos un instante de tiempo y de resolver las colisiones, para hacerlo intenta simular paso a paso el movimiento hasta que todos los objetos se hayan movido sin estar en estado en el que se crucen.

Para definir la interacción física los objetos tienen un `PhysicalObject` que se encarga de resolver las colisiones en última instancia. Este objeto a su vez contiene una forma (`Shape`) y un cuerpo (`Body`), que define las propiedades que tiene el objeto en cuanto a los choques y como reacciona a ellos.

6.3.1. Shape

La forma de un objeto, junto a su posición, permiten saber cuando dos objetos se están chocando en un instante dado. Las únicas formas que tenemos son circunferencias, puntos y segmentos. Son formas básicas comunes a las distintas curvaturas y con las que es fácil trabajar y comprobar las colisiones.

Generalizar la forma a modelos de colisión compuesto (por ejemplo, un objeto está formado por varias circunferencias) no es tan sencillo, ya que en los cambios de parche tendríamos parte de un objeto en un lado y parte en el otro, lo que con el cambio de coordenadas entre parche puede hacer que se desconecten las formas de las que está compuesto.

6.3.2. Body

El cuerpo define como reacciona un objeto ante una colisión, por ejemplo, con qué velocidad rebota o en qué dirección. Igual que con la forma, al no ser geometría euclídea, encontrar la dirección de rebote no es trivial. En un motor usual se usaría el vector normal al cuerpo en el punto de colisión, pero esto no es fácil de calcular en general en nuestra superficie.

Debido a esto, ese tipo de rebotes no están incluidos en el motor. Las reacciones que tenemos son de parar un objeto al colisionar o de atraversarlo. Además de la reacción física del objeto, cuando dos objetos colisionan se les avisa para que ejecuten lógica relacionada con la colisión.

Capítulo 7

Representación del mundo en Unity

En esta capa se decide qué y cómo se pintan los parches y los objetos de una manera que Unity sea capaz de entender. Esto a su vez está divido en tres partes:

1. En los objetos se define qué forma tienen dándoles un mesh, `UMeshModel`, y una textura.
2. Se decide qué y dónde se pinta cada parche usando un *árbol de visibilidad*.
3. Se le pasan los datos necesarios al shader para que pinte.

Todo esto está englobado en el *mundo unity*. El nombre hace referencia a que en última instancia se usan `GameObject` generados de manera dinámica en el *árbol de visibilidad*, con las propiedades necesarias para que Unity se encargue de pintarlos, es decir, al final lo que hacemos es transforma el *mundo azimuth* a algo que pueda entender Unity.

7.1. UMeshModel

Un *mesh* define la forma que tiene el objeto. Este mesh tiene que cumplir la propiedad de que sus coordenadas deben estar en la superficie de azimuth.

En general este mesh estará centrado en el origen, de forma que en el shader, usando las matrices generadas en el *árbol de visibilidad*, se transforman las coordenadas a su posición definitiva en la superficie.

La clase básica es el `UMeshModel`, que dados los vértices, triángulos y coordenadas de textura genera un `Mesh` de Unity. Esta clase funciona como interfaz entre los mesh de Unity y nuestra arquitectura.

Para facilitar la creación de los mesh, la clase que se usa es `GeneratedUMeshModel`, que dado una curvatura, escala y número de puntos genera un mesh cuadrado con una rejilla de NxN.

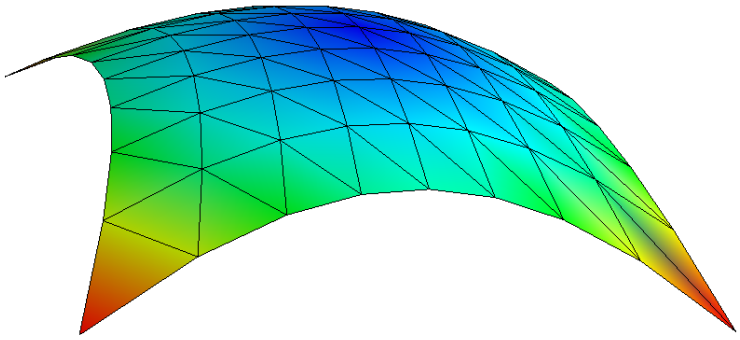


Figura 7.1: Ejemplo del resultado de un `GeneratedUMeshModel` hiperbólico

7.2. Árbol de visibilidad

En este paso se usa un objeto desde el que parten las *líneas de visión*, el `main_object`. Desde este objeto se añade en orden al árbol todo lo que está al alcance de estás línea, además de información necesaria para poder colocar los elementos del árbol según su posición y distancia respecto al `main_object`.

