Project in CS104 – Animation of the Solar System

HE PEILIN 1809853U-I011-0078

CHEN YUXUAN 1809853J-I011-0011

Contents

1 Design Outline

The core of project is to be familiar with transformations, texture mapping, lighting and window event handling, so that general animation of solar system can be realized.

This virtual solar system consists of the Sun, eight planets (Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune) and the Moon which can refer Figure.1.0.1. The Sun is stationary during the whole animation. The eight planets circulate around the sun and self-rotate at constant angular velocities. Moreover, the all of asters in whole solar system are surrounded by a star-sphere, which is a collection of stars located on a surface of a sphere.

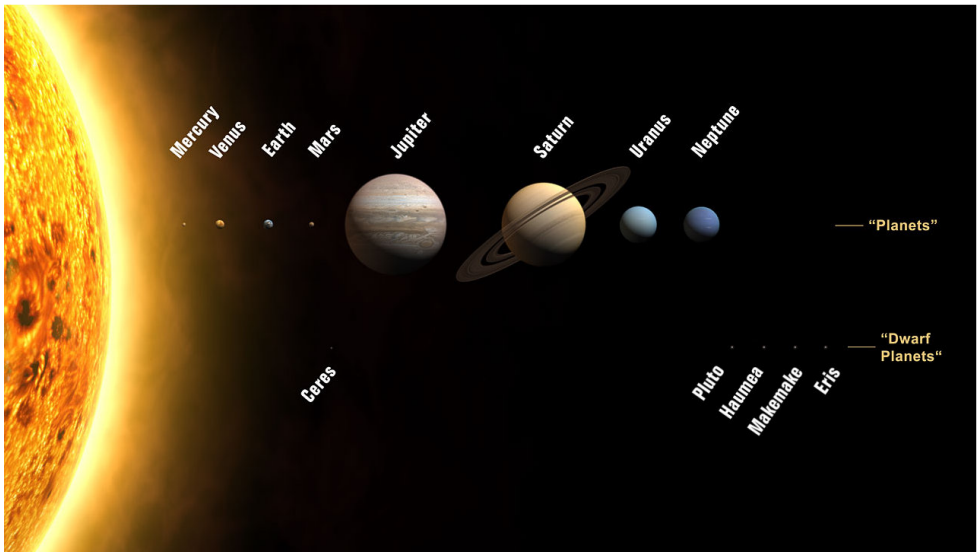

Figure. 1.0.1. Plants of Solar system

[1]

2 Essential Codes and Functions Analysis

For specific programs, we are extracted using samples of codes [2] in textbook. Furthermore, the running environment must to load **Modern OpenGL** (vertex and fragment shaders), **GLEW**, **GLFW**, **GLM**, and **SOIL2 Library** in configuration.

[1] Figure from https://en.wikipedia.org/wiki/Solar_System

[2] Prog6_1_sphere, Prog6_2_torus, Prog7_1_lightingADS and Prog9_3_environmentMapping respectively

2.1 Model Object

The main model objects to animate asters are sphere and torus [3] respectively.

2.1.1 Sphere

Firstly, we analysis the main ideas in **Sphere.cpp** in order to understand how to draw spheres by drawing inferences about other cases from one instance.

Some types of objects, such as spheres, torus, and so forth, have mathematical definitions that lend themselves to algorithmic generation. Consider for example a circle of radius R—coordinates of points around its perimeter are well defined(Figure.2.1.1).

