CITS3003 Graphics and Animation Project Report
William Au (22257005)

Overview

The code fulfils all the required functionalities from Part A to Part J(b) as per the project outline. Part J(c) was not completed.

Implementation

Part A – Camera Control

The display callback needed to be modified for the camera to properly rotate. We used functions *RotateX* and *RotateY* with the camera-angle variables *camRotUpAndOver* and *camRotSidewaysDeg* to implement this. The functions were part of the *mat.h* library and the variables were declared in the skeleton code.

```
mat4 rotate = RotateX(camRotUpAndOverDeg) * RotateY(camRotSidewaysDeg);
view = Translate(0.0, 0.0, -viewDist) * rotate;
```

NOTE: The code provided contained a bug activating a tool anywhere on the screen would cause the camera to reset and move back to before the tool was activated. This proved problematic as tools would essentially be unable to be operate properly. Although recommended against doing so in the comments of the skeleton code, an edit needed to be made to the gnatiread.h.

```
prevPos = currMouseXYscreen(mouseX, mouseY);
```

Part B – Object Rotation

Allowing for object rotation required a rotation matrix to be added. The skeleton code already provided object Scaling and Translation so it was simply a matter of creating the rotation matrix and adding it to the model matrix. Since the multiplication calculations are done from right to left, we need to scale the object, rotate the object then translate the object. In the same way, we need the rotation matrix to rotate X, Y, then Z as shown in the code. This way, we are able to replicate the rotations that are shown in the video.

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

In order to implement the texture scaling feature as shown in the video, a new variable texScale needed to be added to the shader (later moved from the vertex shader to the fragment shader)

```
uniform float texScale;
```

the *gl_FragColor* calculation also needed to be modified to properly show the texture scaling.

```
gl_FragColor = color * texture2D( texture, texCoord * 2.0 * texScale);
```

Part C – Ambient/Diffuse/Specular/Shine

Two call-back functions needed to be created in order to modify ambient, diffuse, specular and shine levels.

```
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];
}
```

These two callback functions were called when the menu entry to modify such values was selected.

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

We also needed to rename the menu entry since it was unimplemented originally

```
glutAddMenuEntry("Ambient/Diffuse/Specular/Shine",20);
```

Part D – Clipping

In the reshape callback, the *nearDist* variable was already provided in the skeleton code. In order to reduce clipping when the object was close to the camera, we simply needed to reduce the value of the *nearDist* variable to allow the camera to give more “close up” views of objects. I found that the value *0.075* gave the closest result to what was shown in the video.

```
GLfloat nearDist = 0.075;
```

Part E – Window Resizing

In order to have all objects that were visible in the window when the window was square, to also be visible when the window was stretched or shortened horizontally, we needed to modify the reshape call-back function. Since this performed as expected when the width was greater than the height, we simply needed to modify the parameter inputs for the *Frustum()* function when the window was less than the height.

```
GLfloat left, right, bot, top;

if(width < height){
        left = -nearDist;
        right = nearDist;
        bot = -nearDist * (float)height/(float)width;
        top = nearDist * (float)height/(float)width;
}
```

```
else{
        left = -nearDist * (float)width/(float)height;
        right = nearDist * (float)width/(float)height;
        bot = -nearDist;
        top = nearDist;
}

projection = Frustum(left, right, bot, top, nearDist, 100.0);
```

Part F – Light Reduction based on Distance

By creating a variable *lightDist* to represent the distance the light was to the object and adding it to the lighting calculations for the object, we could have the light apply to varying degrees depending on that distance.

```
float lightDist = 0.01 + length(Lvec);

color.rgb = globalAmbient + ((ambient + diffuse) / lightDist);
```

Part G – Lighting Calculations in the Fragment Shader

Moving the lighting calculations from the vertex shader to the fragment shader changed how many of the variables and functions behaved and needed to be declared.

The position vector needed to be a vec4 rather than a vec3 to avoid a size and type mismatch for the matrix multiplication with the *ModelView* matrix.

```
vec3 pos = (ModelView * position).xyz;
```

This multiplication was required when transforming the vertex position into eye coordinates. Additional variables were created in both the vertex shader and the fragment shader - some of which also needed their type qualifiers changed.

NOTE: I did not test this until completing Part H, hence the extra light variables are not discussed here. As I was going through Part H and testing it, I noticed my lights were not working as expected and hence needed to fix both Part G and Part H, resulting in the extra variables being added as shown below.

Part H – Shine

Several light variables needed to be added for the light to work as expected in the fragment shader.

```
uniform vec4 LightPosition;
uniform vec3 LightColor;
uniform float LightBrightness;
```

These variables were already used in scene-start.cpp as part of the skeleton code but were not used in the vertex shader.

These variables, in particular *LightBrightness* were also essential in being able to manipulate the shine of the light on an object.

