



Universidad  
Rey Juan Carlos

Escuela Técnica Superior  
Ingeniería de Telecomunicación



# Gráficos y visualización 3D

---

## 3. Conceptos básicos en WebGL

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: [JOSEMIGUEL.GUERRERO@URJC.ES](mailto:JOSEMIGUEL.GUERRERO@URJC.ES)

# Índice de contenidos

---

1. Visión general de WebGL
2. WebGL
3. Ejemplo: colorear un canvas
4. Ejemplo: dibujar un punto
5. Ejemplo: dibujar un triángulo
6. Ejemplo: dibujar un rectángulo
7. Ejemplo: degradado de color en triángulo
8. Ejemplo: degradado de color en rectángulo
9. Resumen

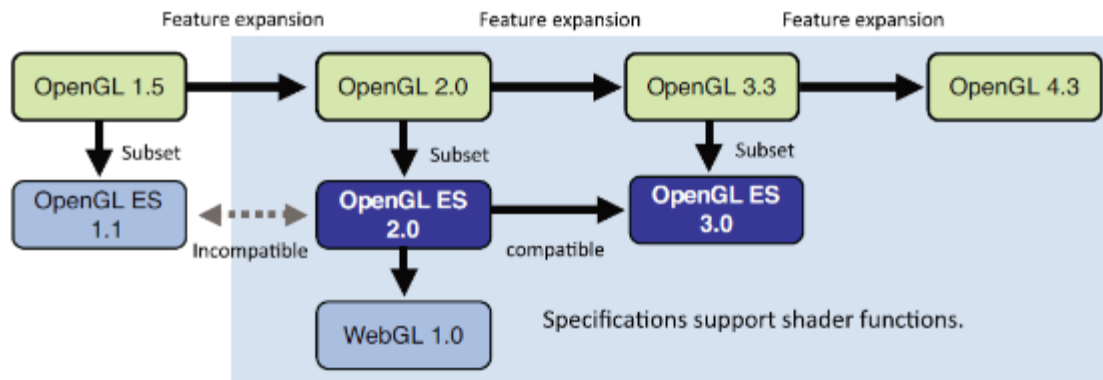
# Índice de contenidos

---

1. Visión general de WebGL
  - I. Introducción
  - II. Shaders
  - III. Pipeline
  - IV. Aplicaciones WebGL
  - V. Canvas
2. WebGL
3. Ejemplo: colorear un canvas
4. Ejemplo: dibujar un punto
5. Ejemplo: dibujar un triángulo
6. Ejemplo: dibujar un rectángulo
7. Ejemplo: degradado de color en triángulo
8. Ejemplo: degradado de color en rectángulo
9. Resumen

# 1. Visión general de WebGL - Introducción

- Como ya sabemos, **WebGL** (*Web Graphics Library*) es una especificación estándar que define una API para la generación de gráficos en 3D en navegadores web
- WebGL deriva de OpenGL ES (*Embedded Systems*)



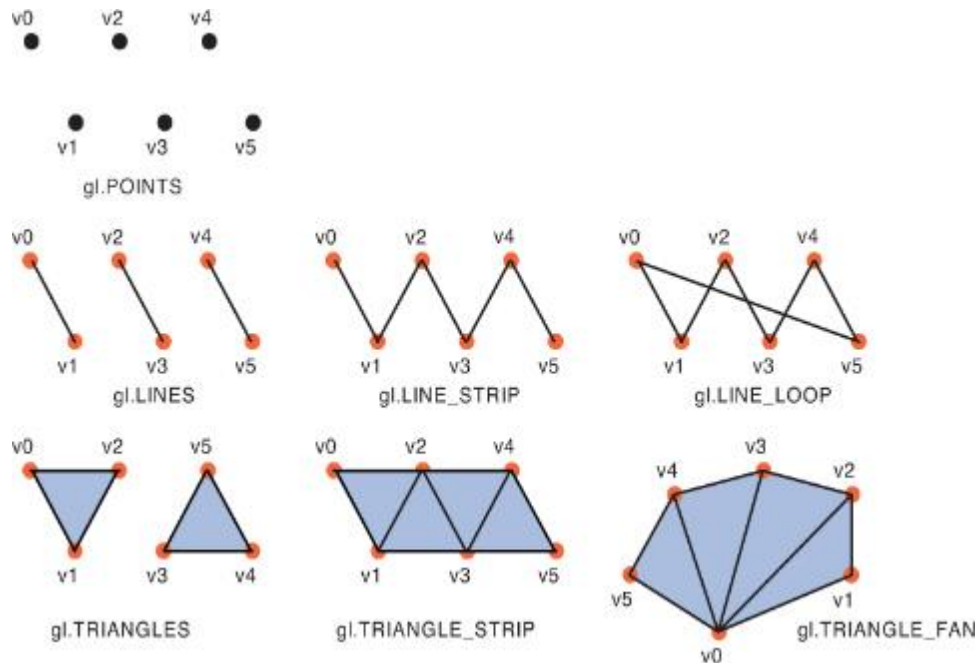
# 1. Visión general de WebGL - Shaders

---

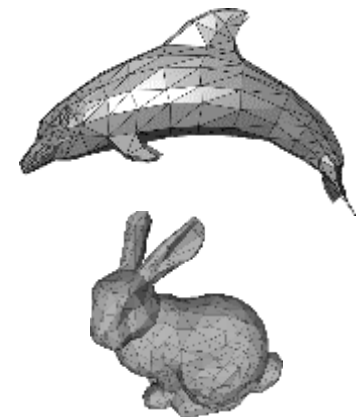
- OpenGL/WebGL proporcionan la capacidad de crear unos programas llamados ***shaders***
- Un *shader* es un fragmento de código que permite generar gráficos 3D mediante un lenguaje de programación denominado **GLSL** (*Graphics Library Shader Language*)
  - La sintaxis de GLSL es similar a C
  - En WebGL se usa una versión reducida de GLSL denominada GLSL ES
- Los shaders se ejecutan en la **GPU** de un ordenador

# 1. Visión general de WebGL - Shaders

- Los gráficos manejados en los shaders estarán formados por formas básicas creadas a través de **primitivas GLSL**:



En base a estas formas básicas se pueden crear gráficos 3D realistas



# 1. Visión general de WebGL - Shaders

---

- Hay dos tipos de shaders necesarios para generar gráficos en WebGL:
  1. Vértices (***vertex shader***): Programa que describe la posición de los elementos de una escena (modelo)
  2. Fragmentos (***fragment shader***): Programa que se ocupa del procesamiento de los vértices, color, iluminación, etc.

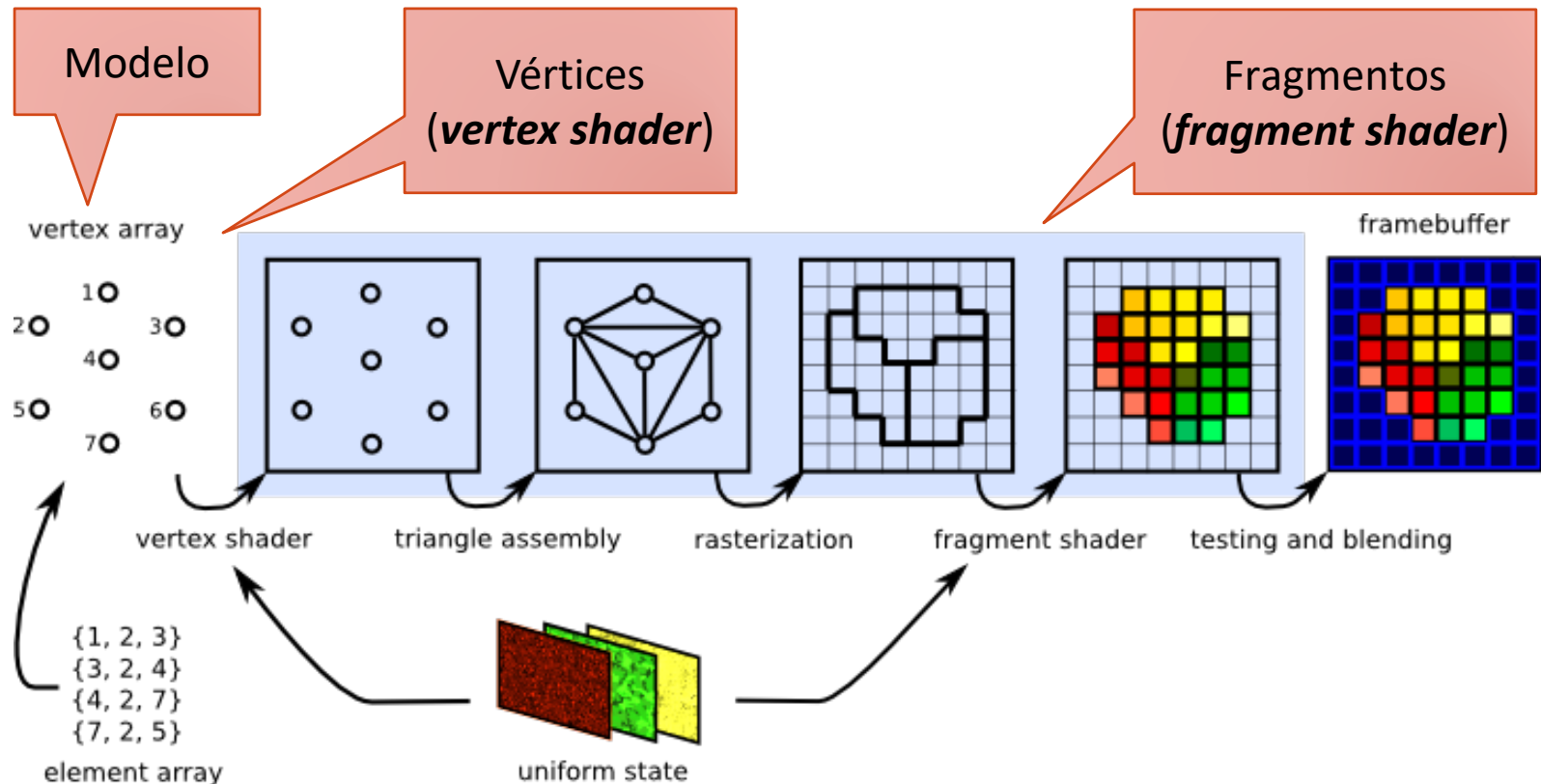
# 1. Visión general de WebGL - Pipeline

---

- OpenGL/WebGL trabaja en un bucle infinito
  - Sitúa elementos en la escena (puntos, líneas, polígonos)
  - Procesa la posición y orientación de la cámara
  - Atiende los eventos del teclado
  - Dibuja la escena
- El funcionamiento de OpenGL/WebGL se basa en el procesamiento de los gráficos que componen la escena en diferentes fases (*pipeline*)