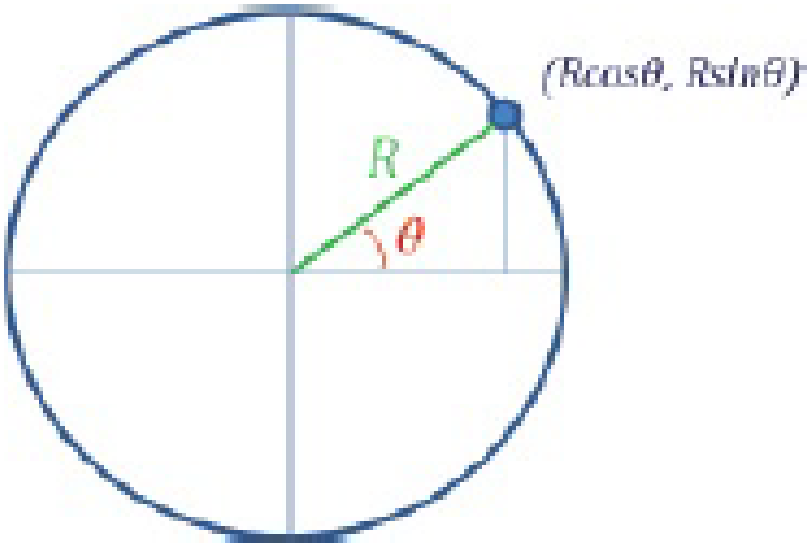

Figure. 2.1.1. R—coordinates
[1]

As the a circle of the coordinates, we can gain the strategy to get geometric sphere systematically.

1. Select a precision representing a number of circular *horizontal slices* through the sphere. (Figure.2.1.2)

2. Subdivide the circumference of each circular slice into some number of points according to the right side of Figure.2.1.2. More points and horizontal slices produces a more accurate

[3]The concrete class and functions also in Prog6_1_sphere, Prog6_2_torus

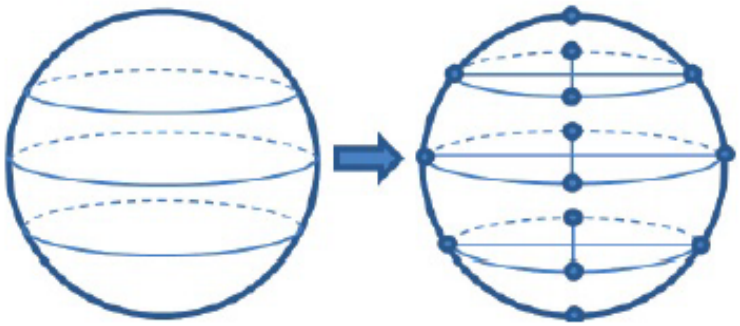

Figure. 2.1.2. R—coordinates
[1]

and smoother model of the sphere. In our model, each slice will have the same number of points.

3. We move along row of the vertices on the sphere in Figure.2.1.3 to group the vertices into triangles.

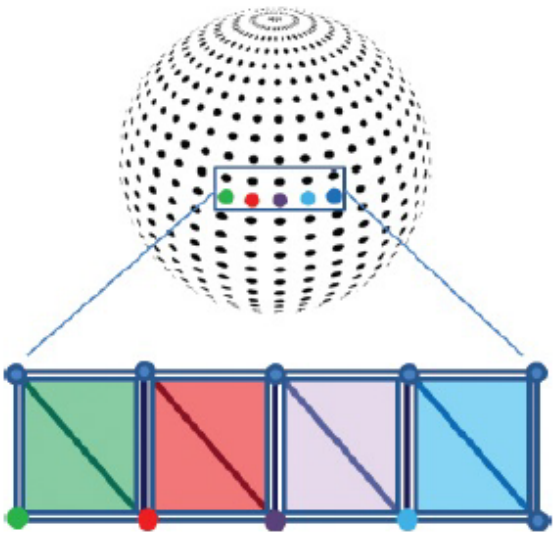

Figure. 2.1.3. R—coordinates
[1]

Then we can then traverse the vertices in a circular fashion around each horizontal slice, starting at the bottom of the sphere. We build two triangles forming a square region above and to its right, as shown earlier in Figure.2.1.3. The processing is organized into nested loops, as Figure.2.1.4.

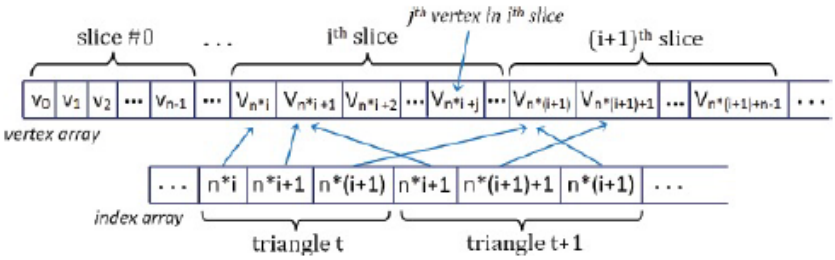

Figure. 2.1.4. Model Vertices
[1]

After getting geometry, the next point is to mapping the texture, which will be mentioned in Section.Texture Mapping.

