COMP 510

Term Project Report

Solar System Simulation using OpenGL

Team Members

Najeeb Ahmad
Muhammad Aditya Sasongko
Burak Bastem

Koç University, Istanbul, Turkey

Introduction

The solar system is a group of astronomical objects and comprises of a star called Sun and objects that orbit it. While there are many types of objects revolving around the Sun including planets, dwarf bodies and small solar system bodies [1], the significant eight bodies are planets while the rest are significantly small. Of these planets, there are certain planets who are further being revolved around by objects known as satellites, each known by a specific name as will be discussed later.

The eight planets revolving around the Sun in the order of increasing distance from the Sun are Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus and Neptune. Of these planets, Jupiter is the largest in size followed by Saturn, Uranus, Neptune, Earth, Venus, Mars and Mercury in that order. Some of these planets have satellites revolving around them. These planets include Earth, Mars, Jupiter, Saturn, Uranus and Neptune.

Sun being the only star in the solar system is the source of light in the solar system. Of all the planets in the solar system, Earth is the only planet to have life on it. Figure 1 shows a pictorial depiction of the solar system.

Figure 1: Our Solar System

Project Overview

The aim of this project is to apply the knowledge learnt in the computer graphics course (COMP 510) to simulate the solar system by graphically depicting the Sun, the eight major planets revolving around the Sun and the satellites for some of the planets. The simulation shows the

Sun being revolved by its major planets while some of the planets being revolved around by their satellites. Also, it shows the objects rotating around their own axes as per actual Solar system. The object sizes and distances have been chosen to be proportional to the actual sizes and distances in the Solar system. The simulation allows the users to navigate around the solar system through keyboard keys and pick objects to get information about them.

Project Motivation

The purpose of choosing this project was to enable ourselves to allow most of the techniques learnt in the computer graphics including object mesh generation, object instantiation, object translation, rotation and resizing, lighting, shading and texture mapping, object picking, view transformations, modelview manipulations etc.

Project Scope

The scope of this project is to simulate the Sun, the eight large planets revolving around the Sun namely Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus and Neptune, the satellite around the Earth i.e. the Moon and the Jupiter satellites namely Ganymede, Callisto, Io and Europa. Each of this objects is to be graphically drawn in proportion to their actual sizes and distances in the actual the Solar system. However, due to graphical restrictions like too large view volume and too small object size when drawn in proportion, not all the objects have been drawn in proportion. The Sun is assumed as the main point light source and Phong shading and reflection model are used for shading the planets and their satellites. Picking mechanism is used to pick objects to view their information on the terminal. Also, user can navigate around using keyboard keys.
More detailed specifications are listed in the following section.

Project Specifications

The simulation of the solar system incorporates some of the concepts taught in the class. These concepts are as follows.

- Transformation (rotation, translation, and scaling)
  Each object rotates around its own axis while all of the objects aside from the Sun revolve around the Sun. The simulated satellites revolve around their planets. The simulated satellites include one satellite for Earth and four satellites for Jupiter.
- Point source of light
  Sun act as the main point source of light
- Texture mapping
  Actual bitmaps of the planets and satellites (for the relevant planets) is used to create the texture on the objects.

- Shading
  All the objects, apart from the Sun, are subject to Phong shading and reflection model.
- View and projection
  User is able to move in the simulation and change his/her view angle. As the user moves towards or away from objects, their sizes change.
- Picking mechanism
  When each object is clicked by the mouse, the name of it appears in the terminal. This makes our simulation educational, as user is able to learn the locations and the names of some relatively unpopular planets and satellites in the Solar system.
- Simulation Speed
  User can use keyboard keys to increase/decrease the speed of the simulation.

Implementation Details

Setting up the Skeleton

In the simulation, we use hierarchical modeling instead of linear modeling. We have a class, *AstronomicalObject*, in which information about an astronomical object like equatorial radius, rotation period, average orbit distance, orbit period etc. and pointers to its orbiting objects are kept. The main object in the Solar System is Sun and all the planets orbit Sun, so Sun is an instance of *AstronomicalObject* class and it has pointers to instances created for each planet. If a planet has satellites, its has pointers to satellites in its instance. Figure 2 shows the hierarchical model in our simulation.

