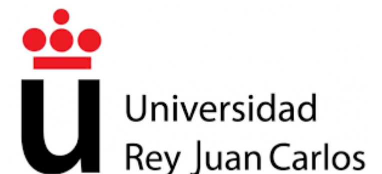
Gráficos y visualización 3D

3. Conceptos básicos en WebGL

Boni García
Web: http://bonigarcia.github.io/
Email: boni.garcia@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)



Índice de contenidos

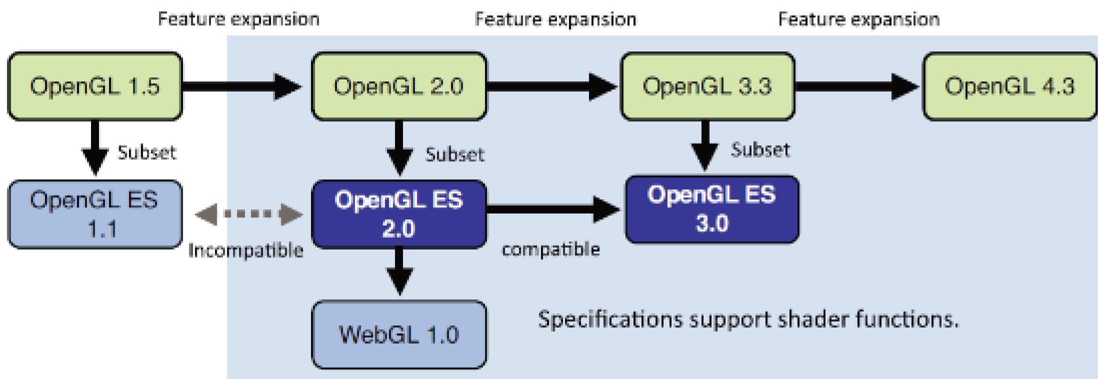
1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. Resumen

Índice de contenidos

1. Visón general de WebGL
   I. Introducción
   II. Shaders
   III. Pipeline
   IV. Aplicaciones WebGL
   V. Canvas
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. 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*)

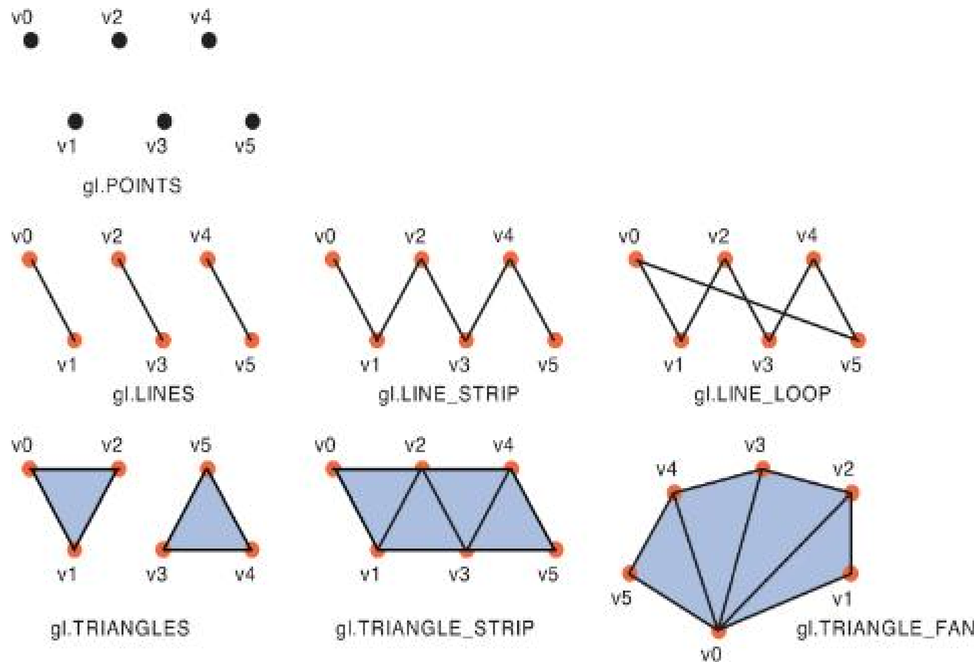


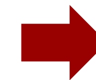
1. Visión general de WebGL - Shaders

- 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

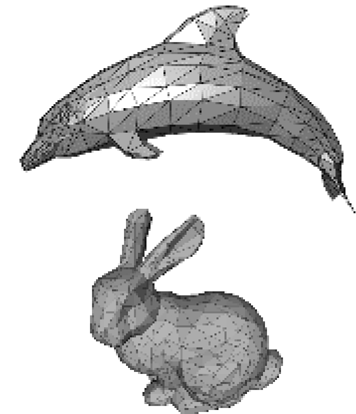
1. Visión general de WebGL - Shaders

- Los gráficos manejados en los shaders estáran 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

