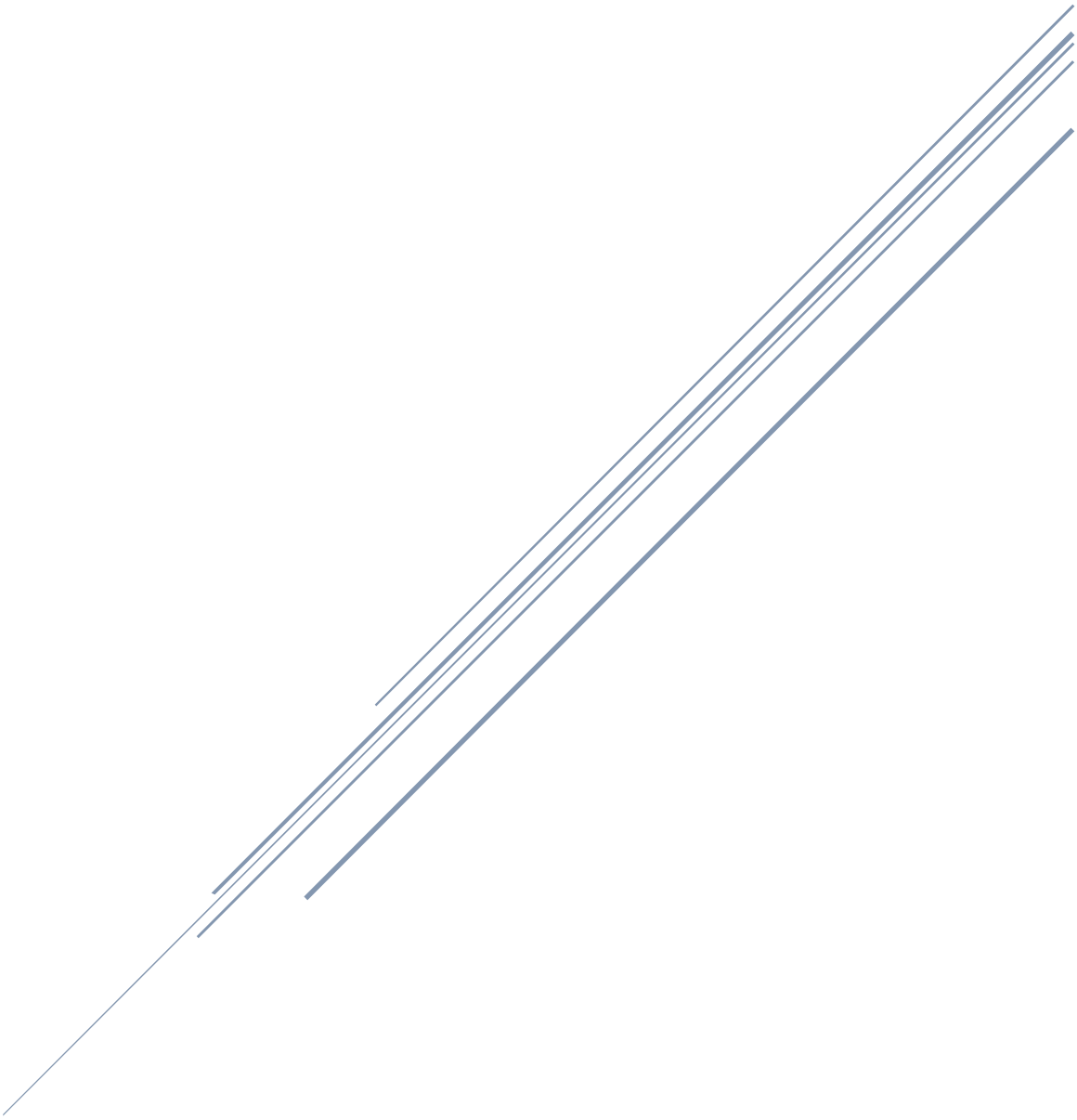
CITS3003: Graphics and Animation

PROJECT REPORT



Aswin Thaikkattu Vinod [22870036]
Callum Brown [22985036]

