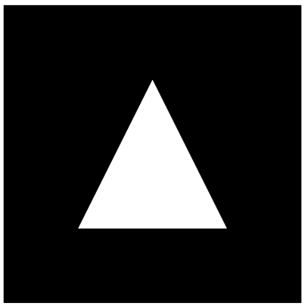
Tutorial 3 A simple WebGL Program

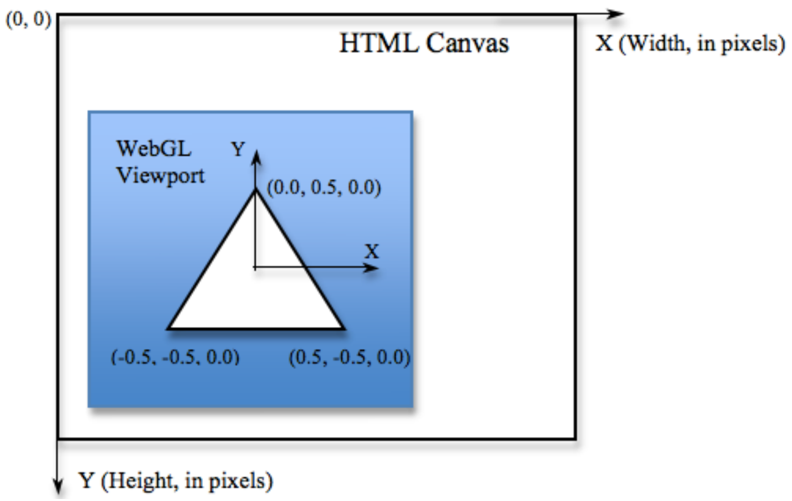
In this first WebGL program, we will have a 2D application that draws a white triangle on a black background. Drawing in 3D will be in place when the uses of transformations are introduced. Although the application is in 2D and simple, the structure and control flows of the program are the same as those of more complicated 3D graphics applications. So, it is important that you thoroughly understand this application. New mechanism, structures and techniques will be introduced and gradually add to your skill.



In a WebGL application, the source code of the *vertex shader* and the *fragment shader* must be written in OpenGL ES Shading Language. The shader source code will be *compiled* and *linked* at run time before it can be used as a shader program by the GPU. This is one of the reasons why several steps are needed to set up a WebGL application that draws even a very simple shape. The required steps for setting up a basic WebGL application are listed here:

1. Write some basic HTML code that includes a <canvas> tag. The <canvas> tag provides the drawing area for WebGL. Then you need to write some JavaScript code to create a reference to your canvas so you can create a WebGLRenderingContext. This context provides the environment for accessing the WebGL API.
2. Write the source code for the vertex shader and fragment shader. The shader source code will be read as a (long) *string*.
3. Write source code that uses the WebGL API to create shader objects for both the vertex shader and the fragment shader. The shade source codes (long strings) will be loaded into the *shader objects* and compiled.
4. Create a *program object* and attach the compiled shader objects to this program object. After this, you can link the program object and then tell WebGL API that you want to use this program object for rendering.
5. Set up the WebGL buffer objects and load the vertex data of the shape (in this case, the triangle) into the buffer.
6. Tell WebGL API which buffer you want to connect to which attribute in the shader, and then, finally, draw the shape.

Note: In the program, the geometry (a triangle) is defined in *viewport coordinate system*, which has the *normalised* coordinates that range from -1 to 1 for both x- and y-axis. We set the viewport to be the same as the HTML canvas, although we do not have to. The general relationship between the viewport and the canvas and their coordinate systems are as shown in the figure below.



Program List

Complete the partially finished program provided by adding the missing statements/blocks. Copy the JavaScript libraries provided on Moodle into your working directory.

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>A Simple WebGL Application</title>
<meta charset="utf-8">

<!External JavaScript file>
<script src="webgl-debug.js"></script>

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

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

<!Scritp for WebGL program >
<script type="text/javascript">

//Define global variables
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;


//This function is the entry point of this webgl application
//It is the first functioned to be called when html doc is loaded into
//the browser. See html code at the end
function startup() {
   //retrieve html canvas
   canvas = document.getElementById("myGLCanvas");
   //create wbegl context. Here, the debugging context is create
   //by calling a function in library "webgl-debug.js"
   gl = WebGLDebugUtils.makeDebugContext(createGLContext(canvas));
   setupShaders();
   setupBuffers();
   //Set the colour to draw with
   gl.clearColor(0.0, 0.0, 0.0, 1.0);
   draw();
}