The program needed to be able to pass the light brightness to the fragment shader hence the following code was used.

```
glUniform1f(glGetUniformLocation(shaderProgram, "LightBrightness"),
lightObj1.brightness);
CheckError();
```

The lighting calculations also needed to be modified to account for the new variable so that it would perform as expected.

```
vec3 ambient = (LightColor * LightBrightness) * AmbientProduct;
vec3 diffuse = (LightColor * LightBrightness) * Kd*DiffuseProduct;
vec3 specular = LightBrightness * Ks * SpecularProduct;

color.rgb = globalAmbient + ((ambient + diffuse) / lightDist) ;
color.a = 1.0;

gl_FragColor = color * texture2D( texture, texCoord * 2.0 * texScale) +
vec4(specular/lightDist, 1.0);
```

By adding these variables, we were able to modify the specular attribute of the light. As a result, we could make it so that the lighting was independent of the light's colour and ensure that the specular component always shined towards white.

Part I – Second Light

Adding a second light meant duplicating many of the variables and calculations that were made in for the first light.

```
uniform vec4 LightPosition2;
uniform vec3 LightColor2;
uniform float LightBrightness2;
```

Slight modifications needed to be made most variables and calculations to make the two lights independent to each other.

Since the second light is directional, its lighting calculation is affected by camera rotations but not object rotation or translation.

(In Fragment Shader *fStart.glsl*)

```
vec3 Lvec2 = LightPosition2.xyz;
...
vec3 L2 = normalize( Lvec2 );   // Direction to the light source
...
vec3 H2 = normalize( L2 + E );  // Halfway vector
...
vec3 ambient2 = (LightColor2 * LightBrightness2) * AmbientProduct;

float Kd2 = max( dot(L2, N), 0.0);
...
vec3 diffuse2 = (LightColor2 * LightBrightness2) * Kd2*DiffuseProduct;
...
float Ks2 = pow( max(dot(N, H2), 0.0), Shininess );
...
vec3 specular2 = LightBrightness2 * Ks2 * SpecularProduct;
...
if (dot(L2, N) < 0.0 ) {
specular2 = vec3(0.0, 0.0, 0.0);
```

(In Program *scene-start.cpp*)

```
void init(void)
{
        ...
        addObject(55); // Sphere for the second light
        sceneObjs[2].loc = vec4(2.0, 1.0, 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).
        ...
}
...
void display(void)
{
        ...
        SceneObject lightObj2 = sceneObjs[2];
        vec4 lightPosition2 = view * lightObj2.loc;
        ...
        glUniform4fv(glGetUniformLocation(shaderProgram, "LightPosition2"), 1,
        lightPosition2); //(Part I)
        CheckError();
        ...
        glUniform3fv(glGetUniformLocation(shaderProgram, "LightColor2"), 1,
        lightObj2.rgb); //(Part I)
        CheckError();
        ...
        glUniform1f(glGetUniformLocation(shaderProgram, "LightBrightness2"),
        lightObj2.brightness); // (Part I)
        CheckError();
        ...
}
```

Part J – Extra Menu Options

J(a) – Deleting Objects

The menu entry "*Delete Object*" was created to delete the currently selected object.

```
if(id == 42 && currObject >= 0){
        deleteObject(currObject);
}
```

The function *deleteObject()* was created similarly to how *addObject* was created
It sets the *meshID* value of the object to *NULL*, this removes them from the scene.

One concern that arose from this was the fact that although the objects were removed from the scene, the object itself was not truly "deleted" as they still existed.

An issue I also came across was the fact that the program did not properly select the previous object and hence the previous object could not be moved or manipulated in any way after deleting an object. Similarly, we could not duplicate and object after deleting an object due to the same issue, as it would seemingly duplicate/modify nothing. This issue was not present when adding or duplicating objects as shown in Part J(b)

NOTE: An idea was explored where rather than creating unique functions to perform these actions, they were simply carried out as the callback function was called, however there were issues in the modifying the toolObj, currObj and nObjects variables this way.

```
static void deleteObject(int id){
        sceneObjs[currObject].meshId = NULL;
        toolObj = currObject = nObjects--;
        setToolCallbacks(adjustLocXZ, camRotZ(),
        adjustScaleY, mat2(0.05, 0, 0, 10.0) );
        glutPostRedisplay();
}
```

J(b) – Duplicating Objects

The menu entry "*Duplicate Object*" was created to duplicate the currently selected object.

```
if(id == 43 && currObject >= 0){
        duplicateObject(currObject);
}
```

The function *duplicateObject* was created similarly to how the *addObject* was created.
It sets the next element in *sceneObjs[]* to be equal to the element in *currObj* position in the array.

toolObj, *currObj* and *nObjects* are all incremented to select the object created and increase the number of objects in the scene.

```
static void duplicateObject(int id){
        sceneObjs[nObjects] = sceneObjs[id];
        toolObj = currObject = nObjects++;
        setToolCallbacks(adjustLocXZ, camRotZ(),
        adjustScaleY, mat2(0.05, 0, 0, 10.0) );
        glutPostRedisplay();
    }
```

J(c) – Adding a spotlight

This function was not implemented.