IGR200 - Interactive 3D Application Development

Solar system project report - **Emile LE GALLIC**

1. The first spheres

At first, I used the Mesh class. The goal of the **`genSphere`** was to generate a sphere mesh with a specific resolution. In order to do this, it was necessary to divide the sphere in sectors. Each point of the sphere can be expressed thanks to its parametric equation :

$$\begin{cases} x = r \cdot \cos\phi \cdot \cos\theta \\ y = r \cdot \cos\phi \cdot \sin\theta \\ z = r \cdot \sin\phi \end{cases}$$

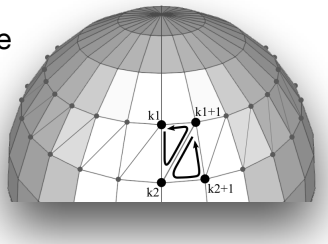
The idea is to gradually place the points by incrementing both phi and theta by $\pi \cdot \frac{i}{\text{resolution}}$ and $2 \cdot \pi \cdot \frac{j}{\text{resolution}}$ where i is the current step for the vertical angle (between 0 and π) and j is for the current step for the horizontal angle (between 0 and $2 \cdot \pi$).

Therefore, we have a first loop for ϕ the vertical angle and a nested one for θ the horizontal angle and so :

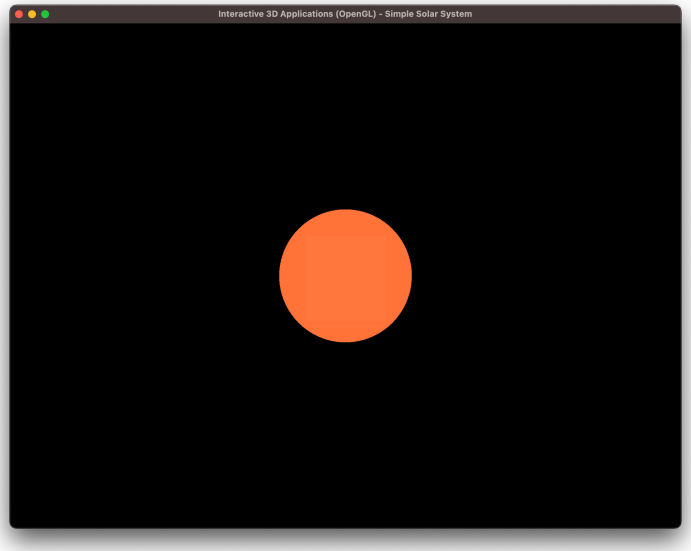
$$\begin{cases} x = r \cdot \cos\left(\pi \cdot \frac{i}{\text{resolution}}\right) \cdot \cos\left(2 \cdot \pi \cdot \frac{j}{\text{resolution}}\right) \\ y = r \cdot \cos\left(\pi \cdot \frac{i}{\text{resolution}}\right) \cdot \sin\left(2 \cdot \pi \cdot \frac{j}{\text{resolution}}\right) \\ z = r \cdot \sin\left(\pi \cdot \frac{i}{\text{resolution}}\right) \end{cases}$$

For the normals, it is easy to compute since for a sphere, it just the line from the center of the sphere to the vertex. Therefore, the coordinates computed previously also work.

When it comes to triangulate adjacent vertices, the task is a bit more complex. We must compute triangles between two horizontal divisions as shown in the picture on the right.



At this point, I managed to render a correct sphere. I also added a color to it using a color vector in my fragment shader.



