**Part 1**

1. **Describe how an index buffer object improves the use of a vertex buffer object in order to represent a 3D mesh. [4 marks]**

Vertex buffers have a disadvantage when objects are required to be drawn that make use of a lot of the same vertices repeatedly, as these vertices are still required to be redefined each and every time. Index buffer objects are used as it not only decreases the number of vertices the developer has to code to produce the same image, however also when colour changes are required in the program, this now involves just a single change, rather than one to each reference of that vertices, saving both time and memory. This will make a significant difference in larger 3D meshes, alongside the memory saved and time saved during development.

1. **Given a fragment shader program as follows:**

***void main() { gl\_FragColor = vec4(0.0, 0.0, 1.0, 1.0); }***

**Describe the functionality of this fragment shader program. If this fragment shader program is adopted by a WebGL program, which was supposed to support point lighting for mesh rendering, explain whether this fragment shader program will perform this job correctly. If yes, describe the rendering result produced. Otherwise, describe how you modify the program to support point lighting. [8 marks]**

This shader program is made up of *gl\_FragColor*, which is initialising by the creation of a *vec4* object vector made from 0, 0, 1 and 1, and is what controls the colour of an individual fragment. This is performed one time for each of the pixels on the shape. The four floating point numbers represent the RGBA values, which, in this case, produce a blue colour.

If this fragment shader were adopted by a WebGL program intended to support point lighting for mesh rendering, it would not work, because, as mentioned above, it would render each and every fragment of the object blue. The correct code for the full fragment shader, including the main method, which was found via the textbook (<http://uniguld.dk/wp-content/guld/DTU/webgrafik//0321902920_WebGL.pdf>), is below:

var FSHADER\_SOURCE =

'#ifdef GL\_ES\n' +

'precision mediump float;\n' +

'#endif\n' +

'uniform vec3 u\_LightColor;\n' + // Light color

'uniform vec3 u\_LightPosition;\n' + // Position of the light source

'uniform vec3 u\_AmbientLight;\n' + // Ambient light color

'varying vec3 v\_Normal;\n' +

'varying vec3 v\_Position;\n' +

'varying vec4 v\_Color;\n' +

'void main() {\n' +

// Normalize the normal because it is interpolated and not 1.0 in length any more

' vec3 normal = normalize(v\_Normal);\n' +

// Calculate the light direction and make its length 1.

' vec3 lightDirection = normalize(u\_LightPosition - v\_Position);\n' +

// The dot product of the light direction and the orientation of a surface (the normal)

' float nDotL = max(dot(lightDirection, normal), 0.0);\n' +

// Calculate the final color from diffuse reflection and ambient reflection

' vec3 diffuse = u\_LightColor \* v\_Color.rgb \* nDotL;\n' +

' vec3 ambient = u\_AmbientLight \* v\_Color.rgb;\n' +

' gl\_FragColor = vec4(diffuse + ambient, v\_Color.a);\n' +

'}\n';

1. **Describe the main functionality of normal vector in terms of 3D mesh rendering. Suppose in a WebGL program, the initial normal vectors of a 3D mesh have been pre-computed. If this 3D mesh is then being rotated in the program, explain how you will update its normal vectors accordingly. [8 marks]**

The normal vector is the vector which is perpendicular (at a right angle) to a surface, and it determines that surface’s orientation.

The main functionality of normal vectors in 3D mesh rending is in lighting calculations, in order to light, provide colour and then add shading to any surface corresponding to the locations of lighting sources, view perspectives and additional objects (which lead to shadows). This functionality is provided because it is possible to change how the normal vectors are calculated for each surface, depending on how you want light to interact with it (in terms of reflection, shadowing etc).

Under translation, the normal direction is constant, however it does change under rotation. In order to calculate this new normal direction, the normal must be multiplied by the inverse transpose of whatever model matrix (the transformational matrix) is being applied.

***(Reference: WebGL Prog. Guide Chapter 4, Table 4.1)* Suppose drawBox(m)is a function to draw a transformed box according to the transformation matrix m. That is, if M is a rotation matrix, the function will draw a rotated box.**

* 1. **Explain the meaning of the following code segment and state the result obtained [7 marks]:**

**m.setTranslate(20.0, 0.0, -30.0); m.rotate(angle, 2.0, 0.0, 0.0); drawBox(m);**

The first function, setTranslate, sets the transformation matrix *m* to be the matrix equivalent to moving the original object 20 units in the positive x-axis direction, not moving in the y-axis plane, and then moving 30 units in the **negative** z-axis direction

The next function, rotate, multiplies (rather than sets) this new translation matrix *m* by the matrix corresponding to a ‘**angle’** degrees rotation about the x rotational axis in this case

The multiplication of these gives the overall transformation matrix m applied to the box.

**Explain whether you will get the same result if m.rotate() has been replaced by m.setRotate(). [3 marks]**

You would not get the same result. In the previous part, the rotational matrix is multiplied to the transformation, whereas in this part, the transformation matrix is set to be the rotational matrix identified previous instead. This means the initial translation is effectively overridden. This means that the vertices of the box in question this time will simply be multiplied by the matrix above, rather than the product produced in the earlier part.