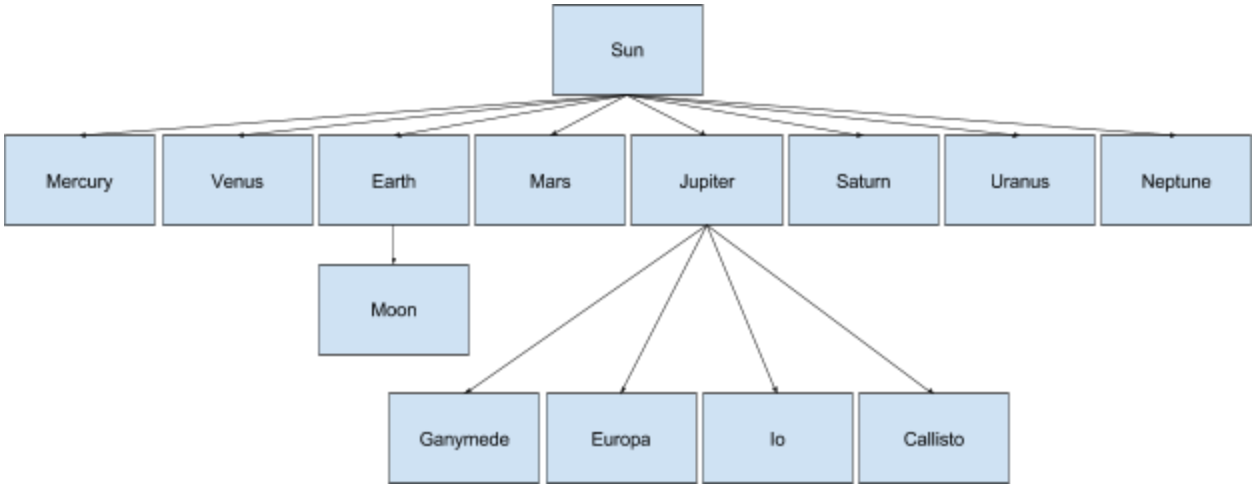


Figure 2: Hierarchical model in the simulation

By referring to Figure 2, an object A which is connected by an edge to another object B and is located below object B hierarchically is referred to as the *child object* of object B. On the other hand, object A in this case is referred to as the *parent object* of object B. An example of this is that Jupiter is a child object of sun and sun is the parent object of Jupiter.

In the simulation, we have one sphere, taken from [2], and all objects (sun, planets and planets) are drawn with this sphere. Based on the information of an object, we produce a model matrix. Then, we produce a view matrix with *LookAt* function described in [2]. This matrices are sent to vertex shader and are multiplied with each vertex of the sphere and a projection matrix to draw each astronomical object. At the beginning, view and projection matrices were identity matrices. Based on this model, we implemented the complete simulation.

Adding Texture

Once the objects in the solar systems have been created as per their size and distance relationships, the next step is to perform textures on the objects. This amounts to giving the Sun, planets and satellites the same appearance as in the actual Solar system.
Texture mapping is a process of mapping a 2D image on the 3D object, which in this case is a sphere. Generally, in texture mapping, three to four coordinate systems are involved.
This includes Parameter coordinates those are used to model curved surfaces, texture coordinates those are used to identify points in the image to be mapped, world coordinates where the mapping actually takes place and finally the screen coordinates where is the image is actually produced.
In texture mapping, the basic problem is to identify which point on the texture image corresponds to which point on the actual image. In actual practice, we follow a backward mapping in which we try to identify which point on the object correspond to which point on the image. For instance, say the object is defined in a 3D space in Cartesian coordinates (x,y,z) and the image is defined in s and t coordinates, then the map we need to find is of the form

$$s = s(x,y,z)$$
$$t = t(x,y,z)$$

Finding such functions is not a trivial task and a graphic programmer would rather prefer that such points are explicitly provided. In case such maps are not explicitly given, there are solutions like two-part mapping in which texture is first mapped to an intermediate surface before being mapped to the actual surface like for instance a cylinder or a sphere, box mapping etc. [2]

The three basic steps in applying texture to an object are given below:

1. Specifying texture by reading an input image or creating texture programmatically.
2. Assigning texture coordinates to vertices
3. Specifying texture parameters

Texture mapping for this project as per steps identified above is listed in the following sections.

Reading texture images

One of the basic tasks for assigning textures is to read textures from image files. First step in this regard is the selection of images. For this project, the images have been borrowed from Planet Texture Map Collection from [3] which is free repository of planet texture maps. All of the textures used in this project are in JPEG format.
To read the images in the our CPP program, an open source library for image manipulation known as FreeImage [4] has been used in this project. The library provides functions to read JPEG images and load them into arrays. It supports various formats including JPEG, PNG, BMP, TIFF etc. For this purpose, a class CTexture has been defined that after loading the texture, binds them to texture objects for each object of the solar system.

