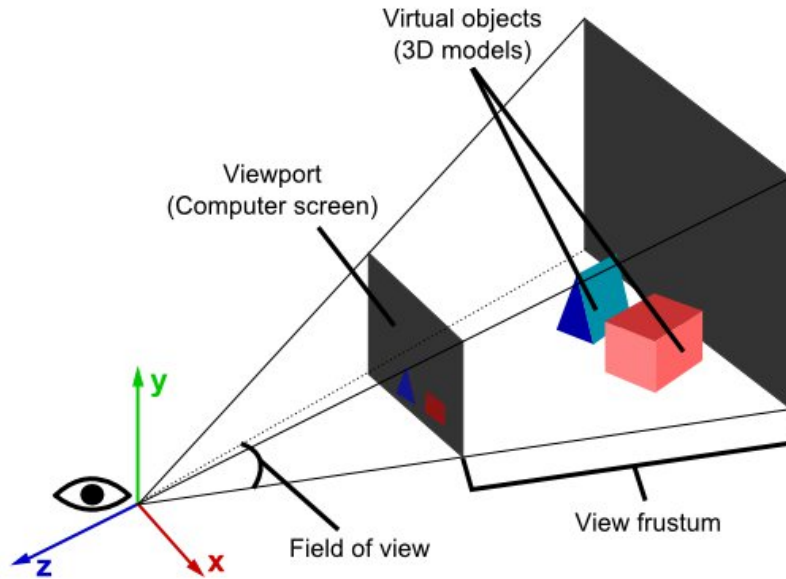




Universidad
Rey Juan Carlos

Escuela Técnica Superior
Ingeniería de Telecomunicación



Gráficos y visualización 3D

5. Proyecciones con WebGL

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

1. Proyecciones
2. Librería JavaScript glMatrix
3. Referencia API WebGL
4. Ejemplo: proyección ortogonal
5. Ejemplo: proyección en perspectiva
6. Área de visión
7. Ejemplo: cambio en viewport
8. Resumen

Índice de contenidos

1. Proyecciones

- I. Introducción
- II. Proyección ortogonal vs. perspectiva
- III. Proyección ortogonal
- IV. Proyección en perspectiva

2. Librería JavaScript glMatrix

3. Referencia API WebGL

4. Ejemplo: proyección ortogonal

5. Ejemplo: proyección en perspectiva

6. Área de visión

7. Ejemplo: cambio en viewport

8. Resumen

1. Proyecciones - Introducción

- En el tema anterior hemos estudiado cómo las **matrices de transformación** permiten realizar diferentes transformaciones básicas (traslación, escalado, rotación) a un conjunto de vértices (modelo 3D)
- En este tema vamos a ver que existen otras matrices (proyección y vista) que nos van a permitir jugar con la perspectiva de una escena

1. Proyecciones - Introducción

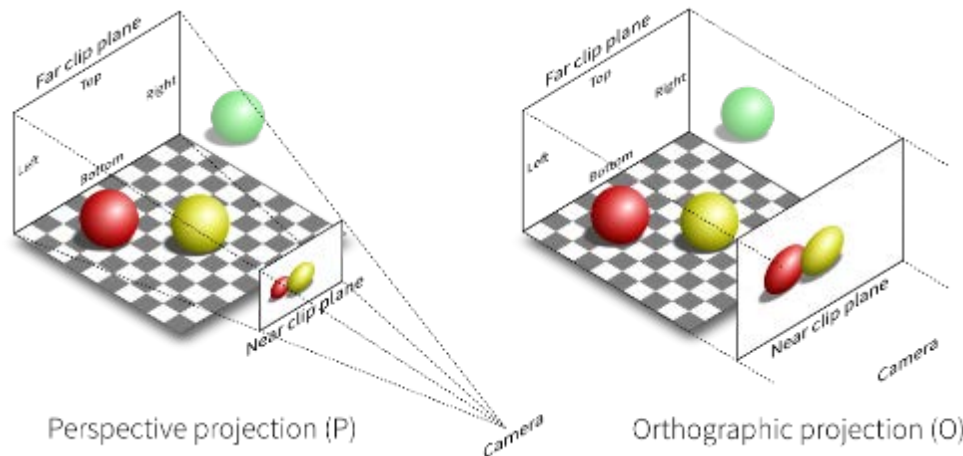
- De este modo, al final vamos a tener 3 matrices en la definición de los vértices
 - Matriz de proyección (*projection*): define el volumen de la escena (*clip space*)
 - Matriz de vista (*viewport*): define las coordenadas de la cámara (*view reference point* o VPR)
 - Matriz de transformación (*modelview*): modifica la posición de los vértices

```
<script id="shaderVs" type="x-shader/x-vertex">
  attribute vec4 a_Position;
  uniform mat4 u_pMatrix;
  uniform mat4 u_vMatrix;
  uniform mat4 u_mvMatrix;
  void main() {
    gl_Position = u_pMatrix * u_vMatrix * u_mvMatrix * a_Position;
  }
</script>
```

Estas 3 matrices se multiplicarán en el vertex shader

1. Proyecciones - Introducción

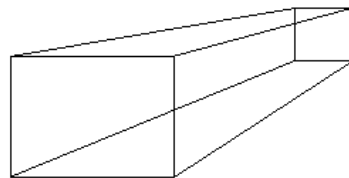
- La **matriz de proyección** cambia las coordenadas del área de visión (*view frustum*) de los vértices hacia un volumen estandarizado (un cubo que se extiende desde -1 a 1 en las 3 dimensiones del espacio)



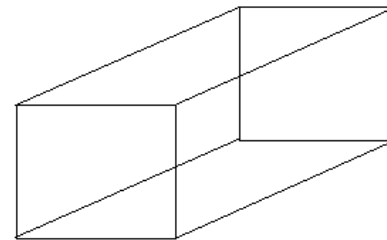
Los vértices se eliminan de la escena si se encuentra fuera del área de visión (*clipping*)

1. Proyecciones - ortogonal vs. perspectiva

- Hay dos tipos de proyecciones principales para los gráficos en WebGL:
 - **Proyección ortogonal:** Define un área de visión de tipo paralelepípedo (**cuadrada** en todas sus caras)
 - **Proyección en perspectiva:** Define un área de visión en forma de sección **piramidal** de forma que los objetos más alejados del observador se reducen en tamaño



