Hello 3D in WebGL

We extend the previous example to create two 3D triangles.

Recap: What have we learned?

- We created a simple WebGL application.
- We learned how to prepare the data for the GPU.
- We learned how to use simple shaders to render a 2D triangle.
- We learned to execute the drawing commands.

Recap: What did we not do?

- We did not use 3D coordinates.
- We did not use 3D transformations.
- We did not use per vertex colors.

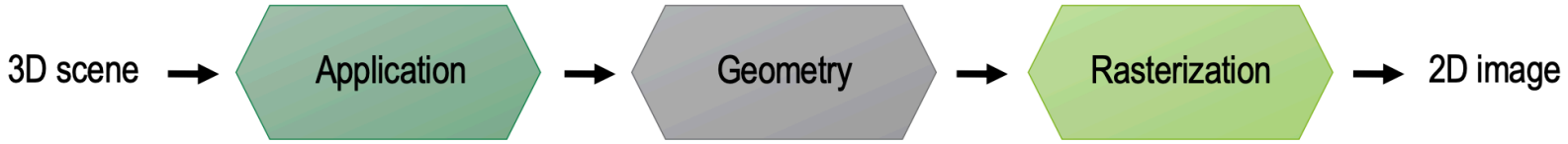
Explanation

👀 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.
❌ indicates that you need to replace the old code with something new.

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>
```

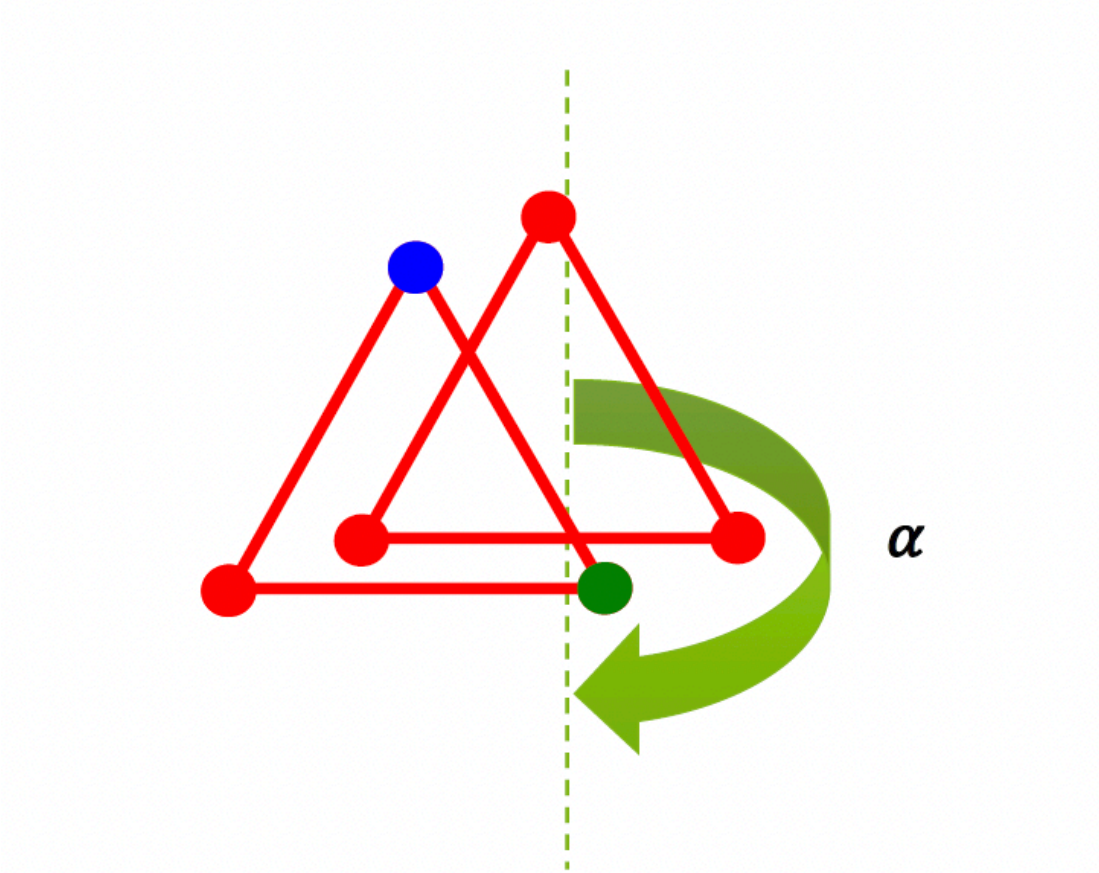
Initializing WebGL2

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

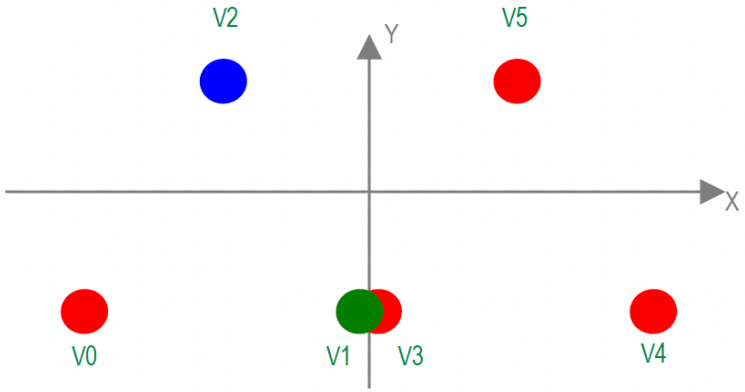
```
const canvas = document.getElementById("myCanvas");
const gl = canvas.getContext("webgl2");

