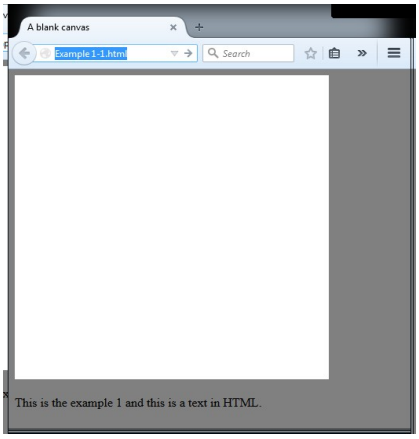
Tutoriales de WebGL

1 Preparando WebGL

1.1 Creación del Lienzo (Canvas)

Este primer ejemplo muestra la preparación de un lienzo (Canvas en HTML) que será el que utilice WebGL para dibujar en un futuo. Realmente no contiene código propio de WebGL. Casi todo el código se corresponde con HTML.

Os animamos a estudiar el código y aprender a retocar el código HTML.



Código Relevante

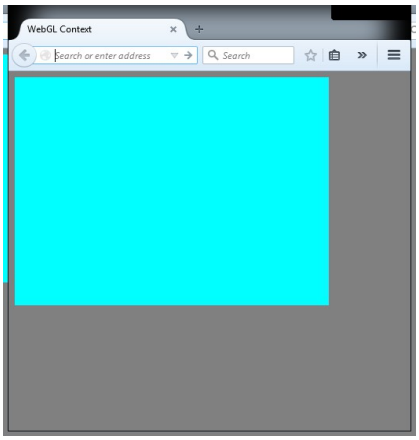
Si observamos el código del ejemplo "***Example 1-1.html***" veremos HTML básico con la creación de un lienzo para el dibujo. Es la etiqueta <canvas> que hemos resaltado en la siguiente figura.

```
<!doctype html>
<html>
<head>
    <title>A blank canvas</title>
    <style>
        body {
            background-color: grey;
        }

        canvas {
            background-color: white;
        }
    </style>
</head>
<body>
    <canvas id="my-canvas" width="400" height="400">
        Your browser does not support the HTML5 canvas element.
    </canvas>
<p>This is the example 1 and this is a text in HTML.</p>
</body>
</html>
```

Ejercicios

1. Cambia el título de la página web. Por ejemplo, pon tu nombre.
2. Cambia los colores de fondo tanto de la página web como del lienzo. Por ejemplo, haz la página web de color blanco y el lienzo en nego.
3. Cambia el tamaño del lienzo (ahora está a 400x400). Pon algo que te resulte cómodo. Por ejemplo 600 x 600.
4. ¿Sabrías crear dos canvas dentro de la misma página web?

1.2 Creación del contexto WebGL



Este segundo ejemplo continua con la preparación de WebGL. Realmente apenas contiene código propio de WebGL. Casi todo el código se corresponde con HTML y un poco de Javascript.

El código tiene dos puntos de interés. Uno de ellos es la la llamada a la creación del contexto ("canvas.getContext") dentro de la función "*setupWebGL*". El otro punto que queremos destacar es el borrado de la pantalla. Este estado de WebGL se irá complicando en los ejercicios posteriores, pero en este primer ejercicio unicamente se fija el color de borrado. (el método en OpenGL es glClear, en Javascript lo veremos como gl.Clear)

Código Relevante

Si observamos el código del ejemplo "***Example 1-2.html***" veremos una nueva etiqueta (<script>...</script>) con código Javascript. Cuando la página haya sido cargada se ejecuta la función "***setupWebGL***". A continuación se obtiene una referencia al canvas (lienzo) y la creación del contexto de WebGL. Si todo es correcto (la variable gl es distinta de null) entonces se pone el color de borrado (**gl.clearColor**) y se borra la pantalla con ese color (**gl.clear**).

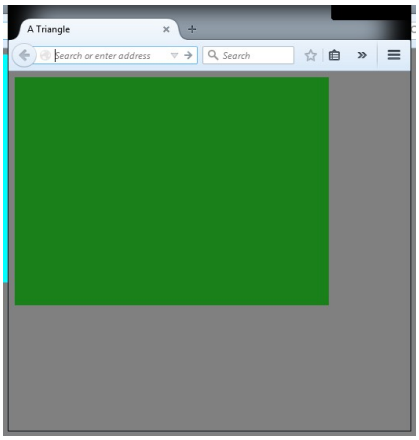
```
<script>
    window.onload = setupWebGL;
    var gl = null;


    function setupWebGL()
    {
        var canvas = document.getElementById("my-canvas");
        try{
            gl = canvas.getContext("experimental-webgl");
        }catch(e){
        }
        if(gl)
        {
            //set the clear color
            gl.clearColor(0.0, 1.0, 1.0, 1.0);
            gl.clear(gl.COLOR_BUFFER_BIT);
        }else{
            alert( "Error: Your browser does not appear to support WebGL.");
        }
    }
</script>
```

Ejercicios.

Os animamos a estudiar el código y aprender a distinguir el código que se corresponde con WebGL con el que es propio de Javascript.

5. ¿Sabrías cambiar el nombre del canvas? Tienes que modificar en dos líneas
6. Cambia el color del fondo de WebGL a un rojo puro (los parámetros de gl.clearColor se refieren a RGBA o red-green-blue-alpha, donde alpha es el nivel de transparencia).

1.3 Aparecen los Shaders



Este ejemplo continua con la preparación de WebGL. Este ejemplo tiene dos objetivos principales. Por una parte organizamos el código con una estructura más cómoda y por otra introducimos las etiquetas con los shaders de vértices y fragmentos. Asegúrate de entender correctamente el concepto de Shader y pregunta a tu profesor si tienes dudas antes de seguir profundizando en el código.

Código Relevante

Este ejemplo se corresponde con el fichero "***Example 1-3.html***". Vemos que hemos añadido dos nuevas etiquetas de tipo script pero en lugar de contener código Javascript hemos introducido un código propio de los shaders. Observa cómo le decimos a HTML que ese código es "especial" indicando en el type que son "shader". Es pronto para entender su funcionamiento y dejamos para futuros ejemplos su explicación más detallada.

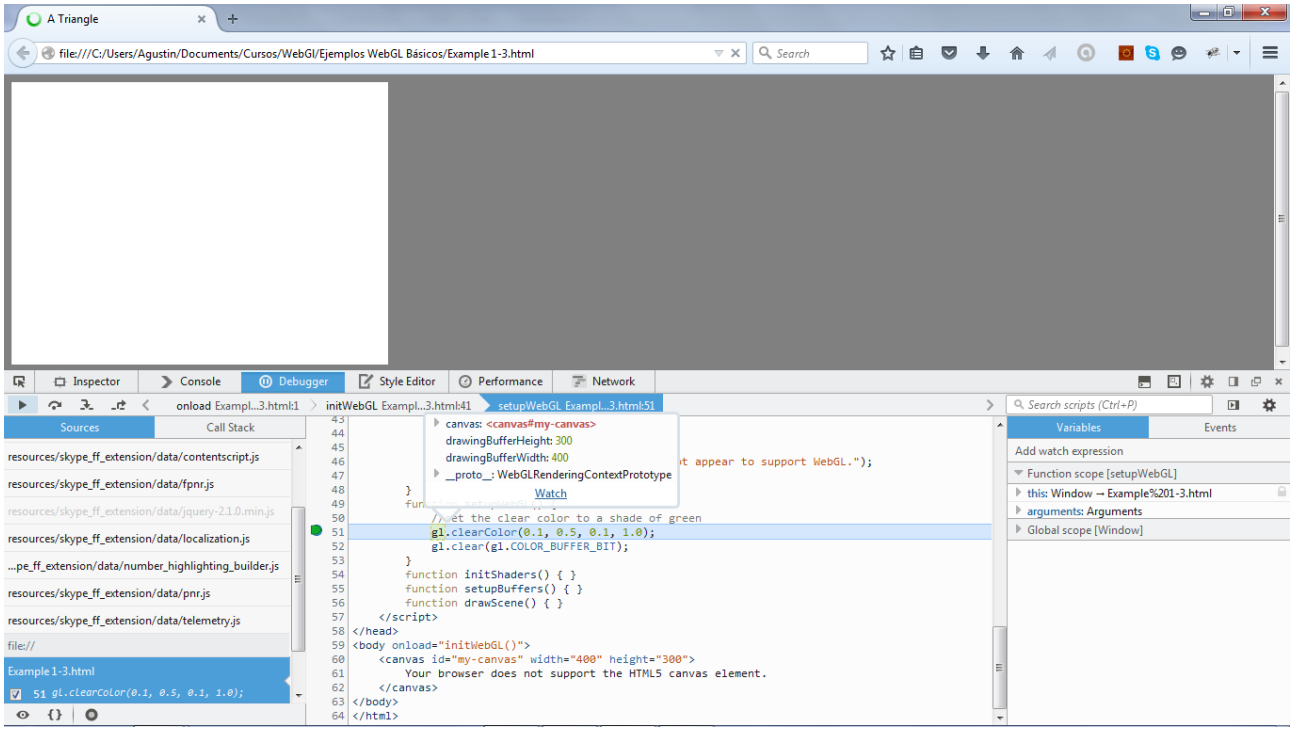
Cuando la página haya sido cargada se ejecuta la funcion "***initWebGL***". Mira la creación del canvas para entender cómo se conecta el canvas con el método. El resto del código es muy similar al anterior pero lo hemos organizado en funciones que iremos rellenando.

```
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    void main(void) {
        gl_Position = vec4(aVertexPosition, 1.0);
    }
</script>
<script id="shader-fs" type="x-shader/x-fragment">
    void main(void) {
        gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
</script>
    var gl = null,
    canvas = null,
    glProgram = null,
    fragmentShader = null,
    vertexShader = null;
    var vertexPositionAttribute = null,
    trianglesVerticeBuffer = null;
    function initWebGL() {
        canvas = document.getElementById("my-canvas");
        try {
            gl = canvas.getContext("webgl") ||
                 canvas.getContext("experimental-webgl");
        } catch (e) {
        }
        if (gl) {
            setupWebGL();
            initShaders();
            setupBuffers();
            drawScene();
        } else {
            alert("Error: Your browser does not appear to support WebGL.");
        }
    }
    function setupWebGL() {
        //set the clear color
        gl.clearColor(0.1, 0.5, 0.1, 1.0);
```

```
            gl.clear(gl.COLOR_BUFFER_BIT);
        }
        function initShaders() { }
        function setupBuffers() { }
        function drawScene() { }
    </script>
```

