Hello World in WebGL

We’ll start from the basics and end with your first rendered object - a triangle.

Recap: What is WebGL?

- WebGL (Web Graphics Library) is a JavaScript API for rendering interactive 2D and 3D graphics.
- It runs in the browser without plugins.
- We will use WebGL 2.0 (spec) which is based on OpenGL ES 3.0.
- Defined by the Khronos Group.

Recap: Why do we use WebGL?

- Runs on all major browsers.
- Allows easy understanding of graphics programming.

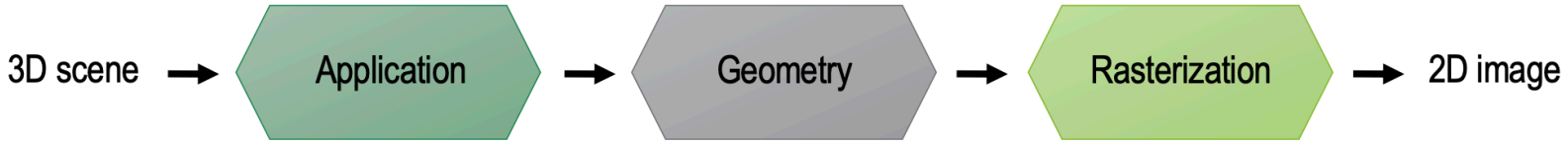
Explanation

👀 means that the code is already in the repository and you just need to look at it.
📋 means you can copy-paste the code and it should work.
📝 means that you need to create a new file
⭕ indicates that you need to do more than just copy-paste the code.

In any case you need to understand what you are doing.

The Rendering Pipeline

1. **Application** — your JavaScript code.
2. **Geometry** — defines shapes (points, lines, triangles).
    i. **Vertex Shader** — processes each vertex.
3. **Rasterization** — converts geometry to pixels.
    i. **Fragment Shader** — determines pixel color.
4. **2D image** — we need to display the result.



Application

WebGL Setup

👀 Start with an HTML canvas:

```
<canvas id="myCanvas" width="800" height="600"></canvas>
```

👀 Connect JavaScript using:

```
<script src="script.js" type="module"></script>
```

Initializing WebGL2 context

⭕ Get the canvas element and create a **WebGL2** context.

```
const canvas = document.getElementById("myCanvas");
const gl = canvas.getContext("webgl2");

if (!gl) {
  console.error("WebGL2 not supported");
}
```

Geometry

Define coordinates

WebGL uses normalized device coordinates (NDC) ranging from -1 to 1 in both x and y directions. 📋

```
const coordinates = [
-0.2, -0.3,
0.3, -0.3,
0.3, 0.6
];
```

These points form one triangle.

Uploading geometry to GPU

- Create an empty buffer object to store the vertex points
- Connect the empty buffer object to the GL context
- Load the vertices into the GL's connected buffer using the right data type (here: `Float32Array`) 📋

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

Fetching shader source code

- Use the `fetch` API to load the shader source code from external files. 📋

```
// Create a variable to fetch the data for our shaders
const shaderSources = await fetchShaderTexts(
    "./vertex.glsl",
    "./fragment.glsl");

console.log(shaderSources);
```

⭕ You need to import the `fetchShaderTexts` function that fetches the shader source code from the files. You find it in the `utils.zip` file.

```
import { fetchShaderTexts } from "../utils/fetchShader.js";
```

Writing the vertex shader

📝 Vertex Shader (`vertex.glsl`):

```
#version 300 es
in vec4 vertex_points;

void main() {
    gl_Position = vertex_points;
}
```

Explanation of the vertex shader

- `#version 300 es` specifies the version of GLSL ES.
- `in vec4 vertex_points;` declares an input (`in`) variable (a.k.a. *attribute*) of type `vec4` (a 4D vector) with the name `vertex_points`.
- The `main()` function is the entry point of the shader program.
- `gl_Position` is a built-in variable that specifies the position of the vertex in clip space.

