Bien, ahora que has regresado, echaremos un vistazo a las funciones de apoyo que facilitaron un poco la vida a la hora de usar webGL. Como dije antes, probablemente seas muy feliz ignorando ciertos detalles, y sólamente copiando y pegando en tu página las funciones de apoyo que vienen por encima de **initBuffers**, al principio del código javascript. Sin embargo no son muy difíciles de entender, y hacerlo significa poder escribir mejor código webGL en el futuro.

**initGL()**

Empezemos con la más aburrida de las funciones, la de la primera llamada que hace **webGLStart**, **initGL**. Es la función que está más arriba del todo en el código javascript:

var gl;  
function initGL(canvas) {

    try {  
        gl = canvas.getContext("experimental-webgl");  
        gl.viewportWidth = canvas.width;  
        gl.viewportHeight = canvas.height;  
        } catch(e) {  
    }  
    if (!gl) {  
        alert("No puede iniciarse webGL en este navegador");  
    }  
}

Es muy simple. Como ya hemos visto, las funciones **initBuffers** y **drawScene** usaban constántemente un objeto llamado gl, que claramente suponíamos que apuntaba a alguna cosa del nucleo de webGL. Esta función obtiene “esa cosa”, que llamaremos contexto webGL, que se obtiene pidiéndole al CANVAS un determinaado tipo de contexto. Y sí, como espero que hayas supuesto, el nombre “experimental-webgl” cambiará a “webgl” cuando se cierren los estándares de html5 y javascript. Una vez obtenido el contexto, usamos algunos métodos de javascript para obtener el ancho y el alto del lienzo, y haciendo uso del débil tipado de javascript, que como dijimos despierta tantas pasiones como odios, creamos dos nuevos atributos a gl para guardar esa valiosa información. Luego la usaremos para definir el tamaño de la zona donde pintar en el canvas, en **viewport** al comienzo de la función **drawScene**. Con ésto, ya tenemos configurado nuestro contexto webGL.

Después de llamar a **initGL**, **webGLStart** llama a **initShaders**. Ésta, por supuesto, inicializa los shaders (quién lo iba a decir). Pero dejemos todavía el misterio en el aire, porque antes tenemos que mirar a nuestra matriz de modelo-vista y a la matriz de proyección que mencionamos no hace mucho.

var mvMatrix = mat4.create();  
var pMatrix = mat4.create();

Definimos una variable llamada mvMatrix, que contendrá la matriz modelo-vista, y otra variable pMatrix, que será nuestra matriz de proyección. Inicialmente están llenas de ceros. Hablemos un poco de la matriz de proyección. Como espero que te acuerdes, se aplicó la función **mat4.perspective** a esta variable para establecer nuestro punto de vista, al principio de **drawScene**. Esto se debió a que webGL no admite directamente la gestión de la perspectiva, al igual que tampoco soportaba directamente la matriz modelo-vista. Y si necesitábamos una matriz donde guardar los resultados de las translaciones y rotaciones, también necesitamos otra matriz que maneje el proceso de hacer más grandes o pequeñas los objetos del espacio según la perspectiva. Y como habrás adivinado, eso es exactamente lo que hace la matriz de proyección. La función mat4.perspective rellena la matriz adecuadamente según la relación de aspecto y campo de visión indicados en los parámetros que le hemos puesto a la perspectiva.

De acuerdo, hasta el momento lo hemos visto todo excepto la función **setMatrixUniforms**, que, como dije en su momento, mueve las matrices de modelo-vista y proyección de javascript a webGL, y las terroríficas funciones de uso de shaders. Ambas están relacionadas, asi que empezaremos por explicar algunas nociones básicas.

**initShaders()**

**¿Qué es un shader?** Bueno, en algún momento de la historia de la informática gráfica en 3D, alguien necesitó de funciones que calculasen el color y las sombras de los objetos de la escena, antes de empezar a dibujarlos. Con el tiempo, los shaders han ido creciendo en utilidades, y actualmente podríamos decir que son una porción de código que puede modificar todos y cada uno de los bits finales de una escena, antes de dibujarla. Y ésto es realmente útil, porque además de que se ejecuta en la tarjeta gráfica (por lo que funciona muy rápido), el tipo de transformaciones que puede realizar son verdaderamente útiles, incluso en un ejemplo tan pequeño e insignificante como el que tenemos entre manos.

