CSCI3260 Computer Graphics Course Project

Students: WONG Chi Kwan Cyrus, LAU Ho Man Elvis

Introduction

In this project, we are asked to render a scene as below, with the name space travel. On top of that, we further extent it to try to implement the classic arcade game: space invader. The spaceship will now be able to shoot rockets to knock out the space vehicles.

Lighting Examples

Fig 1 The figure which shows the general scene

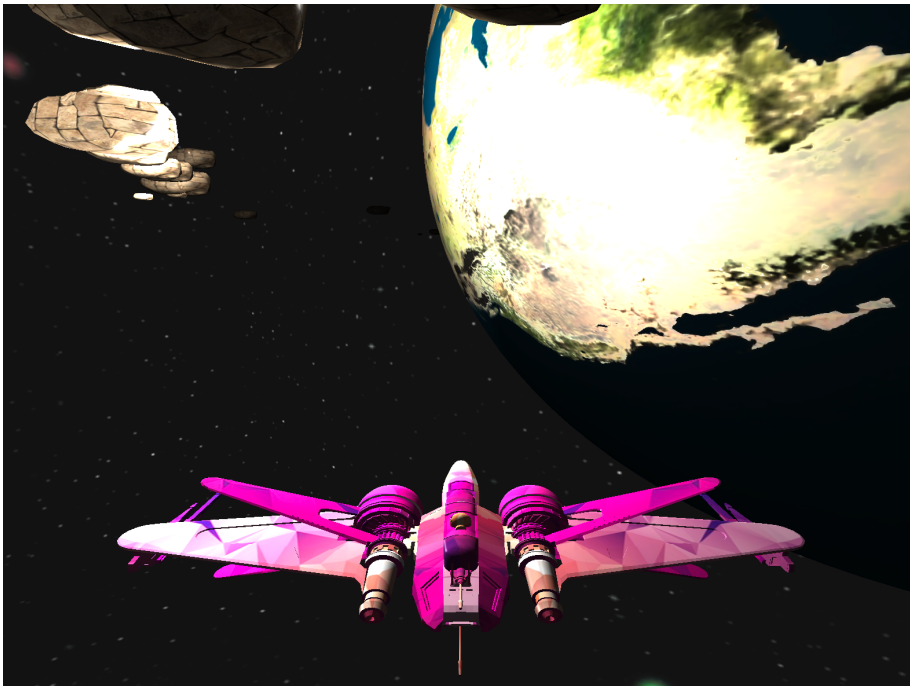

Fig 2.1 Basic light rendering of spaceship, asteroids, and the planet

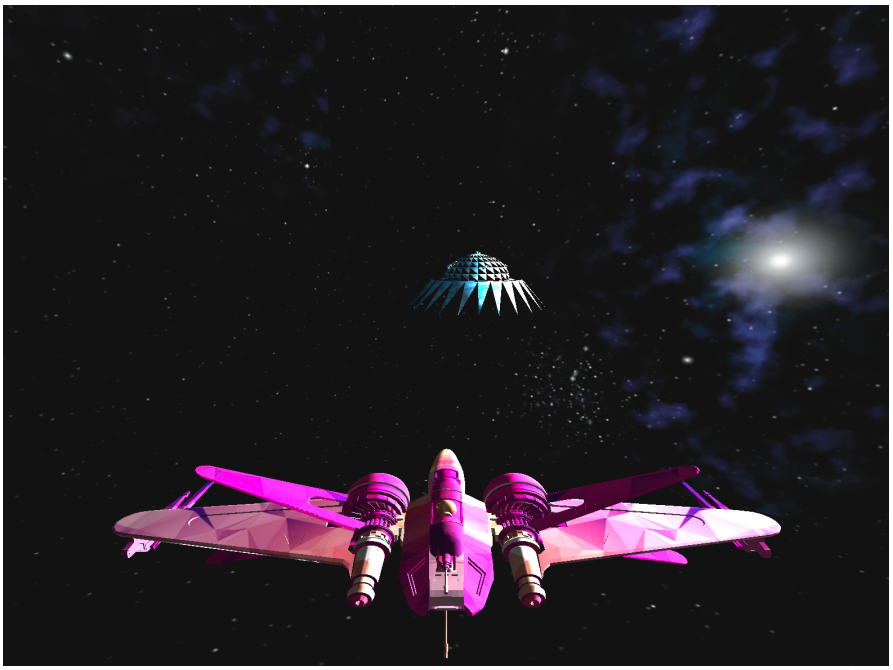

Fig 2.2 Basic Light rendering of the space vehicle

Implementation of Basic requirements