Para ello se genera un árbol que tiene como raíz el parche del objeto principal y a partir de este se recorre el grafo de parches y se van generando nodos en un recorrido en profundidad hasta una profundad máxima. Los nodos ya visitados se re-visitan,lo que genera copias del parche a distintas profundidad en el árbol.

Estos nodos se llaman `UPatch` (unity patch), y contienen lo necesario para que Unity lo pueda pintar en pantalla (un `GameObject` que se añadirá a la interfaz de Unity). Además se encargan de pasar todos los datos que el shader necesita en la siguiente fase.

La clase `ULink` actua como aristas del árbol y son el equivalente a los enlaces del mundo azimuth. La diferencia principal es que los `ULink` no son bidireccionales, si no que apuntan al siguiente parche en el árbol. Cada arista contiene también la matriz de paso desde el origen (el objeto principal), hasta el parche al que apunta. Esta matriz es la acumulada al pasar por los enlaces desde el parche origen hasta el parche destino y es necesaria para calcular las matrices del shader.

Este árbol es potencialmente infinito, pero solo interesa tener construidos los nodos que sean visibles en pantalla en este momento o en un futuro próximo si el objeto principal se moviese. Para ello definimos una profundidad máxima de nodos visibles, de forma que siempre está, como mínimo, construido el árbol hasta esa profundidad.

Cuando el árbol se muestra en Unity, da lugar a una estructura con forma de capas superpuestas a distintas alturas, donde cada capa corresponde a un nivel del árbol, dependiendo de su profundidad.

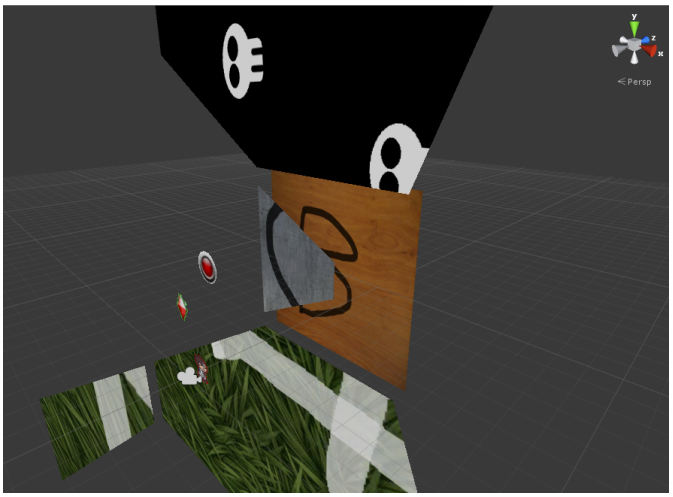


Figura 7.2: Vista de las distintas capas de los objetos y los parches

7.2.1. Cambio de parche

El cambio de parches del objeto principal implica que la raíz del árbol de visibilidad cambia y por lo tanto todas las matrices de los nodos también. En un principio reconstruíamos el árbol desde cero, pero este método era poco eficiente, provocando picos y saltos perceptibles para el jugador. El problema estaba provocado por que el cambio masivo de los `GameObject` de Unity era muy ineficiente.

Para solucionar este problema optamos por rotar el árbol al hijo que corresponda, de forma que tan solo hay que recorrer en profundidad el árbol desde la nueva raíz actualizando las matrices y a su vez, avisando a los `GameObject` del cambio.

En este recorrido en profundidad también se crean nuevas hasta una cierta profundidad máxima para mantener siempre coherente lo que el usuario observa en la pantalla. Además, las hojas que exceden esa profundidad (no son visibles en pantalla), no se borran en el momento, si no que se mantienen por eficiencia, ya que son hojas que pueden necesitarse en la siguiente rotación del árbol.

Por último, cuando un objeto cualquiera cambia de parche, en el mundo azimuth tan sólo implica mover un objeto, pero en el mundo unity hay que mover potencialmente muchos objetos, además tener que quitar o añadir objetos que ya no hagan falta. Para abordar esto, decidimos crear un *pool* de `UObjects`, de forma cuando se descarta un `UObject` se añade al *pool* y se pueden reutilizar el `GameObject` y las estructuras de unity que tiene ya creadas. Con esto mover un objeto consiste en descartar todas sus copias y luego añadirlas en las copias de los `UPatch` correspondientes.