La razón por la que vamos a utilizar shaders en lo que pretende ser un sencillo ejemplo de introducción a webGL, es que se encargarán de que el sistema webGL aplique nuestras matrices modelo-vista y de proyección a nuestra escena con la rapidísima GPU y la memoria de la tarjeta gráfica, evitando que sea la CPU la que tenga que estar calculando con un (en comparación) lento javascript las posiciones finales de cada uno de los vértices de los objetos de la escena. Ésto es increíblemente útil.

Por lo tanto, aquí tenemos la forma de utilizarlo, paso a paso. Recordemos de que la llamada a la función **initShaders** está dentro de **webGLStart**:

var shaderProgram;  
function initShaders() {  
    var fragmentShader = getShader(gl, "shader-fs");  
    var vertexShader = getShader(gl, "shader-vs");  
  
    shaderProgram = gl.createProgram();  
    gl.attachShader(shaderProgram, vertexShader);  
    gl.attachShader(shaderProgram, fragmentShader);  
    gl.linkProgram(shaderProgram);  
  
    if (!gl.getProgramParameter(shaderProgram, gl.LINK\_STATUS)) {  
        alert("No pueden iniciarse los shaders");  
    }  
  
    gl.useProgram(shaderProgram);

Como puede verse, se usa una función llamada getShader para obtener dos objetos, algo llamado “fragmento shader”, **fragmentShader**, y otro llamado “vértice shader”, **vertexShader**, y luego los une para obtener algo llamado “programa shader”, **shaderProgram**. Un “programa” es un trozo de código que reside en el lado webGL del sistema; puedes verlo como una forma de especificar que algo debe funcionar en la tarjeta gráfica. Cada programa debe enlazar con un vértice shader y un fragmento shader. Veremos más de ellos en un momento.

    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "aVertexPosition");  
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

Una vez que hemos configurado el programa shader, y unido a él los shaders, ahora necesita que le asociemos una referencia a uno atributo, llamado **vertexPositionAttribute**, aprovechándonos, como ya lo hemos dicho dos veces, de que javascript permite inventarnos nuevos atributos en cualquier momento. Lo hacemos así porque es conveniente tener esos dos valores en el mismo objeto (programa shader) para usarlos más fácilmente.

Pero… ¿Qué es un **vertexPositionAttribute**? A lo mejor no recuerdas de que lo usamos en la función **drawScene**(éste es un buen momento para buscarla y repasarla). Se utilizaba en el código que establecía los vértices del triángulo al buffer adecuado. Hablaremos un poco más en un momento, por ahora basta con notar que también usamos **gl.enableVertexAttribArray** para decirle a webGL que queremos proporcionarle valores al atributo usando una lista.

    shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");  
    shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix");  
}

Lo último que hacemos en initShaders es recuperar dos valores más del programa shader: Las localizaciones de dos cosas llamadas matrices uniformes. De momento sólo hay que saber que las guardamos en el objeto programa por comodidad, aprovechándonos de que en javascript se pueden crear variables… Lo de siempre.

**getShader()**

Ahora veamos la función **getShader** al completo:

function getShader(gl, id) {  
    var shaderScript = document.getElementById(id);  
    if (!shaderScript) {  
        return null;  
    }  
  
    var str = "";  
    var k = shaderScript.firstChild;  
    while (k) {  
        if (k.nodeType == 3)  
            str += k.textContent;  
        k = k.nextSibling;  
    }  
  
    var shader;  
    if (shaderScript.type == "x-shader/x-fragment") {  
        shader = gl.createShader(gl.FRAGMENT\_SHADER);  
    } else if (shaderScript.type == "x-shader/x-vertex") {  
        shader = gl.createShader(gl.VERTEX\_SHADER);  
    } else {  
        return null;  
    }  
  
