Desarrollo de Software V

Laboratorio # 10 – Prof. Regis Rivera

Objetivo: Introducción a la etiqueta Canvas de HTML5 (en 3 dimensiones)

Instrucciones: Seguir la explicación y pasos a seguir para elaborar ejemplos prácticos del uso de la etiqueta Canvas para 3 dimensiones mediante WebGL.

A. Introduccion a WebGL

WebGL permite que el contenido web use una API basada en OpenGL ES 2.0 para realizar renderizado 2D y 3D en un HTML canvas en navegadores que lo admiten sin el uso de complementos. Los programas WebGL consisten en código de control escrito en JavaScript y código de sombreador (GLSL) que se ejecuta en la Unidad de procesamiento de gráficos (GPU) de una computadora. Los elementos WebGL se pueden mezclar con otros elementos HTML y componer con otras partes de la página o el fondo de la página. Este laboratorio presentará los conceptos básicos del uso de WebGL.

Diferencia entre los ejemplos de este laboratorio vs el anterior, es que ahora utilzaremos WebGL:

- `canvas.getContext("2d");` ← Contexto en 2 dimensiones
- `canvas.getContext("webgl");` ← Contexto en 3 dimensiones

B. Preparándose para renderizar en 3D

Lo primero que necesita para usar WebGL para renderizar es un lienzo. El siguiente fragmento HTML declara un lienzo en el que se dibujará nuestra muestra:

Lab101.html

```
<!DOCTYPE html>
<html>
<head>
<title>Canvas</title>
</head>
<body>

<body>
  <canvas id="glCanvas" width="640" height="480"></canvas>
</body>

<script>

function main() {
  const canvas = document.querySelector("#glCanvas");
  // inicializar contexto GL
  const gl = canvas.getContext("webgl");

  //Continuar solo si WebGL está disponible y funcionando
  if (gl === null) {
    alert("No se puede inicializar WebGL. Es posible que su navegador o máquina no lo admita");
    return;
  }

  //Establecer color claro a negro, totalmente opaco
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  //Borrar el búfer de color con el color claro especificado
  gl.clear(gl.COLOR_BUFFER_BIT);
}

window.onload = main;

</script>

</body>
</html>
```

La main() función en nuestro código JavaScript se llama cuando se carga nuestro script. Su propósito es configurar el contexto WebGL y comenzar a representar el contenido.
Lo primero que hacemos aquí es obtener una referencia al lienzo, asignándolo a una variable llamada canvas.

Una vez que tenemos el lienzo, intentamos obtener un WebGLRenderingContext para él llamando a getContext y pasándole la cadena "webgl". Si el navegador no es compatible, webgl getContext volverá, null en cuyo caso mostraremos un mensaje al usuario y saldremos.
Si el contexto se inicializa con éxito, la variable *gl* es nuestra referencia a él. En este caso, establecemos el color claro en negro y borramos el contexto a ese color (redibujando el lienzo con el color de fondo).

En este punto, tiene suficiente código para que el contexto WebGL se inicialice con éxito, y debe terminar con una gran caja negra y vacía, lista y esperando recibir contenido.

