

Primeros pasos con WebGL

Katia Leal Algara

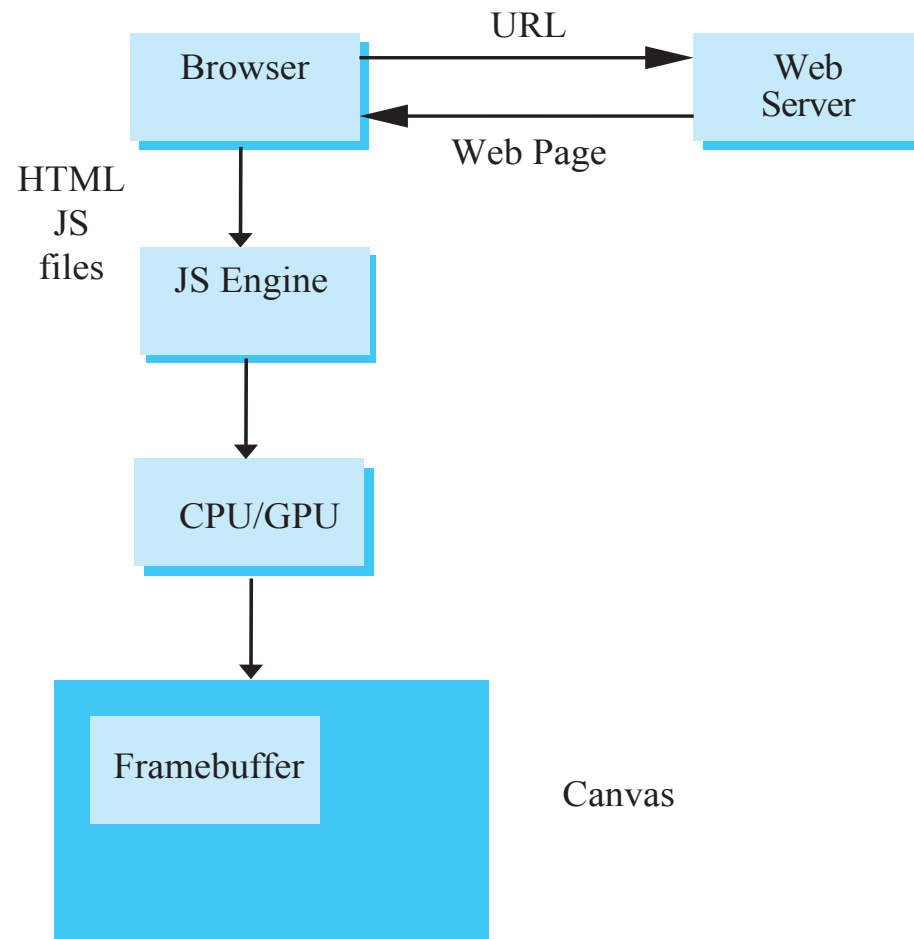
Web: <http://gsyc.urjc.es/~katia/>

Email: katia.leal@urjc.es

Dept. Teoría de la Señal y Comunicaciones y Sistemas Telemáticos y Computación (GSyC)
Escuela Superior De Ingeniería De Telecomunicación (ETSIT)
Universidad Rey Juan Carlos (URJC)



Ejecución en un navegador



Pasos

- Create a ***Canvas***
- Create a WebGL ***Context***
- Define and compile ***Shaders***
- ***Load data*** into WebGL
- Draw ***primitives***

CANVAS

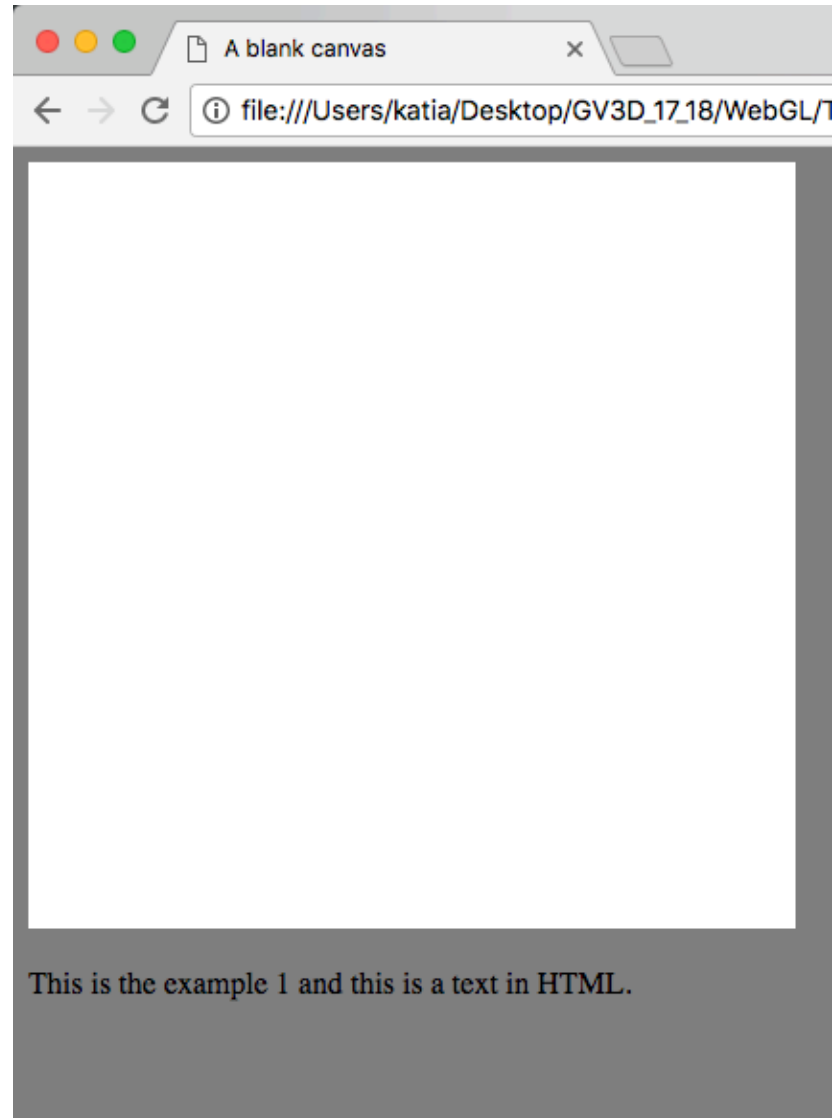
Canvas

- WebGL utiliza el elemento `<canvas>` de HTML5, el cual define un *área de dibujo* en la página web
 - Sin WebGL, este elemento sólo permite dibujar gráficos 2D
 - Con WebGL, el mismo elemento permite dibujar gráficos 3D
- **NOTA:** En Chrome se puede mostrar la consola en Ver→Opciones para desarrolladores→Consola Javascript

Example 1-1: Crear un Canvas

```
<!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>
</body>
</html>
```

Crear Canvas



Example 1-1: 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 negro.
3. Cambia el tamaño del lienzo (ahora está a 400x400). Pon algo que te resulte cómodo. Por ejemplo 600x600.
4. ¿Sabrías crear dos canvas dentro de la misma página web?

CONTEXT

Create a Context

El código tiene dos puntos de interés:

- Uno de ellos es la llamada a la creación del contexto, `canvas.getContext`, dentro de la función `setupWebGL`.
- El otro punto es el **borrado de la pantalla**:
 - Este estado de WebGL se irá complicando en los ejercicios posteriores, pero en este primer ejercicio únicamente se fija el color de borrado.
 - El método en OpenGL es `glClear`, en JavaScript lo veremos como `gl.Clear`.

Example 1-2: Create a Context

```
</style>
<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 to red
            gl.clearColor(1.0, 0.0, 0.0, 1.0);
            gl.clear(gl.COLOR_BUFFER_BIT);
        }else{
            alert( "Error: Your browser does not appear to support WebGL.");
        }
    }
</script>
</head>
```

Create a Context: Sintaxis

canvas.getContext(contextType, contextAttributes);

Parámetros

- *contextType*: contiene el identificador del contexto que define el contexto de dibujo asociado al lienzo. Los posibles valores son:
 - "2d", dando lugar a la creación de un objeto `CanvasRenderingContext2D` que representa un contexto de renderizado de dos dimensiones.
 - "webgl" (o "experimental-webgl") el cual creará un objeto `WebGLRenderingContext` que representa un contexto de renderizado de tres dimensiones. Este contexto sólo está disponible en navegadores que implementan WebGL version 1 (OpenGL ES 2.0).

