Coursework 2: Animated Scene

The animated scene I have chosen to create for this course work is a model Solar System, featuring 8 planets, 10 moons, and artificial satellites. This was constructed using OpenGL, GLUT, and Qt. The basic idea of this model is that there is a Sun, with the 8 planets orbiting it, whilst each planet have moons that also orbit them.

How does this model work?

Whilst this is a graphical model, the actual mechanics of it are encapsulated in a class called `SolarSystem`, which organizes the planets in our model, and each planet in represented by the class `Planet`. Furthermore, each `Planet` object organizes a collection of `Satillite` objects, which represent the moons and artificial satellites that orbit each planet.

The `SolarSystem` class has two important methods which we should talk about: `tick()` and `normalise()`.

The method `tick` (solarsystem.cpp::26) represents movement in our system. When it is called each planets position and rotation is updated in accordance to the current `speed` of the model, which is a parameter of `tick`. This method will called the method `updatePosition` (planet.cpp::58) on each `Planet` object is has stored. The planets new position is calculated by taking the product of the current `speed` and the planets calculated `orbitSpeed`, which is measured as degrees per Earth day. The planets `orbitSpeed` is computed by dividing 360 by the number of days it takes to orbit the sun, which is passed through `Planet`'s constructor (planet.h::28). The exact same process is repeated for each `Planet` in relation to each `Satillite` it has stored. An identical process takes place for the rotation of `Planet`, where the rotation speed is determined by dividing 360 by the number of Earth days it takes a planet to rotate an entire rotation on it's axis.

The `tick` method has a single parameter `float speed`, from where `tick` is called (viewwidget::488), this value is controlled by a slider on the main UI (see figure 1).

Each `Planet`'s position represents the rotation around the y-axis required to place the object in the correct position; this value can range from 0 to 360 (where 0 is the original starting position) and when the position exceeds 360, we then take 360 away from the

position, which allows the object to move smoothly, and prevent any ‘janky’ movements. The exact same logic applies for the rotation of the `Planet` and each `Satillite`.

```
void Planet::updatePosition(float speed)
{
    this->position += speed*this->orbitSpeed;
    if(this->position > 360.)
    {
        this->position -= 360.;
    }
    this->rotation += speed*this->rotationSpeed;
    if(this->rotation > 360.)
    {
        this->rotation -= 360.;
    }
    this->updateSatillitePositions(speed);
}
```

