Course Project: Space Gold Collection

CSCI 3260
PRINCIPLES OF COMPUTER GRAPHICS

Due Time: 11:59pm, December 5, 2021

Students:
KONG Kwai Man 1155125979,
NG Yu Chun Thomas 1155157839

1. Introduction

This project is a course assignment from the Chinese University of Hong Kong. The aim of the project is to create a virtual world and render it using OpenGL. The background is set to be in space and there exist a spacecraft to be controlled by player, numerous meteorites surrounding an undiscovered planet, a few unknown space vehicles, and Earth.

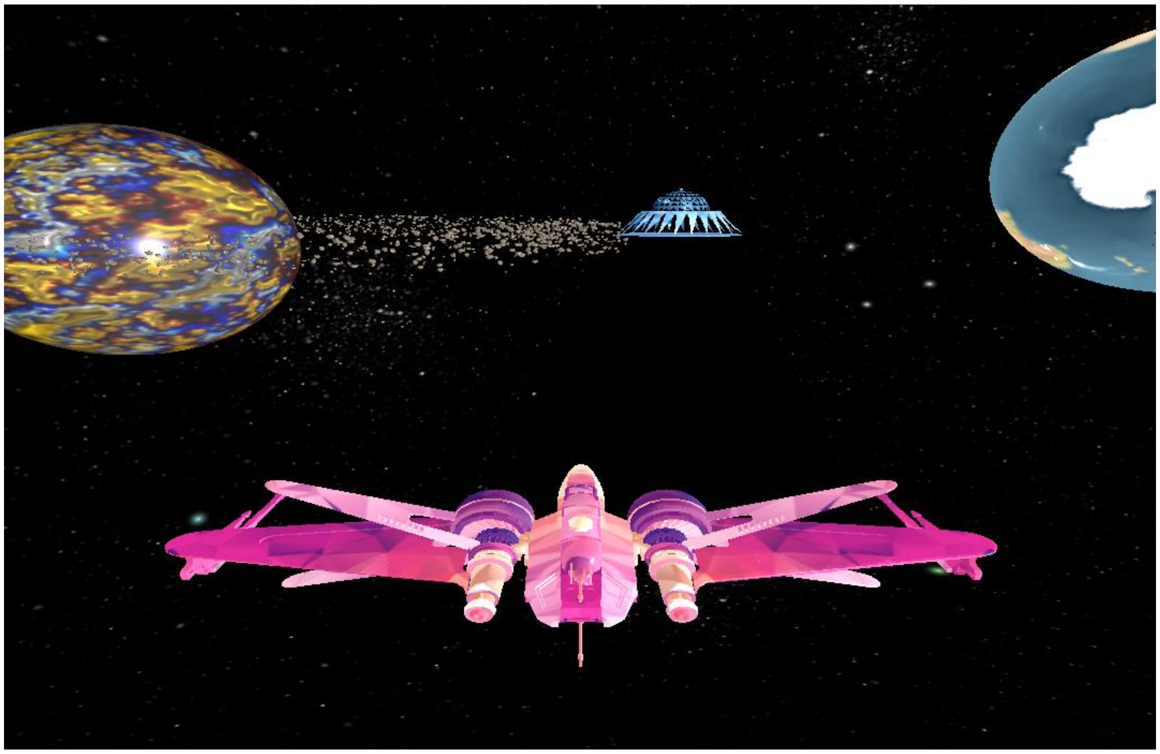


Figure 1.1 - The overall scene

2. Light Rendering

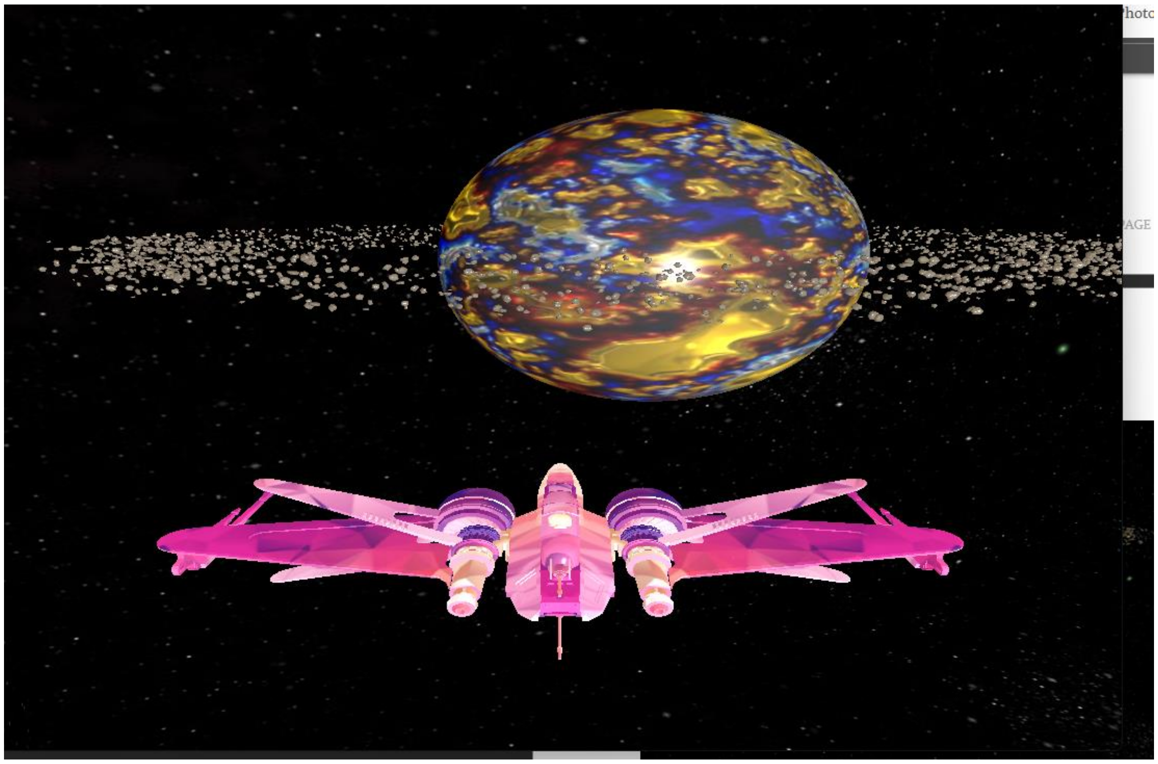


Figure 2.1 – Light rendering of the undiscovered planet and its meteorites

