Macau University of Science and Technology
CS104
Fundamentals of Computer Graphics

Animation of Solar System Report

Mao Zunjie Wang Haocheng
18098531-I011-0019 18098537-I011-0112
D1

11-12-2020

Contents

1 Introduction

This project mainly illustrates the implementaion of solar system by using OpenGL, The report is divided into two main parts, Part one contains the design of the program, the output, and an explanation of some of the code; Part two is my comments and summary. The purpose of the program is to simulate the orbital diagram of a complete solar system with eight planets, including the rotation of the planets. The focus of the experiment is on how to integrate a complete solar system program from matrix stack, sphere drawing, texture mapping, ADS lighting, etc. using what we learned in class.

The whole program has been successfully complied on **Visual Studio 2017** (Version 15) Win 32 with **OpenGL** (version 4.3).

2 Design of Solar System

The design of the program is divided into four parts. The first part is the sphere creation (Report 3.2), where we start from designing a sphere, referring to the textbook 6.1, to complete the basic idea of drawing a planetary sphere. The second part is the generation of the solar system. After the sphere creation process, we draw the eight planets of the solar system, including their positions, rotations, and scaling, using a matrix stack, referring mainly to program 4.4. the procedure explanation is in report 3.4. The third part is texture mapping (Report 3.3), where we add textures for each planet, referring to chapter5, to make them more realistic. In addition, we also added a skybox(Report 3.5) to make the program more complete and beautiful. The last part is lighting (Report 3.6). After the first three parts, the solar system is now taking shape, and we added a light source to the sun in the last part to give it a light/dark contrast.

3 Procedure Explanation

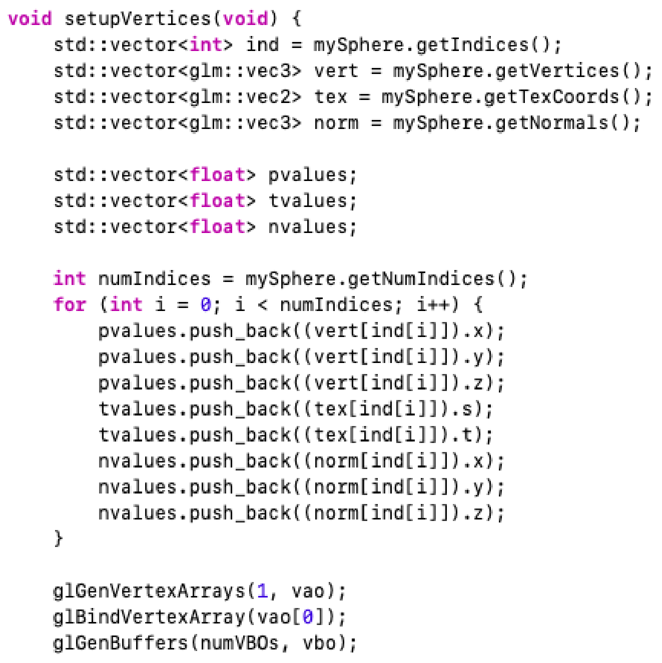
3.1 Variables allocation

At the beginning of the program, we want to build out a space to allocate the variables used in the **display()** function so that they do not have to be allocated during rendering. parts of code on **main.cpp**.

3.2 Sphere Creation

Here we come to the function **setupVertices()** called by **init()**, our goal is to build the sphere, and the header file used is **sphere.h**. When using the sphere class, we need three values for each vertex position and normal vector, but only two values for each texture coordinate.

Because we use c++ to complete this program, we can not use <iostream.h>. And we want to use the function about cin and out, we need to announce the function 'using namespace std' at first. And then in the void of setupVertices function. We are not announce, so if we want to use it we need to add **std::** in the front of each variable. Such as **std::vector<int> ind = mySphere.getIndices()**;. In this program, we use c++, so we can not use 'sphere.h'. Then, we need to add some function about 'sphere.h'. In this function, we set four matrices which are named **getIndices()**, **getVertices()**, **getTexCoords()** and **getNormals()**. The position and the normal need 3 value and the texcoord need 2 value. Because we need float to record numbers, we also define 3 vector to record vertices position, texture value and normal vector. After that, we need enter each number to array. Here we use stack. The function of **push_back** means we add the number to the first position of the array. When the second number **push_back** into the array, the first number change its positon after the first one in the array. We follow this circle, until it record all numbers, we get the complete arrays named vert[ind[i]], tex[ind[i]] and norm[ind[i]].