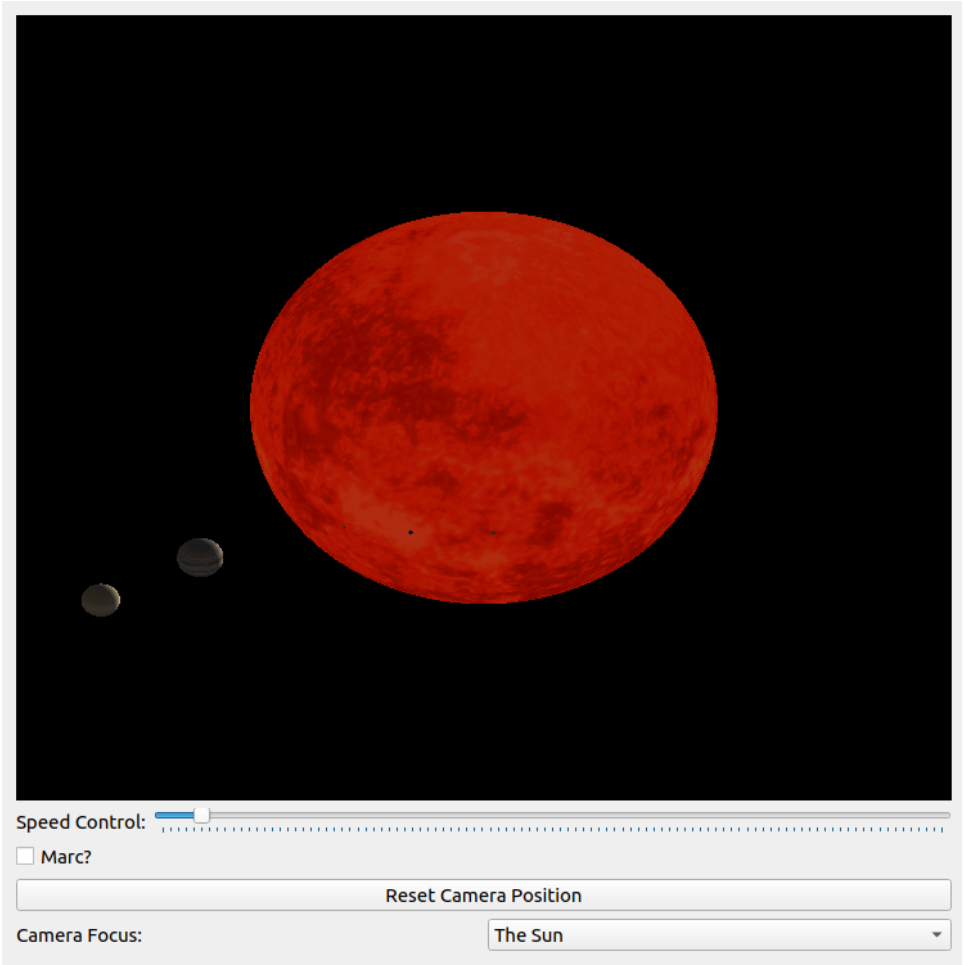

Figure 1: The UI of the animated scene.

The second important function that the `SolarSystem` object has is `normalise` (solarsystem.cpp::39). This function is called only once. The role of `normalise` is to reduce the distances and sizes of the objects so that they are a feasible size and distance from the origin. This function will take the `Planet` object with the largest size and the `Planet` object with the largest distance from the sun, and will normalise each object's size and distance based on those two values. The `Planet`'s with the highest size and distance are not necessarily going to be the same planet. The result of this function will be that each planet has a size that ranges between 0 and 1, and a distance that also ranges between 0 and 1. This means that the objects will stay in frame and are a tractable distance from the origin, whilst also maintaining the ratio of distances and sizes between the planets. If we used actual values, our transformations and vertex coordinates would be using values in the billions, which is completely unnecessary. Finally, these normalised values are multiplied by a scalar s, which means our values will now range between 0 and s. Now clearly with a full set of planets Jupiter will have the largest size and Neptune is the furthest away, but this hasn't been hard coded as say we (theoretically) could have an animation with just the 4 inner planets instead, which would end up normalising against the Earth and Mars respectively.

These are the key points of the class `SolarSystem` which governs the behavior of our planets, but now we need to replicate this model graphically, like we see in figure 1.

Figure 2: The Moon and an artificial satellite orbiting the Earth

The Graphical Model

Object Movement

The core concept of the graphics used to represent this model is that on each frame, each object's distance and angle in respect to the object it's orbiting is used to calculate the exact point it should be in our scene.

The Sun is always placed at position $(0, 0)$. So for each `Planet` in our system, we do the following -

1. Compute the planets polar coordinate in respect to the x-axis
2. Compute the planets polar coordinate in respect to the z-axis
3. Rotate the planet based on it's current rotation

This means each planet has can be located at the point $(d \cos p, 0, d \sin p)$, where d is the planet's distance from the origin and p is the planet's current rotation around the sun. We can then use these coordinates to translate our object to the desired position. We also add the radius of the star into the distance to the sun, to prevent any objects being created inside it.

```
float planetX = (planets.at(i).getDistanceFromSun() +
starSize)*sin(planets.at(i).getPosition());
float planetZ = (planets.at(i).getDistanceFromSun() +
starSize)*cos(planets.at(i).getPosition());
glTranslatef(planetX, 0., planetZ);
glRotatef(planets.at(i).getRotation(), 0, 1., 0.);
glRotatef(90., 1., 0., 0.);
```

