WebGl

2D Graphics in WebGL
The first drawings - two triangles.

Task 1
Setting canvas without getting any context.
We see a white canvas on a gray background.

```
<!DOCTYPE html>
…
<head>
        …
          <style>
            body {
              margin: 0px auto;
              text-align: center;
              background-color: grey;
            }
            canvas {background-color: white}
          </style>
</head>
<body>
        <h3>WebGl</h3>
        <h2>Setting style</h2>
        <canvas id="my-canvas" width="400" height="300">
          Your browser does not support the HTML5 canvas element.
        </canvas>
</body>
</html>
```

Task 2
Getting Context
Setting canvas with getting webgl context.
The initial setting is a white canvas on a gray background as in task 1.
If the JS script was executed correctly then the canvas background should be changed to green.
The code shows that we can use the following commands interchangeably:

- getElementById,
- querySelector.

```
<html>
<head>
  <title>Initialize WebGL context</title>
  <style>
    body {
      margin: 0px auto; text-align: center; background-color: grey;
    }
    canvas {background-color: white}
  </style>
  <script>
    window.onload = setupWebGL;

    function setupWebGL() {
      //const canvas = document.getElementById("my-canvas");
      //const canvas = document.querySelector("#my-canvas");
      const canvas = document.querySelector("canvas");
      const gl = canvas.getContext("webgl");

      if(gl){
```

```
            //set the canvas clear color to green
            gl.clearColor(0.0, 1.0, 0.0, 1.0);
            gl.clear(gl.COLOR_BUFFER_BIT);
         }else{
            alert( "Unable to initialize WebGL. Your browser or machine may not support it.");
         }
      }
   </script>
</head>
<body>
<h2>WebGl 1-01</h2>
<h3>initialize WebGL context</h3>
<canvas id="my-canvas" width="400" height="300">
   Your browser does not support the HTML5 canvas element.
</canvas>
</body>
</html>
```

- The first thing we do here is obtain a reference to the canvas, assigning it to a variable named canvas.
- Once we have the canvas, we try to get a WebGLRenderingContext for it by calling getContext and passing it the string "gl".
- If the browser does not support webgl getContext will return null in which case we will display a message to the user and exit.
- If the context is successfully initialized, the variable gl is our reference to it.
- In this case, we set the clear color to green, and clear the context to that color (redrawing the canvas with the background color).

At this point, you have enough code that the WebGL context should successfully initialize, and we should wind up with a big green, empty box, ready and waiting to receive content.

Drawing the scene

The most important thing to understand before we get started is that even though we're only rendering a square plane object in this example, we're still drawing in 3D space.
It's just we're drawing a square and we're putting it directly in front of the camera perpendicular to the view direction.
We need to define the shaders that will create the color for our simple scene as well as draw our object.
These will establish how the square plane appears on the screen.

The shaders

A shader is a program, written using the OpenGL ES Shading Language (GLSL), that takes information about the vertices that make up a shape and generates the data needed to render the pixels onto the screen: namely, the positions of the pixels and their colors.

There are two shader functions run when drawing WebGL content:

- the vertex shader
- the fragment shader.

You write these in GLSL and pass the text of the code into WebGL to be compiled for execution on the GPU. Together, a set of vertex and fragment shaders is called a shader program.

Let's take a quick look at the two types of shader, with the example in mind of drawing a 2D shape into the WebGL context.

Vertex shader

Each time a shape is rendered, the vertex shader is run for each vertex in the shape.
Its job is to transform the input vertex from its original coordinate system into the **clipspace** coordinate system used by WebGL, in which each axis has a **range from -1.0 to 1.0**, regardless of aspect ratio, actual size, or any other factors.
The vertex shader must perform the needed transforms on the vertex's position, make any other adjustments or calculations it needs to make on a per-vertex basis, then return the transformed vertex by saving it in a special variable provided by GLSL, called **gl_Position**.