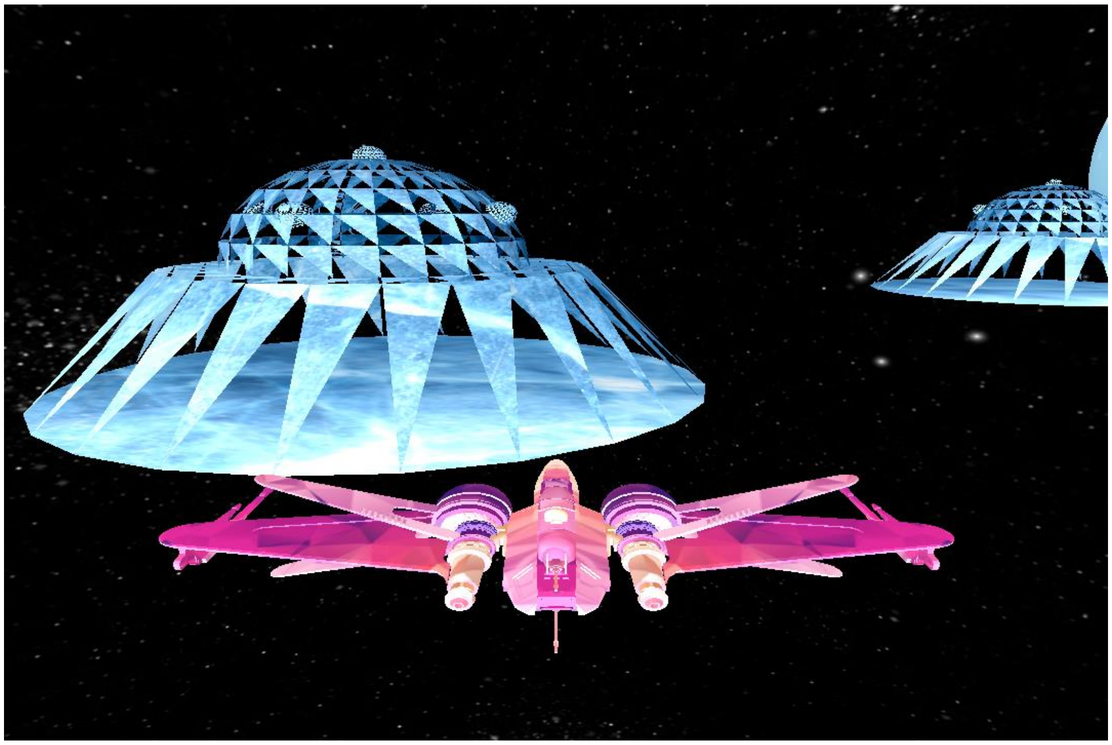


Figure 2.2 – Light rendering of the unknown space vehicle

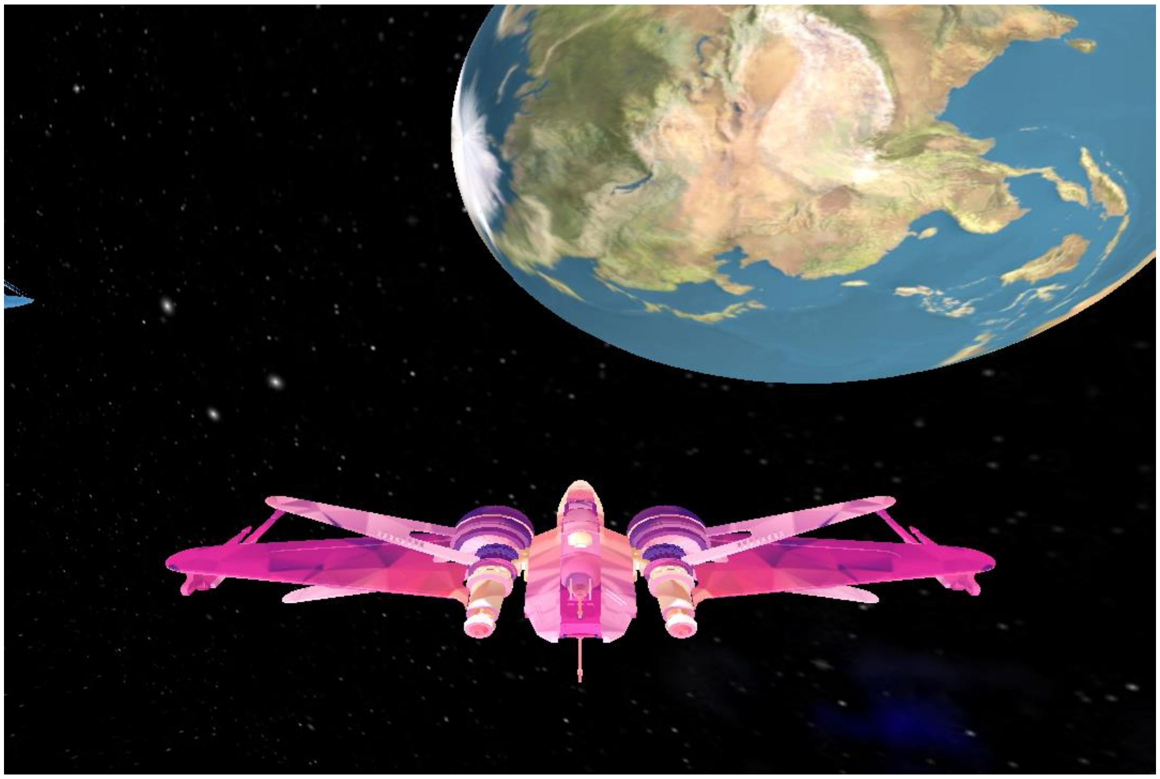
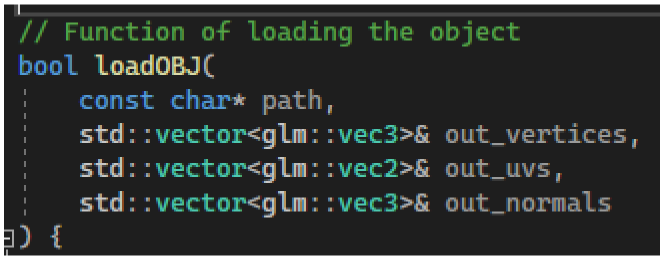


Figure 2.3 – Light rendering of Earth

3. Basic Requirement

a. Rendering objects

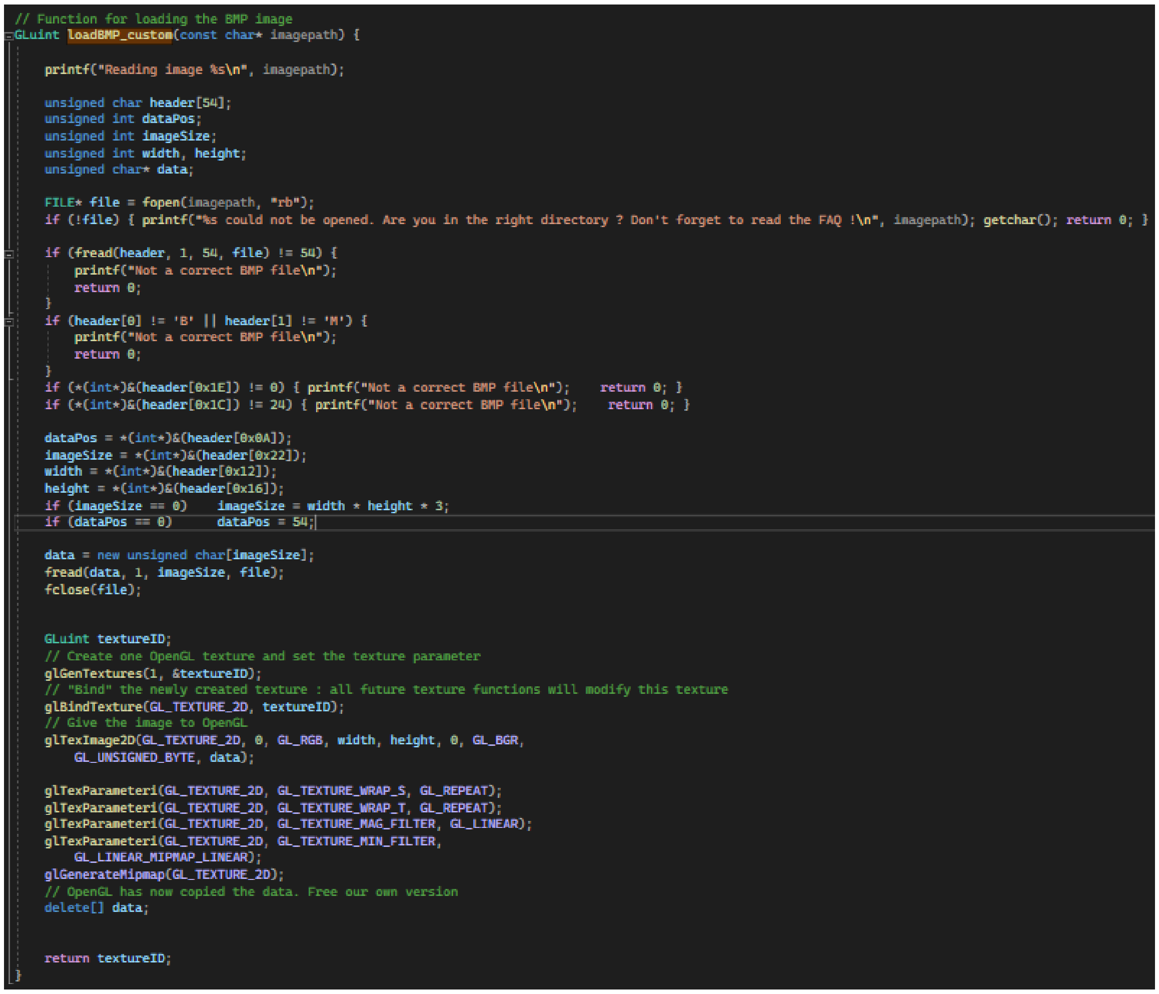
The 3D objects are using .obj file format. The obj information (e.g. vertices) will be imported by the `loadOBJ()` function.



```
// Function of loading the object
bool loadOBJ(
    const char* path,
    std::vector<glm::vec3>& out_vertices,
    std::vector<glm::vec2>& out_uvs,
    std::vector<glm::vec3>& out_normals
) {
```

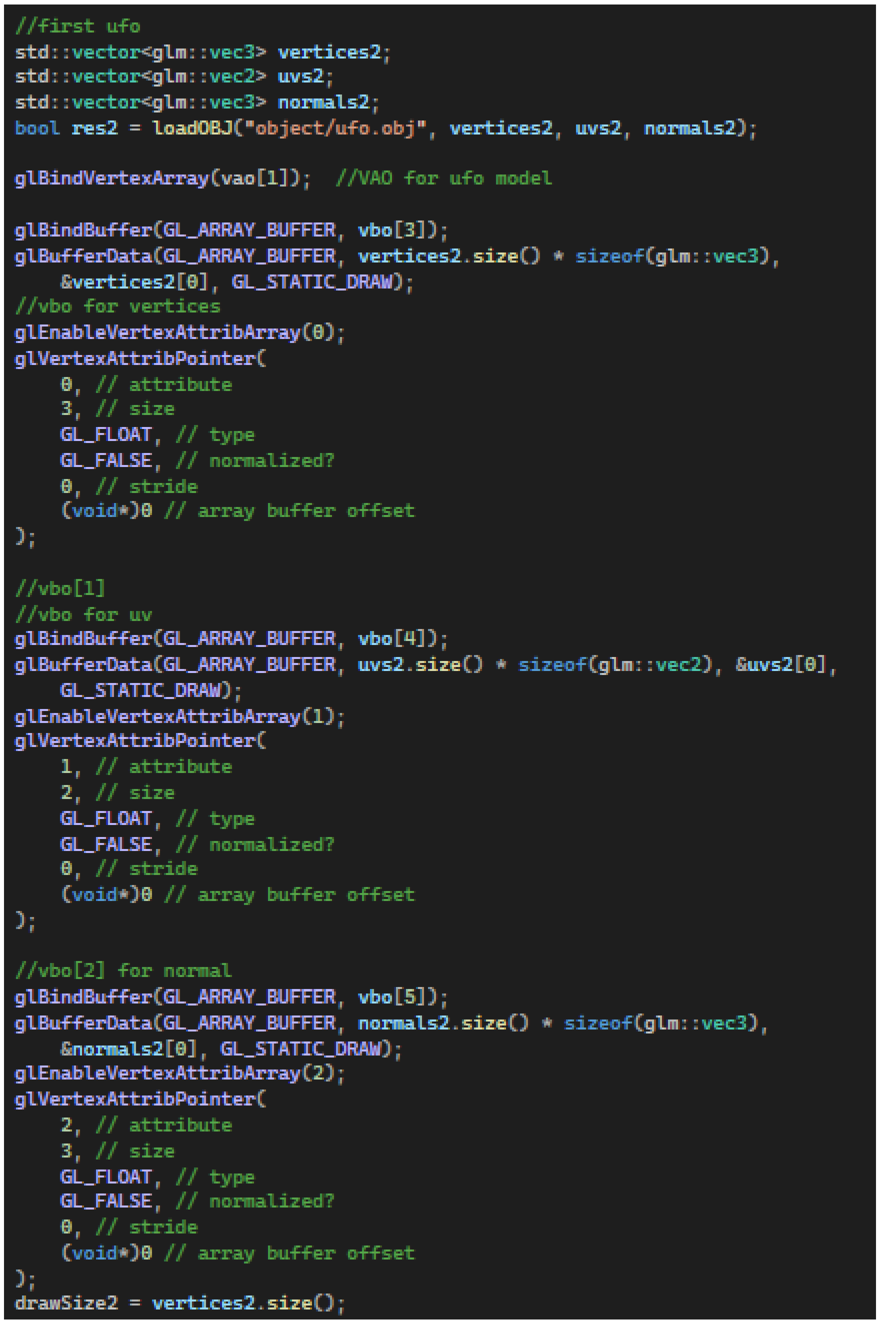
Figure 3.1 – `loadOBJ()` function parameters

For texture mapping, it is handled by `loadBMP_custom()` function.