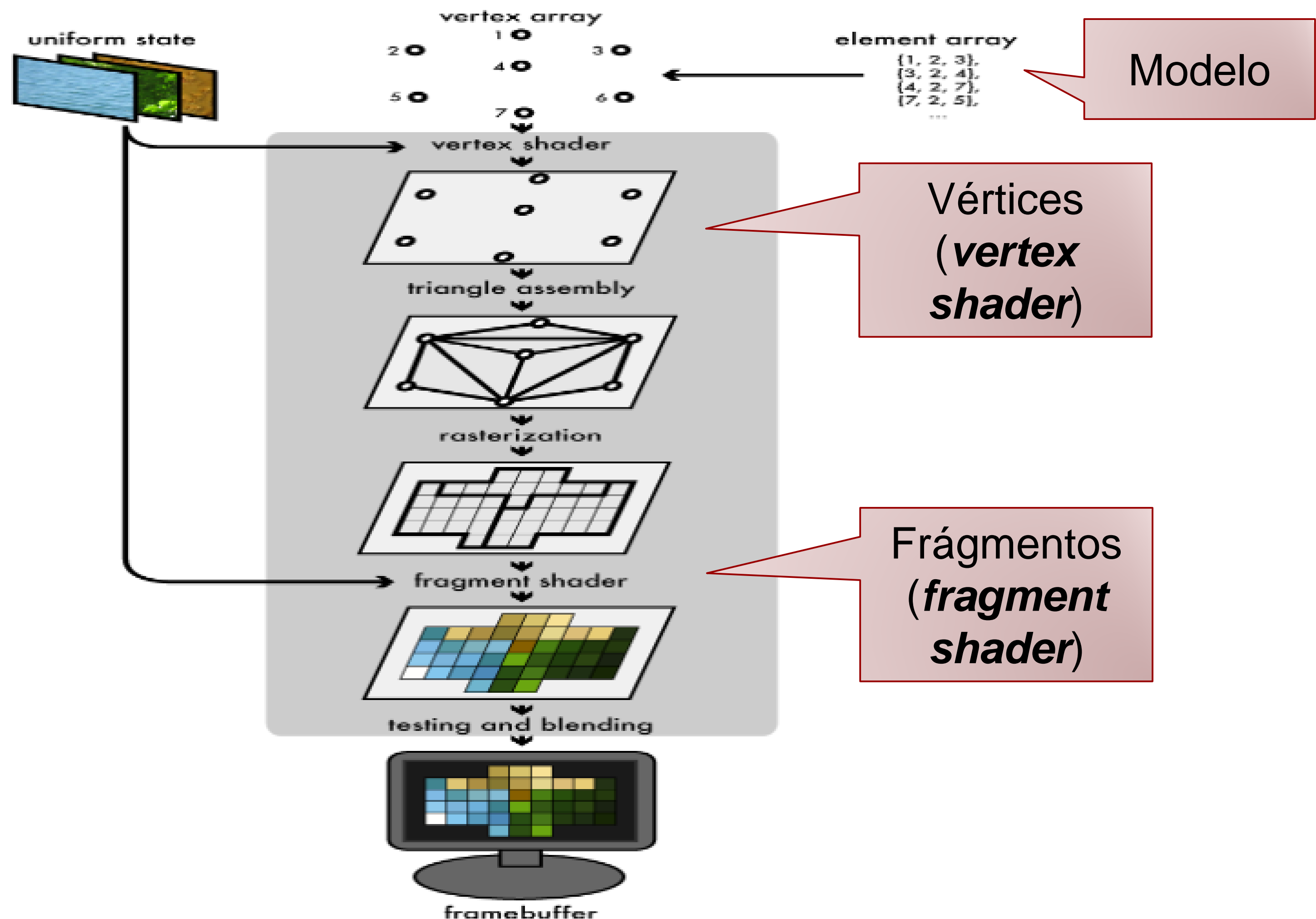
- Hay dos tipos de shaders necesarios para generar gráficos en WebGL:

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 procesado de los gráficos que componen la escena en diferentes fases (***pipeline***)

