Desarrollo en WebGL

Eduardo Antuña Díez[1], Aitor Díaz Solares[2], Daniel González Losada[3].

E.P.I. Gijón, Universidad de Oviedo

[1] UO160788@uniovi.es
[2] UO165918@uniovi.es
[3] UO165455@uniovi.es

Resumen. En el siguiente documento se buscará realizar un mini tutorial tratando de iniciar y familiarizar a los lectores sobre la programación en WebGL, partiendo desde la configuración de los navegadores para poder emplear esta tecnología, mostrando las nociones básicas para poder programar ejemplos simples, así como visualizar ejemplos más complejos en donde mostrar todo el pontencial.

1 Introducción

WebGL es una especificación estándar que está siendo desarrollada actualmente para desplegar gráficos en 3D en navegadores web. El WebGL permite activar gráficos en 3D acelerados por hardware en páginas web, sin la necesidad de plug-ins en cualquier plataforma que soporte OpenGL 2.0 u OpenGL ES 2.0. Técnicamente es un enlace (binding) para javascript para usar la implementación nativa de OpenGL ES 2.0, que será incorporada en los navegadores. WebGL es manejado por el consorcio de tecnología Khronos Group sin fines de lucro.

WebGL creció desde los experimentos del canvas 3D comenzados por Mozilla. Mozilla primero demostró un prototipo de Canvas 3D en 2006. A finales de 2007, tanto Mozilla como Opera habían hecho sus propias implementaciones separadas. A principio de 2009 Mozilla y Khronos comenzaron el WebGL Working Group (Grupo de Trabajo del WebGL).

El Grupo de Trabajo del WebGL incluye Apple, Google, Mozilla, y Opera, y WebGL ya está presente en los builds nocturnos de Mozilla Firefox 3.7, Mozilla Fennec 1.0, WebKit, y Google Chrome 4 developer previews.

Algunas bibliotecas en desarrollo que se están incorporando WebGL incluyen el C3DL y el WebGLU.

Utiliza el elemento canvas del HTML 5.

2 Configurar navegadores

Como punto de partida trataremos de conseguir un navegador que soporte WebGL. La mayoría de las versiones de desarrollo de los principales navegadores lo hace, sólo tendremos que conseguir la versión adecuada y realizar unas pequeñas configuraciones. Veremos las soluciones a adoptar en los principales sistemas operativos.

2.1 Windows

Si tenemos una tarjeta gráfica con GPU ATI o Nvidia, no debería haber problema. Podremos emplear Firefox o Chrome. En caso de tener problemas, tendremos que usar Firefox con software de renderización, es decir, los gráficos 3D serán ejecutados en el procesador normal, no en la tarjeta gráfica.

Para el caso de chipset gráfico Intel, presente en muchos ordenadores portátiles, sus controladores no suelen ser compatibles con OpenGL, por tanto tampoco lo serán con WebGL ya que funciona a través de WebGL (en el futuro esto será distinto). Podríamos probar las instrucciones para Firefox o Chrome, pero no es probable que funcione; usaremos Firefox con software de renderización.

Firefox

La versión de desarrollo se denomina Minifield. Para conseguirla y poder emplear WebGL seguiremos los siguientes pasos:

Descargaremos la versión adecuada a nuestra máquina.

Cerramos Firefox previamente, y una vez instalado Minifield, iremos a la página: *about:config*

Nos ayudamos del filtro de búsqueda, en donde introducimos "WebGL"

Localizamos *webgl.enabled_for_all_sites* y lo ponemos a *true*

Firefox con software de renderización

Si nuestro hardware no es compatible con OpenGL 2.0, la única manera de hacer funcionar WebGL será la utilización de la biblioteca Mesa. Lo que hace principalmente es emular una tarjeta gráfica, por lo que será un poco lento, pero al menos podremos usar WebGL. Se integra con Minifield. Los pasos a seguir son los siguientes:

Descargaremos la versión adecuada de Minifield para nuestra máquina.

Lo instalaremos, cerrando previamente Firefox.

Descargaremos la biblioteca Mesa. Lo descomprimimos y habrá un archivo llamado *OSMESA32.DLL* que guardaremos en algún lugar de nuestra máquina.

Iniciamos Minifield y vamos a la página: *about:config,* nos ayudamos del filtro de búsqueda e introducimos "WebGL"

Localizamos *webgl.enabled_for_all_sites* y lo ponemos a *true*

Localizamos *webgl.osmesalib* e introducimos la ruta donde se encuentra OSMESA32.DLL, por ejemplo le daremos el valor: C:\Mesa\osmesa32.dll

Localizamos *webgl.software_rendering* y lo ponemos a *true*.

Chrome

La versión de desarrollo se denomina Chromium. Para conseguirla y poder emplear WebGL seguiremos los siguientes pasos:

Descargaremos de la pagina de integración continua chrome-win32.zip

Descomprimimos, no es necesaria su instalacíón, simplemente ejecutamos el archivo chrome.exe.

2.2 Macintosh

Si trabajamos con Snow Leopard (OS X 10.6) no deberíamos tener problemas. Usaremos una versión de desarrollo alternativa de Safari. Si por contra la versión que manejamos es OS X 10.5, no podremos emplear dicha versión de desarrollo, deberemos usar Firefox o Chrome (las cuales a su vez también podríamos usar con OS X 10.6).