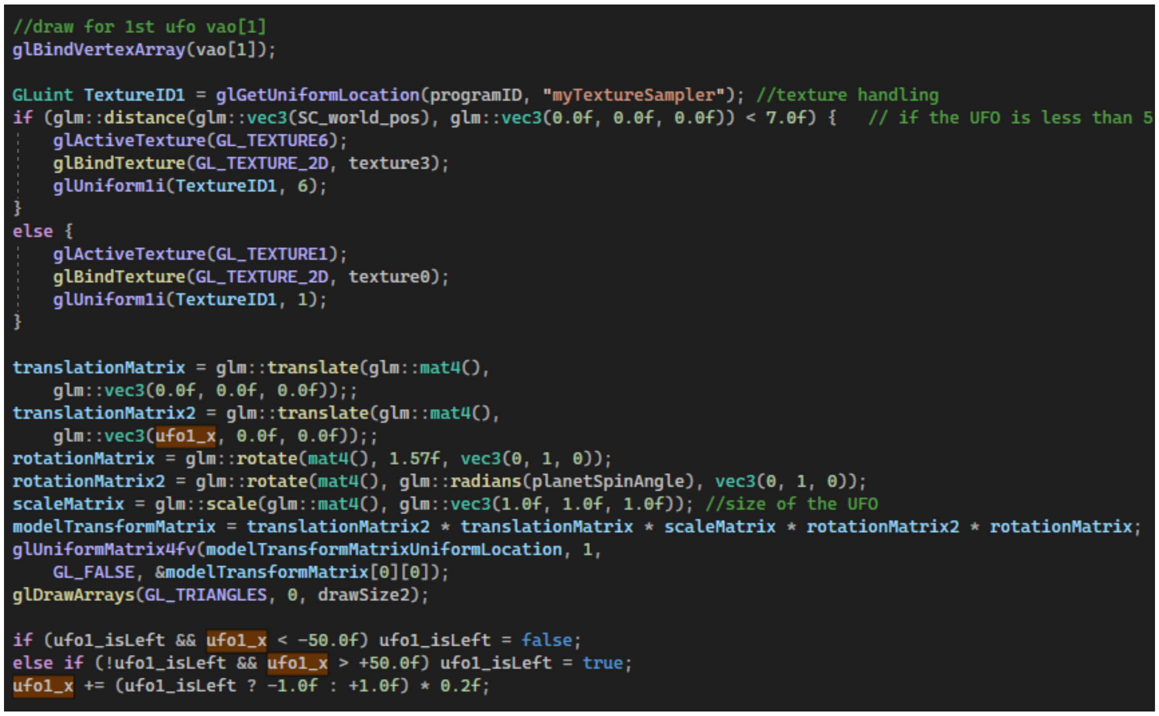
```
// Function for loading the BMP image
GLuint loadBMP_custom(const char* imagepath) {

    printf("Reading image %s\n", imagepath);

    unsigned char header[54];
    unsigned int dataPos;
    unsigned int imageSize;
    unsigned int width, height;
    unsigned char* data;

    FILE* file = fopen(imagepath, "rb");
    if (!file) { printf("%s could not be opened. Are you in the right directory ? Don't forget to read the FAQ !\n", imagepath); getchar(); return 0; }

    if (fread(header, 1, 54, file) != 54) {
        printf("Not a correct BMP file\n");
        return 0;
    }
    if (header[0] != 'B' || header[1] != 'M') {
        printf("Not a correct BMP file\n");
        return 0;
    }
    if (*(int*)&(header[0x1E]) != 0) { printf("Not a correct BMP file\n");    return 0; }
    if (*(int*)&(header[0x1C]) != 24) { printf("Not a correct BMP file\n");    return 0; }

    dataPos = *(int*)&(header[0x0A]);
    imageSize = *(int*)&(header[0x22]);
    width = *(int*)&(header[0x12]);
    height = *(int*)&(header[0x16]);
    if (imageSize == 0)    imageSize = width * height * 3;
    if (dataPos == 0)      dataPos = 54;

    data = new unsigned char[imageSize];
    fread(data, 1, imageSize, file);
    fclose(file);


    GLuint textureID;
    // Create one OpenGL texture and set the texture parameter
    glGenTextures(1, &textureID);
    // "Bind" the newly created texture : all future texture functions will modify this texture
    glBindTexture(GL_TEXTURE_2D, textureID);
    // Give the image to OpenGL
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_BGR,
        GL_UNSIGNED_BYTE, data);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_LINEAR_MIPMAP_LINEAR);
    glGenerateMipmap(GL_TEXTURE_2D);
    // OpenGL has now copied the data. Free our own version
    delete[] data;


    return textureID;
}
```

Figure 3.2 – Implementation of `loadBMP_custom()`



```
//first ufo
std::vector<glm::vec3> vertices2;
std::vector<glm::vec2> uvs2;
std::vector<glm::vec3> normals2;
bool res2 = loadOBJ("object/ufo.obj", vertices2, uvs2, normals2);

glBindVertexArray(vao[1]);  //VAO for ufo model

glBindBuffer(GL_ARRAY_BUFFER, vbo[3]);
glBufferData(GL_ARRAY_BUFFER, vertices2.size() * sizeof(glm::vec3),
    &vertices2[0], GL_STATIC_DRAW);
//vbo for vertices
glEnableVertexAttribArray(0);
glVertexAttribPointer(
    0, // attribute
    3, // size
    GL_FLOAT, // type
    GL_FALSE, // normalized?
    0, // stride
    (void*)0 // array buffer offset
);

//vbo[1]
//vbo for uv
glBindBuffer(GL_ARRAY_BUFFER, vbo[4]);
glBufferData(GL_ARRAY_BUFFER, uvs2.size() * sizeof(glm::vec2), &uvs2[0],
    GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(
    1, // attribute
    2, // size
    GL_FLOAT, // type
    GL_FALSE, // normalized?
    0, // stride
    (void*)0 // array buffer offset
);

//vbo[2] for normal
glBindBuffer(GL_ARRAY_BUFFER, vbo[5]);
glBufferData(GL_ARRAY_BUFFER, normals2.size() * sizeof(glm::vec3),
    &normals2[0], GL_STATIC_DRAW);
glEnableVertexAttribArray(2);
glVertexAttribPointer(
    2, // attribute
    3, // size
    GL_FLOAT, // type
    GL_FALSE, // normalized?
    0, // stride
    (void*)0 // array buffer offset
);
drawSize2 = vertices2.size();
```

Figure 3.3 – Example of setting up the VBOs of an object

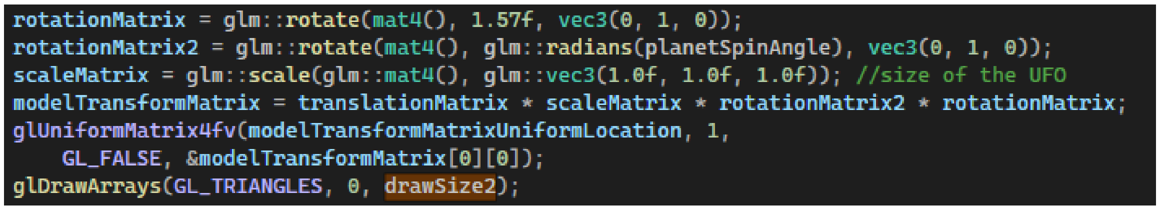


```
//draw for 1st ufo vao[1]
glBindVertexArray(vao[1]);

GLuint TextureID1 = glGetUniformLocation(programID, "myTextureSampler"); //texture handling
if (glm::distance(glm::vec3(SC_world_pos), glm::vec3(0.0f, 0.0f, 0.0f)) < 7.0f) {   // if the UFO is less than 5
    glActiveTexture(GL_TEXTURE6);
    glBindTexture(GL_TEXTURE_2D, texture3);
    glUniform1i(TextureID1, 6);
}
else {
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, texture0);
    glUniform1i(TextureID1, 1);
}

translationMatrix = glm::translate(glm::mat4(),
    glm::vec3(0.0f, 0.0f, 0.0f));;
translationMatrix2 = glm::translate(glm::mat4(),
    glm::vec3(ufo1_x, 0.0f, 0.0f));;
rotationMatrix = glm::rotate(mat4(), 1.57f, vec3(0, 1, 0));
rotationMatrix2 = glm::rotate(mat4(), glm::radians(planetSpinAngle), vec3(0, 1, 0));
scaleMatrix = glm::scale(glm::mat4(), glm::vec3(1.0f, 1.0f, 1.0f)); //size of the UFO
modelTransformMatrix = translationMatrix2 * translationMatrix * scaleMatrix * rotationMatrix2 * rotationMatrix;
glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
    GL_FALSE, &modelTransformMatrix[0][0]);
glDrawArrays(GL_TRIANGLES, 0, drawSize2);

if (ufo1_isLeft && ufo1_x < -50.0f) ufo1_isLeft = false;
else if (!ufo1_isLeft && ufo1_x > +50.0f) ufo1_isLeft = true;
ufo1_x += (ufo1_isLeft ? -1.0f : +1.0f) * 0.2f;
```

Figure 3.4 – Drawing the previous example object inside `paintGL()`

b. Self-rotation of the planet and space vehicles

Both planet and the space vehicles are handled in a similar way. The key is `rotationMatrix2` where the final `modelTransformMatrix` will include the multiplication of the `rotationMatrix2.` The variable `planetSpinAngle` in the `rotationMatrix2` will be **added by 0.6f** after each main loop in order to rotate (or spin) the objects.