Perspective projection

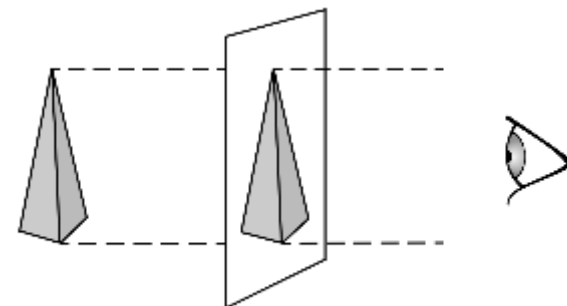


Orthographic projection

1. Proyecciones - ortogonal vs. perspectiva

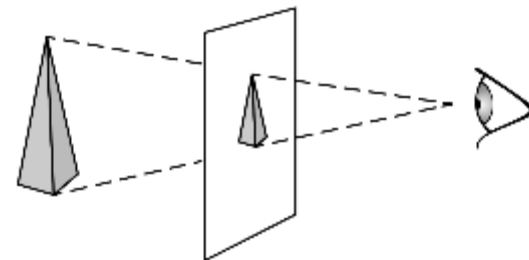
- La proyección ortogonal:

- Idóneo para diseños y planos
- Las líneas paralelas siguen siendo paralelas
- No proporciona sensación de profundidad



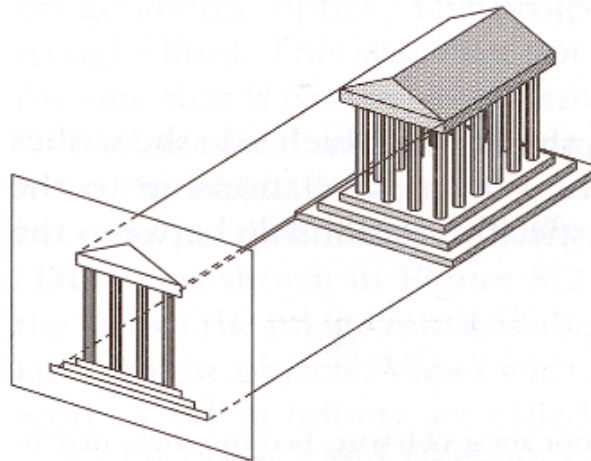
- La proyección en perspectiva:

- Idóneo para escenas realistas
- Las líneas paralelas convergen en los puntos de fuga
- Proporciona sensación de profundidad (el tamaño depende de la distancia)



1. Proyecciones - Proyección ortogonal

- La **proyección ortogonal** trabaja situando el centro de proyección en el infinito
- Todos los puntos tienen la misma dirección de proyección (DOP), son rectas paralelas
- Es la proyección usada por defecto en WebGL

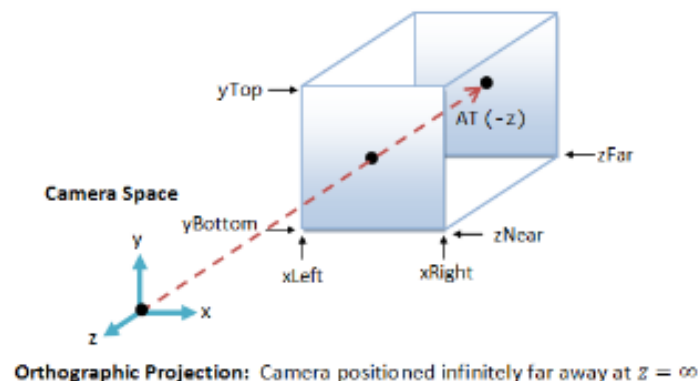


1. Proyecciones - Proyección ortogonal

- Se puede definir una proyección ortogonal en base a los siguiente parámetros:

- *Left*: Límite por la izquierda (eje X)
- *Right*: Límite por la derecha (eje X)
- *Bottom*: Límite inferior (eje Y)
- *Top*: Límite superior (eje Y)
- *Near*: Inicio de la escena (eje Z)
- *Far*: Final de la escena (eje Z)

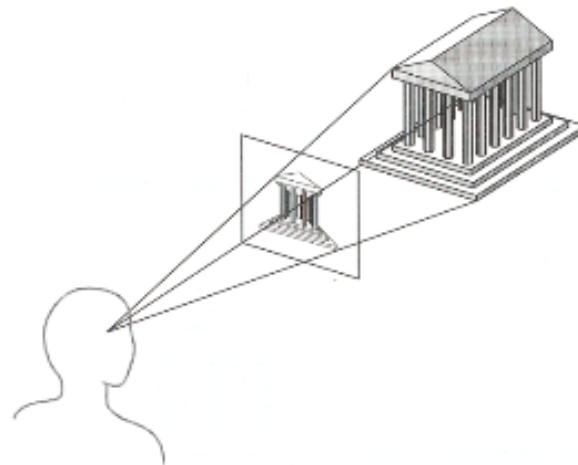
¡Los valores Near y Far no son intuitivos!



http://learnwebgl.brown37.net/08_projections/create_ortho/create_ortho.html

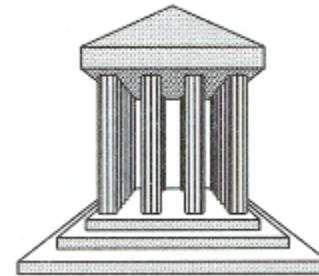
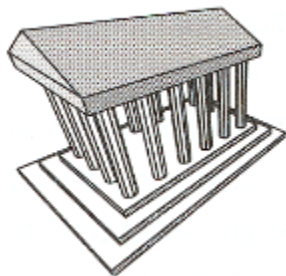
1. Proyecciones - Proyección en perspectiva

- La **proyección en perspectiva** permite representar los objetos en la forma y disposición con que se aparecen al espectador
- La idea es que los objetos lejanos se vean más pequeños
- Esta técnica se empezó a usar en el arte renacentista



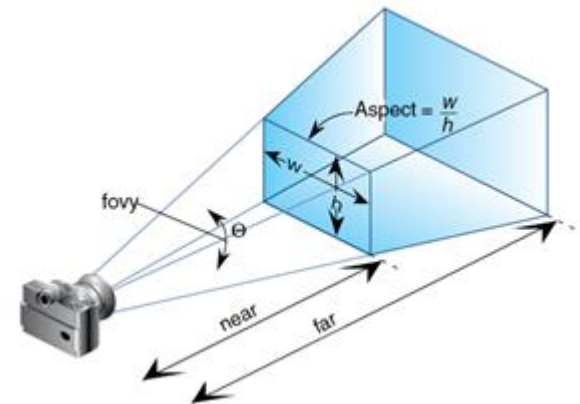
1. Proyecciones - Proyección en perspectiva