    gl.shaderSource(shader, str);  
    gl.compileShader(shader);  
  
    if (!gl.getShaderParameter(shader, gl.COMPILE\_STATUS)) {  
        alert(gl.getShaderInfoLog(shader));  
        return null;  
    }  
  
    return shader;  
}

Ésta es otra de esas funciones que parece más complicada de lo que en realidad es. Lo único que hacemos aquí es buscar en el árbol DOM de nuestra página HTML algún elemento que tenga la ID especificada como parámetro de la función, extraer todo su contenido (que será la creación de un fragmento o el sombreado de los vértices, hablaremos de ello en otra lección), y pasarle el código a webGL para que lo compile y pueda ejecutarse en la tarjeta gráfica. Después, el código controla cualquier posible y error, y acaba. Como alternativa, podríamos crear el código como un string (cadena de caracteres), en vez de utilizar un elemento HTML y perder el tiempo buscándolo y tomando su contenido, pero de esta forma se hace mucho más fácil de leer y modificar, ya que aparentan ser trozos de código javascript, pero ojo, no lo son.

**GLSL**

Aquí los tenemos:

<script id="shader-fs" type="“x-shader/x-fragment">  
    // <![CDATA[  
   #ifdef GL\_ES  
   precision highp float;  
   #endif  
   void main(void) {  
       gl\_FragColor = vec4(1.0, 1.0, 1.0, 1.0);  
   }  
   // ]]>  
</script>  
  
<script id="shader-vs" type="x-shader/x-vertex">  
    // <![CDATA[  
   attribute vec3 aVertexPosition;  
   uniform mat4 uMVMatrix;  
   uniform mat4 uPMatrix;  
   void main(void) {  
       gl\_Position = uPMatrix \* uMVMatrix \* vec4(aVertexPosition, 1.0);  
   }  
   // ]]>  
</script>

Lo primero que hay que destacar es que **NO** están escritos en lenguaje javascript, a pesar de que pueda resultarnos familiar. En realidad, están escritos en un lenguaje llamado [GLSL](http://es.wikipedia.org/wiki/GLSL), claramente inspirado en C. El primero de ellos, que corresponde al  fragment shader, comunica a la tarjeta gráfica de que use el tipo float como grado de precisión para todas las operaciones, e indica que todo los píxeles que se dibujen, lo hagan en color blanco. Cómo pintar con varios colores será motivo para una nueva lección.

El segundo shader es un poquito más interesante. Se trata de un vertex shader (codigo que se ejecuta en la tarjetra gráfica que puede hacer cualquier cosa con los vértices). Asociados a ella, tiene dos variables uniformes, **uMVMatrix** y **uPMatrix**. Las variables uniformes son útiles porque se pueden acceder a ellas desde fuera del shader, y como quizás hayas recordado, se extrajo su ubicación al final de **initShaders**, y será con ellas la forma de pasarle a webGL los valores de nuestras matrices de modelo-vista y proyección que calculamos en javascript.

El vertex shader es llamado una vez por cada vértice, y el vértice se pasa al código del shader a través de **aVertexPosition**, gracias al uso del **vertexPositionAttribute** en el **drawScene**. ¿Comienzas a ver la relación? Los shaders deben tener una rutina principal llamada **main**, y ésta en concreto sólo multiplica la posición del vértice por la matriz de modelo-vista y por la de proyección devolviéndolos la posición final del vértice en la escena.

Por lo tanto, webGLStart llama a **initShaders**, que usará **getShader** para cargar los shaders de fragment y vertex contenidos en dos <script> (con ID propio) de la página web, de modo que puedan ser compilados por la tarjeta gráfica y utilizados más adelante cuando se renderice la escena 3D.

Y finalmente, lo último que nos queda por comentar es **setMatrixUniforms**, el cual es muy fácil de entender si mas o menos se ha comprendido la lección hasta este punto.

function setMatrixUniforms() {  
    gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);  
    gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);  
}

Simplemente actualizamos las matrices uniformes de modelo-vista y proyección que están en la gráfica con las matrices de modelo-vista y proyección que tenemos en la memoria de javascript.