Picture 1 - my first sphere

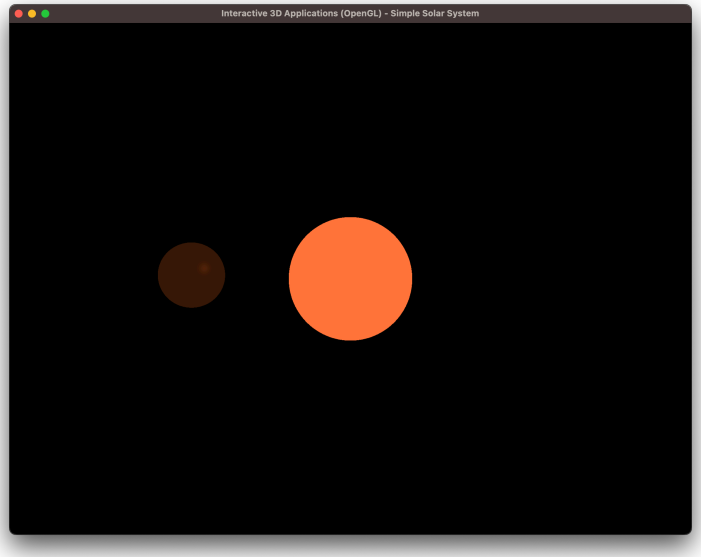
Then, I added the shading and the phong lighting. At this point I didn't realized that my phong lighting wasn't working well as seen in *Picture 2*. I fixed it later in (*3. Textures*).

At this stage I also decided to change the structure of my object and to create a « **`CelestialObject`** » class which would represent an object in my solar system. A celestial object has many properties such as a type (**`Star`** or **`Planet`**), a radius, possibly a parent, possibly a orbit period…

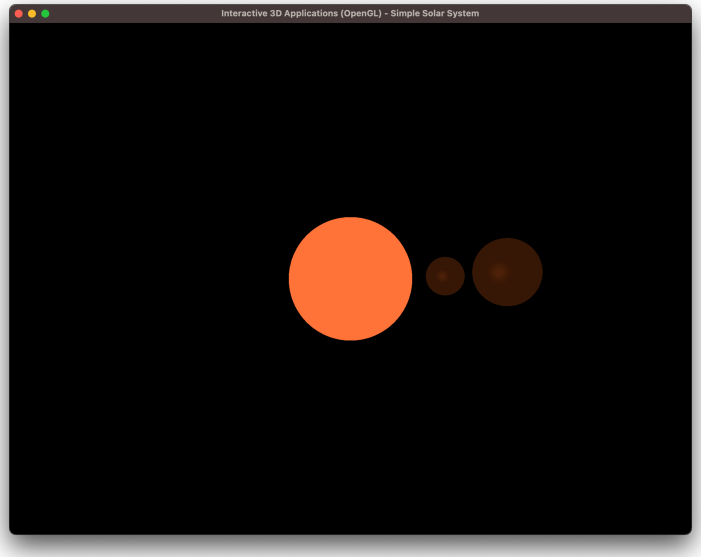
2. Rotations and orbits

The next step was to begin to apply transformation to my celestial objects. I began using **`glm::translate`** and then **`glm::rotate`**.

I managed to move my second planet away from my sun (*Picture 2*).



