Taller WebGL

Brayan Estrada Ramos

Eduardo Arango Serrano

1. ¿Qué es WebGL y cuál es su propósito principal en el desarrollo web?

WebGL, acrónimo de Web Graphics Library, es una API (Interfaz de Programación de Aplicaciones) de JavaScript que permite la renderización interactiva de gráficos 2D y 3D dentro de cualquier navegador web compatible, sin la necesidad de utilizar plugins externos. Está basada en OpenGL ES, una versión de la API de gráficos OpenGL diseñada específicamente para sistemas embebidos y dispositivos móviles, lo que facilita su integración y uso en entornos web.

El propósito principal de WebGL es permitir a los desarrolladores web aprovechar la potencia de procesamiento gráfico (GPU) disponible en los dispositivos modernos para la creación de gráficos complejos y detallados directamente en el navegador. Esto habilita una amplia gama de aplicaciones, desde juegos en línea con gráficos avanzados hasta simulaciones científicas, visualizaciones de datos, aplicaciones de realidad virtual y aumentada, y herramientas de diseño y modelado en 3D.

WebGL funciona integrando el código de JavaScript con el motor de renderizado del navegador, permitiendo así la ejecución de operaciones gráficas complejas de manera eficiente. Gracias a esta capacidad, los desarrolladores pueden crear experiencias visuales ricas e inmersivas que son accesibles universalmente a través de la web, sin requerir instalaciones adicionales por parte del usuario.

En el contexto del desarrollo web, WebGL representa un avance significativo en la capacidad de los navegadores para manejar contenido interactivo y visualmente intenso, lo que abre nuevas posibilidades para el diseño y la funcionalidad de las aplicaciones web. Su uso se ha extendido significativamente en diversos sectores, incluyendo el entretenimiento, la educación, la ingeniería y la ciencia, destacando su versatilidad y su papel crucial en la evolución de las experiencias web.

2. Crear un lienzo (canvas) en una página HTML y establecer su contexto de renderizado como WebGL.

```
<!DOCTYPE html>
<html>
<head>
    <title>Cuadrado en WebGL</title>
</head>
<body>
    <canvas id="webgl-canvas" width="400" height="400"></canvas>
    <script src="triangulo.js"></script>
</body>
</html>
```

Primero, es necesario definir un elemento `canvas` en el HTML, que servirá como el área de dibujo para los gráficos WebGL. Luego, se obtiene el contexto de renderizado de WebGL a través de JavaScript. En nuestro ejemplo, vamos cambiando entre “triangulo.js” y “cuadrado.js”

3. Describir brevemente el proceso de dibujo en WebGL.

El proceso de dibujo en WebGL implica varios pasos:

- **Inicialización:** Configurar el contexto WebGL, cargar y compilar los shaders, y establecer la información de los vértices (posiciones, colores, etc.).
- **Shaders:** Crear y compilar un shader de vértices y un shader de fragmentos. Los shaders son programas que se ejecutan en la GPU para determinar la posición y color de cada píxel.
- **Buffers:** Transferir los datos de vértices al GPU mediante buffers.

- **Dibujado:** Utilizar comandos de WebGL para dibujar los primitivos (puntos, líneas, triángulos) basados en los datos de los buffers.

4. Dibuja un cuadrado en el lienzo utilizando WebGL. Asegúrate de que el cuadrado tenga un color sólido y se visualice correctamente en el lienzo.

El siguiente código de JavaScript realiza las siguientes operaciones con el fin de crear un cuadrado:

1. Inicializa el contexto WebGL a partir del `canvas`.
2. Define los vértices del cuadrado.
3. Crea los shaders y el programa que se ejecutará en la GPU.
4. Asocia los vértices definidos a los shaders.
5. Establece el color de relleno del cuadrado.
6. Limpia el lienzo y dibuja el cuadrado

```
// Obtener el contexto WebGL del canvas
const canvas = document.getElementById('webgl-canvas');
const gl = canvas.getContext('webgl');

if (!gl) {
    alert('WebGL no está disponible');
}

// Aquí se define las posiciones de los vértices
const vertices = new Float32Array([
    -0.5,  0.5, // Vértice superior izquierdo
     0.5,  0.5, // Vértice superior derecho
    -0.5, -0.5, // Vértice inferior izquierdo
     0.5, -0.5  // Vértice inferior derecho
```

```
]);

// Se Crea un buffer para almacenar los vértices del cuadrado
const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

// Se asocia el buffer de vértices al atributo de posición en
// el shader
const vertexShaderSource = `
attribute vec2 position;
void main() {
    gl_Position = vec4(position, 0.0, 1.0);
}`;

const fragmentShaderSource = `
precision mediump float;
void main() {
    gl_FragColor = vec4(0.0, 0.5, 0.0, 1.0); // Color verde
    //oscuro
}`;

// Crear y compilar los shaders
const vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);

const fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentShaderSource);
gl.compileShader(fragmentShader);

// Crear el programa de shader y enlazarlo
const shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);
```

```
gl.useProgram(shaderProgram);

// Asociar el buffer de vértices con el atributo de posición
// en el shader
const positionLocation = gl.getAttribLocation(shaderProgram,
 'position');
gl.enableVertexAttribArray(positionLocation);
gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false
, 0, 0);

// Dibujar el cuadrado
gl.clearColor(0.0, 0.0, 0.0, 1.0); // Fondo negro
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
```