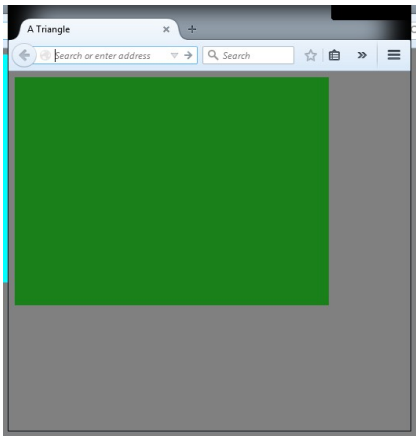
Ejercicios.

Os animamos a aprender a depurar el código Javascript. Todos los navegadores incluyen esa capacidad. La siguiente figura muestra un proceso de depuración con Firefox y asociado a este ejemplo:



7. ¿Sabrías reproducir el proceso de depuración hasta llegar al mismo punto que la pantalla anterior? Ten en cuenta que la pantalla puede cambiar si utilizas otro navegador o incluso otra versión de Firefox. Pero lo importante es que sepas poner un punto de parada en la línea correspondiente al "*clearColor*" y que puedas ver el contenido de la variable "*gl*".
8. ¿Sabrías explicar la razón de que en la pantalla anterior el fondo del canvas se vea de color blanco y no verde como sucede si seguimos ejecutando el ejemplo.
9. Javascript tiene un método muy útil para generar mensajes y saber lo que estamos ejecutando. Prueba a poner varios "alert" dentro de las funciones que ahora mismo están vacíos. Por ejemplo, inserta "alert("Hola, estoy en el metodo drawScene."); " dentro del método "drawScene".

1.4 Compilamos los Shaders



Seguimos preparando el dibujo con WebGL. En este ejemplo compilamos los shaders para que estén disponibles antes de nuestro dibujo. Este ejemplo compila y almacena los shaders dentro de la memoria de WebGL.

Si abrimos la pagina web no veremos ningún cambio aparente ya que toda la funcionalidad está trabajando con WebGL internamente.

Código Relevante

Si observamos el código del ejemplo "***Example 1-4.html***" veremos que el principal cambio afecta al método initShaders. Hemos añadido una nueva función que realiza la compilación denominada "makeShader". Esta función crea un shader, introduce el código del shader y lo compila. Pregunta a tu profesor por esa funcionalidad. También queremos destacar la forma que hemos utilizado para cargar los shaders en Javascript. En este caso, se localiza el elemento HTML denominado 'shader-fs' o 'shader-vs'. Una vez localizado el elemento se extrae el contenido mediante innetHTML. Eso retorna un string con el código del shader a compilar.

```
function initShaders() {
    //get shader source
    var fs_source = document.getElementById('shader-fs').innerHTML,
    vs_source = document.getElementById('shader-vs').innerHTML;
    //compile shaders
    vertexShader = makeShader(vs_source, gl.VERTEX_SHADER);
    fragmentShader = makeShader(fs_source, gl.FRAGMENT_SHADER);
    //create program
    glProgram = gl.createProgram();
    //attach and link shaders to the program
    gl.attachShader(glProgram, vertexShader);
    gl.attachShader(glProgram, fragmentShader);
    gl.linkProgram(glProgram);
    if (!gl.getProgramParameter(glProgram, gl.LINK_STATUS)) {
        alert("Unable to initialize the shader program.");
    }
    //use program
    gl.useProgram(glProgram);
}
function makeShader(src, type) {
    //compile the vertex shader
    var shader = gl.createShader(type);
    gl.shaderSource(shader, src);
    gl.compileShader(shader);
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert("Error compiling shader: " + gl.getShaderInfoLog(shader));
    }
    return shader;
}
```

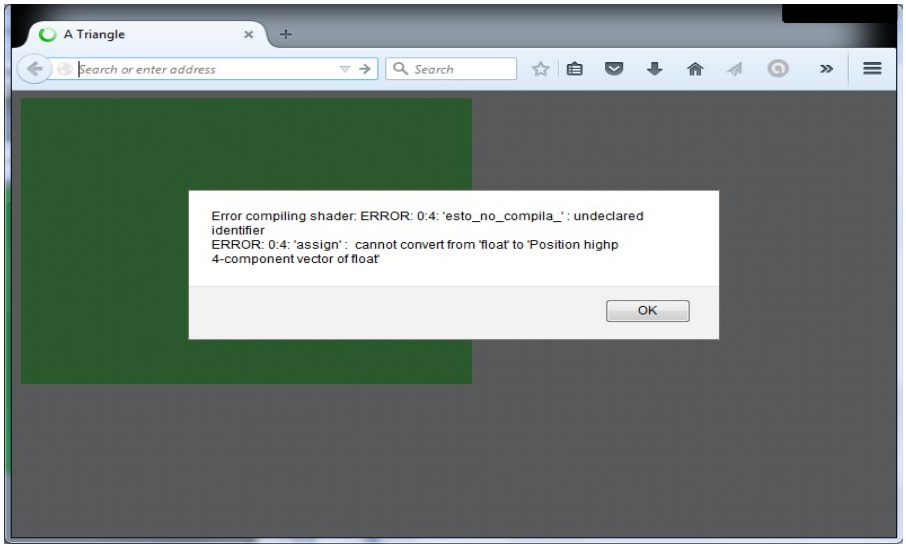
Ejercicios.

Observa que en este ejemplo tenemos una mezcla de HTML, Javascript, WebGL y "OpenGL Shading Language" (el lenguaje utilizado en los shaders). Os animamos a estudiar el código y aprender a distinguir el código que se corresponde con los shaders del propio de WebGL o del Javascript.

10. Localiza de dónde proceden los nombres 'shader-fs' o 'shader-vs'. Dentro del código deben aparecer en dos sitios. En uno es el nombre (id) de un elemento del HTML y en el otro aparece como parámetro de un método Javascript.
11. Cambia el código del vexter shader para generar un error de compilación. Observa el error que produce. ¿Dónde se genera el error en WebGL o en Javascript? ¿Quién notifica el error? ¿Cómo sabes que se ha producido un error? Por ejemplo, cambia

    "gl_Position = vec4(aVertexPosition, 1.0);" por

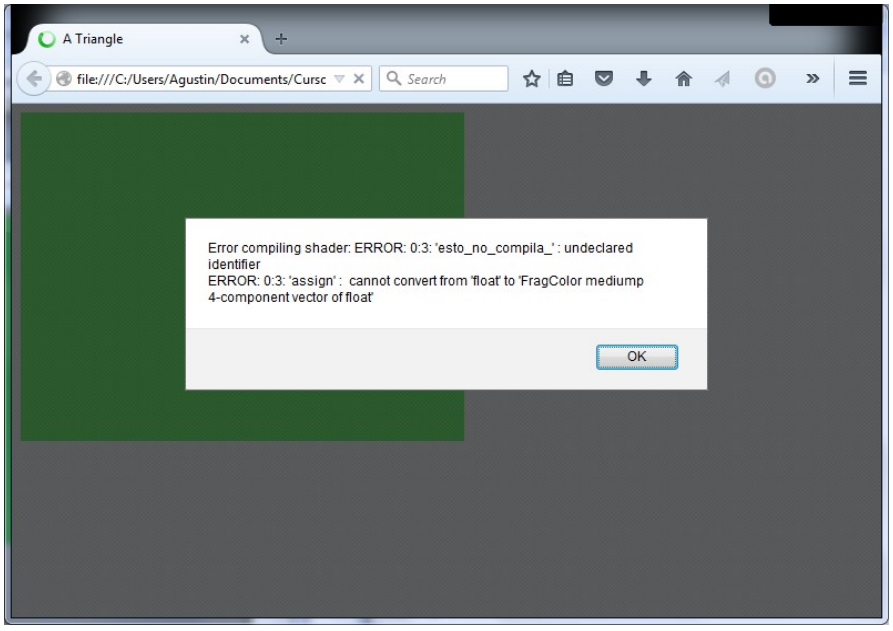
    "gl_Position = esto_no_compila_"