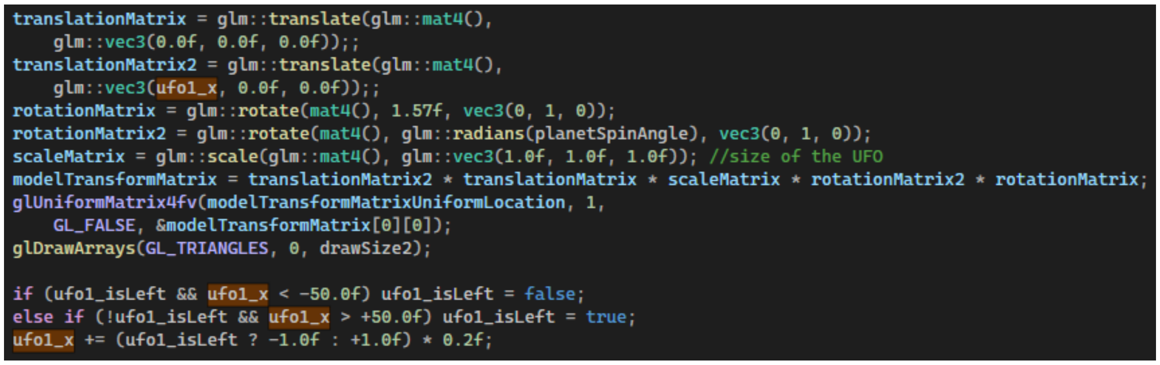


```
rotationMatrix = glm::rotate(mat4(), 1.57f, vec3(0, 1, 0));
rotationMatrix2 = glm::rotate(mat4(), glm::radians(planetSpinAngle), vec3(0, 1, 0));
scaleMatrix = glm::scale(glm::mat4(), glm::vec3(1.0f, 1.0f, 1.0f)); //size of the UFO
modelTransformMatrix = translationMatrix * scaleMatrix * rotationMatrix2 * rotationMatrix;
glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
    GL_FALSE, &modelTransformMatrix[0][0]);
glDrawArrays(GL_TRIANGLES, 0, drawSize2);
```

Figure 3.5 – Self-rotation implementation

c. Movements of space vehicles

The movements are controlled by *ufo1_x* and *ufo_isLeft*. If the space vehicles are reached the minimum or maximum, it changes its direction. The moving is done by *translationMatrix2*.

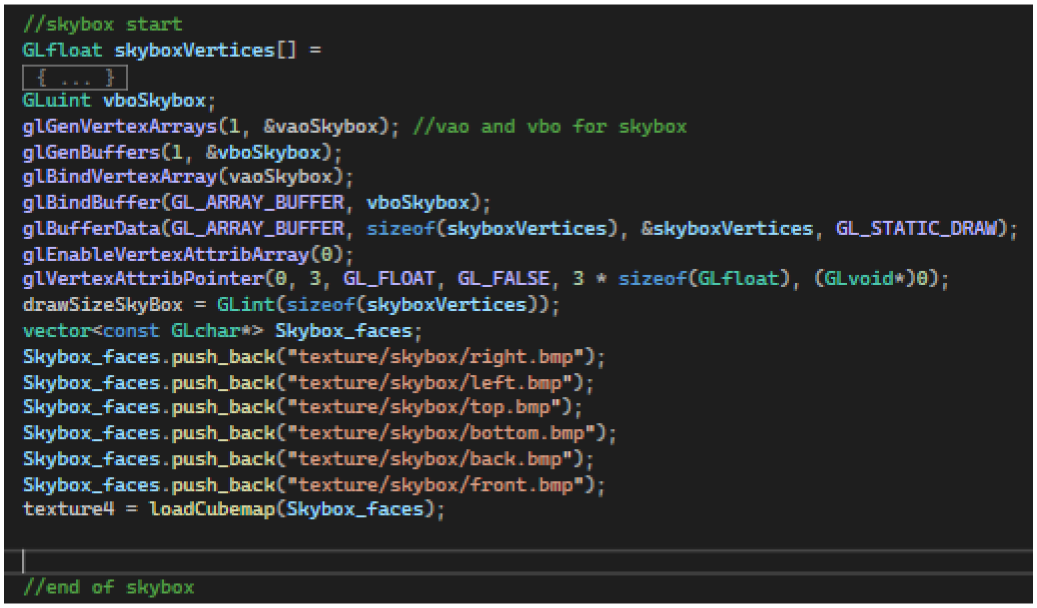


```
translationMatrix = glm::translate(glm::mat4(),
    glm::vec3(0.0f, 0.0f, 0.0f));;
translationMatrix2 = glm::translate(glm::mat4(),
    glm::vec3(ufo1_x, 0.0f, 0.0f));;
rotationMatrix = glm::rotate(mat4(), 1.57f, vec3(0, 1, 0));
rotationMatrix2 = glm::rotate(mat4(), glm::radians(planetSpinAngle), vec3(0, 1, 0));
scaleMatrix = glm::scale(glm::mat4(), glm::vec3(1.0f, 1.0f, 1.0f)); //size of the UFO
modelTransformMatrix = translationMatrix2 * translationMatrix * scaleMatrix * rotationMatrix2 * rotationMatrix;
glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1,
    GL_FALSE, &modelTransformMatrix[0][0]);
glDrawArrays(GL_TRIANGLES, 0, drawSize2);

if (ufo1_isLeft && ufo1_x < -50.0f) ufo1_isLeft = false;
else if (!ufo1_isLeft && ufo1_x > +50.0f) ufo1_isLeft = true;
ufo1_x += (ufo1_isLeft ? -1.0f : +1.0f) * 0.2f;
```

Figure 3.6 – Movements of space vehicles

d. Skybox

Skybox is a special object in a cube shape. The texture mapping, unlike other objects, is handled by *loadCubeMap()*.

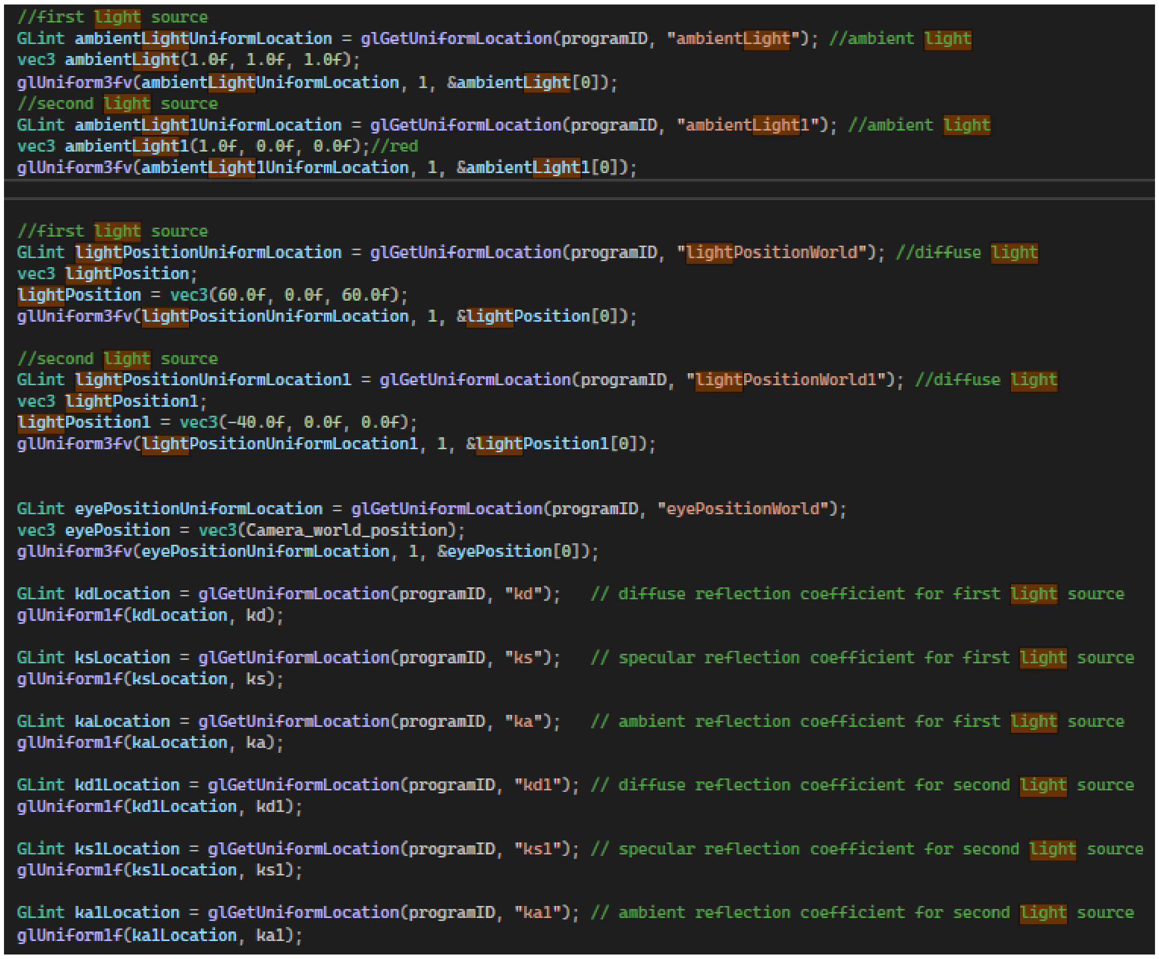


```
//skybox start
GLfloat skyboxVertices[] =
{ ... }
GLuint vboSkybox;
glGenVertexArrays(1, &vaoSkybox); //vao and vbo for skybox
glGenBuffers(1, &vboSkybox);
glBindVertexArray(vaoSkybox);
glBindBuffer(GL_ARRAY_BUFFER, vboSkybox);
glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
drawSizeSkyBox = GLint(sizeof(skyboxVertices));
vector<const GLchar*> Skybox_faces;
Skybox_faces.push_back("texture/skybox/right.bmp");
Skybox_faces.push_back("texture/skybox/left.bmp");
Skybox_faces.push_back("texture/skybox/top.bmp");
Skybox_faces.push_back("texture/skybox/bottom.bmp");
Skybox_faces.push_back("texture/skybox/back.bmp");
Skybox_faces.push_back("texture/skybox/front.bmp");
texture4 = loadCubemap(Skybox_faces);


//end of skybox
```