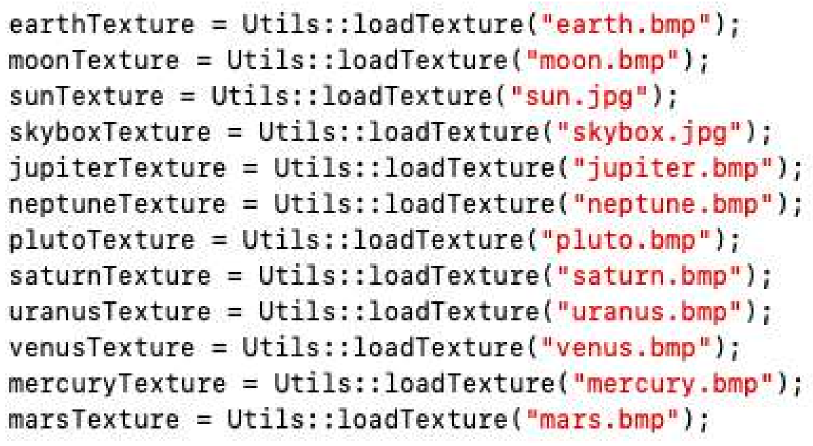


```
void setupVertices(void) {
    std::vector<int> ind = mySphere.getIndices();
    std::vector<glm::vec3> vert = mySphere.getVertices();
    std::vector<glm::vec2> tex = mySphere.getTexCoords();
    std::vector<glm::vec3> norm = mySphere.getNormals();

    std::vector<float> pvalues;
    std::vector<float> tvalues;
    std::vector<float> nvalues;

    int numIndices = mySphere.getNumIndices();
    for (int i = 0; i < numIndices; i++) {
        pvalues.push_back((vert[ind[i]]).x);
        pvalues.push_back((vert[ind[i]]).y);
        pvalues.push_back((vert[ind[i]]).z);
        tvalues.push_back((tex[ind[i]]).s);
        tvalues.push_back((tex[ind[i]]).t);
        nvalues.push_back((norm[ind[i]]).x);
        nvalues.push_back((norm[ind[i]]).y);
        nvalues.push_back((norm[ind[i]]).z);
    }

    glGenVertexArrays(1, vao);
    glBindVertexArray(vao[0]);
    glGenBuffers(numVBOs, vbo);
```

The **glGenVertexArrays** function is used to create the VAO and generate the object ID. the first parameter specifies the number of VAOs to be created. By calling the **glBindVertexArray** function to bind the VAO, the next configuration of the vertex property pointer and the corresponding VBO will be stored in this VAO. If you need to unbind the current VAO, just set the parameter to 0. Similar to creating a VAO, the VBO is created by calling the glGenBuffers function, the VBO buffer type is **GL_ARRAY_BUFFER**, and the VBO is bound to **GL_ARRAY_BUFFER** by calling the **glBindBuffer** function. The **glBufferData** function can copy the vertex data to Buffer memory for GPU use. Here we put the vertex, texture coordinates and normal vector into the buffer in three operations.