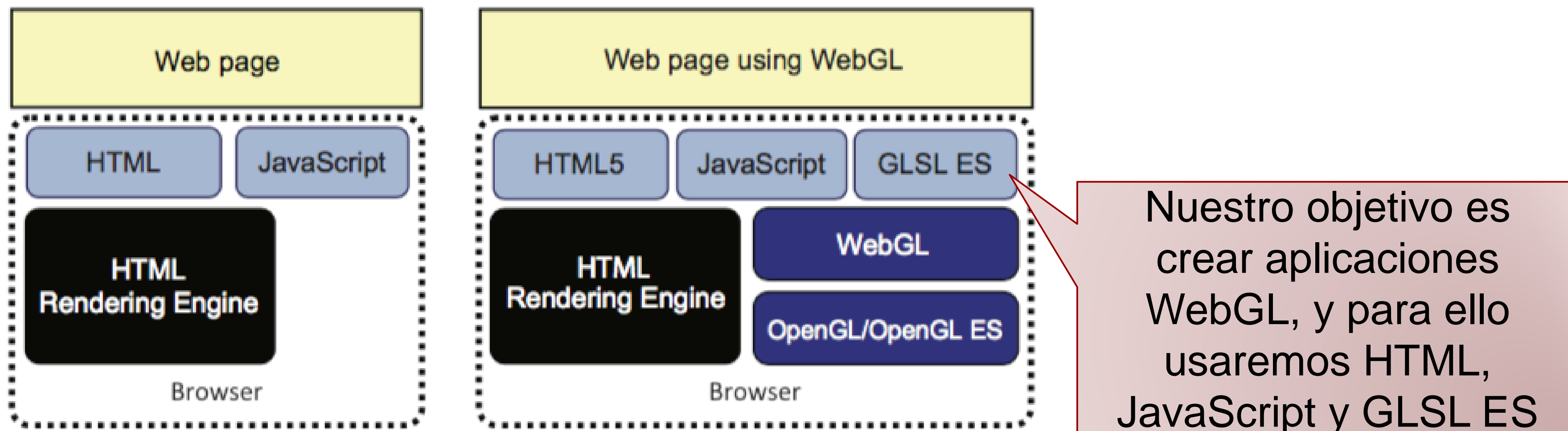
1. Visión general de WebGL - Pipeline



http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html

1. Visión general de WebGL – Aplicaciones 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*)



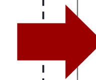
Fork me on GitHub

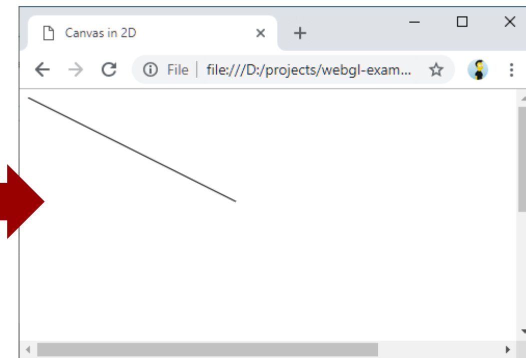
1. Visión general de WebGL – Canvas

- 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>
```





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

Índice de contenidos

1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. Resumen

2. Referencia API WebGL

```
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	
Errors[2]	None	

2. Referencia API WebGL

`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. Referencia API WebGL

`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: `gl.POINTS`, `gl.LINES`, `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, and `gl.TRIANGLE_FAN`.
	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.	

2. Referencia API WebGL

`gl.createShader(type)`

Create a shader of the specified *type*.

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

`gl.compileShader(shader)`

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

Parameters	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*.

Parameters	shader	Specifies the shader object.
	pname	Specifies the information to get from the shader: `gl.SHADER_TYPE`, `gl.DELETE_STATUS`, or `gl.COMPILE_STATUS`.
Return value	The following depending on *pname*:	
	`gl.SHADER_TYPE`	The type of shader (`gl.VERTEX_SHADER` or `gl.FRAGMENT_SHADER`)
	`gl.DELETE_STATUS`	Whether the deletion has succeeded (`true` or `false`)
	`gl.COMPILE_STATUS`	Whether the compilation has succeeded (`true` or `false`)
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*.

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

```
gl.useProgram(program)
```

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

Parameters	program	Specifies the program object to be used.
Return value	None	
Errors	None	

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*.

Parameters	program	Specifies the program object.
	shader	Specifies the shader object to be attached to *program*.
Return value	None	
Errors	INVALID_OPERATION	*Shader* had already been attached to *program*.