TASK A:
Modifying the display function will rotate the camera. This can be done by adding the appropriate X and Y rotations using the predefined camera angle variables.

```
// TASK A:
// add a rotational component to the view so it can be manipulated by user input
mat4 rot = RotateX(camRotUpAndOverDeg) * RotateY(camRotSidewaysDeg);
view = Translate(0.0, 0.0, -viewDist) * rot;
```

TASK B:
We used **RotateY**, **RotateX** and **RotateZ** matrices, fill each rotation matrix with the **sceneObj.angles**[X or Y or Z] and modified them inside the **drawMesh** function

```
// TASK B
// similar to TASK A, adding rotational components to the current object to allow it to be manipulated by user input
mat4 model = Translate(sceneObj.loc) * Scale(sceneObj.scale) * RotateX(sceneObj.angles[0]) * RotateY(sceneObj.angles[1]) * RotateZ(sceneObj.angles[2]);
```

Finally, we corrected the rotation tool's directions by inverting the X and Z rotations.

```
if (id == 55 && currObject >= 0) {
    setToolCallbacks(adjustAngleYX, mat2(400, 0, 0, -400),
                adjustAngleZTexscale, mat2(400, 0, 0, 15));

  }
```

Part II:
the texture scaling can be implemented by multiplying **texCoord** in **gl_FragColour** of the fragment shader by **texScale**.

```
uniform float tex_Scale;
```

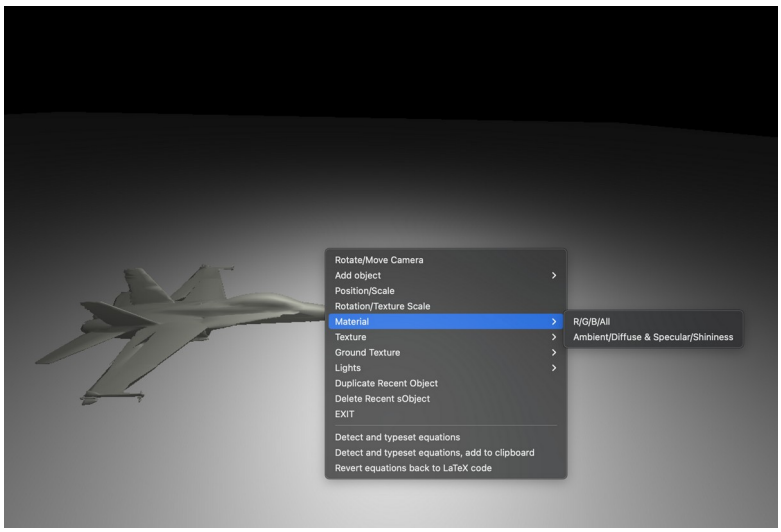
```
gl_FragColor = (color * texture2D( texture, texCoord * 2.0 * tex_Scale)) + vec4(specular + specular2, 1.0);
```

TASK C:
First, we implement callback methods for adjusting ambient/diffuse and specular/shininess
Then a new menu item added within the MaterialMenu function

```
int materialMenuId = glutCreateMenu(materialMenu);
glutAddMenuEntry("R/G/B/All", 10);
glutAddMenuEntry("Ambient/Diffuse/Specular/Shine", 20);
```

```
if(id == 20) {
    toolObj = currObject;
    setToolCallbacks(adjustAmbientDiffuse, mat2(1, 0, 0, 1),
                adjustSpecularShine, mat2(1, 0, 0, 1));
  }
```

```
static void adjustAmbientDiffuse(vec2 ad){
  sceneObjs[toolObj].ambient += ad[0];
  sceneObjs[toolObj].diffuse += ad[1];
}
```



TASK D:

Possibly the easiest task of the project, This task can be implemented by reducing the size of the **nearDist,** which changes the projection from further clipping the object.