- Las líneas paralelas en un sistema en perspectiva convergen hacia un punto denominado punto de fuga
- Puede haber un punto de fuga (vista frontal), dos (vista oblicua) o tres puntos de fuga (vista área)



1. Proyecciones - Proyección en perspectiva

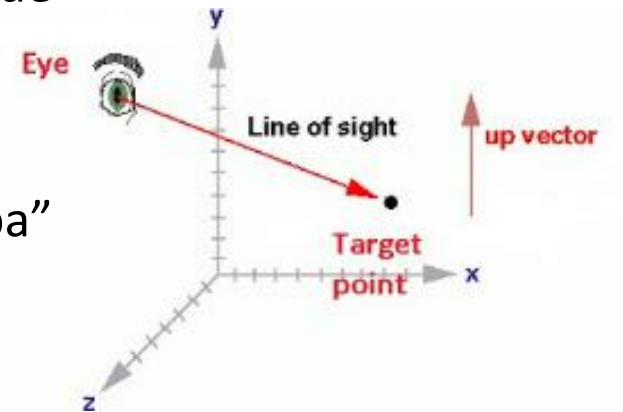
- Se puede definir una proyección en perspectiva en base a los siguiente parámetros:
 - *Fovy*: Ángulo de visión de la cámara respecto a la escena
 - *Aspect*: Relación ancho/alto de la escena
 - *Near*: Distancia de la cámara respecto al inicio de la escena
 - *Far*: Distancia de la cámara respecto al final de la escena



http://learnwebgl.brown37.net/08_projections/create_perspective/create_perspective.html

1. Proyecciones - Proyección en perspectiva

- En la proyección en perspectiva, es habitual cambiar la posición en la que **cámara** “mira” (*lookAt*) a un punto distante
- Esta configuración de la cámara se puede definir a en base a los siguientes parámetros:
 - *Eye*: Posición de la cámara (punto de referencia, VPR)
 - *Center*: Posición del centro objetivo
 - *Up*: Vector de lo que se considera “arriba” (orientación de la cámara)



http://learnwebgl.brown37.net/07_cameras/camera_trunk_lookat/camera_trunk_lookat.html

Índice de contenidos

1. Proyecciones
- 2. Librería JavaScript glMatrix**
3. Referencia API WebGL
4. Ejemplo: proyección ortogonal
5. Ejemplo: proyección en perspectiva
6. Área de visión
7. Ejemplo: cambio en viewport
8. Resumen

2. Librería JavaScript glMatrix

`(static) ortho(out, left, right, bottom, top, near, far) → {mat4}`

Generates a orthogonal projection matrix with the given bounds

Parameters:

Name	Type	Description
out	mat4	mat4 frustum matrix will be written into
left	number	Left bound of the frustum
right	number	Right bound of the frustum
bottom	number	Bottom bound of the frustum
top	number	Top bound of the frustum
near	number	Near bound of the frustum
far	number	Far bound of the frustum

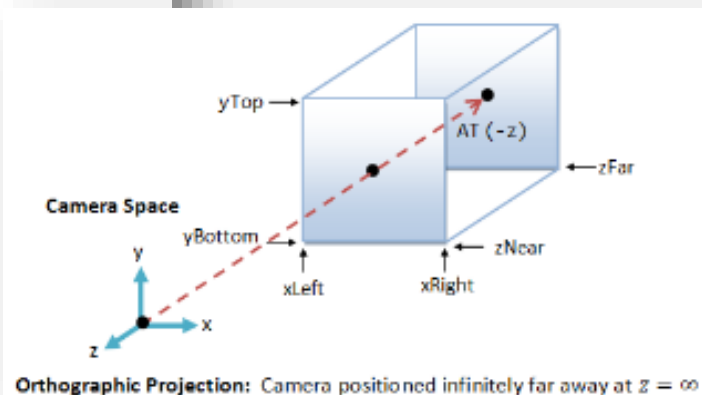
Source: [mat4.js, line 1348](#)

Returns:

out

Type

mat4



<http://glmatrix.net/docs/module-mat4.html>

2. Librería JavaScript glmMatrix

`(static) perspective(out, fovy, aspect, near, far) → {mat4}`

Generates a perspective projection matrix with the given bounds. Passing null/undefined/no value for far will generate infinite projection matrix.

Parameters:

Name	Type	Description
out	mat4	mat4 frustum matrix will be written into
fovy	number	Vertical field of view in radians
aspect	number	Aspect ratio, typically viewport width/height
near	number	Near bound of the frustum
far	number	Far bound of the frustum, can be null or Infinity

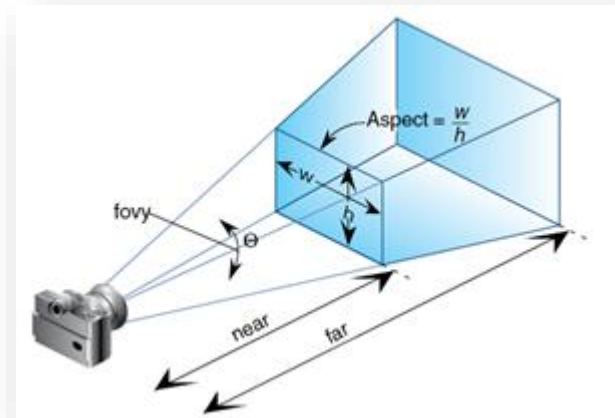
Source: [mat4.js, line 1271](#)

Returns:

out

Type

mat4



<http://glmatrix.net/docs/module-mat4.html>

2. Librería JavaScript glmMatrix

`(static) lookAt(out, eye, center, up) → {mat4}`

Generates a look-at matrix with the given eye position, focal point, and up axis. If you want a matrix that actually makes an object look at another object, you should use `targetTo` instead.