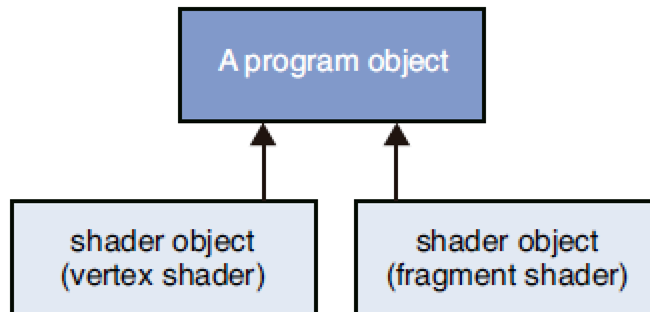


Figure 9.10 The relationship between a program object and shader objects

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

program	Specifies the program object.
pname	Specifies any one of `gl.DELETE_STATUS`, `gl.LINK_STATUS`, `gl.VALIDATE_STATUS`, `gl.ATTACHED_SHADERS`, `gl.ACTIVE_ATTRIBUTES`, or `gl.ACTIVE_UNIFORMS`.

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

`gl.DELETE_STATUS`	Whether the *program* has been deleted (`true` or `false`)
`gl.LINK_STATUS`	Whether the *program* was linked successfully (`true` or `false`)
`gl.VALIDATE_STATUS`	Whether the *program* was validated successfully (`true` or `false`)[1]
`gl.ATTACHED_SHADERS`	The number of attached shader objects
`gl.ACTIVE_ATTRIBUTES`	The number of `attribute` variables in the vertex shader
`gl.ACTIVE_UNIFORMS`	The number of `uniform` variables

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

2. Referencia API WebGL

`gl.createBuffer()`

Create a buffer object.

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

`gl.deleteBuffer(buffer)`

Delete the buffer object specified by *buffer*.

Parameters	buffer	Specifies the buffer object to be deleted.
Return Value	None	
Errors	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:	
	`gl.ARRAY_BUFFER`	Specifies that the buffer object contains vertex data.
	`gl.ELEMENT_ARRAY_BUFFER`	Specifies that the buffer object contains index values pointing to vertex data. (See Chapter 6, "The OpenGL ES Shading Language [GLSL ES].)
	buffer	Specifies the buffer object created by a previous call to `gl.createBuffer()`. When `null` is specified, binding to the *target* 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*.

Parameters	target	Specifies `gl.ARRAY_BUFFER` or `gl.ELEMENT_ARRAY_BUFFER`.	
	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.	
		`gl.STATIC_DRAW`	The buffer object data will be specified once and used many times to draw shapes.
		`gl.STREAM_DRAW`	The buffer object data will be specified once and used a few times to draw shapes.
		`gl.DYNAMIC_DRAW`	The buffer object data will be specified repeatedly and used many times to draw shapes.
Return value	None		
Errors	INVALID_ENUM	*target* is none of the preceding constants	

2. Refer

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

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

Parameters

location — Specifies the storage location of an attribute variable.

size — 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 `gl.vertexAttrib[1234]f()`.

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.

type — Specifies the data format using one of the following:

`gl.UNSIGNED_BYTE`	unsigned byte	for `Uint8Array`
`gl.SHORT`	signed short integer	for `Int16Array`
`gl.UNSIGNED_SHORT`	unsigned short integer	for `Uint16Array`
`gl.INT`	signed integer	for `Int32Array`
`gl.UNSIGNED_INT`	unsigned integer	for `Uint32Array`
`gl.FLOAT`	floating point number	for `Float32Array`

normalized — Either `true` or `false` to indicate whether nonfloating data should be normalized to [0, 1] or [–1, 1].

stride — Specifies the number of bytes between different vertex data elements, or zero for default stride (see Chapter 4).

offset — 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, *offset* is 0.

Return value None

Errors

INVALID_OPERATION — There is no current program object.

INVALID_VALUE — *location* is greater than or equal to the maximum number of attribute variables (8, by default). *stride* or *offset* 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.