3.3 Texture Mapping

At the beginning of the entire main.cpp, we declare a variable of type **GLunit**, which creates the texture object. In the **init()** function, we mainly call the **setupVertice** function from the previous step, and texture the planets. the key adding parts in **init()** is here.



```
earthTexture = Utils::loadTexture("earth.bmp");
moonTexture = Utils::loadTexture("moon.bmp");
sunTexture = Utils::loadTexture("sun.jpg");
skyboxTexture = Utils::loadTexture("skybox.jpg");
jupiterTexture = Utils::loadTexture("jupiter.bmp");
neptuneTexture = Utils::loadTexture("neptune.bmp");
plutoTexture = Utils::loadTexture("pluto.bmp");
saturnTexture = Utils::loadTexture("saturn.bmp");
uranusTexture = Utils::loadTexture("uranus.bmp");
venusTexture = Utils::loadTexture("venus.bmp");
mercuryTexture = Utils::loadTexture("mercury.bmp");
marsTexture = Utils::loadTexture("mars.bmp");
```

Additionally, to make it easier for us to texture the planets, we add the **GLunit LoadTexture** function to Utils.cpp, so that we can create OpenGL textures in the main program by calling the **loadTexture()** function.In order to maximize performance, we wanted to implement texture processing in the hardware. We declare a sampler variable in the shader: *layout (binding=0) uniform sampler2D samp;* we declare a variable named samp, and *layout (binding=0)* specifies that this sampler is associated with texture unit 0. In the subsequent **display()** function, each time we draw a planet, we activate the texture unit with the **glActiveTexture** function and **glBindTexture** function and bind it to a specific texture object. The **glActiveTexture** parameter **GL_TEXTURE0** is the one that makes the 0th texture unit active. Also, we sample the texture object with the interpolated texture coordinates received from the shader, using the texture function: color=*texture (samp, tc).*

3.4 Planet Generation

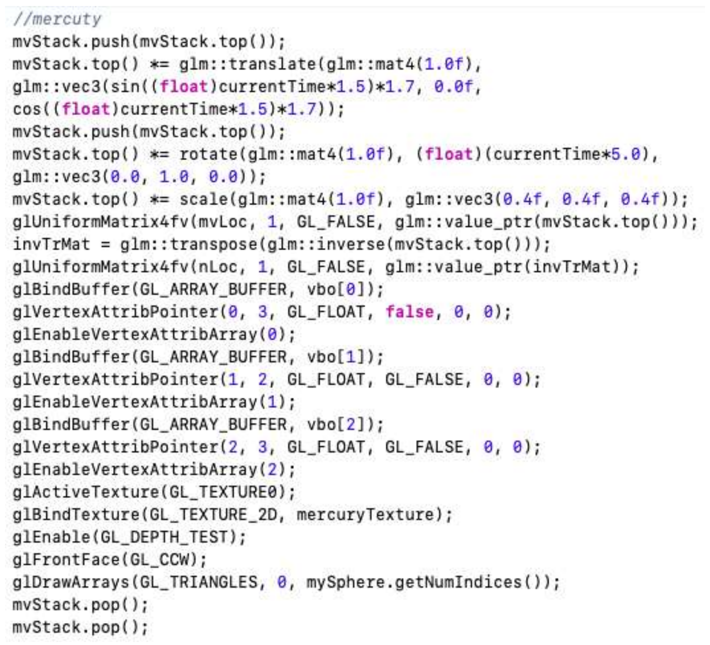
The **glClear()** function is used to clear the specified buffer with the current buffer, i.e. with the value specified by the **glClearColor** function, and also uses the **glDrawBuffer** function to clear multiple color buffers at once. **gl_COLOR_BUFFER_BIT** is the color buffer here. Next, the display function enables the shader by calling the **glUseProgram** function to install the GLSL code on the GPU, which will allow later OpenGL calls to determine the shader's vertex properties and uniform variable locations.

Next, we will prepare for the further demonstration of the planets. First we need to get the uniform variables for the MV matrix and the projection matrix. After that, we build the perspective matrix, the view matrix, the model matrix, and the view-model matrix. We copy the perspective matrix and MV matrix to the corresponding uniform variables, associate the VBO to the corresponding vertex properties in the vertex shader, adjust the OpenGL settings, and draw the model.