Parameters:

Name	Type	Description
out	mat4	mat4 frustum matrix will be written into
eye	vec3	Position of the viewer
center	vec3	Point the viewer is looking at
up	vec3	vec3 pointing up

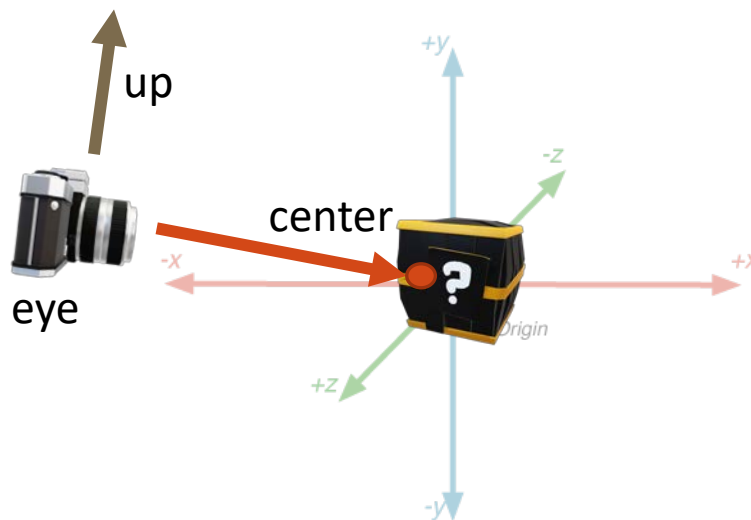
Source: [mat4.js, line 1381](#)

Returns:

out

Type

mat4



<http://glmatrix.net/docs/module-mat4.html>

http://learnwebgl.brown37.net/07_cameras/camera_rotating_motion.html

Índice de contenidos

1. Proyecciones
2. Librería JavaScript glMatrix
- 3. Referencia API WebGL**
4. Ejemplo: proyección ortogonal
5. Ejemplo: proyección en perspectiva
6. Área de visión
7. Ejemplo: cambio en viewport
8. Resumen

2. Referencia API WebGL

`gl.enable(cap)`

Enable the function specified by *cap* (capability).

Parameters	cap	Specifies the function to be enabled.
	<code>gl.DEPTH_TEST</code> ²	Hidden surface removal
	<code>gl.BLEND</code>	Blending (see Chapter 9, “Hierarchical Objects”)
	<code>gl.POLYGON_OFFSET_FILL</code>	Polygon offset (see the next section), and so on ³
Return value	None	
Errors:	<code>INVALID_ENUM</code>	None of the acceptable values is specified in <i>cap</i>

² A “DEPTH_TEST” in the hidden surface removal function might sound strange, but actually its name comes from the fact that it decides which objects to draw in the foreground by verifying (TEST) the depth (DEPTH) of each object.

³ Although not covered in this book, you can also specify `gl.CULL_FACE`, `gl.DITHER`, `gl.SAMPLE_ALPHA_TO_COVERAGE`, `gl.SAMPLE_COVERAGE`, `gl.SCISSOR_TEST`, and `gl.STENCIL_TEST`. See the book *OpenGL Programming Guide* for more information on these.

3. Re

```
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 <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.		
	type	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>
		normalized	Either <code>true</code> or <code>false</code> 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, <i>offset</i> is 0.		
Return value	None			
Errors	<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.		

3. Referencia API WebGL

`gl.viewport(x, y, width, height)`

Set the viewport where `gl.drawArrays()` or `gl.drawElements()` draws. In WebGL, x and y are specified in the `<canvas>` coordinate system.

Parameters	x, y	Specify the lower-left corner of the viewport rectangle (in pixels).
	width, height	Specify the width and height of the viewport (in pixels).
Return value	None	
Errors	None	

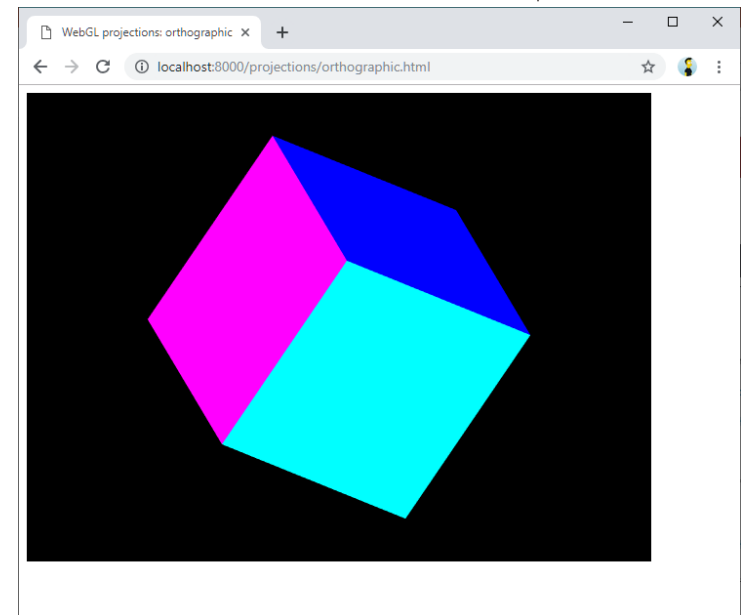
Índice de contenidos

1. Proyecciones
2. Librería JavaScript glMatrix
3. Referencia API WebGL
- 4. Ejemplo: proyección ortogonal**
5. Ejemplo: proyección en perspectiva
6. Área de visión
7. Ejemplo: cambio en viewport
8. Resumen

4. Ejemplo: proyección ortogonal

```
<!DOCTYPE html>
<html>
<head>
  <title>WebGL projections: orthographic</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/gl-matrix/2.8.1/gl-matrix-min.js"></script>
  <script id="shaderVs" type="x-shader/x-vertex">
    attribute vec4 a_Position;
    attribute vec3 a_Color;
    uniform mat4 u_pMatrix;
    uniform mat4 u_mvMatrix;
    varying highp vec4 v_Color;
    void main() {
      gl_Position = u_pMatrix * u_mvMatrix * a_Position;
      v_Color = vec4(a_Color, 1.0);
    }
  </script>
  <script id="shaderFs" type="x-shader/x-fragment">
    varying highp vec4 v_Color;
    void main() {
      gl_FragColor = v_Color;
    }
  </script>
  <script>
    // ...
  </script>
</head>
<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>
</html>
```

Se usa la matriz de
transformación y
proyección



4. Ejemplo: proyección ortogonal

```
<script>
  var gl;
  var count = 0.0;

  function init() {
    // Get canvas object from the DOM
    var canvas = document.getElementById("myCanvas");
    // Init WebGL context
    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;
    initShaders(gl, vs, fs);
    // Init vertex shader
    initVertexShader(gl);