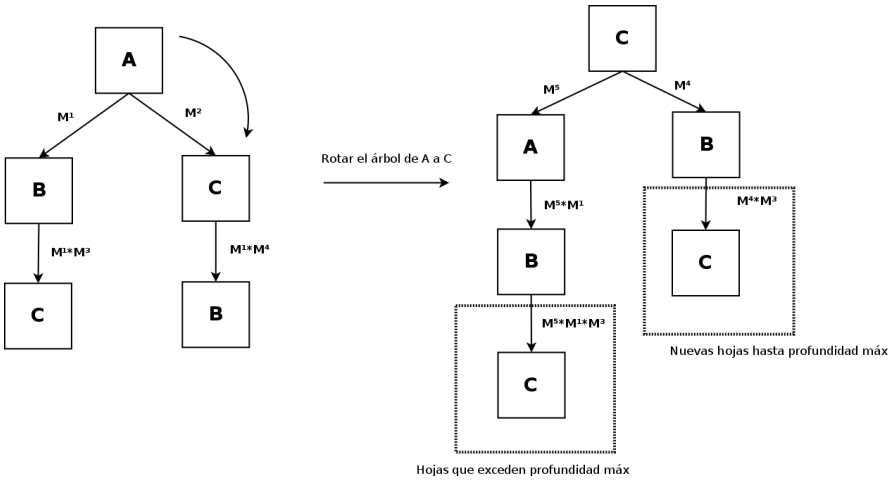


Figura 7.3: Rotación del árbol con profundidad máxima 2

7.3. Shader

El shader usa matrices generadas en el árbol de visibilidad, la `vertex_matrix` y la `fragment_matrix`, que corresponden al *vertex shader* y al *fragment shader*.

En el *vertex shader* se usa la matriz correspondiente para transformar las coordenadas desde el origen al lugar donde deberían estar en la superficie de azimuth y después se proyecta sobre el plano. Además se le asigna una profundidad según la capa a la que corresponda del árbol de visibilidad. El *vertex shader* está diseñado de tal manera que todo esto se calcule usando un producto de matrices y una transformación común para todos los puntos de la malla, lo que minimiza el número de cálculos.

En el *fragment shader* se recortan los vértices según la línea de visión desde el objeto principal, descartando los que quedan fuera de la región indicada por la `fragment_matrix`. Este recorte es un simple producto de matrices para comprobar si está o no dentro de la región de visión.

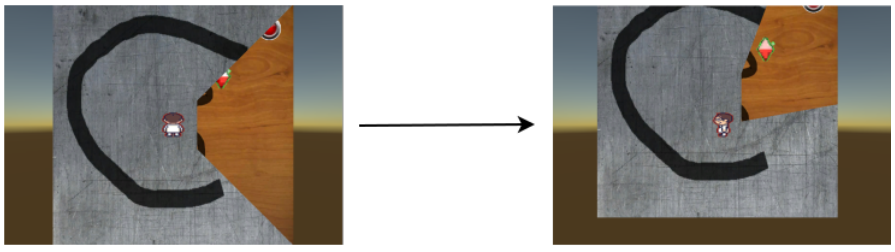


Figura 7.4: Cambio de las líneas de visión al moverse

Capítulo 8

Implementación en Unity

Unity permite diseñar juegos de una manera estándar con la interfaz y todas las herramientas que ofrece, facilitando el trabajo a los diseñadores y programadores.

El uso que se ha hecho de Unity a la hora de implementar la arquitectura y al diseñar un juego no es el usual, ya que había que redefinir e implementar muchos conceptos del mundo que están intrínsecamente a Unity. Por ejemplo, un `GameObject` cuenta siempre con una *transformación*, que indica la posición, rotación y escalado del objeto; pero en el modelo del mundo que usamos esto está representado de otra manera.

Debido a esto, gran parte del trabajo de programación y diseño se sacado de Unity y su API para llevarlo nuestra arquitectura, quedando Unity en una herramienta de pintar e inspeccionar los elementos que se han pintado.