After that, we will use the matrix stack to create the planets. First we push the view matrix into the stack, declaring the stack "mvStack". When we create a new object relative to its parent, we call "**mvStack.push(mvStack.top())**, create a new entry at the top of the stack, make a copy of the matrix currently at the top of the stack, and combine it with other transformations, including rotation and scale, etc. **mvStack.top()* =translate** describes the planet's position, **mvStack,top()*=rotate** describes the planet's rotation, and **mvStack.top()*=scale** describes the planet's scale. As the Mecury Code example below, The number 1.5 behind Code **'currentTime'** is mainly controls the revolution, 1.7 controls distance. Then we pass the model-view matrix of the planet to the shader using **glUniformMatrix4fv** function, **glUniformMatrix4fv**(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value) as a function, location specifies the location of the uniform variable to be changed, count specifies the number of matrices to be changed, and transpose specifies whether to transpose the matrix as the value of the uniform variable. The value must be **GL_FALSE**. value specifies a pointer to the count element to update the specified uniform variable. Its function is to copy the perspective matrix and MV matrix to the corresponding unified variables, then associate the VBO to the corresponding vertex properties in the vertex shader, adjust the OPengl settings, and when we finish drawing a new object, we call **mvStack.pop()** to remove the model-view matrix from the top of the matrix stack.

For further optimization, we have written the **window_size_callback** function block to configure the user to automatically call back the specified function when resizing the window. We provide new widths and heights for the callbacks and set the screen area associated with the frame buffer.

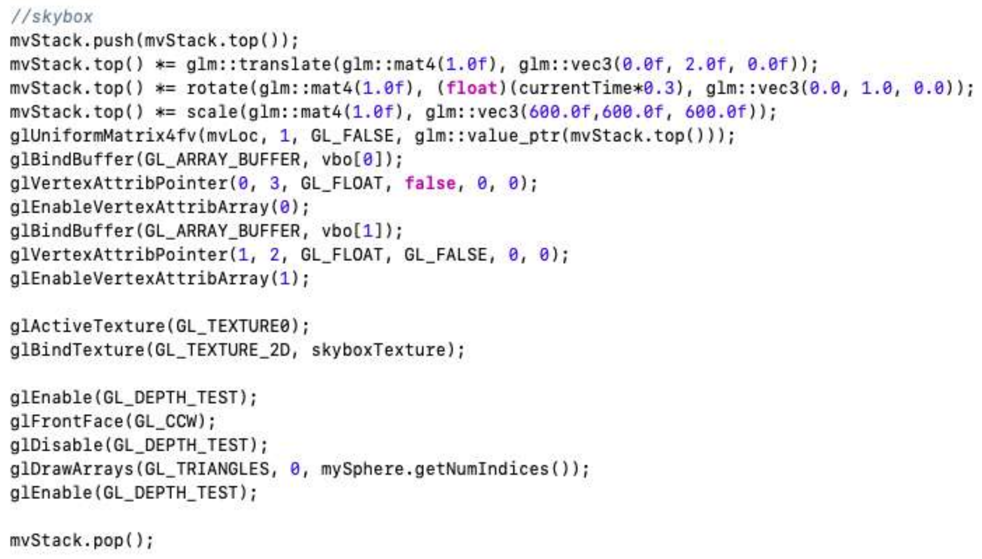
here we takes an example as Mecury gneration process in display() function, which contains Matrix Stack process, key Step of Texture mapping based on Program 4.4



```
//mercuty
mvStack.push(mvStack.top());
mvStack.top() *= glm::translate(glm::mat4(1.0f),
glm::vec3(sin((float)currentTime*1.5)*1.7, 0.0f,
cos((float)currentTime*1.5)*1.7));
mvStack.push(mvStack.top());
mvStack.top() *= rotate(glm::mat4(1.0f), (float)(currentTime*5.0),
glm::vec3(0.0, 1.0, 0.0));
mvStack.top() *= scale(glm::mat4(1.0f), glm::vec3(0.4f, 0.4f, 0.4f));
glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvStack.top()));
invTrMat = glm::transpose(glm::inverse(mvStack.top()));
glUniformMatrix4fv(nLoc, 1, GL_FALSE, glm::value_ptr(invTrMat));
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(2);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, mercuryTexture);
glEnable(GL_DEPTH_TEST);
glFrontFace(GL_CCW);
glDrawArrays(GL_TRIANGLES, 0, mySphere.getNumIndices());
mvStack.pop();
mvStack.pop();
```