Safari

Recordar que WebGL sólo es compatible con Safari en la versión de Snow Leopard OS X 10.6. Para poner en marcha WebGL en esta versión empleando Safari, realizaremos los siguientes pasos:

Asegurarse que al menos tenemos la versión 4 de Safari

Descargar la versión de desarrollo del enlace

Abriremos un terminal donde ejecutamos: *defaults write com.apple.Safari WebKitWebGLEnabled -bool YES*

Ejecutamos la aplicación de desarrollo recién instalada

Firefox

Se realiza del mismo modo que en Windows, con la salvedad que debemos descargar la versión oportuna para este sistema operativo.

Chrome

Vuelve a ser similar a Windows, el proceso a seguir es:

Descargaremos de la página de integración continua chrome-mac.zip

Descomprimimos. Abrimos una terminal, e iremos a la ruta donde lo hemos descomprimido

Nos aseguramos de no tener abierto el Chrome

Ejecutaremos el siguiente comando una vez estemos en esa ruta: *./Chromium.app/Contents/MacOS/Chromium*

2.3 Linux

Si tenemos una tarjeta gráfica con GPU ATI o Nvidia, no debería haber problema. Podremos emplear Firefox o Chrome. En caso de tener problemas, tendremos que usar Firefox con software de renderización, es decir, los gráficos 3D serán ejecutados en el procesador normal, no en la tarjeta gráfica.

Para el caso de chipset gráfico Intel, presente en muchos ordenadores portátiles, sus controladores no suelen ser compatibles con OpenGL, por tanto tampoco lo serán con WebGL ya que funciona a través de WebGL (en el futuro esto será distinto). Podríamos probar las instrucciones para Firefox o Chrome, pero no es probable que funcione; usaremos Firefox con software de renderización.

Firefox

Se realiza como en Windows, tener en cuenta que hay que bajar la versión oportuna para el sistema operativo.

Firefox con software de renderización

Los pasos a seguir son los siguientes:

Descargaremos la versión adecuada de Minifield para nuestra máquina en el siguiente enlace.

Lo instalaremos, cerrando previamente Firefox.

Emplearemos el gestor de paquetes para obtener Mesa, asegurándonos de que ha sido instalado y está actualizado a la última versión.

Iniciamos Minifield y vamos a la página: *about:config,* nos ayudamos del filtro de búsqueda e introducimos "WebGL"

Localizamos *webgl.enabled_for_all_sites* y lo ponemos a *true*

Localizamos *webgl.software_rendering* y lo ponemos a *true*

Localizamos *webgl.osmesalib* e introducimos la ubicación de la biblioteca compartida OSMesa, que suele ser algo como: */usr/lib/libOSMesa.so*

Chrome

Para conseguirla y poder emplear WebGL seguiremos los siguientes pasos:

Descargaremos de la página de integración continua chrome-linux.zip para la versión 32 bits o 64 bits, según cual emplemos.

Descomprimimos. Abrimos una ventana del terminal y vamos al directorio donde se ha descomprimido.

Nos aseguramos de no tener abierto el Chrome.

Ejecutaremos el siguiente comando una vez estemos en esa ruta: *./chromium*

3. Primer Programa

En primer lugar vamos a ver como incrustar nuestro código de webGL en cualquier página html.

Para incrustar el código html debemos:

Definir en la etiqueta <body> la función encargada gestionar el interface gráfico.

Definir un canvas, asignarle una etiqueta(para luego referenciarlo) y su tamaño.

Cargar los scripts externos que vayamos a necesitar (en este caso los propios de la API webGL y otro necesario para trabajar con matrices).

Una vez definidos estos aspectos ya podemos comenzar a programar nuestro propio script en webGL.

Al iniciar nuestra página debemos señalar que función hemos de cargar a la vez que el código, para nuestro caso será webGLStart().

En dicha función vemos como en primer lugar se ha de inicializar el canvas, llamando a la función initGL() donde, entre otras cosas, se comprueba que el navegador soporta webGL;

A continuación se inicializan los Shaders, así como los buffers. Nos vamos a centrar un poco en esta última función, puesto que es donde se definen los objetos que posteriormente se van a mostrar.

Ahora veremos los pasos a seguir para definir los buffers que permiten dibujar un objeto en la escena:

Crear un buffer para el objeto

Asignar el buffer actual con el que acabamos de crear

Definir los vértices del objeto

Cargar los datos en el buffer

Definir como se han introducido los datos en el buffer

Podemos modificar los vértices para obtener la figura que queramos, siempre y cuando tengamos cuidado de especificar correctamente como le estamos pasando esos vértices.

Para nuestro ejemplo vamos a definir un trapecio y un triángulo, pero podemos cambiar los vértices para obtener otros objetos si lo deseamos.

Una vez inicializados los bufferes tenemos que pasar a inicializar la escena con las funciones que podemos encontrar en webGLStart, para finalmente hacer una llamada a setInterval, donde se especifica que se ha de redibujar la escena cada 15 ms y cual es la función que contiene la escena, drawScene.

En esta última función en primer lugar debemos definir una vista y limpiarla, (cada vez que dibujamos en la pantalla hemos de asegurarnos que esta limpia).