Create a Context: Sintaxis

```
gl.clearColor(red, green, blue, alpha)
```

Specify the clear color for a drawing area:

Parameters	red	Specifies the red value (from 0.0 to 1.0).
	green	Specifies the green value (from 0.0 to 1.0).
	blue	Specifies the blue value (from 0.0 to 1.0).
	alpha	Specifies an alpha (transparency) value (from 0.0 to 1.0).
0.0 means transparent and 1.0 means opaque.		
If any of the values of these parameters is less than 0.0 or more than 1.0, it is truncated into 0.0 or 1.0, respectively.		
Return value	None	
Errors²	None	

Create a Context: Sintaxis

`gl.clear(buffer)`

Clear the specified buffer to preset values. In the case of a color buffer, the value (color) specified by `gl.clearColor()` is used.

Parameters `buffer` Specifies the buffer to be cleared. Bitwise OR (`|`) operators are used to specify multiple buffers.

`gl.COLOR_BUFFER_BIT` Specifies the color buffer.

`gl.DEPTH_BUFFER_BIT` Specifies the depth buffer.

`gl.STENCIL_BUFFER_BIT` Specifies the stencil buffer.

Return value None

Errors `INVALID_VALUE` *buffer* is none of the preceding three values.

`clear` recae en OpenGL, el cual utiliza múltiples buffers subyacentes

Create a Context



Example 1-2: Ejercicios Context

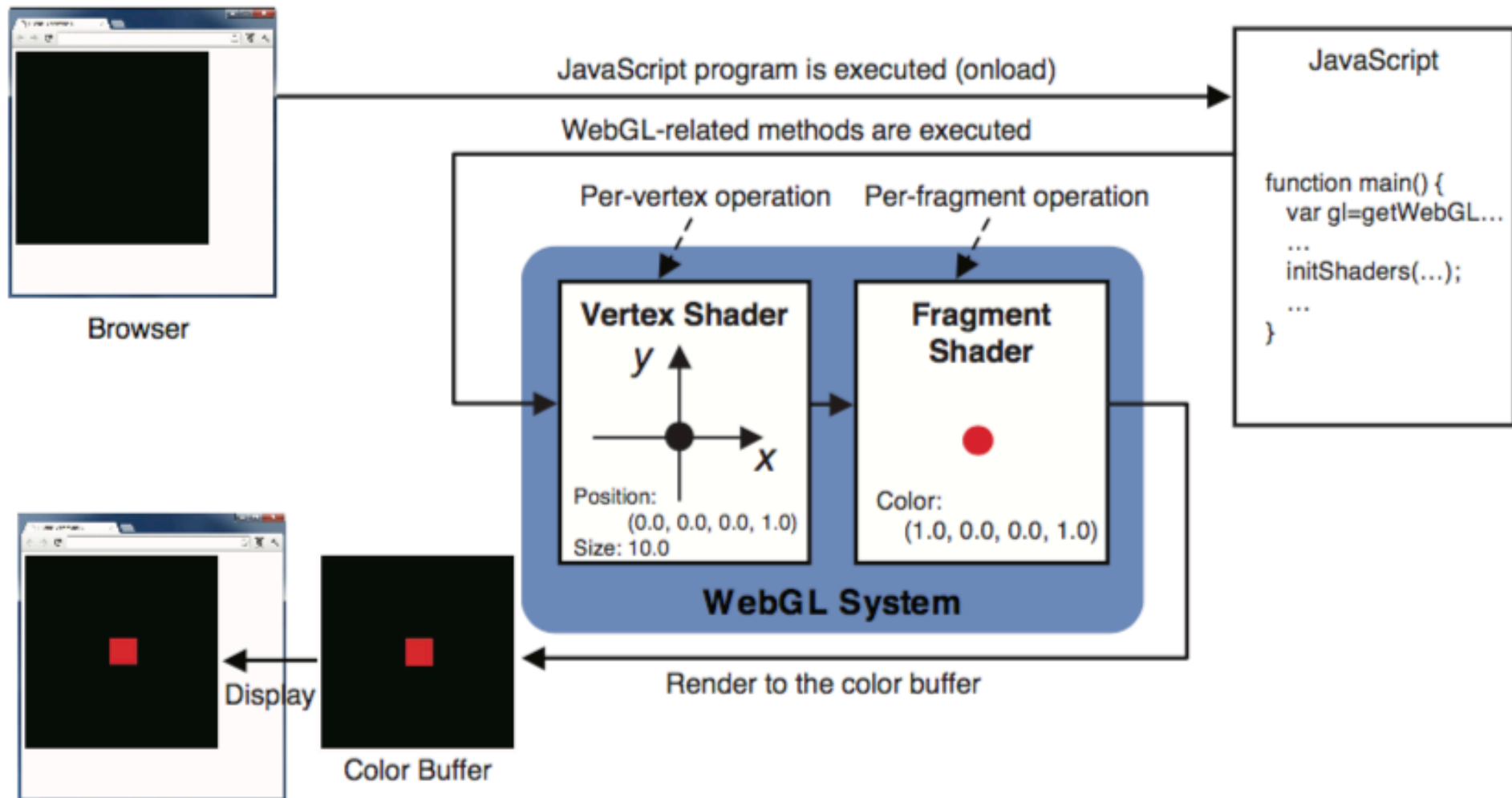
- Os animamos a estudiar el código y aprender a distinguir el código que se corresponde con WebGL con el que es propio de Javascript.
 1. ¿Sabrías cambiar el nombre del canvas?
Tienes que modificar en dos líneas
 2. Cambia el color del fondo de WebGL a un rojo puro (los parámetros de `gl.clearColor` se refieren a RGBA o red-green-blue-alpha, donde alpha es el nivel de transparencia).

SHADERS

Definir y compilar Shaders

- Los *shaders* son programas necesarios para dibujar algo en la pantalla con WebGL.
- WebGL necesita estos dos tipos de *shaders*:
 - **Vertex shader**: los vertex shaders son programas que describen las características de un **vertex** (posición, colores, etc).
 - El **vertex** (vértice) es un punto en el espacio 2D/3D, como la esquina o intersección de una forma 2D/3D.
 - **Fragment shader**: es un programa que se ocupa del procesamiento por fragmento, como la iluminación.
 - Un **fragment** es un término de WebGL que se asemeja a un pixel (elemento de imagen).

Definir y compilar Shaders



Linkar los Shaders con la aplicación

- Leer shaders
- Compilar shaders
- Crear un objeto de tipo programa
- Linkar todo junto
- Linkar variables de la aplicación con variables en los shaders
 - Atributos vertex
 - Variables Uniform

Leer un Shader

- Los shaders se añaden al objeto programa y se compilan.
- La forma más usual de pasar los shaders es como un string con terminación nula por medio de la función:

`gl.shaderSource(fragShdr, fragElem.text);`

- Si el shader está en el fichero HTML, podemos obtenerlo mediante el método:

`getElementById`

- Si el shader está en un fichero aparte, necesitamos convertir el fichero en un string.

Ejemplo: adding a Vertex Shader

```
var vertShdr;  
var vertElem = document.getElementById( vertexShaderId );  
  
vertShdr = gl.createShader( gl.VERTEX_SHADER );  
  
gl.shaderSource( vertShdr, vertElem.text );  
gl.compileShader( vertShdr );  
  
// after program object created  
gl.attachShader( program, vertShdr );
```

Example 1-3: Definir Shaders

```
<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>
```

Example 1-3: Código Relevante

- Se añaden dos nuevas etiquetas de tipo ***script***, pero en lugar de contener código Javascript se introduce un código propio de los shaders.
- Le decimos a HTML que ese código es “especial” indicando en el ***type*** que son “shader”.
- Cuando la página haya sido cargada se ejecuta la función “***initWebGL***”.
- Mira la creación del canvas para entender cómo se conecta el canvas con el método.

Definir Shaders

- IMPORTANTE: las aplicaciones WebGL consisten en un programa JavaScript ejecutado por el navegador y en programas shaders que son ejecutados dentro del sistema WebGL.
- El *Vertex Shader* especifica la ***posición*** del elemento y su ***tamaño***.
- El *Fragment Shader* especifica el ***color*** de los fragmentos que muestran el elemento.

Definición del Vertex Shader

- Debe contener una única función `main()`, muy similar a C:
 - `void` indica que la función no devuelve nada.
 - No se pueden pasar argumentos al `main`.
- Built-in variables:

Type and Variable Name	Description
<code>vec4 gl_Position</code>	Specifies the position of a vertex
<code>float gl_PointSize</code>	Specifies the size of a point (in pixels)

Tipos de datos en GLSL ES

- GLSL ES es un lenguaje fuertemente tipado, al igual que C y Java.