    // Init projection matrix
    initProjection(gl, canvas);
    // Set clear canvas color
    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    // Hidden surface removal
    gl.enable(gl.DEPTH_TEST);
    // Draw Scene
    drawScene();
  }
</script>
```

La función `init()` es similar a la que hemos visto en ejemplos anteriores

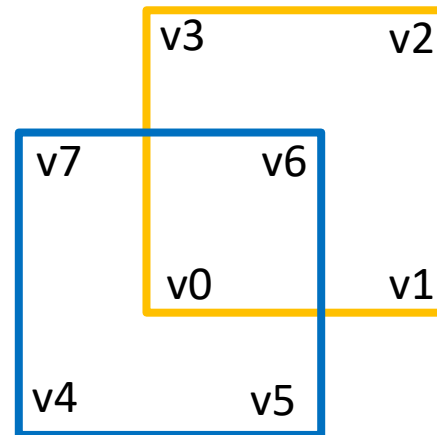
Como primera novedad, se llama a la función `initProjection` que implementará la matriz de proyección

Además se fuerza la eliminación de las superficies ocultas para que el cubo sea sólido

4. Ejemplo: proyección ortogonal

```
<script>
function initVertexShader(gl) {
    var vertexesAndColors = [ // Vertexes and colors (X, Y, Z, R, G, B)
        // Back - Yellow
        -0.5, -0.5, -0.5, 1, 1, 0, // v0
        0.5, -0.5, -0.5, 1, 1, 0, // v1
        0.5, 0.5, -0.5, 1, 1, 0, // v2
        -0.5, 0.5, -0.5, 1, 1, 0, // v3
        // Front - Blue
        -0.5, -0.5, 0.5, 0, 0, 1, // v4
        0.5, -0.5, 0.5, 0, 0, 1, // v5
        0.5, 0.5, 0.5, 0, 0, 1, // v6
        -0.5, 0.5, 0.5, 0, 0, 1, // v7
        // Left - Light Blue
        -0.5, -0.5, -0.5, 0, 1, 1, // v8
        -0.5, 0.5, -0.5, 0, 1, 1, // v9
        -0.5, 0.5, 0.5, 0, 1, 1, // v10
        -0.5, -0.5, 0.5, 0, 1, 1, // v11
        // Right - Red
        0.5, -0.5, -0.5, 1, 0, 0, // v12
        0.5, 0.5, -0.5, 1, 0, 0, // v13
        0.5, 0.5, 0.5, 1, 0, 0, // v14
        0.5, -0.5, 0.5, 1, 0, 0, // v15
        // Bottom - Pink
        -0.5, -0.5, -0.5, 1, 0, 1, // v16
        -0.5, -0.5, 0.5, 1, 0, 1, // v17
        0.5, -0.5, 0.5, 1, 0, 1, // v18
        0.5, -0.5, -0.5, 1, 0, 1, // v19
        // Top - Green
        -0.5, 0.5, -0.5, 0, 1, 0, // v20
        -0.5, 0.5, 0.5, 0, 1, 0, // v21
        0.5, 0.5, 0.5, 0, 1, 0, // v22
        0.5, 0.5, -0.5, 0, 1, 0, // v23
    ];
    // ...
}
```

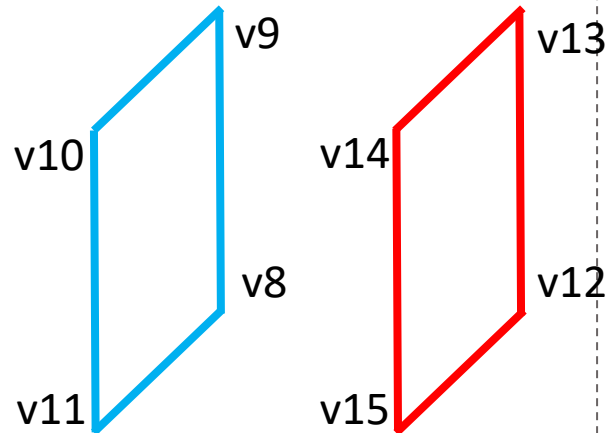
En este ejemplo vamos a definir las posición de los vértices (x, y, z) en el mismo array que los colores (r, g, b) de cada vértice



4. Ejemplo: proyección ortogonal

```
<script>
function initVertexShader(gl) {
    var vertexesAndColors = [ // Vertexes and colors (X, Y, Z, R, G, B)
        // Back - Yellow
        -0.5, -0.5, -0.5, 1, 1, 0, // v0
        0.5, -0.5, -0.5, 1, 1, 0, // v1
        0.5, 0.5, -0.5, 1, 1, 0, // v2
        -0.5, 0.5, -0.5, 1, 1, 0, // v3
        // Front - Blue
        -0.5, -0.5, 0.5, 0, 0, 1, // v4
        0.5, -0.5, 0.5, 0, 0, 1, // v5
        0.5, 0.5, 0.5, 0, 0, 1, // v6
        -0.5, 0.5, 0.5, 0, 0, 1, // v7
        // Left - Light Blue
        -0.5, -0.5, -0.5, 0, 1, 1, // v8
        -0.5, 0.5, -0.5, 0, 1, 1, // v9
        -0.5, 0.5, 0.5, 0, 1, 1, // v10
        -0.5, -0.5, 0.5, 0, 1, 1, // v11
        // Right - Red
        0.5, -0.5, -0.5, 1, 0, 0, // v12
        0.5, 0.5, -0.5, 1, 0, 0, // v13
        0.5, 0.5, 0.5, 1, 0, 0, // v14
        0.5, -0.5, 0.5, 1, 0, 0, // v15
        // Bottom - Pink
        -0.5, -0.5, -0.5, 1, 0, 1, // v16
        -0.5, -0.5, 0.5, 1, 0, 1, // v17
        0.5, -0.5, 0.5, 1, 0, 1, // v18
        0.5, -0.5, -0.5, 1, 0, 1, // v19
        // Top - Green
        -0.5, 0.5, -0.5, 0, 1, 0, // v20
        -0.5, 0.5, 0.5, 0, 1, 0, // v21
        0.5, 0.5, 0.5, 0, 1, 0, // v22
        0.5, 0.5, -0.5, 0, 1, 0, // v23
    ];
    // ...
};
```

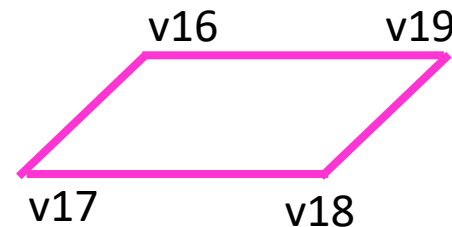
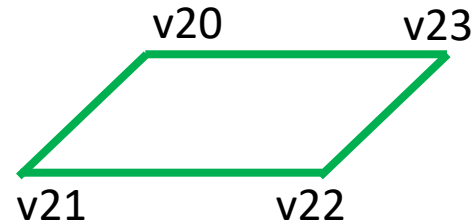
En este ejemplo vamos a definir las posición de los vértices (x, y, z) en el mismo array que los colores (r, g, b) de cada vértice



4. Ejemplo: proyección ortogonal