C. Entendiendo WebGL

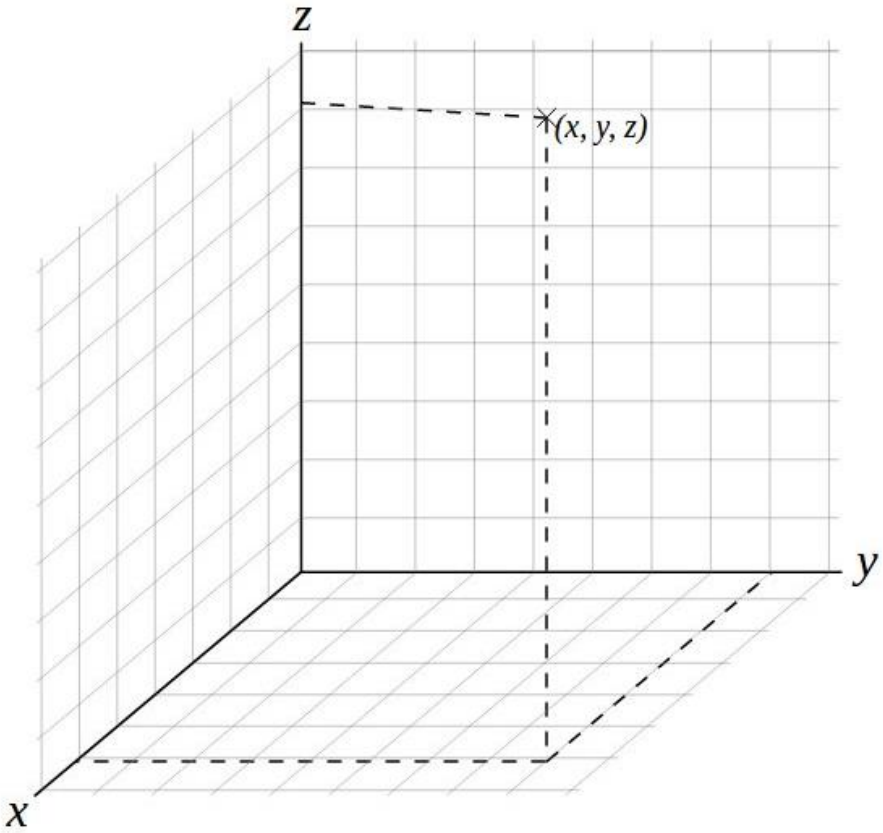
WebGL no se trata solo de dibujar algunos píxeles específicos en un lienzo. Para dibujar una imagen, tenemos que definir un vector espacial que represente la imagen, que se convertirá, utilizando la especificación OpenGL, a su representación de píxeles.

Esto puede sonar aterrador, y es por eso que debemos entender todo el proceso para escribir aplicaciones WebGL.

Sistema coordinado

Al ser un sistema tridimensional, tenemos tres ejes: X, Y y Z, siendo el último la profundidad. En WebGL, las coordenadas están limitadas a (1, 1, 1) y (-1, -1, -1).

Tenemos que entender que, cuando definimos figuras, no tenemos que pensar en píxeles, sino en representaciones vectoriales en un sistema de coordenadas dentro de un sistema cartesiano.

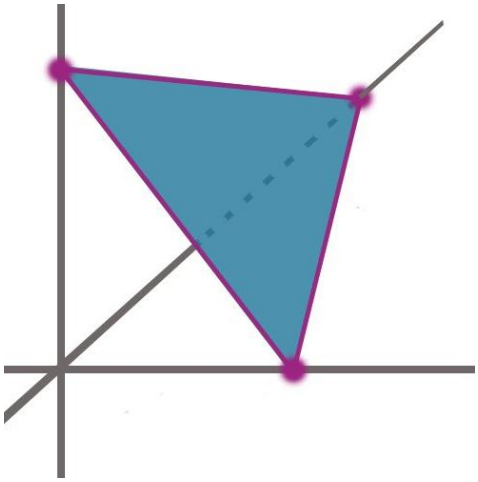


Imagine el sistema como en la imagen de arriba (eso es realmente lo que vamos a tratar). El punto dibujado (x, y, z) está representado por las coordenadas en los ejes X, Y y X.

Ahora supongamos que queremos construir un cubo, y que las coordenadas comentadas anteriormente son para uno de los límites del sistema WebGL. Esa coordenada sería (1, 1, 1). Eso significa que tenemos solo una de las esquinas del cubo, y un cubo tiene 8 esquinas. Entonces necesitamos varias coordenadas más.

Las otras coordenadas serían (1, 1, -1), (1, -1, -1), (1, -1, 1), (-1, -1, -1), (-1, 1, -1), (-1, 1, 1) y (-1, -1, 1). Rastreando los vórtices entre estas coordenadas, definiríamos el volumen del cubo.

Esto podría ser más fácil de entender en la segunda dimensión, solo con los ejes X e Y, donde el principio subyacente es el mismo. Suponga tres coordenadas, por ejemplo (0.8, 0), (0, 1) y (1, 0.8). Con estos vórtices, dibujaríamos un triángulo como en la siguiente imagen.



