CITS3003

Graphics and Animation: Project

Submitted to: Dr. Naveed Akhtar

Submitted by: Swastik Raj Chauhan (22556239)
Theoridho Andily (22764884)

Submitted on: 17 May 2021

Due on: 18 May 2021 (1700 hrs)

Contents

Part A

In this section, we were required to add rotations implementations into the code. Since it relates to the viewport/camera operations, we need to look for the view matrix and apply some transformations. In this case, we needed the viewport to be able to move over and around the centre of the origin. It was written on the project specification page to specifically use the *camRotSidewaysDeg* and *camRotUpAndOverDeg* variables. We then used the functions that were a part of the lab which was included in the project named *mat.h* and used the *RotateY* given *camRotSidewaysDeg* as it was the y-axis rotation and *RotateX* given *camRotUpAndOverDeg* as it was the x-rotations. Lastly, we apply those rotations to the viewport matrix called view. Here the order of transformation is important as otherwise, the rotations would not work. Figure 1 shows the code update required.

```
mat4 rotateY = RotateY(camRotSidewaysDeg);
mat4 rotateX = RotateX(camRotUpAndOverDeg);
view = Translate(0.0, 0.0, -viewDist) * rotateX * rotateY; //Multiply to the viewport variable to change the view of angle
```

Figure 1: Apply rotation to view matrix.

Part B

This section requires rotations on the object model in the scene. Here we also use the same function from part A but on specific attributes of the object. To obtain these attributes we used the selection method which was introduced in lecture 5. The values required from the model were the x, y, and z angles to change the direction of the models after the movement. The angle of rotations is also particularly important here, so we start with x rotations, then y and lastly z. We also added the *texScale* variable to the fragment shader to allow the change of the scale as we rotate the object and include it in the *gl_Frag_Color* calculation.

Figure 2 shows the code changes required in the application program and Figure 3 and Figure 4 are changes required in the fragment shader. Figure 5 shows how the texture is being changed when the object is rotated which proves that the function to change texScale work.

```
/* Part B
* Rotating the model by using built in function specified in mat.h which refers to lab 5.
* Here we use object slicing and swizzling which uses the indexing of array using [] and selection (.) operator refer to lecture 5 pg 30.
* each object has angles for x,y,z and we use the rotation matrix from mat.h to transform at each specific angle.
* angle[0] is x, angle[1] is y and angle[2] is z
* Order of transformation does not matter
*/

mat4 rotate = RotateX(sceneObj.angles[0]) *  RotateY(sceneObj.angles[1]) * RotateZ(sceneObj.angles[2]);
mat4 model = Translate(sceneObj.loc) * Scale(sceneObj.scale) * rotate;
```

Figure 2: Apply rotation to the object model.

```
uniform float texScale;
```

Figure 3: Bring texScale into fStart.glsl for changing the scale.

```
gl_FragColor = color * texture2D( texture, texCoord * texScale) + vec4((specular * reduction), 0.0) + vec4(specular2, 0.0) + vec4((specular3 * reduction3), 0.0);
```

Figure 4: Place the texScale into the gl_Frag_Color calculation.

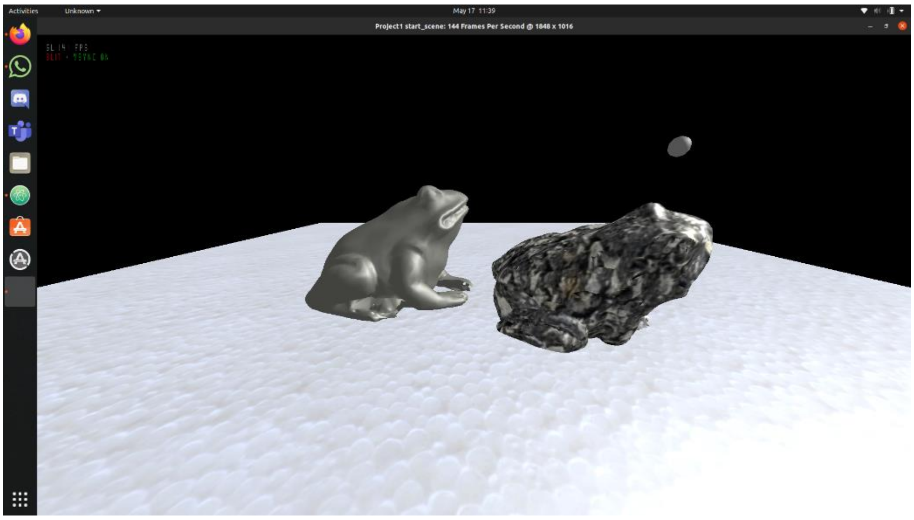

Figure 5: Frog on the left having different texture after a rotation.

Part C

For this section, we needed to understand the setToolCallbacks function which was defined in gnatidred.h. From reading the code, we understood that that setToolCallbacks takes in the parameter as (function-name, identity matrix, function-name, identity matrix). The first function is for the left mouse click and the second function is for the middle mouse click. We create the function to change the ambient and diffuse based on the left mouse click, the specular and shine on the middle mouse button.

We added the menu using glutAddMenuEntry function with a unique id set from Figure 8. In Figure 8 we can see the function working as the right frog model has higher ambient and diffuse value. In addition, Figure 9 shows the frog model on the left with a higher specular and shine value applied from the function while Figure 10 . Figure 6 shows how the function is defined and Figure 8 shows how the menu entry is added.

```
/* Part C
* To understand the fundemental of lighting we used: https://learnopengl.com/Lighting/Basic-Lighting.
* Here we also use swizzling from lecture 5 pg 30  to obtains the values required to change for both functions
* We also followed the styling of parameters from previous functions
* Here we set the two function one to bind to the left mouse button and the second for the middle mouse buttons
* We adjust ambient and diffuse with left mouse button and specular and shine with the middle mouse button
*/

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

}

static void adjustSpecularShine(vec2 ss){
        sceneObjs[toolObj].specular += ss[0];
        sceneObjs[toolObj].shine += ss[1];
}
```

