WebGl

2D Graphics in WebGL
The first drawings - two triangles, part 2

Task 4
We define makeShader function:
We will use makeShader function in initShader function.

```
function makeShader(src, type){
   //compile the vertex shader
   var shader = gl.createShader(type);
   gl.shaderSource(shader, src);
   gl.compileShader(shader);
   if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert("Error compiling shader: " + gl.getShaderInfoLog(shader));
      return null;
   }
   return shader;
}
```

The **makeShader()** function use the WebGL context, the source code, and the shader type, then creates and compiles the shader as follows:

- A new shader is created by calling gl.createShader().
- The shader's source code is sent to the shader by calling gl.shaderSource().
- Once the shader has the source code, it's compiled using gl.compileShader().
- To check to be sure the shader successfully compiled, the shader parameter gl.COMPILE_STATUS is checked. To get its value, we call gl.getShaderParameter(), specifying the shader and the name of the parameter we want to check (gl.COMPILE_STATUS). If that's false, we know the shader failed to compile, so show an alert with log information obtained from the compiler using gl.getShaderInfoLog(), then delete the shader and return null to indicate a failure to load the shader.
- If the shader was loaded and successfully compiled, the compiled shader is returned to the caller.

Task 5
Now we initialize shaders, create program with those shaders, and use program:
We will do it by initShader() function:

```
function initShader() {
   //get shader source
   const fs_source = document.getElementById('shader-fs').innerHTML;
   const vs_source = document.getElementById('shader-vs').innerHTML;

   //compile shaders
   vertexShader = makeShader(vs_source, gl.VERTEX_SHADER);
   fragmentShader = makeShader(fs_source, gl.FRAGMENT_SHADER);

   //create program
   glProgram = gl.createProgram();
   //attach and link shaders to the program
   gl.attachShader(glProgram, vertexShader);
   gl.attachShader(glProgram, fragmentShader);
   gl.linkProgram(glProgram);
   if (!gl.getProgramParameter(glProgram, gl.LINK_STATUS)) {
      alert("Unable to initialize the shader program.");
   }

   //use program
```

```
    gl.useProgram(glProgram);
}
```

Task 6
Creating the triangle on the plane
Before we can render our triangle, we need to create the buffer that contains its vertex positions and put the vertex positions in it.
We'll do that using a function we call setupBuffers().
In future this routine will be augmented to create more and more complex 3D objects.

```
function setupBuffers() {
    var triangleVertices = [
        //left triangle
        -0.5, 0.5, 0.0,
        0.0, 0.0, 0.0,
        -0.5, -0.5, 0.0,
        //right triangle
        0.5, 0.5, 0.0,
        0.0, 0.0, 0.0,
        0.5, -0.5, 0.0
    ];
    trianglesVerticeBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices), gl.STATIC_DRAW);
}
```

This routine is pretty simple given the basic nature of the scene in this example.

- We create a JavaScript array *triangleVertices* containing the position for each vertex of two triangles.
- By the context.createBuffer() method of the WebGL API we create and initialize *trianglesVerticeBuffer* a WebGLBuffer storing data such as vertices or colors.
- Then we bound to the context by calling the bindBuffer() method.
- The ArrayBuffer object is used to represent a generic, fixed-length raw binary data buffer.
    - The ArrayBuffer() constructor creates a new ArrayBuffer of the given length in bytes
    - It is an array of bytes. You cannot directly manipulate the contents of an ArrayBuffer; instead, you create one of the typed array objects or a DataView object which represents the buffer in a specific format, and use that to read and write the contents of the buffer.
- The *bufferData* the WebGLRenderingContext.bufferData() method of the WebGL API initializes and creates the buffer object's data store.

Task 7
Rendering the scene
Once the shaders are established, the locations are looked up, and the square plane's vertex positions put in a buffer, we can actually render the scene.
Since we're not animating anything in this example, our drawScene() function is very simple. It uses a few utility routines we'll cover shortly.

```
function drawScene() {
    vertexPositionAttribute = gl.getAttribLocation(glProgram, "aVertexPosition");
    gl.enableVertexAttribArray(vertexPositionAttribute);
    gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
    gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT, false, 0, 0);
    gl.drawArrays(gl.TRIANGLES, 0, 6);
}
```

Task 8
Rendering the scene with colors.

We have to change setupBuffers() function.
We have to add different color to each vertex.

```
function setupBuffers(){
	var triangleVertices = [
		//left triangle
		-0.5, 0.5, 0.0,
		0.0, 0.0, 0.0,
		-0.5, -0.5, 0.0,
		//right triangle
		0.5, 0.5, 0.0,
		0.0, 0.0, 0.0,
		0.5, -0.5, 0.0
	];
	trianglesVerticeBuffer = gl.createBuffer();
	gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
	gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices), gl.STATIC_DRAW);

	var triangleVerticeColors = [
		//red left triangle
		1.0, 0.0, 0.0,
		1.0, 1.0, 1.0,
		0.0, 1.0, 0.0,
		//blue right triangle
		0.0, 1.0, 0.0,
		1.0, 1.0, 1.0,
		0.0, 0.0, 1.0
	];
	trianglesColorBuffer = gl.createBuffer();
	gl.bindBuffer(gl.ARRAY_BUFFER, trianglesColorBuffer);
	gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVerticeColors), gl.STATIC_DRAW);
}
```

We have to change drawScene() function.
We have to add different color to each vertex.

```
function drawScene(){
		vertexPositionAttribute = gl.getAttribLocation(glProgram, "aVertexPosition");
		gl.enableVertexAttribArray(vertexPositionAttribute);
		gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
		gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT, false, 0, 0);

		vertexColorAttribute = gl.getAttribLocation(glProgram, "aVertexColor");
		gl.enableVertexAttribArray(vertexColorAttribute);
		gl.bindBuffer(gl.ARRAY_BUFFER, trianglesColorBuffer);
		gl.vertexAttribPointer(vertexColorAttribute, 3, gl.FLOAT, false, 0, 0);

		gl.drawArrays(gl.TRIANGLES, 0, 6);
		//gl.drawArrays(gl.LINES, 0, 6);
		//gl.drawArrays(gl.POINTS, 0, 6);
}
```

Task 9
Rendering the scene with movements.

We have to change the initWebGL() function:

```
function initWebGL(){
	canvas = document.getElementById("my-canvas");
	try{
		gl = canvas.getContext("webgl");
	}catch(e){	}
	if(gl){
		initShaders();
		setupBuffers();
		(function animLoop(){
			setupWebGL();
			setupDynamicBuffers();
			drawScene();
			requestAnimationFrame(animLoop, canvas);
		})();
	}else{
		alert( "Error: Your browser does not appear to" +
			"support WebGL.");
	}
}
```

We have to replace the setupBuffers() function by setupDynamicBuffers() function:

```
function setupDynamicBuffers(){
	//limit translation amount from -0.5 to 0.5
	var x_translation = Math.sin(angle)/2.0;

	var triangleVertices = [
		//left triangle
		-0.5 + x_translation, 0.5, 0.0,
		 0.0 + x_translation, 0.0, 0.0,
		-0.5 + x_translation, -0.5, 0.0,

		//right triangle
		0.5 + x_translation, 0.5, 0.0,
		 0.0 + x_translation, 0.0, 0.0,
		0.5 + x_translation, -0.5, 0.0,
	];
	angle += 0.01;

	trianglesVerticeBuffer = gl.createBuffer();
	gl.bindBuffer(gl.ARRAY_BUFFER, trianglesVerticeBuffer);
	gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices), gl.DYNAMIC_DRAW);

}
```