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

Geometry

The plan: two triangles, scene rotated by $\alpha = 30^\circ$



The data table



ID	X	Y	Z	R	G	B
V0	-1.0	-0.5	-2.0	1.0	0.0	0.0
V1	0.0	-0.5	-2.0	0.0	1.0	0.0
V2	-0.5	0.5	-2.0	0.0	0.0	1.0
V3	0.0	-0.5	-3.0	1.0	0.0	0.0
V4	1.0	-0.5	-3.0	1.0	0.0	0.0
V5	0.5	0.5	-3.0	1.0	0.0	0.0

Define coordinates

We now use 3D coordinates and define two triangles ❌ 📋

```
const twoTrianglesVertices = [
    // front triangle
    -1.0, -0.5, -2.0,
    0.0, -0.5, -2.0,
    -0.5, 0.5, -2.0,

    // back triangle
    0.0, -0.5, -3.0,
    1.0, -0.5, -3.0,
    0.5, 0.5, -3.0,
];
```

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
- What do we need to change in the old code? ⭕

New: Add values for the color

- 📋 Create a new array for the colors

```
const twoTrianglesColors = [
    // front triangle
    1.0,  0.0,  0.0,  1.0,
    0.0,  1.0,  0.0,  1.0,
    0.0,  0.0,  1.0,  1.0,

    // back triangle
    1.0,  0.0,  0.0,  1.0,
    1.0,  0.0,  0.0,  1.0,
    1.0,  0.0,  0.0,  1.0
];
```

Uploading color to GPU

- 📋 Create a buffer object to store the vertex colors

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

Fetching shader source code

- Same as before with the `fetchShaderTexts()` function, but we need to update the shader source code to use 3D coordinates and colors.

Writing the vertex shader

📋 Vertex Shader (vertex.glsl):

```
#version 300 es

in vec4 aPosition;
in vec4 aColor;
uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;
out vec4 vFragColor;

void main() {
    gl_Position = uProjectionMatrix * uModelViewMatrix * aPosition;
    vFragColor = aColor;
}
```

Note that it is necessary to add `#version 300 es` as **first** line to indicate that we are using WebGL 2.0.

Understanding the vertex shader (Variables)

- `in vec4 aPosition;` — input vertex position as **attribute** with keyword `in`
- `in vec4 aColor;` — input vertex color
- `uniform mat4 uModelViewMatrix;` — model-view matrix as **uniform**
- `uniform mat4 uProjectionMatrix;` — projection matrix
- `out vec4 vFragColor;` — output color to fragment shader as **varying** variable with keyword `out`

Understanding the vertex shader (main function)

- `gl_Position = uProjectionMatrix * uModelViewMatrix * aPosition;` — transform vertex position
- `vFragColor = aColor;` — pass color to fragment shader

Writing the fragment shader

📋 Fragment Shader (fragment.glsl):

```
#version 300 es
precision mediump float;

in vec4 vFragColor;
out vec4 outColor;  // you can pick any name

void main() {
   outColor = vFragColor;
}
```

Note that it is necessary to add `#version 300 es` as **first** line to indicate that we are using WebGL 2.0. And we need to add `precision mediump float;` to define the precision of the floating point numbers.

Understanding the fragment shader

- `in vec4 vFragColor;` — input fragment color as **attribute** with keyword `in`
- `out vec4 outColor;` — output color to framebuffer as **varying** variable with keyword `out`
- `outColor = vFragColor;` — pass color to framebuffer (draw to screen)

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

❌ 📋 Create a new function (cleaner code) that gets the shader source code and compiles it:

```
function compileShader(shader, shaderName) {
    gl.compileShader(shader);
    if (gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        console.log(shaderName + ' shader compiled successfully.');
    } else {
        console.error('Error compiling ' + shaderName + ' shader: ' + gl.getShaderInfoLog(shader));
        gl.deleteShader(shader);
    }
}
```

⭕ Call this function for vertex and 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

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. We start with the position attribute.