We use this exact same method for a planet's moons/ satellites, but we compute the polar coordinates in respect to the planet it orbits, as opposed to the origin. To position the satellite, we first translate to it's planet position, and then translate by the satellite's coordinates. Also, for satellites, we add double the radius of itself to the distance to space out them out more, else they look very squished together in their orbits.

```
float satilliteX = (currentSatillite->getDistanceFromPlanet() +
currentSatillite->getRadius()*2)*cos(currentSatillite->getPosition());
float satilliteZ = (currentSatillite->getDistanceFromPlanet() +
currentSatillite->getRadius()*2)*sin(currentSatillite->getPosition());
glTranslatef(satilliteX, 0., satilliteZ);
glTranslatef(planetX, 0., planetZ);
```

The use of polar coordinates for object positioning makes features we will talk about later on (such as camera movement) significantly easier to implement, and reduces the amount of transformations we need to perform later on.

It should be noted that we only need to compute the pair of polar coordinates as our system lies on a single plane i.e. $y = 0$.

Lighting

The lighting is relatively simple in our model, as the only light source within our system is the Sun. Thus, we position a point light at the origin, and place the sun over the top of this. We then give the Sun a high emission factor, which gives the appearance that it is a source of light. Given the nature of our model, lighting was always going to be simple.

All planets in this model were given a shininess of 0.5, which reflects (no pun intended) how much light bounces off them from the sun.

The moons that didn't have textures were the ones that more lighting parameters set. None of the moons had a specular option set, as in reality none of them produce a specular affect. Instead a variety of different ambient and diffuse options were set, depending on the colour of the object in question.

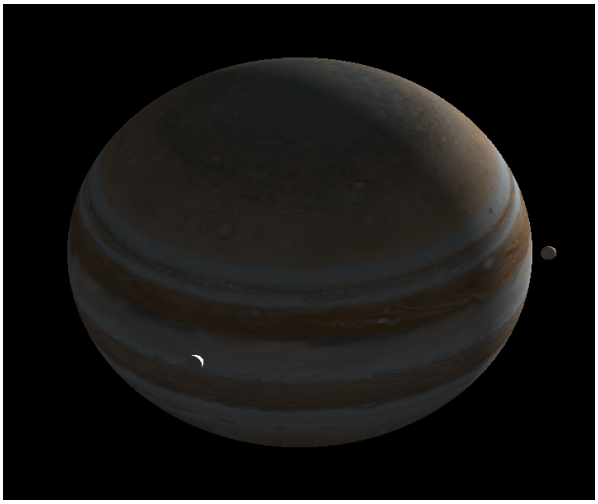

Figure 3: Jupiter and two of it's moons; we can see that they are not textured but have different lighting settings instead.

The artificial satellite object also has some lighting options set, but we will discuss that later.

Texturing

The texturing in our model is what brings it to life. Each `Planet` object is given it's own texture, and this is implemented by having the object store the relevant texture unit, which we can get at the time of generating the scene to select the correct texture. We also give the Sun and Earth's Moon a texture, because we consider them fairly

important objects in our system. We can see the textures used in this model throughout the report.

The textures used in this animation were downloaded from the website ‘The Solar Scope’ (https://www.solarsystemscope.com/textures/).

Camera Movement

In this model, the user is able to interact with the model by moving the camera around. There are various ways for the user to do this (mainwindow.cpp::57), which are -

- Q Key - move camera upwards
- E Key - move camera downwards
- W Key - move camera forwards
- S Key - move camera backwards
- A Key - move camera left
- D Key - move camera right
- Z Key - increase zoom
- X Key - decrease zoom
- Up Key - rotate camera around z-axis clockwise
- Down Key - rotate camera around z-axis counter-clockwise
- Left Key - rotate around y-axis clockwise
- Right Key - rotate around y-axis counter-clockwise

These operations are implemented as transformations to the model refer than changes to the View matrix. The movement of the camera was implemented as translations to our model, where the model had to be translated in the opposite direction to the intended camera movement to simulate such movement e.g. to simulate our camera moving up the model will be translated in the negative y direction (viewwidget.cpp::139). The same principle applies to our rotations (viewwidget.cpp::175).

