Recap of introduction to WebGL

WebGL is a derived from OpenGL (Open Graphics Library). This library is a standard that is to be implemented by makers of Graphics cards. The library specifies all methods that need to be implemented. The graphics card makers (eg. Nvidia, AMD, Asus, Intel, Gigabyte) then write the implementation of these methods/functions however way they feel best. The user of the method (you, the programmer) need not know anything about the implementation of the method, you only need to know the name, and signature of the method. This is similar to how when you use printf in C, C++, Java or println in Java you don't concern yourself with how the method works (i.e implementation) you only care about what the method does.

WebGL is run on the browser using the HTML5 canvas

eg.

```
1  <html>
2  <head>
3  <title> WebGL Intro</title>
4  <script src ="myscript.js" defer></script>  </head>
5  <body>
6  <canvas id = "myglcanvas" width="800" height="600" style="border:1px solid
7  #000000;">
8  </canvas> </body>
9  </html>
```

This creates a basic page with a 800x600 pixel canvas with 1point black border whose id is myglcanvas. This id can be used to refer to this canvas in the Javascript (myscript.js in line 4).

To refer to this canvas use the document.getElementByID("myglcanvas")

If you did not give this canvas an id you can use document.querySelector("canvas") and this will return the **first canvas** in the document. Below is simple code to draw a square in the canvas

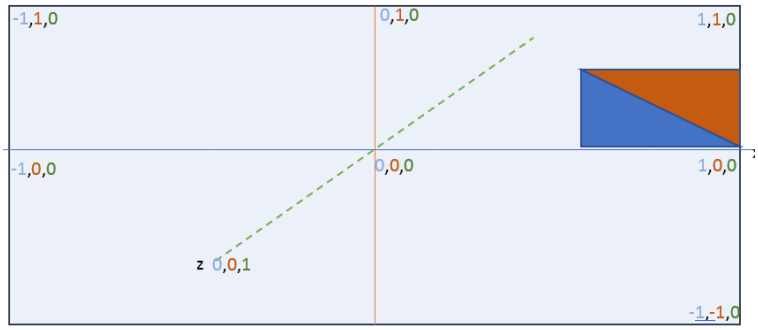
Webgl canvas is from -1 to 1

To draw the square we need 2 triangles

X-Y co-ordinates of the 2 triangles are:

0.5, 0.0 1.0, 0.0 0.5, 0.5

1.0, 0.0 1.0, 0.0 0.5, 0.5



Because this is a 2D shape, the Z value of all the co-ordinates is zero (∴ no need to write it)

```
1  const canvas = document.querySelector(`canvas`);
2  const webgl = canvas.getContext(`webgl`);
3  if(!webgl){    throw new Error("WebGL not available/supported");}
4  webgl.clearColor(0,1,0.2,0.3);
5  webgl.clear(webgl.COLOR_BUFFER_BIT);
6  const vertices = new Float32Array([0.5, 0, 1, 0, 0.5, 0.5,1, 0,1, 0,0.5, 0.5]);
7  const buffer = webgl.createBuffer();
8  webgl.bindBuffer(webgl.ARRAY_BUFFER, buffer);
9  webgl.bufferData(webgl.ARRAY_BUFFER, vertices, webgl.STATIC_DRAW);
10 const vertexShader = webgl.createShader(webgl.VERTEX_SHADER);
11 webgl.shaderSource(vertexShader,
12    `attribute vec2 pos;
13    void main()   {       gl_Position = vec4(pos,0,1);    }` );
14 webgl.compileShader(vertexShader);
15 const fragmentShader =  webgl.createShader(webgl.FRAGMENT_SHADER);
16 webgl.shaderSource(fragmentShader,
17    `void main()   {       gl_FragColor = vec4(0,0,.7,1);    }`);
18 webgl.compileShader(fragmentShader);
19 const program = webgl.createProgram();
20 webgl.attachShader(program, vertexShader);
21 webgl.attachShader(program, fragmentShader);
22 webgl.linkProgram(program);
23 const positionLocation = webgl.getAttribLocation(program, `pos`);
24 webgl.enableVertexAttribArray(positionLocation);
25 webgl.vertexAttribPointer(positionLocation, 2, webgl.FLOAT, false, 0, 0);
26 webgl.useProgram(program);
27 webgl.drawArrays(webgl.TRIANGLES, 0, 6);
```