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 `gl_` or `webgl_`.

```
gl.enableVertexAttribArray(location)
```

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

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

2. Referencia API WebGL

```
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.
Return value	non-null	The location of the specified uniform variable.
	null	The specified uniform variable does not exist or its name starts with the reserved prefix `gl_` or `webgl_`.
Errors	INVALID_OPERATION	*program* has not been successfully linked (See Chapter 9.)
	INVALID_VALUE	The length of *name* is more than the maximum length (256 by default) of a uniform variable.

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*.

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

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`.

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

Índice de contenidos

1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo

3. Referencia API GLSL ES

Table 2.2 Built-In Variables Available in a Vertex Shader

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

Storage Qualifier Type Variable Name

attribute vec4 a_Position;

Figure 2.21 The declaration of the attribute variable

Storage Qualifier Type Variable Name

uniform vec4 u_FragColor;

Figure 2.31 The declaration of the uniform variable

3. Referencia API GLSL ES

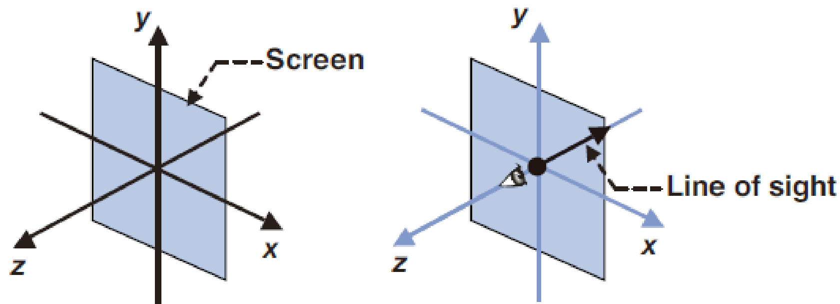


Figure 2.16 WebGL coordinate system

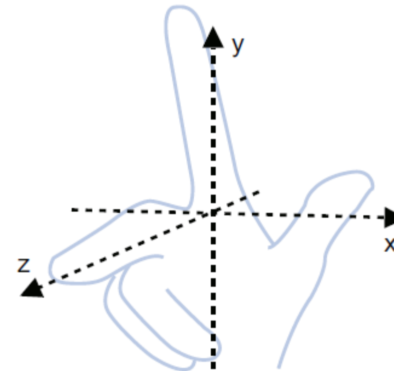


Figure 2.17 The right-handed coordinate system

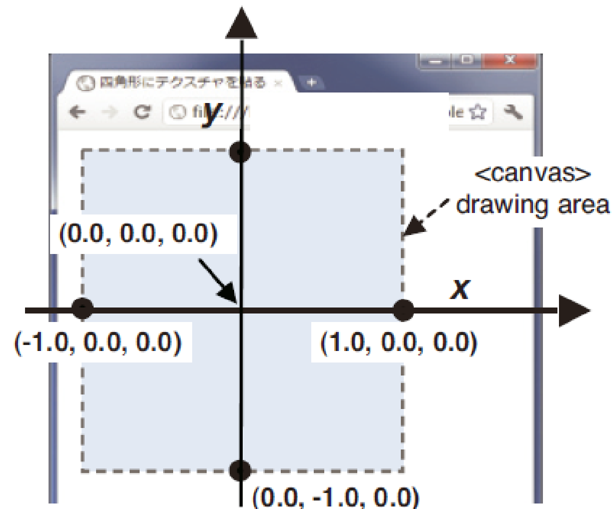


Figure 2.18 The <canvas> drawing area and WebGL coordinate system

3. Referencia API GLSL ES

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

float	float	Float	float

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

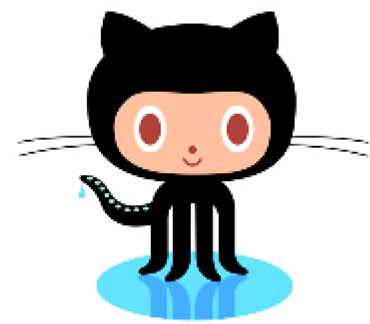
Type and Variable Name	Description
vec4 `gl_FragColor`	Specify the color of a fragment (in RGBA)

Índice de contenidos

1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. Resumen

4. Ejemplo: colorear un canvas

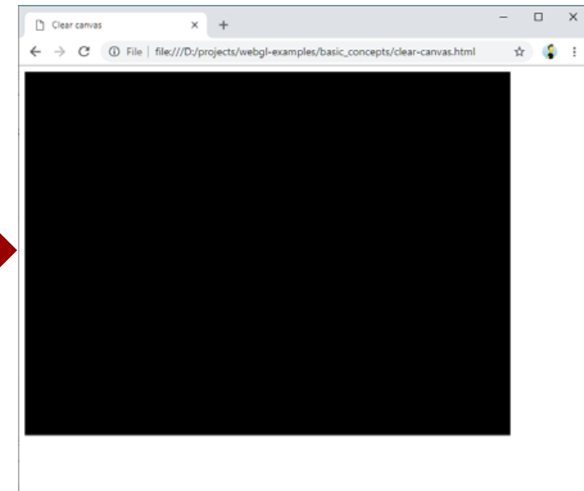
- Este ejemplo es el más sencillo posible a realizarse con WebGL
- Se pinta todo la superficie de un canvas de un color usando la API de WebGL
- El código fuente de todos los ejemplos está disponible en GitHub, en el repositorio https://github.com/bonigarcia/webgl-examples



Fork me on GitHub

4. 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>
```