```
<script>
function initVertexShader(gl) {
    var vertexesAndColors = [ // Vertexes and colors (X, Y, Z, R, G, B)
        // Back - Yellow
        -0.5, -0.5, -0.5, 1, 1, 0, // v0
        0.5, -0.5, -0.5, 1, 1, 0, // v1
        0.5, 0.5, -0.5, 1, 1, 0, // v2
        -0.5, 0.5, -0.5, 1, 1, 0, // v3
        // Front - Blue
        -0.5, -0.5, 0.5, 0, 0, 1, // v4
        0.5, -0.5, 0.5, 0, 0, 1, // v5
        0.5, 0.5, 0.5, 0, 0, 1, // v6
        -0.5, 0.5, 0.5, 0, 0, 1, // v7
        // Left - Light Blue
        -0.5, -0.5, -0.5, 0, 1, 1, // v8
        -0.5, 0.5, -0.5, 0, 1, 1, // v9
        -0.5, 0.5, 0.5, 0, 1, 1, // v10
        -0.5, -0.5, 0.5, 0, 1, 1, // v11
        // Right - Red
        0.5, -0.5, -0.5, 1, 0, 0, // v12
        0.5, 0.5, -0.5, 1, 0, 0, // v13
        0.5, 0.5, 0.5, 1, 0, 0, // v14
        0.5, -0.5, 0.5, 1, 0, 0, // v15
        // Bottom - Pink
        -0.5, -0.5, -0.5, 1, 0, 1, // v16
        -0.5, -0.5, 0.5, 1, 0, 1, // v17
        0.5, -0.5, 0.5, 1, 0, 1, // v18
        0.5, -0.5, -0.5, 1, 0, 1, // v19
        // Top - Green
        -0.5, 0.5, -0.5, 0, 1, 0, // v20
        -0.5, 0.5, 0.5, 0, 1, 0, // v21
        0.5, 0.5, 0.5, 0, 1, 0, // v22
        0.5, 0.5, -0.5, 0, 1, 0, // v23
    ];
    // ...
};
```

En este ejemplo vamos a definir las posición de los vértices (x, y, z) en el mismo array que los colores (r, g, b) de cada vértice



```
// Indexes (for drawing squares using triangles)
```

```
var indexes = [
    0, 1, 2,
    0, 2, 3,

    4, 5, 6,
    4, 6, 7,

    8, 9, 10,
    8, 10, 11,

    12, 13, 14,
    12, 14, 15,

    16, 17, 18,
    16, 18, 19,

    20, 21, 22,
    20, 22, 23
];
```

Vamos a usar índices
para dibujar el cubo en
base a triángulos

Se crea el buffer
`gl.ARRAY_BUFFER` con
el contenido de vértices
y colores

```
// Write a_Position and a_Color using gl.ARRAY_BUFFER
```

```
gl.bindBuffer(gl.ARRAY_BUFFER, gl.createBuffer());
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexesAndColors), gl.STATIC_DRAW);
```

```
var vertexPositionAttribute = gl.getAttribLocation(gl.program, "a_Position");
gl.enableVertexAttribArray(vertexPositionAttribute);
gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT, false, 4 * (3 + 3), 0);
```

```
var vertexColorAttribute = gl.getAttribLocation(gl.program, "a_Color");
gl.enableVertexAttribArray(vertexColorAttribute);
gl.vertexAttribPointer(vertexColorAttribute, 3, gl.FLOAT, false, 4 * (3 + 3), 4 * 3);
```

```
// Write indexes in gl.ELEMENT_ARRAY_BUFFER
```

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, gl.createBuffer());
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indexes), gl.STATIC_DRAW);
```

```
}
```

```
</script>
```

El desplazamiento en
bytes de los vértices será
 $4 * (3 + 3)$, donde 4
es el número de bytes de
un `gl.FLOAT` y $(3 + 3)$
es la posición
(x, y, z, r, g, b)

4. Referencia API - vertexAttribPointer

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

- Une el búfer actualmente vinculado a gl.ARRAY_BUFFER a un atributo de vértice genérico y especifica su diseño
- Size: indica el número de componentes por vértice (1, 2, 3 o 4)
- Type: indica el tipo de dato de cada componente del array:
 - gl.BYTE: 8-bits | 1 byte
 - gl.SHORT: 16-bits | 2 bytes
 - gl.UNSIGNED_BYTE: 8-bits | 1 byte
 - gl.UNSIGNED_SHORT: 16-bits | 2 bytes
 - gl.FLOAT: 32-bits | 4 bytes
- Offset: indica el desplazamiento en bytes de el primer componente en el array de vertex. Debe ser múltiplo de la longitud de bytes de type

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

4. Ejemplo: proyección ortogonal

```
<script>
function drawScene() {
    // Clear
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Rotate
    var mvMatrix = glMatrix.mat4.fromRotation(glMatrix.mat4.create(), count, [0.5, 0.5, 0.5]);
    var uMvMatrix = gl.getUniformLocation(gl.program, "u_mvMatrix");
    gl.uniformMatrix4fv(uMvMatrix, false, mvMatrix);

    // Draw
    gl.drawElements(gl.TRIANGLES, 6 * 2 * 3, gl.UNSIGNED_SHORT, 0);

    // Call drawScene again in the next browser repaint
    count += 0.01;
    requestAnimationFrame(drawScene);
}

function initProjection(gl, canvas) {
    // Write u_pMatrix
    var pMatrixUniform = gl.getUniformLocation(gl.program, "u_pMatrix");
    var ratio = canvas.width / canvas.height;
    var pMatrix = glMatrix.mat4.ortho(glMatrix.mat4.create(), -ratio, ratio, -1.0, 1.0, -1.0, 1.0);
    gl.uniformMatrix4fv(pMatrixUniform, false, pMatrix);
}
</script>
```