Type	Description				
float	Indicates a floating point number				
vec4	Indicates a vector of four floating point numbers				
<table><tr><td>float</td><td>float</td><td>Float</td><td>float</td></tr></table>		float	float	Float	float
float	float	Float	float		

Definición del Fragment Shader

- fragment = pixel + posición + color + etc
 - `void` indica que la función no devuelve nada.
- Igualmente, el punto de entrada a este shader es la función `main()`.
- Built-in variables:

Type and Variable Name	Description
<code>vec4 gl_FragColor</code>	Specify the color of a fragment (in RGBA)

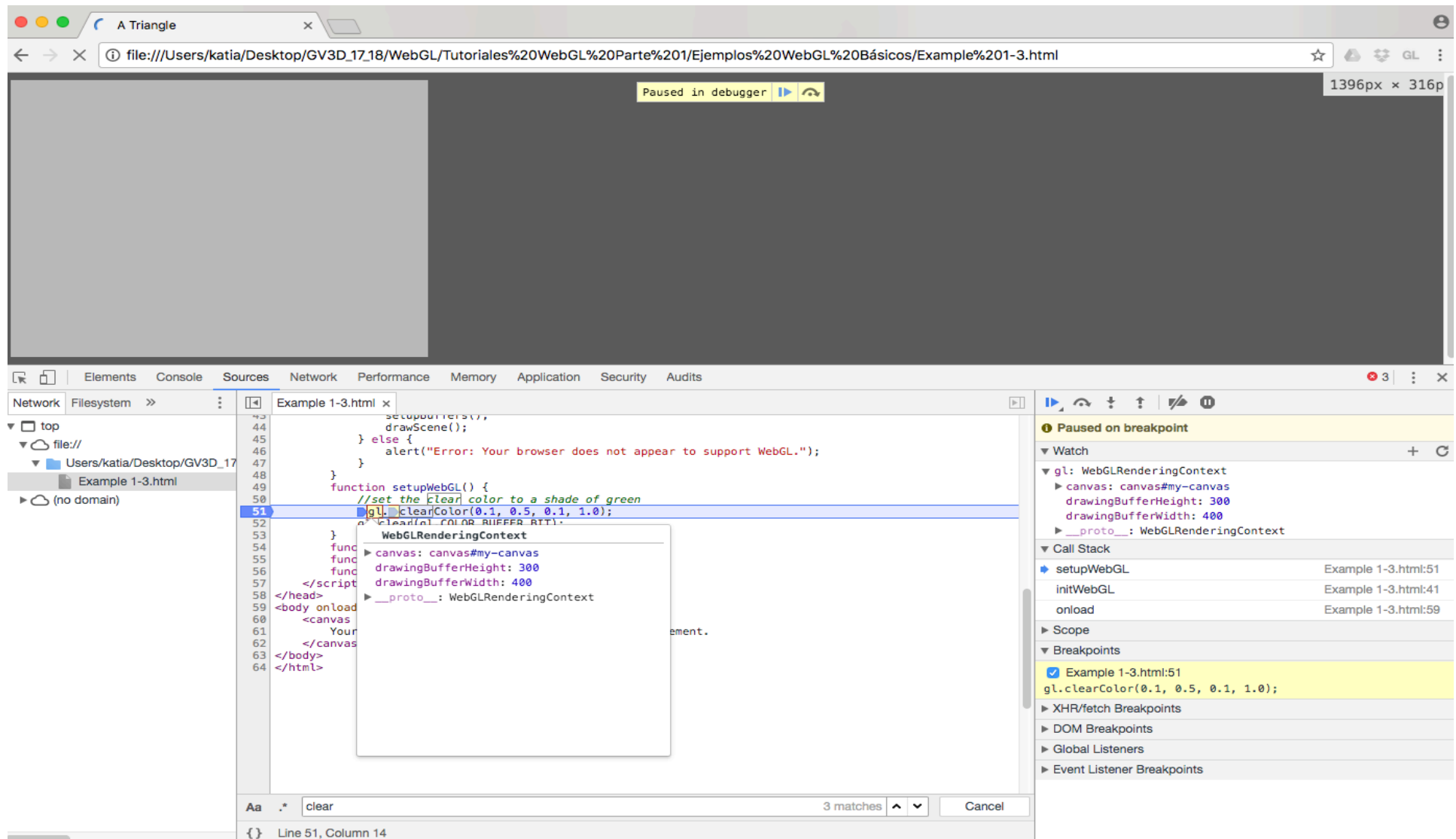
Example 1-3: *Aparecen los Shaders*

- Este ejemplo continúa con la preparación de WebGL y 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.

Código Relevante

- Se han 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”.
- Cuando la página haya sido cargada se ejecuta la función ***“initWebGL”***.

Example 1-3: *Aparecen los Shaders*



Example 1-3: Ejercicios

1. ¿Sabrías reproducir el proceso de depuración hasta llegar al mismo punto que la pantalla anterior? 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*”.
2. ¿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?
3. 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”.

Example 1-4: Compilar Shaders (I)

```
function initShaders() {  
    //get shader source  
    var fs_source = document.getElementById('shader-fs').html(),  
        vs_source = document.getElementById('shader-vs').html();  
    ...  
}
```


Example 1-4: Compilar Shaders (II)

```
function initShaders() {  
    ...  
    //compile shaders  
    vertexShader    = makeShader(vs_source, gl.VERTEX_SHADER);  
    fragmentShader = makeShader(fs_source, gl.FRAGMENT_SHADER);  
    ...  
}
```

Example 1-4: Compilar Shaders (II) cont.

```
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;  
}
```

Example 1-4: Crear objeto programa

El contenedor de Shaders puede contener múltiples shaders.

Otras funciones GLSL:

```
var program = gl.createProgram();  
  
gl.attachShader( program, vertShdr );  
gl.attachShader( program, fragShdr );  
gl.linkProgram( program );
```

Example 1-4: Compilar Shaders (III)

```
function initShaders() {  
    ...  
    //create program  
    glProgram = gl.createProgram();  
    //attach and link shaders to the program  
    gl.attachShader(glProgram, vertexShader);  
    gl.attachShader(glProgram, fragmentShader);  
    ...  
}
```

Example 1-4: Compilar Shaders (IV)

```
function initShaders(){  
    ...  
    gl.linkProgram(glProgram);  
    if (!gl.getProgramParameter(glProgram, gl.LINK_STATUS)){  
        alert("Unable to initialize the shader program.");  
    }  
    //use program  
    gl.useProgram(glProgram);  
}
```