Como se dijo, el principio es el mismo que para las figuras tridimensionales con un tercer eje, pero más fácil de entender si el ejemplo anterior era confuso.

Sombreadores

Los sombreadores son fragmentos de código que se ejecutan en la GPU para representar cada píxel que formará la escena final. Hay dos tipos de sombreadores:

- Sombreador de vértices: este, como probablemente ya habrá adivinado, es el responsable de manipular y representar los vértices, calcular la textura y la posición de cada vértice, entre otras cosas.
- Sombreador de fragmentos: este manipula cada píxel que compone el área delimitada por los vértices.

D. Cubo Rotatorio

Tomemos nuestro plano cuadrado en tres dimensiones agregando cinco caras más para crear un cubo. Para hacer esto de manera eficiente, vamos a pasar del dibujo usando los vértices directamente llamando al gl.drawArrays() método para usar la matriz de vértices como una tabla, y haciendo referencia a los vértices individuales en esa tabla para definir las posiciones de los vértices de cada cara, llamando gl.drawElements().

Considere: cada cara requiere cuatro vértices para definirla, pero cada vértice es compartido por tres caras. Podemos pasar muchos menos datos construyendo una matriz de los 24 vértices, luego refiriéndonos a cada vértice por su índice en esa matriz en lugar de mover conjuntos completos de coordenadas. Si se pregunta por qué necesitamos 24 vértices, y no solo 8, es porque cada esquina pertenece a tres caras de diferentes colores, y un solo vértice debe tener un solo color específico; por lo tanto, crearemos tres copias de cada vértice en tres diferentes colores, uno para cada cara.

```
const positions = [
  // cara frontal
  -1.0, -1.0, 1.0,
   1.0, -1.0, 1.0,
   1.0, 1.0, 1.0,
  -1.0, 1.0, 1.0,

  // Cara posterior
  -1.0, -1.0, -1.0,
  -1.0, 1.0, -1.0,
   1.0, 1.0, -1.0,
   1.0, -1.0, -1.0,

  // cara superior
  -1.0, 1.0, -1.0,
  -1.0, 1.0, 1.0,
   1.0, 1.0, 1.0,
   1.0, 1.0, -1.0,

  // cara inferior
  -1.0, -1.0, -1.0,
   1.0, -1.0, -1.0,
   1.0, -1.0, 1.0,
  -1.0, -1.0, 1.0,

  // cara derecha
   1.0, -1.0, -1.0,
   1.0, 1.0, -1.0,
   1.0, 1.0, 1.0,
   1.0, -1.0, 1.0,

  // cara izquierda
  -1.0, -1.0, -1.0,
  -1.0, -1.0, 1.0,
  -1.0, 1.0, 1.0,
  -1.0, 1.0, -1.0,
];
```

Definir las posiciones de los vértices del cubo

Primero, construyamos el búfer de posición de vértice del cubo actualizando el código initBuffers(). Esto es más o menos lo mismo que para el plano cuadrado, pero algo más largo ya que hay 24 vértices (4 por lado):

Como hemos agregado un componente z a nuestros vértices, necesitamos actualizar el atributo numComponents de nuestro vertexPosition a 3.

```
// Dile a WebGL cómo sacar las posiciones de la posición
// buffer en el atributo vertexPosition
{
  const numComponents = 3;
  ...
  gl.vertexAttribPointer(
      programInfo.attribLocations.vertexPosition,
      numComponents,
      type,
      normalize,
      stride,
      offset);
  gl.enableVertexAttribArray(
      programInfo.attribLocations.vertexPosition);
}
```