Figure 6: Define the function to adjust ambient, diffuse, specular and shine.

```
    /* PART C
    * Added a call to a tool object and when the referred id is called it will call the function.
    * We use the function setToolCallbacks to refer to the required function menu.
    * The function is called by: setToolCallbacks(functioName1, matrix(2*2): |1, 0| , funcationName2, |1, 0| )
    *                                                                        |0, 1|                   |0, 1|
    * The matrices are used to scale and rotate the effect of the mouse movement vector* from project page.
    * We are also able to move the position of the light source.
    * Hold left mouse button and move them in all 4 directions will move the light source in the x and z axis.
    * Hold down the scroll wheel and move the mouse vertically will move the light source in the y-axis.
    */

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

Figure 7: Add the new menu to the material function.

Figure 8: Add the menu using glutAddMenuEntry().

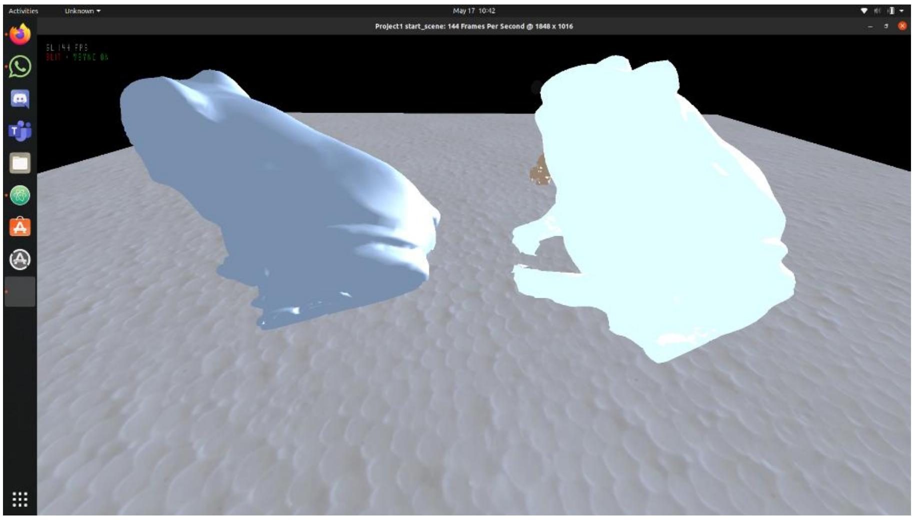

Figure 9: The frog model on the right has higher ambient and diffuse values.

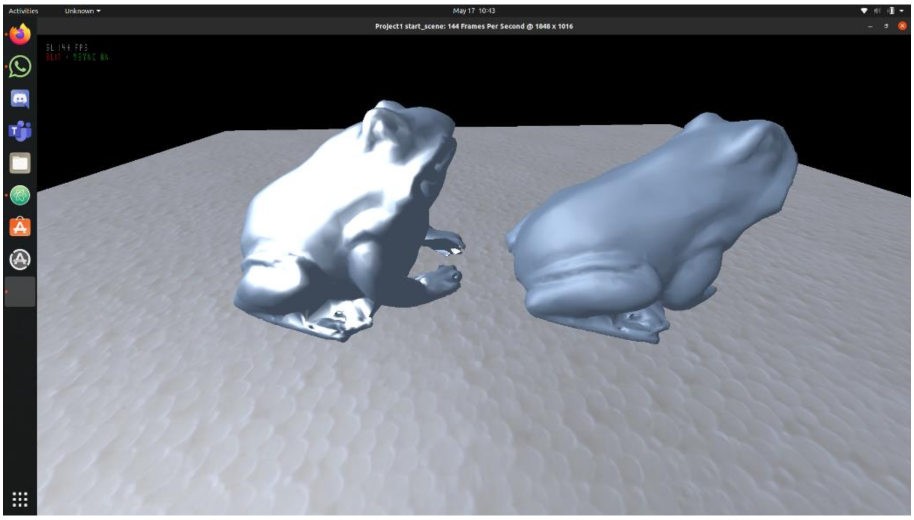

Figure 10: The frog model on the left has higher specular and shine value.

Part D

In this section, we had to change the value of the near parameter in Frustum which is the nearDist value. To increase the viewing volume we decrease the nearDist values, so the objects are not being clipped.

Figure 11 shows the new nearDist value applied to the application program. Figure 12 shows how the face of the dog is being clipped with having nearDist value of 0.2 compared to Figure 13 showing the full face of the dog without being clipped to show a closer shot.

```
/* Part D
 * To fix the clipping of the triangles to show a better close up view
 * we decrease the nearDist to a smaller value as the default value was 0.2.
 * The smaller the unit the more detail it will have when viewed closer to the mesh.
 * Refer to lab 5 Q.2 to the default nearDist value.
 */