Example 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

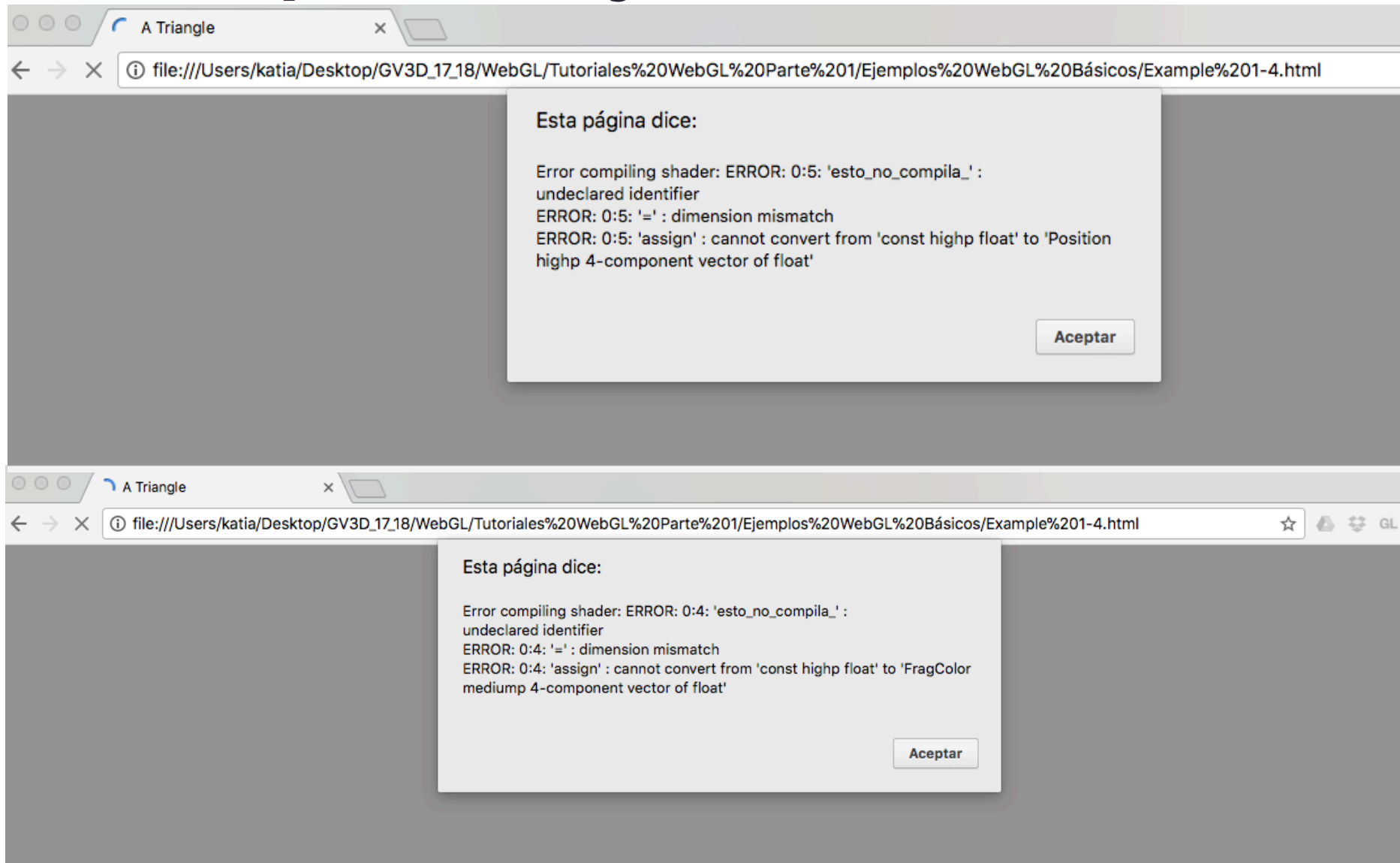
- Si observamos el código del ejemplo veremos que el principal cambio afecta al método `initShaders`. Hemos añadido una nueva función que realiza la compilación denominada “`makeShader`”. Esta función crea un shader, introduce el código del shader y lo compila.

Example 1-4: Ejercicios

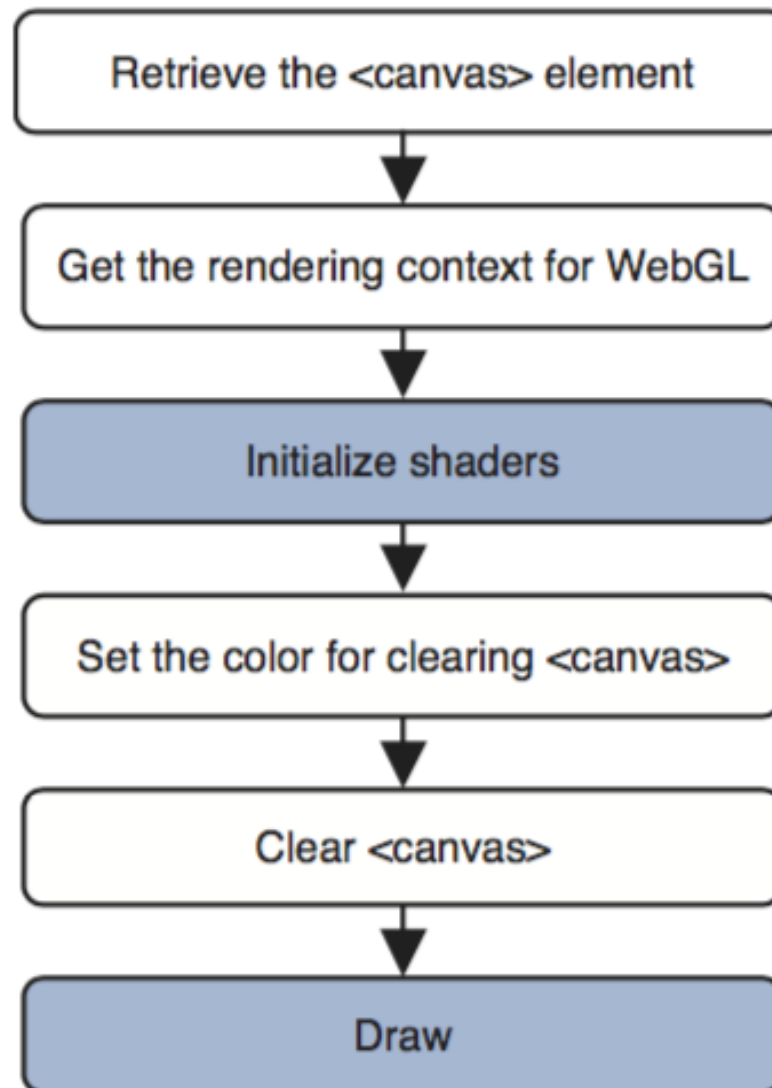
Observa que en este ejemplo tenemos una mezcla de HTML, Javascript, WebGL y “OpenGL Shading Language” (el lenguaje utilizado en los shaders). Estudia el código y aprende a distinguir el código que se corresponde con los shaders del propio de WebGL o del Javascript.

1. 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.
2. Cambia el código del vertex 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_;
3. 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_;

Example 1-4: Ejercicios

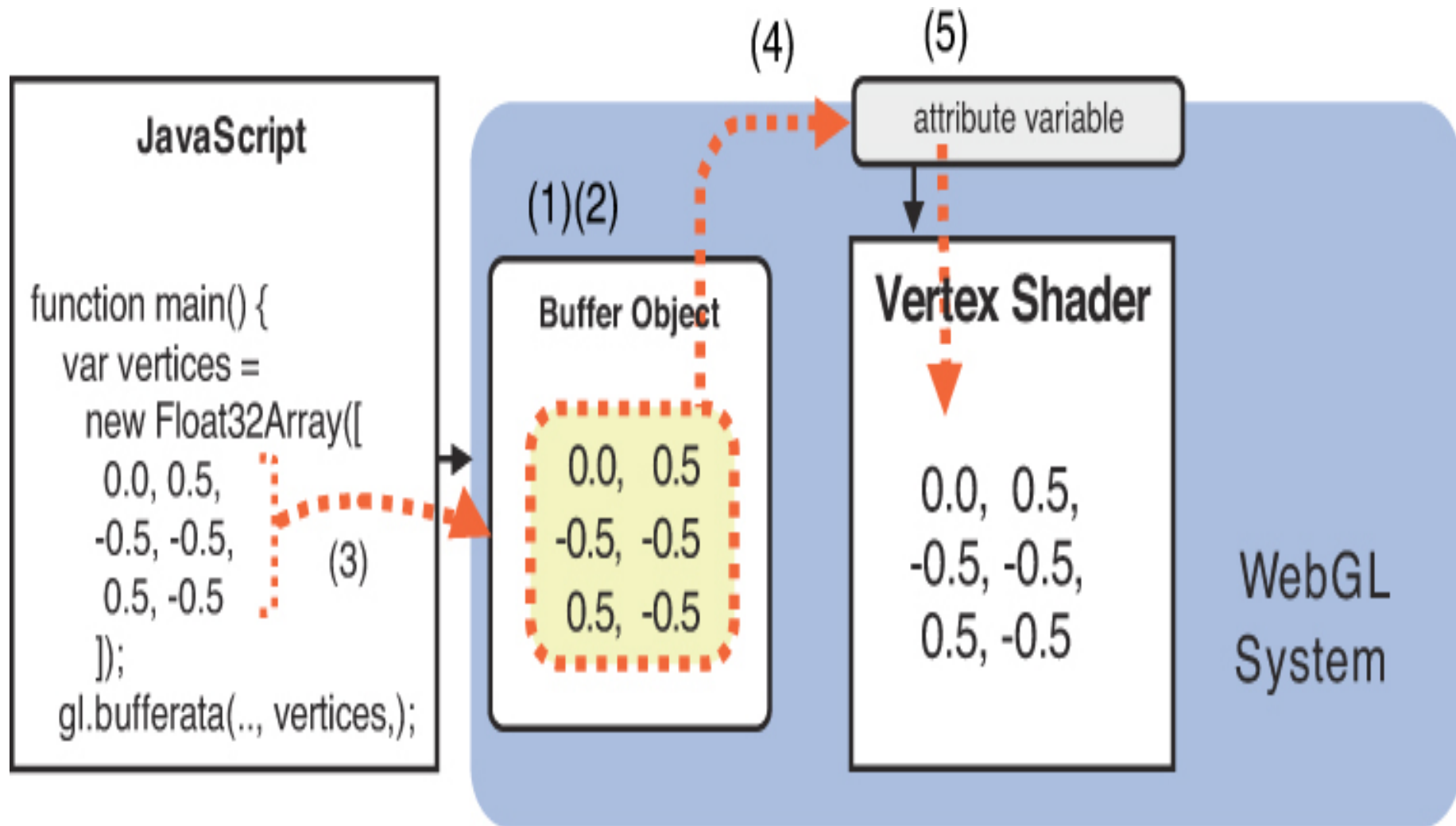


Flujo aplicación WebGL



Load Data

Example 1-5: *Cargamos los datos y dibujamos*



Example 1-5: *Cargamos los datos y dibujamos*

```
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);
```