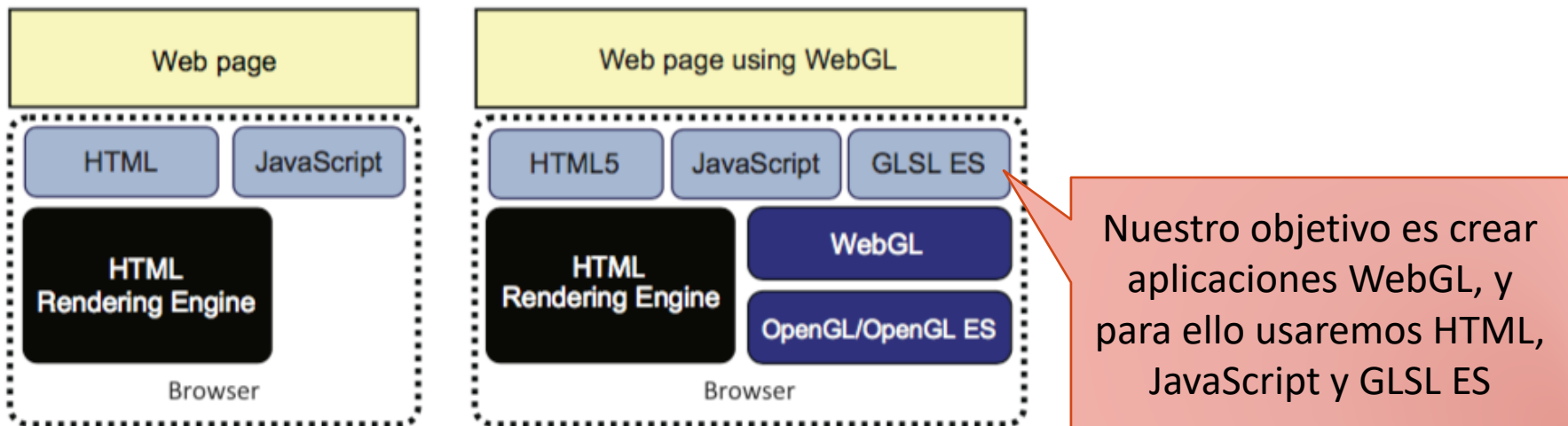
# 1. Visión general de WebGL - Pipeline



<https://www.enlightenment.org/playground/evas-gl.md>

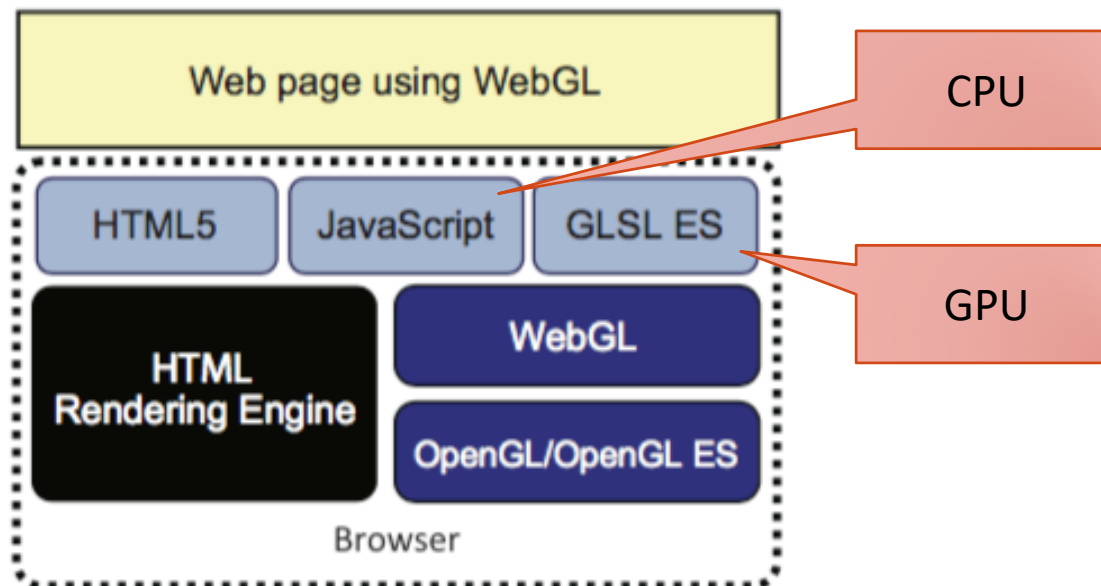
# 1. Visión general de WebGL – Aps WebGL

- Una aplicación WebGL es una aplicación web que hace uso de WebGL para generar gráficos 3D. Por lo tanto estará formada como mínimo por HTML, JavaScript, y GLSL ES (para los *vertex* y *fragment shaders*)



# 1. Visión general de WebGL – Aps WebGL

- Su ejecución se llevará a cabo parte en la CPU y parte en la GPU



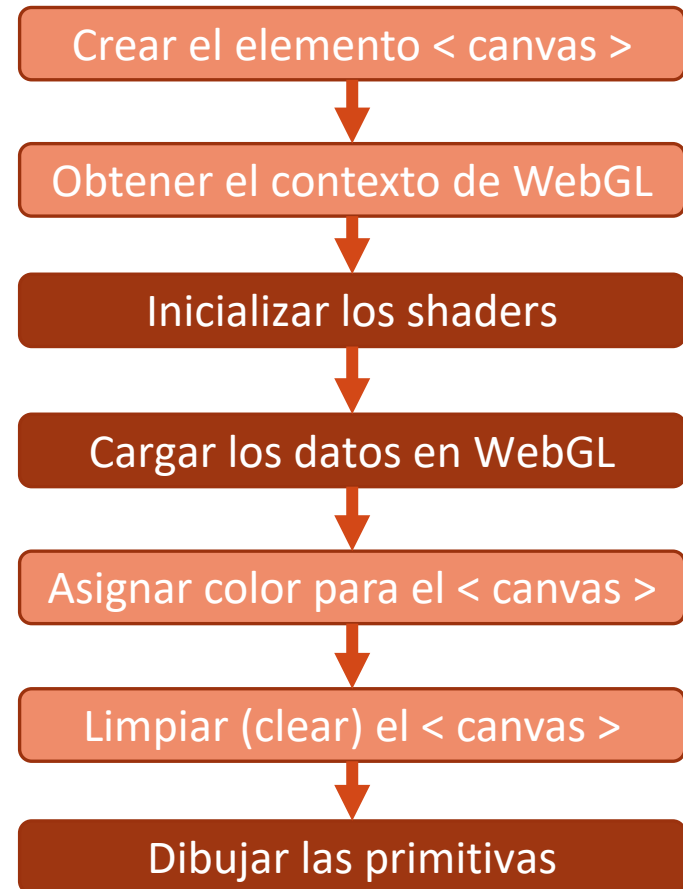
# Índice de contenidos

---

1. Visión general de WebGL
- 2. WebGL**
3. Ejemplo: colorear un canvas
4. Ejemplo: dibujar un punto
5. Ejemplo: dibujar un triángulo
6. Ejemplo: dibujar un rectángulo
7. Ejemplo: degradado de color en triángulo
8. Ejemplo: degradado de color en rectángulo
9. Resumen

## 2. WebGL – Flujo de una aplicación

- Por regla general, para desarrollar un programa en WebGL se tiene que:
  - Crear un **canvas**
  - Crear un **contexto** WebGL
  - Inicializar los **shaders**
  - **Cargar los datos** en WebGL
  - Preparar el **canvas**
  - **Dibujar** las primitivas

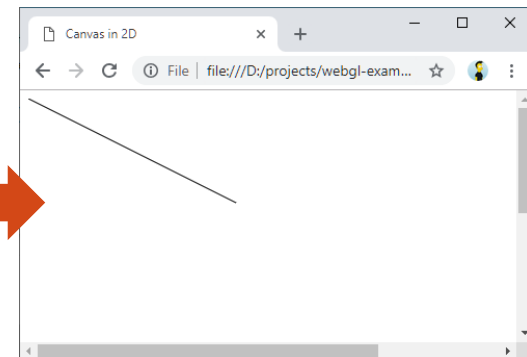


## 2. WebGL – Crear un canvas 2D

- El elemento `<canvas>` de HTML5 permite definir un área de dibujo dentro de una página web
  - Sin WebGL, `<canvas>` permite dibujar gráficos 2D
  - Con WebGL, `<canvas>` permite dibujar gráficos 3D

Ejemplo de  
dibujo 2D en  
`<canvas>`

```
<!DOCTYPE html>
<html>
<head>
  <title>Canvas in 2D</title>
</head>
<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>
<script>
  function init() {
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
    context.moveTo(0, 0);
    context.lineTo(200, 100);
    context.stroke();
  }
</script>
</html>
```



## 2. WebGL – Crear un canvas 2D

```
<!DOCTYPE html>
<html>
<head>
  <title>Canvas in 2D</title>
</head>
<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>
<script>
  function init() {
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
    context.moveTo(0, 0);
    context.lineTo(200, 100);
    context.stroke();
  }
</script>
</html>
```

Creamos el  
<canvas>

En JavaScript necesitamos obtener el <canvas> que hemos creado, para ello utilizamos la función `getElementById` seguido del nombre de la variable en el HTML

Usamos el contexto para dibujar, en este caso una línea

Creamos el contexto a partir de la variable canvas del JavaScript, en este caso para elementos 2D (`2d`)

<https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D>

## 2. WebGL – Contexto

```
canvas.getContext(contextType, contextAttributes);
```

### Parámetros

#### contextType

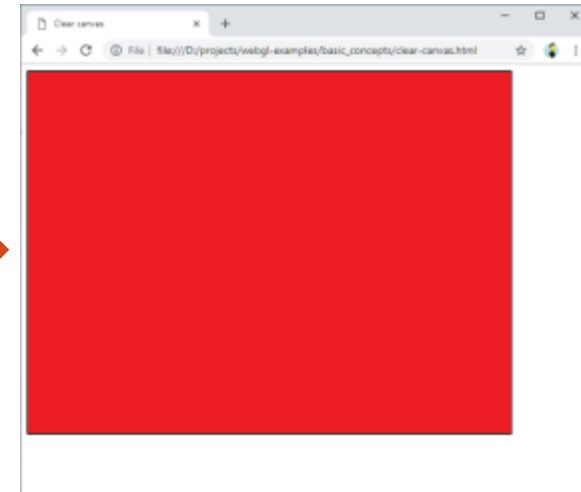
Es una `DOMString` que contiene el identificador del contexto que define el contexto de dibujo asociado a el lienzo. Los posibles valores son:

- `"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).



## 2. WebGL – Crear un canvas 3D

```
<!DOCTYPE html>
<html>
<head>
  <title>Canvas in 3D</title>
</head>
<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>
<script>
  function init() {
    var canvas = document.getElementById("myCanvas");
    var gl = canvas.getContext("webgl");
    gl.clearColor(1.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
  }
</script>
</html>
```



Usamos las funciones  
de WebGL para dar  
color al fondo

Ahora creamos el contexto  
para elementos 3D,  
utilizando `webgl`

## 2. WebGL – Referencia API

---

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

---

Specify the clear color for a drawing area:

<b>Parameters</b>	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.	
<b>Return value</b>	None	
<b>Errors<sup>2</sup></b>	None	

## 2. WebGL – Referencia API

### `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

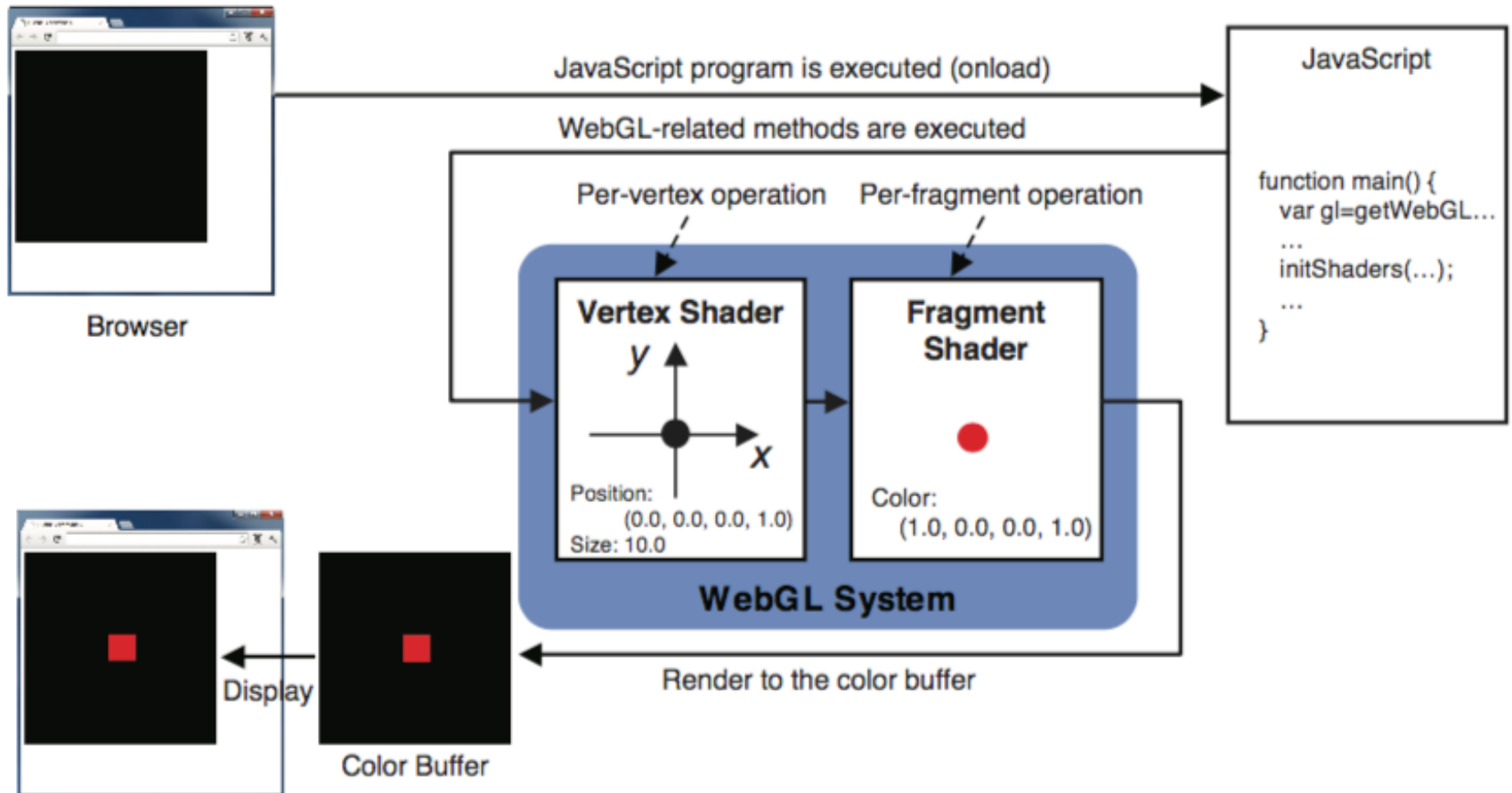
**Errors**    `INVALID_VALUE`    *buffer* is none of the preceding three values.