Picture 2 - a second sphere…

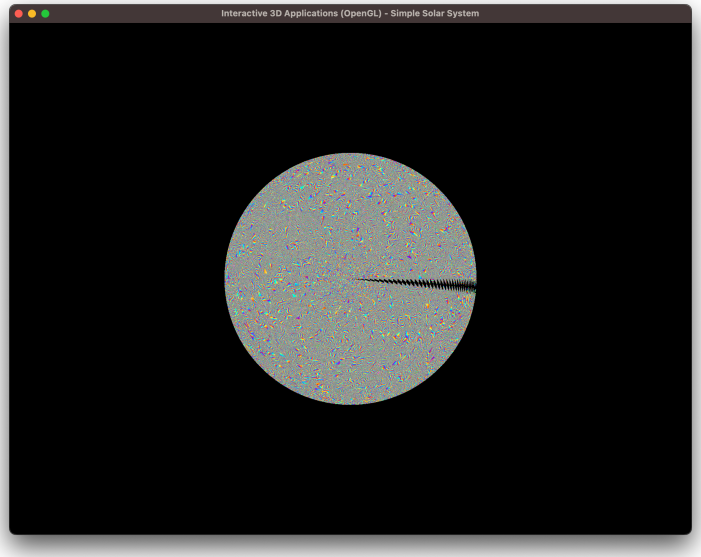


Picture 3 - … and a third one

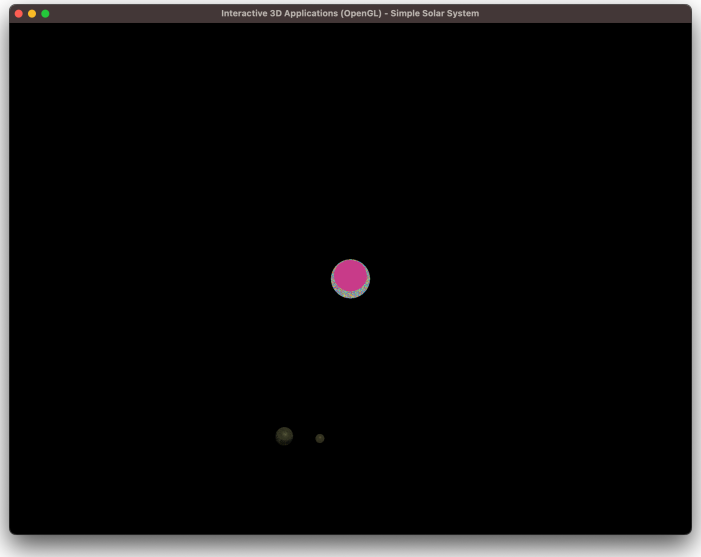
I then added planets rotation around the sun. The idea was to compute, thanks to the current delta time, the position of the planet knowing the orbit radius and period since $\theta_{\text{orbit}} = 2 \cdot \pi \cdot \frac{1}{\text{orbitPeriod}} \cdot \text{deltaTime}$. We then translate the planet to its newly computed center in the render loop.

3. Textures

I struggled a bit (*Picture 4 & 5*) with the textures especially to compute the vertex texture coordinates. The idea is to divide the texture image 2D image into square sectors of the length of the resolution.