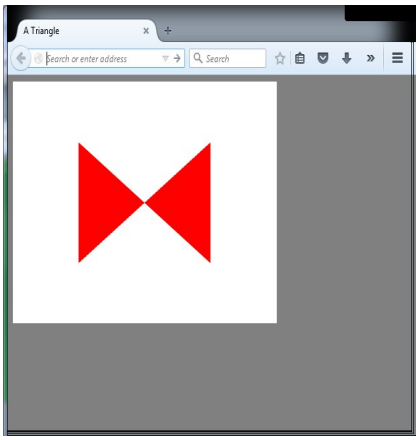
12. Cambia el código del fragment shader para generar un error de compilación. Observa el error que produce. ¿Dónde se genera el error en WebGL o en Javascript? ¿Quién notifica el error? ¿Cómo sabes que se ha producido un error? Por ejemplo, cambia

    "gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);" por

    "gl_FragColor = esto_no_compila_"



1.5 Cargamos los datos y dibujamos



Finalmente, en este ejemplo dibujamos un par de triángulos. Para ello cargamos unos datos en los buffer de WebGL, los conectamos con los Shaders y realizamos el dibujo.

Recuerda que estos pasos serán siempre necesarios: cargar los datos en WebGL, conexión de los buffers con los shaders (con las variables attributes) y finalmente la orden de dibujo. Estos pasos se pueden hacer una o varias veces. Normalmente, la carga de los datos solo querremos hacerla una sola vez. Los otros pasos seguramente la hagamos varias veces según las necesidades.

Código Relevante

Si observamos el código del ejemplo "***Example 1-5.html***" veremos que hemos completado las dos funciones que nos quedaban pendientes. Por un lado "setupBuffers" es la responsable de cargar los datos en WebGL. En ella se crean los buffers (estructuras de memoria internas a WebGL) y se cargan los datos (gl.bindBuffer y gl.bufferData). Por otro lado, en la función "drawScene " se conectan los datos con los atributos de los shader. Tu profesor debería explicarte qué es una variable de tipo "attribute". Finalmente se hace el dibujo mediante "gl.drawArrays" (el número 6 se corresponde con el número de vertices que hemos almacenado).

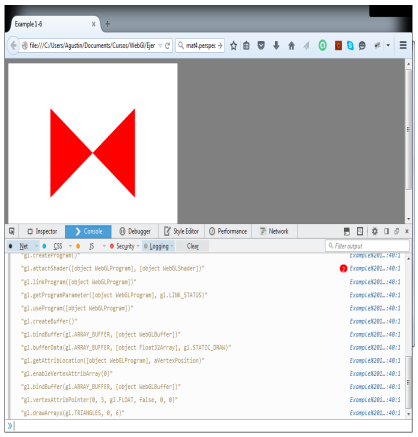
```
   function setupBuffers() {
            var triangleVertices = [
                        //left triangle
                        -0.5, 0.5, 0.0,
                        0.0, 0.0, 0.0,
                        -0.5, -0.5, 0.0,
                        //right triangle
                        0.5, 0.5, 0.0,
                        0.0, 0.0, 0.0,
                        0.5, -0.5, 0.0
            ];
            trianglesVerticeBuffer = gl.createBuffer();
            gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
            gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
gl.STATIC_DRAW);
        }
        function drawScene() {
            vertexPositionAttribute = gl.getAttribLocation(glProgram,
"aVertexPosition");
            gl.enableVertexAttribArray(vertexPositionAttribute);
            gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
            gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT, false, 0, 0);
            gl.drawArrays(gl.TRIANGLES, 0, 6);
        }
```

Ejercicios.

Primero os animamos a cambiar el modelo a dibujar. Para ello tendréis que retocar el código que guarda los datos en "triangleVertices ".

13. Cambia los triángulos de lado para que parezca un rombo.
14. ¿Sabrías dibujar un cuadrado?
15. Y ¿sabrías generar un círculo?

1.6 Depurando el código



Este ejemplo es visualmente idéntico al anterior. La principal diferencia es que vamos a introducir técnicas que nos ayuden a depurar nuestro código. Además de la depuración que nos ofrecen los propios navegadores, en este ejemplo os vamos a enseñar como depurar con una librería de Javascript específica para WebGL.

Esta librería nos permite activar trazas o detectar llamadas a WebGL incorrectas. Tenéis más información en:

https://www.khronos.org/webgl/wiki/Debugging

Código Relevante

Como hemos comentado en la introducción, el código del ejemplo "***Example 1-6.html***" es muy parecido al ejercicio anterior, pero hemos incluido una librería que nos permite realizar logs y trazas a las llamadas de WebGL. En la URL que os hemos aportado encontraréis mucho más detalle sobre el funcionamiento de esta magnifica librería.

```
   <script src="webgl-debug.js"></script>

       function throwOnGLError(err, funcName, args) {
          throw WebGLDebugUtils.glEnumToString(err) + " was caused by call to: " +
funcName;
        };

        function logGLCall(functionName, args) {
           console.log("gl." + functionName + "(" +
              WebGLDebugUtils.glFunctionArgsToString(functionName, args) + ")");
        }

        function validateNoneOfTheArgsAreUndefined(functionName, args) {
          for (var ii = 0; ii < args.length; ++ii) {
            if (args[ii] === undefined) {
              console.error("undefined passed to gl." + functionName + "(" +
                    WebGLDebugUtils.glFunctionArgsToString(functionName, args) + ")");
            }
          }
        }

        function logAndValidate(functionName, args) {
           logGLCall(functionName, args);
           validateNoneOfTheArgsAreUndefined (functionName, args);
        }

        function initWebGL() {
            canvas = document.getElementById("my-canvas");
            try {
                gl = canvas.getContext("webgl") ||
                canvas.getContext("experimental-webgl");
              gl = WebGLDebugUtils.makeDebugContext(gl, throwOnGLError,
logAndValidate);
            } catch (e) {
            }
            if (gl) {
                setupWebGL();
                initShaders();
                setupBuffers();
                drawScene();
            } else {
```

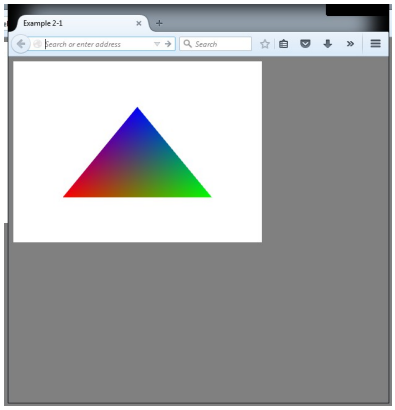
```
            alert("Error: Your browser does not appear to support WebGL.");
        }
    }
```

Ejercicios.

16. Aplica estas técnicas a los ejemplos anteriores y comprueba su correcto funcionamiento.
17. Introduce algún error y observa como la librería te ayuda a la depuración.

2 Buffers y “attributes”

2.1 Uso de Atributos y Buffer



Este ejemplo dibuja de un triángulo sencillo con colores diferentes para cada vértice.

Ahora cada vértice tiene especificado un color. Para poner colores a los vértices tenemos que jugar con los shaders y buffers. En este ejemplo definimos en el shader de vértices una nueva variable para recibir el color. También modificamos la construcción del buffer. ¿Puedes localizar esos cambios?

Observa cómo los cambios afectan a los elementos del código. Se modifican las definiciones de ambos Shaders, la construcción de los buffers y la sentencias que los conectan con los shaders.

Código Relevante

Si observamos el código del ejemplo “***Example 2-1.html***” veremos que los cambios son numerosos. Por una parte hemos tenido que tocar en los shaders, tanto en el de vértices como en el de fragmentos. Si recuerdas, el objetivo del fragment shader es determinar el color de cada pixel. Por lo tanto, ahora lo que hacemos es hacerle llegar un color mediante una variable llamada vColor. Esa variable está declarada como “varying” (es decir, es una variable generada en el vertex shader y traspasada al fragment shader). Si ahora miramos el vertex shader, veremos que esa variable toma el valor que procede de “aVertexColor”. Y esa variable a su vez, está declarada como “attribute”. Recordad que una variable “attribute” es una variable que toma valores desde un buffer.