Fork me on GitHub

4. 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()"`)

Fork me on GitHub

4. 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

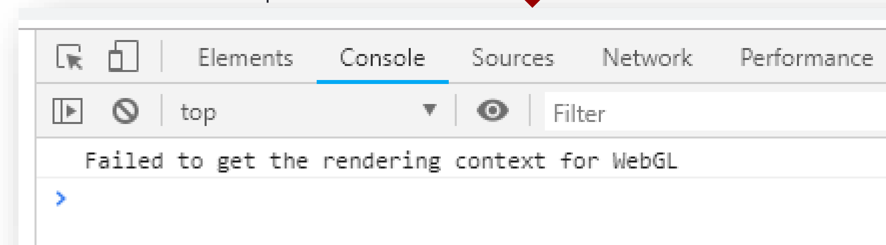
Después, obtenemos el contexto WebGL, que nos da acceso a toda la API JavaScript de WebGL

Fork me on GitHub

4. 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>
```

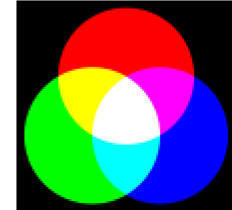
Mediante esta condición, estamos verificando que la variable `gl` tiene valor. Si WebGL no estuviese soportado, se mostraría la un mensaje de error por la consola del navegador `console.log("...");`



Fork me on GitHub

4. 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

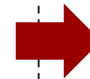
Índice de contenidos

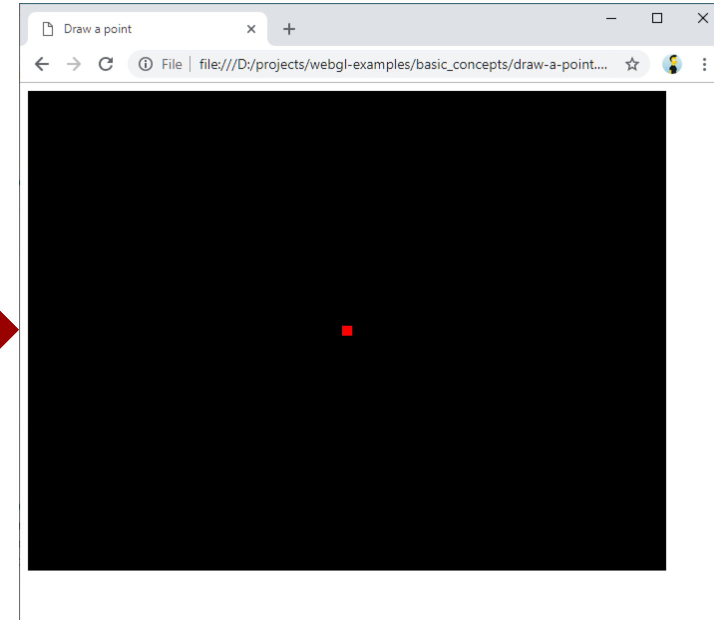
1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. Resumen

Fork me on GitHub

5. 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>
```





Fork me on GitHub

5. 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*)

Fork me on GitHub

5. 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

Fork me on GitHub

5. 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)

Fork me on GitHub

5. 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 intialize 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)

Fork me on GitHub

5. 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

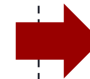
Índice de contenidos

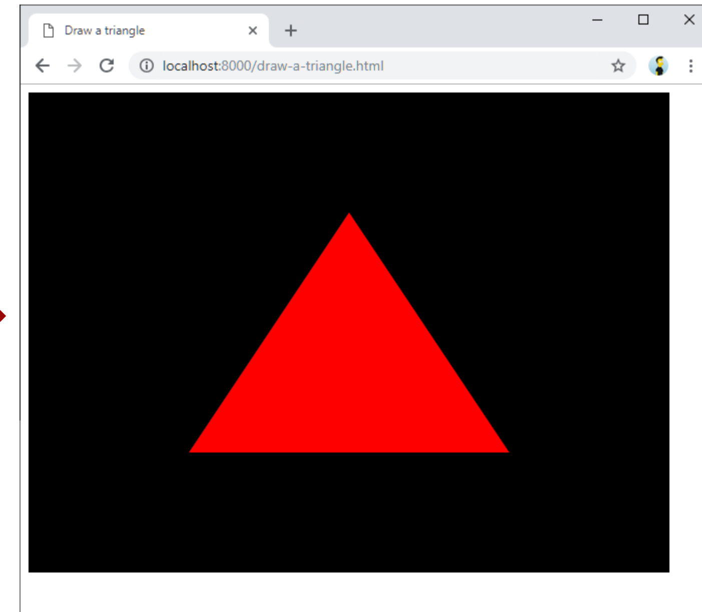
1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. Resumen