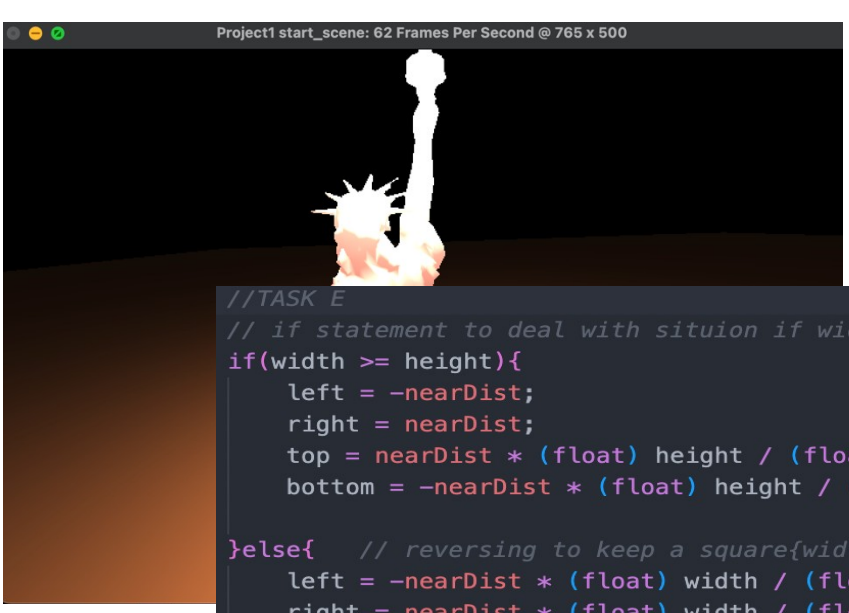
```
GLfloat nearDist = 0.01;
```

more “close-up” views of object can be done by changing, the near distance by a factor of 5. Which in turn increases the camera’s field of view. The initial view distance and Z movement speed were also scaled to compensate.

```
static float viewDist = 7.5; // Distance from the camera to the centre of the scene
static void doRotate() {
   setToolCallbacks(adjustCamrotsideViewdist, mat2(400, 0, 0, -9),
                adjustcamSideUp, mat2(400, 0, 0, -90));
}
```

TASK E:

The reshape function is modified to scale the viewport in the same way as when the window is resized vertically. This was done by using the height-to-width ratio to scale the bottom and top planes when the width is less than the height. This allowed the frustum to always be implemented with square like coordinates




```
//TASK E
// if statement to deal with situion if width is greter than hei
if(width >= height){
    left = -nearDist;
    right = nearDist;
    top = nearDist * (float) height / (float) width;
    bottom = -nearDist * (float) height / (float) width;

}else{   // reversing to keep a square{width  less than height}
    left = -nearDist * (float) width / (float) height;
    right = nearDist * (float) width / (float) height;
    top = nearDist;
    bottom = -nearDist;

projection = Frustum(left, right, bottom, top, 0.2, 500.0);
```

TASK F:
In the physical world, attenuation is known formally as ‘the reduction of the intensity of a ray (of light) as it traverses matter.’ The drop off in intensity occurs due to photons of light being obstructed by the medium (air). In our simulation, we must find a solution that ‘looks right’ while maintaining frame rate for the scene to appear realistic.

Inspiration from https://learnopengl.com/Lighting/Light-casters was taken to calculate the ‘attenuation factor’ or the drop off rate of the brightness (intensity) of the light source interacting with objects of the scene.

$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

In the above equation, the K values are selected from a table based on how large a distance the light is to cover, in our case we selected the values for a distance of ‘50’.

```
// PART F: values for light attenuation equation
float constant = 1.0;
float linear = 0.09;
float quadratic = 0.032;
```

```
// PART F: Light Attenuation as in vStart.glsl
float attenuationFactor = 1.0 / (constant + linear * length(Lvec)) +
                          quadratic * (length(Lvec) * length(Lvec));

ambient *= attenuationFactor;
diffuse *= attenuationFactor;
specular *= attenuationFactor;
```

The above code calculates the attenuation factor with the variable **distance** vector between the vertex and the light source. The three light factors then take into account the attenuation, after which they are used to ‘shade’ the vertices.