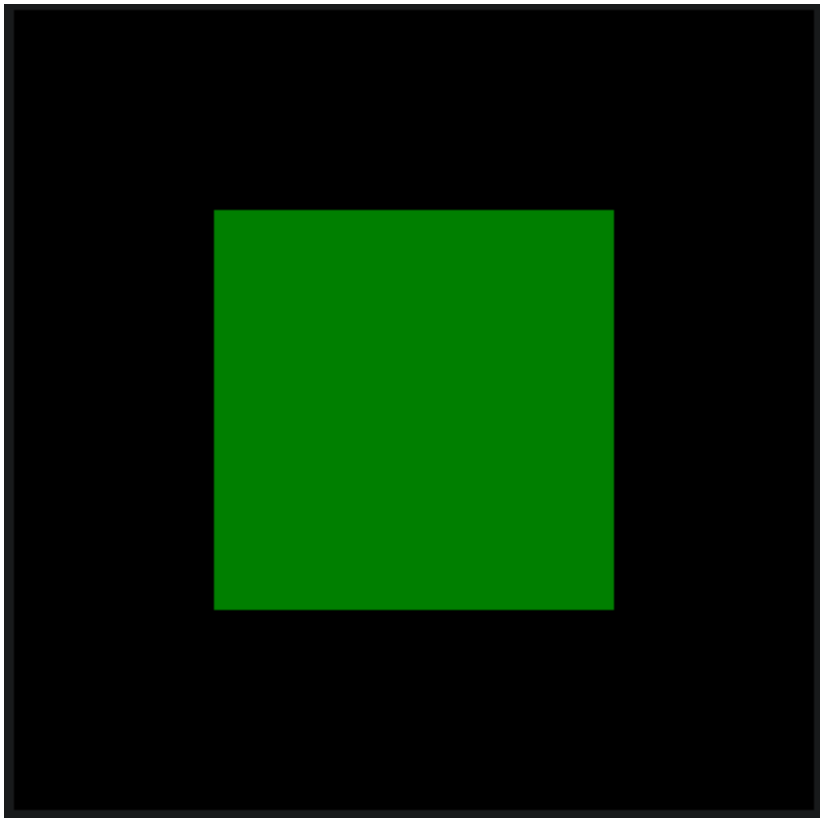

Imagen del cuadrado generado.

5. ¿Qué son los vértices y los fragmentos en WebGL y cómo se relacionan con el proceso de renderizado?

En el proceso de renderizado, los vértices son procesados primero por el shader de vértices, que determina su posición final en la pantalla. Luego, el rasterizador de WebGL interpola los vértices para formar fragmentos, los cuales son procesados por el shader de fragmentos para determinar su color final.

- **Vértices:** Son puntos en un espacio tridimensional que definen la forma de un objeto gráfico. Cada vértice tiene atributos como posición, color, y coordenadas de textura.
- **Fragmentos:** Son los 'píxeles en potencia' que resultan de la rasterización de primitivas gráficas (triángulos, líneas, puntos). Los shaders de fragmentos determinan el color y otros atributos de estos fragmentos antes de que se escriban en el framebuffer.

6. Dibuja un triángulo en el lienzo utilizando WebGL. Experimenta con diferentes colores y posiciones para los vértices del triángulo.