2.1.2 Torus

For the torus, We ues **Torus.cpp**, which is rather different from what was done to build the sphere in **Sphere.cpp** according to the algorithm which positions a vertex to the right of the origin and then rotates that vertex in a circle on the **XY** plane using a rotation around the **Z** axis to form a *ring*. The ring is then moved outward by the *inner radius* distance. (Figure.2.1.5)

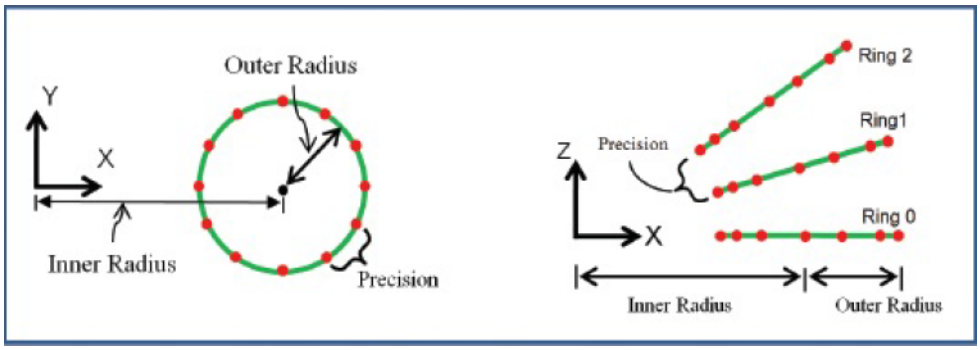

Figure. 2.1.5. Building Torus
[1]

To animate the ring of Saturn, the more difficult part is to regulate the parameters of ring, including thickness. When we see the original model in source code [4], the graphics is shown in Figure.2.1.6.

Thus, by controlling corresponding functions(Code.2.1.2), we can get a model resembling a flat ring(Figure.2.1.7).