TASK G:
Implementing the functionality of the vertex shader (vStart.glsl) in the fragment shader (fStart.glsl) requires a couple of changes as the vertex and fragment processors occur at different stages of the graphics pipeline. Vertices are the corners of polygons constructing the shape of an object in the scene, while fragments are ‘potential pixels’ between the vertices, stored in the frame buffer. Fragments allow for much smoother colouring and shading in the scene.

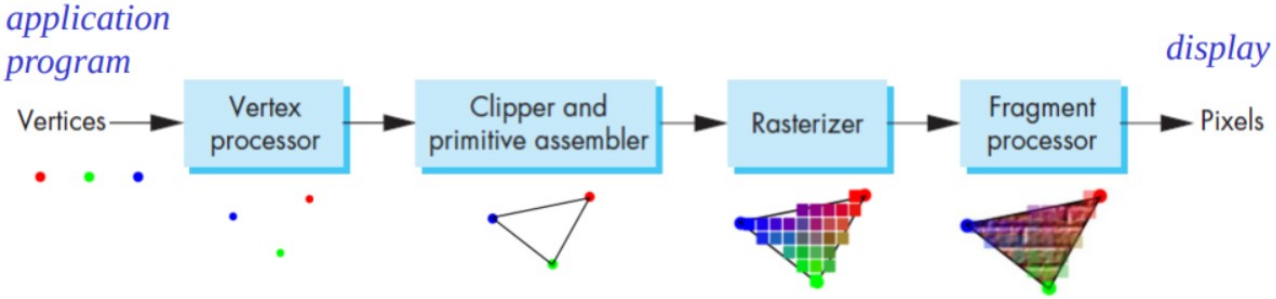


image from CITS3003 2022 S1: Week 2 Lecture 1.

The program takes the fragment position for each pixel to calculate the distance vector to the light source, instead of the vertex position as in the vertex shader. Implementing this involved moving the variables used in the lighting calculations to the fragment shader, and defining the fragment attributes (position, normal) in the vertex shader. The fragment shader then performs the lighting calculations almost exactly as they were done in the vertex shader.

```
attribute vec3 vPosition;
attribute vec3 vNormal;
attribute vec2 vTexCoord;

// for fragment shader
varying vec4 position;
varying vec3 normal;

varying vec2 texCoord;
varying vec4 color;

uniform mat4 ModelView;
uniform mat4 Projection;

void main()
{
    vec4 vpos = vec4(vPosition, 1.0);
    gl_Position = Projection * ModelView * vpos;
    texCoord = vTexCoord;
    position = vpos;
    normal = vNormal;
}
```

The new vertex shader, here you can see position and normal are declared and initialised for use in the fragment shader.

```
varying vec4 position;
varying vec3 normal;
vec4 color;
varying vec2 texCoord;  // The third coordinate is always 0.0

uniform sampler2D texture;

uniform vec3 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform float Shininess;
```

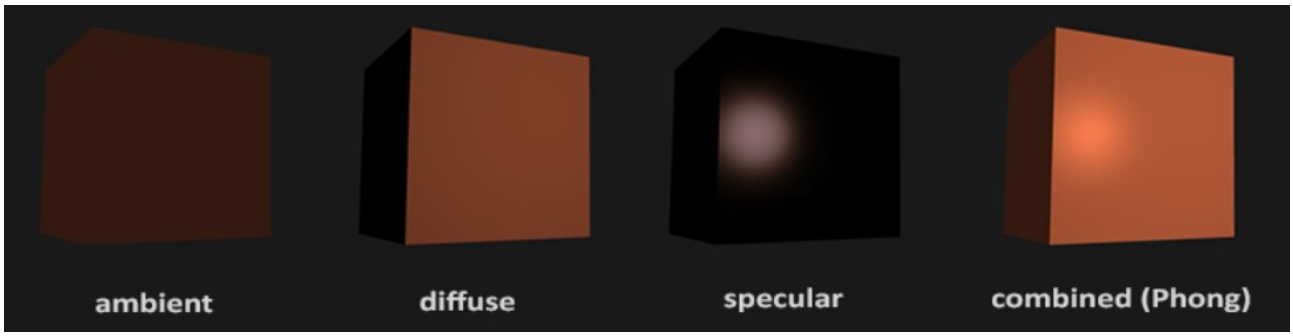
The new fragment shader, the light properties are now declared here instead.