Lo que significa esto, es que el color de cada vértice vendrá determinado por un valor que tendremos que almacenar en un buffer. El color de los pixels intermedios serán calculados mediante interpolación de los vértices.

```
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec3 aVertexColor;
    varying highp vec4 vColor;
    void main(void) {
         gl_Position = vec4(aVertexPosition, 1.0);
         vColor = vec4(aVertexColor, 1.0);
    }
</script>
<script id="shader-fs" type="x-shader/x-fragment">
    varying highp vec4 vColor;
    void main(void) {
        gl_FragColor = vColor;
    }
</script>
```

El siguiente cambio es la generación y uso de los buffer de color. Este código es una mera repetición de lo que ya teníamos antes.

```
        var triangleColors = [
                    // triangle Colors
                    1.0, 0.0, 0.0,
                    0.0, 1.0, 0.0,
```

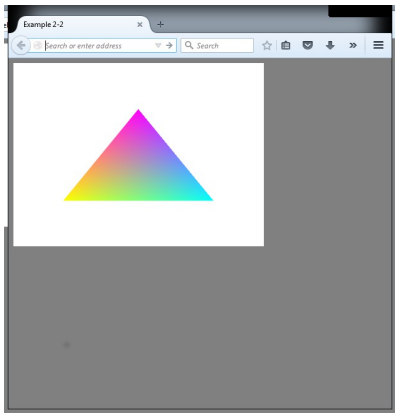
```
                        0.0, 0.0, 1.0,
            ];
            trianglesColorBuffer = gl.createBuffer();
            gl.bindBuffer(gl.ARRAY_BUFFER, trianglesColorBuffer);
            gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleColors),
gl.STATIC_DRAW);


            vertexColorAttribute = gl.getAttribLocation(glProgram, "aVertexColor");
            gl.enableVertexAttribArray(vertexColorAttribute);
            gl.bindBuffer(gl.ARRAY_BUFFER, trianglesColorBuffer);
            gl.vertexAttribPointer(vertexColorAttribute, 3, gl.FLOAT, false, 0, 0);
```

Ejercicios.

Para profundizar en el ejemplo, os proponemos los siguientes ejercicios.

18. Cambia los colores de los vértices para adaptarlos a tu gusto.
19. ¿Sabrías dibujar un cuadrado con colores? Simplemente añade un nuevo vértice, pero no te olvides de darle un color.
20. Dibuja un círculo de tamaño unidad centrado en el origen. Haz que el círculo tenga 30 vértices (separados por 12 grados entre ellos). Haz que cada vértice tenga un color y que el centro sea blanco.

2.2 Combinando los Buffers



De nuevo volvemos a dibujar el mismo triángulo, pero esta vez vamos a cambiar la forma de crear y conectar los buffers. En lugar de tener dos buffers con los datos de los vértices y de los colores, se va a crear un único buffer. En el mezclamos los datos de los vértices y los datos de los colores. Observad que el código de los shaders no cambia. Unicamente cambiamos el proceso de definición y uso de los buffers. Es interesante el cambio que se ha producido en las llamadas a glVertexAttribPointer. ¿Podéis determinar las razones?

Nota. Observad que hemos cambiado la paleta de colores para distinguirlo del ejemplo anterior.

Código Relevante

Si observamos el código del ejemplo "***Example 2-2.html***" veremos que los cambios son esta vez más limitados. El cambio solo afecta a la generación y uso de los buffers. Ahora mezclamos dentro del mismo buffer los datos del vértice y su color. Además, cambiamos los parámetros de las llamadas a " gl.vertexAttribPointer" para indicar cómo debe encontrar los datos dentro del buffer. El valor "6*4" se corresponde al tamaño en bytes que hay entre un vértice y otro. Esos números proceden del hecho de que un float ocupa 32 bits (4 bytes) y que hay 6 datos entre vértices (3 son de coordenadas y otros 3 son de colores). El otro valor "3*4" le indica la posición del color dentro del buffer (hemos consumido 3 floats para indicar la posición).

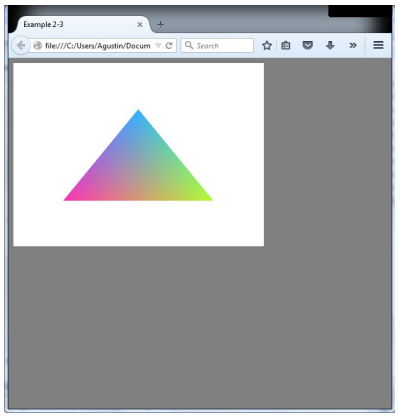
```
      function setupBuffers() {
            var triangleVerticesColor = [
                        //triangle Vertices & Color
                        -0.6, -0.5, 0.0,
                         1.0, 1.0, 0.0,

                         0.6, -0.5, 0.0,
                        0.0, 1.0, 1.0,

                         0.0,  0.5, 0.0,
                         1.0, 0.0, 1.0,
            ];
            trianglesVerticeColorBuffer = gl.createBuffer();
            gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeColorBuffer);
            gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVerticesColor),
gl.STATIC_DRAW);


        }
        function drawScene() {
            vertexPositionAttribute = gl.getAttribLocation(glProgram,
"aVertexPosition");
            gl.enableVertexAttribArray(vertexPositionAttribute);
            gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeColorBuffer);
            gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT,false, 6*4, 0);

            vertexColorAttribute = gl.getAttribLocation(glProgram, "aVertexColor");
            gl.enableVertexAttribArray(vertexColorAttribute);
            gl.vertexAttribPointer(vertexColorAttribute, 3, gl.FLOAT, false, 6*4, 3*4);

            gl.drawArrays(gl.TRIANGLES, 0, 3);
        }
```

Ejercicios.

Para profundizar en el ejemplo, os proponemos una repetición de los ejercicios anteriores.

21. Cambia los colores de los vértices para adaptarlos a tu gusto.
22. ¿Sabrías dibujar un cuadrado con colores? Simplemente añade un nuevo vértice, pero no te olvides de darle un color.

2.3 Profundizando en los Atributos



De nuevo volvemos a dibujar el mismo triángulo, pero de hemos vuelto a cambiar la forma de crear y conectar los buffers. Seguimos usando un único buffer con los datos de los vértices y los datos de los colores. Pero ahora vamos a cambiar el tamaño y naturaleza de los datos. Vamos a transmitir solo dos coordenadas para la posición.

Nota. Observad que hemos vuelto a cambiar la paleta de colores para distinguirlo del ejemplo anterior.

Código Relevante

Si observamos el código del ejemplo "***Example 2-3.html***" veremos que los cambios son mínimos. Por un lado hemos cambiado los vertex shaders para indicar que solo hay dos coordenadas por vértice y ponemos la coordenada "Z" igual a cero por defecto.

```
        attribute vec2 aVertexPosition;
        attribute vec3 aVertexColor;
        varying highp vec4 vColor;
        void main(void) {
             gl_Position = vec4(aVertexPosition, 0.0, 1.0);
            vColor = vec4(aVertexColor, 1.0);
        }
```

Por otro lado, los parámetros al usar los buffer han cambiado ligeramente reflejando ese cambio. En el array de datos ya no está la coordenada Z.

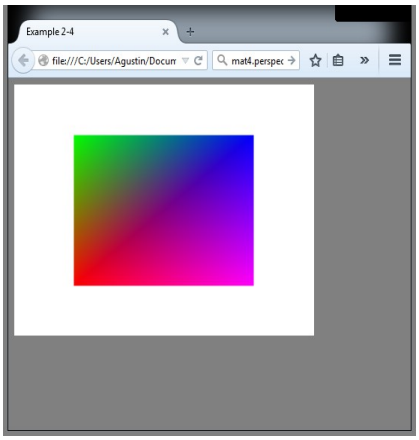
```
var triangleVerticesColor = [
                        //triangle Vertices & Color
                        -0.6, -0.5,
                         1.0, 0.2, 0.7,

                         0.6, -0.5,
                         0.7, 1.0, 0.2,

                         0.0,  0.5,
                        0.2, 0.7, 1.0,
            ];

            vertexPositionAttribute = gl.getAttribLocation(glProgram,
"aVertexPosition");
            gl.enableVertexAttribArray(vertexPositionAttribute);
            gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeColorBuffer);
            gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT, false, 5*4,
0);

            vertexColorAttribute = gl.getAttribLocation(glProgram, "aVertexColor");
            gl.enableVertexAttribArray(vertexColorAttribute);
            gl.vertexAttribPointer(vertexColorAttribute, 3, gl.FLOAT, false, 5*4, 2*4);
```

Ejercicios.