[4]Prog6_2_torus, Prog7_1_lightingADS

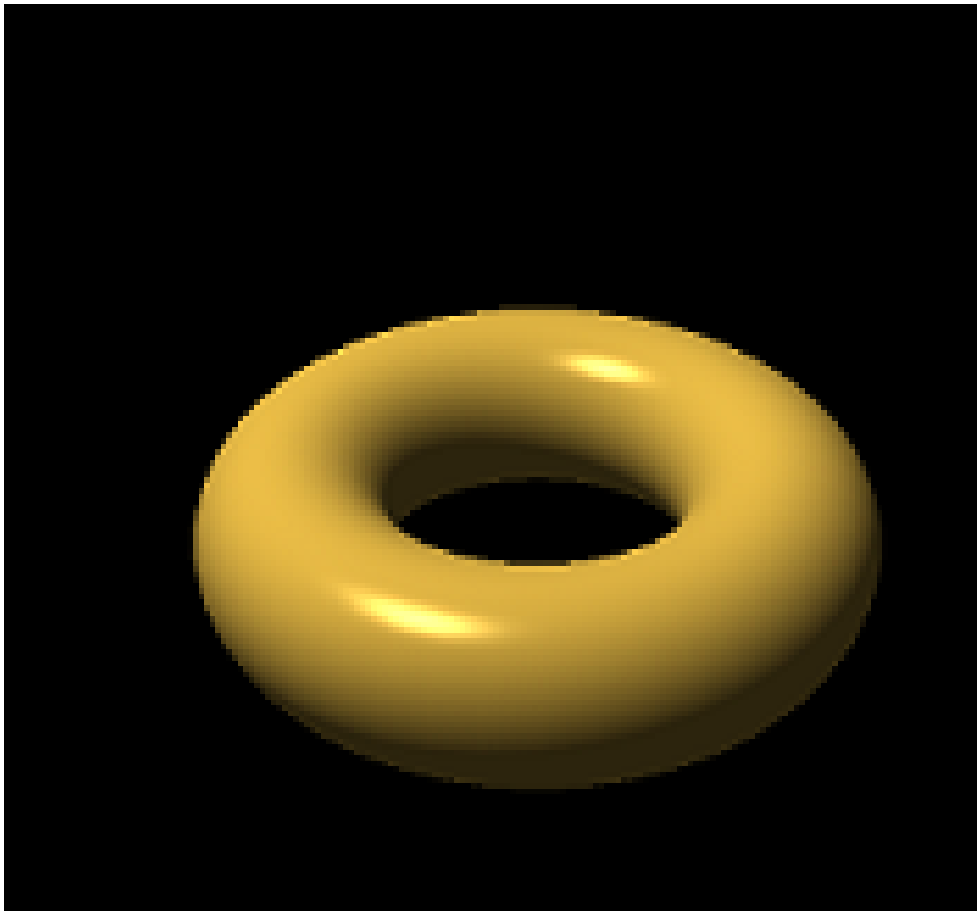

Figure. 2.1.6. Model Torus

```
Torus myTorus = Torus(1.5f, 0.15f, 48);

for (int i = 0; i < myTorus.getNumVertices(); i++) {
    pvalues1.push_back(vert1[i].x);
    vert1[i].y = 0;//torus thickness
    pvalues1.push_back(vert1[i].y);
    pvalues1.push_back(vert1[i].z);
    tvalues1.push_back(tex1[i].s);
    tvalues1.push_back(tex1[i].t);
    nvalues1.push_back(norm1[i].x);
    nvalues1.push_back(norm1[i].y);
    nvalues1.push_back(norm1[i].z);
}
```

Figure. 2.1.7. Model Ring

2.1.3 Matrix Stack

For single object, we can easily call specific functions in different class, but we aim to use a kind of data structure to contain the multiply objects displaying in the whole window. Therefore, we choose *stack* in **C++ Standard Template Library (STL)** relatively straightforward to adapt as a matrix stack. An overview of how a *display()* function using a matrix stack and the sequence of steps for the asters are typically organized in our textbook, so we may not repeat twice in report.

What we have changed compared with sample consists of scale, orbital position, inclination revolution and rotation. The specifically emphasizing points would be that the data of these characteristics is converted to parameters of functions in **GLM Library** by calculating, but to give a friendly angle of view for audiences, the some orbital position of asters are adjusted factitiously. The Code.2.1.3 which draws earth and moon shows how to realize.

```
//function which convert degree to radian
float toRadians(float degrees)
    { return (degrees * 2.0f * 3.14159f) / 360.0f; }
//Earth
mvStack.push(mvStack.top());
//scale
mvStack.top() *= scale(glm::mat4(1.0f),
    glm::vec3(1.05f, 1.05f, 1.05f));
//orbital position and revolution
mvStack.top() *= glm::translate(glm::mat4(1.0f),
    glm::vec3(sin((float)currentTime * 2.9) * 12.0,
        0.0f, cos((float)currentTime * 2.9) * 12.0));
mvStack.push(mvStack.top());
//inclination
mvStack.top() *= rotate(glm::mat4(1.0f),
    toRadians(23.43f), glm::vec3(0.0, 0.0, 1.0));
//rotation
mvStack.top() *= rotate(glm::mat4(1.0f),
    (float)(currentTime * 2.0),
        glm::vec3(0.0, 1.0, 0.0));
glUniformMatrix4fv(mvLoc, 1, GL_FALSE,
    glm::value_ptr(mvStack.top()));
/*some codes which are same as sample codes can be omitted*/
mvStack.pop();

//Moon no rotation
```

```
mvStack.push(mvStack.top());
//orbital position
mvStack.top() *= glm::translate
    (glm::mat4(1.0f),
        glm::vec3(sin((float)currentTime) * 2.0,
            0.0f, cos((float)currentTime) * 2.0));
//revolution
mvStack.top() *= rotate
    (glm::mat4(1.0f), (float)(currentTime),
        glm::vec3(0.0, 0.0, 1.0));
//scale
mvStack.top() *= scale(glm::mat4(1.0f),
    glm::vec3(0.25f, 0.25f, 0.25f));
glUniformMatrix4fv(mvLoc, 1, GL_FALSE,
    glm::value_ptr(mvStack.top()));
/*some codes which are same as sample codes can be omitted*/
mvStack.pop(); mvStack.pop();
```

2.2 Texture Mapping

After the object is fully created, the next step is to map the texture to the object. To know which texture image is picked for certain mesh, there are points that represent mesh coordinates and the mesh texture image path file.

