Worksheet 1 – Base – Framework for exercises

```
<html>
<head>
  <title>Base</title>
  <style> canvas { width: 100%; height: 100% } </style>
</head>
<body onload="main()">
  <h1>Base</h1>
</body>

<script src="three.min.js"></script>
<script type='text/javascript' src='DAT.GUI.min.js'></script>
<script src="OrbitControls.js"></script>
<!--script type="text/javascript" src="http://benvanik.github.io/WebGL-Inspector/core/embed.js"></script-->
<script id="vertex-shader" type="x-shader/x-vertex">
  void main() {}
</script>

<script id="fragment-shader" type="x-shader/x-fragment">
  void main() {}
</script>

<script>
  var cube;
  var scene;

  var parameters = {
    time: 0.0,
    radius: 4.0,
    x: 1,
    y: 1,
    z: 1,
    wireframe: false,
    reset: function () {
      resetCube()
    }
  }

  function setupBox() {
    var geometry = new THREE.BoxGeometry(5, 5, 5, parameters.x, parameters.y, parameters.z);
    var material = new THREE.MeshBasicMaterial();

    //var material = createMaterial("vertex-shader", "fragment-shader");

    material.wireframe = parameters.wireframe;
    cube = new THREE.Mesh(geometry, material);
    scene.add(cube);
  }

  function updateBox(value) {
    scene.remove(cube);
    setupBox();
  }

  function main() {
    scene = new THREE.Scene();
    var camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000);

    var renderer = new THREE.WebGLRenderer();
    renderer.setSize(window.innerWidth * 0.75, window.innerHeight * 0.75);
    document.body.appendChild(renderer.domElement);

    setupBox();

    camera.position.z = 10;

    var controls = new THREE.OrbitControls(camera, renderer.domElement);

    (function setupGUI() {
      var gui = new dat.GUI();
```

```
70
71          var folder1 = gui.addFolder('Cube Segments');
72          var cubeX = folder1.add(parameters, 'x').min(1).max(20).step(1).listen();
73          var cubeY = folder1.add(parameters, 'y').min(1).max(20).step(1).listen();
74          var cubeZ = folder1.add(parameters, 'z').min(1).max(20).step(1).listen();
75          var radius = folder1.add(parameters, 'radius').min(0.1).max(10.0).step(0.1).listen();
76          var wireframe = gui.add(parameters, 'wireframe').name("Wireframe").listen();
77
78          cubeX.onChange(updateBox);
79          cubeY.onChange(updateBox);
80          cubeZ.onChange(updateBox);
81          radius.onChange(updateBox);
82          wireframe.onChange(updateBox);
83
84          folder1.open();
85        })();
86
87        (function animate() {
88          requestAnimationFrame(animate);
89          renderer.clear();
90          renderer.render(scene, camera);
91          controls.update();
92        })();
93      }
94
95      function createMaterial(vertexShader, fragmentShader) {
96        var vertShader = document.getElementById(vertexShader).innerHTML;
97        var fragShader = document.getElementById(fragmentShader).innerHTML;
98
99        var attributes = {};
100       var uniforms = {};
101
102       var meshMaterial = new THREE.ShaderMaterial({
103         uniforms: uniforms,
104         attributes: attributes,
105         vertexShader: vertShader,
106         fragmentShader: fragShader,
107       });
108       return meshMaterial;
109     }
110   </script>
111   </html>
```

This is the skeleton code for the workshop. As it stands it's very dull, just a plain white cube, with a basic material.

I've tried to group the code and provide enough structure to allow us to concentrate on writing the shader.

It uses dat.GUI (https://code.google.com/p/dat-gui/) to allow us to change parameters on the fly rather than switching between editing and view, particularly later on.

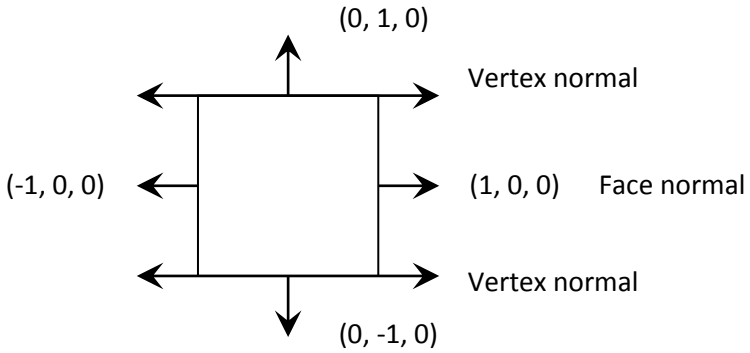
Note: they've been updating Three.js again, CubeGeometry is being deprecated and replaced with BoxGeometry.