La función `drawScene()` realiza una rotación en los 3 ejes y se llama a sí misma mediante `requestAnimationFrame`

En la función `initProjection` se calcula la matriz de proyección usando `mat4.ortho`. El valor de esta matriz se escribe en la variable `u_pMatrix` del vertex shader

4. Ejemplo: proyección ortogonal

```
<script>
function drawScene() {
    // Clear
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Rotate
    var mvMatrix = glMatrix.mat4.fromRotation(glMatrix.mat4.create(), count, [0.5, 0.5, 0.5]);
    var uMvMatrix = gl.getUniformLocation(gl.program, "u_mvMatrix");
    gl.uniformMatrix4fv(uMvMatrix, false, mvMatrix);

    // Draw
    gl.drawElements(gl.TRIANGLES, 6 * 2 * 3, gl.UNSIGNED_SHORT, 0);

    // Call drawScene again in the next frame
    count += 0.01;
    requestAnimationFrame(drawScene);
}

function initProjection(gl, canvas) {
    // Write u_pMatrix
    var pMatrixUniform = gl.getUniformLocation(gl.program, "u_pMatrix");
    var ratio = canvas.width / canvas.height;
    var pMatrix = glMatrix.mat4.ortho(glMatrix.mat4.create(), -ratio, ratio, -1.0, 1.0, 1.0, -1.0);
    gl.uniformMatrix4fv(pMatrixUniform, false, pMatrix);
}
</script>
```

Es importante indicar el tamaño de los elementos a dibujar, en este caso son **6** caras, formadas por **2** triángulos los cuales tienen **3** vértices cada uno. Coincide con el **número de índices** que se han definido, no con los vértices

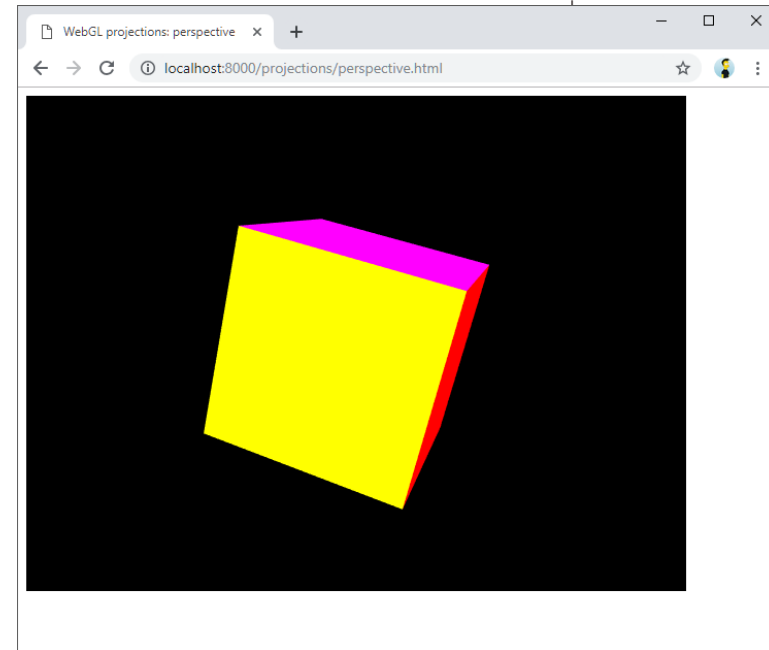
Índice de contenidos

1. Proyecciones
2. Librería JavaScript glMatrix
3. Referencia API WebGL
4. Ejemplo: proyección ortogonal
- 5. Ejemplo: proyección en perspectiva**
6. Área de visión
7. Ejemplo: cambio en viewport
8. Resumen

5. Ejemplo: proyección en perspectiva

```
<!DOCTYPE html>
<html>
<head>
  <title>WebGL projections: perspective</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/gl-matrix/2.8.1/gl-matrix-min.js"></script>
  <script id="shaderVs" type="x-shader/x-vertex">
    attribute vec4 a_Position;
    attribute vec3 a_Color;
    uniform mat4 u_pMatrix;
    uniform mat4 u_vMatrix;
    uniform mat4 u_mvMatrix;
    varying highp vec4 v_Color;
    void main() {
      gl_Position = u_pMatrix * u_vMatrix * u_mvMatrix * a_Position;
      v_Color = vec4(a_Color, 1.0);
    }
  </script>
  <script id="shaderFs" type="x-shader/x-fragment">
    varying highp vec4 v_Color;
    void main() {
      gl_FragColor = v_Color;
    }
  </script>
  <script>
    // ...
  </script>
</head>
<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas>
</body>
</html>
```

Se usa la matriz de
transformación,
proyección, y vista



5. Ejemplo: proyección en perspectiva

```
<script>
function toRadian(g) {
    return g*Math.PI/180;
}

function initProjection(gl, canvas) {
    // Write u_pMatrix
    var pMatrixUniform = gl.getUniformLocation(gl.program, "u_pMatrix");
    var ratio = canvas.width / canvas.height;
    var pMatrix = glMatrix.mat4.perspective(glMatrix.mat4.create(), toRadian(60), ratio, 0.1, 100);
    gl.uniformMatrix4fv(pMatrixUniform, false, pMatrix);

    // Write u_vMatrix
    var vMatrixUniform = gl.getUniformLocation(gl.program, "u_vMatrix");
    var vMatrix = glMatrix.mat4.lookAt(glMatrix.mat4.create(), [0, 0, 3], [0, 0, 0], [0, 1, 0]);
    gl.uniformMatrix4fv(vMatrixUniform, false, vMatrix);
}
</script>
```

Cálculo del ángulo FOV en radianes

En la función `initProjection` se calcula la **matriz de proyección** usando `mat4.perspective`. El valor de esta matriz se escribe en la variable `u_pMatrix` del vertex shader

Además, se calcula la **matriz de vista** usando `mat4.lookAt` y se escribe el valor de esta matriz en la variable `u_vMatrix` del vertex shader