To have the texture wrapped in the object, first, the texture image must be called by *loadTexture* function from file **Utils.cpp** which is be encapsulated from *SOIL_load_OGL_texture* function of **OpenGL SOIL Library**. The sample of this function are provided in Code.2.2. Then we are supposed to initialize textures of all asters in *init()* function.

```
GLuint myTexture = Utils::loadTexture("image.jpg")
```

Subsequently, the image can be assigned to a particular model or object. For the solar system project, the texture mapping process is quite easy because the texture image will only wrap the object. Then we add an uniform sampler variable(Code.2.2) declared in the shade to your set of uniforms.

```
layout (binding=0) uniform sampler2D samp;
```

The *layout (binding=0)* portion of the declaration specifies that this sampler is to be associated with texture unit 0. A texture unit (and associated sampler) can be used to sample whichever texture object you wish, and that can change at runtime. The *display()* function will need to specify which texture object you want the texture unit to sample for the current frame. So

each time drawing an object, we still need to activate a texture unit and bind it to a particular texture object in Code.2.2.

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, myTexture);
```

Finally, the whole process of texture mapping can be abstracted as Figure.2.2.1. [5]

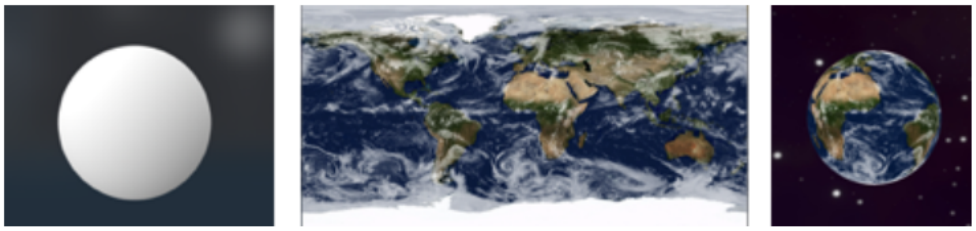

Figure. 2.2.1. Texture Mapping

2.3 Camera

We use **W**,**S**,**A** and **D** in keyboard to adjust the position of camera by Code.2.3. And the original view of camera is on the position of sun, in other words, we just to control the position of sun. And the operation of keyboard can be seen in Table.2.3.1.

```
void cameraMove(double currentTime) {
      //GLfloat curFrameTime = glfwGetTime();
      deltaTime = currentTime - lastFrameTime;
      lastFrameTime = currentTime;
      GLfloat cameraSpeed = 5.0f * deltaTime;
      //WASD
      if (keys[GLFW_KEY_W])
            cameraPos += cameraSpeed * cameraFront;
      if (keys[GLFW_KEY_S])
            cameraPos -= cameraSpeed * cameraFront;
      if (keys[GLFW_KEY_A])
            cameraPos += glm::normalize
                  (glm::cross(cameraFront, cameraUp)) * cameraSpeed;
      if (keys[GLFW_KEY_D])
            cameraPos -= glm::normalize
                  (glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```

[5] figure from https://medium.com/@keynekassapa13/creating-the-solar-system-opengl-and-c-9d4e4798d759

W	translate forward
S	translate backward
A	translate left
D	translate right

Table. 2.3.1. How to control keyboard

Then, in *display(currentTime)* function we must pass the position parameters into *vMat* and next set *vLoc* which is a variable allocation for display according to Code.2.3.

```
cameraMove(currentTime);
vMat = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
vLoc = glGetUniformLocation(renderingProgramCubeMap, "v_matrix");
```

Finally, to apply camera in window we need *key_callback* function in Code.2.3. Besides, in *main()* function we need call *glfwSetKeyCallback* to check whether the keyboard choosing

```
void key_callback(GLFWwindow* window, int key,
    int scancode, int action, int mode)
{
      if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
            glfwSetWindowShouldClose(window, GL_TRUE);
      if (key >= 0 && key < 100000)
      {
            //true & false
            if (action == GLFW_PRESS)
                  keys[key] = true;
            else if (action == GLFW_RELEASE)
                  keys[key] = false;
      }

}
```