Para profundizar en el ejemplo, os proponemos una repetición de los ejercicios anteriores.

23. Cambia los colores de los vértices para adaptarlos a tu gusto.
24. ¿Sabrías dibujar un cuadrado con colores? Simplemente añade un nuevo vértice, pero no te olvides de darle un color.

2.4 Usando Índices



Este ejemplo dibuja un cuadrado, pero hemos cambiado la forma de procesar el buffer. Hasta ahora teníamos un buffer con el array de puntos que queríamos dibujar. Si el punto se dibujaba dos veces, lo almacenábamos dos veces.

Ahora lo que hacemos es almacenar por un lado el array de puntos (pero cada punto una única vez) y por otro, guardamos los índices de los vértices que vamos a dibujar. De esta forma, solo tenemos cuatro vértices pero seis índices (tres por cada triángulo que forma el cuadrado)

Código Relevante

Si observamos el código del ejemplo "***Example 2-4.html***" veremos que la construcción de un buffer de índices se parece mucho a lo que habíamos visto hasta el momento pero hay dos diferencias principales. Por un lado el tipo del buffer es ahora "gl.ELEMENT_ARRAY_BUFFER". Eso le indica a WebGL que se tratan de índices. Y por otro lado, al ser números enteros sin decimales, el buffer se crea como enteros de 16 bits (unsigned short).

Recordad que a la hora de dibujar debemos utilizar "drawElements" en lugar de "drawArrays".

```
            var modelElements = [
                        // Indices
                        0, 1, 2,
                        0, 2, 3,
            ];
            modelElementBuffer = gl.createBuffer();
            gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, modelElementBuffer);
            gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(modelElements),
gl.STATIC_DRAW);

. . .

            gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, modelElementBuffer);
            gl.drawElements(gl.TRIANGLES, 6, gl.UNSIGNED_SHORT, 0);
```

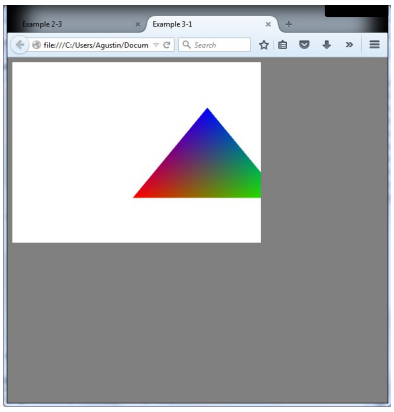
Ejercicios.

Os animamos a cambiar el modelo a dibujar siguiendo los ejercicios anteriores.

25. Cambia el modelo para que parezca un hexágono.
26. Y ¿Sabrías generar un círculo con esta técnica?

3 Variables “uniform”

3.1 Uso de variables Uniform



Este ejercicio introduce un nuevo tipo de variable en los shaders. Hasta ahora hemos utilizado “attribute” o “varying”. En este ejercicio se introduce el tipo “uniform” explicado en clase. Este tipo de variable toma un valor fijo (por eso su nombre) para todo el proceso de dibujo.

En este ejemplo se utiliza una variable uniform para mover el triangulo. El ejemplo utiliza una técnica de Javascript que nos permite animar el dibujo mediante refrescos continuos del dibujo.

Código Relevante

Este ejemplo se corresponde con el código “***Example 3-1.html***”. Veremos que hemos introducido un nueva variable en el vertex shader. Esta variable (llamada “posX”) es de tipo uniform. Se utiliza para calcular la posición del vértice sumando una cantidad a la coordenada “X”.

```
<script id="shader-vs" type="x-shader/x-vertex">
       attribute vec3 aVertexPosition;
       attribute vec3 aVertexColor;
       varying highp vec4 vColor;

        uniform float posX;
        void main(void) {
             gl_Position = vec4(aVertexPosition.x + posX, aVertexPosition.yz, 1.0);
            vColor = vec4(aVertexColor, 1.0);
        }
    </script>
```

El cambio que nos permite animar la figura aparece en la función principal.

```
            if (gl) {
                initShaders();
                setupBuffers();
                getUniforms();
                (function animLoop() {
                    setupWebGL();
                    drawScene();
                    requestAnimationFrame(animLoop, canvas);
                })();
            } else {
                alert("Error: Your browser does not appear to support WebGL.");
            }
```

Finalmente, el cambio más relevante es la conexión con las variables uniform. Recuerda preguntarle a tu profesor sobre este tipo de variables.

```
       function getUniforms() {
            glProgram.position = gl.getUniformLocation(glProgram, "posX");
        }

        var angle = 0.0;
```

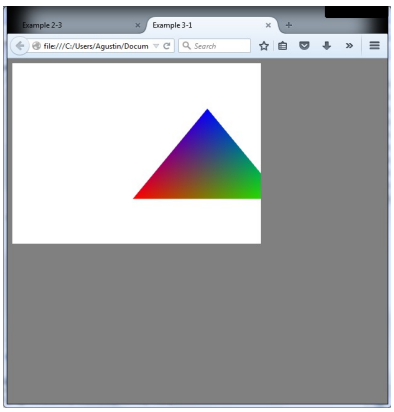
```
function setUniforms() {
    var newPos = 1.5*Math.sin(angle);
    angle += 0.01;

    gl.uniform1f(glProgram.position, newPos);
}
```

Ejercicios.

Este ejemplo introduce un concepto muy relevante y es importante que hagáis los siguientes ejercicios.

27. Cambia el vertex shader para que el cambio afecte al eje "Y". Es decir que en lugar de moverse de derecha a izquierda se mueva de arriba a abajo.
28. Cambia el vertex shader para que el cambio afecte al tamaño del triangulo. Es decir que en lugar de moverse se haga grande y pequeño.
29. Añade una nueva variable uniform en el vertex shader para que se mueva a la vez de derecha a izquierda y de arriba a abajo. Intenta que tenga un movimiento circular.

3.2 Uso de Uniform para el color



Este ejercicio es muy similar al anterior pero ahora jugamos con la variable uniform para determinar el color del triángulo.

El código es prácticamente el mismo. La diferencia radica en el uso que le damos a la variable uniform en el shader.

Con estos ejemplos podemos entender las posibilidades que nos ofrece el WebGL sobre el OpenGL antiguo. Hacer esto en el OpenGL antiguo hubiera sido posible, pero a costa de una pérdida en el rendimiento si el modelo fuera complejo.

Código Relevante

Este ejemplo se corresponde con el código "***Example 3-2.html***". Es muy similar al anterior, pero ahora usamos la variable uniform para alterar el color del triángulo. Notad que la escala de color altera TODOS los vértices del triángulo. El resto del código es muy similar a lo visto hasta el momento y no lo incluimos.

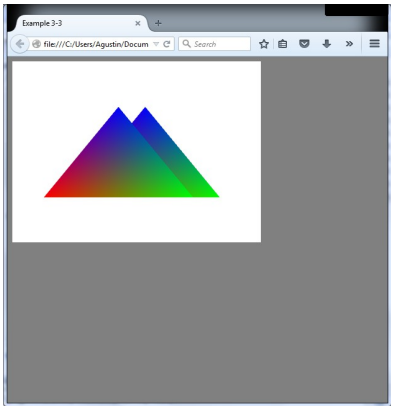
```
<script id="shader-vs" type="x-shader/x-vertex">
       attribute vec3 aVertexPosition;
       attribute vec3 aVertexColor;
       varying highp vec4 vColor;

        uniform float colorScale;
        void main(void) {
             gl_Position = vec4(aVertexPosition, 1.0);
            vColor = vec4(aVertexColor +vec3(colorScale, colorScale, colorScale), 1.0);
        }
</script>
```

Ejercicios.

Este ejemplo sigue trabajando con el concepto de uniform.

30. Cambia el vertex shader para que el cambio afecte únicamente a un canal el color (por ejemplo solo al rojo o al verde).
31. Cambia el vertex shader para que el cambio afecte más a un canal de color que a otro (por ejemplo, que el rojo se vea afectado el doble que el verde, y el azul permanece sin cambiar).
32. Añade una nueva variable uniform en el vertex shader. Con dos variables uniform podrás cambiar de forma independiente el canal rojo y el verde.

3.3 Nuevos usos de Uniform



Este ejercicio juega de nuevo con la misma idea pero ahora aprovechamos la variable uniform para mover dos triángulos de forma independiente.

Este ejemplo es interesante ya que reutilizamos el mismo código de shaders y el mismo modelo (los mismos buffers).

La única diferencia es que se activa un valor de la variable uniform antes de dibujar el triángulo. Jugando con ese parámetro conseguimos dibujar dos triángulos en movimiento.