The zoom didn’t require this. If the user wants to increase the zoom, we can just scale the model directly (viewwidget.cpp::225).

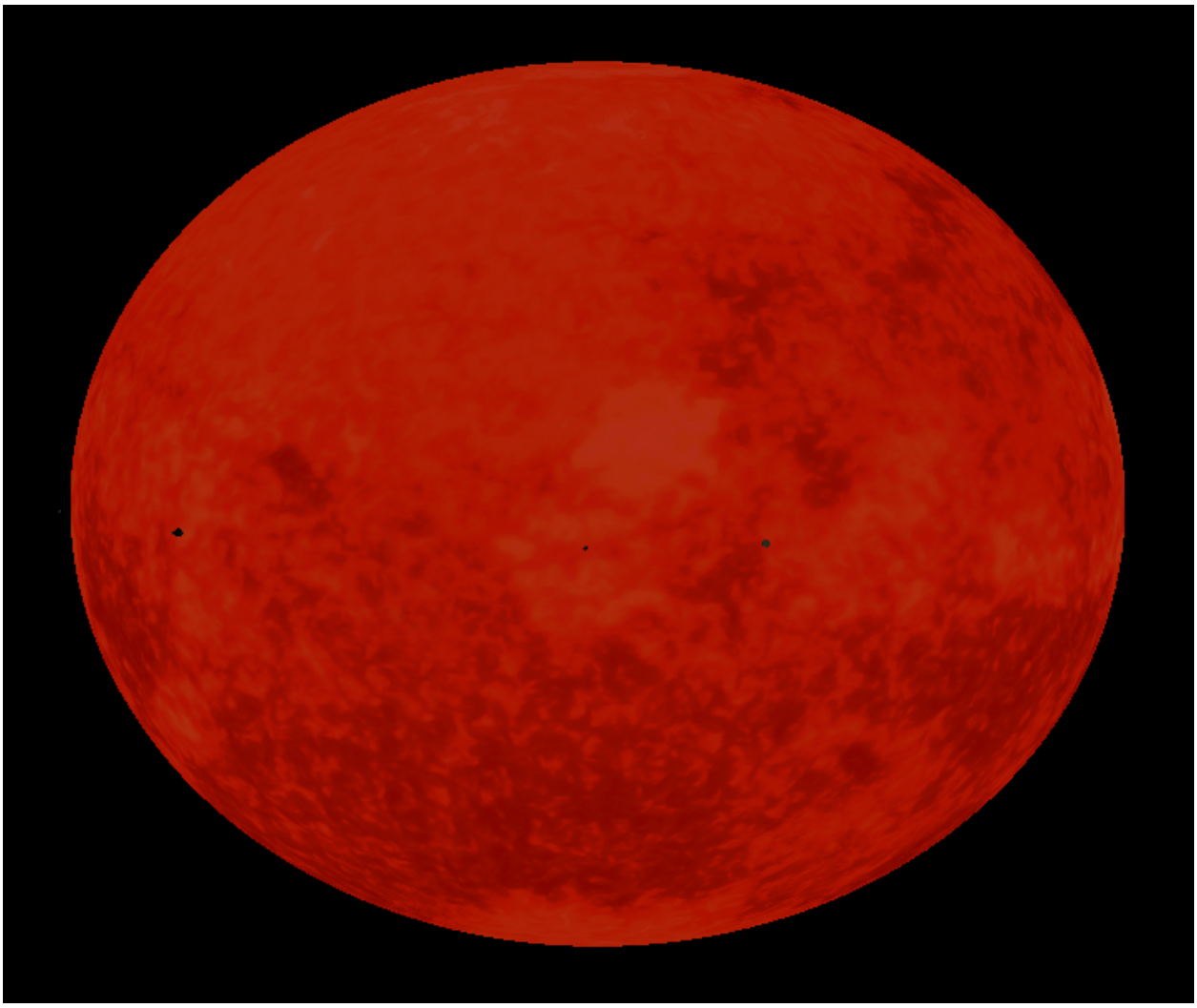

Figure 4: We can see an altered camera angle with zoom pointing towards the Sun; we can see the planets (from left to right): Earth, Mars, and Venus

Model Construction

There are two models of interest within this application. This does not include spheres, because they have been constructed using `glutSphere`. The two interesting models is the hemisphere and the artificial satellite (our hierarchical model).