## 2. WebGL – Shaders

---

- Los *shaders* son programas necesarios para dibujar algo en la pantalla con WebGL
- WebGL necesita dos tipos de *shaders*:
  - **Vertex shader**: los vertex shaders son programas que describen las características de un **vertex** (posición, tamaño)
    - 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 o el color
    - Un **fragment** es un término de WebGL que se asemeja a un pixel (elemento de imagen)

## 2. WebGL – Shaders



## 2. WebGL – Shaders

---

- Los ***shaders*** hay que incluirlos en la aplicación, para ello hay que seguir los siguientes pasos:
  - Definir los shaders
  - 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

## 2. WebGL – Shaders

---

- Las aplicaciones WebGL consisten en un programa JavaScript ejecutado por el navegador, y además 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.

## 2. WebGL – Definir 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.
- Variables GLSL ES:

**Table 2.2** Built-In Variables Available in a Vertex Shader

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)



## 2. WebGL – Definir Fragment 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.
- fragment = pixel + posición + color + etc
- Variables GLSL ES:

Table 2.4 The Built-In Value Available in a Fragment Shader

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

## 2. WebGL – API GLSL ES

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

Table 2.3 Data Types in GLSL ES

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		

## 2. WebGL – Definir shaders

- 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”.

```
<script id="shaderVs" type="x-shader/x-vertex">
    void main() {
        gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
        gl_PointSize = 10.0;
    }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
    void main() {
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
    }
</script>
```

Vertex Shader

Fragment Shader

## 2. WebGL – Leer shaders

---

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

```
var vs = document.getElementById('shaderVs').innerHTML;  
var fs = document.getElementById('shaderFs').innerHTML;
```

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

## 2. WebGL – Inicializar shaders

- Una vez leídos los shaders, hay que inicializarlos, para ello creamos la siguiente función:

```
initShaders(gl_context, vs_source, fs_source)
```

- En primer lugar hay que crear y compilar los shaders. Creamos la función `makeShader(context, source, type)`:

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

    // ...
}
```

## 2. WebGL – Compilar shaders

- `makeShader(context, source, type)`, esta función, utilizando el contexto indicado (`context`), se encarga de:
  - Crear el shader del tipo (`type`) indicado: `createShader`
  - Linkar el shader que hemos creado a la variable JavaScript que contiene la función (`source`): `shaderSource`
  - Compilar el shader creado y linkado: `compileShader`

```
function makeShader(gl, src, type) {
    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;
    }
    return shader;
}
```

Es bueno  
comprobar que la  
compilación ha  
tenido éxito

## 2. Referencia API WebGL

### `gl.createShader(type)`

Create a shader of the specified *type*.

<b>Parameters</b>	<code>type</code>	Specifies the type of shader object to be created: either <code>gl.VERTEX_SHADER</code> (a vertex shader) or <code>gl.FRAGMENT_SHADER</code> (a fragment shader).
<b>Return value</b>	Non-null	The created shader object.
	null	The creation of the shader object failed.
<b>Errors</b>	<code>INVALID_ENUM</code>	The specified type is none of the above.

## 2. Referencia API WebGL

### `gl.shaderSource(shader, source)`

Store the source code specified by *source* in the shader object specified by *shader*. If any source code was previously stored in the shader object, it is replaced by new source code.

<b>Parameters</b>	shader	Specifies the shader object in which the program is stored.
	source	Specifies the shader source code (string)

**Return value**      None

**Errors**              None

### `gl.compileShader(shader)`

Compile the source code stored in the shader object specified by *shader*.

<b>Parameters</b>	shader	Specifies the shader object in which the source code to be compiled is stored.
-------------------	--------	--

**Return Value**      None

**Errors**              None



## 2. Referencia API WebGL

### `gl.getShaderParameter(shader, pname)`

Get the information specified by *pname* from the shader object specified by *shader*.

<b>Parameters</b>	<code>shader</code>	Specifies the shader object.
	<code>pname</code>	Specifies the information to get from the shader:  <code>gl.SHADER_TYPE</code> , <code>gl.DELETE_STATUS</code> , or  <code>gl.COMPILE_STATUS</code> .
<b>Return value</b>	The following depending on <i>pname</i> :	
	<code>gl.SHADER_TYPE</code>	The type of shader ( <code>gl.VERTEX_SHADER</code> or <code>gl.FRAGMENT_SHADER</code> )
	<code>gl.DELETE_STATUS</code>	Whether the deletion has succeeded (true or false)
	<code>gl.COMPILE_STATUS</code>	Whether the compilation has succeeded (true or false)
<b>Errors</b>	<code>INVALID_ENUM</code>	<i>pname</i> is none of the above values.

## 2. WebGL – Crear objeto del programa

- Una vez se tienen creados los shaders:
  - Se crea el objeto contenedor del programa (`gl.createProgram()`)
  - Se añaden los dos shaders (`gl.attachShader(glProgram, vertexShader/fragmentShader)`)
  - Se linkan al programa (`gl.linkProgram(glProgram)`)
  - Se indica a JavaScript que utilice el programa (`gl.useProgram(glProgram)`)

```
function initShaders(gl, vs_source, fs_source) {
    // Compile shaders
    // Create program
    var 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");
        return false;
    }

    // Use program
    gl.useProgram(glProgram);
    gl.program = glProgram;

    return true;
}
```

Llamada a los  
makeShader para  
crear el vertex y el  
fragment shader

Indica que la  
inicialización de los  
shaders ha tenido  
éxito

## 2. Referencia API WebGL

### `gl.createProgram()`

Create a program object.

**Parameters**      None

**Return value**      non-null      The newly created program object.  
                          null              Failed to create a program object.

**Errors**              None

### `gl.linkProgram(program)`

Link the program object specified by *program*.

**Parameters**      program      Specifies the program object to be linked.

**Return value**      None

**Errors**              None

## 2. Referencia API WebGL

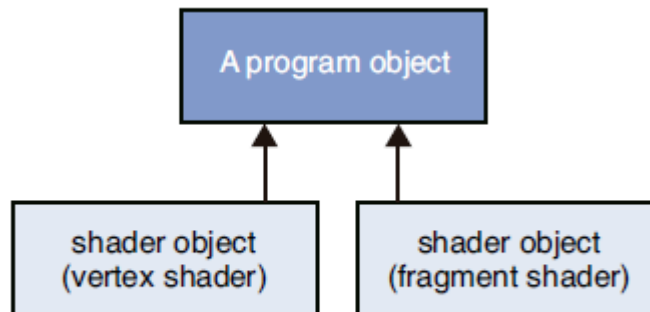
---

**gl.attachShader(program, shader)**

---

Attach the shader object specified by *shader* to the program object specified by *program*.

<b>Parameters</b>	program	Specifies the program object.
	shader	Specifies the shader object to be attached to <i>program</i> .
<b>Return value</b>	None	
<b>Errors</b>	INVALID_OPERATION	Shader had already been attached to <i>program</i> .



**Figure 9.10** The relationship between a program object and shader objects

## 2. Referencia API WebGL

### **gl.getProgramParameter(program, pname)**

Return information about *pname* for the program object specified by *program*. The return value differs depending on *pname*.

**Parameters**

<i>program</i>	Specifies the program object.
<i>pname</i>	Specifies any one of <code>gl.DELETE_STATUS</code> , <code>gl.LINK_STATUS</code> , <code>gl.VALIDATE_STATUS</code> , <code>gl.ATTACHED_SHADERS</code> , <code>gl.ACTIVE_ATTRIBUTES</code> , or <code>gl.ACTIVE_UNIFORMS</code> .

**Return value** Depending on *pname*, the following values can be returned:

<code>gl.DELETE_STATUS</code>	Whether the <i>program</i> has been deleted (true or false)
<code>gl.LINK_STATUS</code>	Whether the <i>program</i> was linked successfully (true or false)
<code>gl.VALIDATE_STATUS</code>	Whether the <i>program</i> was validated successfully (true or false) <sup>1</sup>
<code>gl.ATTACHED_SHADERS</code>	The number of attached shader objects
<code>gl.ACTIVE_ATTRIBUTES</code>	The number of attribute variables in the vertex shader
<code>gl.ACTIVE_UNIFORMS</code>	The number of uniform variables

**Errors** `INVALID_ENUM` *pname* is none of the above values.

## 2. Referencia API WebGL

### `gl.getShaderInfoLog(shader)`

Retrieve the information log from the shader object specified by *shader*.

<b>Parameters</b>	shader	Specifies the shader object from which the information log is retrieved.
<b>Return value</b>	non-null	The string containing the logged information.
	null	Any errors are generated.
<b>Errors</b>	None	

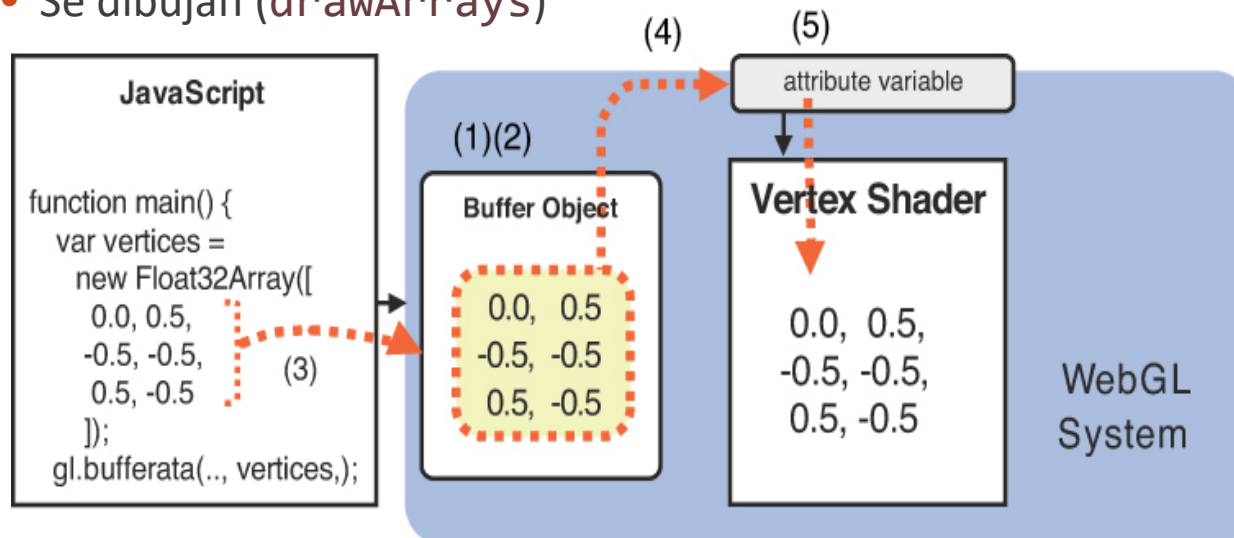
### `gl.useProgram(program)`

Tell the WebGL system that the program object specified by *program* will be used.

<b>Parameters</b>	program	Specifies the program object to be used.
<b>Return value</b>	None	
<b>Errors</b>	None	

## 2. WebGL – Cargar los datos

- Una vez se tienen inicializados los shaders, se cargan los datos en WebGL. Para ello:
  - Se crean los buffers con los datos (`gl.createBuffer()`)
  - Se conectan los buffers con los shaders (con las variables **atributes** definidas previamente en los shaders) (`bindBuffer` y `bufferData`)
  - Se dibujan (`drawArrays`)



## 2. WebGL – Cargar los datos

- Para cargar los datos, podemos crear una función que inicializa los vértices y les da color (`initVertexBuffers`)
  - Después se asigna color al canvas (`clearColor` y `clear`)
  - Finalmente se dibuja con las variables creadas y almacenadas por la función anterior en los shaders