Figure 3.7 – Skybox

e. Light Source

Lighting are handled as shown in Figure 3.8. Note that player can change the lighting using keyboard control to alternate the value of `kd, ks, ka, kd1, ks1, ka1`.



```
//first light source
GLint ambientLightUniformLocation = glGetUniformLocation(programID, "ambientLight"); //ambient light
vec3 ambientLight(1.0f, 1.0f, 1.0f);
glUniform3fv(ambientLightUniformLocation, 1, &ambientLight[0]);
//second light source
GLint ambientLight1UniformLocation = glGetUniformLocation(programID, "ambientLight1"); //ambient light
vec3 ambientLight1(1.0f, 0.0f, 0.0f);//red
glUniform3fv(ambientLight1UniformLocation, 1, &ambientLight1[0]);

//first light source
GLint lightPositionUniformLocation = glGetUniformLocation(programID, "lightPositionWorld"); //diffuse light
vec3 lightPosition;
lightPosition = vec3(60.0f, 0.0f, 60.0f);
glUniform3fv(lightPositionUniformLocation, 1, &lightPosition[0]);

//second light source
GLint lightPositionUniformLocation1 = glGetUniformLocation(programID, "lightPositionWorld1"); //diffuse light
vec3 lightPosition1;
lightPosition1 = vec3(-40.0f, 0.0f, 0.0f);
glUniform3fv(lightPositionUniformLocation1, 1, &lightPosition1[0]);


GLint eyePositionUniformLocation = glGetUniformLocation(programID, "eyePositionWorld");
vec3 eyePosition = vec3(Camera_world_position);
glUniform3fv(eyePositionUniformLocation, 1, &eyePosition[0]);

GLint kdLocation = glGetUniformLocation(programID, "kd");   // diffuse reflection coefficient for first light source
glUniform1f(kdLocation, kd);

GLint ksLocation = glGetUniformLocation(programID, "ks");   // specular reflection coefficient for first light source
glUniform1f(ksLocation, ks);

GLint kaLocation = glGetUniformLocation(programID, "ka");   // ambient reflection coefficient for first light source
glUniform1f(kaLocation, ka);

GLint kd1Location = glGetUniformLocation(programID, "kd1"); // diffuse reflection coefficient for second light source
glUniform1f(kd1Location, kd1);

GLint ks1Location = glGetUniformLocation(programID, "ks1"); // specular reflection coefficient for second light source
glUniform1f(ks1Location, ks1);

GLint ka1Location = glGetUniformLocation(programID, "ka1"); // ambient reflection coefficient for second light source
glUniform1f(ka1Location, ka1);
```

Figure 3.8 – Light Sources

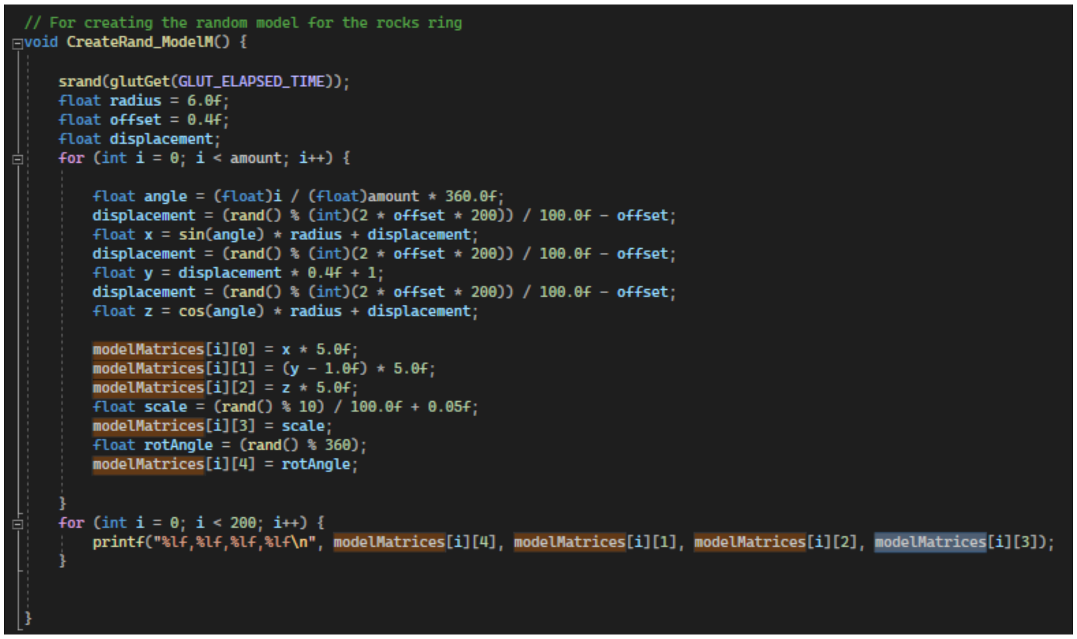
f. An asteroid ring

For generating the asteroid ring, a for loop is used. The new asteroid will be created and translated to a certain position from the planet which is calculated in the `CreateRand_ModelM()` function. Simple math is used such as

$$x = r \sin\theta$$
$$z = r \cos\theta$$

for converting coordinates system. The variable, `displacement`, is the crucial variable for making the asteroids do not stay at the same position.

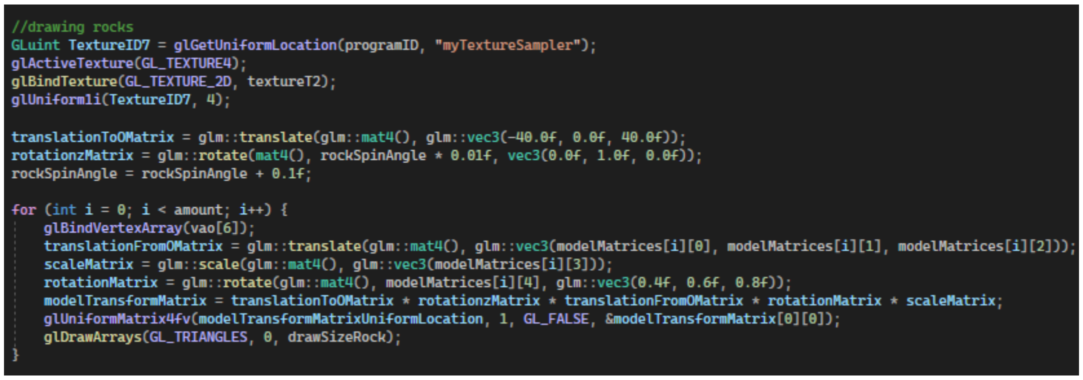
Model Matrices [#][0] refers to translation of x-axis.
Model Matrices [#][1] refers to translation of y-axis.
Model Matrices [#][2] refers to translation of z-axis.
Model Matrices [#][3] refers to scale of the asteroid.
Model Matrices [#][4] refers to rotation of the asteroid.



```
// For creating the random model for the rocks ring
void CreateRand_ModelM() {

    srand(glutGet(GLUT_ELAPSED_TIME));
    float radius = 6.0f;
    float offset = 0.4f;
    float displacement;
    for (int i = 0; i < amount; i++) {

        float angle = (float)i / (float)amount * 360.0f;
        displacement = (rand() % (int)(2 * offset * 200)) / 100.0f - offset;
        float x = sin(angle) * radius + displacement;
        displacement = (rand() % (int)(2 * offset * 200)) / 100.0f - offset;
        float y = displacement * 0.4f + 1;
        displacement = (rand() % (int)(2 * offset * 200)) / 100.0f - offset;
        float z = cos(angle) * radius + displacement;

        modelMatrices[i][0] = x * 5.0f;
        modelMatrices[i][1] = (y - 1.0f) * 5.0f;
        modelMatrices[i][2] = z * 5.0f;
        float scale = (rand() % 10) / 100.0f + 0.05f;
        modelMatrices[i][3] = scale;
        float rotAngle = (rand() % 360);
        modelMatrices[i][4] = rotAngle;

    }
    for (int i = 0; i < 200; i++) {
        printf("%lf,%lf,%lf,%lf\n", modelMatrices[i][4], modelMatrices[i][1], modelMatrices[i][2], modelMatrices[i][3]);
    }


}
```

Figure 3.9 – Calculation of `modelMatrices[][]` for the Asteroids.

With the calculation of `modelMatrices[][]` is done, in `paintGL()` we just need to render the asteroids just like other objects.

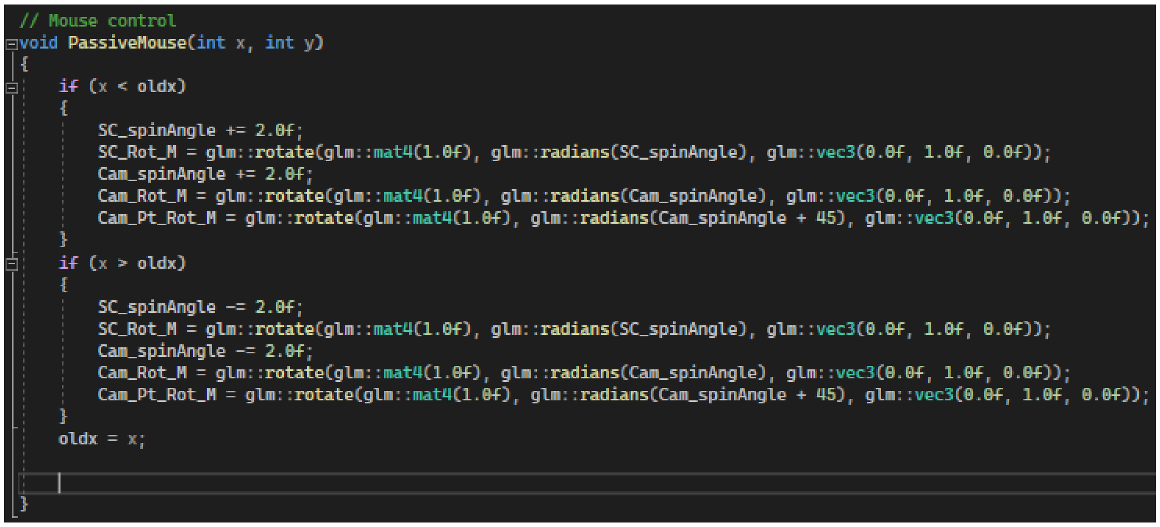


```cpp
//drawing rocks
GLuint TextureID7 = glGetUniformLocation(programID, "myTextureSampler");
glActiveTexture(GL_TEXTURE4);
glBindTexture(GL_TEXTURE_2D, textureT2);
glUniform1i(TextureID7, 4);

translationToOMatrix = glm::translate(glm::mat4(), glm::vec3(-40.0f, 0.0f, 40.0f));
rotationzMatrix = glm::rotate(mat4(), rockSpinAngle * 0.01f, vec3(0.0f, 1.0f, 0.0f));
rockSpinAngle = rockSpinAngle + 0.1f;

for (int i = 0; i < amount; i++) {
    glBindVertexArray(vao[6]);
    translationFromOMatrix = glm::translate(glm::mat4(), glm::vec3(modelMatrices[i][0], modelMatrices[i][1], modelMatrices[i][2]));
    scaleMatrix = glm::scale(glm::mat4(), glm::vec3(modelMatrices[i][3]));
    rotationMatrix = glm::rotate(glm::mat4(), modelMatrices[i][4], glm::vec3(0.4f, 0.6f, 0.8f));
    modelTransformMatrix = translationToOMatrix * rotationzMatrix * translationFromOMatrix * rotationMatrix * scaleMatrix;
    glUniformMatrix4fv(modelTransformMatrixUniformLocation, 1, GL_FALSE, &modelTransformMatrix[0][0]);
    glDrawArrays(GL_TRIANGLES, 0, drawSizeRock);
}
```