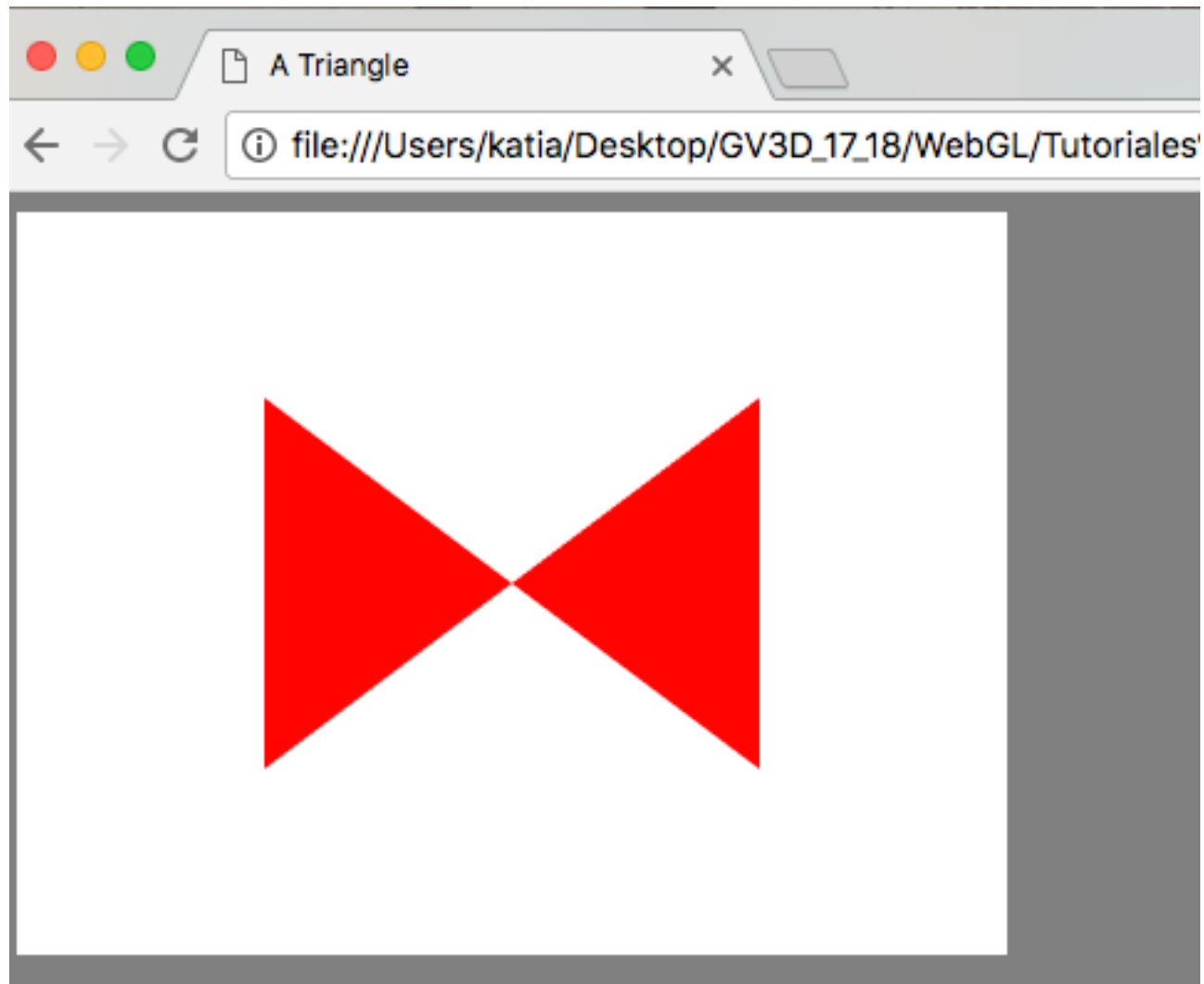
Example 1-5: *Cargamos los datos y dibujamos*

- 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.

Example 1-5: *Código relevante*

- “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).
- En la función “drawScene “ se conectan los datos con los atributos de los shader.
- Finalmente se hace el dibujo mediante “gl.drawArrays” (el número 6 se corresponde con el número de vértices que hemos almacenado).