8.1. Matrices y vectores

Unity cuenta con una librería con implementaciones de álgebra lineal para trabajar con posiciones y transformaciones, de la que se ha hecho un uso extensivo a lo largo de todo la implementación. Esta librería no es completa, ya que su objetivo es usarla en el entorno de un motor de juego y carece del algunas operaciones y estructuras necesarias.

Sobre todo es relevante el caso de las matrices. Todas las matrices que se necesitan a nivel teórico para transformaciones y posiciones son matrices en $\mathbb{R}^{3x3}$, pero en Unity tan solo están implementadas las matrices en $\mathbb{R}^{4x4}$, representadas por la clase `Matrix4x4`, con lo cual se han adaptado las matrices en $\mathbb{R}^{3x3}$ a matrices con la forma

$$\begin{pmatrix} M & 0 \\ 0 & 1 \end{pmatrix}$$

donde M es la matriz en $\mathbb{R}^{3x3}$ que originalmente queríamos usar.

8.2. Organización de los archivos

Los archivos relevantes con los que se trabaja en un proyecto de Unity están localizados dentro `Assets/`. En esta carpeta dependiendo, de la extensión de los archivos o de donde estén colocados se hace un uso u otro de ellos.

Para implementar la arquitecta hemos hecho uso de la carpeta `Plugins/`, ya que el código y las librerías en esta carpeta están disponible de manera global. Las carpetas importantes son:

- `azimuth-architecture/`: contiene el código *C#* principal de la arquitectura y está dividido de manera lógica en distintos archivos correspondientes a las distintas partes con las que se ha diseñado.
- `azimuth-scripts/`: contiene instancias de la clase `AzimuthScript` y se usan para extender y personalizar los `AzimuthObject`.

Las imágenes y los mapas (archivos .json) deben ser accesibles desde la API de Unity, ya que se cargan en tiempo de ejecución en distintas partes del código. Para ellos se ha usado la carpeta `Resources/`, en la cual se colocan las texturas y mapas en las carpetas `textures/` y `maps/` respectivamente, y se accede a ellos usando direcciones relativas a `Resources/`.

Por último, los shaders y los `MonoBehavior` pueden colocarse en cualquier lugar dentro de `Assets/`, ya que Unity se encarga de cargarlos, pero por conveniencia hemos usado las carpetas `Shaders/` y `Sources/` para contener estos archivos.

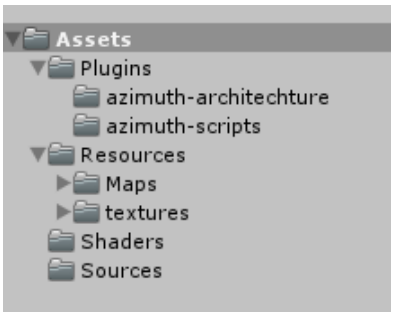


Figura 8.1: Estructura de los directorios desde la interfaz de Unity

8.3. Creación de escenas

Debido a las decisiones de diseño que hemos tomado, la definición de las escenas no se crea directamente en la interfaz de Unity, si no que esto se hace en los archivos de los mapas. Es decir, en lugar de crear los `GameObject` y añadirles

componentes, se crea un archivo `.json` con el mapa, que en ejecución generará todas los `GameObject` necesarios para Unity.

Teniendo esto en cuenta, en una escena antes de darle a ejecutar, lo único que es necesario es un `GameObject` con un `MonoBehavior` desde el que se cargue el `AzimuthManager` y el mapa.