TASK H:

The specular light component highlights an object based on the material's 'shininess.' in order to tend the colour of the highlight to a white brightness, it was a case of removing the specular component from the color.rgb of the fragment, and instead including it as a component of gl_FragColor.

Specular lighting is based also on the view direction, in the form of a reflection vector, as well as the distance between the light and the object. The closer the angle of reflection of the light and the view direction, the greater the specular intensity (or shiny highlights) should appear on the object.

```
// PART H: seperating specular lighting component from the colour aspect of the light source.
gl_FragColor = (color * texture2D( texture, texCoord * 2.0 )) + vec4(specular + specular2, 1.0);
```



TASK I:

The first part of implementing a new light was returning to the c++ code and adding a new scene object in the initialisation function to appear when the program runs.

```
// TASK I: Solar directional light
addObject(55); // Sphere for second light
sceneObjs[2].loc = vec4(0.0, -0.1, 0.0, 1.0);
sceneObjs[2].scale = 0.1;
sceneObjs[2].texId = 0;
sceneObjs[2].brightness = 0.5;
```

Then, in the display function, the light's position is passed into the shader files for manipulation.

```
glUniform4fv(glGetUniformLocation(shaderProgram, "LightPosition2"), 1, lightPosition2);
CheckError();
```

The light menu is then updated to link to the new scene object (light 2) so the user can interact with the light.

```
// TASK I: allow light2 to be moved
} else if (id == 80) {
    toolObj = 2;
    setToolCallbacks(adjustLocXZ, camRotZ(),
                     adjustBrightnessY, mat2(1.0, 0.0, 0.0, 10.0));
} else if (id == 81) {
    toolObj = 2;
    setToolCallbacks(adjustRedGreen, mat2(1.0, 0, 0, 1.0),
                     adjustBlueBrightness, mat2(1.0, 0, 0, 1.0));
```

The fragment code is then updated to include exact functionality of the first light, however without the attenuation, and instead with the vertical position of the light as a proportional factor affecting it's intensity/brightness.

```
// TASK I: adjust brightness of light 2 based on y value (height);
float sunFactor = LightPosition2.y/3.5;
LightBrightness2 += sunFactor;

color.rgb = globalAmbient + ambient + diffuse + ambient2 + diffuse2;
color.a = 1.0;

// PART H: seperating specular lighting component from the colour aspect of the light source.
gl_FragColor = (color * texture2D( texture, texCoord * 2.0 * tex_Scale)) + vec4(specular + specular2, 1.0);

// adding some light protection for when the 'sun' is below the 'horizon' (the light has -y coordinates)
if (LightPosition2.y < 0.0) {
    color.rgb = globalAmbient + ((ambient + diffuse) / lightDropoff);
    gl_FragColor = (color * texture2D(texture, texCoord * 2.0 * tex_Scale)) + vec4(specular, 1.0);
}
```

The fragment colours are then updated to include the effect of light2 (ambience, diffusion and specular) and a feature is added to turn the light off when it drops below y = 0, as if the sun has set.