1. Self-rotation of the planet and space vehicle

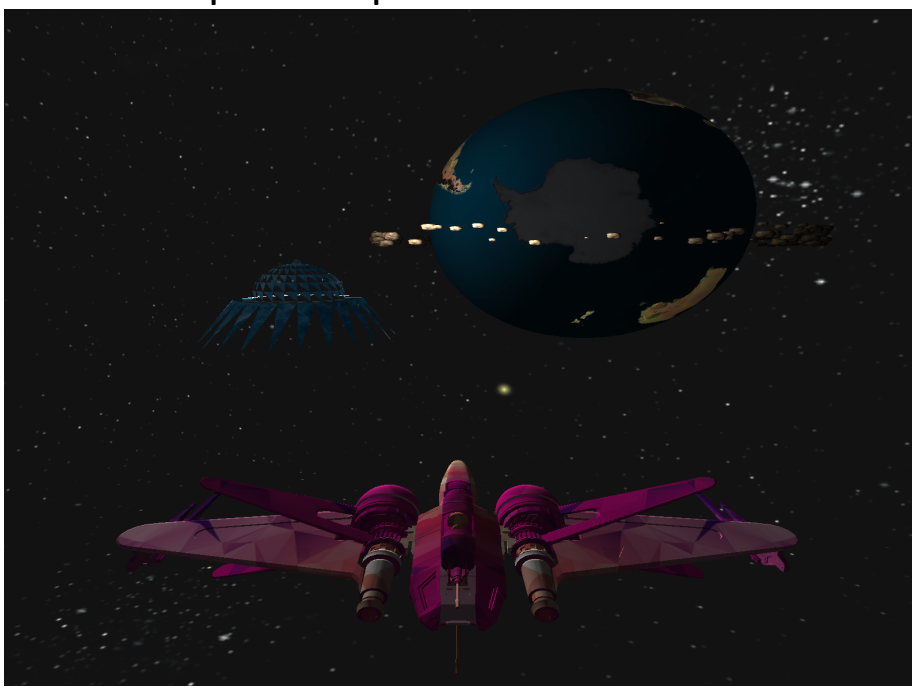

Fig 3.1.1 Self-rotation of planet and space vehicle

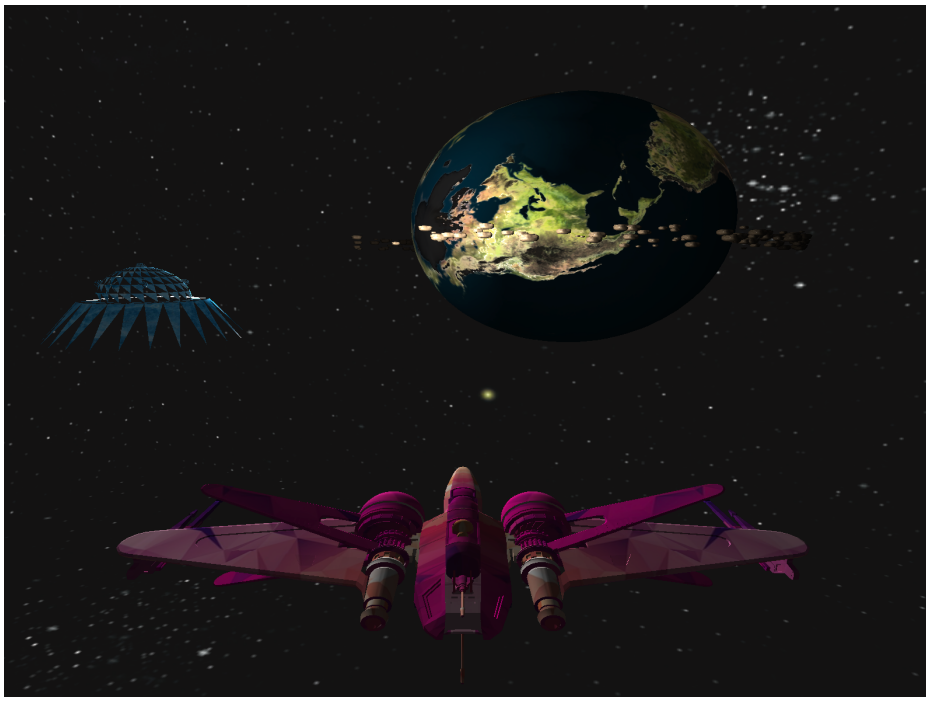

Fig 3.1.1 Self-rotation of planet and space vehicle

For the implementation of self-rotation, a global variable ‘rotation’ is used to keep track of the degree of rotation of the planet and the space vehicle. We use the function glfwGetTime() to obtain the time whenever paintGL() is called. We set another global variable prevTime to record the previous time that the rotation is updated. By such we can ensure that for every 0.05 seconds, rotation will be incremented by 1 to realize self-rotation.