Figure 3.10 – Implementation of the asteroid ring

g. Viewpoint

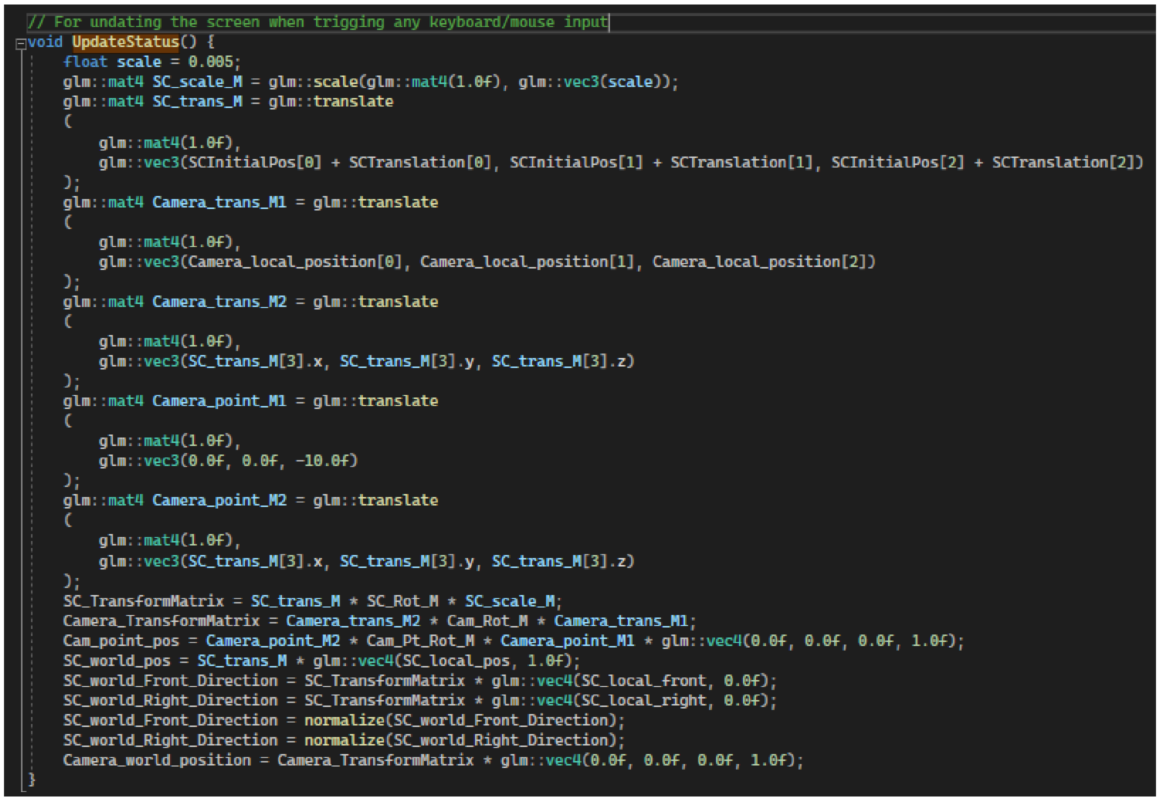
As the mouse can "spin" the view, the movement of mouse is handled by *PassiveMouse()* function separately.



```cpp
// Mouse control
void PassiveMouse(int x, int y)
{
    if (x < oldx)
    {
        SC_spinAngle += 2.0f;
        SC_Rot_M = glm::rotate(glm::mat4(1.0f), glm::radians(SC_spinAngle), glm::vec3(0.0f, 1.0f, 0.0f));
        Cam_spinAngle += 2.0f;
        Cam_Rot_M = glm::rotate(glm::mat4(1.0f), glm::radians(Cam_spinAngle), glm::vec3(0.0f, 1.0f, 0.0f));
        Cam_Pt_Rot_M = glm::rotate(glm::mat4(1.0f), glm::radians(Cam_spinAngle + 45), glm::vec3(0.0f, 1.0f, 0.0f));
    }
    if (x > oldx)
    {
        SC_spinAngle -= 2.0f;
        SC_Rot_M = glm::rotate(glm::mat4(1.0f), glm::radians(SC_spinAngle), glm::vec3(0.0f, 1.0f, 0.0f));
        Cam_spinAngle -= 2.0f;
        Cam_Rot_M = glm::rotate(glm::mat4(1.0f), glm::radians(Cam_spinAngle), glm::vec3(0.0f, 1.0f, 0.0f));
        Cam_Pt_Rot_M = glm::rotate(glm::mat4(1.0f), glm::radians(Cam_spinAngle + 45), glm::vec3(0.0f, 1.0f, 0.0f));
    }
    oldx = x;

}
```

Figure 3.11 – Mouse input handle

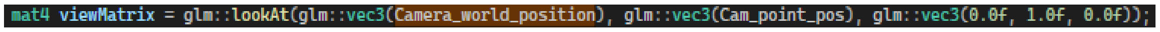
The camera transform matrix will be updated in the *paintGL()*, handled by a separate function, *UpdateStatus()*.



```
// For undating the screen when trigging any keyboard/mouse input
void UpdateStatus() {
    float scale = 0.005;
    glm::mat4 SC_scale_M = glm::scale(glm::mat4(1.0f), glm::vec3(scale));
    glm::mat4 SC_trans_M = glm::translate
    (
        glm::mat4(1.0f),
        glm::vec3(SCInitialPos[0] + SCTranslation[0], SCInitialPos[1] + SCTranslation[1], SCInitialPos[2] + SCTranslation[2])
    );
    glm::mat4 Camera_trans_M1 = glm::translate
    (
        glm::mat4(1.0f),
        glm::vec3(Camera_local_position[0], Camera_local_position[1], Camera_local_position[2])
    );
    glm::mat4 Camera_trans_M2 = glm::translate
    (
        glm::mat4(1.0f),
        glm::vec3(SC_trans_M[3].x, SC_trans_M[3].y, SC_trans_M[3].z)
    );
    glm::mat4 Camera_point_M1 = glm::translate
    (
        glm::mat4(1.0f),
        glm::vec3(0.0f, 0.0f, -10.0f)
    );
    glm::mat4 Camera_point_M2 = glm::translate
    (
        glm::mat4(1.0f),
        glm::vec3(SC_trans_M[3].x, SC_trans_M[3].y, SC_trans_M[3].z)
    );
    SC_TransformMatrix = SC_trans_M * SC_Rot_M * SC_scale_M;
    Camera_TransformMatrix = Camera_trans_M2 * Cam_Rot_M * Camera_trans_M1;
    Cam_point_pos = Camera_point_M2 * Cam_Pt_Rot_M * Camera_point_M1 * glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);
    SC_world_pos = SC_trans_M * glm::vec4(SC_local_pos, 1.0f);
    SC_world_Front_Direction = SC_TransformMatrix * glm::vec4(SC_local_front, 0.0f);
    SC_world_Right_Direction = SC_TransformMatrix * glm::vec4(SC_local_right, 0.0f);
    SC_world_Front_Direction = normalize(SC_world_Front_Direction);
    SC_world_Right_Direction = normalize(SC_world_Right_Direction);
    Camera_world_position = Camera_TransformMatrix * glm::vec4(0.0f, 0.0f, 0.0f, 1.0f);
}
```

Figure 3.12 – Update the Camera Matrix with given keyboard and mouse input

At last, the *viewMatrix* takes *Camera_world_position* and other value, which were calculated in the above function shown in Figure 3.12, for generating the *viewMatrix* with the aid of *glm::lookAt()* function



```
mat4 viewMatrix = glm::lookAt(glm::vec3(Camera_world_position), glm::vec3(Cam_point_pos), glm::vec3(0.0f, 1.0f, 0.0f));
```

Figure 3.13 - viewMatrix

h. Normal mapping

Normal mapping is implemented. A separate *normalMap* is introduced in the fragment shader.

```
vec3 normal = normalize(normalWorld);
        if(normalMap == 1.0) {
                normal = texture( myTextureSampler1, UV ).rgb;
                normal = normalize(normal * 2.0 - 1.0);

        }
```

4. Bonus Requirement

a. Second light source

A second light source is introduced as shown below in Figure 4.1.