3.5 Skybox

the operation of skybox is the same as the planet generation. For Simple, We also declare the stack "mvStack". when we create skybox relative to its solar system, we call "mvStack.push(mvStack.top()), create a new entry at the top of the stack, make a copy of the matrix currently at the top of the stack, and combine it with other transforms, including scaling, etc. We can also use "mvStack.top(mvStack.top())" to create a new entry at the top of the stack.



```
//skybox
mvStack.push(mvStack.top());
mvStack.top() *= glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 2.0f, 0.0f));
mvStack.top() *= rotate(glm::mat4(1.0f), (float)(currentTime*0.3), glm::vec3(0.0, 1.0, 0.0));
mvStack.top() *= scale(glm::mat4(1.0f), glm::vec3(600.0f,600.0f, 600.0f));
glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvStack.top()));
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, skyboxTexture);

glEnable(GL_DEPTH_TEST);
glFrontFace(GL_CCW);
glDisable(GL_DEPTH_TEST);
glDrawArrays(GL_TRIANGLES, 0, mySphere.getNumIndices());
glEnable(GL_DEPTH_TEST);

mvStack.pop();
```

then we also use **glUniformMatrix4fv** function is to copy the perspective matrix and MV matrix to the corresponding unified variables, then associate the VBO to the corresponding vertex properties in the vertex shader, adjust the OPengl settings, and perform texture mapping operations. In this program, we also adjusted the parameters to make the skybox have the rotation function, and then successfully draw the skybox.

3.6 Lighting

For the part of adding light sources, we mainly base **Phong** coloring. The operation of the vertex shader and fragment shader will be explained in Report 3.7 and 3.8. At the beginning of the main.cpp program, we define the lighting and material properties. The spherical vertices and associated normal vectors are read into the buffer immediately afterwards. In the **display()** function, we pass the lighting and material information to the vertex shader. To pass the information, it calls the **installLights()** function. First, the light positions are converted to view space coordinates and stored in a floating-point array. Second, it gets the light position and material attributes from the shader and assigns values to the light and material uniform variables. The unified variables are defined for better performance.

3.7 Vertex Shader

In the **vertShader.glsl**, First, we need to declare a vertex attribute variable, which should be declared in each buffer. The keyword "in" means that the vertex attribute will receive a value from the buffer, e.g, "vec3" means that the shader will grab three floating-point values per call. The command uses a "layout" qualifier to associate a vertex attribute with a particular buffer. *out vec2 tc* means that the declaration defines a vertex attribute named "tc" that outputs a value of type "vec2". Next, we use the uniform keyword to declare uniform variables for storing the model-view and projection matrices. In addition to these, we declare a sampler variable in the shader: *layout (binding=0) uniform sampler 2D t*; we declare a variable named "t", and "layout (binding=0)" specifies that this sampler is associated with texture unit 0. We apply the matrix transformation to the incoming position vertices and assign the result to the **gl_Position** function. The transformed vertices are automatically output to the raster shader and finally the resulting pixel positions are passed to the fragment shader.

For the first time in vertex shaders, we use the structured body syntax for declaring variables. The variable light is declared as type "PositionalLights" and the variable material is declared as type "Material" with **Struct** statement. **varyingNormal** refers to the vertex normal vector in the visual space, **varyingLightDir** points to the vector of light sources, and **varyingVertPos** refers to the vertex position in the visual space. Most of the lighting calculations take place in the vertex shader, where the color output and position are sent to the fragment shader after we complete the vector operations. *Uniform mat4 norm_matrix* is used to transform the normal vector. And in *void main* function, the vertex position, light direction and normal vector are output to the rasterizer for interpolation.