GLfloat nearDist = 0.050;
```

Figure 11: Decrease the nearDist value from 0.2 to 0.05.

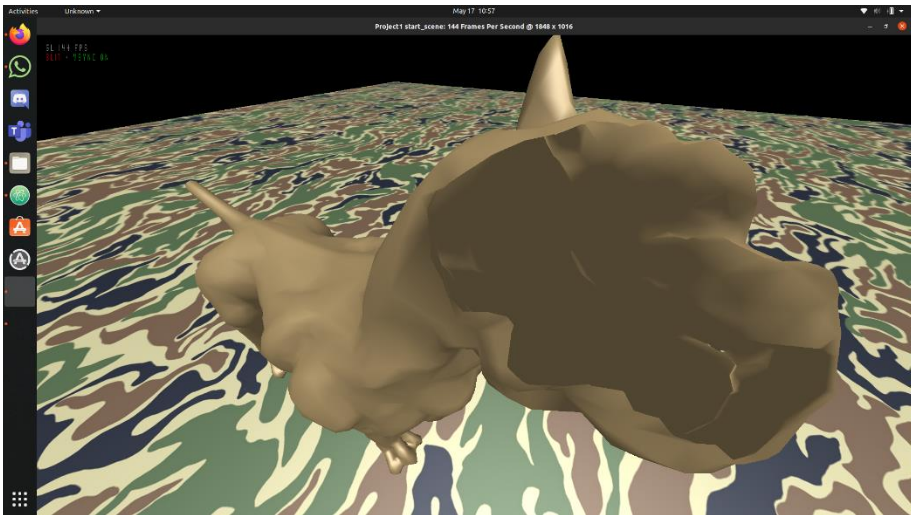

Figure 12: Here is an image of a dog with nearDist value of 0.2 which clips out part of its face.

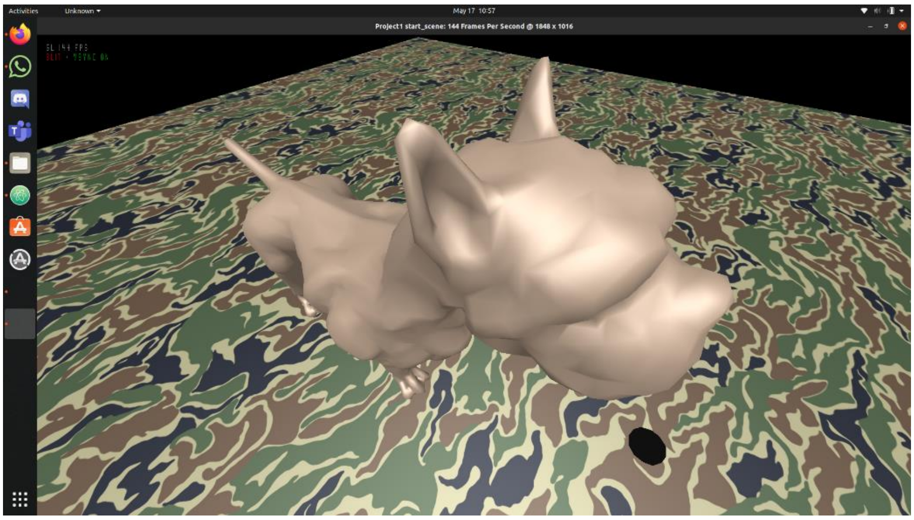

Figure 13: Here is an image of a dog with nearDist value of 0.05 which allows for a closer shot and not having any clipping.

Part E

This section refers to the function Frustum which was defined in the reshape function. To get the window to work properly we had to add an if statement to check whether the width is larger than the height. In the initial skeleton code, the window would reshape fine if the width is larger than height but if the width were bigger than the height it would lose the intended perspective view. In addition, if the height is larger than the width, we change the bottom and top value with respect to the difference in width/height.

Figure 14 shows the changes being made to the application program. Figure 15 shows how the viewport keeps the perspective of the car while the width is more than height and Figure 16 shows how the viewport also keeps the perspective view while having height > width.

```
/* Part E
* We fix the rezising of the window by checking if height is more than width.
* Frustum(left, right, bottom, top), The parameters are the size of window is the referred side.
* If height > width then we change the left and right side projection using Frustum() function.
* If height < width then we change the bottom and top projection using the smae function as previous.
* refer to lab 5 Q.2 for more details in the function used.
*
*/


if(width > height){
    projection = Frustum(-nearDist * (float)width/(float)height,
                          nearDist * (float)width/(float)height,
                         -nearDist,
                          nearDist,
                          nearDist, 100.0);
}

else{
    projection = Frustum(-nearDist,
                          nearDist,
                         -nearDist * (float)height/(float)width,
                         nearDist * (float)height/(float)width,
                         nearDist, 100.0);
}
```

Figure 14: Changes for the reshape function.

Figure 15: Window size with width > height.

Figure 16: Window size with width < height.

Part F

The section relates to light attenuation, i.e., reduction of light with distance. The formula for light attenuation is from Lecture 15's slide 15.

The formula used is $\frac{1}{(a+bd+bd^2)}$, where a, b are constant terms, and d is the distance from the light source.

The attenuation as in the scene can be seen in Figure 17.

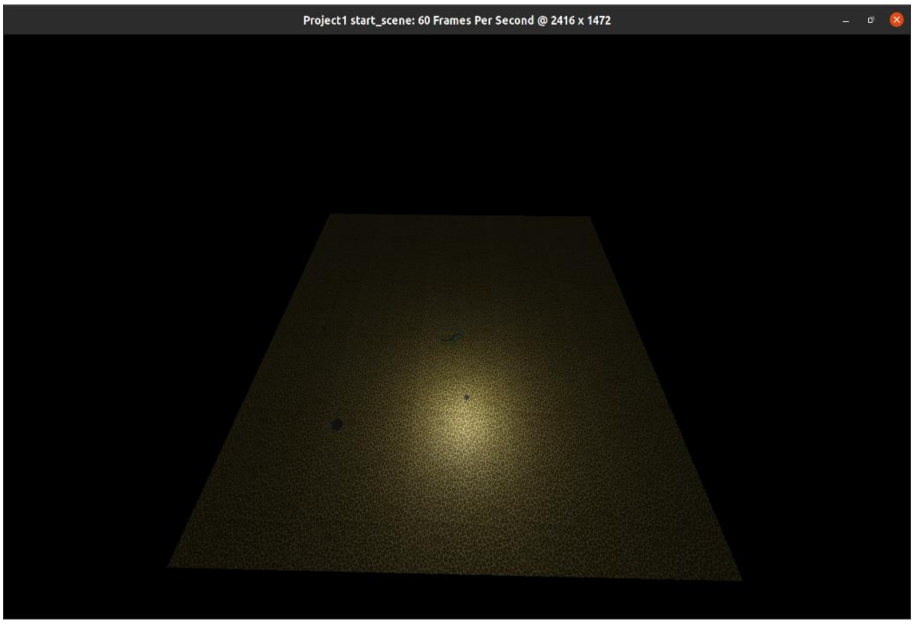

Figure 17: Light attenuation.

The major components for light attenuation in code are in Fragment Shader. Figure 18 and Figure 19 represent code components responsible for light attenuation in the first light source.

```
    // Distance light reduction for light 1
    float d = length(Lvec);
    float b = 0.1;
    float c = 0.1;
    float reduction = 1.0 / (1.0 + (b * d) + (c * d * d));

    // Colour for FragColor
    vec4 color;
    color.rgb = globalAmbient + (( diffuse +  ambient) * reduction) + diffuse2 + ambient2;
        color.a = 1.0;

```

Figure 18: Light attenuation component declaration and implementation for diffuse and ambient components.