❌ 📋 Create a new function (cleaner code) that connects the attributes to the shader:

```
function connectShaderAttributes(program, attribName, buffer, size, type, normalized, stride, offset) {
    const attribLocation = gl.getAttribLocation(program, attribName);
    gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
    gl.vertexAttribPointer(attribLocation, size, type, normalized, stride, offset);
    gl.enableVertexAttribArray(attribLocation);
}
```

⭕ Call this function for both the position and the color attribute

- 👀 You find the attribute names in the vertex shader

Connecting uniforms (1)

The next step is to connect the uniforms that contain the transformation data to the shader.

- 📋 First we need to create the projection matrix

```
const projMatrixLocation = gl.getUniformLocation(program, 'uProjectionMatrix');
const projectionMatrix = mat4.create();
const fieldOfView = 45 * Math.PI / 180; // in radians
const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
const zNear = 0.1;
const zFar = 100.0;
// note: glmatrix.js always has the first argument as the destination to receive the result.
mat4.perspective(projectionMatrix, fieldOfView, aspect, zNear, zFar);
mat4.translate(projectionMatrix, // destination matrix
    projectionMatrix, // matrix to translate
    [1.0, 0.0, 0.0]);
console.log('ProjectionMatrix: %s', mat4.str(projectionMatrix));
```

Connecting uniforms (2)

Note that we need to import the `mat4` module from `gl-matrix` to use the matrix functions. It is part of the already used `utils.zip` and you can do this by adding the following line at the top of your JavaScript file:

```
import * as mat4 from "./utils/mat4.js"
```

Connecting uniforms (3)

- 📋 Now we need to create the model-view matrix

```
const modelViewMatrixLocation = gl.getUniformLocation(program, 'uModelViewMatrix');
const modelViewMatrix = mat4.create();
mat4.rotate(modelViewMatrix, // destination matrix
    modelViewMatrix, // matrix to rotate
    30 * Math.PI / 180,// amount to rotate in radians
    [0, 1, 0]); // axis to rotate around (Y)
console.log('ModelviewMatrix: %s', mat4.str(modelViewMatrix));
```

Connecting uniforms (4)

📋 Finally, we need to send the matrices to the GPU

```
gl.uniformMatrix4fv(projMatrixLocation, false, projectionMatrix);
gl.uniformMatrix4fv(modelViewMatrixLocation, false, modelViewMatrix);
```

⭕ A task for later: Set the second argument to `true` for the `modelViewMatrix` and find out what happens.

Drawing to the Screen

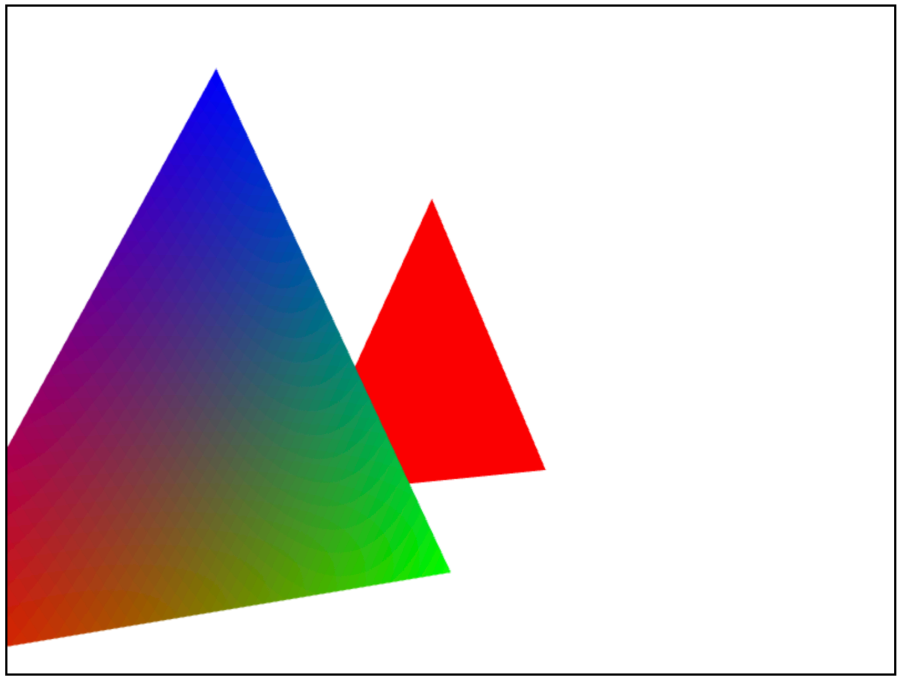
📋 Now we need to add a depth test to the WebGL context to ensure that the triangles are rendered correctly in 3D space. We need to clear both the color buffer and the depth buffer before drawing the triangles.

```
gl.clearColor(1, 1, 1, 1);
gl.enable(gl.DEPTH_TEST);
gl.depthFunc(gl.LEQUAL);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

⭕ Update the `count` to match the correct number of vertices in the `twoTrianglesVertices` array.

Result: Hello WebGL World

You should see two triangles on the canvas.



🎉 Congratulations, you've created your first WebGL 3D scene!