Definir los colores de los vértices

También necesitamos construir una variedad de colores para cada uno de los 24 vértices. Este código comienza definiendo un color para cada cara, luego usa un bucle para ensamblar una matriz de todos los colores para cada uno de los vértices.

```
const faceColors = [
  [1.0, 1.0, 1.0, 1.0], // Cara frontal: blanco
  [1.0, 0.0, 0.0, 1.0], // Cara posterior: rojo
  [0.0, 1.0, 0.0, 1.0], // Cara superior: verde
  [0.0, 0.0, 1.0, 1.0], // Cara inferior: azul
  [1.0, 1.0, 0.0, 1.0], // Cara derecha: amarillo
  [1.0, 0.0, 1.0, 1.0], // Cara izquierda: púrpura
];

// Convierte la matriz de colores en una tabla para todos los vértices.

var colors = [];

for (var j = 0; j < faceColors.length; ++j) {
  const c = faceColors[j];

  // Repite cada color cuatro veces para los cuatro vértices de la cara
  colors = colors.concat(c, c, c, c);
}

const colorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
```

Definir la matriz de elementos

Una vez que se generan las matrices de vértices, necesitamos construir la matriz de elementos.

```
  const indexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);

  // Esta matriz define cada cara como dos triángulos, usando el
  // índices en la matriz de vértices para especificar cada triángulo
  // posición.

  const indices = [
    0,  1,  2,      0,  2,  3,    // frente
    4,  5,  6,      4,  6,  7,    // atras
    8,  9,  10,     8,  10, 11,   // arriba
    12, 13, 14,     12, 14, 15,   // abajo
    16, 17, 18,     16, 18, 19,   // derecha
    20, 21, 22,     20, 22, 23,   // izquierda
  ];

  // Ahora envíe la matriz de elementos a GL

  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
      new Uint16Array(indices), gl.STATIC_DRAW);

  return {
    position: positionBuffer,
    color: colorBuffer,
    indices: indexBuffer,
  };
}
```

La constante indices matriz define cada cara como un par de triángulos, especificando los vértices de cada triángulo como un índice en las matrices de vértices del cubo. Así, el cubo se describe como una colección de 12 triángulos.

Dibujando el cubo

A continuación, debemos agregar código a nuestra drawScene()función para dibujar usando el búfer de índice del cubo, agregando nuevos gl.bindBuffer() y gl.drawElements() llamadas:

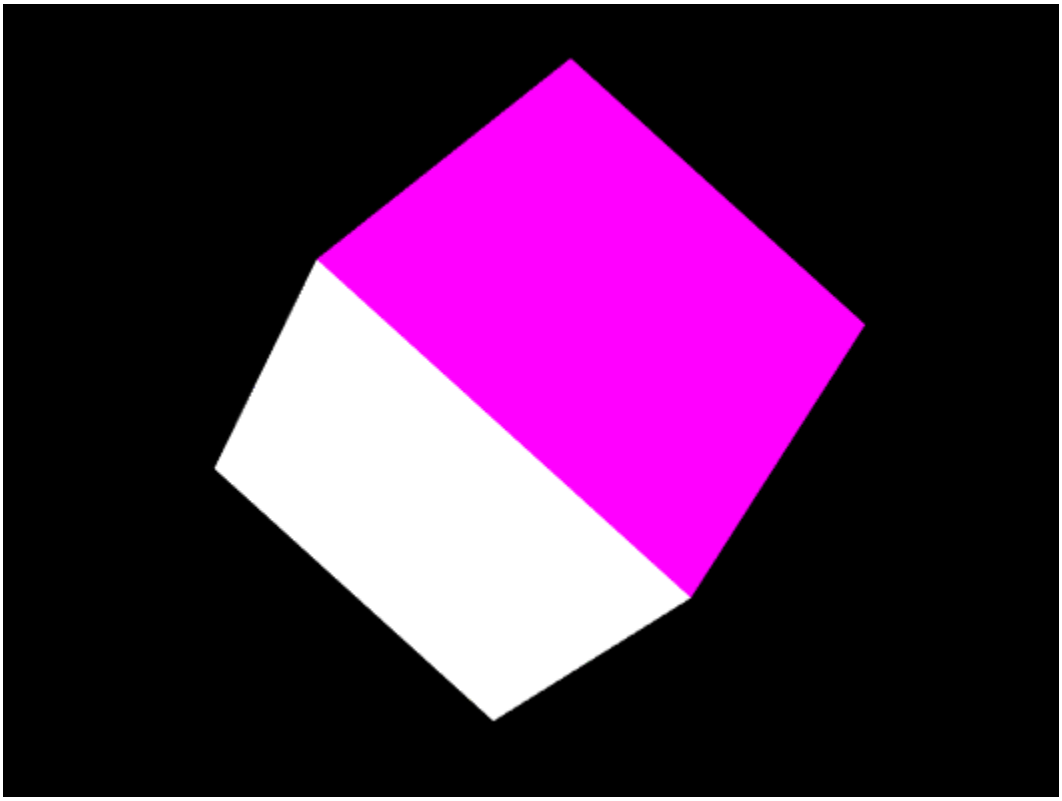
```
  // Dile a WebGL qué índices usar para indexar los vértices
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, buffers.indices);

...

  {
    const vertexCount = 36;
    const type = gl.UNSIGNED_SHORT;
    const offset = 0;
    gl.drawElements(gl.TRIANGLES, vertexCount, type, offset);
  }
```