```

    gl_FragColor = color * texture2D( texture, texCoord * texScale) + vec4((specular * reduction), 0.0) + vec4(specular2, 0.0) + vec4((specular3 * reduction3), 0.0);
}
```

Figure 19: Implementation of light attenuation in specular component of the first.

Part G

In this section, all the lighting calculations were transferred from Vertex shader to Fragment shader. We implemented Phong Shading Method here. Initially, this was done using ideas in Lecture 17's slides 26 – 30.

Specific changes were made later in the code to make it more suitable for the particular implementation we made. One of the biggest differences from lecture notes implementation was the calculation of fN, fV and fL in fragment shader instead of the vertex shader.

The difference in shading is easily noticeable in plain texture ground as shown in Figure 20.

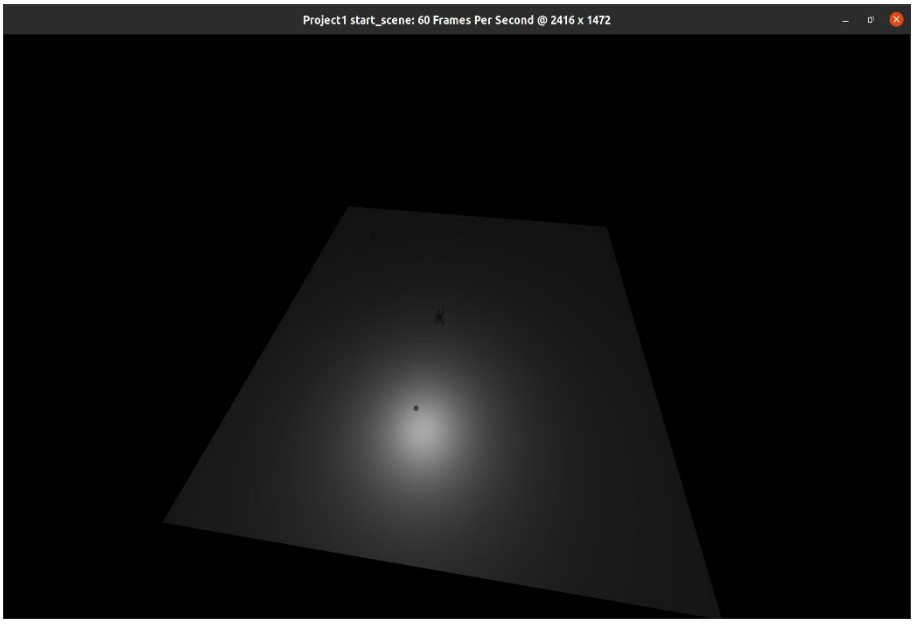

Figure 20: Phong shading model's noticeable difference in plain texture.

Figure 21 represent the entire vertex shader and Figure 22 represent the entire fragment shader. It can be seen in Figure 21 that there is a drastic reduction in code components after adopting Phong shading model and moving all the light calculations to the Fragment shader.

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

varying vec2 texCoord;
varying vec4 position;
varying vec3 normal;

uniform mat4 ModelView;
uniform mat4 Projection;

void main()
{
    position = vec4(vPosition, 1.0);

    normal = vNormal;
    texCoord = vTexCoord;
    gl_Position = Projection * ModelView * vec4(vPosition, 1.0);
}
```

Figure 21: Vertex Shader

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

varying vec4 vPosition;
uniform vec3 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform float Shininess;

// Everything lights needed
uniform vec4 LightPosition1, LightPosition2, LightPosition3;
uniform vec3 LightColor1, LightColor2, LightColor3;
uniform float LightBrightness1, LightBrightness2, LightBrightness3;
uniform vec4 LightDirection;

// Textures
uniform float texScale;
uniform sampler2D texture;

void main()
{

    vec3 pos = (ModelView * position).xyz;

    // The vector to the light from the vertex
    vec3 Lvec = LightPosition1.xyz - pos;
    vec3 Lvec2 = LightPosition2.xyz;
    vec3 Lvec3 = LightPosition3.xyz - pos;

    //normalize
    vec3 L = normalize(Lvec);
    vec3 L2 = normalize(Lvec2);
    vec3 L3 = normalize(Lvec3);

    vec3 E = normalize(-pos);

    vec3 H = normalize(L + E);
    vec3 H2 = normalize(L2 + E);
    vec3 H3 = normalize(L3 + E);

    // Ambient Calculation
    vec3 ambient = AmbientProduct * (LightColor1 * LightBrightness1);
    vec3 ambient2 = AmbientProduct * (LightColor2 * LightBrightness2);
    vec3 ambient3 = AmbientProduct * (LightColor3 * LightBrightness3);

    vec3 N = normalize((ModelView * vec4(normal, 0.0)).xyz);

    // Diffuse Calculation
    float Kd = max(dot(L, N), 0.0);
    float Kd2 = max(dot(L2, N), 0.0);
    float Kd3 = max(dot(L3, N), 0.0);
    vec3 diffuse = Kd * DiffuseProduct * (LightColor1 * LightBrightness1);
    vec3 diffuse2 = Kd2 * DiffuseProduct * (LightColor2 * LightBrightness2);
    vec3 diffuse3 = Kd3 * DiffuseProduct * (LightColor3 * LightBrightness3);

    // Specular Calculation
    float Ks = pow(max(dot(N, H), 0.0), Shininess);
    float Ks2 = pow(max(dot(N, H2), 0.0), Shininess);
    float Ks3 = pow(max(dot(N, H3), 0.0), Shininess);
    vec3 specular = Ks * (SpecularProduct) * (LightColor1 * LightBrightness1);
    vec3 specular2 = Ks2 * (SpecularProduct) * (LightColor2 * LightBrightness2);
    vec3 specular3 = Ks3 * (SpecularProduct) * (LightColor3 * LightBrightness3);
    if(dot(L, N) < 0.0)
    {
        specular = vec3(0.0, 0.0, 0.0);
    }
    if(dot(L2, N) < 0.0)
    {
        specular2 = vec3(0.0, 0.0, 0.0);
    }
    if(dot(L3, N) < 0.0)
    {
        specular3 = vec3(0.0, 0.0, 0.0);
    }