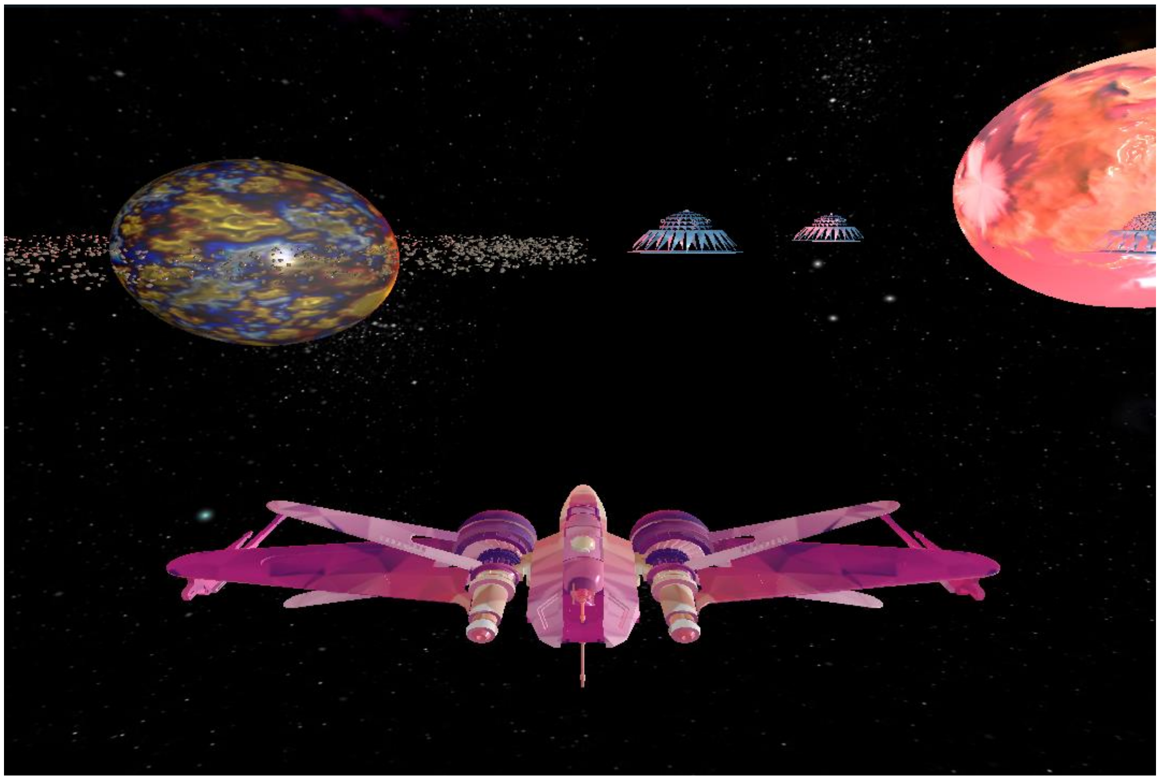
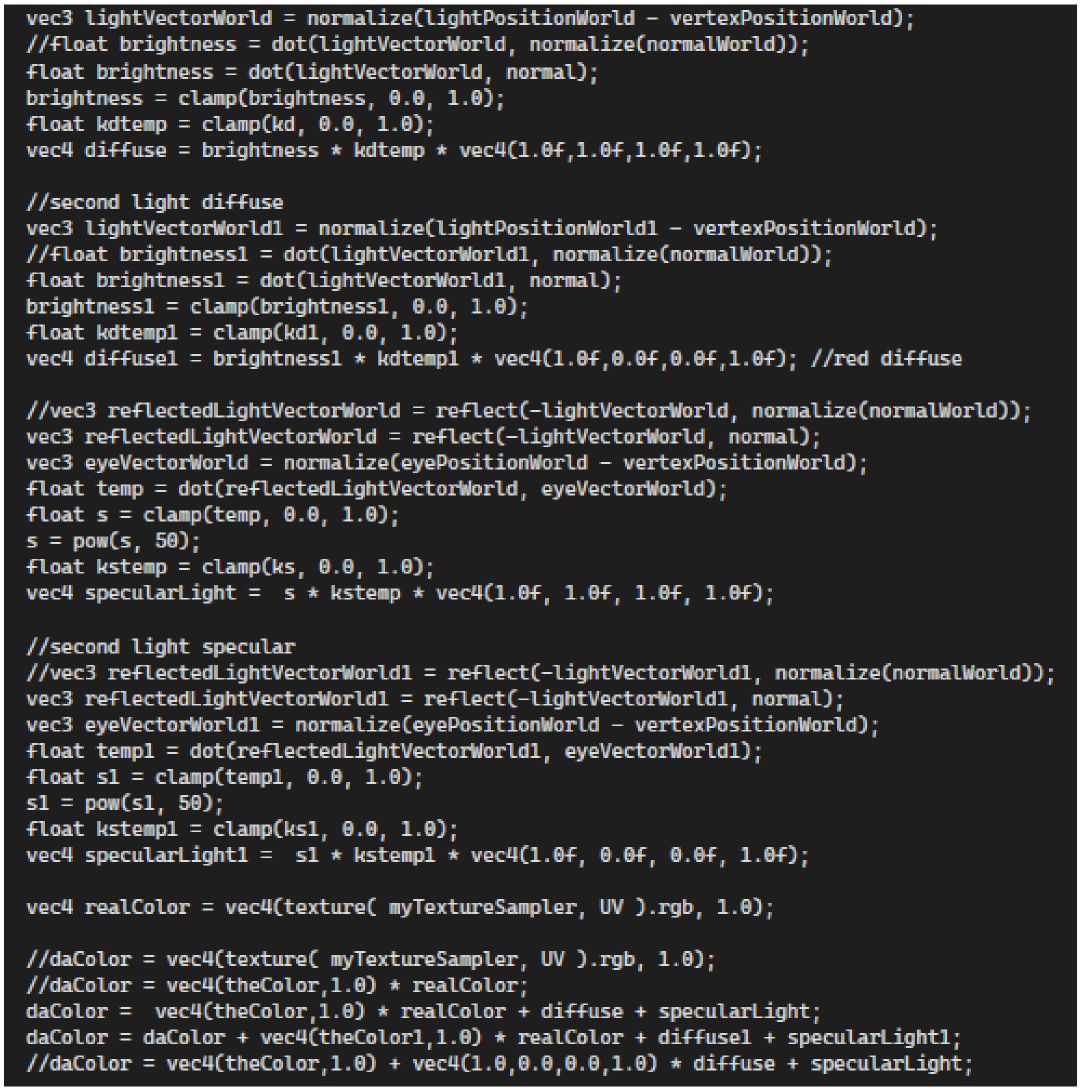


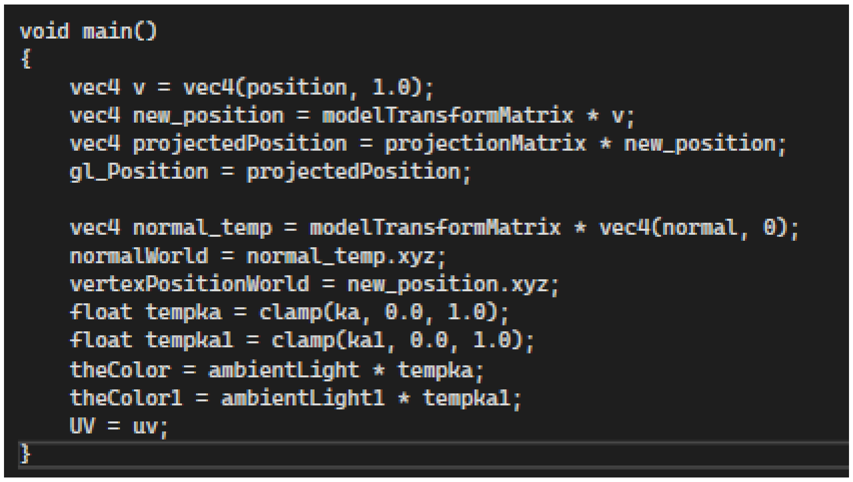
Figure 4.1 – Scene with second light source enabled

The calculation of the light sources, are mainly done by the fragment shader and vertex shader. Ambient, diffuse, specular are included according to the summation property of Phong Illumination Model.



```
vec3 lightVectorWorld = normalize(lightPositionWorld - vertexPositionWorld);
//float brightness = dot(lightVectorWorld, normalize(normalWorld));
float brightness = dot(lightVectorWorld, normal);
brightness = clamp(brightness, 0.0, 1.0);
float kdtemp = clamp(kd, 0.0, 1.0);
vec4 diffuse = brightness * kdtemp * vec4(1.0f,1.0f,1.0f,1.0f);

//second light diffuse
vec3 lightVectorWorld1 = normalize(lightPositionWorld1 - vertexPositionWorld);
//float brightness1 = dot(lightVectorWorld1, normalize(normalWorld));
float brightness1 = dot(lightVectorWorld1, normal);
brightness1 = clamp(brightness1, 0.0, 1.0);
float kdtemp1 = clamp(kd1, 0.0, 1.0);
vec4 diffuse1 = brightness1 * kdtemp1 * vec4(1.0f,0.0f,0.0f,1.0f); //red diffuse

//vec3 reflectedLightVectorWorld = reflect(-lightVectorWorld, normalize(normalWorld));
vec3 reflectedLightVectorWorld = reflect(-lightVectorWorld, normal);
vec3 eyeVectorWorld = normalize(eyePositionWorld - vertexPositionWorld);
float temp = dot(reflectedLightVectorWorld, eyeVectorWorld);
float s = clamp(temp, 0.0, 1.0);
s = pow(s, 50);
float kstemp = clamp(ks, 0.0, 1.0);
vec4 specularLight =  s * kstemp * vec4(1.0f, 1.0f, 1.0f, 1.0f);

//second light specular
//vec3 reflectedLightVectorWorld1 = reflect(-lightVectorWorld1, normalize(normalWorld));
vec3 reflectedLightVectorWorld1 = reflect(-lightVectorWorld1, normal);
vec3 eyeVectorWorld1 = normalize(eyePositionWorld - vertexPositionWorld);
float temp1 = dot(reflectedLightVectorWorld1, eyeVectorWorld1);
float s1 = clamp(temp1, 0.0, 1.0);
s1 = pow(s1, 50);
float kstemp1 = clamp(ks1, 0.0, 1.0);
vec4 specularLight1 =  s1 * kstemp1 * vec4(1.0f, 0.0f, 0.0f, 1.0f);

vec4 realColor = vec4(texture( myTextureSampler, UV ).rgb, 1.0);

//daColor = vec4(texture( myTextureSampler, UV ).rgb, 1.0);
//daColor = vec4(theColor,1.0) * realColor;
daColor =  vec4(theColor,1.0) * realColor + diffuse + specularLight;
daColor = daColor + vec4(theColor1,1.0) * realColor + diffuse1 + specularLight1;
//daColor = vec4(theColor,1.0) + vec4(1.0,0.0,0.0,1.0) * diffuse + specularLight;
```

Figure 4.2 – Fragment shader



```
void main()
{
    vec4 v = vec4(position, 1.0);
    vec4 new_position = modelTransformMatrix * v;
    vec4 projectedPosition = projectionMatrix * new_position;
    gl_Position = projectedPosition;

    vec4 normal_temp = modelTransformMatrix * vec4(normal, 0);
    normalWorld = normal_temp.xyz;
    vertexPositionWorld = new_position.xyz;
    float tempka = clamp(ka, 0.0, 1.0);
    float tempka1 = clamp(ka1, 0.0, 1.0);
    theColor = ambientLight * tempka;
    theColor1 = ambientLight1 * tempka1;
    UV = uv;
}
```

Figure 4.3 – Vertex shader

b. Additional object

After discussion with my groupmate, we decided to add another planet with an asteroid ring, instead of adding it into Earth. Nothing is special as it is just like other rendered objects.