TASK J:
An object can be duplicated by,
Copying the last object (Recent) into the next position of the scene objects array. Then increment the number of objects, and the rotation tool is then selected on the new object. Object can be moved in the plane after this. Mark the normal plane of current window as needing to be redisplayed. The next iteration through **glutMainLoop**, the window's display callback will be called to redisplay the window's normal plane.
A new menu entry is also added to make the function accessible.

```
static void duplicate_Object(int id)
 {
   if (nObjects == maxObjects) return;

sceneObjs[nObjects] = sceneObjs[id];
toolObj = currObject = nObjects++;
setToolCallbacks(adjustLocXZ, camRotZ(),
adjustScaleY, mat2(0.05, 0.0, 0.0, 10.0));
glutPostRedisplay();
}
```

```
else if (id == 90 && currObject >= 0) {
          duplicate_Object(currObject);  }

glutAddMenuEntry("Duplicate Recent-Object", 90);
```

Deletion is also done the similar way as duplication but the last object in the scene objects array is removed. The number of objects is decremented, and the current object is replaced to the one be last one in the array. Finally the glutpostRedisplay() is called.

```
static void Delete_Object(int id) {
   if (nObjects == 3) return;

   nObjects--;

   if(nObjects>3){
      currObject=nObjects-1;
      }
   else{
       currObject=-1;
       }
   toolObj = -1;
   doRotate();
   glutPostRedisplay();
}
```

```
else if (id == 91 && currObject >= 0) {
      Delete_Object(currObject);
      }
glutAddMenuEntry("Delete Recent-Object", 91);
```

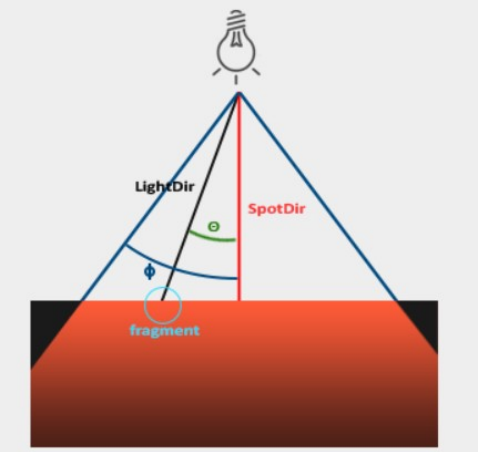
SPOTLIGHT: incomplete

The implementation of a spotlight began similarly to TASK I, with the addition of a new scene object with some of its attributes passed to the shader files. There are also new menu items required in the light menu including a rotation tool for the direction of the spotlight. This was implemented exactly the same as the menu item for rotating an object. The angles of the light are passed to the shader files below.

```
// TASK J
glUniform4fv(glGetUniformLocation(shaderProgram, "LightPosition3"), 1, lightPosition3);
glUniform3fv(glGetUniformLocation(shaderProgram, "LightDir"), 1, lightObj3.angles);
CheckError();
```

```
} else if (id == 84) {
    currObject = 3;
    // add rotational component for light direction, taken from main menu
    setToolCallbacks(adjustAngleYX, mat2(400, 0, 0, -400),
                     adjustAngleZTexscale, mat2(400, 0, 0, 15));
}
```

Inside the vertex shader the light requires some new properties the other ones did not, including a cutoff radius which establishes the angle between the edge of the cone and the perpendicular distance between the light source and the fragment. Image from https://learnopengl.com/Lighting/Light-casters



Here is a test case with manual input of the spotlight pointing almost straight down.

```
// light 3 'spotlight'
uniform vec4 LightPosition3;
//uniform vec3 LightDir;
vec3 LightDir = vec3(0.1, -4.0, 0.1);
float cutOff = 0.997;
vec3 diffuse3;
```

The calculations are then performed to determine whether or not the fragment lands inside the spotlight's radius.

```
// TASK J: spotlight calculations
vec3 lightDir = normalize(Lvec3);
float theta = dot(LightDir, lightDir);

if (theta > cutOff && LightPosition3.y > 0.0) {

    float Kd3 = max(dot(lightDir, N), 0.0);
    diffuse3 = Kd3 * DiffuseProduct;
}
```