a)

A Vertex Buffer stores the co-ordinates of each vertex in the mesh. The Index Buffer stores pointers to these vertices, indicating how the triangles should be drawn on the cluster of vertices.

Without an Index Buffer, we must specify the co-ordinates for each vertex in each triangle. Consider a Cube. Each vertex will be part of at least three triangles. In this case, we would need to write the co-ordinates of each vertex at least three times. This also adds another point of error, because to change one vertex we must change it at least three times in the vertex buffer.

Using an Index Buffer, we simply declare the vertices once in the Vertex Buffer, then use pointers to the vertices. This method has the additional benefit of reducing the memory footprint of the model, since each vertex needs only be stored once in memory, as opposed to 3+ times.

It also reduces computation time, as we don't have duplicate vertices so the vertex shader does not need to be run multiple times per vertex.

b)

This code currently colours every fragment in a solid blue color. This will result in the whole screen being blue. This obviously does not support point lighting, nor does it support showing anything meaninful on the screen other than lots of blue.

Vertex Shader for Point Lighting with per-vertex colouring:

------------------------------------

attribute vec4 a\_Position;

attribute vec4 a\_Color;

attribute vec4 a\_Normal;

uniform mat4 u\_ModelMatrix;

uniform vec3 u\_LightPosition;

uniform vec3 u\_LightColor;

varying vec4 v\_Color;

void main(){

gl\_Position = u\_ModelMatrix \* a\_Position;

vec4 vertexPosition = u\_ModelMatrix \* a\_Position;

vec3 lightDirection = normalize(u\_LightPosition - vec3(vertexPosition));

float nDotL = max(dot(lightDirection, normal), 0.0);

vec3 diffuse = u\_LightColor \* a\_Color.rgb \* nDotL;

v\_Color = vec4(diffuse.x, diffuse.y, diffuse.z, a\_Color.a);

}

------------------------------------

Fragment Shader for Point lighting with per-vertex colouring:

------------------------------------

varying vec4 v\_Color;

void main() {

gl\_FragColor = v\_Color;

}

------------------------------------

Vertex Shader for Point Lighting with per-fragment colouring:

------------------------------------

attribute vec4 a\_Position;

attribute vec4 a\_Color;

attribute vec4 a\_Normal;

uniform mat4 u\_ModelMatrix;

uniform vec3 u\_LightPosition;

uniform vec3 u\_LightColor;

varying vec3 v\_Position;

varying vec4 v\_Color;

varying vec3 v\_Normal;

varying vec3 v\_LightColor;

varying vec3 v\_LightPoisition;

void main(){

gl\_Position = u\_ModelMatrix \* a\_Position;

v\_Position = vec3(u\_ModelMatrix \* a\_Position);

v\_Normal = vec3(u\_NormalMatrix \* a\_Normal);

v\_Color = a\_Color;

v\_LightColor = u\_LightColor;

v\_LightPosition = u\_LightPosition;

}

------------------------------------

Fragment Shader for Point lighting with per-fragment colouring:

------------------------------------

varying vec4 v\_Color;

void main() {

precision mediump float;

varying vec3 v\_LightColor;

varying vec3 v\_LightPosition;

varying vec4 v\_Color;

varying vec3 v\_Normal;

varying vec3 v\_Position;

void main() {

vec3 normal = normalize(v\_Normal);

vec3 pointLightDirection = normalize(v\_LightPosition - v\_Position);

float nDotL = max(dot(pointLightDirection, normal), 0.0);

vec3 diffuse = v\_LightColor \* nDotL \* v\_Color.rgb;

gl\_FragColor = vec4(diffuse.x, diffuse.y, diffuse.z, v\_Color.a);

}

}

------------------------------------

c)

A normal vector is the vector representing the normal (perpendicular) to a surface. Normal vectors are used when calculating lighting on a surface because they are the midpoint between the incidence ray and the reflected ray. We can calculate the intensity of light coming from a surface by taking the dot product between the normal vector and the vector representing the direction of the incoming light.

The normal vectors are computed alongside the mesh. Then, when the mesh is transformed, the normal vectors are transformed in the same way. If we translate the mesh, then translate the normal vectors, the resulting vectors will be correct for the mesh in its new location.

The same is true for rotation and scaling. When we rotate/scale the mesh, we simply perform the same transformation on the normal vectors and the result is correct for the new mesh.

d) i)

This code generates a transformation matrix. The matrix, when applied, will cause a translation of +20 in the X direction, and -30 in the Z direction. Then, it causes a rotation of *angle* degrees clockwise around the rotation axis (2, 0, 0), which is the same as saying around the X axis.

The result obtained is a box drawn at +20 in the X direction, and 30 units away from the x axis (on the circle drawn around the x Axis with a radius of 30). Where it is on the circle will depend on the angle passed to the function.

ii)

setRotate() replaces the current values in the matrix. If we use setRotate(), the values in the matrix will be reset before applying the rotation. With this in mind, replacing rotate() with setRotate() will produce a matrix which simply causes a rotation of *angle* degrees clockwise around the x axis.