    // globalAmbient is independent of distance from the light source
    vec3 globalAmbient = vec3(0.1, 0.1, 0.1);

    // Distance light reduction for light 1
    float d = length(Lvec);
    float b = 0.1;
    float c = 0.1;
    float reduction = 1.0 / (1.0 + (b * d) + (c * d * d));

    // Colour for FragColor
    vec4 color;
    color.rgb = globalAmbient + (( diffuse +  ambient) * reduction) + diffuse2 + ambient2;
        color.a = 1.0;


    vec3 dL = normalize(LightDirection.xyz);
    vec4 color3;
    if(dot(L3, dL) < 0.5)
    {
        ambient3 = vec3(0.0, 0.0, 0.0);
        diffuse3 = vec3(0.0, 0.0, 0.0);
        specular3 = vec3(0.0, 0.0, 0.0);
    }

    // Distance light reduction for light 3
    float d3 = length(Lvec3);
    float b3 = 0.1;
    float c3 = 0.1;
    float reduction3 = 1.0 / (1.0 + (b3 * d3) + (c3 * d3 * d3));
    color3.rgb = (diffuse3 + ambient3) * reduction3;
        color3.a = 1.0;

    color = color + color3;

    /*Part H
    * https://stackoverflow.com/questions/35917678/opengl-lighting-specular-higlight-is-colored
    */

    gl_FragColor = color * texture2D( texture, texCoord * texScale) + vec4((specular * reduction), 0.0) + vec4(specular2, 0.0) + vec4((specular3 * reduction3), 0.0);
}
```

Figure 22: Fragment Shader

Part H

For part H, we needed to keep the colour of the specular highlight be white and not get affected by the colour of the texture. A solution was to move the specular calculations from color.rgb to gl_FragColour. The overall idea of updating shading method is adopted from here: https://stackoverflow.com/questions/35917678/opengl-lighting-specular-higlight-is-colored.

Figure 23 shows the code changes made to accommodate for specular highlights to be white. Figure 24 shows the colour of the specular highlight reflected from the head model to be white which means that it does not get affected by the texture colour.

```
/*Part H
* https://stackoverflow.com/questions/35917678/opengl-lighting-specular-higlight-is-colored
*/

/*
* Part B for changing texture scale
*/
gl_FragColor = color * texture2D( texture, texCoord * texScale) + vec4((specular * reduction), 0.0) + vec4(specular2, 0.0) + vec4((specular3 * reduction3), 0.0);
```

Figure 23: Move the specular calculation to gl_FragColor.

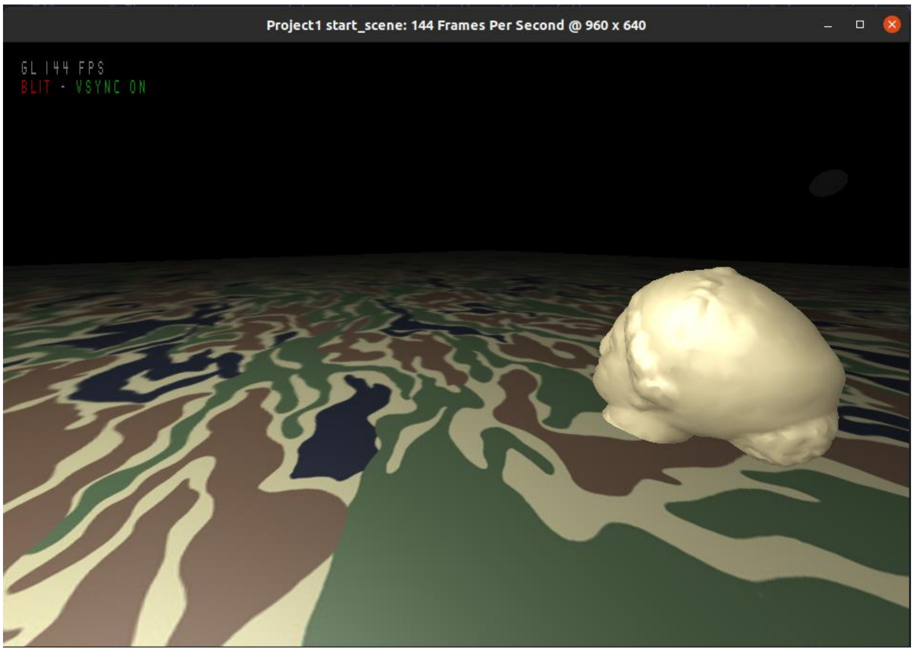

Figure 24: Colour of specular highlight is white.

Part I

We created a second light using the code structure similar to the first light.

- The light is directional,
- The direction of the light is always towards the origin,
- Moving light upward increase the y-component of the light.

The idea of light direction can be explained by Figure 25.

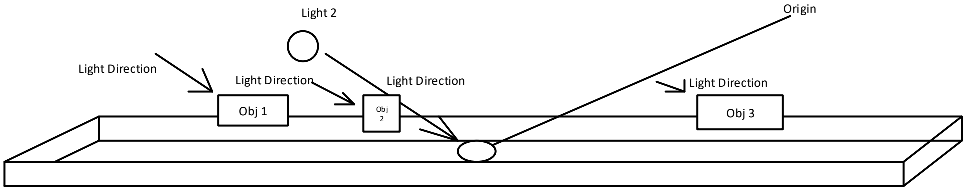

Figure 25: Drawing to show how the light direction would interact with different objects in the scene

Figure 26 shows the second light in the scene. It also depicts how the direction is always towards the origin.

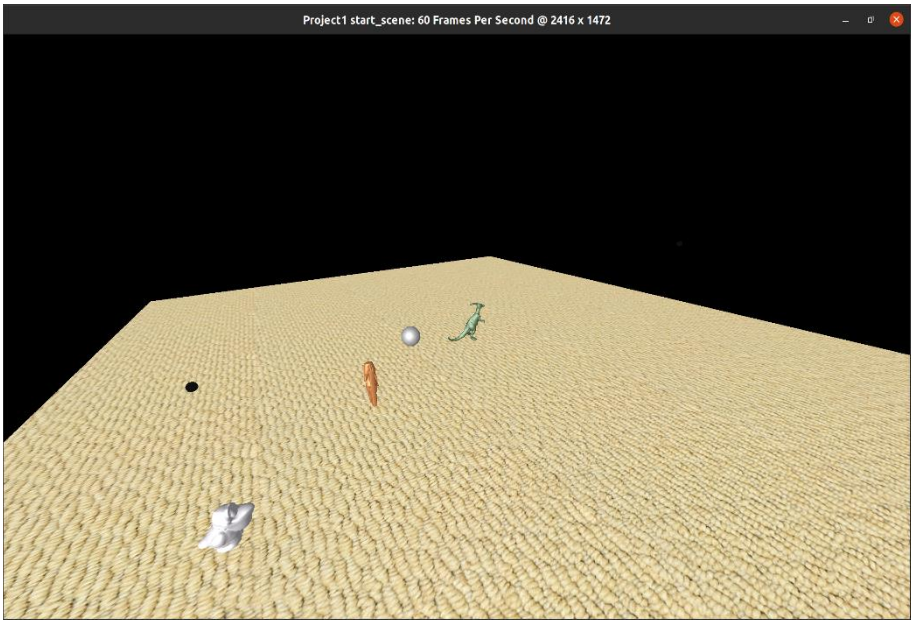

Figure 26: The second light direction is always towards the origin and can be seen in reflection of these objects.

Figure 27 show the code required for the addition of this second light. Figure 28 shows the code which is used to pass the data from the application program to the graphics pipeline. Figure 29 shows the updates for colour and brightness components. Figure 30 and Figure 31 shows the code changes required in the menu items. Figure 32 and Figure 33 encompasses all the code changes made in the Fragment shader to accommodate for the second light.