Calculating and Assigning Texture Coordinates

In solar system, all the objects can be approximated as spheres (although some are not perfectly spherical). For the sake of this simulation, it has been assumed that all the objects are spheres. Hence the problem becomes that of mapping textures to vertices of spherical objects of various sizes. To map the texture image points to object vertices, following equations have been used

$$s = \frac{\tan^{-1}\left(\frac{x}{z}\right)}{2\pi} + 0.5$$

$$t = \frac{\sin^{-1}(x)}{\pi} + 0.5$$

Where x and z are x and z values of normals. To make the mapping smooth, the following corrective statements have been added after calculation of the above two equations [5].

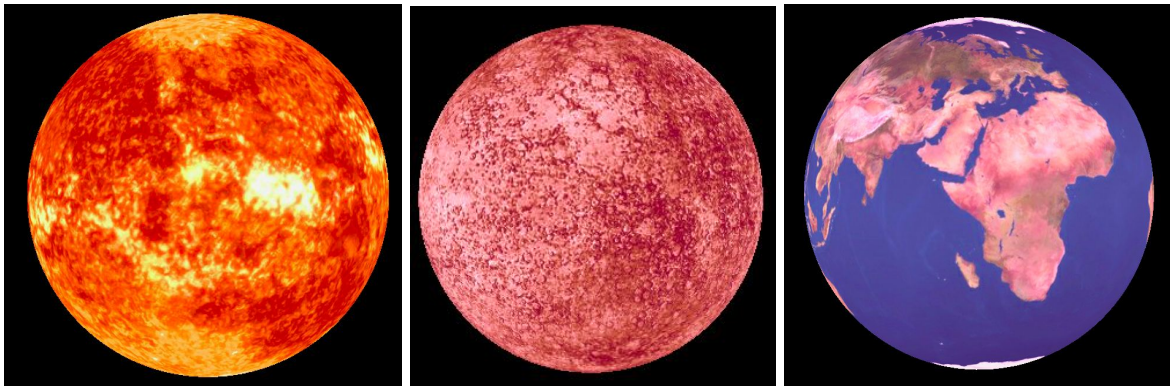
```
if(s < 0.75  && s_prev > 0.75)
                    s+=1.0
if(t > 0.75 && t_prev < 0.75)
                    s-=1.0
```

Once the texture coordinates are calculated for each vertex, they are sent to GPU and appropriate variable is mapped from application to the shader.

Specifying Texture Parameters

After loading the image, a texture object is created for each object in the Solar System. The texture object is created as part of the AstronomicalObject class for each solar system object. In this way, each astronomical object has texture object corresponding to its loaded image.
After binding the texture object to the astronomical object, texture parameters are specified.

For this purpose, both texture magnification and minification parameters are set to GL_LINEAR_MIPMAP_LINEAR. This will help to solve both magnification and minification problem as mipmapping stores textures maps of decreasing resolutions.



The Sun The Mercury The Earth

Figure 3: Texture mapping for Sun, Mercury and Earth from the simulation

Object Transformations and Rotations

In order to guide the transformation of an object in simulation and its positioning with respect to other objects in the simulation, several attributes of that object which are declared in the *AstronomicalObject* class must be set. These attributes are “equatorial_radius”, “rotation_period”, “RotationTheta”, “TiltingAngle”, “average_orbit_distance”, “orbit_period” and “RevolutionTheta”.

“equatorial_radius” attribute defines the size of an object in proportion to the size of the sun in the simulation. This value is used in the scaling transformation of each object in the simulation. Scaling is the first transformation applied to every object instance in the simulation.

“rotation_period” attribute defines the speed at which an object revolves around its axis. The second transformation that is applied to every single object in the simulation is to rotate it around the Y-axis of the origin. The angular speed of the rotation is determined by the value of the “rotation_period” attribute of the object which is the rotation period of the real planet or satellite normalized by the revolution period of earth. It means that the value of “orbit_period” attribute of earth is set to 1 in the simulation, and the time-related attributes of all objects in the simulation, i.e. “rotation_period” and “orbit_period” will be set by using earth’s “orbit_period” as the unit of time.

“RotationTheta” attribute directs the rotation transformation more directly by being used in the matrix multiplication of object’s model. It defines the angle of rotation when the object is

rendered. Its value is updated in idle callback function by making use of the rotation_period value in order to change the rotation angles between rendering.

The next attribute to be used in transformation is “TiltingAngle”. This attribute defines the tilting formed by the planet with respect to the world coordinate. The use of this attribute in transformation causes the sun, planets or satellites not to rotate exactly around its own y-axis. This reflects real-life examples in which space objects don’t exactly rotate around their “vertical” axes.