2.4 Surface Lighting

There are a number kinds of light to apply by *.glsl* files in source code[6], and for this project we use **Position Light** initialization in **main.cpp**(Code.2.4)

```
// white light
float globalAmbient[4] = { 0.7f, 0.7f, 0.7f, 1.0f };
float lightAmbient[4] = { 0.0f, 0.0f, 0.0f, 1.0f };
```

[6]Prog7_1_lightingADS

```
4 float lightDiffuse[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
5 float lightSpecular[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

Subsequently, attenuation factors (Formula.2.4.1) which can be modeled in a variety of ways, have implication to **Position Light**.

$$attenuation = \frac{1}{k_c + k_l * D + k_q * D^2} \quad (k_c \geqslant 1, 0 \leqslant attenuation \leqslant 1) \tag{2.4.1}$$

$$\lim_{D \to \infty} attenuation = 0 \tag{2.4.2}$$

[7]

Then, the basic ADS computation that we need to perform is to determine the reflection intensity (I) which includes **ambient**, **diffuse** and **specular** for each pixel(Figure.2.4.1). This computation takes the following Formula.2.4.3.

$$I_{observed} = I_{ambient} + I_{diffuse} + I_{specular} \tag{2.4.3}$$

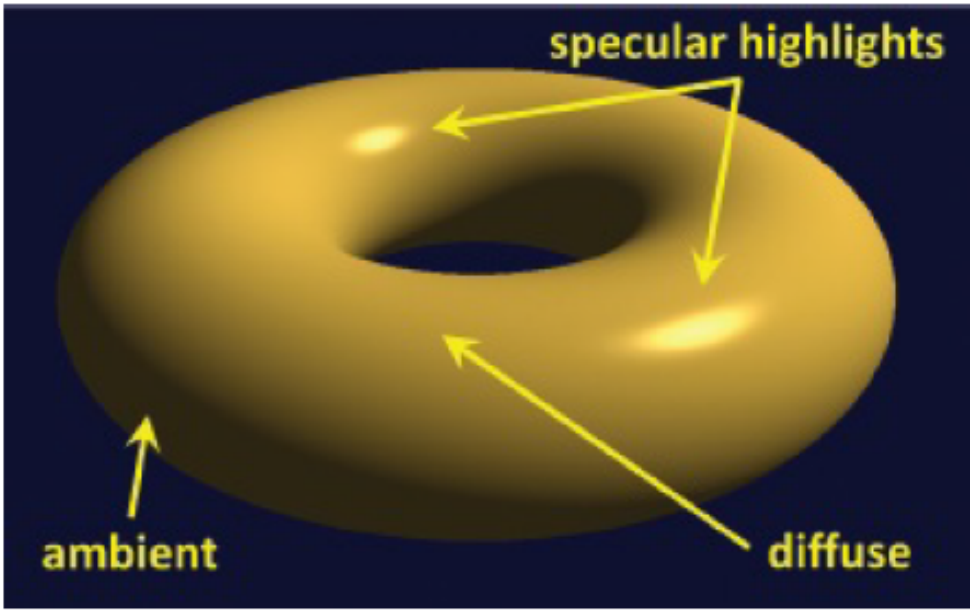

Figure. 2.4.1. ADS Lighting Contributions
[1]

Ambient contribution is the product of the specified ambient light and the specified ambient

[7] k_c, k_l, k_q, and D respectively refer to parameters for constant, linear, quadratic attenuation and the distance from the light source.

coefficient of the material(Formula.2.4.4).

$$I_{ambient} = k_a * Light_{ambient} \tag{2.4.4}$$

[8]