Como cada cara de nuestro cubo está compuesta por dos triángulos, hay 6 vértices por lado, o 36 vértices totales en el cubo, aunque muchos de ellos son duplicados.

Finalmente, reemplacemos nuestra variable squareRotation por cubeRotation y agreguemos una segunda rotación alrededor del eje x:

```
mat4 . rotar (modelViewMatrix, modelViewMatrix, cubeRotation * .7 , [ 0 , 1 , 0 ]);
```

En este punto, ahora tenemos un cubo animado girando, sus seis caras de colores bastante vivos:



Codificar ejemplo de Cubo giratorio

```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>WebGL Demo</title>
    <link rel="stylesheet" href="webgl.css" type="text/css">
  </head>

  <body>
    <canvas id="glcanvas" width="640" height="480"></canvas>
  </body>

  <script src="gl-matrix.js"></script>
  <script src="webgl.js"></script>
</html>
```

Lab102.html

```css
canvas {
    border: 2px solid black;
    background-color: black;
}
video {
    display: none;
}
```

webgl.css

Scripts de uso de Web GL, vórtices y fragmentos:

webgl.js

Scripts de librería GL-Matrix de Moxilla.org

(ambos adjuntos en su laboratorio)

gl-matrix.js

Librería Three.JS

Hemos visto cómo lidiar con WebGL en el nivel más bajo. Y quizás, nos decepcionamos: demasiado trabajo, demasiado código, para la tarea más simple.

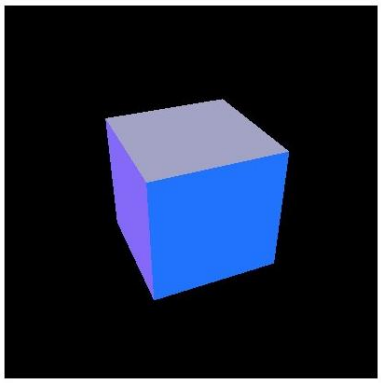
Como ya se mencionó, afortunadamente, hay bibliotecas disponibles para dibujar figuras en 3D que ahorrarán mucho tiempo, además de ser más fáciles. Probablemente el más conocido es three.js, cuya simplicidad es fascinante, más después de tratar con WebGL nativo.

Para usarlo, solo tenemos que descargar la última compilación minimizada (más de 500kb). Para este caso.

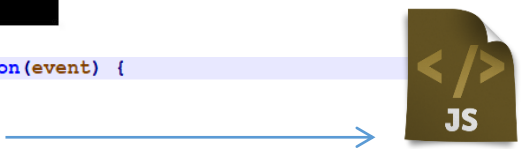
Disponible desde: https://threejs.org/build/three.min.js