Fork me on GitHub

6. 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>
```





Fork me on GitHub

6. 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`)

Fork me on GitHub

6. 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)

Fork me on GitHub

6. 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 intialize 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`

Fork me on GitHub

6. Ejemplo: dibujar un triángulo

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

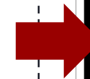
Índice de contenidos

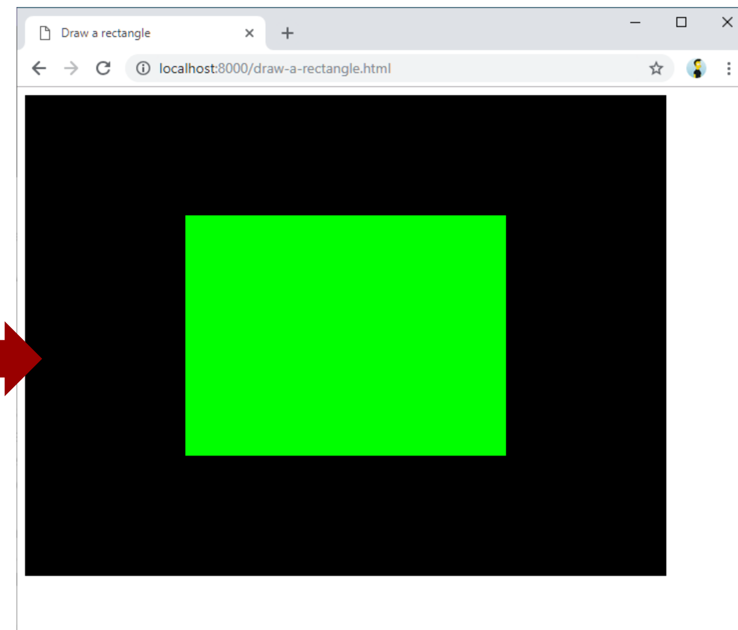
1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. Resumen

Fork me on GitHub

7. 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_Color;
    void main() {
        gl_FragColor = u_Color;
    }
</script>

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

</html>
```





Fork me on GitHub

7. 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_Color;
    void main() {
        gl_FragColor = u_Color;
    }
</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 leída desde el fragment shader
2. Dado que no existe la primitiva WebGL para pintar rectángulos, usamos dos triángulos (`gl.TRIANGLES`)

Fork me on GitHub

7. Ejemplo: dibujar un rectángulo

```
<!DOCTYPE html>
<html>

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

<body onload="init()">
    <canvas id="myCanvas" width="640" height="480"></c
</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 vertices desde la variable `a_Position` , que se cargará mediante JavaScript)

Para cargar los vértices definidos en JavaScript desde el fragment shaders, 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;`

Fork me on GitHub

7. 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`

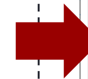
Índice de contenidos

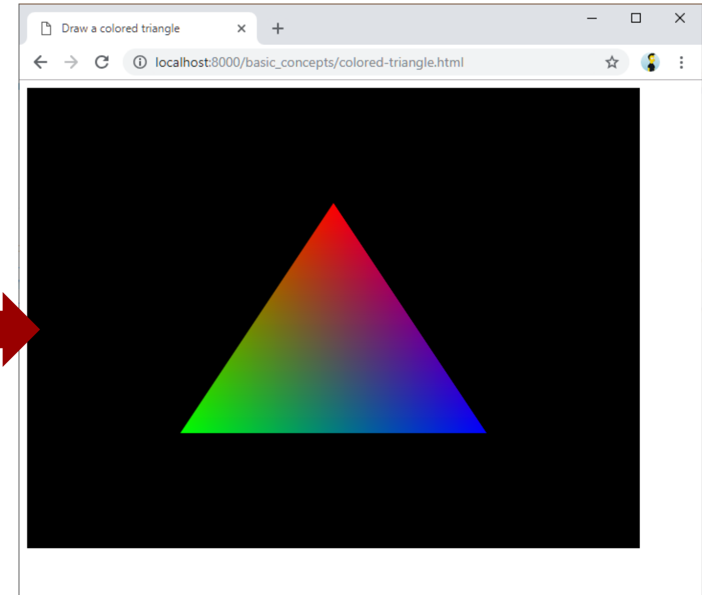
1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. Resumen

Fork me on GitHub

8. Ejemplo: degradado de color en triángul