```
function init() {
    // Get canvas object from the DOM
    // Init WebGL context
    // Init shaders

    // Write the positions of vertices to a vertex shader
    var n = initVertexBuffers(gl);
    if (n < 0) {
        console.log('Failed to set the positions of the vertices');
        return;
    }

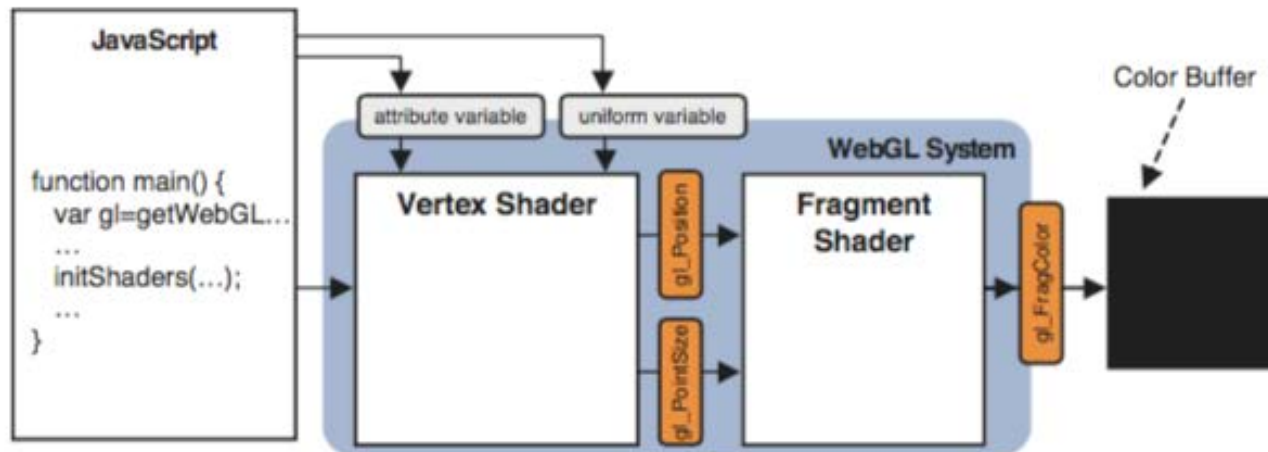
    // Clear canvas
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Draw
    gl.drawArrays(gl.TRIANGLES, 0, n);
}
```



## 2. WebGL – Cargar los datos

- Hay dos tipos de parámetros que se pueden pasar entre JavaScript y los shaders:
  - **Attribute variable:** pasa datos que difieren para cada vertex
  - **Uniform variable:** pasa datos que son los mismos para todos los vertex



## 2. WebGL – Cargar los datos Vertex

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

Storage Qualifier    Type    Variable Name

↓                      ↓                      ↓

**attribute vec4 a\_Position;**

**Figure 2.21** The declaration of the attribute variable

- Cuando se quieren utilizar estas variables, se debe preguntar al sistema WebGL el lugar de almacenamiento de la variable atributo (`getAttribLocation`)

## 2. WebGL – Cargar los datos Vertex

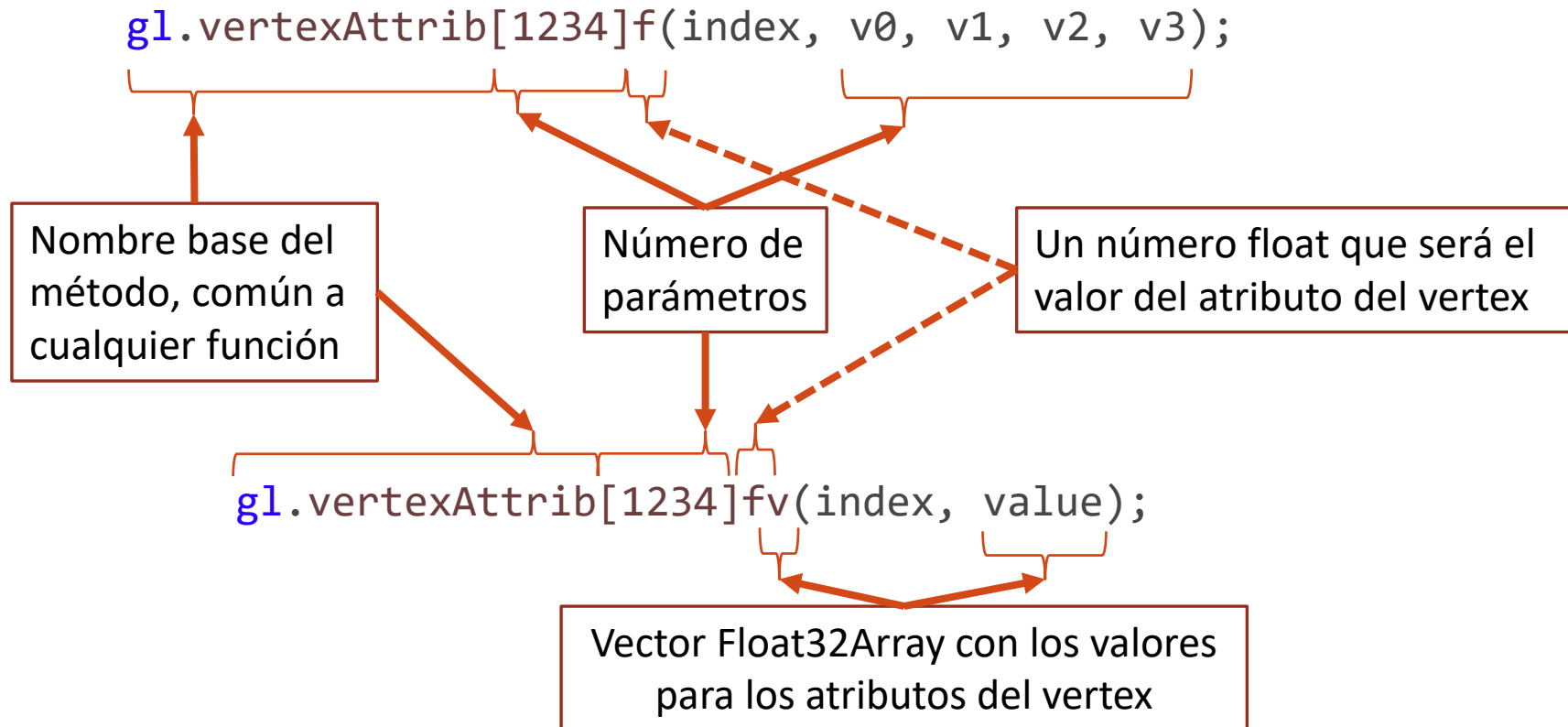
- Asignar un valor a una variable Attribute
  - Partiendo de la ubicación de la variable, podemos asignar el valor a la variable del vertex shader por medio de uno de los siguientes métodos:

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

<b>Parameters</b>	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.
<b>Return value</b>	None	
<b>Errors</b>	INVALID_VALUE	<i>location</i> is greater than or equal to the maximum number of attribute variables (8 by default).

## 2. WebGL – Cargar los datos Vertex

- Nomenclatura de las funciones:



## 2. WebGL – Cargar los datos Vertex

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

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

```
void gl.enableVertexAttribArray(index);
```

- 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

## 2. WebGL – Cargar los datos Vertex

```
function initVertexBuffers(gl) {
    // Vertices
    var dim = 3;
    var vertices = new Float32Array([
        0, 0.5, 0, // Vertex #1
        -0.5, -0.5, 0, // Vertex #2
        0.5, -0.5, 0 // Vertex #3
    ]);

    // Create a buffer object
    var vertexBuffer = gl.createBuffer();
    if (!vertexBuffer) {
        console.log('Failed to create the buffer object');
        return -1;
    }
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

    // Assign the vertices in buffer object to a_Position variable
    var gl_Position = gl.getAttribLocation(gl.program, 'gl_Position');
    if (gl_Position < 0) {
        console.log('Failed to get the storage location of gl_Position');
        return -1;
    }
    gl.vertexAttribPointer(gl_Position, dim, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(gl_Position);

    // Return number of vertices
    return vertices.length / dim;
}
```

Creamos los  
vértices, en este  
caso de un triángulo

Creamos el buffer

Conectamos los  
buffers a los shaders

Se pregunta a WebGL  
dónde están los  
datos. Esta variable  
ha sido definida en el  
código JavaScript del  
vertex shader

Se dibujan, para ello, en esta  
función se le indica qué  
variable tiene los vértices y se  
devuelve el número de  
vértices a dibujar

## 2. Referencia API WebGL

### **gl.createBuffer()**

Create a buffer object.

<b>Return value</b>	non-null	The newly created buffer object.
	null	Failed to create a buffer object.
<b>Errors</b>	None	

### **gl.deleteBuffer(buffer)**

Delete the buffer object specified by *buffer*.

<b>Parameters</b>	buffer	Specifies the buffer object to be deleted.
<b>Return Value</b>	None	
<b>Errors</b>	None	

## 2. Referencia API WebGL

### **`gl.bindBuffer(target, buffer)`**

Enable the buffer object specified by *buffer* and bind it to the *target*.

**Parameters** Target can be one of the following:

<code>gl.ARRAY_BUFFER</code>	Specifies that the buffer object contains vertex data.
<code>gl.ELEMENT_ARRAY_BUFFER</code>	Specifies that the buffer object contains index values pointing to vertex data. (See Chapter 6, “The OpenGL ES Shading Language [GLSL ES].)”)
<code>buffer</code>	Specifies the buffer object created by a previous call to <code>gl.createBuffer()</code> .  When <code>null</code> is specified, binding to the <i>target</i> is disabled.

**Return Value** None

**Errors** `INVALID_ENUM` *target* is none of the above values. In this case, the current binding is maintained.



## 2. Referencia API WebGL

### **`gl.bufferData(target, data, usage)`**

Allocate storage and write the data specified by *data* to the buffer object bound to *target*.

<b>Parameters</b>	target	Specifies <code>gl.ARRAY_BUFFER</code> or <code>gl.ELEMENT_ARRAY_BUFFER</code> .					
	data	Specifies the data to be written to the buffer object (typed array; see the next section).					
	usage	Specifies a hint about how the program is going to use the data stored in the buffer object. This hint helps WebGL optimize performance but will not stop your program from working if you get it wrong.					
		<table><tr><td><code>gl.STATIC_DRAW</code></td><td>The buffer object data will be specified once and used many times to draw shapes.</td></tr><tr><td><code>gl.STREAM_DRAW</code></td><td>The buffer object data will be specified once and used a few times to draw shapes.</td></tr><tr><td><code>gl.DYNAMIC_DRAW</code></td><td>The buffer object data will be specified repeatedly and used many times to draw shapes.</td></tr></table>	<code>gl.STATIC_DRAW</code>	The buffer object data will be specified once and used many times to draw shapes.	<code>gl.STREAM_DRAW</code>	The buffer object data will be specified once and used a few times to draw shapes.	<code>gl.DYNAMIC_DRAW</code>
<code>gl.STATIC_DRAW</code>	The buffer object data will be specified once and used many times to draw shapes.						
<code>gl.STREAM_DRAW</code>	The buffer object data will be specified once and used a few times to draw shapes.						
<code>gl.DYNAMIC_DRAW</code>	The buffer object data will be specified repeatedly and used many times to draw shapes.						
<b>Return value</b>	None						
<b>Errors</b>	<code>INVALID_ENUM</code>	<i>target</i> is none of the preceding constants					

## 2. Ref