En el siguiente código, se crean diferentes posiciones y colores para el triángulo dibujado. Si desea verlo, descomente las lineas donde se cambian los vértices o los colores.

```
// Obtener el contexto WebGL del canvas
const canvas = document.getElementById('webgl-canvas');
const gl = canvas.getContext('webgl');

if (!gl) {
```

```
    alert('WebGL no está disponible');
}

// Ejemplos de posiciones para los vértices del triángulo
// Descomenta uno de los siguientes bloques para probar
// diferentes posiciones

// Posición 1 (Centrado, Descomentado)
const vertices = new Float32Array([
    -0.5, -0.5,   // Vértice inferior izquierdo
     0.0,  0.5,   // Vértice superior
     0.5, -0.5    // Vértice inferior derecho
]);

// Posición 2 (Hacia la esquina superior derecha, Comentado)
// const vertices = new Float32Array([
//     0.0,  0.0,   // Vértice inferior izquierdo
//     0.5,  1.0,   // Vértice superior
//     1.0,  0.0    // Vértice inferior derecho
// ]);

// Posición 3 (Hacia la esquina inferior izquierda,
//Comentado)
// const vertices = new Float32Array([
//    -1.0, -1.0,   // Vértice inferior izquierdo
//    -0.5, -0.5,   // Vértice superior
//    -1.0,  0.0    // Vértice inferior derecho
// ]);

// Crear un buffer para almacenar los vértices del triángulo
const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

// Crear y compilar los shaders
const vertexShaderSource = `
```

```
attribute vec2 position;
void main() {
    gl_Position = vec4(position, 0.0, 1.0);
}`;

// Ejemplos de colores para el triángulo
// Descomenta uno de los siguientes bloques para probar