For diffuse contribution, it is more complex because it depends on the angle of incidence between the light and the surface(Figure.2.4.2) to get Formula.2.4.5.

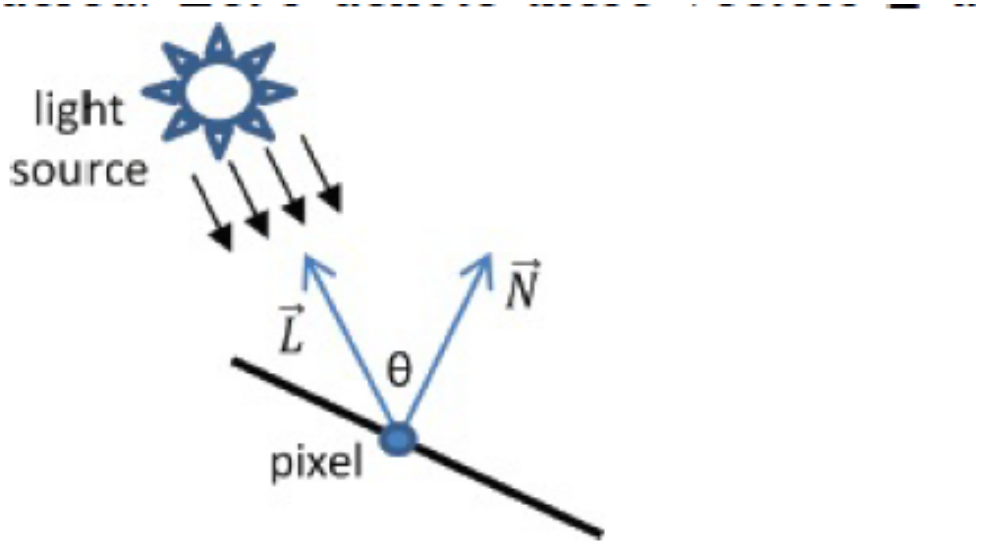

Figure. 2.4.2. Angle of Light Incidence
[1]

$$I_{diffuse} = k_d * Light_{diffuse} * \cos(\theta) \tag{2.4.5}$$

[9] And Formula.2.4.5 can be deducted to Formula.2.4.6.

$$I_{diffuse} = k_d * Light_{diffuse} * \max((\overrightarrow{N} \bullet \overrightarrow{L}), 0) \tag{2.4.6}$$

Specular contribution determines whether the pixel being rendered should be brightened because it is part of a **specular highlight**. It involves not only the angle of incidence of the light source, but also the angle between the reflection of the light on the surface and the viewing angle

[8] k_a refer to **Ambient Radiation Coefficient** of materials

[9] k_d refer to **Diffuse Radiation Coefficient** of materials

of the eye relative to the object' s surface(Figure.2.4.3). So the we calculate it by Formula.2.4.7.

$$I_{specular} = k_s * Light_{specular} * \max((\overrightarrow{N} \bullet \overrightarrow{L})^n, 0) \tag{2.4.7}$$

[10]

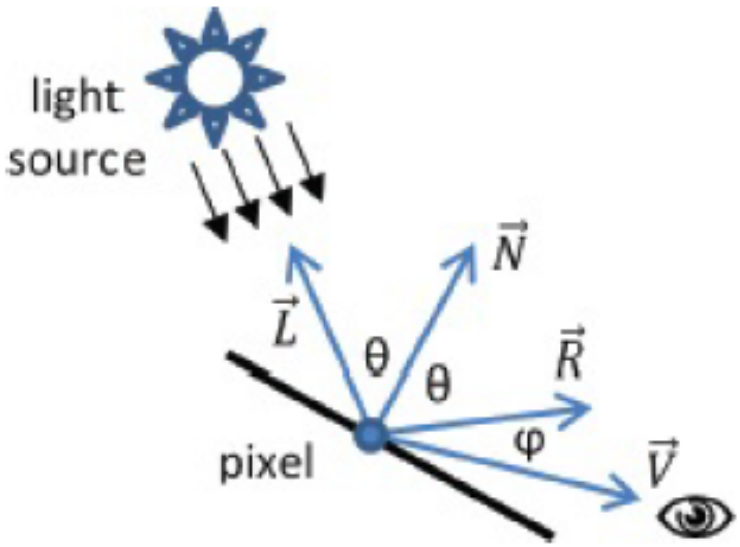

Figure. 2.4.3. View Angle Incidence
[1]

Finally, we combine surface texture and light to the asters except sun according to Formula.2.4.8. Because sun is the position of light without adding shade to make it dark.

$$C_{frag} = C_{texture} * (I_{ambient} + I_{diffuse} + I_{specular}) \tag{2.4.8}$$

[11] We can easily add the code(Code.2.4) in *fraghShade.glsl* to realize the aim.