A continuación definimos una perspectiva, esto es importante puesto que al definirla conseguimos que según la posición del objetos en el espacio estos se verán de distintos tamaños, si no hiciésemos esto independientemente de la coordenada z que pongamos el objeto se vería igual. Además de definir un ratio (ancho entre alto) especificamos los puntos en los cuales los objetos son visibles.

Una vez definida la perspectiva hemos de crear una matriz identidad, lo cual hacemos con loadIdentity(). Esto es una característica mas propia de OpenGL, todos los movimientos en el espacio, tanto desplazamientos como rotaciones, se pueden representar a través de una matriz de 4x4.

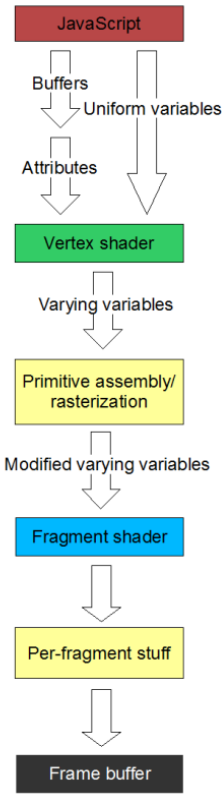
No nos vamos a parar a explicar como funciona OpenGL, simplemente mencionar que mediante operaciones sobre la matriz identidad podemos obtener todo tipo de desplazamientos y rotaciones, y en base a este principio opera OpenGL para la representación de objetos en el espacio. Como webGL no implementa las funciones necesarias para llevar esto a cabo hemos de crearlas nosotros, por eso necesitamos utilizar un script para trabajar con matrices, y además podemos ver como hay varias funciones que se encargan de "traducir" nuestros movimientos a valores en la matriz identidad.

Basándonos en esto podemos realizar movimientos en nuestra escena mediante mvTranslate, donde se llevaran a cabo las operaciones necesarias para dar a webGL una matriz que pueda entender y así poder realizar el movimiento deseado.

Para poder dibujar objetos en la escena hemos de seguir los siguientes pasos:
Ponernos en el punto del espacio en el que queramos dibujar
Asociarnos al buffer del objeto que deseamos dibujar
Definir que vamos a especificar con ese buffer
Mandar dibujar el array que contiene al objeto

Una vez visto estos pasos ya tenemos los conocimientos básicos para dibujar una escena, en posteriores lecciones veremos como ir añadiendo funcionalidades a nuestra escena.

4. Color



Antes de comenzar a analizar los cambios necesarios para añadir color a las figuras representadas en la lección anterior necesitamos tener un mínimo conocimiento de cómo WebGL es capaz de representar imágenes 3D a partir de los datos que obtiene del código HTML del navegador, para ello vamos a ayudarnos del diagrama de la parte inferior. El diagrama busca darnos una idea de cómo los datos de las funciones JavaScript se convierten en pixeles mostrados en el canvas de WebGL.
El proceso seguido es el siguiente: cada vez que llamamos a funciones del tipo drawArrays, WebGL procesa los datos que le pasamos como atributos a esa función, además de las variables uniformes que son utilizadas por las matrices model-view y de perspectiva con las que trabaja OpenGL, y todo ello se pasa a través del vertex shader. Esta acción se realiza para cada vértice con sus atributos correspondientes en ese instante.
El vertex shader trabaja con estos datos para diseñar el estado de la escena, y almacena los resultados en unas variables especiales denominadas varying variables. El vertex shader puede generar distinto número de varying variables, pero una en especial es obligatoria, gl_Position, la cual contiene las coordenadas de los vértices una vez que el shader las ha procesado.
Una vez generadas estas varying variables, WebGL trabaja con ellas para convertirlas en una imagen 2D y después llama al fragment shader para cada pixel de la imagen. Debemos notar que no todos los píxeles tienen un vértice asociado, pero WebGL realiza un proceso de interpolación lineal entre píxeles que permite delimitar correctamente las formas que se quieren representar. El propósito del fragmnet shader no es otro que retornar el color para cada uno de los puntos resultantes de este proceso de interpolación, y lo almacena en una varying variable denominada gl_FragColor.
Una vez realizado el fragment shader WebGL pasa los resultados al Frame buffer el cual los hace llegar a la tarjeta gráfica del equipo y son representados en la pantalla.

Esta pequeña explicación sobre el flujo de los datos desde la función JavaScript hasta que los resultados son mostrados en la pantalla nos sirve para hacernos una idea de cómo WebGL accede al color de los vértices, el cual está en la función JavaScript. La idea que debemos sacar de todo esto es que no se tiene un acceso directo desde JavaScript hasta el fragment shader, pero sí que podemos pasarle un número de variables con la información de los vértices, no sólo su posición, sino también su color. Una vez que tenemos una ligera idea de cómo WebGL trabaja con el color vamos a echar un vistazo al código de esta lección, comenzamos por los cambios en el vertex shader. Vemos como además del atributo que hace referencia a la posición, aparece uno nuevo para trabajar con el color aVertexColor.

```
varying vec4 vColor;
void main(void) {
   gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
   vColor = aVertexColor;
 }
```

A parte del cálculo de la posición de la misma manera que se explicó en la lección anterior, el main del vertex shader no hace más que pasar el atributo de color a una varying variable de salida.

```
gl_FragColor = vColor;
```

Una vez realizado este proceso para cada vértice, WebGL hace la interpolación para generar los fragmentos los cuales son pasados al fragment shader.

