COMPUTER GRAPHICS
SUMMATIVE ASSIGNMENT

Question 1a

We represent a 3D mesh as a collection of triangles. This is because we know that 3 points will always form a triangle, even in 3D space. However, this means that each point on the mesh will correspond to multiple triangles (presuming that the mesh is connected). Therefore, if we just list the vertex data in the vertex buffer object (VBO) then we will need to repeat the data for each position for each triangle. By using an index buffer object (IBO), we instead define the data for each vertex only once. We then use the IBO to reference these position for each triangle, greatly reducing the amount of memory needed to store the VBO.

Question 1b

This fragment shader will not perform spot lighting correctly, and will instead colour every pixel in the mesh blue (#0000FF). *gl_FragColour* determines the colour (and alpha) to use for each pixel. Because this is set to a constant *vec4(0.0, 0.0, 1.0, 1.0)* then the blue and alpha components will always be 1.0 (and the red and green components 0.0).

To support point lighting, we must first send the colour of each vertex from the vertex shader, to the fragment shader as the interpolated colour. To do this we use a varying in both the vertex and fragment shader and pass the colour of the vertex into the varying in the vertex shader. The varying in the fragment shader will then contain the interpolated colour for the given fragment (pixel). However, this just correctly sets up the colour, but does not actually support point lighting.

To support point lighting the fragment shader will need several other varyings given to it from the vertex shader. These are the VertexPosition and NormalMatrix, which are interpolated per fragment. We could also pass in uniforms to change the light position, light colour, light intensity, etc., but these could be hardcoded as constants and point lighting would still be possible (but not configurable). We subtract the VertexPosition from the light position to get a vector representing the light direction, which we normalize. Once we have a normalized vector representing the light direction we then use it to apply directional lighting.

We calculate the directional lighting by finding a scalar which represents the intensity of light on the fragment. This angle ($\cos \theta$) is the dot product of the light direction and the orientation of the surface (which is the NormalMatrix for the fragment normalized). Once we have this scalar we can then multiply it by the light colour and vertex colour which gives us the final colour to display (from point lighting).

Question 1c

The normal vector represents the orientation of a surface. In 3D mesh rendering this can be calculated per vertex, or also interpolated per fragment (ie. Using Phong Shading/Interpolation). Regardless of the method used, we use the normal per vertex/fragment to calculate the angle at which light hits the surface of the mesh. This is then used to calculate the colour/alpha value of each pixel.

If the 3D mesh is updated, then there are two ways to update the normal vectors. The first way which is most commonly used is to recompute the vectors. This is calculated by finding the transpose of the inverse of the model matrix. If we do not wish to recompute the normal vectors, then we can instead apply matrix transformations to them. For each axis that we wish to rotate, we compute the following matrices:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix}$$

$$R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix}$$

$$R_z(\gamma) = \begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Source: mathworld.wolfram.com/RotationMatrix.html

To find the overall rotation matrix we multiply R_x, R_y, and R_z together. Then for each normal vector we multiply it by the rotation matrix to find the new normal vector.

Question 1d

i)

The first statement replaces the matrix with a transformation matrix. This translation matrix translates the box by 20 in the x direction and -30 in the z direction. It does not matter what M used to be, as 'setTranslate' will override this. Secondly the 'rotate' function combines the existing matrix with a rotation matrix. This rotation matrix has an angle of *angle* degrees and rotates in the x axis. The last 3 arguments represent a vector of which axis to rotate in. Because this vector is normalized, the fact that the x-axis parameter is 2 is irrelevant, since it will be normalized to 1. This would only be important if the y or z had a non-zero value, as they would be scaled appropriately.

The last statement draws the box according to the model matrix. In this case it is a box drawn at (20, 0, -30) that has been rotated *angle* degrees around the x-axis. The order of the first two statements is important, if 'setTranslate' had been called second, then it would have overridden the rotation matrix, leading to only a translation. However, if we swapped the order around and called 'setRotate and then translate' the result would not be the same. This is because due to the order of matrix multiplication, the translated box would then be rotated around the origin, leading to it being in a completely different position.

ii)

No, because 'setRotate' will override the existing transformation matrix with a rotation matrix. The transformation matrix will no longer be a composite transformation consisting of a translation and rotation, but instead just be a rotation transformation. Therefore, the box will not be drawn rotated at (20, 0, -30), but instead drawn rotated at (0, 0, 0).