Example 1-5



Example 1-5: *Ejercicios*

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

1. Cambia los triángulos de lado para que parezca un rombo.
2. ¿Sabrías dibujar un cuadrado?

Draw

Example 1-5: *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);  
}
```

gl.drawArrays(mode, first, count)

gl.drawArrays(mode, first, count)

Execute a vertex shader to draw shapes specified by the *mode* parameter.

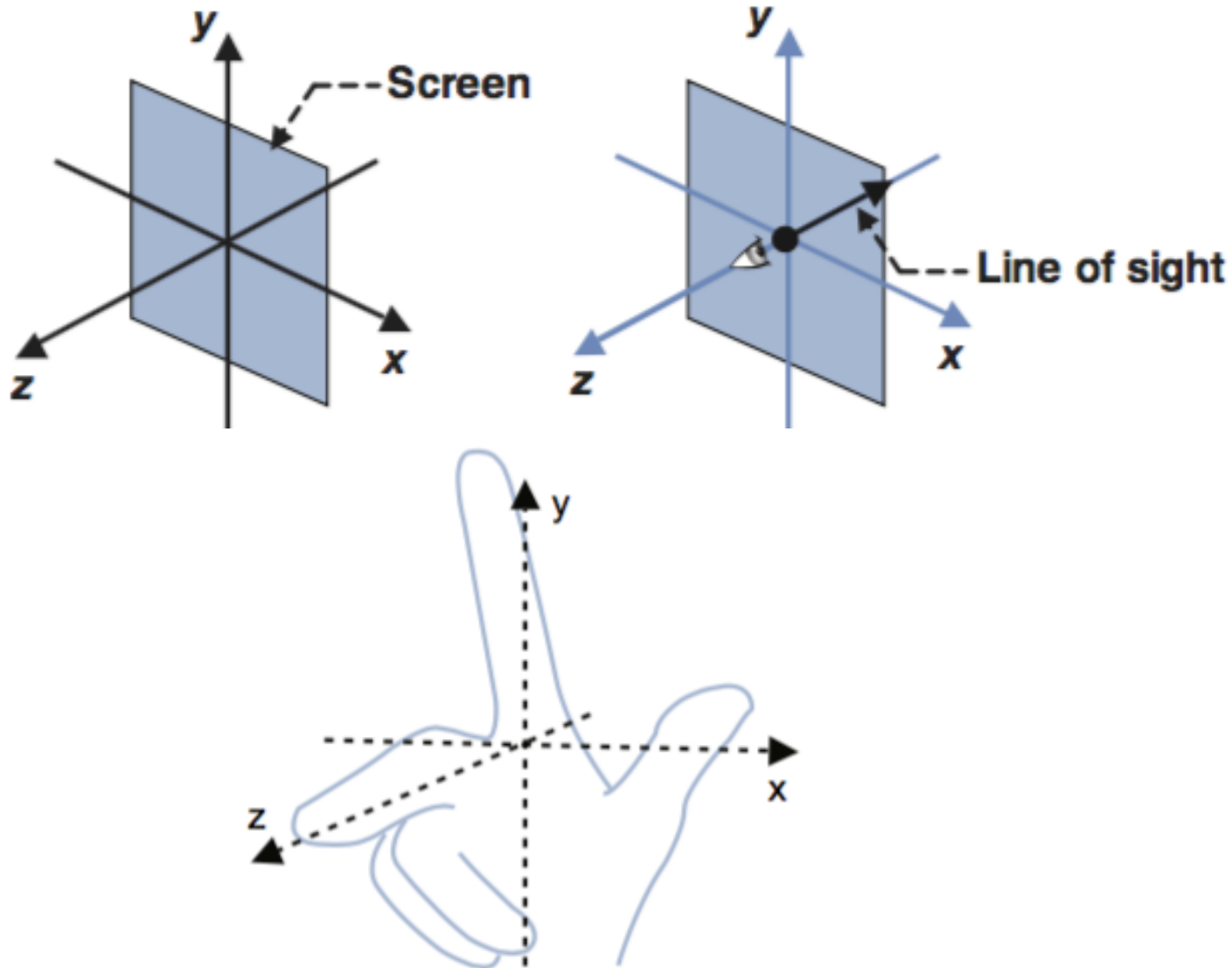
Parameters	mode	Specifies the type of shape to be drawn. The following symbolic constants are accepted: <code>gl.POINTS</code> , <code>gl.LINES</code> , <code>gl.LINE_STRIP</code> , <code>gl.LINE_LOOP</code> , <code>gl.TRIANGLES</code> , <code>gl.TRIANGLE_STRIP</code> , and <code>gl.TRIANGLE_FAN</code> .
	first	Specifies which vertex to start drawing from (integer).
	count	Specifies the number of vertices to be used (integer).

Return value None

Errors

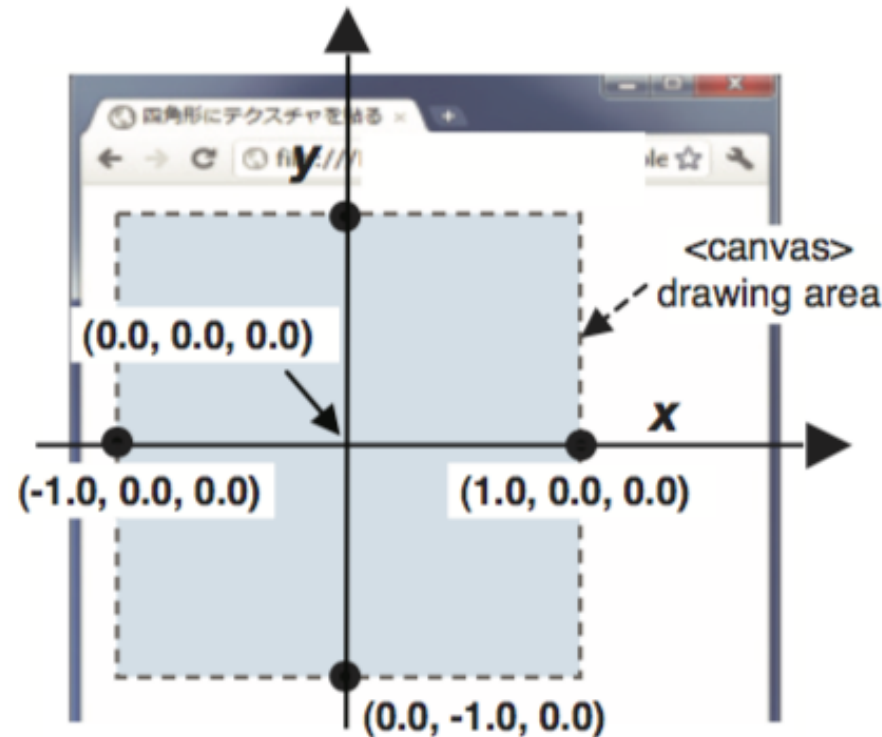
- `INVALID_ENUM` *mode* is none of the preceding values.
- `INVALID_VALUE` *first* is negative or *count* is negative.

Sistema de coordenadas de WebGL

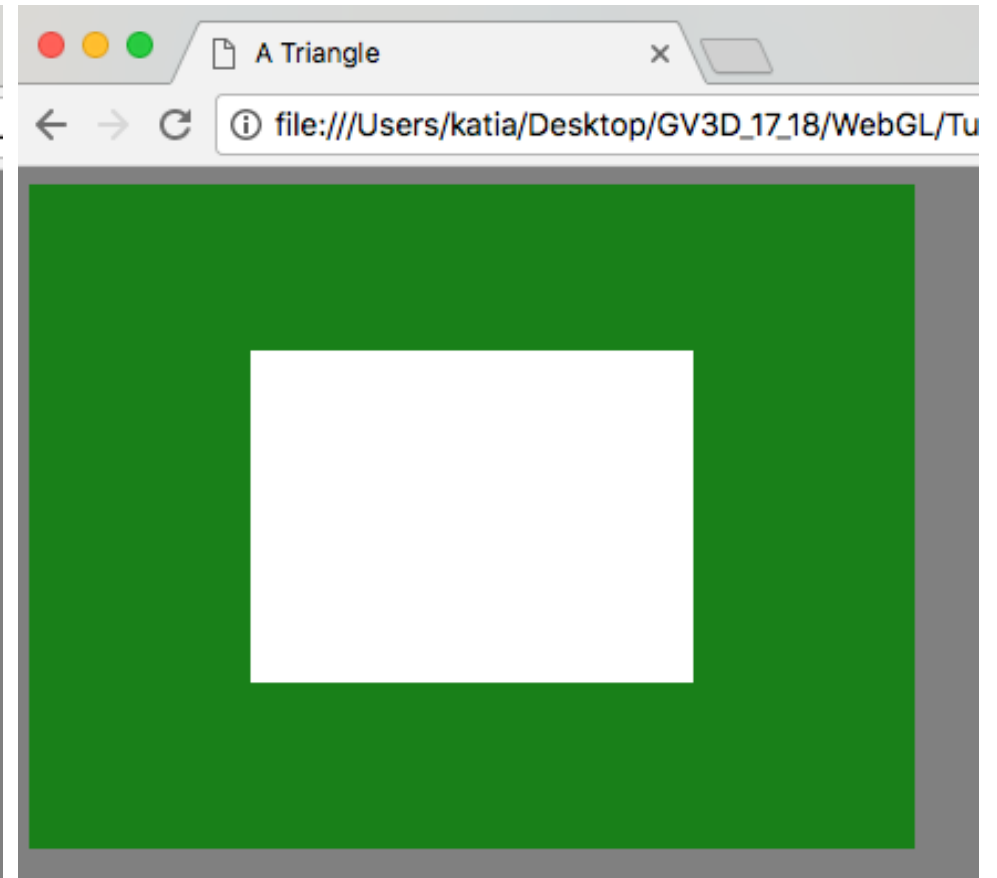
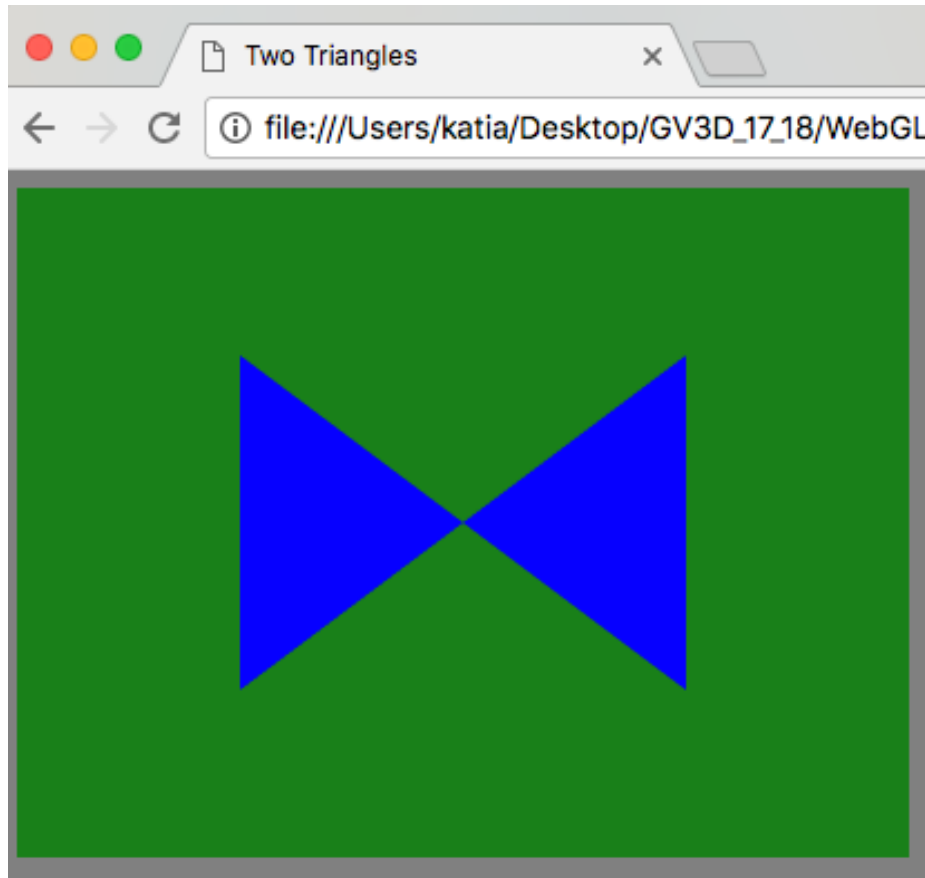


Sistema de coordenadas de WebGL

El área de dibujo del elemento `<canvas>` en JavaScript es diferente del sistema de coordenadas de WebGL.



Example 1-5 Two triangles y 1-5 Square



Example 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.
- 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>

Example 1-6: *Código relevante*