```
gl.vertexAttribPointer(location, size, type, normalized, stride,
offset)
```

Assign the buffer object bound to `gl.ARRAY_BUFFER` to the attribute variable specified by *location*.

<b>Parameters</b>	<code>location</code>	Specifies the storage location of an attribute variable.		
	<code>size</code>	Specifies the number of components per vertex in the buffer object (valid values are 1 to 4). If size is less than the number of components required by the attribute variable, the missing components are automatically supplied just like <code>gl.vertexAttrib[1234]f()</code> .  For example, if size is 1, the second and third components will be set to 0, and the fourth component will be set to 1.		
	<code>type</code>	Specifies the data format using one of the following:		
		<code>gl.UNSIGNED_BYTE</code>	unsigned byte	for <code>Uint8Array</code>
		<code>gl.SHORT</code>	signed short integer	for <code>Int16Array</code>
		<code>gl.UNSIGNED_SHORT</code>	unsigned short integer	for <code>Uint16Array</code>
		<code>gl.INT</code>	signed integer	for <code>Int32Array</code>
		<code>gl.UNSIGNED_INT</code>	unsigned integer	for <code>Uint32Array</code>
		<code>gl.FLOAT</code>	floating point number	for <code>Float32Array</code>
	<code>normalized</code>	Either <code>true</code> or <code>false</code> to indicate whether nonfloating data should be normalized to <code>[0, 1]</code> or <code>[-1, 1]</code> .		
	<code>stride</code>	Specifies the number of bytes between different vertex data elements, or zero for default stride (see Chapter 4).		
	<code>offset</code>	Specifies the offset (in bytes) in a buffer object to indicate what number-th byte the vertex data is stored from. If the data is stored from the beginning, <i>offset</i> is 0.		

**Return value** None

<b>Errors</b>	<code>INVALID_OPERATION</code>	There is no current program object.
	<code>INVALID_VALUE</code>	<i>location</i> is greater than or equal to the maximum number of attribute variables (8, by default). <i>stride</i> or <i>offset</i> is a negative value.

## 2. Referencia API WebGL

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

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

<b>Parameters</b>	<code>program</code>	Specifies the program object that holds a vertex shader and a fragment shader.
	<code>name</code>	Specifies the name of the attribute variable whose location is to be retrieved.
<b>Return value</b>	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> .

### `gl.enableVertexAttribArray(location)`

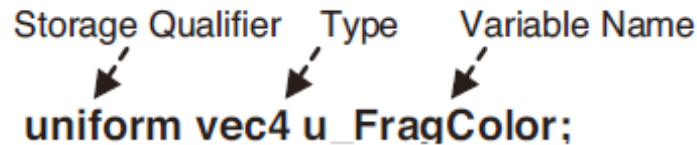
Enable the assignment of a buffer object to the attribute variable specified by *location*.

<b>Parameters</b>	<code>location</code>	Specifies the storage location of an attribute variable.
<b>Return value</b>	None	
<b>Errors</b>	<code>INVALID_VALUE</code>	<i>location</i> is greater than or equal to the maximum number of attribute variables (8 by default).

## 2. WebGL – Cargar los datos Fragment

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

Storage Qualifier    Type    Variable Name



```
uniform vec4 u_FragColor;
```

**Figure 2.31** The declaration of the uniform variable

- Cuando se quieren utilizar estas variables, se debe preguntar al sistema WebGL el lugar de almacenamiento de la variable atributo (`getUniformLocation`)

## 2. WebGL – Cargar los datos Fragment

```
function initVertexBuffers(gl) {
    // Vertices
    // Create a buffer object (same as triangle example)
    // ...

    // Fragment color
    var rgba = [0.0, 1, 0.0, 1.0];

    // Assign the color to u_FragColor variable
    var gl_FragColor = gl.getUniformLocation(gl.program, 'gl_FragColor');
    if (gl_FragColor < 0) {
        console.log('Failed to get the storage location of gl_FragColor');
        return -1;
    }
    gl.uniform4fv(gl_FragColor, rgba);

    // Return number of vertices
    return vertices.length / dim;
}
```

Creamos la  
variable que  
contiene el color

Se pregunta a  
WebGL dónde están  
los datos. Esta  
variable ha sido  
definida en el  
código JavaScript  
del fragment shader

Se asigna a la  
variable el valor

## 2. Referencia API WebGL

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

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

<b>Parameters</b>	<code>program</code>	Specifies the program object that holds a vertex shader and a fragment shader.
	<code>name</code>	Specifies the name of the uniform variable whose location is to be retrieved.
<b>Return value</b>	<code>non-null</code>	The location of the specified uniform variable.
	<code>null</code>	The specified uniform variable does not exist or its name starts with the reserved prefix <code>gl_</code> or <code>webgl_</code> .
<b>Errors</b>	<code>INVALID_OPERATION</code>	<i>program</i> has not been successfully linked (See Chapter 9.)
	<code>INVALID_VALUE</code>	The length of <i>name</i> is more than the maximum length (256 by default) of a uniform variable.

## 2. WebGL – Cargar los datos Fragment

- 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)
```

<b>Parameters</b>	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.
<b>Return value</b>	None	
<b>Errors</b>	INVALID_OPERATION	There is no current program object.
		<i>location</i> is an invalid uniform variable location.

## 2. Referencia API WebGL

**`gl.uniform4f(location, v0, v1, v2, v3)`**

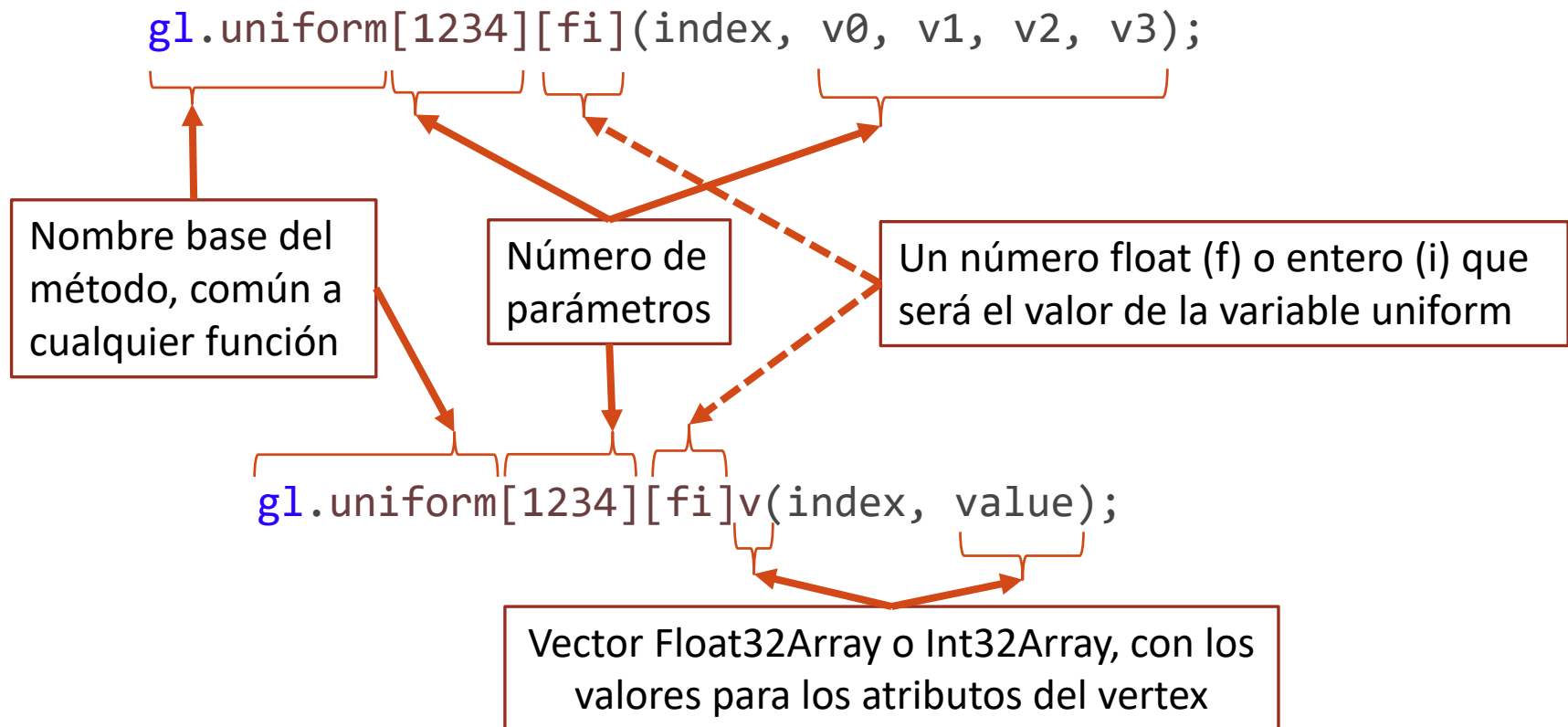
Assign the data specified by *v0*, *v1*, *v2*, and *v3* to the uniform variable specified by *location*.

<b>Parameters</b>	<i>location</i>	Specifies the storage location of a uniform variable to be modified.
	<i>v0</i>	Specifies the value to be used as the first element of the uniform variable.
	<i>v1</i>	Specifies the value to be used as the second element of the uniform variable.
	<i>v2</i>	Specifies the value to be used as the third element of the uniform variable.
	<i>v3</i>	Specifies the value to be used as the fourth element of the uniform variable.
<b>Return value</b>	None	
<b>Errors</b>	INVALID_OPERATION	There is no current program object.
		<i>location</i> is an invalid uniform variable location.



## 2. Referencia API WebGL

- Nomenclatura de las funciones:



## 2. WebGL – Cargar los datos

---

- 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`

## 2. Referencia API WebGL

```
gl.drawArrays(mode, first, count)
```

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

<b>Parameters</b>	<b>mode</b>	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> .
	<b>first</b>	Specifies which vertex to start drawing from (integer).
	<b>count</b>	Specifies the number of vertices to be used (integer).
<b>Return value</b>	None	
<b>Errors</b>	<code>INVALID_ENUM</code> <i>mode</i> is none of the preceding values.	
	<code>INVALID_VALUE</code> <i>first</i> is negative or <i>count</i> is negative.	

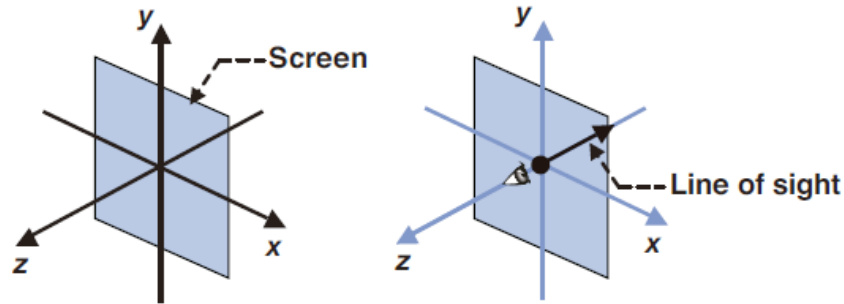
## 2. Referencia API WebGL

### **`gl.drawElements(mode, count, type, offset)`**

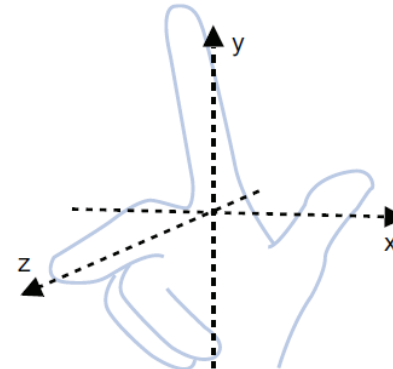
Executes the shader and draws the geometric shape in the specified *mode* using the indices specified in the buffer object bound to `gl.ELEMENT_ARRAY_BUFFER`.