```
vec4 textureColor = texture(s,tc);
color = textureColor*vec4(ambient + diffuse + specular, 1.0);
```

[10] k_s refer to **Specular Radiation Coefficient** of materials

[11] C refer to color

2.5 Sky cube Mapping

To mapping the texture to the whole background, it is simple to create cube model in 3D to achieve this aim which is similar to the contents of Section.Texture Mapping. Firstly, we initiate the vertices of cube model by *float cubeVertexPositions[108]* in *setupVertices()* function. Then we add codes in *init()* function, but we use *loadCubeMap()* function from **Utils.cpp** in order to load texture on six sides of cube in Code.2.5.

```
renderingProgramCubeMap =
    Utils::createShaderProgram("vertCubeShader.glsl", "fragCubeShader.glsl");
//load map to cube as background
backgroundTexture = Utils::loadCubeMap("background");
glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS);
```

Because we add lighting in **vertShader.glsl** and **fragShader.glsl**, we need create new files of shade–**vertCubeShader.glsl** and **fragCubeShader.glsl** realization of sky cube.

3 Output and Program Test

When running the program, the view of camera is on the position of sun which can only see the asters surround, so we need to press down **S** in keyboard to put camera far form the origin. After that we repeat adjust to find different angle to observe the solar system model. And the output in window can be seen in Figure.3.0.1. (Specific way to use keyboard to control camera can be review in Section.Camera)

4 Summary

Through this experiment, we learned and mastered how to transformations, texture mapping, lighting and window event handling and realized the region filling. We also know that graphics programming has a reputation for being among the most challenging computer science topics to learn. In the process of compiling the algorithm by pushing and popping stack repeatably, we constantly improve the composition of a picture and solve the difficulties with the help of classmate. However, there are still some improved effective function which has not been written and implemented, and the program is constantly optimized in the continuous improvement,

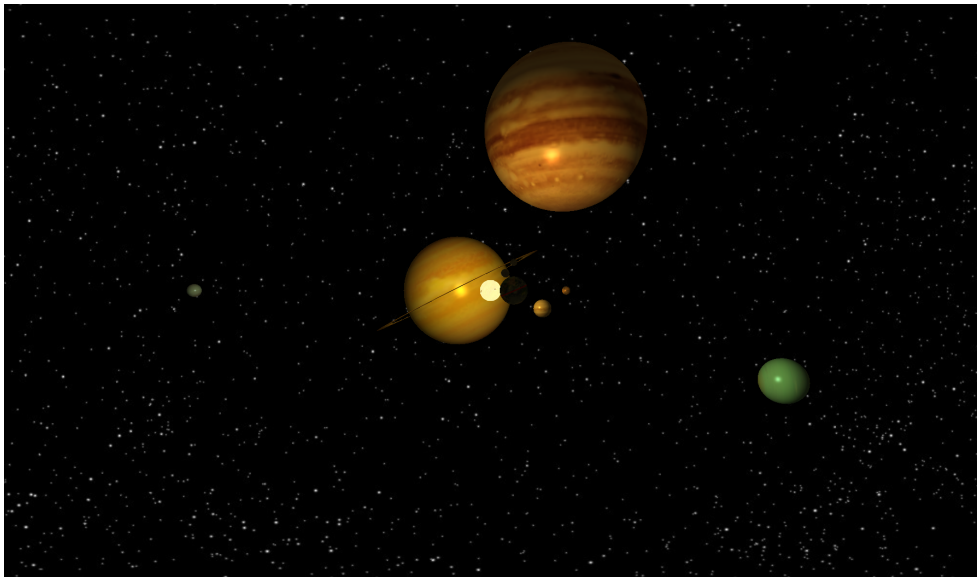

Figure. 3.0.1. Sample of Output

Reference

[1] V. Scott Gordon. *Computer graphics programming in OpenGL with C++ / V. Scott Gordon, Ph. D., California State University, Sacramento, John Clevenger, Ph. D., California State University, Sacramento.* Mercury Learning and Information, Dulles, VA, 2019.