//Create WebGL context. Recall that we have use getContext("2D")
//to create a 2D context for drawing 2D graphics
function createGLContext(canvas) {
   var names = ["webgl", "experimental-webgl"];
   var context = null;
   for (var i=0; i < names.length; i++) {
      try {
            context = canvas.getContext(names[i]);
      } catch(e) {}
      if (context) {
            break;
      }
   }

   if (context) {
      context.viewportWidth = canvas.width;
      context.viewportHeight = canvas.height;
   } else {
      alert("Failed to create WebGL context!");
   }
   return context;
}

//Load shaders from DOM (document object model). This function will be
//called in setupShaders(). The parameters for argument id will be
//"shader-vs" and "shader-fs"

function loadShaderFromDOM(id) {
   var shaderScript = document.getElementById(id);

   // If there is no shader scripts, the function exist
   if (!shaderScript) {
      return null;
   }

   // Otherwise loop through the children for the found DOM element and
   // build up the shader source code as a string
   var shaderSource = "";
   var currentChild = shaderScript.firstChild;
   while (currentChild) {
      if (currentChild.nodeType == 3) { // 3 corresponds to TEXT_NODE
         shaderSource += currentChild.textContent;
      }
      currentChild = currentChild.nextSibling;
   }

   //Create a WebGL shader object according to type of shader, i.e.,
   //vertex or fragment shader.
   var shader;
   if (shaderScript.type == "x-shader/x-fragment") {
      //call WebGL function createShader() to create fragment
      //shader object
      shader = gl.createShader(gl.FRAGMENT_SHADER);
   } else if (shaderScript.type == "x-shader/x-vertex") {
      //call WebGL function createShader() to create vertx shader obj.
      shader = gl.createShader(gl.VERTEX_SHADER);
   } else {
      return null;
   }

   //load the shader source code (shaderSource) to the shader object.
   gl.shaderSource(shader, shaderSource);
   gl.compileShader(shader); //compile the shader

   //check compiling status.
   if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert(gl.getShaderInfoLog(shader));
      return null;
   }

   return shader;
}
```

```
function setupShaders() {
   //Create vertex and fragment shdaers
   vertexShader = loadShaderFromDOM("shader-vs");
   fragmentShader = loadShaderFromDOM("shader-fs");

   //create a webgl program object
   shaderProgram = gl.createProgram();

   //load the shaders to the program object
   gl.attachShader(shaderProgram, vertexShader);
   gl.attachShader(shaderProgram, fragmentShader);

   //link shaders and check linking status
   gl.linkProgram(shaderProgram);
   if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert("Failed to setup shaders");
   }

   //activate the program
   gl.useProgram(shaderProgram);

   //add a property to the shader program object. The property is the
   //attribute in the vertex shader, which has been loaded to the program
   //object. Function getAttribLocation() finds the pointer to this
   //attribute
   shaderProgram.vertexPositionAttribute =
   gl.getAttribLocation(shaderProgram, "aVertexPosition");
   gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}

//Buffers are places for data. All data, e.g., vertex coordinates,
//texture coordinates, indices, colours must be stored in their
//buffers. Here, the buffer is for the vertex coordinates of a triangle

function setupBuffers() {
   //A buffer object is first created by calling gl.createBuffer()
   vertexBuffer = gl.createBuffer();
   //Then bind the buffer to gl.ARRAY_BUFFER, which is the WebGL built-in
   //buffer where the vertex shader will fetch data from
   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);

   //Actual coordinates for the vertices
   var triangleVertices = [
                        0.0,  0.5, 0.0,
                       -0.5, -0.5, 0.0,
                        0.5, -0.5, 0.0
                        ];
   //Load the vertex data to the buffer
   gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
                                       gl.STATIC_DRAW);
   //Add properties to vertexBuffer object
   vertexBuffer.itemSize = 3;      //3 coordinates of each vertex
   vertexBuffer.numberOfItems = 3; //3 vertices in all in this buffer
}


function draw() {
     //setup a viewport that is the same as the canvas using
     //function viewport(int x, int y, sizei w, sizei h)
     //where x and y give the x and y window coordinates of the
     //viewport’s lower left corner and w and h give the viewport’s width
     //and height.
   gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
   //fill the canvas with solid colour. Default is black
   //If other colour is desirable using function gl.clearColor (r,g,b,a)
   gl.clear(gl.COLOR_BUFFER_BIT);

   //Inform webgl pipeline with pointer of the attribute
   //"aVertexPosition". Still remember it?
   gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
   vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

   //Draw the triangle
   gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);
}
</script>

</head>
<body onload="startup();">
<canvas id="myGLCanvas" width="500" height="500"></canvas>
</body>
</html>
```