2. Skybox rendering

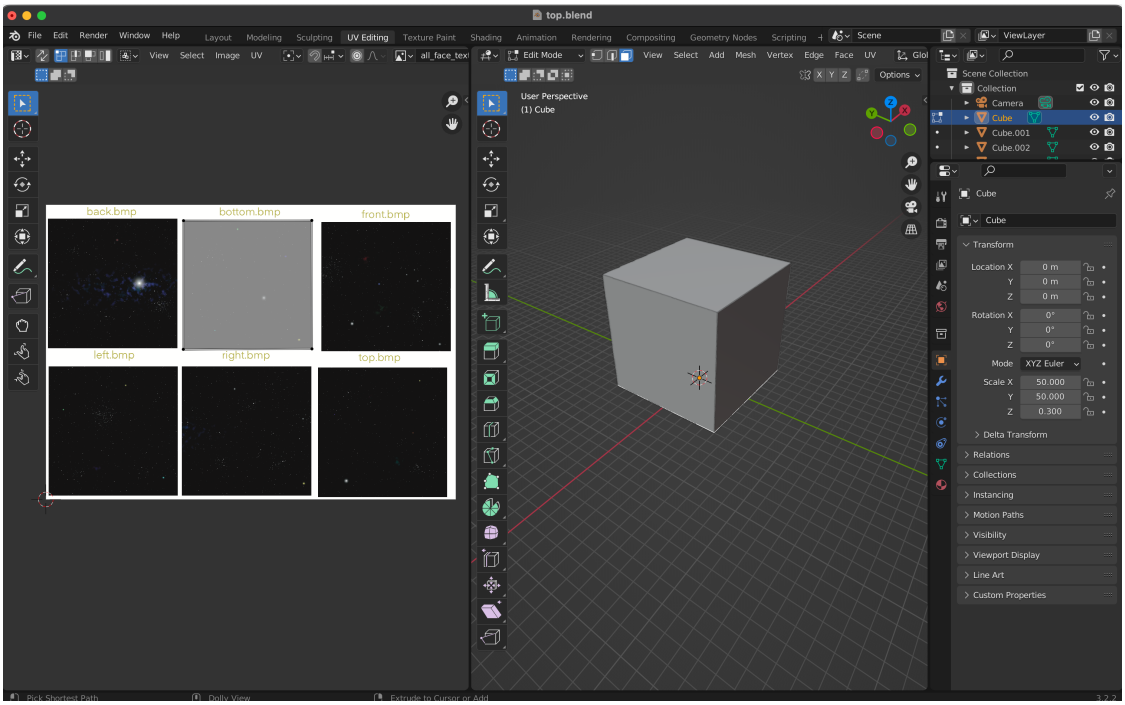

Fig 3.2.1 Modeling of skybox in Blender

Rendering skybox is simple but bothersome. In our implementation, we draw 6 planes, each correspond to the 6 sides of skybox. We also put the 6 given .bmp image files into one .png file, for the sake of simple UV mapping. With these simple setups, we can map the textures to the corresponding faces with the aid of Blender. Then, put it inside of our space invader's world, and skybox is done.

3. Asteroid ring cloud rendering & rotation of rocks

Fig 3.3.1 Close up of the asteroid cloud ring

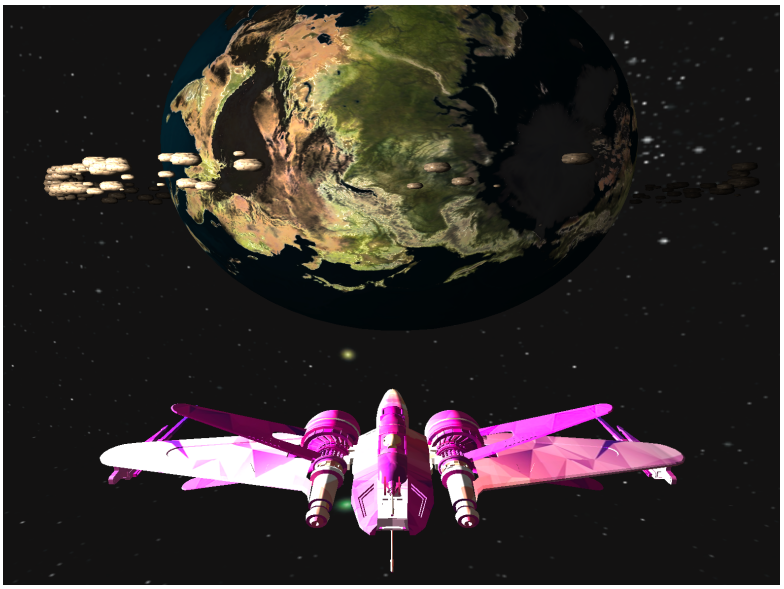

Fig 3.3.2 Another close up of the asteroid cloud ring

The randomized positions of the asteroids are determined when initializedGL() is called. We first set the space where asteroids could appear. This is bounded by the global variables yMax, yMin, circleMax, circleMin. We also want to have different sizes of the rocks. Therefore, we have the scaling bounds scaleMax and scaleMin. The y, z coordinates are then generated according to the above bounding parameters. The x coordinate is generated using the result of equation of circle: $x^2 + z^2 = 35^2$, to give a circular path.
The implementation details of the rotation part is the same as 'Self-rotation of the planet and space vehicle'.