VertShader Code

```
layout (location = 0) in vec3 vertPos;
layout (location = 1) in vec2 texCoord;
layout (location = 2) in vec3 vertNormal;
layout (location = 3) in vec3 vertTangent;
out vec3 varyingLightDir;
out vec3 varyingVertPos;
out vec3 varyingNormal;
out vec3 varyingTangent;
out vec2 tc;
layout (binding=0) uniform sampler2D t;
layout (binding=1) uniform sampler2D n;
layout (binding=2) uniform sampler2D h;
```

```
struct PositionalLight
{
vec4 ambient;
vec4 diffuse;
vec4 specular;
vec3 position;
};
struct Material
{ vec4 ambient;
vec4 diffuse;
vec4 specular;
float shininess;
};
uniform vec4 globalAmbient;
uniform PositionalLight light;
uniform Material material;
uniform mat4 mv_matrix;
uniform mat4 view_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;

void main(void)
{ varyingNormal = (norm_matrix * vec4(vertNormal,1.0)).xyz;
varyingTangent = (norm_matrix * vec4(vertTangent,1.0)).xyz;

vec4 P1 = vec4(vertPos, 1.0);
vec4 P2 = vec4((vertNormal*((texture(h,texCoord).r)/15.0)), 1.0);
vec4 P = vec4((P1.xyz + P2.xyz),1.0);

varyingVertPos = (mv_matrix * P).xyz;
vec4 lp = vec4(light.position,1);
varyingLightDir = (view_matrix*lp).xyz;// light.position - varyingVertPos;

tc = texCoord;
gl_Position = proj_matrix * mv_matrix * P;

}
```

3.8 Fragment Shader

In the **fragShader.glsl**, to actually perform texture processing, we sample the texture object using the interpolated texture coordinates received from the vertex shader, using the **texture** function, in void main(), we put texture(t,tc) to tcc which is declared as a vector. it is also included in if condition i.e. *if(sun==1)*, to gnerate the fragcolor. details are in code part below. In addition to these, a uniform variable for storing the model-view and projection matrices is declared in the fragment shader using the uniform keyword.

The same with Vertex Shader, we also use struct body to declare variables. we move to the void main() function, We use **normalize()** to normalize the light vector, the normal vector and the visual vector. The normalize function is mainly used to convert a vector to a unit length. get the angle between the light and surface normal,It is copied into variable H here for convenience later. then get angle between the normal and the halfway vector, then Combining light and texture, compute ADS contributions (per pixel).

Additionally, all operations on normals are encapsulated in the **CalcBumpedNormal()** function. First we normalize the normal and tangent vectors, then we subtract the tangent vector from its component in the normal direction to get its component perpendicular to the normal. This is our new tangent vector. The new tangent vector and the normal vector are multiplied by the bitangent vector. After that we sample the normal information of this tangent (in tangent/texture space) from the normal texture. We need to transform the normal information from the tangent space to the world coordinate system. The normal information extracted from the texture in tangent space is multiplied by the TBN matrix and the result is normalized and returned to the caller, which gives the final normal information of the tessellation.