E. Cubo giratorio 3D con Librería Three.JS

Dibujaremos un cubo 3D, que rotará en los ejes X e Y.



```
document.addEventListener('DOMContentLoaded', function(event) {
    window.requestAnimationFrame = (function() {
        return window.requestAnimationFrame;
    })();

    function animateScene() {
        requestAnimationFrame(animateScene);

        cube.rotation.y += 0.02;
        cube.rotation.x += 0.01;

        renderScene();
    }

    function createCube() {
        var cubeMaterials = [
            new THREE.MeshBasicMaterial({color:0x2173fd}),
            new THREE.MeshBasicMaterial({color:0xd5d918}),
            new THREE.MeshBasicMaterial({color:0xd2dbeb}),
            new THREE.MeshBasicMaterial({color:0xa3a3c6}),
            new THREE.MeshBasicMaterial({color:0xfe6b9f}),
            new THREE.MeshDepthMaterial({color:0x856af9})
        ];

        var cubeMaterial = new THREE.MeshFaceMaterial(cubeMaterials);
        var cubeGeometry = new THREE.BoxGeometry(2, 2, 2);

        cube = new THREE.Mesh(cubeGeometry, cubeMaterial);

        return cube;
    }
```



cubo3d.js

```
32
33      function startScene(cube) {
34          var canvas = document.getElementById('canvas');
35          render = new THREE.WebGLRenderer();
36
37          render.setClearColor(0x000000, 1);
38
39          var canvasWidth = canvas.getAttribute('width');
40          var canvasHeight = canvas.getAttribute('height');
41          render.setSize(canvasWidth, canvasHeight);
42
43          canvas.appendChild(render.domElement);
44
45          scene = new THREE.Scene();
46          var aspect = canvasWidth / canvasHeight;
47
48          camera = new THREE.PerspectiveCamera(45, aspect);
49          camera.position.set(0, 0, 0);
50          camera.lookAt(scene.position);
51          scene.add(camera);
52
53          cube.position.set(0, 0, -7.0);
54          scene.add(cube);
55      }
56
57      function renderScene() {
58          render.render(scene, camera);
59      }
60
61      var cube = createCube();
62      startScene(cube);
63      animateScene();
64      renderScene();
65
66  });
```

Creando el cubo

Crear el cubo (línea 15) es muy simple. Simplemente creamos una matriz donde definimos cada lado (6 en total), especificando el material de cada lado y el color del mismo.

Comenzando la escena

La función en la línea 33. Para comenzar la escena, también tenemos que configurar el render y la cámara.

Para el renderizado, instanciamos la THREE.WebGLRendererclase, establecemos un color claro (negro), definimos sus dimensiones (que ya están definidas en el elemento del lienzo en el HTML) y lo agregamos como un hijo al lienzo.

Para la cámara, en este caso, instanciamos THREE.PerspectiveCameraclase, pasando 2 parámetros:

- El primer parámetro define el campo visual de la cámara, en grados. Entonces, establecerlo en 45, sería como mirar al cubo que está frente a él.
- El segundo es la relación de aspecto. Para esto, generalmente, queremos establecer el ancho del elemento dividido por la altura. De lo contrario, la imagen se verá deformada.

Luego, colocamos la cámara en una posición determinada, le decimos que mire la posición de la escena instanciada y agregamos la cámara a la escena. Finalmente, podemos agregar el cubo a la escena.

Animando la escena

Para animar la escena (línea 6), debemos hacerlo solicitando el cuadro de animación al navegador. Para ello, pasamos la función a ejecutar como devolución de llamada, que es la de animar la escena.

Renderizando la escena

La representación de la escena consiste simplemente en llamar al método de representación del objeto de representación, pasando la escena y la cámara.