```
/* Part I adding extra object to store values for light 2
*
*/
addObject(55); // Sphere for the second light
sceneObjs[2].loc = vec4(2.0, 2.0, 1.0, 1.0);
sceneObjs[2].scale = 0.2;
sceneObjs[2].texId = 0; // Plain texture
sceneObjs[2].brightness = 0.2; // The light's brightness is 5 times this (below).
sceneObjs[2].angles[1] = 90.0;
```

Figure 27: Declaration of second light in application program.

```
/* Part I
* Adding an extra light object for display
*/
mat4 origin_perspective = rotateY * rotateX;
SceneObject lightObj2 = sceneObjs[2];
vec4 lightPosition2 = origin_perspective * lightObj2.loc;
glUniform4fv(glGetUniformLocation(shaderProgram, "LightPosition2"),
             1, lightPosition2);
CheckError();
glUniform3fv(glGetUniformLocation(shaderProgram, "LightColor2"),
             1, lightObj2.rgb);
CheckError();
glUniform1f(glGetUniformLocation(shaderProgram, "LightBrightness2"),
            lightObj2.brightness);
CheckError();
```

Figure 28: Passing necessary data to the shader programs from application program. Note that we are not multiplying light position with scene to make it always point towards the origin.

```
for (int i = 0; i < nObjects; i++) {
    SceneObject so = sceneObjs[i];

    // Part I accouring for different lights and brightness and colour calculation from light are now done in shaders
    vec3 rgb = so.rgb * so.brightness * 2.0;
    glUniform3fv(glGetUniformLocation(shaderProgram, "AmbientProduct"), 1, so.ambient * rgb);
    CheckError();
    glUniform3fv(glGetUniformLocation(shaderProgram, "DiffuseProduct"), 1, so.diffuse * rgb);
    glUniform3fv(glGetUniformLocation(shaderProgram, "SpecularProduct"), 1, so.specular * rgb);
    glUniform1f(glGetUniformLocation(shaderProgram, "Shininess"), so.shine);
    CheckError();

    drawMesh(sceneObjs[i]);
}
```

Figure 29: Moving RGB components calculation with respect to light object of ambient, diffuse, and specular component of lights from application program to fragment shader.

```
/* Part I
* Adding extra menus for the second light
*/
else if (id == 80) {
     toolObj = 2;
     setToolCallbacks(adjustLocXZ, camRotZ(),
                      adjustBrightnessY, mat2(1.0, 0.0, 0.0, 10.0));
} else if (id >= 81 && id <= 84) {
    toolObj = 2;
    setToolCallbacks(adjustRedGreen, mat2(1.0, 0, 0, 1.0),
                     adjustBlueBrightness, mat2(1.0, 0, 0, 1.0));
```

Figure 30: Light menu for second light.

```
glutAddMenuEntry("Move Light 2", 80);
glutAddMenuEntry("R/G/B/All Light 2", 81);
```

Figure 31: Appropriate menu item for second light.

```
// Everything lights needed
uniform vec4 LightPosition1, LightPosition2, LightPosition3;
uniform vec3 LightColor1, LightColor2, LightColor3;
uniform float LightBrightness1, LightBrightness2, LightBrightness3;
```

Figure 32: Light position, colour, and brightness for second light in fragment shader.

```
vec3 Lvec2 = LightPosition2.xyz;
vec3 L2 = normalize(Lvec2);
vec3 H2 = normalize(L2 + E);
vec3 ambient2 = AmbientProduct * (LightColor2 * LightBrightness2);
float Kd2 = max(dot(L2, N), 0.0);
vec3 diffuse2 = Kd2 * DiffuseProduct * (LightColor2 * LightBrightness2);
float Ks2 = pow(max(dot(N, H2), 0.0), Shininess);
vec3 specular2 = Ks2 * (SpecularProduct) * (LightColor2 * LightBrightness2);
if(dot(L2, N) < 0.0)
{
    specular2 = vec3(0.0, 0.0, 0.0);
}
// Colour for FragColor
vec4 color;
color.rgb = globalAmbient + (( diffuse +  ambient) * reduction) + diffuse2 + ambient2;
    color.a = 1.0;
gl_FragColor = color * texture2D( texture, texCoord * texScale) + vec4((specular * reduction), 0.0) + vec4(specular2, 0.0) + vec4((specular3 * reduction3), 0.0);
```

Figure 33: All the relevant calculations related to light 2 in fragment shader.

Part J

Part a

In this part, we added a function to delete the current object being chosen. This raises an issue which was that we did not know which object to choose or just delete from the last object.

The deletion function limits the user to only delete the object which does not include any lights or the ground object. In assisting in choosing the object we added a feature to change the current object chosen. We included a keyboard key function, if an ‘a’ key is pressed it will move the current object to be the previous object and if a ‘d’ key is pressed it will change to the next object. These changes are not instant and thus, the user needs to bring up the menu and pick an attribute to change after pressing either ‘a’ key or ‘d’ key.

Figure 34 shows how the function was made in the application program and figure 35 shows the changes made to accommodate the keyboard functions.