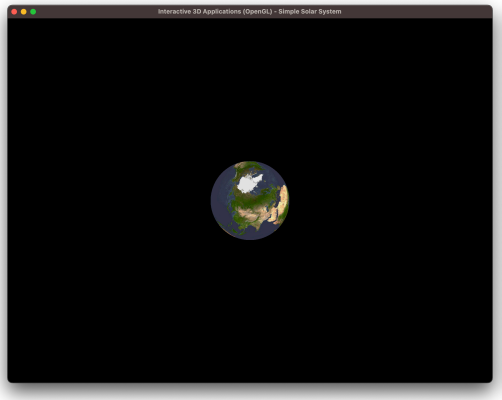


Picture 4 - psychedelic earth

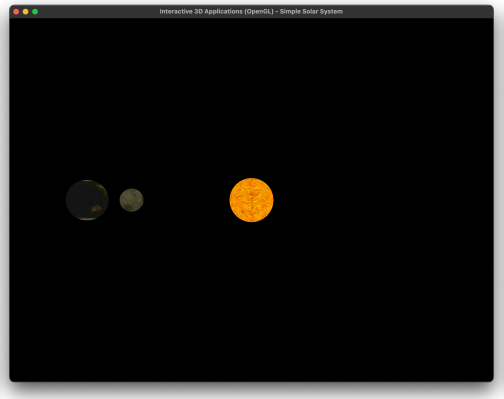


Picture 5 - a pink world

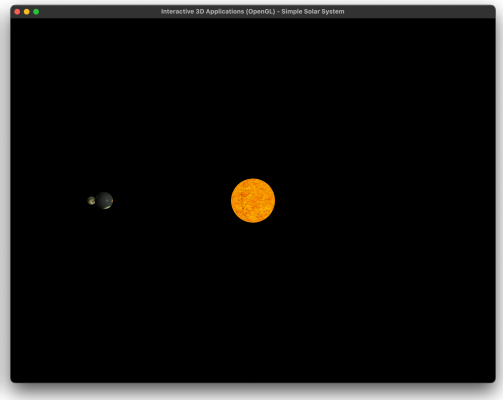
I then added the different textures for the Earth, the sun and the moon and managed to get a first working solar system (*Picture 7*). Yet at this point I realized that my lighting was broken. The reason was that when I added transformations to my spheres, I did not take those into account in my vertex shaders. Therefore, the fragment normals and positions where wrong. I changed it and managed to fix the lighting.



Picture 6 - working textures



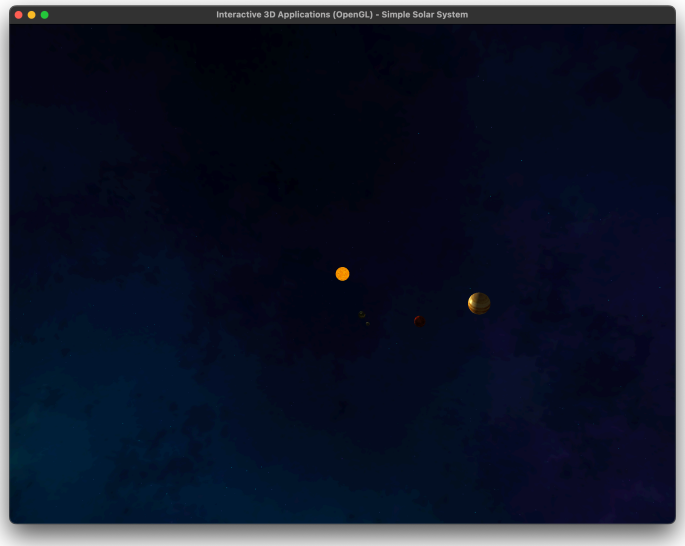
Picture 7 - textured solar system



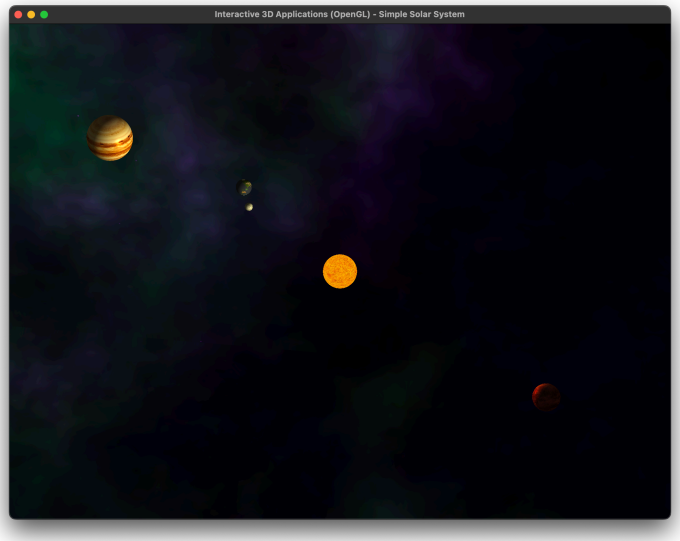
Picture 8 - fixed lighting

4. Further features

Finally, I decided to customize it a little bit. I added the ability to control the zoom with the wheel and move the camera with the arrow keys. Then, I added a skybox to render a nice space background.



Picture 9 - skybox !



Picture 10 - camera movements

5. Notes and usage

The project uses 3 main classes :

- **`CelestialObject`** which is either a **`Planet`** or a **`Star`** and which as all the properties of a celestial object (an orbit period, a radius, a rotation period, a parent star around which it orbits, a texture, …)
- **`Camera`** which controls the camera and handle the inputs from the keyboard / mouse
- **`Skybox`** which generates the space background

The **`main.cpp`** file makes takes care of the running of the program.