Vemos como se toma el valor de la variable vColor que contiene el color generado en el proceso de interpolación, y lo retorna como el color de cada fragmento, es decir, el color de cada pixel.

Vamos a fijarnos ahora en una pequeña variación introducida en la función initShaders, en las líneas siguientes se puede ver que aparece un nuevo atributo que hace referencia al color.

```
shaderProgram.vertexColorAttribute=
    gl.getAttribLocation(shaderProgram, "aVertexColor");
gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
```

En la primera lección se explicó cómo esta función inicializaba los shaders con la información indicada por el vertex shader para cada vértice, en esta nueva lección necesitamos inicializar otro nuevo parámetro obviamente, el color.

Por último vamos a ver el principal cambio que debemos introducir para trabajar con color en la función initBuffers, la cual recordamos que era la encargada de cargar la información de las figuras que queremos dibujar en la escena. Aparecerán ahora dos nuevos buffers encargados cada uno de ellos de almacenar el color de las dos figuras a representar.

```
var triangleVertexColorBuffer;
var squareVertexColorBuffer;
```

Tras crear los buffers de color e instar al correspondiente bind al igual que habíamos hecho con los buffers de posición, debemos cargar los datos en ellos.

```
triangleVertexColorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
var colors = [
   0.0, 0.0, 1.0, 1.0,
   0.0, 1.0, 0.0, 1.0,
   1.0, 0.0, 0.0, 1.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors),
              gl.STATIC_DRAW);
triangleVertexColorBuffer.itemSize = 4;
triangleVertexColorBuffer.numItems = 3;
```

Para esto utilizamos una nueva matriz de colores, la cual almacenará en cada 'fila' la información del color de un vértice. Esta información consta de 4 datos:

Primera columna → Cantidad de color rojo.

Segunda columna → Cantidad de color verde.

Tercera columna → Cantidad de color azul.

Cuarta columna → Parámetro Alpha, que indica la opacidad.

Notar que esta matriz ha de definirse, al igual que se hacía con las de posición, con la ayuda de las variables itemSize y numItems del buffer.

Como hemos explicado en la introducción de esta lección, WebGL realiza una interpolación lineal entre vértices para poder dibujar la figura, esta interpolación también afecta al color. Así podemos verlo en el resultado de nuestra lección, a la figura triangular se le definen colores diferentes para cada vértice, de ahí que el color vaya cambiando progresivamente de uno a otro. En cambio el cuadrilátero tiene sus vértices del mismo color, por lo que la figura entera presenta ese color.

```
colors = [ ]
   for (var i=0; i < 4; i++) {
       colors = colors.concat([0.0, 1.0, 0.5, 1.0]);}
```

Una vez tenemos los cuatro buffers cargados con la información de nuestras dos figuras, vemos como la función drawScene toma esta información para representar la escena.

```
gl.bindBuffer(gl.ARRAY_BUFFER,triangleVertexColorBuffer);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
            triangleVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexColorBuffer);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
             squareVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

Con estos pequeños cambios introducidos en el código ya podemos trabajar con el color de nuestras figuras como nos plazca. Para modificar el color simplemente debemos cargar los valores adecuados en el buffer de color para cada vértice como hemos explicado.

5. Primeros movimientos

La forma de animar una escena 3D en WebGL consiste en ir dibujando repetidamente un dibujo distinto cada vez. No consiste en decir estamos en el punto X, lleva la imagen al punto Y, hay que hacer toda la animación intermedia.

Hasta ahora empleábamos la función *drawScene* para dibujar todo, del modo:

```
setInterval(drawScene, 15);
```

Lo que necesitamos hacer es cambiar el código para que cada vez que llamemos a dicha función cada 15 mseg de una sensación de movimiento a nuestras figuras. Por tanto la mayor parte de los cambios con respecto a la lección anterior, se encontrarán en la función drawScene.

Justo antes de la declaración de la función definiremos dos variables globales:

```
var rTri = 0;
var rSquare = 0;
```

Que usaremos para seguir la rotación de las figuras, comienzan en cero grados y luego irán aumentando a medida que se hagan girar. Más adelante veremos una forma más elegante de de estructurar estas variables globales.

El siguiente cambio en *drawScene* se hace en el punto donde dibujamos las figuras, tendremos que añadir estas líneas en el sitio oportuno:

```
mvPushMatrix();
mvRotate(rTri,[0,1,0]);
mvPopMatrix();
```

mvRotate(rTri, [0, 1, 0]). Esta función se codifica en JavaScript. Parte del estado actual de la figura almacenado en la matriz model-view e indica que rotaríamos rTri grados sobre la vertical ([0,1,0]).

Como no siempre querremos movernos a partir del estado inicial, podríamos querer partir de un estado anterior almacenado, usaremos las funciones mvPushMatrix() y mvPopMatrix().

mvPushMatrix() pone la matriz de estado actual en una pila.

mvPopMatrix() se deshace de la matriz actual, toma uno de la parte superior de la pila, y lo restaura.

Tener los dibujos anidados almacenados en una pila significa que podremos recuperarlos en cualquier momento. De este modo, al acabar de rotar el triángulo podremos restablecer la matriz model-view. De igual forma, podremos hacer girar el cuadrado sobre su eje horizontal:

```
mvPushMatrix();
mvRotate(rSquare, [1,0,0]);
```

```
mvPopMatrix();
```

Estos son los cambios que realizaremos dentro de la función drawScene.

Pero obviamente para animar la escena debemos cambiar los valores rTri y rSquare a lo largo el tiempo, de esta forma cada vez que se dibuja la escena es ligeramente diferente. Esto lo realizamos llamando a la función animate de forma regular:

```
var lastTime=0;
function animate()
{
    var timeNow=new Date() getTime();
    if (lastTime !=0){
        var elapsed = timeNow - lastTime;
        rtri += (360*elapsed)/1000.0;
        rSquare += (75*elapsed)/1000.0;
    }
    lastTime = timeNow;
}
```

Esto es la forma más adecuada de animar, en concreto el triángulo gira a 360º por minuto y el cuadrado a 75º por minuto. Haciéndolo así todos ven el mismo tipo de movimiento independientemente de la potencia de la máquina con que se trabaje. La animación será más continua cuanta mayor potencia desarrolle.

El siguiente cambio consiste en llamar a animate y drawScene de forma regular. Sustituimos la llamada que hacíamos a drawScene por una nueva función llamada tick que llamará a ambas cada 15 milisegundos:

```
function tick(){
    drawScene();
    animate();
}
```

Tras ello en la función webGLStart(), cambiamos setInterval(drawScene, 15); por:

```
setInterval(tick, 15);
```

Estos son los cambios añadidos para lograr la animación. Veremos ahora el código de apoyo que hemos tenido que añadir, definimos una pila para almacenar las matrices, así como las operaciones Push y Pop de forma adecuada:

```
var mvMatrixStack = [];
function mvPushMatrix (m) { ... }
function mvPopMatrix (m) { ... }
```

Creamos también la función mvRotate:

```
function mvRotate (ang, v){
    var arad = ang*Math.PI / 180.0;
    var m = Matrix.Rotation (arad, $V([ v[0], v[1], v[2] ])).ensure4x4();
    multMatrix (m);
}
```

Es bastante simple, el trabajo de crear una matriz para representar la rotación corre a cargo de la librería *Sylvester*.

6. Figuras 3D

Para el dibujo de una escena en 3D en primer lugar vamos a cambiar el nombre de los objetos de la escena, con el fin de no confundirlos con los anteriores en 2D. Para nuestro ejemplo vamos a dibujar una pirámide y un cubo, por lo que ahora llamaremos a las variables:

```
rPyramid += (360 * elapsed) / 1000.0;
rCube += (75 * elapsed) / 1000.0;
```

A continuación tenemos que modificar las variables definidas antes de drawScene, y posteriormente dentro de esta función, donde se realizan los movimientos.

```
var rPyramid = 0;
var rCube = 0;

mvRotate(rPyramid, [0, 1, 0]);
mvRotate(rCube [1, 1, 0]);
```

Luego hemos de cambiar las variables referentes a los buffers para hacerlas coincidir con la nomenclatura seguida.

```
gl.bindBuffer(gl.ARRAY_BUFFER, pyramidVertexPositionBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
    pyramidVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, pyramidVertexColorBuffer);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
    pyramidVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);
setMatrixUniforms();
gl.drawArrays(gl.TRIANGLES, 0, pyramidVertexPositionBuffer.numItems);
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
    cubeVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexColorBuffer);
gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
    cubeVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

Por otro lado, el mayor cambio que debemos hacer es la adicción de nuevos vértices para conseguir las figuras deseadas. Antes de ponerse a definir los vértices del cubo, vamos a evaluar como podemos definirlo.

1. En primer lugar podríamos definir el cubo, como una tira de triángulos (característica de OpenGL). Esto parece relativamente sencillo, siempre y cuando únicamente queramos un color para todo el cubo. Partimos de los vértices anteriores, y con dos puntos más tendremos otra cara, y así sucesivamente hasta tener por completo el cubo. Este es un método bastante efectivo y cómodo, pero como habíamos mencionado presenta la problemática de ser únicamente valido cuando el cubo solo tiene un color.

2. El segundo método más lógico sería el de dibujar 6 caras independientes, de tal forma que cada una tenga un color, y evidentemente juntas formen un cuadrado. Este, a priori, es un buen método, pero realmente presenta un problema, tenemos que hacer muchas llamadas para dibujar los objetos. Para nuestros ejemplos, como son sencillos no sería un problema muy serio, pero si que nos podemos encontrar con dificultades en códigos más complejos, por lo que veremos a continuación la forma óptima de realizar dibujos en 3 dimensiones.

3. La opción final pasa por dibujar un cubo mediante 6 cuadrados, y cada uno de estos cuadrados estar compuesto por dos triángulos. Esto en principio puede parecer más complejo que el caso anterior, pero como vamos a comprobar si definimos bien los vértices esto se puede hacer de una forma mas sencilla, porque podemos compartir vértices entre triángulos, mandarlos en un bloque, y así que se podrá dibujar de una pasada. Es parecido a definir tiras de triángulos, pero en este caso aunque compartan vértices, estos son independientes, por lo que se podrán definir colores distintos para cada cara.