```
/*
* Part J.a defining  the deletion function
*/

//------Delete an object to the scene-------------------------------------------
static void deleteObject(int id) {
    // Check if there is an object to delete
    if (nObjects <= 2) {
        // Do nothing
        return;
    }

    // If the current object is last
    else if (currObject == nObjects - 1){
        // Decrease the number of object
        nObjects--;
        currObject--;
    }

    // If the current object is not last
    else {
        // Decrease the number of object
        for (int i=currObject; i < nObjects; i++) {
            sceneObjs[i] = sceneObjs[i+1];
        }
        nObjects--;
        glutPostRedisplay();
    }
}
```

Figure 34: Function to delete an object.

```
void keyboard(unsigned char key, int x, int y) {
    switch (key) {
        case 033: {
            exit(EXIT_SUCCESS);
            break;
        }
        // Select previous object
        case  'a':

            // Make sure the index doesn`t change any lights or grounds
            if (currObject >= 5) {
                currObject--;
            }
            break;

        // Select next object
        case  'd':

            // Make sure the index doesn`t go over the number of object
            if(currObject != nObjects - 1){
                currObject++;
                toolObj++;
            }
            break;
        case 'w': {
            if (glutGetModifiers() == GLUT_ACTIVE_ALT) { // up + alt
                zoomIn();
            }
            break;
        }
        case 's': {
            if (glutGetModifiers() == GLUT_ACTIVE_ALT) { // down + alt
                zoomOut();
            }
            break;
        }
    }
}

void specialKeys(int key, int x, int y) {
    switch (key) {
        case GLUT_KEY_UP: {
            if (glutGetModifiers() == GLUT_ACTIVE_ALT) { // up + alt
                zoomIn();
            }
            break;
        }
        case GLUT_KEY_DOWN: {
            if (glutGetModifiers() == GLUT_ACTIVE_ALT) { // down + alt
                zoomOut();
            }
            break;
        }
    }
}
```

Figure 35: Keyboard function to move current object.

Part b

In this section, we had to create a duplicate function to duplicate the current object. This function will create a new object by the addObject() function created in skeleton code and as well as passing the attribute of the current object being duplicated into the new object created. The location is slightly different to be able to easily differentiate the position of the new object compared to the previous object. Figure 36 shows how the function is made in the application program.

```
/*
* Part J.b defining  the duplication function
*/

//------Duplicate an object to the scene------------------------------------------

static void duplicateObject(int id)
{
    // Check if there is an object to duplicate
    if(nObjects <= 4){
        // Do nothing
        return;
    }

    else {
        // Create the newObject and add it to the stack
        addObject(sceneObjs[id].meshId);

        // Set the values that should be the same from the previous object
        sceneObjs[currObject].scale = sceneObjs[id].scale;
        sceneObjs[currObject].loc = sceneObjs[id].loc+0.1;
        sceneObjs[currObject].texId = sceneObjs[id].texId;
        sceneObjs[currObject].texScale = sceneObjs[id].texScale;

        sceneObjs[currObject].rgb[0] = sceneObjs[id].rgb[0];
        sceneObjs[currObject].rgb[1] = sceneObjs[id].rgb[1];
        sceneObjs[currObject].rgb[2] = sceneObjs[id].rgb[2];

        sceneObjs[currObject].brightness = sceneObjs[id].brightness;
        sceneObjs[currObject].diffuse = sceneObjs[id].diffuse;
        sceneObjs[currObject].specular = sceneObjs[id].specular;
        sceneObjs[currObject].ambient = sceneObjs[id].ambient;
        sceneObjs[currObject].shine = sceneObjs[id].shine;

        sceneObjs[currObject].angles[0] = sceneObjs[id].angles[0];
        sceneObjs[currObject].angles[1] = sceneObjs[id].angles[1];
        sceneObjs[currObject].angles[2] = sceneObjs[id].angles[2];

        setToolCallbacks(adjustLocXZ, camRotZ(),
                         adjustScaleY, mat2(0.05, 0, 0, 10.0) );
        glutPostRedisplay();
    }
}
```

Figure 36: Function to duplicate an object.

Part c

We created a spotlight using the code structure similar to the first light.

- The light's position can be changed,
- The illumination direction/spot direction/cone direction can be adjusted interactively.

One of the biggest differences in the spotlight against an omnidirectional light was setting up the angle of illumination. The specific code snippet to set it up can be seen in Figure 37.

We calculated dot product between light vector and directional vector to get a cut off angle based on idea presented at https://learnopengl.com/Lighting/Light-casters.