Hemisphere

The creation of the hemisphere is very intuitive and simple. It utilizes the stacks and slices method for creating a sphere, but instead of creating the entire sphere, it only generates half of it. This is done by only generating half the stacks, and returning from the function when we are half way through the ‘stack-loop’, thus, creating a hemisphere (see figure 5).

The method to create a hemisphere is in `ViewWidget::satilliteDish` (viewwidget.cpp::323).

Artificial Satellite

The artificial satellite is a generated by `ViewWidget` and generates a satellite model, and is located in the method `artificialSatellite` (viewwidget.cpp::366). The satellite is our hierarchical model, and is created in the following steps -

- Rotate our axis 90 degrees - this gives the affect of the satellite being on it's side.
- Create a satellite body - this is a simple sphere.
- Create the first set of arms - one arm is translated to the top of the sphere whilst the other is translated towards the bottom. The arms are cylinders created with `glutCylinder`.
- Create the hemispheres - these are the dishes of the satellite, and are translated to either end of the satellite arms.
- Create the second set of arms - same as the the previous set, but they are translated left and right, both still created with `glutCylinder`.
- Create the panels - the satellite panels are simply planes, which are rotated and translated to be placed at the end of the second set of arms.

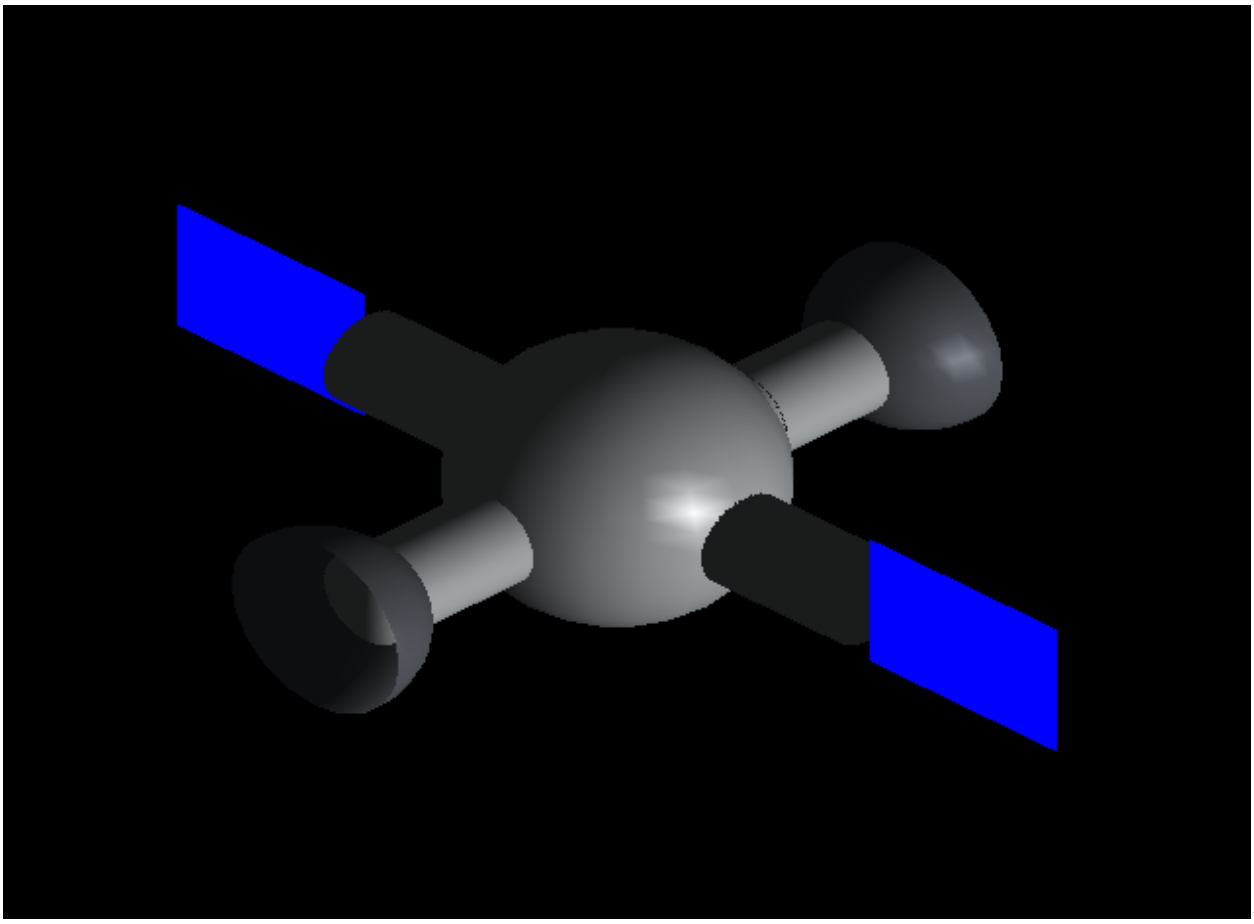

Figure 5: The artificial satellite

It should be noted that on step, the Model/View matrix was pushed at the start of the step and popped at the end of the step, which is a key component of hierarchical modelling.

The satellite body (sphere) and the arms are given a titanium look under our lighting system, whilst the dishes have a steel look. The solar panels are blue, because solar panels are generally blue.

These satellites appear orbiting several of the planets.

Interactive Features

Excluding the previously mentioned camera movement features, there are also some other interactive elements to our scene. See figure 1 for the mentions UI elements.