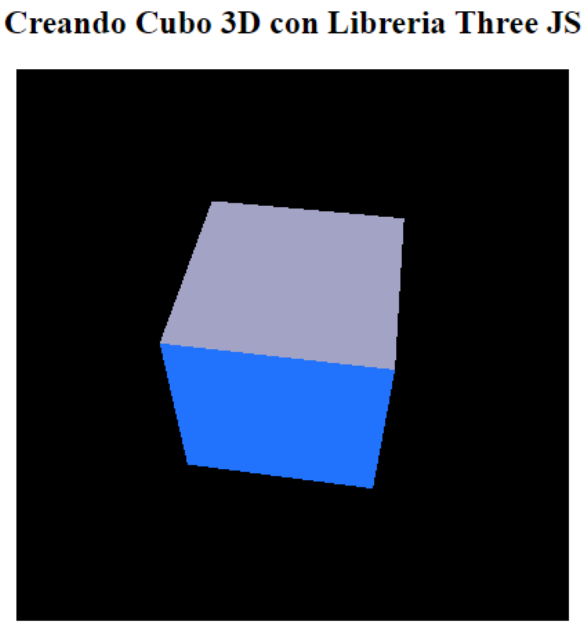
```
1   <!DOCTYPE HTML>
2   <html>
3       <head>
4           <title>Cubo 3D con Canvas WebGL</title>
5           <meta charset="utf-8">
6           <script src="three.min.js"></script>
7           <script src="cubo3d.js"></script>
8           <style>
9               #wrapper {
10                  width: 80%;
11                  margin: 0 auto;
12                  text-align: center;
13              }
14          </style>
15
16      </head>
17      <body>
18          <div id="wrapper">
19              <h2>Creando Cubo 3D con Libreria Three JS</h2>
20              <div id="canvas" width="400" height="400"></div>
21          </div>
22      </body>
23  </html>
```

Lab103.html

Este utiliza por supuesto la librería three JS versión minimizada (adjunto en su laboratorio)

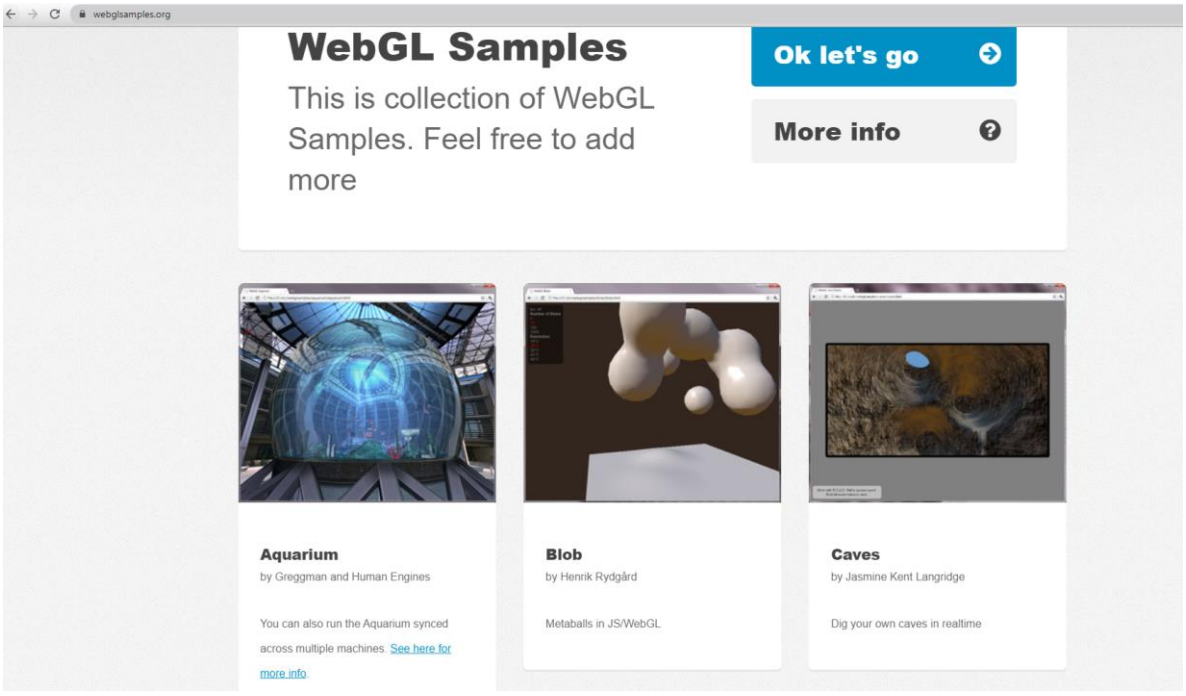
three.min.js

Resultado:



F. WebGL Avanzado

Puede navegar ejemplos avanzados que utilizan WebGL
https://webglsamples.org/



No es parte del laboratorio, pero es sugerido para ver las capacidades a alto nivel del WebGL dentro de Canvas HTML5 y conocer su potencial, a ser tomado de referencia a futuro.