fragShader Code

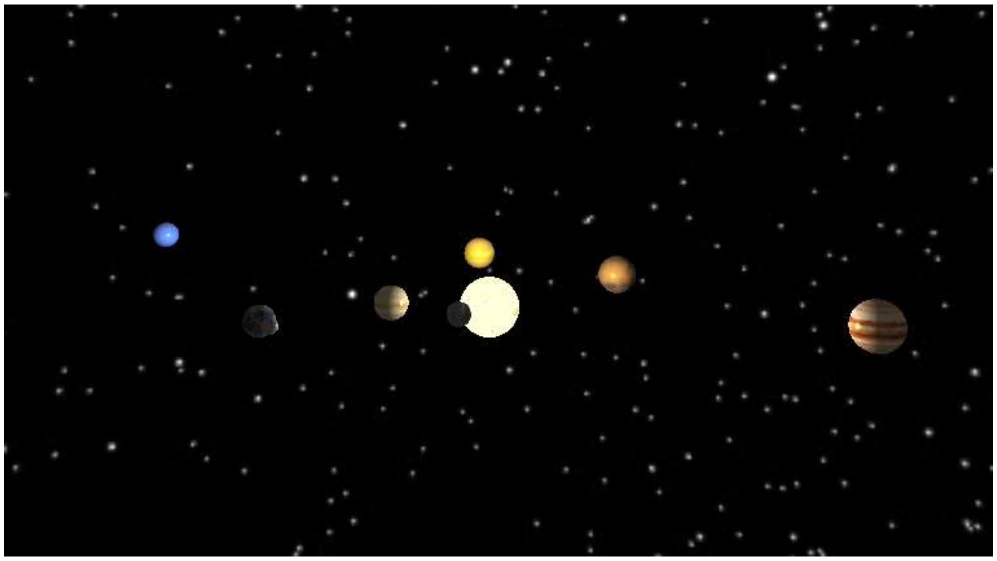
```
in vec3 varyingLightDir;
in vec3 varyingVertPos;
in vec3 varyingNormal;
in vec3 varyingTangent;
in vec2 tc;
out vec4 fragColor;

layout (binding=0) uniform sampler2D t;
layout (binding=1) uniform sampler2D n;
layout (binding=2) uniform sampler2D h;
struct PositionalLight
{ vec4 ambient;
```

```
vec4 diffuse;
vec4 specular;
vec3 position;
};

struct Material
{ vec4 ambient;
vec4 diffuse;
vec4 specular;
float shininess;
};

uniform vec4 globalAmbient;
uniform PositionalLight light;
uniform Material material;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;
uniform int sun;
vec3 CalcBumpedNormal()
{
vec3 Normal = normalize(varyingNormal);
return Normal;
vec3 Tangent = normalize(varyingTangent);
Tangent = normalize(Tangent - dot(Tangent, Normal) * Normal);
vec3 Bitangent = cross(Tangent, Normal);
vec3 BumpMapNormal = texture(n,tc).xyz;
BumpMapNormal = BumpMapNormal * 2.0 - 1.0;
mat3 TBN = mat3(Tangent, Bitangent, Normal);
vec3 NewNormal = TBN * BumpMapNormal;
NewNormal = normalize(NewNormal);
return NewNormal;
}
void main(void)
{
if(sun==1)
{
vec4 tcc = texture(t,tc);
fragColor = tcc*1.3;
return;
}
```

```
vec3 L = normalize(varyingLightDir-varyingVertPos);
vec3 V = normalize(-varyingVertPos);
vec3 N = CalcBumpedNormal();
float cosTheta = dot(L,N);
vec3 R = normalize(reflect(-L, N));
float cosPhi = dot(V,R);
vec4 tcc = texture(t,tc);
fragColor =0.25*tcc +
2*tcc*(fragColor = globalAmbient * material.ambient
+ light.ambient * material.ambient
+ light.diffuse * material.diffuse * max(cosTheta,0.0)
+ light.specular * material.specular
pow(max(cosPhi,0.0), material.shininess));
}
```

4 Output of Program

The Program is successfully programmed. The Screenshot has been pasted as follow.



5 My Opinions and Summary

From the theoretical study of OpenGL in the classroom to working on our own program for a planet in the solar system, we really got a lot out of it. For the program, we started by designing the program, which was mainly a review of the textbook, including but not limited to the programs listed in the textbook; after reviewing the examples in the professor's class, we started to design the program. We started by integrating a complete program from four aspects: sphere generation, texture mapping, matrix stack planet generation, and lighting.Although there was a slight bug after completing the program when we ran it, we eventually found and solved it.

When it comes to our understanding of this project, and what we've learned since we completed it, we've come to understand OpenGL even better. The program mainly contains a main.cpp, a glsl-type vertex shader file, a glsl-type fragment shader file, and Utils.h, Utils.cpp, Sphere.h, Sphere.cpp. Throughout the semester, we were able to go from generating spheres, to texture mapping, to adding light sources, to drawing a complete solar system. Unlike other subjects, Fundamentals of Computer Graphics gave us a great interest in Computer Graphics field, and I think we were able to design something more complex for this project as well. We can take what we have in our minds and make it happen through code. Not only for what we have now, but I think this course in graphics will also have an impact on our careers and provide a solid foundation.