4. **Camera viewpoint**

We rotate the world while keeping the camera and spacecraft constant for a static camera viewpoint. Whenever the mouse cursor moves, we calculate its offset relative to the midpoint of screen. This offset is then converted to angle of rotation, for both spacecraft and camera. Hence, viewpoint could be kept relatively static.

5. **Normal mapping of planet**

Fig 3.5.1 Planet without normal mapping

Fig 3.5.2 Planet with normal mapping

Normal mapping is done on the planet. We use another texture sampler in the fragment shader to load the normal map as image, then convert it and normalize it as normal vectors with range bounded in between [-1.0, 1.0].

6. Movements of spacecrafts

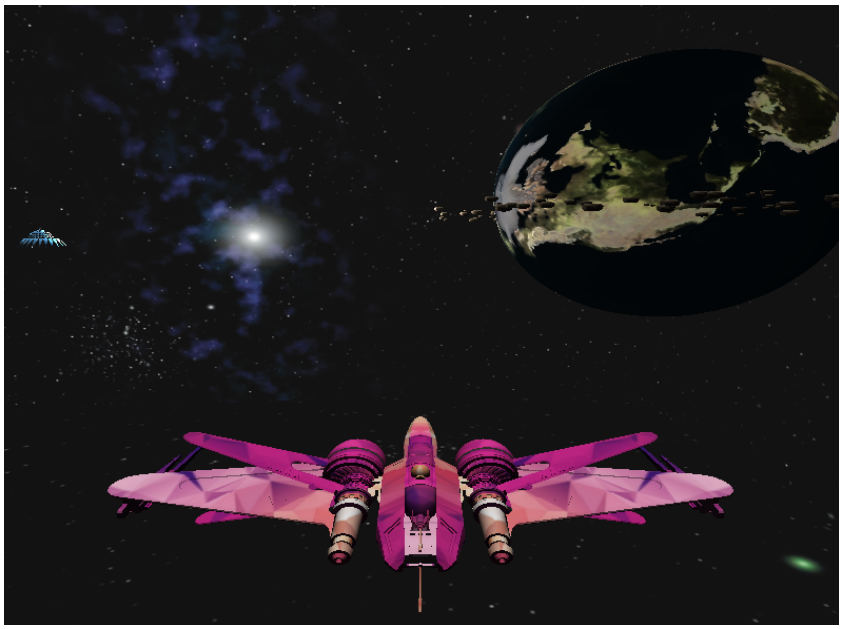

Fig 3.6.1 Moving spacecraft to a new position

Fig 3.6.2 Moving spacecraft to another position

The implementation of spacecraft rotation is described together with the camera viewpoint. For the translation part, we keep global variables storing the number of times <up>, <down>, <left>, <right> buttons are pressed, then use these variables to obtain the location of spacecraft in world coordinates.

Implementation of Bonus requirements

1. Shooting rockets

Fig 4.1.1 Shooting rockets

Fig 4.1.2 Shooting rockets while moving viewpoint

We keep an array to store the locations of rockets with its normalized direction of shooting, which is initially empty.

If <space> is pressed, append a location with the direction of spacecraft, which is the current position of spaceship and current direction of spacecraft, to the array.

Every time a certain time interval is passed, update the current position of rockets, by the direction of shooting.

Once it is outside of the skybox, remove it from the array.

Once it collides with a UFO, remove it from the array.

We only need to draw according to array of the location of rockets and its orientation.

2. Collision

Fig 4.2.1 Space vehicle is “damaged”

Given the locations of rockets, the locations of UFOs, we can have a function to calculate whether the rocket is collided with any of the UFO. This can be realized by calculating the distance of UFO and rocket. If true, in other words, if it is lower than a certain threshold, on the first hit, UFO should turn into red. (Change the texture mapping). On the second hit, it should disappear. (Remove its location from array)

Manipulation

Key <w>
- Increase the intensity directional light

Key <s>
- Decrease the intensity of directional light

Key <up>
- Move spacecraft and viewpoint towards the z direction (front)

Key <down>
- Move spacecraft and viewpoint towards the negative z direction (back)

Key <left>
- Move spacecraft and viewpoint towards the x direction (left)

Key <right>
- Move spacecraft and viewpoint towards the negative x direction (right)

Key <esc>
- Exit

Key <space>
- Shoot rocket to the direction spacecraft is facing

Key <r>
- Reset space vehicle to intact