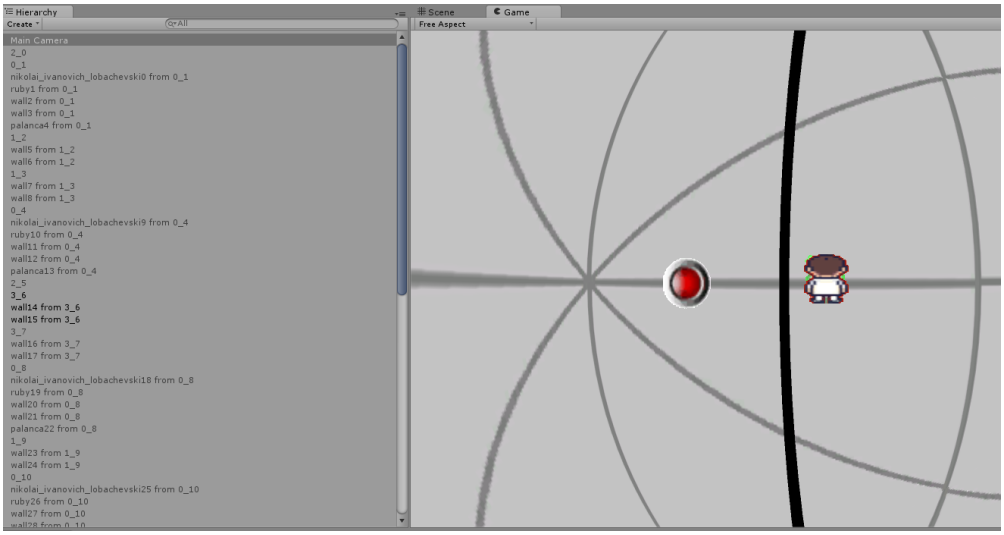


Figura 8.2: Escena de Unity tras ejecutarla, mostrando los `GameObject` que se han creado

8.3.1. Posición de la cámara

Para poder mostrar el mundo, Unity hace uso de cámaras que están posicionados en un punto del espacio, enfocando en una dirección concreta, por lo que al crear una escena es necesario decidir donde colocar la cámara.

Cuando se genera el mundo y se representa en Unity, se usa el plano XY $(z = n)$ para proyectar las distintas capas, además estas capas están orientadas mirando a *z positivo*. La primera capa está posicionada en $z = 0$, y a partir de ahí en capas por encima $(z = n, n > 0)$, hasta una cierta altura máxima, y una capa por encima de la altura máxima están colocados los objetos. Además, las capas se generan centradas en el $(0, 0)$, que es la posición del objeto principal, con lo que la cámara habrá que crearla centrada o cerca de este punto.

Debido a que las capas están a distinta altura, para poder mostrarlas sin que la perspectiva afecte a lo que se ve finalmente, se usa una cámara *ortográfica* en lugar de la cámara por defecto.

Capítulo 9

Conclusión y trabajo futuro

El objetivo inicial de este trabajo era ver que es posible crear un videojuego basado en espacio no euclideo. Finalmente hemos implementado una arquitectura basada en este tipo de espacios, con la que hemos creado algunos niveles a los que se puede jugar. Para mostrar la potencia del motor hemos hecho una demo con cinco puzles donde mostramos las propiedades del espacio y la potencia de este para crear mecánicas intrínsecamente no euclídeas.

Aun así aun quedaría mucho por mejorar y perfeccionar, por ejemplo, en estos momentos crear un juego puede ser complicado ya que no hay herramientas preparadas para generar mapas no euclídeos, con lo que hay que hacer muchos cálculos para poner bien los enlaces y dividir la superficie en parches que encajen, además de posteriormente tener que escribirlo en el json del mapa.

También la física habría que estudiarla más en profundidad, ya que no existe una forma canónica y satisfactoria de definir nociones como la gravedad o las fuerzas que afectan a un objeto ya que la noción de perpendicularidad es más difusa cuando trabajamos con distintas curvaturas.

Uno de los problemas que hemos tenido es que el modelo matemático, y junto a él el diseño del mundo, ha ido evolucionando conforme avanzábamos en el proyecto, enfrentándonos a problemas y encontrando soluciones y mejoras, con lo que se hace necesario una refactorización del código para mejorar el diseño de algunas estructuras.

A pesar de estos elementos que faltan, la evolución del proyecto ha sido satisfactoria, al pasar de una idea en papel que era muy difícil de visualizar y entender si no tenías nociones claras de los fundamentos y estabas inmerso en el proyecto, hasta ahora, que hemos depurado estas ideas y conceptos para dar con algo que hemos implementado y con lo que ya se puede jugar.

Bibliografía

[1] J. Darby. *Beginning OpenGL Game Programming*. IT Pro. Course Technology, 2014.

[2] R. Fernando y M.J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-time Graphics*. Graphics programming. Addison-Wesley, 2003.

[3] Unity Technologies. *Unity Manual*. 2015. URL: `http://docs.unity3d.com/Manual/index.html` (visitado 22-08-2015).

[4] Unity Technologies. *Unity Scripting API Reference*. 2015. URL: `http://docs.unity3d.com/ScriptReference/index.html` (visitado 10-09-2015).