# CITS3003 Graphics and Animation
# Project Part 1 Report

Bruce How (22242664)

May 15, 2019

## 1 Overview

The submitted code covers all the required project functionalities including some additional code which contributes to the optional part of the project. Contribution breakdown is not required as I have worked individually.

My progress throughout this project was not "bug-free". When working on the first part of the project, I noticed that the scroll wheel did not zoom the camera at all. To debug this, I added a simple print statement in the *mouseClickOrScroll* call-back function to see if the function was called when the scroll wheel was triggered. This was not the case. Fortunately, camera zooming can still be achieved by clicking and dragging the mouse. GLUT implementations, such as **FreeGLUT**, which contains the *glutMouseWheelFunc* function, is another alternative solution to this issue, but will be ignored in this project. This issue seems to be specific to Mac users.

Testing my scene builder was extremely hard on the eye due to the very low default brightness. I have doubled the brightness for each object in the scene to increase visibility.

## 2 Implementation

### A Camera Rotatation

For the camera rotation implementation, the display callback function had to be modified. This was done using two built in functions, *RotateX* and *RotateY*; and two camera angle variables, *camRotUpAndOverDeg* and *camRotSidewaysDeg*, which were already implemented in the skeleton code.

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

I came across a weird issues when implementing this part of the project where the camera would snap-back after the tool was activated. This issue was resolved by modifying a line

of code in the *activateTool* function, in *gnatidread.h*. The redundant variable *clickPrev*, was also removed. The code modification is shown below.

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

## B Object Rotatation

Object translation and scaling have already been implemented in the skeleton code. The variable *angles[3]*, is used to control the object's rotation. The same built-in functions used above have also been used to rotate the object.

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

I had to negate the rotation on the x-axis (*-sceneObj.angles[0]*) in order to a produce a rotation that is identical to the one in the video.

## C Materials

Two call-back functions were created which modify the ambient, diffuse, specular and shine values respectively.

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

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

These call-back functions are used by a menu object with an id of 20 as shown below.

```
else if (id == 20) { // Ambient/Diffuse/Specilar/Shine
  setToolCallbacks(adjustAmbientDiffuse, mat2(1, 0, 0, 1),
                adjustSpecularShine, mat2(1, 0, 0, 1));
}
```

The menu entry name has also been renamed to remove the "UNIMPLEMENTED" label.

## D Clipping

In order to avoid close-up clipping, the value of the built-in variable *nearDist* in the *reshape* call-back function needed to be modified.

```
GLfloat nearDist = 0.01;
```

By reducing the value of *nearDist* to a smaller value of 0.01, the camera is able to move in much closer to the objects without its triangles being clipped. I have found that reducing the value of *nearDist* further had little to no impact.

## E Reshape

Additional modifications to the reshape call-back function was also required in order to make sure that all objects that are visible when the window is a square, are also visible when the window is stretched or shortened.

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

## F Light Reduction

The *reduction* variable is applied to the light calculation so that lighting would vary depending on the distance to the light source.

```
// Reduction in light after a particular distance
float lightDropoff = 0.01 + length(Lvec);

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

## G Per-Fragment Lighting

Per-fragment lighting was achieved by moving the lighting calculations from the vertex shader to the fragment shader, *fStart.glsl*. As the light directions are calculated for each individual fragments, light interaction on surfaces are much more smoother.

Several changes were required in order for this to happen including the need for additional variables in the vertex shader, *vStart.glsl*, and the fragment shader *fStart.glsl*. No changes were required in the cpp file *scene-start.cpp*.

## H Shine

The following code was added in the *scene-start.cpp* file to pass the light brightness to the fragment shader.

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

This allows me to modify the specular attribute ensuring that the lighting is independent of the light's colour and that the light's color tend towards white. It should also not affect the colour of the texture. The below code was added to the fragment shader *fStart.glsl*.

```
float Ks = pow(max(dot(N, H), 0.0), Shininess);
vec3 specular = LightBrightness * Ks * SpecularProduct;

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

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

## I Additional Light

The creation of an additional light is similar of that of the first light. The below code was added in the *scene-start.cpp* file.

```
// Second light
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).
```

Several new variables were also declared in the fragment shader including *LightPosition2*, *LightColor2* and *LightBrightness2*. These variables are initialised in the *scene-start.cpp* file accordingly.

```
// Light Object 2
uniform vec4 LightPosition2;
uniform vec3 LightColor2;
uniform float LightBrightness2;
```

As the second light is **directional**, its lighting calculation is only be affected by camera rotations and not rotation nor translation (code below). The remaining *ambient*, *diffuse* and *specular* calculations were performed in the same manner as the first light.

```
// Set the view matrix
mat4 rotate = RotateX(camRotUpAndOverDeg) * RotateY(camRotSidewaysDeg);

// Second light
SceneObject lightObj2 = sceneObjs[2];
vec4 lightPosition2 = rotate * lightObj2.loc;
```

Changes to the light menu were also performed to link the second light object *lightObj2*. Light reduction does not apply to directional light sources and is therefore not implemented for this light, but directly added to the end of *gl_FragColor*.