Task 1 – Colour faces based on face (vertex) normal

The cube looks exceedingly dull, let's colour it.

In CGI, a colour usually consists of 4 components, one each for red, green, blue and transparency or RGBA. Each component takes a value representing an intensity (between full off and full on) in the range 0·0 – 1·0.

For a cube each face has a normal with 3 components, and since each normal is different we can use it to generate a colour (XYZ1 → RGBA), ***but*** the normals range from (-1, -1, -1) to (1, 1, 1) so we'll need to offset and normalise them.



WebGL defines the normal by vertex rather than by face, then interpolates it for the fragment shader, but since Three.js provides the same normal for each of the vertices they remain constant.

1. Write a ***vertex shader*** to do the following:
   a. set gl_Position using the usual method (see below†)
   b. convert the vertex normal to a colour
   c. pass the colour as a varying to the fragment shader
2. Write a ***fragment shader*** to do the following:
   a. set gl_FragColor
3. Pick up the custom ShaderMaterial (using createMaterial) rather than the MeshBasicMaterial.

WebGL defines the normal by vertex rather than by face, then interpolates it for the fragment shader, but since Three.js provides the same normal for each of the vertices they remain constant.

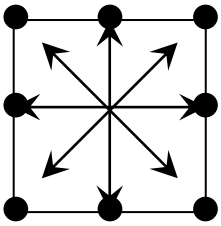
†Here's a pair of simple shaders for reference:

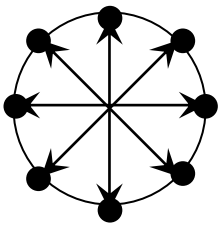
```
<script id="vertex" type="x-shader">

  void main() {
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
  }
</script>

<script id="fragment" type="x-shader">

  void main() {
    gl_FragColor = vColor;
  }
</script>
```

Task 2 – Make the box spherical

Now let's make the box spherical.

All points on the surface of a sphere are equidistant from the centre, so we need to move all the boxes' vertices so that they are.





1. Change the ***vertex shader*** to do the following:
    a. Add uniforms for the centre and radius of the sphere
    b. For a vertex, find the vector from the centre to it
    c. Multiply the vector by the radius and add it to the centre
2. Add a uniform for the radius to the empty list in createMaterial

Who can spot the bug?

Task 3 – Procedural texturing part 1

Let's get ambitious and replace the plain colours with a procedural texture.

The texture is taken from http://glsl.heroku.com/e#7868.1

1. Change the ***vertex shader***:
    a. Remove the varying used for the face colour
2. Change the fragment shader:
    a. Replace the shader with code below
3. In function animate() add lines to increment parameter time by 0.005 and assign to cube.material.uniforms.time.value
4. In createMaterial add uniforms for time (type 'f', value 0.2), scale (type 'f', value 0.2), alpha (type 'f', value 0.2) and resolution (type 'v2', value new THREE.Vector2(5,5)).

Task 4 – Procedural texturing part 2

The texture looks “wrong”, it doesn’t change when the cube rotates; this is because the shader uses gl_FragCoord which works in screen space and so never changes for a given pixel. For a better effect we need to use uv co-ordinates for each face.

1. Change the ***vertex shader***:
    a. Add a varying for a texture co-ordinate (i.e. vec2 vUv)
    b. Assign uv to vUv
2. Change the fragment shader:
    a. Add a varying for a texture co-ordinate (i.e. vec2 vUv)
    b. Replace gl_FragCoord.xy with vUv:
        i. Assign vUv to center (about line 38)
        ii. Assign (vUv.xy - resolution) / min(resolution.y,resolution.x) * 15.0; to v (about line 43)

Task 5 – Where are all those varyings and uniforms coming from?

You may recall from the first workshops that WebGL provides very few built-in variables; no projection matrix model view matrix, etc. Instead, we have to supply them in ***raw*** WebGL. Something like this:

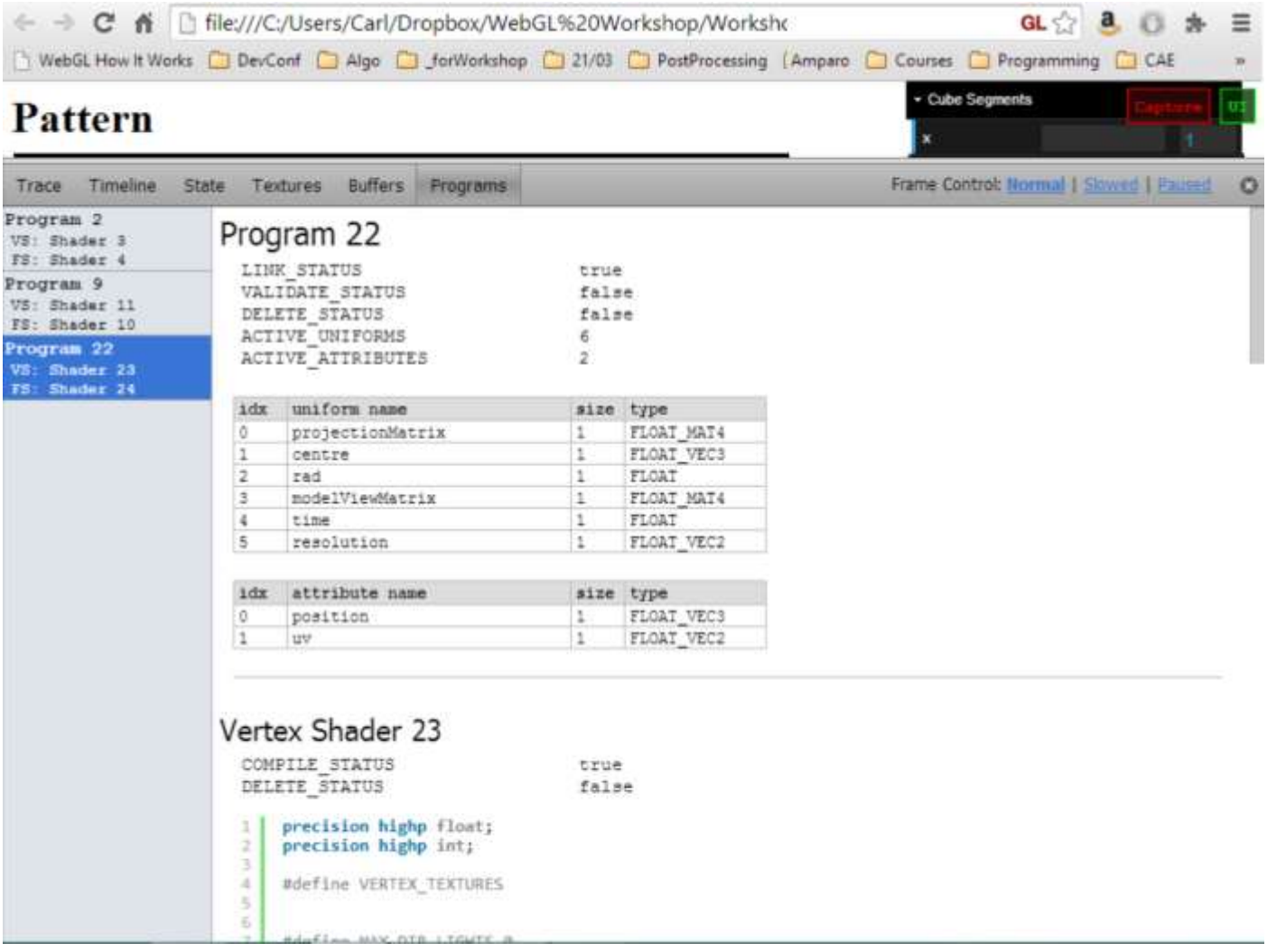
```
<script id="vertex" type="x-shader">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
```

```
  varying vec4 vColor;

  void main(void) {
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
    vColor = aVertexColor;
  }
</script>

<script type="text/javascript">
function initShaderProgram() {
  pMatrixUniform = gl.getUniformLocation(shaderProgram, "uPMatrix");
  mvMatrixUniform = gl.getUniformLocation(shaderProgram, "uMVMatrix");

  vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "aVertexPosition");
  gl.enableVertexAttribArray(vertexPositionAttribute);
  vertexColorAttribute = gl.getAttribLocation(shaderProgram, "aVertexColor");
  gl.enableVertexAttribArray(vertexColorAttribute);
}
</script>
```

However, our JavaScript code doesn't supply them, so where are projectionMatrix, modelViewMatrix and the rest coming from?

If you have WebGL Inspector‡ you can look at the shader programs being used. Click the GL symbol in the URL window, click the programs tab, "our" programs should be the last in the list, click that. At the top of the listing there should be a list of all uniforms and attributes in the program. Amongst those we defined there should be a few more we didn't.

‡ If you don't, uncomment line 13 and it should be pulled in remotely.



References and resources

Material and code based in part on chapter 4 of *Learning Three.js: The JavaScript 3D Library for WebGL* by Jos Dirksen.

Shader based on http://glsl.heroku.com/e#7868.1

dat.GUI can be found at https://code.google.com/p/dat-gui/