The vertex shader can, as needed, also do things like determine the coordinates within the face's texture of the **texel** (texture element, texture pixel) to apply to the vertex, apply the normals to determine the lighting factor to apply to the vertex, and so on. This information can then be stored in:

- varyings

or

- attributes

as appropriate to be shared with the fragment shader.

Vertex shader below receives vertex position values from an attribute we define called aVertexPosition.
gl_Position is set to the result.

```
<script id="shader-vs" type="x-shader/x-vertex">
   attribute vec3 aVertexPosition;
   void main(void){
      gl_Position = vec4(aVertexPosition, 1.0);
   }
</script>
```

It's worth noting that we're using a vec4 attribute for the vertex position, which doesn't actually use a 4-component vector; that is, it could be handled as a vec2 or vec3 depending on the situation.
But when we do our math, we will need it to be a vec4, so rather than convert it to a vec4 every time we do math, we'll just use a vec4 from the beginning.
This eliminates operations from every calculation we do in our shader.

Fragment shader
The fragment shader is called once for every pixel on each shape to be drawn, after the shape's vertices have been processed by the vertex shader.
Its job is to determine the color of that pixel by figuring out which texel (that is, the pixel from within the shape's texture) to apply to the pixel, getting that texel's color, then applying the appropriate lighting to the color.
The color is then returned to the WebGL layer by storing it in the special variable **gl_FragColor**.
That color is then drawn to the screen in the correct position for the shape's corresponding pixel.
In this case, we're simply returning white every time, since we're just drawing a white square, with no lighting in use.

```
<script id="shader-fs" type="x-shader/x-fragment">
   void main(void){
      gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
   }
</script>
```

Initializing the shaders
Now that we've defined the source code for two shaders we need:

- create them,
- pass them to WebGL,
- compile them,
- link them together.

The code below creates the two shaders by calling loadShader(), passing the type and source for the shader.
It then creates a program, attaches the shaders and links them together.
If compiling or linking fails the code displays an alert.

Task 3
We put everything in JavaScript function “initWebGL”.
This function will be called when the “onload” event occurs.
The onload event occurs when an object has been loaded.
“onload” is most often used within the <body> element to execute a script once a web page has completely loaded all content (including images, script files, CSS files, etc.).

```
<body onload="initWebGL()">
	<canvas id="my-canvas" width="400" height="300">
		Your browser does not support the HTML5 canvas element.
	</canvas>
</body>
```

Function “initWebGL” we will put in <header> tag.
Function “initWebGL” will do all the work, dividing it into four functions:

- setupWebGL()
- initShaders()
- setupBuffers()
- drawScene()

```
<head>
	<title>A Triangle 0</title>
	<style>
		body{
			background-color: grey;
			margin: 0px auto; text-align: center;
		}
		canvas{ background-color: white; }
	</style>
	<script id="shader-vs" type="x-shader/x-vertex">
		attribute vec3 aVertexPosition;
		void main(void) {
			gl_Position = vec4(aVertexPosition, 1.0);
		}
	</script>
	<script id="shader-fs" type="x-shader/x-fragment">
		void main(void) {
			gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
		}
	</script>
	<script>
		var gl = null,
		canvas = null,
		glProgram = null,
		fragmentShader = null,
		vertexShader = null;
		var vertexPositionAttribute = null,
		trianglesVerticeBuffer = null;

		function initWebGL(){
			canvas = document.getElementById("my-canvas");
			try{
				gl = canvas.getContext("webgl");
			}catch(e){}
			if(gl){
				setupWebGL();
```

```
                        initShaders();
                        setupBuffers();
                        drawScene();
                    }else{
                        alert( "Error: Your browser does not appear to" +
                        "support WebGL.");
                    }
                }

                function setupWebGL(){
                    //set the clear color to a shade of green
                    gl.clearColor(0.1, 0.5, 0.1, 1.0);
                    gl.clear(gl.COLOR_BUFFER_BIT);
                }
                function initShaders(){}
                function setupBuffers(){}
                function drawScene(){}
            </script>
    </head>
```