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

# J Extensions

## J.1 Below Surface Lighting

I have implemented code in the fragment shader and the cpp file to ensure that both lights will have no effect if they are beneath the surface. This is done by checking the y locations of each light in the fragment shader and omitting particular lighiting calculations where neccessary. In order to do this, the location of each light, *lightObj.loc*, needed to be passed to the fragment shader.

```
glUniform4fv(glGetUniformLocation(shaderProgram, "LightObj"), 1, lightObj1.loc);
    CheckError();
glUniform4fv(glGetUniformLocation(shaderProgram, "LightObj2"), 1, lightObj2.loc);
    CheckError();
```

## J.2 Object Selection

My initial plan was to utilise *stencil* to allow for easy object selection via the mouse. I ended up implementing a simpler alternative by adding an additional submenu that allowed objects to be individually selected.

Each submenu item represents an object in the scene (excluding the ground and light), and has a menu ID of 100 plus its respective index in the sceneObjs[] array. This ensures that each ID object is unique. Note that objects with no *meshID* are excluded (explained in the Object Deletion section), and the object that is selected is represented in the sub-menu with an asterisk.

```
int selectObjMenuId = glutCreateMenu(selectObjectMenu);
for (int i = 3; i < nObjects; i++) { // Exclude ground, lightObj1 and lightObj2
    char objectName[128]; // Same size used in gnatidread.h
    if (sceneObjs[i].meshId != NULL) {
```

```
        int objectId = 100 + i;
        strcpy(objectName, objectMenuEntries[sceneObjs[i].meshId - 1]);
        strcat(objectName, " (");
        strcat(objectName, textureMenuEntries[sceneObjs[i].texId - 1]);
        strcat(objectName, ")");
        if (currObject == i) { // Indicate currently selected object
            strcat(objectName, " *");
        }
        glutAddMenuEntry(objectName, objectId);
    }
}
```

The *selectObjectMenu* function is responsible for converting the 100+ *objectID* back to its respective array index value and assigning it to *currObject* and *toolObj*.

### J.3 Object Duplication

A new menu entry "Duplicate Object" was created which duplicates the currently selected object. This is done in the *duplicateObject* function which simply sets the next element in the *sceneObjs[]* array to be equal to the element in the *currObj* position in the array. Prior to this, a check is put a place to ensure that the numer of objects do not exceed the *maxObjects*. The function also sets *currObj* and *toolObj* to the index of the newly created duplicated object, and increments *nObjects*, the number of objects.

### J.4 Object Deletion

The *deleteObject* function is responsible for deleting objects from the scene. Objects that are deleted have their *meshID* values set to *NULL*, ensuring that they are not displayed in the object selection sub-menu. The value of *currObj* is set to -1 to indicate that no objects are selected.

```
static void deleteObject(int id) {
    sceneObjs[currObject].meshId = NULL;
    currObject = -1;
    delObjects++;
    makeMenu(); // PART J. Update object selection sub-menu
}
```

### J.5 Smarter Menu

Several menu options such as *Delete Object* and *Duplicate Object* will only be displayed under appropriate conditions. For example, if no object is selected, then deletion, duplication, position etc. cannot happen.

```
// Part J. Show sub-menu when there are > 1 objects (excluding the ground and
    lights)
if (nObjects - delObjects - 3 > 1) {
    glutAddSubMenu("Select Object", selectObjMenuId);
}
if (currObject != -1) { // Part J. Show only when an object is selected
    glutAddMenuEntry("Duplicate Object", 51);
    glutAddMenuEntry("Delete Object", 52);
    glutAddMenuEntry("Position/Scale", 41);
    glutAddMenuEntry("Rotation/Texture Scale", 55);
    glutAddSubMenu("Material", materialMenuId);
    glutAddSubMenu("Texture", texMenuId);
}
```

Note that the *delObjects* variable was created to keep count of the number of deleted objects as deleted objects still contribute to the *nObjects* variable.

# 3 Experience

I had initially found this project to be tough due to how overwhelming the entire *start-scene.cpp* code was. Despite this, the project acted as a stepping stone for me into the world of graphics, and has helped consolidate my understanding on lab and lecture contents. Implementing extensions to this part of the project has raised by my confidence and understanding in OpenGL, which will definitely aid me in the final exam.