//diferentes colores

// Color 1: Verde oscuro (Descomentado)
const fragmentShaderSource = `
precision mediump float;
void main() {
    gl_FragColor = vec4(0.0, 0.5, 0.0, 1.0); // Verde oscuro
}`;

// Color 2: Azul (Comentado)
// const fragmentShaderSource = `
// precision mediump float;
// void main() {
//     gl_FragColor = vec4(0.0, 0.0, 1.0, 1.0); // Azul
// }`;

// Color 3: Rojo (Comentado)
// const fragmentShaderSource = `
// precision mediump float;
// void main() {
//     gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); // Rojo
// }`;

const vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);

const fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentShaderSource);
```

```
gl.compileShader(fragmentShader);

// Crear el programa de shader y enlazarlo
const shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
gl.linkProgram(shaderProgram);
gl.useProgram(shaderProgram);

// Asociar el buffer de vértices con el atributo de posición
//en el shader
const positionLocation = gl.getAttribLocation(shaderProgram,
'position');
gl.enableVertexAttribArray(positionLocation);
gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false,
0, 0);

// Dibujar el triángulo
gl.clearColor(0.0, 0.0, 0.0, 1.0); // Fondo negro
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

IMAGENES GENERADAS

1.

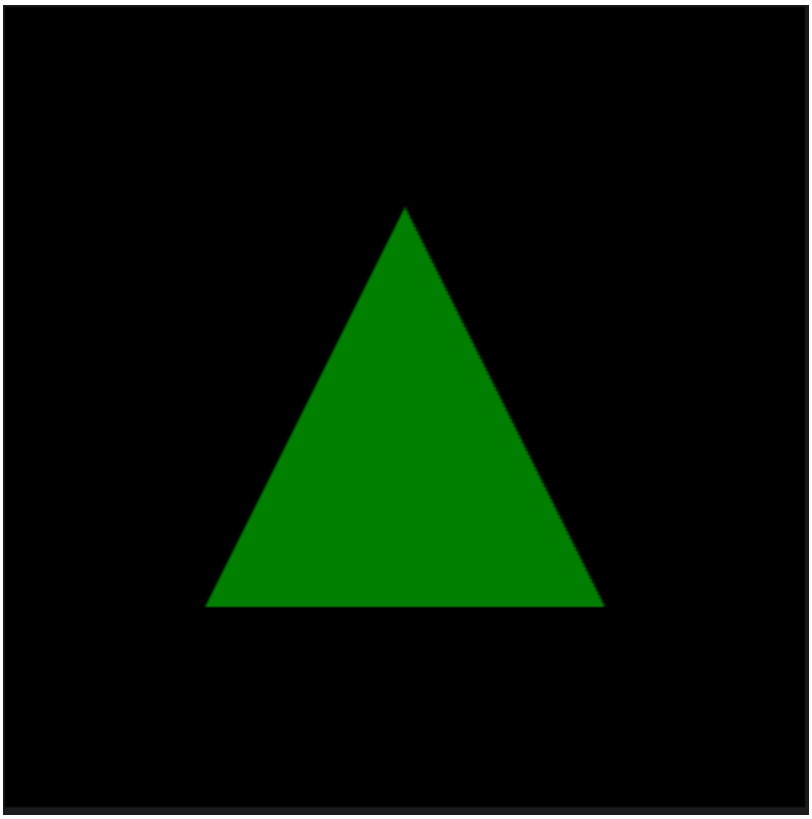

Imagen Triangulo Posición 1 Color Verde

2.

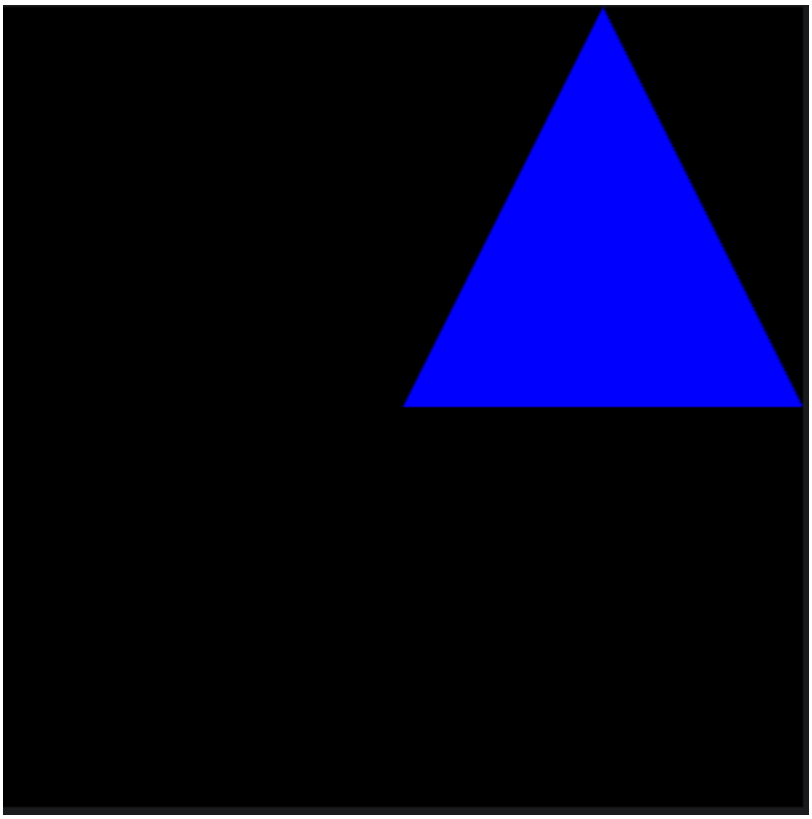

Imagen Triángulo Posición 2 Color Azul.

3.

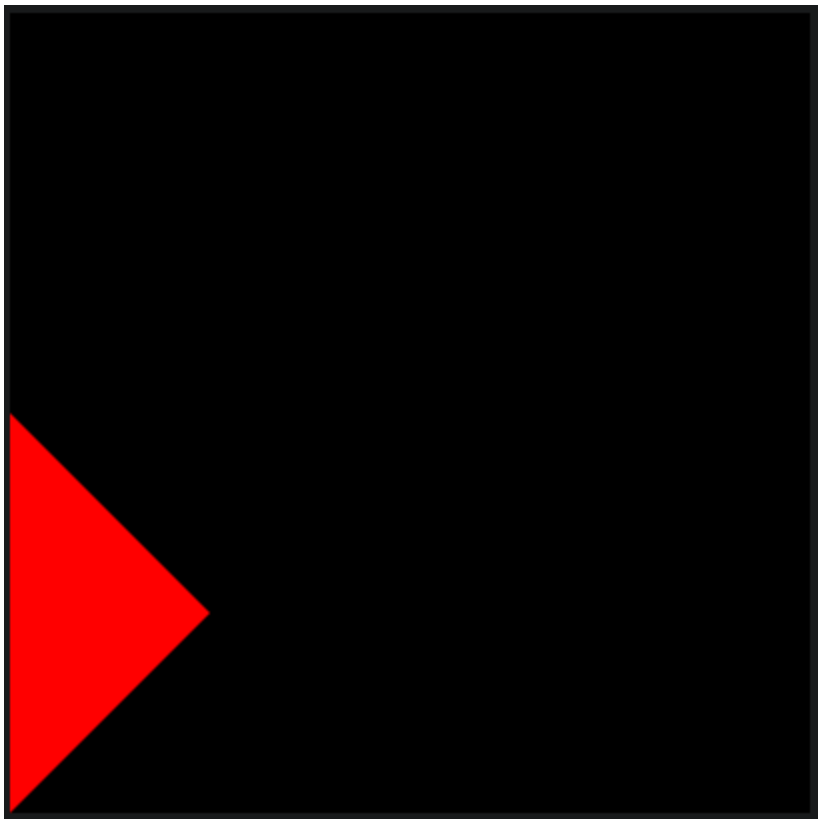

Imagen Triángulo Posición 3 Color Rojo