Código Relevante

Este ejemplo se corresponde con el código "***Example 3-3.html***". Es muy similar a los anteriores, pero hacemos dos dibujos con valores de uniform diferentes. Observad que reutilizamos los mismos buffers para ambos dibujos.

```
            angle += 0.01;
            setUniforms(angle);
            gl.drawArrays(gl.TRIANGLES, 0, 3);

            setUniforms(angle*2);
            gl.drawArrays(gl.TRIANGLES, 0, 3);
```

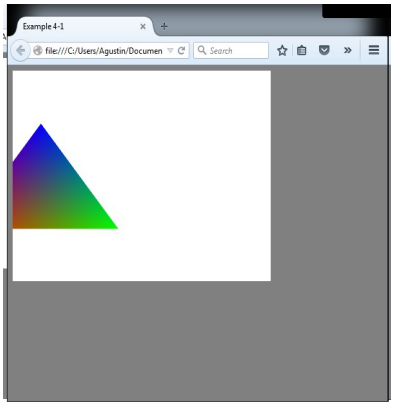
Ejercicios.

Este ejemplo sigue trabajando con el concepto de uniform.

33. Añade una nueva variable uniform en el vertex shader. Con dos variables uniform podrás mover de forma independiente los triángulos en los ejes "X" e "Y".
34. Sin cambiar el shader inicial, cambia la forma de poner la variable uniform para que los triángulos se acerquen y se alejen mutuamente.
35. Aplica una técnica similar a la utilizada en estos ejemplos para que los dos triángulos cambien de posición y de color a la vez pero de forma independiente.
36. Dibuja dos triángulos moviéndose en círculo. Un triangulo se mueve en un círculo exterior de radio 1. El otro triángulo se mueve en un círculo interior de radio ½. Escala los triángulos para que el dibujo sea razonable.
37. Lo mismo que el ejercicio anterior, pero que uno se mueva en el sentido horario y el otro en sentido contrario al reloj.

4 Usos de vectores y matrices

4.1 Uso de matrices



Este ejercicio es similar a los ejercicios anteriores, pero en lugar de mover el objeto utilizando un escalar (un float) vamos a utilizar una matriz. El uso de vectores y matrices en WebGL es fundamental ya que nos permite mover, escalar o rotar los objetos en el espacio.

En este primer ejemplo, empezamos introduciendo una matriz que nos permitirá mover el triángulo de una forma un poco más interesante en el futuro. Por ahora lo moveremos igual que antes, lateralmente. Comparad esto con los ejercicios anteriores :-)

Código Relevante

Este ejemplo se corresponde con el código "***Example 4-1.html***". Lo primero que tenemos que observar es que hemos incluido una librería de javascript llamada "gl-matrix-min.js". Nos servirá para crear y modificar matrices.

El siguiente punto más relevante es la creación de una variable llamada "uMVMatrix". Esta es de tipo uniform pero esta vez en lugar de ser un float, se trata de una matriz. Eso es lo que indica la palabra "mat4". Ahora usamos las propiedades de las matrices para cambiar la posición del triangulo en el eje "X".

```
<script src="gl-matrix-min.js"></script>
<script id="shader-vs" type="x-shader/x-vertex">
       attribute vec3 aVertexPosition;
       attribute vec3 aVertexColor;
       varying highp vec4 vColor;

        uniform mat4 uMVMatrix;
        void main(void) {
             gl_Position = uMVMatrix * vec4(aVertexPosition, 1.0);
            vColor = vec4(aVertexColor, 1.0);
        }
</script>
```

Finalmente, el cambio que nos permite utilizar las matrices aparece cuando tratamos con las variables uniform desde Javascript. Con "mat4.identity" reconvertimos la matriz a un estado inicial correspondiente a la matriz identidad. Y con "mat4.translate" hacemos que la matriz contenga la información necesaria para hacer una translación.

```
        function getUniforms() {
            glProgram.uMVMatrix = gl.getUniformLocation(glProgram, "uMVMatrix");
        }

        var angle = 0.0;

        function setUniforms() {
            var newPos = 1.5*Math.sin(angle);
            angle += 0.01;
            mat4.identity(mvMatrix);
            mat4.translate(mvMatrix, [newPos , 0.0, 0.0]);
            gl.uniformMatrix4fv(glProgram.uMVMatrix, false, mvMatrix);
```

```
}
```

Ejercicios.

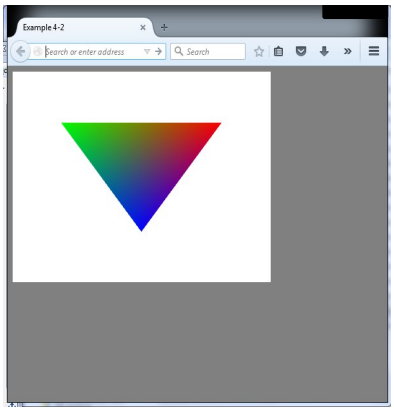
Este ejemplo introduce el concepto de matrices y podemos repetir los ejercicios que hicimos en el apartado 3-1 pero usando matrices.

38. Sin cambiar el vertex shader modifica la construcción de la matriz para que el cambio afecte al eje "Y". Es decir que en lugar de moverse de derecha a izquierda se mueva de arriba a abajo.
39. Sin cambiar el vertex shader modifica la construcción de la matriz para que se mueva a la vez de derecha a izquierda y de arriba a abajo.
40. Sin cambiar el vertex shader modifica la construcción de la matriz para que el triángulo se mueva haciendo círculos. Tienes la solución en el fichero " ***Example 4-1 Bis.html***"

4.2 Escalado

Este ejercicio es muy similar al anterior pero esta vez usamos la matriz para escalar el triángulo. Observa que no modificamos los shaders. Solo la forma de construir la matrix. De hecho, solo cambiamos una sola línea. Interesante, verdad!

Esa es la potencia de usar matrices.



Código Relevante

Este ejemplo se corresponde con el código "***Example 4-2.html***". Lo que vemos que haya cambiado es la línea que construye la matrix.

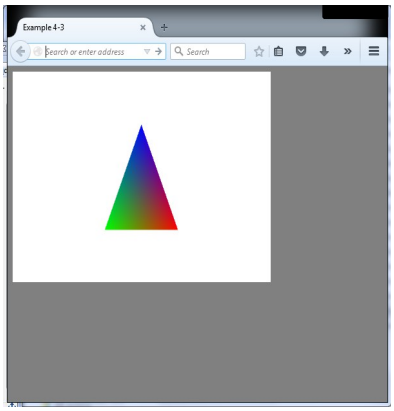
```
function setUniforms() {
     var scale = 1.5*Math.sin(angle);
     angle += 0.01;
     mat4.identity(mvMatrix);
     mat4.scale(mvMatrix, [scale, scale, 0.0]);
     gl.uniformMatrix4fv(glProgram.uMVMatrix, false, mvMatrix);
}
```

Ejercicios.

Este ejemplo sigue trabajando con el concepto de matrices. Seguimos intentando hacer ejercicios similares a los ya realizados.

41. Sin cambiar el vertex shader modifica la construcción de la matriz para que el escalado afecte solo al eje "Y".
42. Dibuja dos triángulos, uno que se escale en el eje "X" y el otro que se escale en el eje "Y".
43. Combina el ejercicio anterior para que además se muevan en círculos. Es decir combina un escalado con una traslación. Ten mucho cuidado con el orden de cálculo. No es lo mismo "trasladar y escalar" que "escalar y trasladar".

4.3 Rotación



Este ejercicio es muy similar al anterior pero esta vez usamos la matriz para rotar el triángulo. Al igual que nos sucedía antes, observa que no modificamos los shaders. Solo la forma de construir la matrix.

De nuevo vemos en juego la potencia de usar matrices.

Código Relevante

Este ejemplo se corresponde con el código "***Example 4-3.html***". Lo que vemos que haya cambiado es la línea que construye la matrix.

```
function setUniforms() {
     var scale = 1.5*Math.sin(angle);
     angle += 0.01;
     mat4.identity(mvMatrix);
     mat4.rotate(mvMatrix, angle, [0.0, 1.0, 0.0]);
     gl.uniformMatrix4fv(glProgram.uMVMatrix, false, mvMatrix);
}
```

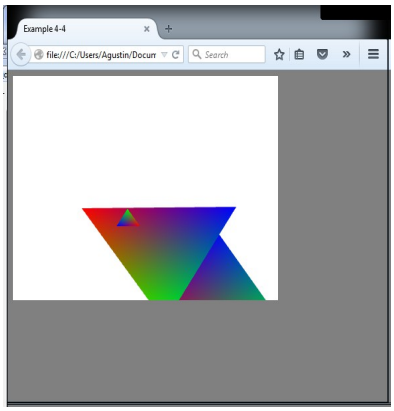
Ejercicios.

Este ejemplo sigue trabajando con el concepto de matrices. Seguimos intentando hacer ejercicios similares a los ya realizados.

44. Sin cambiar el vertex shader modifica la construcción de la matriz para que la rotación afecte solo al eje "X".
45. Sin cambiar el vertex shader modifica la construcción de la matriz para que la rotación afecte tanto al eje "X" como al eje "Y"
46. Dibuja dos triángulos, uno que rote en sentido horarioy el otro en el sentido contrario.