<b>Parameters</b>	<b>mode</b>	Specifies the type of shape to be drawn (refer to Figure 3.17).  The following symbolic constants are accepted:  <code>gl.POINTS</code> , <code>gl.LINE_STRIP</code> , <code>gl.LINE_LOOP</code> , <code>gl.LINES</code> , <code>gl.TRIANGLE_STRIP</code> , <code>gl.TRIANGLE_FAN</code> , or <code>gl.TRIANGLES</code>
	<b>count</b>	Number of indices to be drawn (integer).
	<b>type</b>	Specifies the index data type: <code>gl.UNSIGNED_BYTE</code> or <code>gl.UNSIGNED_SHORT</code> <sup>5</sup>
	<b>offset</b>	Specifies the offset in bytes in the index array where you want to start rendering.
<b>Return value</b>	None	
<b>Errors</b>	<code>INVALID_ENUM</code>	<i>mode</i> is none of the preceding values.
	<code>INVALID_VALUE</code>	A negative value is specified for <i>count</i> or <i>offset</i>

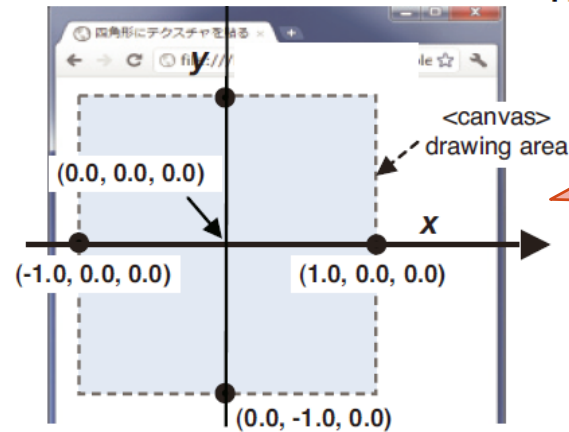
## 2. WebGL – Sistemas de coordenadas



**Figure 2.16** WebGL coordinate system



**Figure 2.17** The right-handed coordinate system



**Figure 2.18** The <canvas> drawing area and WebGL coordinate system

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

# Índice de contenidos

---

1. Visión general de WebGL
2. WebGL
- 3. Ejemplo: colorear un canvas**
4. Ejemplo: dibujar un punto
5. Ejemplo: dibujar un triángulo
6. Ejemplo: dibujar un rectángulo
7. Ejemplo: degradado de color en triángulo
8. Ejemplo: degradado de color en rectángulo
9. Resumen

### 3. Ejemplo: colorear un canvas

---

- Este ejemplo es el más sencillo posible a realizarse con WebGL
- Se pinta toda la superficie de un canvas de un color usando la API de WebGL
- El código fuente de todos los ejemplos está disponible en GitHub, en el repositorio <https://github.com/jmguerrero/GyV3D>



## 3. Ejemplo: colorear un canvas

```
<!DOCTYPE html>
<html>

<head>
  <title>Clear canvas</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

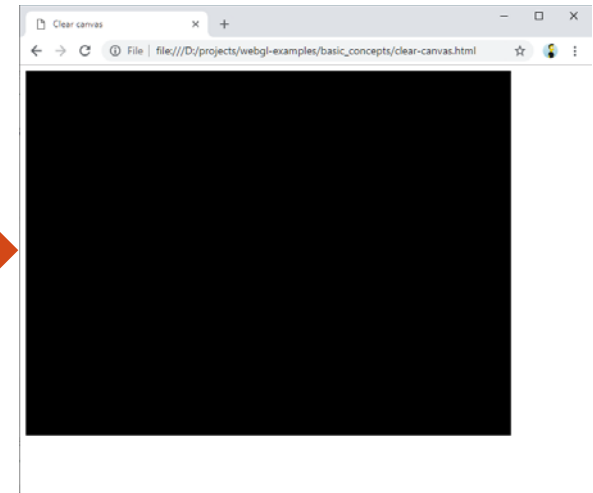
<script>
  function init() {
    // Get canvas object from the DOM
    var canvas = document.getElementById("myCanvas");

    // Get the rendering context for WebGL
    var gl = canvas.getContext("webgl");
    if (!gl) {
      console.log("Failed to get the rendering context for WebGL");
      return;
    }

    // Set clear color
    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    // Clear canvas
    gl.clear(gl.COLOR_BUFFER_BIT);
  }
</script>

</html>
```





## 3. Ejemplo: colorear un canvas

```
<!DOCTYPE html>
<html>

<head>
  <title>Clear canvas</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script>
  function init() {
    // Get canvas object from the DOM
    var canvas = document.getElementById("myCanvas");

    // Get the rendering context for WebGL
    var gl = canvas.getContext("webgl");
    if (!gl) {
      console.log("Failed to get the rendering context for WebGL");
      return;
    }

    // Set clear color
    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    // Clear canvas
    gl.clear(gl.COLOR_BUFFER_BIT);
  }
</script>

</html>
```

Nuestra página web contiene únicamente un elemento `<canvas>`

En código JavaScript hemos implementado la función `init()` que se ejecuta al cargarse el cuerpo de la página web (`onload="init()"`)

## 3. Ejemplo: colorear un canvas

```
<!DOCTYPE html>
<html>

<head>
  <title>Clear canvas</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script>
  function init() {
    // Get canvas object from the DOM
    var canvas = document.getElementById("myCanvas");

    // Get the rendering context for WebGL
    var gl = canvas.getContext("webgl");
    if (!gl) {
      console.log("Failed to get the rendering context for WebGL");
      return;
    }

    // Set clear color
    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    // Clear canvas
    gl.clear(gl.COLOR_BUFFER_BIT);
  }
</script>

</html>
```

En la función `init()` en primer lugar se lee el objeto canvas del DOM

Después, obtenemos el contexto WebGL, que nos da acceso a toda la API JavaScript de WebGL

### 3. Ejemplo: colorear un canvas

```
<!DOCTYPE html>
<html>

<head>
  <title>Clear canvas</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script>
  function init() {
    // Get canvas object from the DOM
    var canvas = document.getElementById("myCanvas");

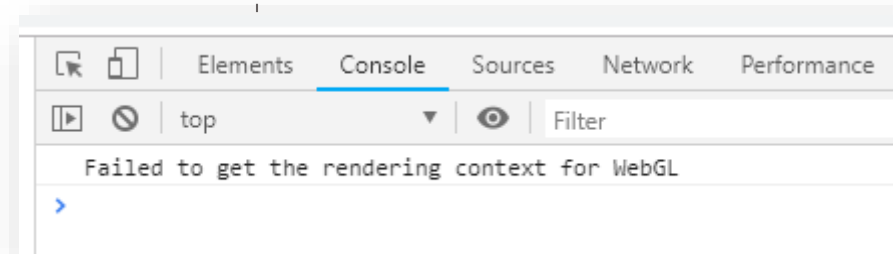
    // Get the rendering context for WebGL
    var gl = canvas.getContext("webgl");
    if (!gl) {
      console.log("Failed to get the rendering context for WebGL");
      return;
    }

    // Set clear color
    gl.clearColor(0.0, 0.0, 0.0, 1.0);

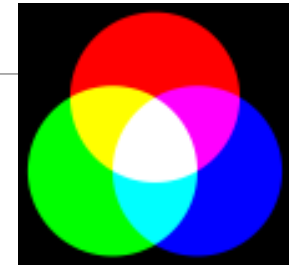
    // Clear canvas
    gl.clear(gl.COLOR_BUFFER_BIT);
  }
</script>

</html>
```

Mediante esta condición, estamos verificando que la variable `gl` tiene valor. Si WebGL no estuviese soportado, se mostraría un mensaje de error por la consola del navegador `console.log("...");`



### 3. Ejemplo: colorear un canvas



```
<!DOCTYPE html>
<html>

<head>
  <title>Clear canvas</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script>
  function init() {
    // Get canvas object from the DOM
    var canvas = document.getElementById("myCanvas");

    // Get the rendering context for WebGL
    var gl = canvas.getContext("webgl");
    if (!gl) {
      console.log("Failed to get the rendering context for WebGL");
      return;
    }

    // Set clear color
    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    // Clear canvas
    gl.clear(gl.COLOR_BUFFER_BIT);
  }
</script>

</html>
```

Mediante el método `clearColor` especificamos el color RGBA (modelo de color **aditivo** con transparencia) con el que se coloreará el canvas. En este ejemplo será negro (red=0, green=0, blue=0, alpha=0)

Mediante el método `clear` se realiza el coloreado en base al color previamente definido

# Índice de contenidos

---

1. Visión general de WebGL
2. WebGL
3. Ejemplo: colorear un canvas
- 4. Ejemplo: dibujar un punto**
5. Ejemplo: dibujar un triángulo
6. Ejemplo: dibujar un rectángulo
7. Ejemplo: degradado de color en triángulo
8. Ejemplo: degradado de color en rectángulo
9. Resumen

## 4. Ejemplo: dibujar un punto

```
<!DOCTYPE html>
<html>

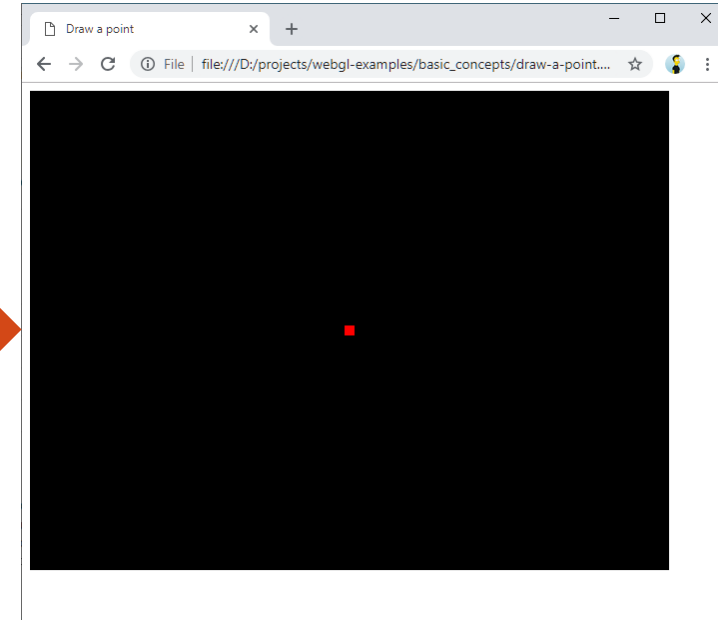
<head>
  <title>Draw a point</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  void main() {
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  }
</script>

<script>
  function init() {
    // ...
  }
</script>
</html>
```



## 4. Ejemplo: dibujar un punto

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a point</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  void main() {
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  }
</script>

<script>
  function init() {
    // ...
  }
</script>
</html>
```

En este ejemplo ya estamos implementando **shaders** usando GLSL ES. Estos shaders deben contener un método `main()` que hace de función principal del shader

En este `<script>` se define un vértice (*vertex shader*)

En este `<script>` se define un fragmento (*fragment shader*)

## 4. Ejemplo: dibujar un punto

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a point</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  void main() {
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```

Mediante la variable `gl_Position` de GLSL se define la posición del vértice actual en **coordenadas homogéneas**. En este ejemplo, la posición del estará en las coordenadas  $x=0$ ,  $y=0$ ,  $z=0$  (o sea, en el centro del canvas). El cuarto parámetro ( $w=1$  en este ejemplo) permite realizar transformaciones del vértice

Mediante la variable `gl_PointSize` de GLSL se define el tamaño en pixeles del vértice, en este ejemplo vale 10



## 4. Ejemplo: dibujar un punto

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a point</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  void main() {
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```

Mediante la variable `gl_FragColor` de GLSL se define el color del fragmento actual en formato RGBA. En este ejemplo corresponde al color rojo sólido (R=1, G=0, B=0, A=1)

## 4. Ejemplo: dibujar un punto

Desde JavaScript vamos a asociar el canvas con la ejecución de los shaders (WebGL)

