Blue text on a black background

Description automatically generated

Simple 3D representation of planets with OpenGL

By: Maxime Pichet

INFO-H502

Presented to Gauthier Lafruit

December 19th 2023

Contents

[Project description 1](#_Toc153117000)

[Sun 1](#_Toc153117001)

[Earth 3](#_Toc153117002)

[Reflective and refractive sphere 5](#_Toc153117003)

[Particle system 6](#_Toc153117004)

[Cube Map 7](#_Toc153117005)

[Annex A: 9](#_Toc153117006)

# Project description

This project was made using OpenGL and C++. It consists of 4 planets, notably the sun, the earth, a reflective sphere and a refractive one. The sun rotates on itself as the earth rotates on itself and around the sun. The reflective sphere also rotates around the sun. In sum, it is a 3d model of some planets in space.

# Sun

The center piece of the scene is the Sun. The sun was made using a basic sphere that was modeled in blender and then imported in the program as an object file. In order to add some texture to this sphere, a picture of the sun was mipmapped onto the sphere.

Picture 1. Texture of the sun

A close up of a fire

Description automatically generated

This picture was loaded using the stb library (see Annex). The texture data was then passed to the vertex shader using the buffer.

A computer screen shot of a black screen

Description automatically generated

Figure 1. vertex shader for the Sun

We can see here that the tex\_coord which are the coordinates for the texture, were passed to the fragment shader.

A computer screen shot of white text

Description automatically generated

Figure 2. Sun fragment shader

In the fragment shader, I decided to only apply diffuse lighting as all points of the sphere should emit the same amount of light. Using the texture() function of glsl, we can sample the texel of the texture and apply it at the f\_tex\_coord or the pixel of the texture.



Figure3. Texture sampler passed to shader.

To have access to the texture, we pass as a uniform a 2d sampler that correspond to the texture. Here we passed 2 since when loading the texture (see Annex), we saved it in location 2. Then the sun was ready to be drawn.

# Earth

The earth was a bit more complex to make as I added multiple elements to it. The texture was done the same way as for the sun but using a picture of the earth instead. What is interesting about the earth is I added bump mapping to make it seem like there is elevation on the surface.

A planet earth with the sun shining

Description automatically generated with medium confidence A planet earth from space

Description automatically generated

Figure 4. Bump mapping

This was achieving using a normal map.

A blue background with a white line

Description automatically generated with medium confidence

Figure 5. Normal map of the earth

The normal map seems all purple but if you look closely, you can see some differences. To correctly apply the normal map to the texture, the tangents and bitangents of the surface had to be calculated. Then those calculations were passed to the vertex shader using the buffer where the TBN (tangent, bitangent, normal) matrix was computed and passed to the fragment shader.

A screen shot of a computer code

Description automatically generated

Figure 6. Vertex shader of the Earth

A computer screen shot of a black screen

Description automatically generated

Figure 7. Fragment shader of the Earth

In the fragment shader presented above, we create and normalize a texture (normalMap) which is the picture presented in figure 5. This is then multiplied with the TBN matrix and normalized to have unit vectors. We then have the new normals which are used to calculate the lighting equations.

The earth also has different model transformations for the animation which you can see in the source code and the short video.

# Reflective and refractive sphere

The implementation of the reflective sphere was straightforward. The work must be done in the fragment shader in order to have a cubemap reflection on the sphere.

A computer screen with white text

Description automatically generated

Figure 8. Fragment shader for reflection

From the normalized normals and the view vector, we can calculate the reflection vector used to sample the cube. The nifty reflect function from glsl does the math for us. Otherwise, equation 1 could be used to calculate it.

Eq 1. Reflection calculations from vector

The refractive sphere was quite similar but the fragment shader differs a bit.

A computer screen shot of a program code

Description automatically generated

Figure 9. Fragment shader for refraction

Here, the refract function from glsl is used with a ratio which is denoted as eta in the documentation. It is the ratio from material A to B in other words going from air to water let’s say. We used 1.00 since we are in space so we should not have any refraction from the environment. Then we call the refract function and generate the texture with the refraction vector. In the simulation the two balls were put side to side to see the difference between reflection and refraction.

# Particle system

One aspect of the animation that was complex is the particle system. Around the sun, we can see some orange particles flying of in every direction. This is representing the sparkling of the sun. To achieve this particle system, hundred of particles were generated following this struct.

A screen shot of a computer screen

Description automatically generated

Figure 10. Particles struct.

Each particle has a position, velocity, lifetime and a color. A vector of Particle could then be generated with random starting positions inside the sphere and move out with a certain velocity. At the end of their lifetime, the particles would be deleted, and new ones would be generated.

A screen shot of a computer program

Description automatically generated

Figure 11. Particle update function

Basically, at each iteration, particles would move at a constant speed, diminish in color and reduce their lifetime. The last few lines of code shows a concise way of removing dead particles. It uses two iterators to rearrange the vector by putting the particles meeting the condition at the end. After that, it erases the particles that are at the end of the vector. Then, we render the vector as points, and we have a particle system.

A black background with white letters

Description automatically generated

Figure 12. Particle generation

A computer code on a black background

Description automatically generated

Figure 13. Particle vertex shader

A black screen with white text

Description automatically generated

Figure 14. Particle fragment shader

The fragment and vertex shader are very simple as only colored points were required.

# Cube Map

The cube map consists of loading a picture on each face of a cube and then making each vertex the deepest possible.

A screen shot of a computer code

Description automatically generated

Figure 15. Vertex shader of cubemap

It is explained in figure 15 that we remove the translation because we want to make it infinitely far so we should not move the cube. Afterwards, we calculate the position of each vertex as their position with z = w since w will always be 1.0 because we set it so in vec4(position, 1,0). Hence, we can pass z as w and we have the cube at maximum depth. Then it’s just a matter of loading the right images on the right faces. We can do so by creating a GL\_TEXTURE\_CUBE\_MAP which is similar to a 2d texture but each mipmap level has 6 faces. So, using a samplerCube in glsl let’s us create a cubemap with ease.

# Annex A:

A screen shot of a computer program

Description automatically generated

Figure 16. texture loading

Loads texture and assigns it to a textureUnit number to have multiple textures.