```
vec3 dL = normalize(LightDirection.xyz);
vec4 color3;
if(dot(L3, dL) < 0.5)
{
    ambient3 = vec3(0.0, 0.0, 0.0);
    diffuse3 = vec3(0.0, 0.0, 0.0);
    specular3 = vec3(0.0, 0.0, 0.0);
}
```

Figure 37: The code snippet which limits the cone illumination angle.

Figure 38 shows the spotlight in scene. The spotlight has a specific diameter which is lit up and is in an angle not directly facing down towards the ground.

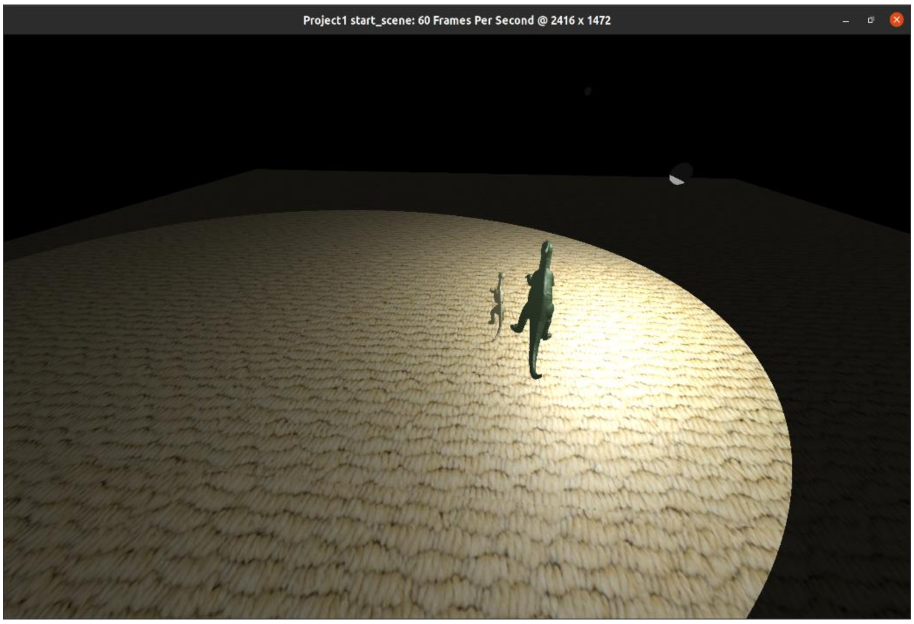

Figure 38: Spotlight at an angle illuminating the 2 objects.

Figure 39 show the code required for the addition of spotlight light. Figure 40 shows the code which is used to pass the data from application program to the graphics pipeline. Figure 41 shows the functions created to allow angle adjustment for the spotlight. Figure 42 and Figure 43 shows the code changes required in the menu items. Figure 44 and Figure 45 encompasses all the code changes made in the Fragment shader to accommodate for the spotlight.

```
/* Part J 3 adding extra object to store values for light 3
*
*/
addObject(55); // Sphere for the third light
sceneObjs[3].loc = vec4(3.0, 1.0, 1.0, 1.0);
sceneObjs[3].scale = 0.1;
sceneObjs[3].texId = 0; // Plain texture
sceneObjs[3].brightness = 0.2; // The light's brightness is 5 times this (below).
```

Figure 39: Declaration of spotlight in application program.

```
/* Part J  3
* Adding an extra light object for display
*/
SceneObject lightObj3 = sceneObjs[3];
vec4 lightPosition3 = view * lightObj3.loc;
glUniform4fv(glGetUniformLocation(shaderProgram, "LightPosition3"),
             1, lightPosition3);
CheckError();
glUniform3fv(glGetUniformLocation(shaderProgram, "LightColor3"),
             1, lightObj3.rgb);
CheckError();
glUniform1f(glGetUniformLocation(shaderProgram, "LightBrightness3"),
            lightObj3.brightness);
CheckError();
```

Figure 40: Passing necessary data to the shader programs from application program.

```
static void adjustAngleYX_spot(vec2 angle_yx) {
    sceneObjs[3].angles[1] += angle_yx[0];
    sceneObjs[3].angles[0] += angle_yx[1];
}
```

Figure 41: Allows adjusting spot angle.

```
/* Part J 3
* Adding extra menus for the spotlight
*/
else if (id == 90) {
     toolObj = 3;
     setToolCallbacks(adjustLocXZ, camRotZ(), // change funnel
                      adjustBrightnessY, mat2(1.0, 0.0, 0.0, 10.0));
} else if (id >= 91 && id <= 94) {
    toolObj = 3;
    setToolCallbacks(adjustRedGreen, mat2(1.0, 0, 0, 1.0),
                     adjustBlueBrightness, mat2(1.0, 0, 0, 1.0));

}
else if (id == 95)  {
    toolObj = 3;
    setToolCallbacks(adjustAngleYX_spot, mat2(400, 0, 0, -400),
                     adjustBrightnessY, mat2(1.0, 0.0, 0.0, 10.0));
}
```

Figure 42: Light menu for spotlight.

```
glutAddMenuEntry("Move Light 3", 90);
glutAddMenuEntry("R/G/B/All Light 3", 91);
glutAddMenuEntry("Direction  Light 3", 95);
```

Figure 43: Appropriate menu item for spotlight.

```
// Everything lights needed
uniform vec4 LightPosition1, LightPosition2, LightPosition3;
uniform vec3 LightColor1, LightColor2, LightColor3;
uniform float LightBrightness1, LightBrightness2, LightBrightness3;
```

Figure 44: Light position, colour, and brightness for spotlight in fragment shader.

```
    vec3 Lvec3 = LightPosition3.xyz - pos;
    vec3 L3 = normalize(Lvec3);
    vec3 H3 = normalize(L3 + E);
    vec3 ambient3 = AmbientProduct * (LightColor3 * LightBrightness3);
    float Kd3 = max(dot(L3, N), 0.0);
    vec3 diffuse3 = Kd3 * DiffuseProduct * (LightColor3 * LightBrightness3);
    float Ks3 = pow(max(dot(N, H3), 0.0), Shininess);
    vec3 specular3 = Ks3 * (SpecularProduct) * (LightColor3 * LightBrightness3);
    if(dot(L3, N) < 0.0)
    {
        specular3 = vec3(0.0, 0.0, 0.0);
    }
    vec3 dL = normalize(LightDirection.xyz);
    vec4 color3;
    if(dot(L3, dL) < 0.5)
    {
        ambient3 = vec3(0.0, 0.0, 0.0);
        diffuse3 = vec3(0.0, 0.0, 0.0);
        specular3 = vec3(0.0, 0.0, 0.0);
    }

    // Distance light reduction for light 3
    float d3 = length(Lvec3);
    float b3 = 0.1;
    float c3 = 0.1;
    float reduction3 = 1.0 / (1.0 + (b3 * d3) + (c3 * d3 * d3));
    color3.rgb = (diffuse3 + ambient3) * reduction3;
        color3.a = 1.0;

    color = color + color3;
    gl_FragColor = color * texture2D( texture, texCoord * texScale) + vec4((specular * reduction), 0.0) + vec4(specular2, 0.0) + vec4((specular3 * reduction3), 0.0);
```

Figure 45: All the relevant calculations related to spotlight in fragment shader.