```
<script>
function init() {
  // Get canvas object from the DOM
  var canvas = document.getElementById("myCanvas");

  // Init WebGL context
  var gl = canvas.getContext("webgl");
  if (!gl) {
    console.log("Failed to get the rendering context for WebGL");
    return;
  }

  // Clear canvas
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT);

  // Init shaders
  var vs = document.getElementById('shaderVs').innerHTML;
  var fs = document.getElementById('shaderFs').innerHTML;
  if (!initShaders(gl, vs, fs)) {
    console.log('Failed to initialize shaders. ');
    return;
  }

  // Draw
  gl.drawArrays(gl.POINTS, 0, 1);
}
</script>
```

La primera parte es igual que el ejemplo anterior (se colorea en negro el canvas)

Después leemos el código fuente de los shaders y lo inicializamos mediante el método `initShaders`

Por último invocamos el dibujo del vértice mediante el método `drawArrays` haciendo uso de la primitiva `gl.POINTS` (o sea, pintamos 1 punto)

## 4. Ejemplo: dibujar un punto

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

    // Create program
    var 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");
        return false;
    }

    // Use program
    gl.useProgram(glProgram);
    gl.program = glProgram;

    return true;
}

function makeShader(gl, src, type) {
    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;
    }
    return shader;
}
```

La función `makeShader` nos permite crear un objeto shader en base a su código fuente GLSL

Hay que crear un objeto de tipo program para usar los shaders (vértice y fragmento)

Para crear el objeto shader hay que compilar su código fuente y comprobar que todo ha ido correctamente

# Índice de contenidos

---

1. Visión general de WebGL
2. WebGL
3. Ejemplo: colorear un canvas
4. Ejemplo: dibujar un punto
- 5. Ejemplo: dibujar un triángulo**
6. Ejemplo: dibujar un rectángulo
7. Ejemplo: degradado de color en triángulo
8. Ejemplo: degradado de color en rectángulo
9. Resumen

## 5. Ejemplo: dibujar un triángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a triangle</title>
</head>

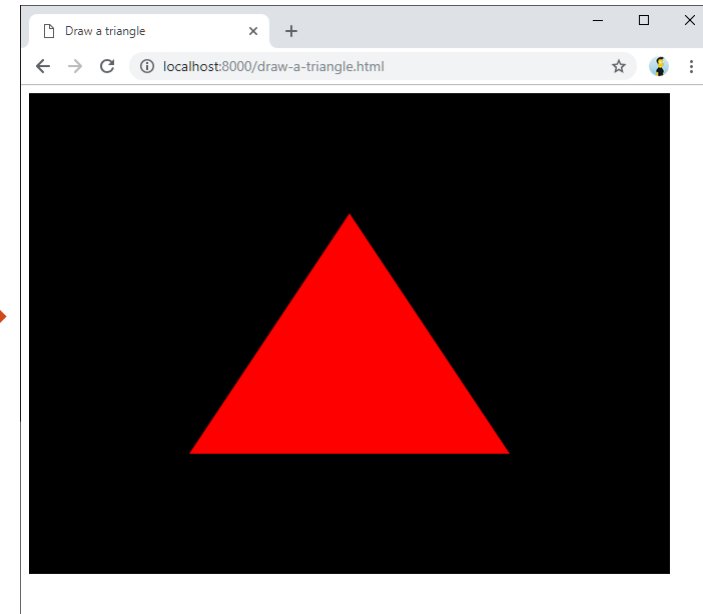
<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  void main() {
    gl_Position = a_Position;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```



## 5. Ejemplo: dibujar un triángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a triangle</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  void main() {
    gl_Position = a_Position;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```

Este ejemplo es diferente al anterior en varios aspectos:

1. La posición de los vértices (modelo) se define en JavaScript y es leída desde el vertex shader
2. Se usa la primitiva de WebGL para pintar triángulos (`gl.TRIANGLES`) en base a los vértices definidos en lugar de pintar puntos (`gl.POINTS`)

## 5. Ejemplo: dibujar un triángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a triangle</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  void main() {
    gl_Position = a_Position;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  }
</script>
```

Para cargar los vértices definidos en JavaScript desde el vertex shaders, en primer lugar hay que definir la variable `a_Position` de forma global

Dentro la función `main()` del vertex shader, se asigna el valor de `a_Position` a la variable GLSL `gl_Position`, que como hemos visto antes, define la posición de los vértices

El fragment shader no varía con respecto al ejemplo anterior (uso de color rojo sólido para colorear los vértices)

## 5. Ejemplo: dibujar un triángulo

```
function init() {
    // Get canvas object from the DOM
    var canvas = document.getElementById("myCanvas");

    // Init WebGL context
    var gl = canvas.getContext("webgl");
    if (!gl) {
        console.log("Failed to get the rendering context for WebGL");
        return;
    }

    // Init shaders
    var vs = document.getElementById('shaderVs').innerHTML;
    var fs = document.getElementById('shaderFs').innerHTML;
    if (!initShaders(gl, vs, fs)) {
        console.log('Failed to initialize shaders. ');
        return;
    }

    // Write the positions of vertices to a vertex shader
    var n = initVertexBuffers(gl);
    if (n < 0) {
        console.log('Failed to set the positions of the vertices');
        return;
    }

    // Clear canvas
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Draw
    gl.drawArrays(gl.TRIANGLES, 0, n);
}
```

En la función JavaScript `init()` se hace una llamada a la función `initVertexBuffers` donde se inicializarán los datos de los vértices

Al final se invoca el método `drawArrays` para que se genere el gráfico a partir de los vértices usando la primitiva `gl.TRIANGLES`



## 5. Ejemplo: dibujar un triángulo

```
function initVertexBuffers(gl) {
  // Vertices
  var dim = 3;
  var vertices = new Float32Array([
    0, 0.5, 0, // Vertex #1
    -0.5, -0.5, 0, // Vertex #2
    0.5, -0.5, 0 // Vertex #3
  ]);

  // Create a buffer object
  var vertexBuffer = gl.createBuffer();
  if (!vertexBuffer) {
    console.log('Failed to create the buffer object');
    return -1;
  }
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

  // Assign the vertices in buffer object to a_Position variable
  var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
  if (a_Position < 0) {
    console.log('Failed to get the storage location of a_Position');
    return -1;
  }
  gl.vertexAttribPointer(a_Position, dim, gl.FLOAT, false, 0, 0);
  gl.enableVertexAttribArray(a_Position);

  // Return number of vertices
  return vertices.length / dim;
}
```

En este ejemplo definimos **tres puntos** dadas 3 componentes

Creamos un buffer, que es una memoria interna de WebGL

Asociamos los vértices definidos al buffer

Usamos la variable **a\_Position** definida como atributo global en el vertex shader para cargar los vértices que están en el buffer

Devolvemos el número de vértices en función del número de elementos del array y la dimensión

# Índice de contenidos

---

1. Visión general de WebGL
2. WebGL
3. Ejemplo: colorear un canvas
4. Ejemplo: dibujar un punto
5. Ejemplo: dibujar un triángulo
- 6. Ejemplo: dibujar un rectángulo**
7. Ejemplo: degradado de color en triángulo
8. Ejemplo: degradado de color en rectángulo
9. Resumen

## 6. Ejemplo: dibujar un rectángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a rectangle</title>
</head>

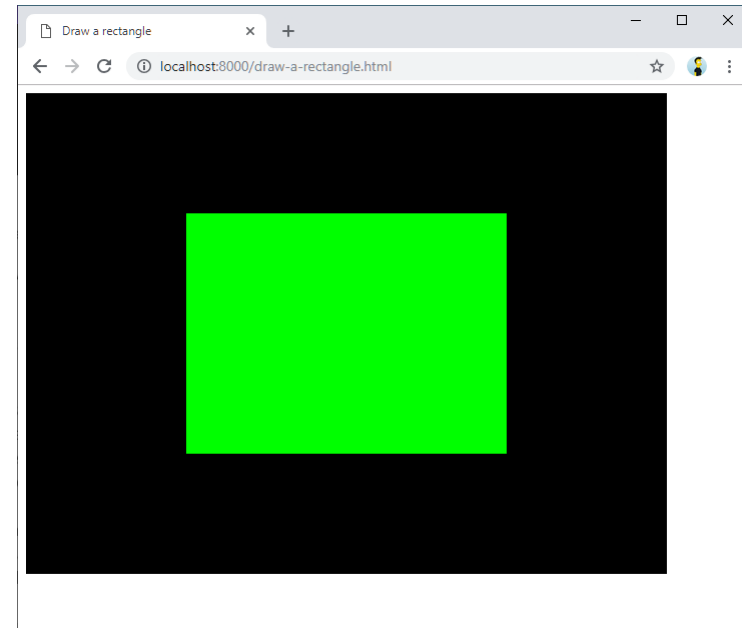
<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  void main() {
    gl_Position = a_Position;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  precision mediump float;
  uniform vec4 u_FragColor;
  void main() {
    gl_FragColor = u_FragColor;
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```



## 6. Ejemplo: dibujar un rectángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a rectangle</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  void main() {
    gl_Position = a_Position;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  precision mediump float;
  uniform vec4 u_FragColor;
  void main() {
    gl_FragColor = u_FragColor;
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```

Este ejemplo es diferente al anterior en varios aspectos:

1. El color del fragmento se define en JavaScript y es leído desde el fragment shader
2. Dado que no existe la primitiva WebGL para pintar rectángulos, usamos dos triángulos (`gl.TRIANGLES`)

## 6. Ejemplo: dibujar un rectángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a rectangle</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  void main() {
    gl_Position = a_Position;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  precision mediump float;
  uniform vec4 u_FragColor;
  void main() {
    gl_FragColor = u_FragColor;
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```

El vertex shader no varía con respecto al ejemplo anterior (se cargan los vértices desde la variable `a_Position`, que se cargará mediante JavaScript)

Para cargar los vértices definidos en JavaScript desde el fragment shader, en primer lugar hay que definir una variable global (de nombre `u_FragColor` en ese ejemplo) usando el calificador `uniform`.

Además, es necesario definir la precisión que la GPU requería para el calculo de valores en coma flotante (*float*). Eso se hace mediante la sentencia `precision mediump float;`

## 6. Ejemplo: dibujar un rectángulo

```
function initVertexBuffers(gl) {
    // Vertices
    var dim = 2;
    var vertices = new Float32Array([
        -0.5, 0.5, 0.5, 0.5, 0.5, -0.5, // Triangle 1
        -0.5, 0.5, 0.5, -0.5, -0.5, -0.5 // Triangle 2
    ]);

    // Fragment color
    var rgba = [0.0, 1, 0.0, 1.0];

    // Create a buffer object (same as triangle example)
    // ...