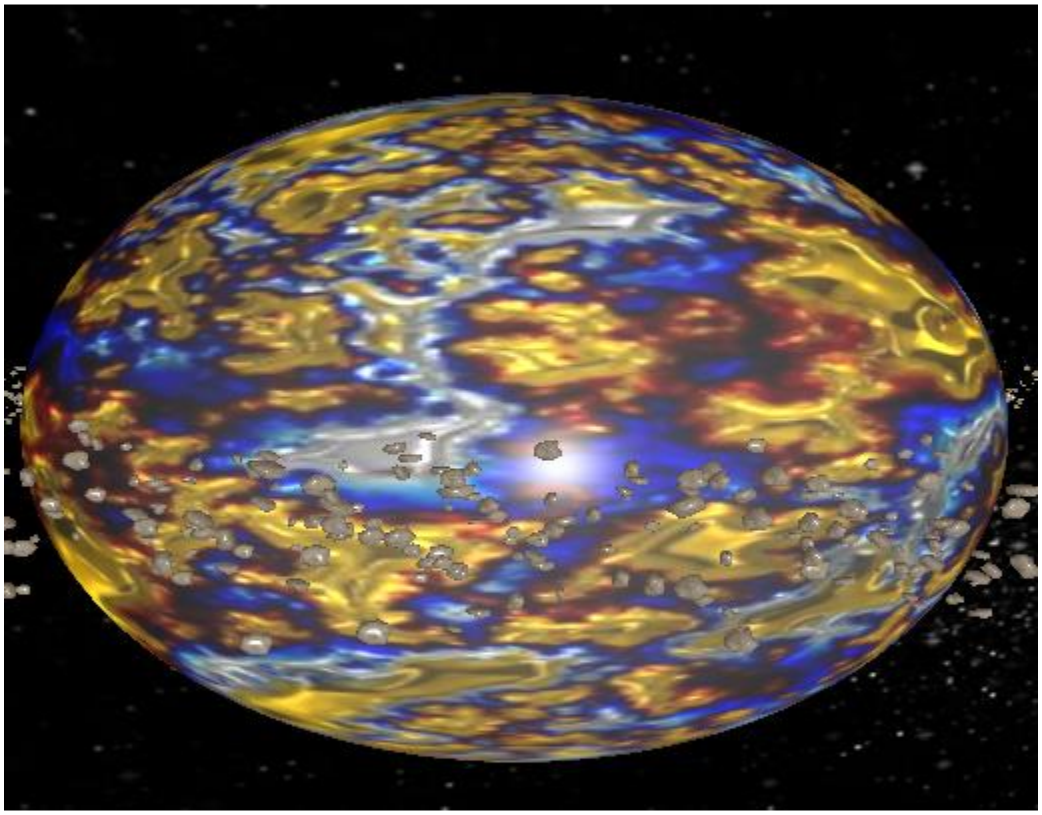


Figure 4.4 – A new planet

c. Scene interaction

If the distance between a space vehicle and the player is less than 7.0f, the texture will be changed to a grey texture to indicate there is a collision.

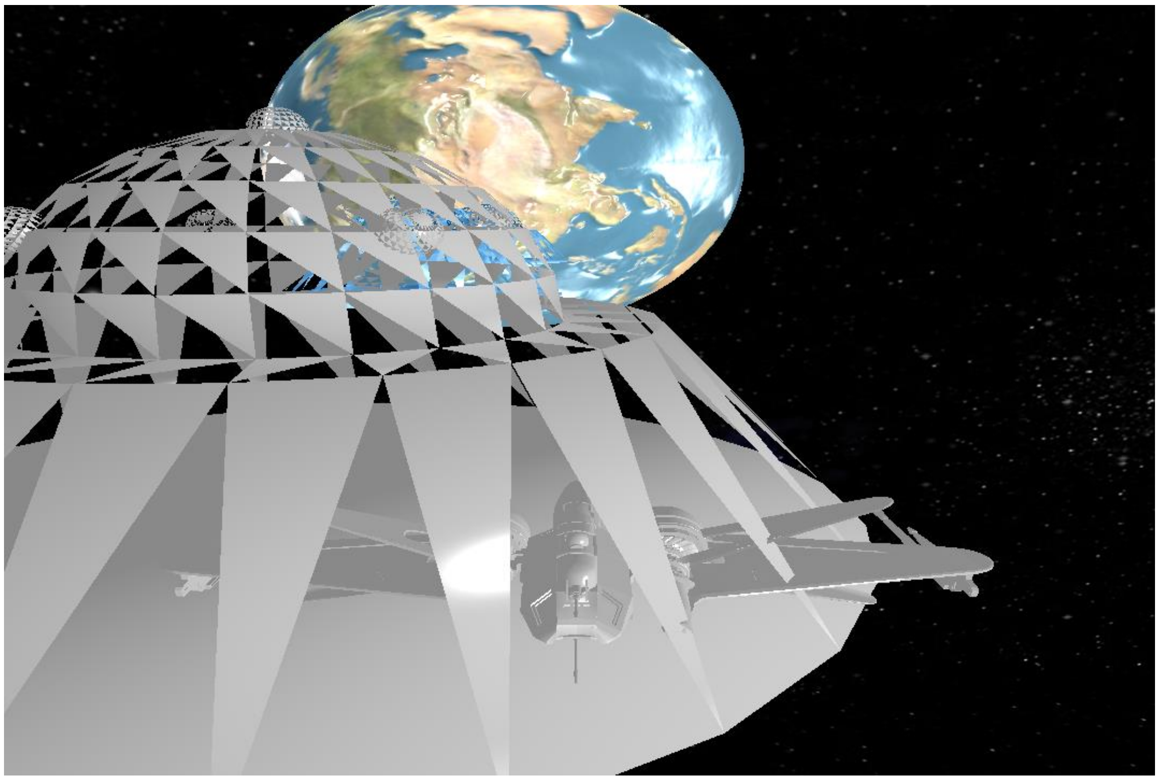
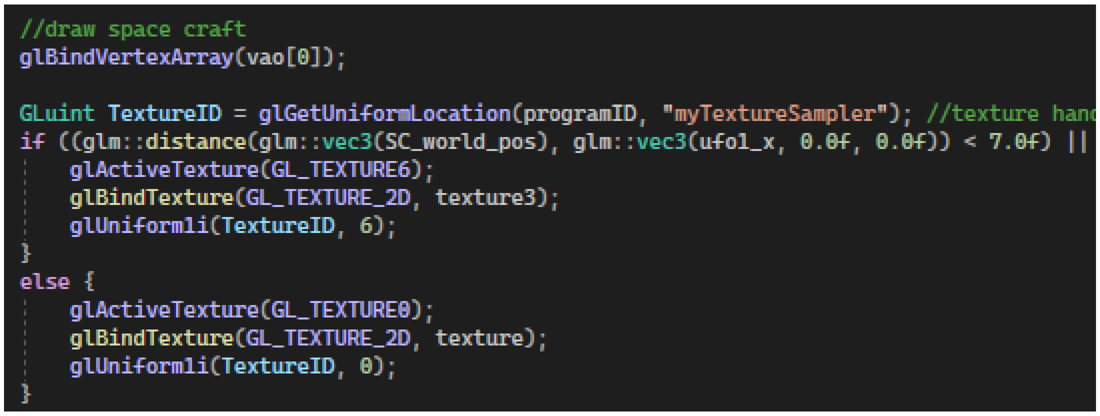
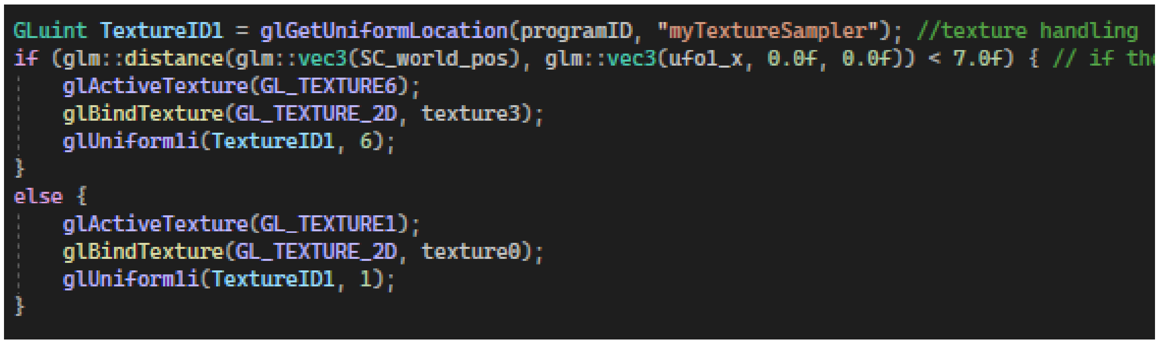


Figure 4.5 – Collision between player and space vehicles



```
//draw space craft
glBindVertexArray(vao[0]);

GLuint TextureID = glGetUniformLocation(programID, "myTextureSampler"); //texture hand
if ((glm::distance(glm::vec3(SC_world_pos), glm::vec3(ufo1_x, 0.0f, 0.0f)) < 7.0f) ||
    glActiveTexture(GL_TEXTURE6);
    glBindTexture(GL_TEXTURE_2D, texture3);
    glUniform1i(TextureID, 6);
}
else {
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);
    glUniform1i(TextureID, 0);
}
```

Figure 4.6 – Player collision detection code



```
GLuint TextureID1 = glGetUniformLocation(programID, "myTextureSampler"); //texture handling
if (glm::distance(glm::vec3(SC_world_pos), glm::vec3(ufo1_x, 0.0f, 0.0f)) < 7.0f) { // if th
    glActiveTexture(GL_TEXTURE6);
    glBindTexture(GL_TEXTURE_2D, texture3);
    glUniform1i(TextureID1, 6);
}
else {
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, texture0);
    glUniform1i(TextureID1, 1);
}
```

Figure 4.7 – Space vehicle collision detection code

5. Control

Key	Action
Up arrow	Move the spacecraft to the front
Down arrow	Move the spacecraft to the back
Left arrow	Move the spacecraft to the left
Right arrow	Move the spacecraft to the right
q or w	increase/decrease diffuse reflection coefficient for 1st light source
a or s	increase/decrease ambient reflection coefficient for 1st light source
z or x	increase/decrease specular reflection coefficient for 1st light source
o or p	increase/decrease diffuse reflection coefficient for 2nd light source
k or l	increase/decrease ambient reflection coefficient for 2nd light source
n or m	increase/decrease specular reflection coefficient for 2nd light source