4.4 Combinando

En este ejemplo jugamos con varios triángulos que cambian de forma y posición independientemente. Pero a pesar de que el resultado parece complejo realmente estamos usando el mismo modelo y los mismos shaders para el dibujo. La única diferencia es que cada triangulo construye la matriz de modelo de una forma diferente.



Código Relevante

Este ejemplo se corresponde con el código "***Example 4-4.html***". La principal diferencia con los ejemplos anteriores es que ahora antes de dibujar el triángulo se prepara una matriz diferente.

```
        function drawTriangle1() {
            /* Render triangle */
            var posX = Math.cos(count);
            var posY = Math.sin(count);
            mat4.identity(mvMatrix);
            mat4.translate(mvMatrix, [posX, posY, 0.0]);
            gl.uniformMatrix4fv(glProgram.uMVMatrix, false, mvMatrix);
            gl.drawArrays(gl.TRIANGLES, 0, 3);
        }

        function drawTriangle2() {
            /* Render triangle */
            var posX = Math.cos(count)/2;
            var posY = Math.sin(count)/2;
            mat4.identity(mvMatrix);
            mat4.translate(mvMatrix, [posX, posY, 0.0]);
            mat4.rotate(mvMatrix, count, [0.0, 0.0, 1.0]);
            gl.uniformMatrix4fv(glProgram.uMVMatrix, false, mvMatrix);
            gl.drawArrays(gl.TRIANGLES, 0, 3);
        }

        function drawTriangle3() {
            /* Render triangle */
            var posX = Math.cos(count*2)/3;
            var posY = Math.sin(count*2)/3;
            mat4.identity(mvMatrix);
            mat4.translate(mvMatrix, [posX, posY, 0.0]);
            mat4.rotate(mvMatrix, count, [0.0, 0.0, 1.0]);
            mat4.scale(mvMatrix, [posX, posX, 1.0]);
            gl.uniformMatrix4fv(glProgram.uMVMatrix, false, mvMatrix);
            gl.drawArrays(gl.TRIANGLES, 0, 3);
        }
```

Ejercicios.

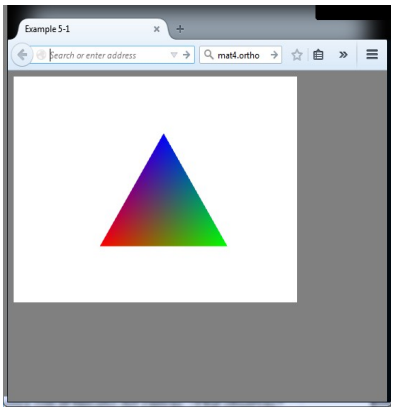
Este ejemplo trabajando con la idea de combinar matrices. Intenta hacer ahora los siguientes ejercicios.

47. Dibuja cuadrados pequeños que se muevan por pantalla con movimientos independientes.
48. Dibuja un cuadrado pequeño en el centro y haz que dos o tres cuadrados se muevan a su alrededor. Cada cuadrado debe tener un tamaño diferente (pero utiliza el mismo buffer para todos). La idea es simular un sistema planetario de tu invención.
49. (Avanzado). Las técnicas de "flocking" inventadas por Craig Reynolds en 1986, muy

interesantes e instructivas. Hay multitud de implementaciones realizadas en Java o Javascript. Y el material explicando el algoritmo es inmenso (https://en.wikipedia.org/wiki/Boids). Intenta hacer algo parecido usando los ejemplos que hemos realizado hasta la fecha con WebGL. Pero hazlo sencillo. Los "boids" serán triángulos. Un ejemplo de lo que podría lo que buscamos lo tienes en http://www.paulboxley.com/blog/2012/05/boids. Pero tu implementación debe usar únicamente lo que hemos visto hasta la fecha.

5 Matrices de Proyección

5.1 Ortogonal



Hasta ahora estamos representando triángulos planos, pero para dibujar en tres dimensiones debemos introducir una nueva matriz. Se llama la matriz de proyección. En nuestras prácticas vamos a utilizar dos tipos de proyección: ortográfica y en perspectiva. Es importante que revises el material teórico relacionado con esta matriz.

En esta primera práctica, vamos a representar un triángulos en perspectiva ortográfica. Visualmente apenas vamos a ver ninguna diferencia con lo que ya teníamos, pero en nuestro código hemos introducido una nueva matriz.

Código Relevante

Este ejemplo se corresponde con el código "***Example 5-1.html***". Observa que hemos definido una nueva variable llamada "uPMatrix". Al igual que nuestra anterior matriz, esta variable es de tipo uniform. También es importante destacar el orden de la multiplicación de las matrices. Observa que, si leemos el código de derecha a izquierda, primero se calculan las coordenadas del punto, posteriormente le aplicamos la transformada definida por la matriz ModelView y finalmente la de proyección.

En este ejemplo vemos una representación muy parecida a la que ya teníamos en los ejemplos anteriores. Eso ha sido así ya que siempre hemos tenido figuras planas y nos hemos movido por el plano "Z = 0".

```
<script id="shader-vs" type="x-shader/x-vertex">
        attribute vec3 aVertexPosition;
        attribute vec3 aVertexColor;
        varying highp vec4 vColor;

        uniform mat4 uMVMatrix;
        uniform mat4 uPMatrix;
        void main(void) {
            gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
            vColor = vec4(aVertexColor, 1.0);
        }
    </script>
```

Otro de los cambios que hemos realizado es la preparación y conexión de la matriz de proyección. Queremos llamar la atención sobre dos puntos de interés. Por una parte, la matriz de proyección se suele construir una sola vez. Normalmente en nuestros ejemplos no vamos a cambiar el tipo de proyección (en los juegos si suele ser habitual). Para evitar realizar operaciones innecesarias, solo se almacena esa variable una sola vez. En nuestro código hemos elegido la función "getUniforms".

Otro de los aspectos que queremos destacar es el uso de la función "mat4.ortho". Busca información sobre sus parámetros y pregunta a tu profesor sobre ellos si tienes dudas.

```
        function getUniforms() {
            glProgram.uMVMatrix = gl.getUniformLocation(glProgram, "uMVMatrix");

            glProgram.uPMatrix = gl.getUniformLocation(glProgram, "uPMatrix");
```

```
            var pMatrix = mat4.create();
            var ratio = canvas.width/canvas.height;
            mat4.ortho(-ratio, ratio, -1.0, 1.0, -10.0, 10.0, pMatrix);
            gl.uniformMatrix4fv(glProgram.uPMatrix, false, pMatrix);
        }
```

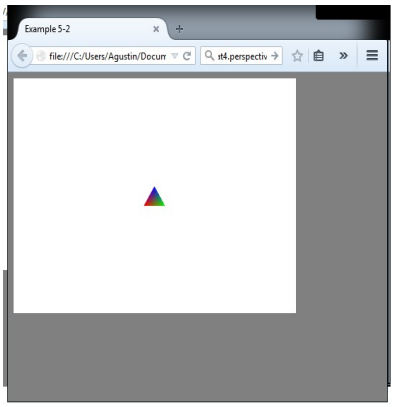
Ejercicios.

Para entender el concepto de proyección ortográfica, te recomendamos los siguientes ejercicios.

50. Nuestro triángulo se está moviendo en el eje "X". Intenta que se mueva en el eje "Z" haciendo que el valor de "Z" esté entre -10 y 10. ¿Notas algún cambio? ¿Tiene sentido lo que ves? ¿Debería hacerse más pequeño cuando se aleja y más grande cuando se acerca?
51. ¿Dónde tienes la cámara? ¿Cómo puedes saberlo?
52. Haz lo mismo que antes pero haz que el rango del valor de "Z" sea entre -12 y 12. ¿Has notado algo diferente? ¿Tiene sentido?
53. Cambia el tamaño del canvas. Pon algo que sea más alargado (por ejemplo, 800x200). Observa que el triángulo sigue teniendo la misma forma. Eso es debido a que en nuestro código se calcula un "ratio". Encuentra el punto en el que se realiza esa operación. Intenta cambiar el código por "var ratio = 1.0;" y juega con el tamaño del canvas. ¿Qué observas?

5.2 Perspectiva

El siguiente paso es cambiar nuestra proyección por una perspectiva. Ahora podemos ver que el triángulo realmente se hace grando o pequeño cuando se mueve en el eje Z. Según el triángulo se aleja de nosotros se ve más pequeño. Antes, con una matriz de tipo ortográfica eso no sucedía. ¿Sabrías explicar la razón de ese comportamiento?



Código Relevante

Este ejemplo se corresponde con el código "***Example 5-2.html***". Este código se parece bastante al anterior. De hecho comparte el mismo código de shaders. La principal diferencia es cómo construye la matriz de proyección. Te animamos a que investigues sobre los parámetros utilizados y pregunta a tu profesor si tienes alguna duda.