    // Assign the color to u_FragColor variable
    var u_FragColor = gl.getUniformLocation(gl.program, 'u_FragColor');
    if (u_FragColor < 0) {
        console.log('Failed to get the storage location of u_FragColor');
        return -1;
    }
    gl.uniform4fv(u_FragColor, rgba);

    // Return number of vertices
    return vertices.length / dim;
}
```

En este ejemplo definimos 2 triángulos (formado por 3 puntos cada uno) dadas 2 componentes

La creación del buffer es igual que en el ejemplo anterior (se omite en este fragmento, el código completo está en GitHub)

Usamos el método `uniform4fv` para asignar el valor de la variable JavaScript `rgba` al variable del fragment shader llamada `u_FragColor`

# Índice de contenidos

---

1. Visión general de WebGL
2. WebGL
3. Ejemplo: colorear un canvas
4. Ejemplo: dibujar un punto
5. Ejemplo: dibujar un triángulo
6. Ejemplo: dibujar un rectángulo
- 7. Ejemplo: degradado de color en triángulo**
8. Ejemplo: degradado de color en rectángulo
9. Resumen

## 7. Ejemplo: degradado de color en triángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a colored triangle</title>
</head>

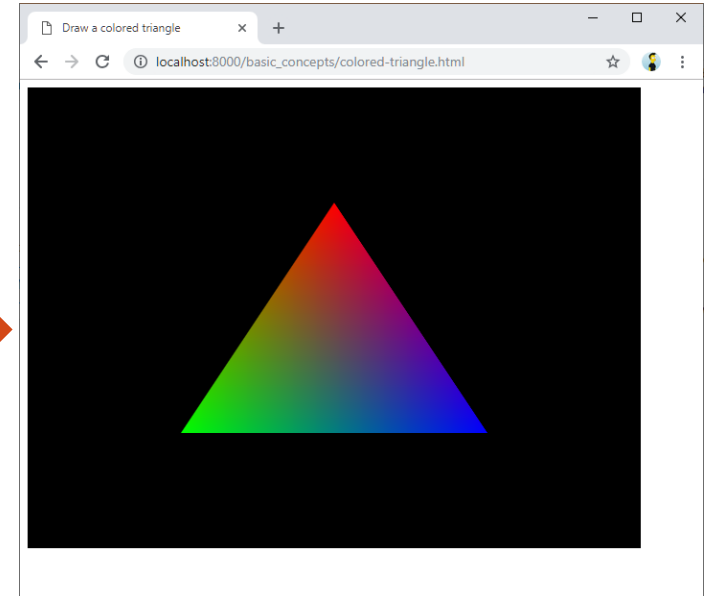
<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  attribute vec4 a_Color;
  varying highp vec4 v_Color;
  void main() {
    gl_Position = a_Position;
    v_Color = a_Color;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  varying highp vec4 v_Color;
  void main() {
    gl_FragColor = v_Color;
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```





## 7. Ejemplo: degradado de color en triángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a colored triangle</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  attribute vec4 a_Color;
  varying highp vec4 v_Color;
  void main() {
    gl_Position = a_Position;
    v_Color = a_Color;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  varying highp vec4 v_Color;
  void main() {
    gl_FragColor = v_Color;
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```

Este ejemplo, el color de los fragmentos es variable:

1. Se define como una variable (**attribute**), y es procesado en el vertex
2. Por cada fragmento (pixel) se obtiene un valor que es comunicado al fragment shader mediante una variable **varying**

## 7. Ejemplo: degradado de color en triángulo

```
function initVertexBuffers(gl) {
    // Vertices
    var dim = 3;
    var vertices = new Float32Array([
        0, 0.5, 0, // Vertex #1
        -0.5, -0.5, 0, // Vertex #2
        0.5, -0.5, 0 // Vertex #3
    ]);
    var colors = new Float32Array([
        1.0, 0.0, 0.0, // Color #1 (red)
        0.0, 1.0, 0.0, // Color #2 (green)
        0.0, 0.0, 1.0, // Color #3 (blue)
    ]);

    // Create a buffer object for vertices and assign to a_Position variable
    var vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
    var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
    gl.vertexAttribPointer(a_Position, dim, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(a_Position);

    // Create colors buffer
    var trianglesColorBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, trianglesColorBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, colors, gl.STATIC_DRAW);
    var a_Color = gl.getAttribLocation(gl.program, 'a_Color');
    gl.vertexAttribPointer(a_Color, dim, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(a_Color);

    // Return number of vertices
    return vertices.length / dim;
}
```

En JavaScript, tanto los vértices como los colores son comunicados a las variables **attribute** del vertex shader usando el mecanismo habitual

# Índice de contenidos

---

1. Visión general de WebGL
2. WebGL
3. Ejemplo: colorear un canvas
4. Ejemplo: dibujar un punto
5. Ejemplo: dibujar un triángulo
6. Ejemplo: dibujar un rectángulo
7. Ejemplo: degradado de color en triángulo
- 8. Ejemplo: degradado de color en rectángulo**
9. Resumen

## 8. Ejemplo: degradado de color en rectángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a colored rectangle</title>
</head>

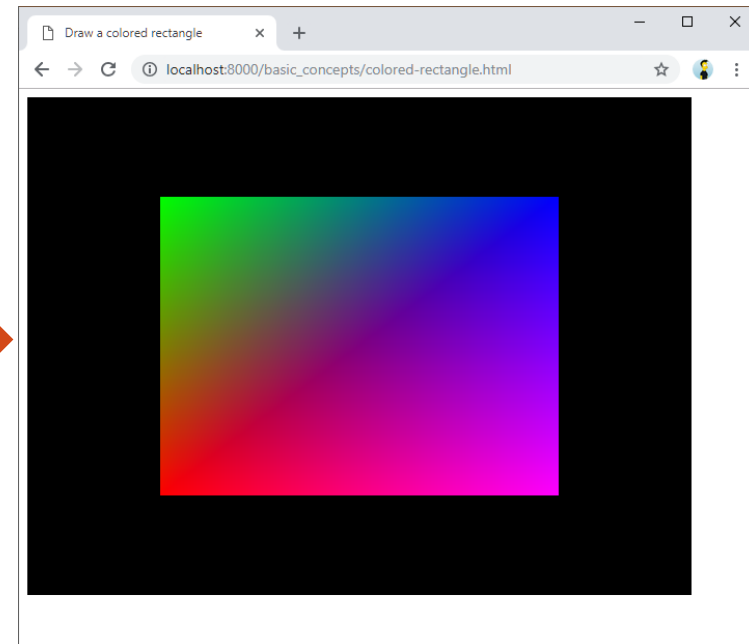
<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  attribute vec4 a_Color;
  varying highp vec4 v_Color;
  void main() {
    gl_Position = a_Position;
    v_Color = a_Color;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  varying highp vec4 v_Color;
  void main() {
    gl_FragColor = v_Color;
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```



## 8. Ejemplo: degradado de color en rectángulo

```
<!DOCTYPE html>
<html>

<head>
  <title>Draw a colored rectangle</title>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>

<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  attribute vec4 a_Color;
  varying highp vec4 v_Color;
  void main() {
    gl_Position = a_Position;
    v_Color = a_Color;
  }
</script>

<script id="shaderFs" type="x-shader/x-fragment">
  varying highp vec4 v_Color;
  void main() {
    gl_FragColor = v_Color;
  }
</script>

<script>
  function init() {
    // ...
  }
</script>

</html>
```

La definición de los shaders es exactamente igual en este ejemplo que en el anterior (uso de **attribute** para los vértices y colores en el vertex y **varying** para el color variable en el fragment)

## 8. Ejemplo: degradado de color en rectángulo

```
function initBuffers(gl) {
    // Vertices
    var dim = 3;
    var vertices = new Float32Array([
        -0.6, -0.6, 0.0, // 0
        -0.6, 0.6, 0.0, // 1
        0.6, 0.6, 0.0, // 2
        0.6, -0.6, 0.0, // 3
    ]);
    gl.bindBuffer(gl.ARRAY_BUFFER, gl.createBuffer());
    gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
    var vertexPositionAttribute = gl.getAttribLocation(gl.program, "a_Position");
    gl.enableVertexAttribArray(vertexPositionAttribute);
    gl.vertexAttribPointer(vertexPositionAttribute, dim, gl.FLOAT, false, 0, 0);

    // Colors
    var colors = new Float32Array([
        1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0,
        1.0, 0.0, 1.0,
    ]);
    gl.bindBuffer(gl.ARRAY_BUFFER, gl.createBuffer());
    gl.bufferData(gl.ARRAY_BUFFER, colors, gl.STATIC_DRAW);
    var vertexColorAttribute = gl.getAttribLocation(gl.program, "a_Color");
    gl.enableVertexAttribArray(vertexColorAttribute);
    gl.vertexAttribPointer(vertexColorAttribute, dim, gl.FLOAT, false, 0, 0);

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

    // Return number of vertices
    return indices.length;
}
```

La diferencia principal es el uso de índices para seleccionar los vértices. Estos vértices se especifican usando un buffer de tipo

**`gl.ELEMENT_ARRAY_BUFFER`**

## 8. Ejemplo: degradado de color en rectángulo

```
function init() {
    // Get canvas object from the DOM
    var canvas = document.getElementById("myCanvas");

    // Init WebGL context
    var gl = canvas.getContext("webgl");
    if (!gl) {
        console.log("Failed to get the rendering context for WebGL");
        return;
    }

    // Init shaders
    var vs = document.getElementById('shaderVs').innerHTML;
    var fs = document.getElementById('shaderFs').innerHTML;
    if (!initShaders(gl, vs, fs)) {
        console.log('Failed to initialize shaders.');
```

El método de dibujo es  
diferente en este ejemplo.  
Se usa el método  
`drawElements`

```
    // Draw
    gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_SHORT, 0);
}
```

# Índice de contenidos

---

1. Visión general de WebGL
2. WebGL
3. Ejemplo: colorear un canvas
4. Ejemplo: dibujar un punto
5. Ejemplo: dibujar un triángulo
6. Ejemplo: dibujar un rectángulo
7. Ejemplo: degradado de color en triángulo
8. Ejemplo: degradado de color en rectángulo
- 9. Resumen**



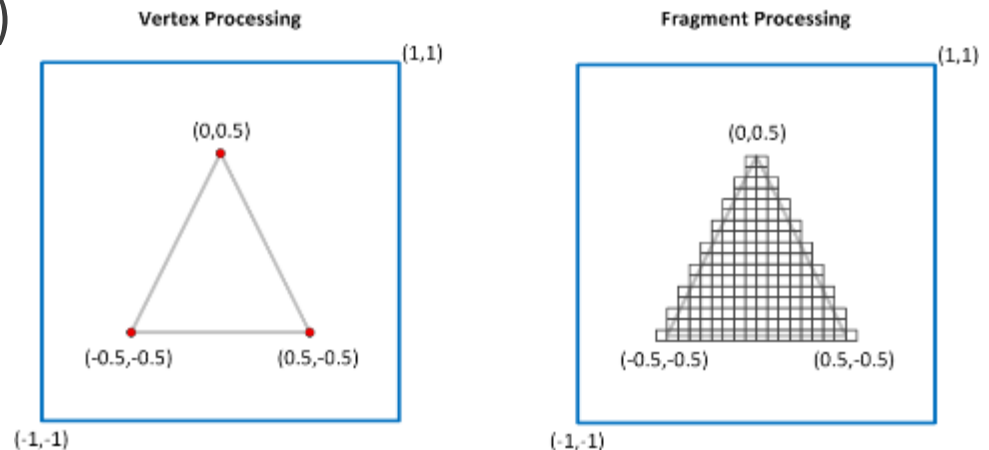
## 9. Resumen

---

- **WebGL** define una API **JavaScript** para la generación de gráficos en 3D en navegadores web
- Los gráficos WebGL se dibujan en el navegador usando la etiqueta **HTML5** `<canvas>`
- WebGL usa unos programas en lenguaje **GLSL** que se ejecutan la GPU de un ordenador llamados ***shaders***

# 9. Resumen

- Hay dos tipos de *shaders*:
  - *Vertex shader*: Procesado de los **vértices** (puntos geométricos del modelo 3D)
  - *Fragment shader*: Procesado de los **píxeles** que se renderizan en base a los vértices y la primitiva de dibujo (`gl.POINTS`, `gl.TRIANGLES`, etc)



## 9. Resumen

---

- La comunicación entre código JavaScript (API WebGL) y los shaders se realiza en base a variables globales GLSL:
  - `attribute`: Parámetros definidos por vértice. Se comunican a través de un buffer. Su valor puede variar durante la ejecución del método `draw`
  - `uniform`: Parámetros definidos por primitiva. Su valor es constante durante la ejecución del método `draw`
- La comunicación entre vertex y fragment shader se realiza usando variables globales GLSL:
  - `varying`: Parámetros definidos por fragmento. Su valor puede ser variable