Firstly, the user has the ability to adjust the speed at which the simulation runs. There exists a slider which the user can drag to increase the speed at which the objects orbit

and rotate (viewwidget.cpp::199). This slider will affect the `speed` parameter that is passed into `SolarSystem::tick`. Interestingly, This parameter is initially set to 0.0005, as higher values cause the inner planets to orbit so quickly that you can't see them.

Next, we have an elusive checkbox which says 'Marc?'. When you click this checkbox, the following happens to the sun -

Figure 6: Marc De Kamps if he were the sun

This was really a matter of swapping textures, and in truth, became part of the program accidentally when this happened trying to get textures to work (viewwidget.cpp::243).

Then, we have the Reset Camera Position button, which allows the user to reset the camera position, which is of course technically the model position, back to where it is at

start up. This is done by setting all the translation and rotation values to 0, and setting the scale to 1, and placing the camera focus back on the Sun (viewwidget.cpp::249).

Finally, we have the camera focus. The user is able to select an object to focus the camera on, to which that object is then placed into the center of the screen. This feature was made very easy with the use of polar coordinates, which allows the model to be translated in the negative direction of the equivalent coordinates, placing the planet's position to the center (viewwidget.cpp::475). We also reset the camera positions to ensure that the camera does indeed focus on the correct planet.

```
if(this->cameraFocus > -1)
    {
        Planet focusPlanet = planets.at(this->cameraFocus);
        float planetX = (planets.at(this->cameraFocus).getDistanceFromSun() +
 starSize)*sin(planets.at(this->cameraFocus).getPosition());
        float planetZ = (planets.at(this->cameraFocus).getDistanceFromSun() +
starSize)*cos(planets.at(this->cameraFocus).getPosition());
        glScalef(this->zoom, this->zoom, this->zoom);
        glTranslatef(-planetX, 0., -planetZ);
        this->movementX = 0.;
        this->movementZ = 0.;
        this->rotationY = 0.;
    }
```