```
<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 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 {
```

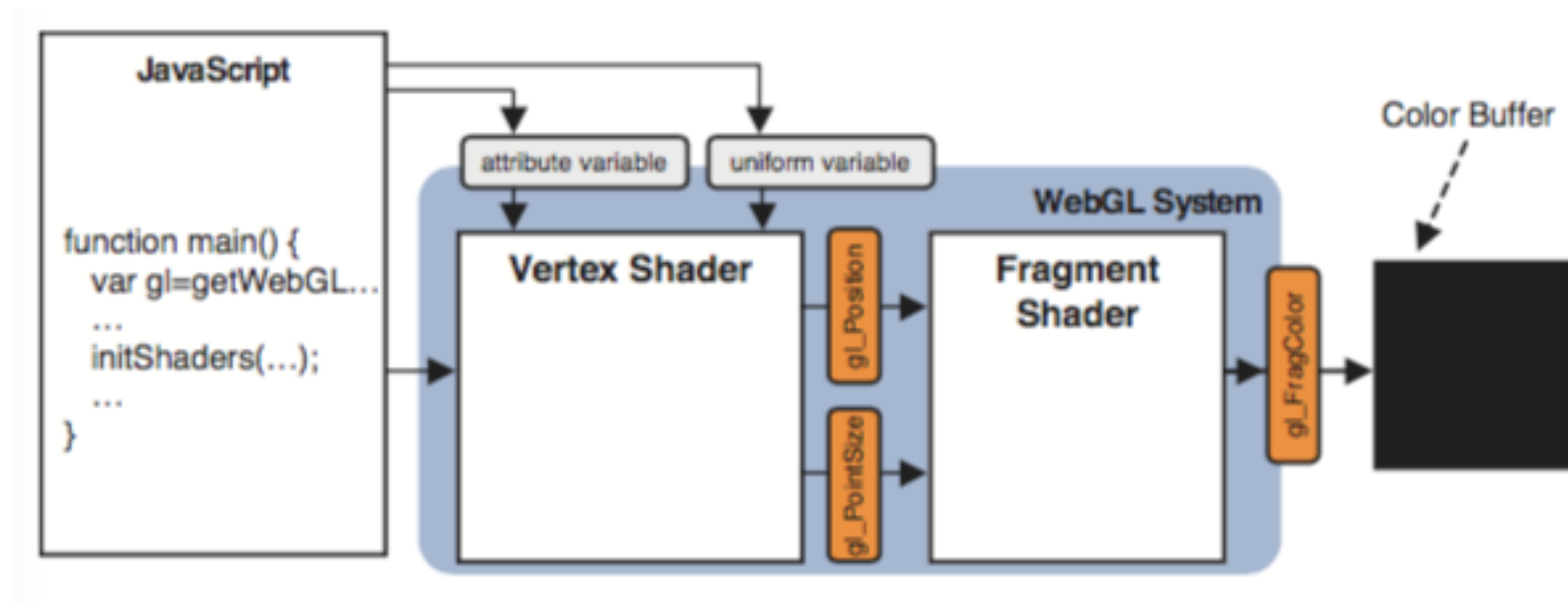

Example 1-6: *Depurando el código*

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

Pasar datos entre JavaScript y los shaders

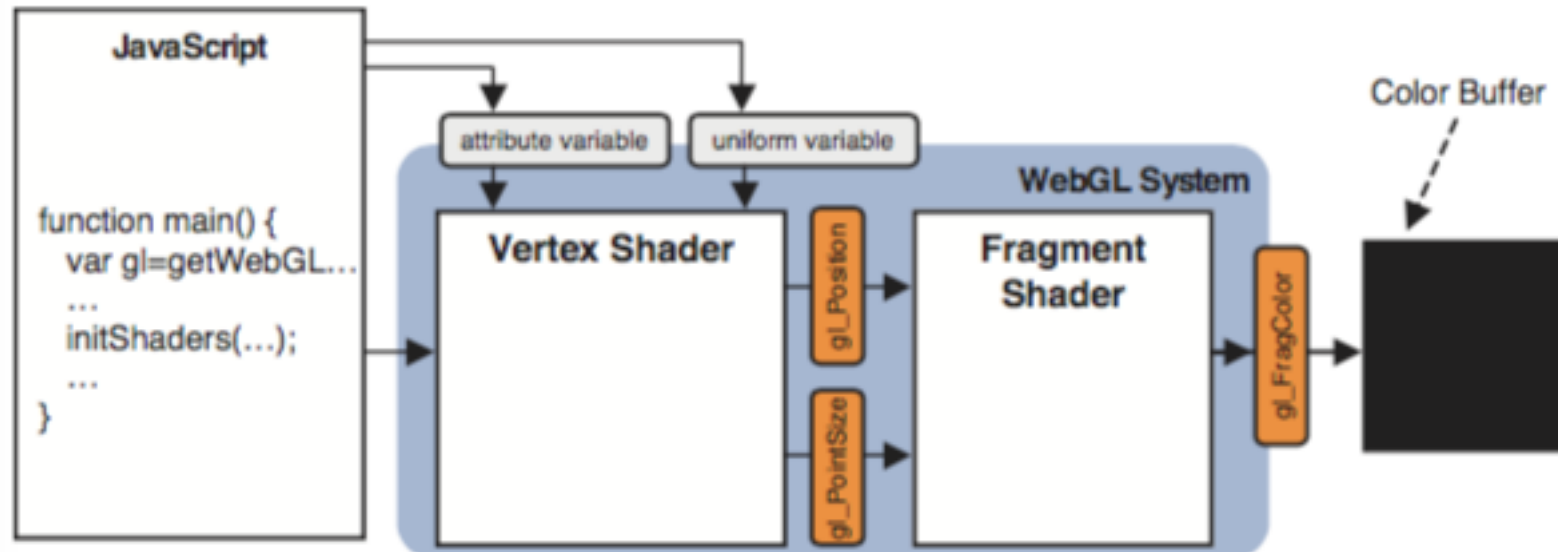
Variables *attribute* y *uniform*

- Se pueden pasar datos a un vertex de dos maneras:
 - Attribute variable
 - Uniform variable



Variables *attribute* y *uniform*

- **Attribute variable:** pasa datos que difieren para cada vertex.
- **Uniform variable:** pasa datos que son los mismos para todos los vertex.



Attribute variable

- Para usar una variable atributo:
 1. Se debe preparar la variable *atributo* para la posición del vertex en el vertex shader.
 2. Asignar la variable atributo a la variable `gl_Position`.
 3. Pasar los datos a la variable atributo

Attribute variable

- Definir una variable de tipo *attribute*:
attribute vec4 a_Position;

Storage Qualifier Type Variable Name

↙ ↙ ↙

attribute vec4 a_Position;

Attribute variable: ubicación

- Se debe preguntar al sistema WebGL el lugar de almacenamiento de la variable atributo.