Como hemos visto, en esta última opción en la practica tendremos 4 vértices en el buffer, pero con ellos podemos especificar 6, para definir dos triángulos independientes, esto lo conseguimos compartiendo vértices en la definición de los triángulos. Será algo como, "con los vértices 1,2 y 3 dibújame un triángulo, y luego con los vértices 1,2 y 4 dibújame otro". Para conseguir esto haremos una llamada a element array buffer y una nueva llamada a drawelements. Así que haremos un bind al buffer de elementos, para a continuación dibujar dichos elementos.

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,cubeVertexIndexBuffer);
setMatrixUniforms();

gl.drawElements(gl.TRIANGLES,cubeVertexIndexBuffer.numItems,
gl.UNSIGNED_SHORT, 0);
```

Las nuevos buffers seran las siguientes: (hemos de crear uno para los índices de los vértices del cubo):

```
var pyramidVertexPositionBuffer;
var pyramidVertexColorBuffer;
var cubeVertexPositionBuffer;
var cubeVertexColorBuffer;
var cubeVertexIndexBuffer;
```

A continuacion pasamos a definir los valores de los vertices de la piramide para cada cara, teniendo en cuenta que hemos de cambiar el valor de numItems:

```
pyramidVertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, pyramidVertexPositionBuffer);
var vertices = [
// Front face
```

```
    0.0,   1.0,   0.0,
    -1.0, -1.0,   1.0,
    1.0, -1.0,   1.0,
  // Right face
    0.0, 1.0, 0.0,
    1.0, -1.0, 1.0,
    1.0, -1.0, -1.0,
  // Back face
    0.0, 1.0, 0.0,
    1.0, -1.0, -1.0,
    -1.0, -1.0, -1.0,
  // Left face
    0.0, 1.0, 0.0,
    -1.0, -1.0, -1.0,
    -1.0, -1.0, 1.0 ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
        gl.STATIC_DRAW); pyramidVertexPositionBuffer.itemSize = 3;
  pyramidVertexPositionBuffer.numItems = 12;
```

De igual forma creamos el buffer con los vértices para el color:

```
  pyramidVertexColorBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, pyramidVertexColorBuffer);
   var colors = [
  // Front face
    1.0, 0.0, 0.0,
    1.0,0.0, 1.0,
    0.0, 1.0, 0.0,
    0.0, 1.0, 1.0,
  // Right face
    1.0, 0.0, 0.0, 1.0,
    0.0, 0.0, 1.0, 1.0,
    0.0, 1.0, 0.0, 1.0,
  // Back face
    1.0, 0.0, 0.0, 1.0,
    0.0, 1.0, 0.0, 1.0,
    0.0, 0.0, 1.0, 1.0,
  // Left face
    1.0, 0.0, 0.0, 1.0,
    0.0, 0.0, 1.0, 1.0,
    0.0, 1.0, 0.0, 1.0 ];

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors),
        gl.STATIC_DRAW); pyramidVertexColorBuffer.itemSize = 4;
```

```
pyramidVertexColorBuffer.numItems = 12;
```

Una vez terminado de definir la pirámide pasamos a trabajar con el cubo:

```
cubeVertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);

vertices = [
// Front face
-1.0, -1.0, 1.0, 1.0,
 -1.0, 1.0, 1.0, 1.0,
 1.0, -1.0, 1.0, 1.0,
// Back face
-1.0, -1.0, -1.0,
 -1.0, 1.0, -1.0,
1.0, 1.0, -1.0,
 1.0, -1.0, -1.0,
// Top face
-1.0, 1.0, -1.0,
 -1.0, 1.0, 1.0,
1.0, 1.0, 1.0,
1.0, 1.0, -1.0,
// Bottom face
 -1.0, -1.0, -1.0,
1.0, -1.0, -1.0,
 1.0, -1.0, 1.0,
-1.0, -1.0, 1.0,
// Right face
1.0, -1.0, -1.0,
1.0, 1.0, -1.0,
1.0, 1.0, 1.0,
1.0, -1.0, 1.0,
// Left face
-1.0, -1.0, -1.0,
-1.0, -1.0, 1.0,
-1.0, 1.0, 1.0,
-1.0, 1.0, -1.0, ];

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
gl.STATIC_DRAW); cubeVertexPositionBuffer.itemSize = 3;

cubeVertexPositionBuffer.numItems = 24;
```

La definición del buffer de color es algo mas complicada porque usamos un bucle para crear la lista de vértices para el color, por lo que hemos de especificar cada color cuatro veces para cada vértice.