```
<!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>
```





Fork me on GitHub

8. Ejemplo: degradado de color en triángul

```
<!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 shadre mediante una variable `varying`

8. Ejemplo: degradado de color en triángul

Fork me on GitHub

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

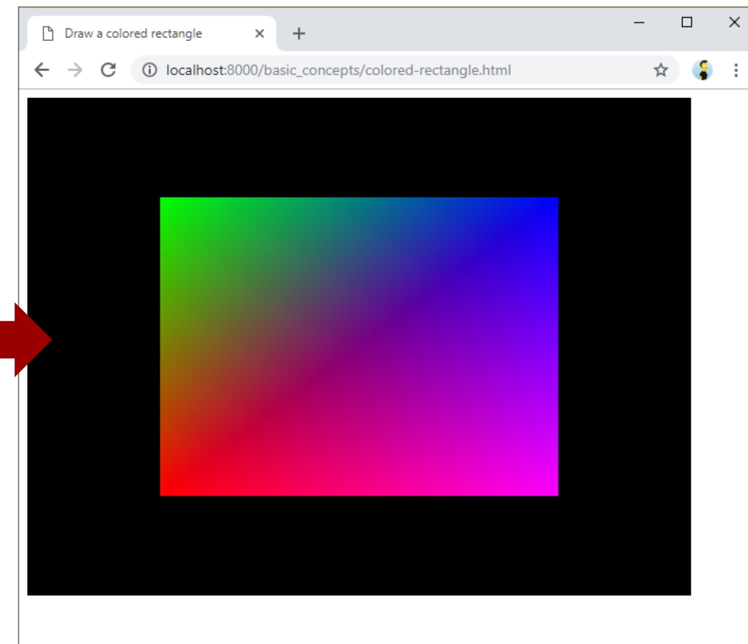
Índice de contenidos

1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. Resumen

Fork me on GitHub

9. 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>
```



Fork me on GitHub

9. 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)

Fork me on GitHub

9. 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`

Fork me on GitHub

9. 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 intialize shaders.');
        return;
    }

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

    // Init buffers
    var n = initBuffers(gl);
    if (n < 0) {
        console.log('Failed to init buffers');
        return;
    }

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

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

Índice de contenidos

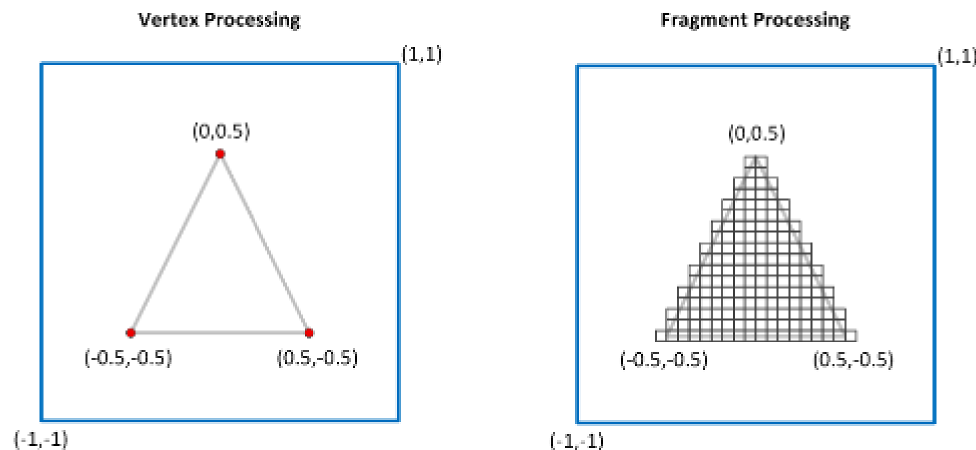
1. Visón general de WebGL
2. Referencia API WebGL
3. Referencia API GLSL ES
4. Ejemplo: colorear un canvas
5. Ejemplo: dibujar un punto
6. Ejemplo: dibujar un triángulo
7. Ejemplo: dibujar un rectángulo
8. Ejemplo: degradado de color en triángulo
9. Ejemplo: degradado de color en rectángulo
10. Resumen

10. 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***

10. 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)



10. 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