```
gl.getAttribLocation(program, name)
```

Retrieve the storage location of the attribute variable specified by the *name* parameter.

Parameters	program	Specifies the program object that holds a vertex shader and a fragment shader.
	name	Specifies the name of the attribute variable whose location is to be retrieved.
Return value	greater than or equal to 0	The location of the specified attribute variable.
	-1	The specified attribute variable does not exist or its name starts with the reserved prefix <code>gl_</code> or <code>webgl_</code> .

Otras funciones interesantes

```
function drawScene() {  
    vertexPositionAttribute = gl.getAttributeLocation(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);  
}
```

```
trianglesVerticeBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);  
gl.bufferData(gl.ARRAY_BUFFER, new  
              Float32Array(triangleVertices), gl.STATIC_DRAW);
```


enableVertexAttribArray()

- Los atributos se referencian por un número de índice en la lista de atributos mantenidos por la GPU.
- Los atributos no se pueden usar a menos que estén habilitados (están deshabilitados por defecto). Se debe llamar a `enableVertexAttribArray()` para habilitar atributos de forma individual para que puedan ser utilizados. Una vez hecho esto, se pueden usar otros métodos para acceder al atributo.

Syntax

- `void gl.enableVertexAttribArray(index);`

Parameters

- `index`: the index number that uniquely identifies the vertex attribute to enable. If you know the name of the attribute but not its index, you can get the index by calling `getAttribLocation()`.

vertexAttribPointer()

- Este método especifica la disposición del objeto vertex buffer en memoria.
- Se debe invocar una vez para cada atributo vertex.

Syntax

- `void gl.vertexAttribPointer(index, size, type, normalized, stride, offset);`

Parameters

- **index**: the index of the vertex attribute that is to be modified.
- **size**: the number of components per vertex attribute. Must be 1, 2, 3, or 4.
- **type**: data type of each component in the array. Possible values:
 - `gl.FLOAT`: 32-bit IEEE floating point number.
- **normalized**: specifying whether integer data values should be normalized into a certain range when being casted to a float.
 - For types `gl.FLOAT` and `gl.HALF_FLOAT`, this parameter has no effect.
- **stride**: If stride is 0, each attribute is in a separate block, and the next vertex attribute follows immediately after the current vertex.
- **offset**: an offset in bytes of the first component in the vertex attribute array. Must be a multiple of type.

createBuffer()

- Crea e inicializa un *WebGLBuffer* que almacena datos como vértices o colores.

Syntax

- `WebGLBuffer gl.createBuffer();`

Parameters

- None

Return value

- A *WebGLBuffer* storing data such as vertices or colors.

bindBuffer()

- Vincula un *WebGLBuffer* a un *target*.

Syntax

- `void gl.bindBuffer(target, buffer);`

Parameters

- **target**: the binding point (target). Possible values:
 - `gl.ARRAY_BUFFER`: Buffer containing vertex attributes, such as vertex coordinates, texture coordinate data, or vertex color data.
 - `gl.ELEMENT_ARRAY_BUFFER`: Buffer used for element indices.
- **buffer**: A *WebGLBuffer* to bind.

bufferData()

- Crea e inicializa el almacén de datos del objeto buffer.

Syntax

- `void gl.bufferData(target, ArrayBuffer srcData, usage);`

Parameters

- **target**: the binding point (target). Possible values:
 - `gl.ARRAY_BUFFER`: Buffer containing vertex attributes, such as vertex coordinates, texture coordinate data, or vertex color data.
 - `gl.ELEMENT_ARRAY_BUFFER`: Buffer used for element indices.
- **srcData**: An *ArrayBuffer*, *SharedArrayBuffer* or one of the *ArrayBufferView* typed array types that will be copied into the data store. If null, a data store is still created, but the content is uninitialized and undefined.
- **usage**: the usage pattern of the data store. Possible values:
 - `gl.STATIC_DRAW`: Contents of the buffer are likely to be used often and not change often. Contents are written to the buffer, but not read.
 - `gl.DYNAMIC_DRAW`: Contents of the buffer are likely to be used often and change often. Contents are written to the buffer, but not read.
 - `gl.STREAM_DRAW`: Contents of the buffer are likely to not be used often. Contents are written to the buffer, but not read.

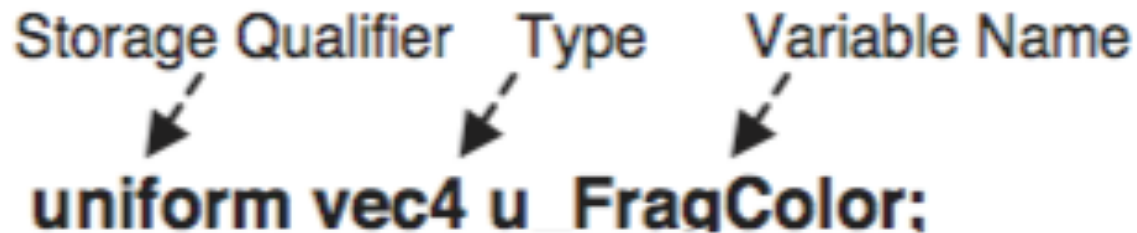
Uniform variable

- Para pasarle un dato al fragment shader:
 1. Se debe preparar la variable *uniforme* para el color en el fragment shader.
 2. Asignar la variable *uniforme* a la variable `gl_FragColor`.
 3. Pasar el dato del color a la variable *uniforme* desde el programa JavaScript.

Uniform variable

- Definir una variable de tipo *uniform*:
uniform vec4 u_FragColor ;

Storage Qualifier Type Variable Name



uniform vec4 u_FragColor;

Uniform variable: ubicación

- Se debe preguntar al sistema WebGL el lugar de almacenamiento de la variable atributo.

```
gl.getUniformLocation(program, name)
```

Retrieve the storage location of the uniform variable specified by the *name* parameter.

Parameters	program	Specifies the program object that holds a vertex shader and a fragment shader.
	name	Specifies the name of the uniform variable whose location is to be retrieved.

Asignar un valor a una variable *Attribute*

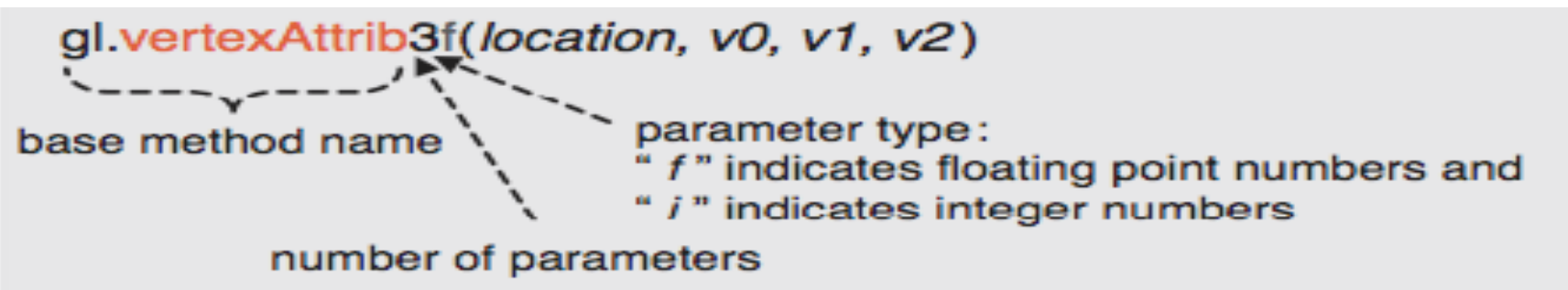
- Partiendo de la ubicación de la variable, podemos asignar el valor a la variable del vertex shader por medio de uno de los siguientes métodos:

```
gl.vertexAttrib1f(location, v0)
gl.vertexAttrib2f(location, v0, v1)
gl.vertexAttrib3f(location, v0, v1, v2)
gl.vertexAttrib4f(location, v0, v1, v2, v3)
```

Parameters	location	Specifies the storage location of the attribute variable.
	v0, v1, v2, v3	Specifies the values to be assigned to the first, second, third, and fourth components of the attribute variable.
Return value	None	
Errors	INVALID_VALUE	location is greater than or equal to the maximum number of attribute variables (8 by default).

Asignar un valor a una variable *Attribute*

- Reglas para nombrar los métodos de WebGL:



When `v` is appended to the name, the methods take an array as a parameter. In this case, the number in the method name indicates the number of elements in the array.

```
var positions = new Float32Array([1.0, 2.0, 3.0, 1.0]);  
gl.vertexAttrib4fv(a_Position, positions);
```

Asignar un valor a una variable *Uniform*

- Partiendo de la ubicación de la variable, podemos asignar el valor a la variable del fragment shader por medio de uno de los siguientes métodos:

```
gl.uniform1f(location, v0)
gl.uniform2f(location, v0, v1)
gl.uniform3f(location, v0, v1, v2)
gl.uniform4f(location, v0, v1, v2, v3)
```

Parameters	location	Specifies the storage location of a uniform variable.
	v0, v1, v2, v3	Specifies the values to be assigned to the first, second, third, and fourth component of the uniform variable.
Return value	None	
Errors	INVALID_OPERATION	There is no current program object.
		<i>location</i> is an invalid uniform variable location.

Otros calificativos

- Hay variables que se pasan desde el vertex shader a el fragment shader.
- En WebGL, GLSL utilizar el calificativo ***varying*** en ambos shaders:

```
varying vec4 color;
```

- Versiones más recientes de WebGL utilizan ***out*** en el vertex shader e ***in*** en el fragment shader.

```
out vec4 color; //vertex shader
```

```
in vec4 color; // fragment shader
```