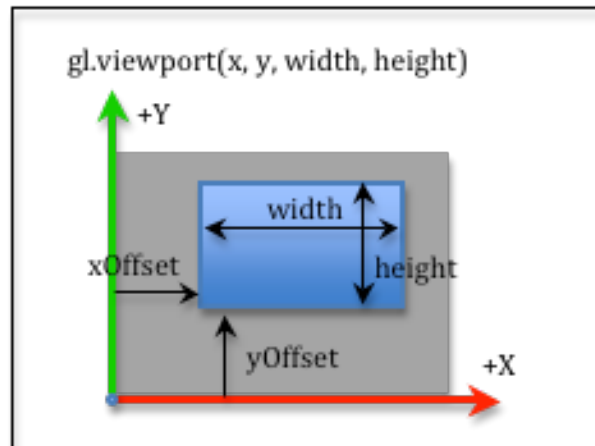
Índice de contenidos

1. Proyecciones
2. Librería JavaScript glMatrix
3. Referencia API WebGL
4. Ejemplo: proyección ortogonal
5. Ejemplo: proyección en perspectiva
- 6. Área de visión**
7. Ejemplo: cambio en viewport
8. Resumen

6. Área de visión

- La API JavaScript de WebGL nos permite elegir cuál será el área de visión de nuestro gráfico dentro del canvas HTML5, mediante la función:

```
gl.viewport(xOffset, yOffset, width, height);
```



Índice de contenidos

1. Proyecciones
2. Librería JavaScript glMatrix
3. Referencia API WebGL
4. Ejemplo: proyección ortogonal
5. Ejemplo: proyección en perspectiva
6. Área de visión
- 7. Ejemplo: cambio en viewport**
8. Resumen

7. Ejemplo: cambio en viewport

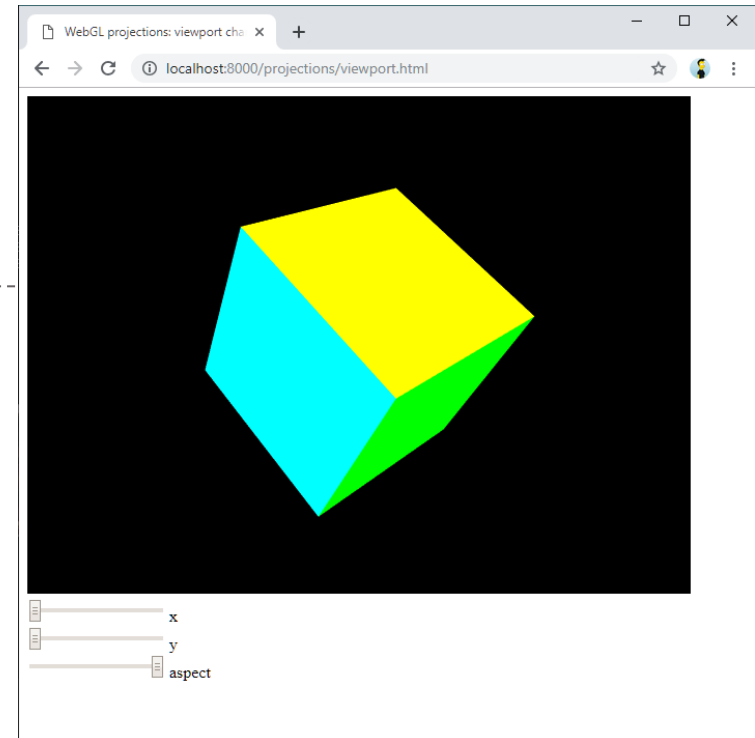
Partimos del ejemplo anterior y añadimos controles de tipo rango (**type="range"**) en la interfaz de usuario

```
<!DOCTYPE html>
<html>

<head>
  <title>WebGL projections: viewport change</title>

  <script>
    // ...
  </script>
</head>

<body onload="init()">
  <canvas id="myCanvas" width="640" height="480"></canvas><br>
  <input type="range" id="x" min="0" max="640" value="0" step="1"> x<br>
  <input type="range" id="y" min="0" max="640" value="0" step="1"> y<br>
  <input type="range" id="aspect" min="0" max="1" value="1" step="0.01"> aspect<br>
</body>
</html>
```



7. Ejemplo: cambio en viewport

Cada vez que se dibuja nuestra escena (`drawScene()`) se lee el valor de los rangos y se usan para invocar el método `viewport` de la API WebGL

```
function drawScene() {
  // Change viewport
  var x = document.getElementById("x").value;
  var y = document.getElementById("y").value;
  var aspect = document.getElementById("aspect").value;
  gl.viewport(x, y, aspect * canvas.width, aspect * canvas.height);

  // Clear
  gl.clear(gl.COLOR_BUFFER_BIT);

  // Rotate
  var mvMatrix = glMatrix.mat4.fromRotation(glMatrix.mat4.create(), count, [0.5, 0.5, 0.5]);
  var uMvMatrix = gl.getUniformLocation(gl.program, "u_mvMatrix");
  gl.uniformMatrix4fv(uMvMatrix, false, mvMatrix);

  // Draw
  gl.drawElements(gl.TRIANGLES, 6 * 2 * 3, gl.UNSIGNED_SHORT, 0);

  // Call drawScene again in the next browser repaint
  count += 0.01;
  requestAnimationFrame(drawScene);
}
```


Índice de contenidos

1. Proyecciones
2. Librería JavaScript glMatrix
3. Referencia API WebGL
4. Ejemplo: proyección ortogonal
5. Ejemplo: proyección en perspectiva
6. Área de visión
7. Ejemplo: cambio en viewport
- 8. Resumen**

8. Resumen

- Hay dos tipos de proyecciones principales:
 - **Proyección ortogonal**
 - **Proyección en perspectiva**
- A cada una de estas proyecciones le corresponde una **matriz de proyección**
- La generación de estas matrices es muy sencilla en JavaScript usando una librería, como **glMatrix**
- Podemos cambiar el área de visión (**viewport**), de manera que la escena se adapte al área disponible en el navegador

8. Resumen

- El orden correcto de multiplicación de matrices dentro del **vertex shader** es inverso al deseado:
 - Proyección (**pMatrix**: perspective/ortho)
 - Vista (**vMatrix**: lookAt)
 - Modelo:
 - Transformación (**mvMatrix** : translación/rotación/escalado)
 - Posición de los vértices (**a_Position**: vertex)

```
gl_Position = u_pMatrix * u_vMatrix * u_mvMatrix * a_Position;
```