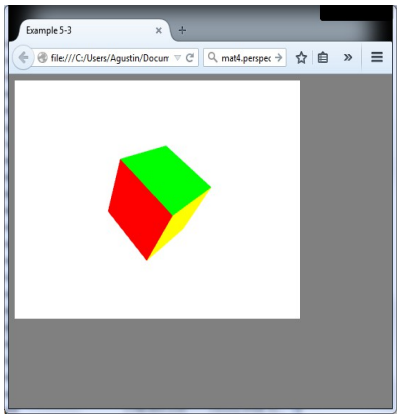
```
        function getUniforms() {
            glProgram.uMVMatrix = gl.getUniformLocation(glProgram, "uMVMatrix");

            glProgram.uPMatrix = gl.getUniformLocation(glProgram, "uPMatrix");
            var pMatrix = mat4.create();
            var ratio = canvas.width/canvas.height;
            mat4.perspective(60, ratio, 0.1, 100, pMatrix);
            gl.uniformMatrix4fv(glProgram.uPMatrix, false, pMatrix);
        }
```

Ejercicios.

Para entender el concepto de proyección ortográfica, te recomendamos los siguientes ejercicios.

54. Nuestro triángulo ya se está moviendo en el eje "Z", pero ¿Podrías decir el rango de valores que está tomando? ¿Tiene sentido que tenga un rango negativo?
55. ¿Dónde tienes la cámara? ¿Cómo puedes saberlo?
56. Intenta que el triángulo se mueva en el eje "Z" entre -6 y 6. ¿Has notado algo diferente? ¿Tiene sentido?
57. Dibuja varios triángulos moviéndose por los ejes "X" e "Y" pero sin moverse de la coordenada "Z". Haz que cada triángulo tenga una coordenada "Z" diferente. Tendrás que ver varios triángulos de diferentes tamaños moviéndose por la pantalla.
58. Cambia los movimientos del ejercicio anterior para que también se muevan en profundidad.
59. Cambia el tamaño del canvas. Pon algo que sea más alargado (por ejemplo, 800x200). Observa que el triángulo sigue teniendo la misma forma. Eso es debido a que en nuestro código se calcula un "ratio". Encuentra el punto en el que se realiza esa operación. Intenta cambiar el código por "var ratio = 1.0;" y juega con el tamaño del canvas. ¿Qué observas?

5.3 Dibujando un Cubo



El siguiente paso es introducir un modelo algo más complejo. Vamos a representar un cubo utilizando el código del ejemplo anterior. En este ejemplo solo cambiamos el modelo. El resto sigue siendo muy similar.

En este ejemplo hemos guardado las coordenadas del cubo en un array y hemos utilizado índices (Elements). Debido al número de puntos, es más cómodo guardarlo como indica el ejemplo. Con los colores sucede algo similar.

Notad también que hemos activado el test de profundidad mediante "gl.Enable(gl.DEPTH_TEST);".

Código Relevante

Este ejemplo se corresponde con el código "***Example 5-3.html***". Este código tiene cambios significativos con respecto a los ejemplos anteriores, aunque casi todo lo utilizado ya deberías conocerlo. Hemos utilizado índices con la misma técnica que utilizamos en el ejemplo 2-4. La forma de dibujar es similar (usando drawElements). También hemos empaquetado todos los datos del cubo en un solo buffer mezclando coordenadas y colores. Vimos algo parecido en los ejercicios del apartado 2. Quizás el único cambio llamativo es la utilización de un control de profundidad para distinguir las partes vistas de las ocultas. Ahora cuando hacemos un clear, también le indicamos a WebGL que debe borrar el buffer de profundidad.

```
        function setupWebGL() {
            gl.enable(gl.DEPTH_TEST);
            gl.depthFunc(gl.LEQUAL);

            //set the clear color to a shade of green
            gl.clearColor(1.0, 1.0, 1.0, 1.0);

            gl.viewport(0, 0, canvas.width, canvas.height);
        }
. . .
        function drawScene() {
            gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

            gl.bindBuffer(gl.ARRAY_BUFFER, cubeAttrBuffer);
            gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT, false,
4*(3+3), 0);
            gl.vertexAttribPointer(vertexColorAttribute, 3, gl.FLOAT, false, 4*(3+3),
3*4);

            setUniforms();
            gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeElemBuffer);
            gl.drawElements(gl.TRIANGLES, 6*2*3, gl.UNSIGNED_SHORT, 0);
        }
```

Ejercicios.

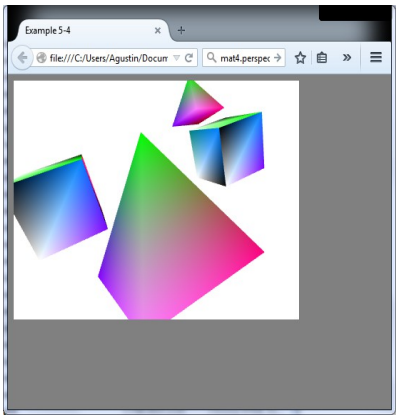
Para entender estos conceptos, os recomendamos los siguientes ejercicios.

60. Dibuja varios cubos rotando en diferentes posiciones.
61. Simula un sistema solar representando los planetas como cubos (reutiliza el modelo que hemos construido en el ejemplo). Has un cubo más grande en el centro y varios cubos más pequeños orbitando alrededor de éste.

5.4 Dibujando figuras

El último ejercicio de esta serie consiste en dibujar varias figuras en movimiento. De nuevo, no tendremos que modificar los shaders. Solo tocaremos la construcción de las matrices y la forma de dibujar los buffers.

Hasta el momento no hemos introducido iluminación por lo que jugamos con los colores de las caras para tener sensación de profundidad.



Código Relevante

Este ejemplo se corresponde con el código "***Example 5-4.html***". En este código no existen conceptos nuevos. Simplemente lo hemos juntado todo para que se genere algo más complejo. Te animamos a que estudies el código y a preguntar a tu profesor si tienes alguna duda.

Ejercicios.

Para entender estos conceptos, os recomendamos los siguientes ejercicios.

62. Dibuja alguna otra figura sencilla pero que sea diferente a las utilizadas.
63. Dibuja una o varias esferas.

6 Resumen

6.1 Conceptos

En esta serie de ejemplos hemos visto una serie de conceptos introductorios a OpenGL/WebGL. Antes de seguir en tus estudios repasa la teoría y las prácticas para asegúrate de que has entendido correctamente los siguientes conceptos:

- Historia de OpenGL y diferencias entre **OpenGL y WebGL**
- Separación entre **HTML, Javascript y OpenGL Shading Language**
- Proceso de **depuración de Javascript y de WebGL**
- **Vertex Shader y Fragment Shader**. Concepto de **Pipeline**
- Proceso de **compilación y uso** de los shaders. Concepto de **Program**
- Variables de tipo **Atributte, Uniform y Varying**: concepto, uso y diferencias
- **Buffer de Array** (usado para pasar datos a los attributos)
- Buffer de Indices (**Elements Array**, usado para indicar los índices del dibujo)
- Atributos habituales en un modelo: **coordenadas y color**.
- **Framebuffer** (zona de memoria donde dibuja OpenGL)
- Importancia y uso de **Vectores y Matrices**
- Creación y uso de matrices para transformaciones (**escalado, traslación, rotación, etc.**)
- Coordenadas homogéneas.
- **Proyección Ortogonal**: uso y características
- **Proyección en perspectiva**: uso y características
- Creación y uso de **matrices para definir la proyección**.

6.2 Métodos de WebGL utilizados

En esta serie de ejemplos hemos utilizado un conjunto de métodos o funciones propios de WebGL. Te recordamos los más importantes. Observa que no hemos incluido los correspondientes al tratamiento de matrices ya que éstos son propios de la librería mat4. En cualquier caso, asegúrate de que conoces el funcionamiento de todas la funciones utilizadas (tanto de WebGL como de librerías de apoyo) y que entiendes sus parámetros.

- gl.clearColor y gl.clear (trabajan con el framebuffer)
- gl.createShader, gl.shaderSource, gl.compileShader (creación y compilación de shaders)
- gl.createProgram, gl.attachShader, gl.linkProgram, gl.useProgram (concepto de Program)
- gl.createBuffer, gl.bindBuffer, gl.bufferData (creación de buffer de arrays o elements)
- gl.getAttribLocation, gl.enableVertexAttribArray, gl.vertexAttribPointer (activación y

conexión de atributos)

- gl.getUniformLocation, gl.uniform1f, gl.uniformMatrix4fv, (activación y conexión de variables uniform)
- gl.drawArrays, gl.drawElements (Orden de dibujo, normal o mediante índices)
- gl.enable, gl.depthFunc (activación de propiedades de WebGL, usado para la profundidad)