After “TiltingAngle”, the next attribute to be used is “average_orbit_distance”. This attribute defines how far an object is from the other object that it revolves around. Since sun is the root of the tree illustrated in Figure 1, it will be placed in the origin of the world coordinate in the simulation, and the “average_orbit_distance” attribute of a planet will define how far the planet object is from the origin of the world coordinate. On the other hand, for a satellite object, this attribute determines how far its distance is from the planet that it revolves around.

The next attribute to be used for transformation is “orbit_period” which defines the speed at which an object revolves around its parent object. Similar like “rotation_period”, the value in this attribute is the real life value normalized by the revolution period of earth.

“RevolutionTheta” attribute influences the revolution transformation more directly by being used in the matrix multiplication of object’s model. It determines the angle of rotation around origin/revolution when the object is rendered. Its value is updated in idle callback function by making use of the revolution_period value in order to change the rotation angles between rendering.

By taking all of the attributes which are mentioned above into account, the sequence of transformation for each object becomes to scale it by using “equatorial_radius” value, to rotate it around its y-axis by using “rotation_period” and “RotationTheta” values, to tilt it with respect to world coordinate by using “TiltingAngle” value, to translate it from the position of its parent object by using “average_orbit_distance” value, and finally, to rotate it around the origin by using “orbit_period” and “RevolutionTheta” values. For satellite objects, after applying this sequence, two last transformations that are applied to its parent object, i.e. translation and then rotation, will also be applied to it, since the point of reference of its movement is its parent object.

Reflection and Shading

For the shading process in the simulation, a single point of light source is considered which is the origin of the world coordinate, i.e the location of the sun. This parameter of light source position will be used in the rendering of all planets and satellites. Phong shading combined with Phong reflection is used in creating shading effects on the objects. The shading will ensure that the part of each object that looks shined by light will be that which faces the direction of the sun.

An example of this is illustrated in Figure 4 where parts of Mercury, Venus, earth, and moon which look bright are those facing the sun.

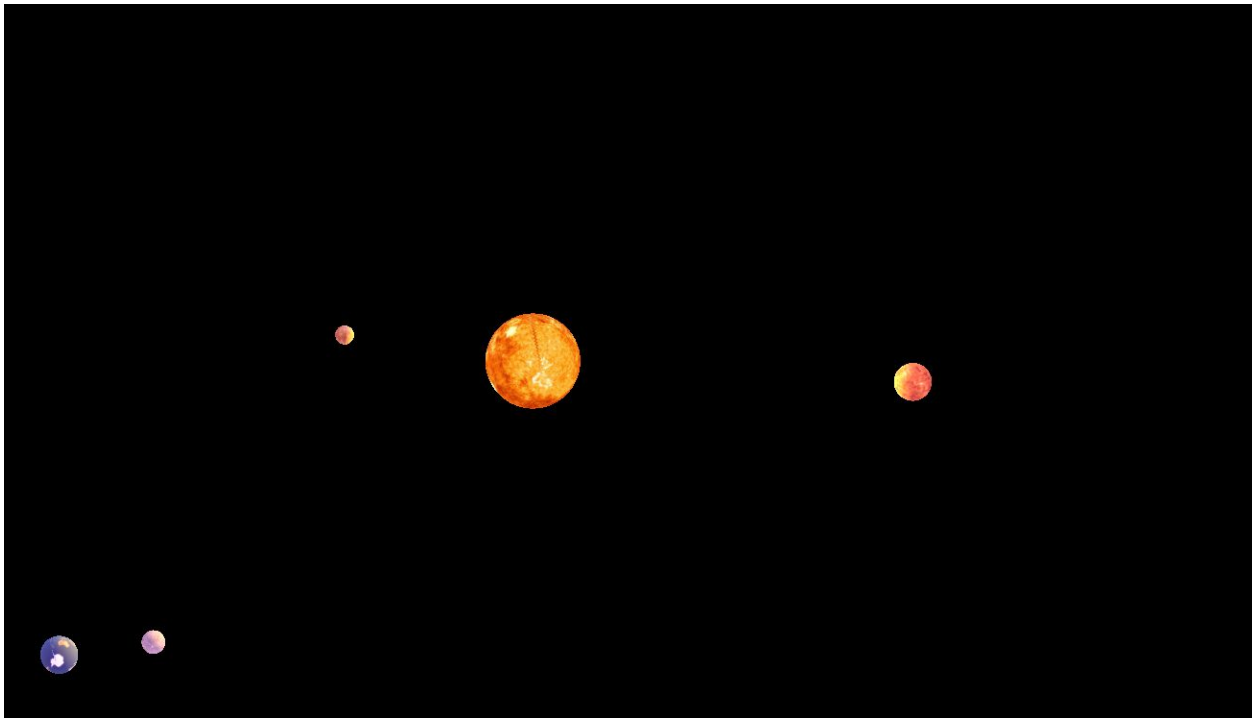


Figure 4: Shading effects

Picking

Picking mechanism is also implemented on each object in the simulation. This mechanism enables user to get the name of the object which he/she clicks on.