Writing the fragment shader

📝 Fragment Shader (fragment.glsl):

```
#version 300 es
precision mediump float;

out vec4 outColor;  // you can pick any name

void main() {
   outColor = vec4(0.0, 0.0, 0.5, 1.0);
}
```

Explanation of the fragment shader

- `precision mediump float;` sets the default precision for floating-point variables.
- `out vec4 outColor;` declares an output (`out`) variable of type `vec4` named `outColor` .
  - in WebGL 2.0, you can pick any name for the output variable, while in WebGL 1.0, the output variable must be `gl_FragColor` .
  - in WebGL 1.0, you use `varying` instead of `out` .
- The `main()` function is the entry point of the shader program.
- `outColor = vec4(0.0, 0.0, 0.5, 1.0);` sets the color

Initialize shader programs

- Create a vertex shader and a fragment shader
- Compile the shaders
- Link the shaders to a program
- Use the program

Compiling shaders

📋 Get the vertex shader source code and compile it:

```
const vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, shaderSources.vertexShaderText);
gl.compileShader(vertexShader);
let success = gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS);
if (success) {
    console.log('Vertex Shader successfully compiled.');
} else {
    console.error('Vertex Shader did not compile.');
    console.log(gl.getShaderInfoLog(vertexShader));
}
```

⭕ Repeat similarly for the fragment shader

📋 Attach link and use shaders

- Create a carry-out container that will pass the shader functions to the GPU
- Attach the vertex and fragment shaders to the program
- Link the program
- Use the program

```
const program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
gl.useProgram(program);
```

Connecting Buffers and Attributes (1)

The vertex shader needs to know where to find the vertex data. We do this by connecting the buffer with the attribute in the shader.

- 📋 Get the location of the attribute in the shader

```
const pointsAttributeLocation = gl.getAttribLocation(program, "vertex_points");
```

- ⭕ Connect the attribute to the points data currently in the buffer object

```
// TODO: Define variables size, type, normalize, stride, and offset
gl.vertexAttribPointer(pointsAttributeLocation, size, type,
normalize, stride, offset);
```

Connecting Buffers and Attributes (2)

- ⭕ To find the correct parameter values for the `vertexAttribPointer()` call:
  - `size` : number of components per vertex
  - `type` : data type of each component (we used it similarly in the buffer)
  - `normalize` : whether to normalize the data (check if needed)
  - `stride` : number of bytes to skip before the next set of values
  - `offset` : number of bytes to skip from the start of the buffer

Connecting Buffers and Attributes (3)

- 📋 Send the points data to the GPU by enabling the attribute

```
gl.enableVertexAttribArray(pointsAttributeLocation);
```

Drawing to the Screen

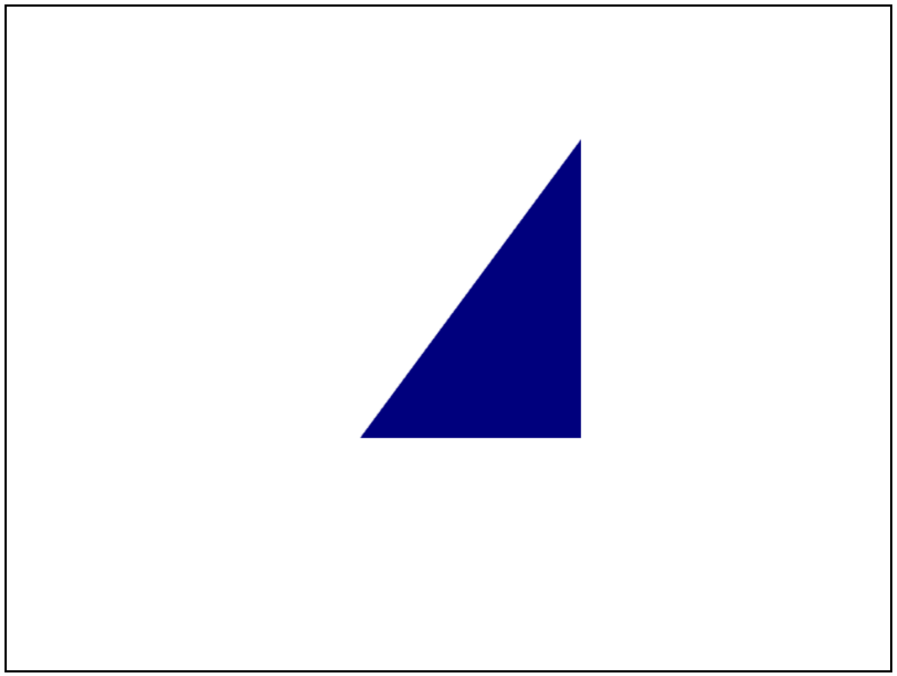
- Clear the canvas and make the background white
- Clear the color buffer
- Draw the points on the screen

```
gl.clearColor(1, 1, 1, 1);
gl.clear(gl.COLOR_BUFFER_BIT);

const mode = gl.TRIANGLES;
const first = 0;
const count = coordinates.length / 2; // 2 coordinates per point
gl.drawArrays(mode, first, count);
```

Result: Hello WebGL World

You should see a dark blue triangle on the canvas.



🎉 Congratulations, you've created your first WebGL scene!