As we already said before, Line 1 is getting a reference to the canvas . We have also said you can also get the reference by using canvas = document.getElementByID("myglcanvas"). The variable canvas could have been defined by var canvas =, let canvas = or const canvas = as we have chosen. If you do not intend changing a variable (even by mistake) then it is best to make it const.

Line 2 gets the webgl context from the canvas. If there is no webgl context (because the browser is old and does not support webgl eg. IE) the null will be returned (ie webgl = null).

Line 3 checks if null was returned, if it was then it logs an error on the console.

Lines 4 and 5 clear the canvas to the given colour, in this case half transparent blue.e

Line 6 creates a variable that holds the 6 vertex points that make up our two triangle which form the rectangle looking square. The buffer expects this data to be a 32 bit (4 byte) floating point, but JavaScript numbers are 64 bit (8 byte) floats. So to get 32 bit float use the constructor new Float32().

Lines 7,8 and 9: create the buffer; bind the buffer as the current buffer; pass the float32 data to the buffer. If in line 6 you used a 64 bit array, in line 9 where you have vertices, you would have to have new Float32(vertices)

Lines 10 and 15 create a JS reference to the shader

Lines 11 and 16 pass the source code of the shaders.

Lines 14 and 18 compile the shaders

Lines 12,13 and 17 are the shaders in C-like GLSL (openGL Shading Language).

Line 12 – the data in the buffer contains attributes of your each vertex point. At the moment each vertex point only has position attribute data, however it could additionally have normal, color, etc. Also the position attribute only has two elements, which are x and y (the position attribute can have up to 4 elements x,y,z and w). To hold two element data we use a vec2 data type. If you opt to use a vec4 to hold the 2 element data, the value for z and w will be 0 and 1 respectively.

Line 13 – in the main method we need to pass the data from the vec2 attribute we have created in line 12 to the global variable that holds position data which is gl_Position (which expects vec4 data, i.e x, y,z,w). Converting the vec2 to vec4 is as simple as vec4(vec2,z,w) In this case vec2 is pos, z is 0 and w is 1, so it will be vec4(pos,0,1). As pointed out in the previous paragraph, if you made pos to be a vec4 in line 12 then gl_Position = pos.

Line 17 does the same thing as line 13 except that the colour data does not come from outside the program. We need to pass the colour to the global variable gl_FragColor. We've set it as a solid (no transparency) variant of blue (0 red, 0 green, 0.7 blue, 1 opacity)

Lines 19-22: create a program; attach the 2 shaders to it; link the program

Line 23 locate the attribute you created in line 12 and have a reference to it.

Line 24 use that location reference to enable the attribute. The attribute is disabled by default. If the attribute is not enabled, it cannot be used. You need to do this (line 23&24) for all your attributes.

Line 25 vertexAttribPointer(index, size, type, normalized, stride, offset) we specify in what order the attributes are stored, and what data type they are in vertexAttribPointer(index

=start at positionLocation, size =2 for x and y data, type =float which is a 32 bit float, normalized = false because the data is already between -1 and 1 there is therefore no need to normalize it to -1 to 1, stride = 0 which is the number of bytes to be skipped before reaching the next attribute data - in this case you only have one attribute, offset =0x4 this is an offset in bytes of the first component in the vertex attribute array)

Line 26 use the program

Line 27 drawArrays(mode, first, count) draws primitives from the array data
drawArrays(mode = primitive to draw - in our case triangles, first = 0 starting index in the array of vector points, count = 6 number of indices to be rendered 2 triangle with three indices each)

Other types of primitives we can draw are points (dot), line strip, line loop, line, triangle strip and triangle fan