```
cubeVertexColorBuffer = gl.createBuffer(); gl.bindBuffer(gl.ARRAY_BUFFER,
cubeVertexColorBuffer);
var colors = [
[1.0, 0.0, 0.0, 1.0], // Front face
[1.0, 1.0, 0.0, 1.0], // Back face
[0.0, 1.0, 0.0, 1.0], // Top face
[1.0, 0.5, 0.5, 1.0], // Bottom face
[1.0, 0.0, 1.0, 1.0], // Right face
[0.0, 0.0, 1.0, 1.0], // Left face ];

var unpackedColors = []
for (var i in colors) {
   var color = colors[i];
   for (var j=0; j < 4; j++) {
      unpackedColors = unpackedColors.concat(color); } }

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(unpackedColors),
      gl.STATIC_DRAW); cubeVertexColorBuffer.itemSize = 4;

cubeVertexColorBuffer.numItems = 24;
```

Por ultimo definimos el array de elementos que anteriormente mencionamos.

```
cubeVertexIndexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVertexIndexBuffer);

var cubeVertexIndices = [
0, 1, 2, 0, 2, 3,          // Front face
4, 5, 6, 4, 6, 7,          // Back face
8, 9, 10, 8, 10, 11,       // Top face
12, 13, 14, 12, 14, 15, // Bottom face
16, 17, 18, 16, 18, 19, // Right face
20, 21, 22, 20, 22, 23  // Left face ]

gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
      new Uint16Array(cubeVertexIndices), gl.STATIC_DRAW);

cubeVertexIndexBuffer.itemSize = 1; cubeVertexIndexBuffer.numItems = 36;
```

Hay que recordar que cada numero de este buffer es un índice para la posición del vértice y de su color. Como podemos ver los dos primeros triángulos que definimos son adyacentes y además al ser del mismo color dan como lugar a un cuadrado, y al igual para el resto de caras.

7. Texturas

En esta nueva lección de nuestro tutorial vamos a ver cómo podemos hacer para añadir una textura a los objetos 3D que hemos ido creando, la textura la cargaremos desde un archivo de imagen el cual instaremos desde nuestro programa. Para entender cómo trabajan las texturas debemos recordar lo que explicábamos en la lección 3 a cerca del color, en esta lección habíamos visto como el color era especificado por el fragment shader. En este caso lo que debemos hacer es cargar la imagen y enviársela al fragment shader, pero además debemos indicarle también algo de información acerca de la misma para que pueda trabajar correctamente con ella.

Vamos a empezar a echarle un vistazo a nuestro código para ir viendo las modificaciones que vamos a ir añadiendo para cargar las texturas. Recordemos que la función que inicializa la ejecución de la página JavaScript es wegGLStart, pues si echamos un vistazo a esta función podemos ver como es ahí donde se carga la textura mediante una nueva función.

```
initTexture();
```

Vamos a analizar esta nueva función:

```
var parafusaTexture;
function initTexture() {
parafusaTexture = gl.createTexture();
parafusaTexture.image = new Image();
parafusaTexture.image.onload = function() {
handleLoadedTexture(parafusaTexture);
}
 parafusaTexture.image.src = "parafusa.gif";
}
```

Vemos que estamos creando una variable global para manejar la textura, aunque obviamente en un ejemplo más complejo no podríamos usar variables globales ya que tendríamos múltiples texturas, pero para nuestro ejemplo resulta más intuitivo. Lo primero que hacemos es crear una referencia a la textura mediante gl.CreateTexture() y después creamos un objeto de imagen JavaScript para asociarlo a nuestra textura, una vez más nos estamos aprovechando de la capacidad de JavaScript para establecer cualquier campo de los objetos ya que por defecto los objetos de textura no tienen un campo imagen, pero sí que podemos crearlo nosotros.

El paso siguiente será obviamente será cargarle al objeto de imagen la imagen que contendrá, pero antes debemos llamar a una función de callback, la cual nos asegurará que la imagen ha sido cargada completamente. Una vez que se ha establecido correctamente indicamos el src de la imagen y listo.

Notar que la imagen se cargará de forma asíncrona, es decir, el código que indica el src de la imagen retornará inmediatamente, pero dejará un hilo ejecutandose en segundo plano que irá cargando la imagen del servidor. Una vez se ha cargado completamente se insta a la función de callback y esta llama a handleLoadedTexture. Vamos ahora a fijarnos en esta última función:

```
function handleLoadedTexture(texture){
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
        gl.UNSIGNED_BYTEtexture.image);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
        gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
        gl.NEAREST);
gl.bindTexture(gl.TEXTURE_2D, null);
}
```

Lo primero que hacemos en esta función es decirle a WebGL cual es la textura con la que estamos trabajando, para eso se llama a la función bindTexture, la cual actúa de manera similar a como lo hacía el bind de los buffers en la primera lección. Tras esto esta función le indica a WebGL que todas las imágenes que tenemos cargadas como texturas deben voltearse a posición vertical para que se representen correctamente sobre nuestro objeto.

El siguiente paso será cargar la imagen que acabamos de obtener del servidor al espacio de texturas de nuestra tarjeta gráfica mediante la función textImage2D. Este función recibe como parámetros el tipo de imagen que estamos usando, el nivel de detalle deseado, el formato que queremos que represente la tarjeta gráfica, el tamaño de cada canal RGB de la imagen y por último la imagen a cargar. Veremos estos parámetros con más detalle más adelante.

Las siguientes dos líneas sirven para indicar los parámetros de escalado para la textura. La primera de ellas le indica a WebGL como debe escalar la imagen cuando la textura en la escena es mayor que su tamaño original, la segunda hace lo mismo pero en el caso que el escalado deba ser hacia una imagen menor. En este caso se usa la aproximación NEAREST, la cual no es presenta buena calidad pero si es más rápida en equipos de menores prestaciones. Por último se inicializa curretTexture a null para dejar la variable libre para futuras texturas.

Hasta aquí hemos visto todo el código necesario para cargar correctamente las texturas que queremos utilizar, el siguiente paso será mover estas texturas a los buffers para representarlas en nuestra escena. Para ello vamos a analizar los cambios introducidos en intiBuffers. Para esta lección vamos a quedarnos únicamente con la figura del cubo para poder ver las texturas mejor, notar que ya no necesitamos los buffers de color que habíamos utilizado ya que ahora serán reemplazados por un buffer de coordenadas de textura:

```
cubeVertexTextureCoordBuffer = gl.createBuffer();
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexTextureCoordBuffer);
var textureCoords = [
   // Para una cara
   0.0, 0.0,
   1.0, 0.0,
   1.0, 1.0,
   0.0, 1.0,
  //Igual para las demás
 …
  ];
```

Fijándonos en este nuevo buffer vemos que se definen dos atributos para cada vértice. Lo que este par de valores indican es en qué coordenada cartesiana (x,y) de la textura se sitúa ese vértice. La coordenadas de la textura están normalizadas, por tanto (0,0) será la esquina de abajo a la izquierda y (1,1) la esquina de arriba a la derecha de la imagen.

Vamos a ver ahora cómo se modifica drawScene observando los cambios que realmente nos interesan para esta lección, es decir, aquellos que hacen referencia a la textura de la figura. En initBuffers habíamos inicializado adecuadamente las coordenadas de textura por lo que ahora debemos realizar un bind hacia ellos con los atributos adecuados de manera que los shaders de la escena puedan utilizar correctamente las texturas.

```
gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexPositionBuffer);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
      cubeVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

Con esto WebGL ya conoce qué bit de la textura se corresponde con cada vértice, necesitamos ahora indicarla qu use la textura que hemos cargado al principio del programa para luego proceder a dibujar el cubo:

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, parafusaTexture);
gl.uniform1i(shaderProgram.samplerUniform, 0);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeVertexIndexBuffer);
setMatrixUniforms();
gl.drawElements(gl.TRIANGLES, cubeVertexIndexBuffer.numItems,
            gl.UNSIGNED_SHORT, 0);
```

Notar que WebGL puede manejar hasta 32 texturas en llamadas a funciones del tipo gl.drawElements, y estas estan numeradas como TEXTURE0...TEXTURE31. En este caso se le indica a la función que use la textura 0 que es la única con la que estamos trabajando en esta ocasión. La funcion gl.uniform1i le indica al shader que use esta textura. Una vez se ha indicado la textura a utilizar ya no queda más que proceder a dibujar el cubo como se explicó en las lecciones anteriores.

Por último los únicos cambios que nos queda por comentar son aquellos que hacen referencia a los shaders, comenzamos echándole un vistazo al vertex shader:

```
attribute vec2 aTextureCoord;
varying vec2 vTextureCoord;
void main(void) {
gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
vTextureCoord = aTextureCoord;}
```

Los cambios que observamos aquí son muy similares a los introducidos en la segunda lección cuando comenzamos a utilizar color. Todo lo que estamos haciendo es aceptar las coordenadas de textura como un par de atributos por vértice y pasarlo hacia el fragment shader como una varying variable.

Una vez se ha realizado este paso para cada vértice, WebGL trabajará para cada fragmento (pixel) y realizará la interpolación lineal entre vértices para representar los puntos intermedios. Esta interpolación también afectará a las coordenadas de textura con lo que cada fragmento intermedio tendrá sus coordenadas de textura apropiadas.

Por último, al igual que habíamos visto con el color vamos a fijarnos en fragment shader:

```
#ifdef GL_ES
precision highp float;
#endif
varying vec2 vTextureCoord;
uniform sampler2D uSampler;
void main(void) {
   gl_FragColor = texture2D(uSampler, vec2(vTextureCoord.s, vTexture-
Coord.t));}
```

Aquí se toman las coordenadas de textura interpoladas y se pasa a una variable de tipo sampler, la cual es la manera en que OpenGL representa las texturas. En drawScene se había indicado qué textura debía utilizar el shader mediante la función texture2D, por lo que lo único que hace el shader es usar esta función para obtener cual es el color apropiado para la coordenada de textura asociada al pixel que se esté representando. Llegados a este punto WebGL ya sabe el color que debe representar para cada pixel por lo que lo único que queda es representarlo en la pantalla y podrá representar correctamente la textura.

8. Bibliografía

8.1 OpenGL

http://www.khronos.org/files/opengl41-quick-reference-card.pdf
http://worldspace.berlios.de/fase1/index.html
http://www.scribd.com/doc/31291543/OpenGL-Basico
http://sabia.tic.udc.es/gc/Tutorial%20OpenGL/tutorial/cap1.htm

8.2 WebGL

http://www.khronos.org/webgl/
http://khronos.org/webgl/wiki/Main_Page
http://playwebgl.com/demos/worlds-of-webgl/
http://www.inmensia.com/blog/20100619/webgl_cheat_sheet.html