This mechanism is implemented by using mouse click event callback function. Every time this function is triggered by mouse click event, each object will be rendered briefly (only once) with a single color, instead of with texture and shading. The RGB value of each object will be made different, so that when the RGB value of a clicked object is read with glReadPixels function, that value will be associated only with that particular object. By using this acquired value, an object will be identified and its name will be printed on the standard output, i.e. terminal.

Navigating through the Simulation

User navigates through the simulation with w, a, s, d. He/she moves forward with w, backward with s, left with a and right with d. Viewing and projection is used to implement navigation. Ortho(left, right, bottom, top, zNear, zFar) function from [2] generates the projection matrix. User adjusts a coefficient used in generating Ortho function parameters with w and s keys. Hence, he/she goes forward or backward in the simulation. LookAt function from [2] generates the view

matrix. It takes *eye, at* and *up* as parameters. *eye* is a point and indicates the position of camera. *at* is also a point and indicates the desired position. *up* is a vector and indicates up direction for the camera. Figure 5 shows the look-at positioning. A translation matrix is multiplied with view matrix to change the camera position. a and d is used to get a value for translation, which enables user to go right or left in the simulation.

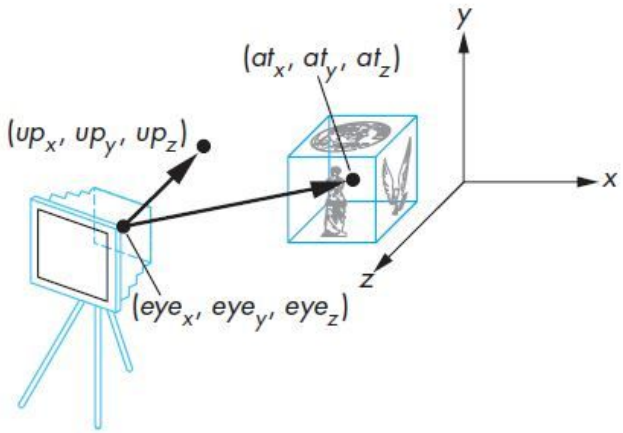


Figure 5: Look-at positioning [2]

User can also change view angles in the simulation. View matrix is again used to change the view angle. Followings are the parameters we give to LookAt function.

eye(-sin(theta)*cos(phi), sin(phi), cos(theta)*cos(phi), 1.0)
at(0.0, 0.0, 0.0, 1.0)
up(0.0, 1.0, 0.0, 0.0)

eye point is generated based on theta and phi angles which are increased or decreased with arrow keys, enabling user to change his/her view angles.

Simulation Speed Control

A global variable, namely “EARTH_REVOLUTION_ANGULAR_SPEED” is used in controlling the speed of simulated rotations and revolutions of all objects. This variable is used by making it the numerator and making the “orbit_period” or “rotation_period” value the denominator in the calculation of revolution or rotation angle of each object for each rendering. By this way, the larger the revolution or rotation period of an object is, the smaller the angle that will result from the calculation. Therefore, that object will look slower than other objects with greater period of revolution or rotation. On the other hand, the magnitude of rotation and revolution angles resulting from the calculation will be proportional to the value of “EARTH_REVOLUTION_ANGULAR_SPEED” global variable. The greater the value of this variable is, the greater the angles become, and vice versa.

Summary/Conclusion

In this project, we implemented Solar System with OpenGL. We applied most of the techniques we learned in the course. The simulation contains object mesh generation, object instantiation, object translation, rotation and resizing, lighting, shading and texture mapping, object picking, viewing, projection and hierarchical modeling.

References

[1] Wikipedia, "Solar System," 28 May 2017. [Online]. Available: https://en.wikipedia.org/wiki/Solar_System.
[2] E. Angel and S. Dave, Interactive Computer Graphics: A top-down approach with shader-based OpenGL, 6th Edition, Addison-Wesley, 2012.
[3] J. Hastings, "JHTs Planetary Pixel Emporium," 2017. [Online]. Available: http://planetpixelemporium.com/planets.html.
[4] "Free Image, The productivity booster," 2017. [Online]. Available: http://freeimage.sourceforge.net/.
[5] M. S. U. webpage, "Texture Mapping," [Online]. Available: https://www.cse.msu.edu/~cse872/tutorial4.html