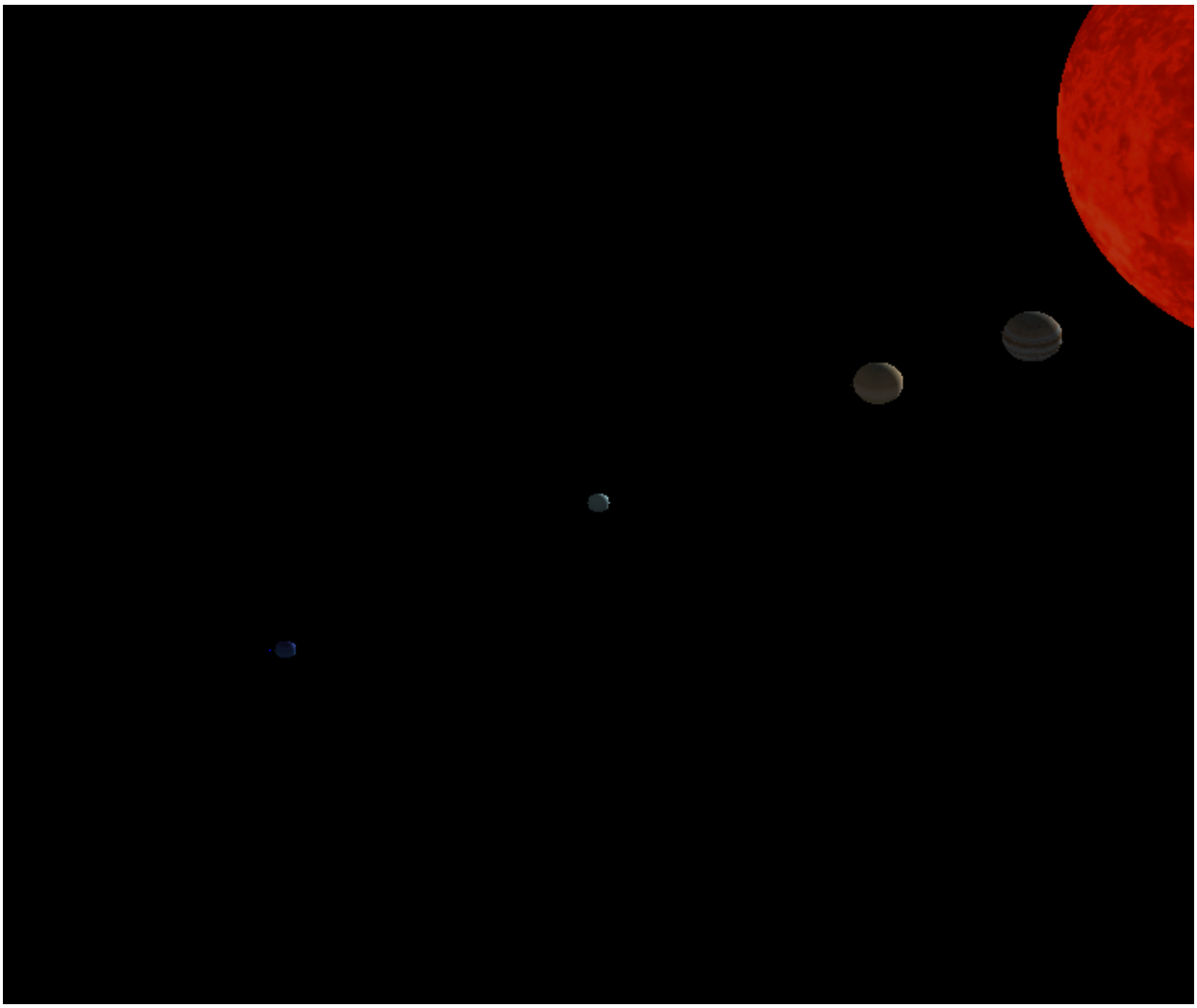

Figure 7: Uranus as the camera focus