Solar System Simulation

Alexander Chuckas
Rensselaer Polytechnic Institute

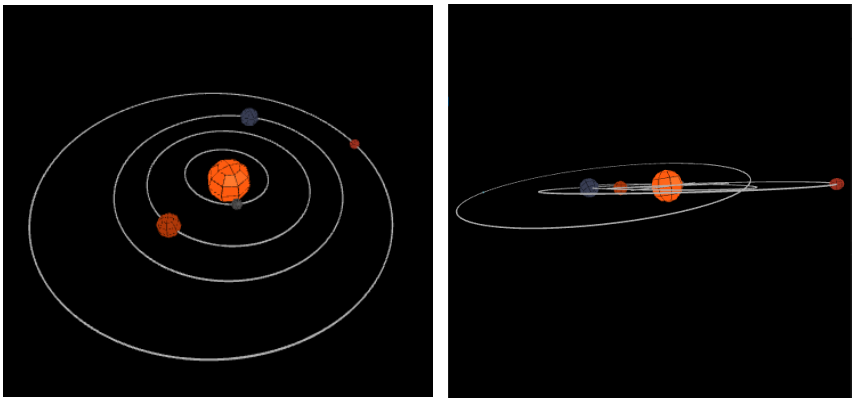

Figure 1: *Left is a top view rendering of the first 4 planets in the solar system and their orbits around the sun. Right is the first 4 planets, the sun, and the comet Encke from a side view to show extreme orbits.*

ABSTRACT

In this paper, I describe a method for simulating a solar system of unlimited planetary bodies (system hardware determined). The methodology is based on real physics with orbit paths and day-night cycles for simulated masses based on real-life planetary data and orbital parameters. The blunt of the work is done via physical equations and matrix transformations on isolated planetary meshes.

1. INTRODUCTION

My inspiration for this project comes directly from an interest in space phenomenon since I was a kid and from the iOS application "Night Sky". "Night Sky" is an app which effectively acts as a planetarium in your pocket, showing an accurate map of various stars, constellations, and planets. The principal behind the geolocation of each of these celestial bodies over time is effectively the same as what my project is trying to accomplish. Ultimately, this project was born out of a curiosity of our solar system.

My project builds off the base library provided in homework 3, namely the spherical mesh generation. Initially I was using a much more complex spherical mesh, but manipulating its location was much more complicated than need be which led me back to homework 3. The homework 3 library also provided code for simulating line paths with OpenGL which allowed for orbit path visuals to be added.

I will cover related work in section 2 and then the core features of my project in section 3. Discussion of the results can be found in section 4. Limitations & bugs, and future work can be found in sections 5 and 6 respectively.

2. RELATED WORK

The topic of this paper is not new at least in the sense of the physical calculations for planetary positions. However, the actual implementation of the algorithms in a real-time rendering software is what this project tries to accomplish. Prior work in this field most generally comes from the information provided by NASA's website regarding orbital parameters. Each of these parameters taken directly from the website are what make up the key components of the my .obj files, allowing me to render the solar system. Other prior work involves known matrix transformation formulae and research into general forms of the Rodrigues formula. Ultimately, this project is an implementation of worldly phenomenon based on real-world data.

3. CORE FEATURES

Prior to delving into the individual sections, a few terms need to be defined:

- **Periapsis**: the point in orbit when the planet is closest to the body it orbits (in our case the sun)
- **Ecliptic:** the plane of the Earth's orbit
- **Eccentric Anomaly:** angular position within the reference orbit.
- **Mean Anomaly:** the fraction of an elliptical orbit's period which has been completed.

- **True Anomaly:** major angular parameter in Keplerian orbit. Angle between direction of periapsis and current position of body.

3.1 Planet Mesh Models

The planet mesh models were heavily based on Professor Cutler's provided homework 3 code, namely the sphere.cpp and sphere.h classes. The generic sphere classes provided code for generating a simplistic, spherical mesh based around a central location provided in a .obj file. The original input parameters for the sphere class were:

s x y z r

where *s* indicates a sphere, *x, y, z* is a representation for a Vec3f point location for the center of the mesh, and *r* is the radius of the sphere. The meshes were generated with few vertices. I chose to use a simple mesh since using a complex mesh would have significantly slowed mesh updates.

While this class provided a basic setup, it would need to be expanded for use. A number of new input parameters were needed. The below table gives a general format for a planet within the .obj file with the new parameters.

m	num						
p	x	y	z	r	cycle	orbit	tilt
N	N	u					
i	i	u					
w	w	u					
a	a	u					
e	e	u					
M	M						

Table 1: *General structure of a .obj input file*

The above table represents the general format for inserting a planet into the solar system. The leftmost column in the table represents a token indicating to the program a certain element is being read in. All other columns are values associated with that token. The first row is the material to be wrapped around the sphere (a texture as a .ppm file or a solid color). The second row indicates this object is a planet (similar to sphere but now we use a *p*) and once again has the central point vector formed form *x, y, z* and the radius *r*. The new parameters are the cycle, orbit, and tilt. Cycle is represented in hours and is the time it takes a planet to complete a full day-night cycle. Orbit is the time in days it takes a planet to complete one revolution around a central body. Tilt is represented in degrees and is the axis the planet spins on. All other rows are the orbital parameters which effect the shape and orientation of a body's orbit.

3.2 Elliptical Orbits

This subsection is broken down into three parts. The shape section discusses how to create the elliptical shape of the orbit and the orientation section discusses how the orbit path is oriented. The final section discusses implementation. The orbit paths of the planets are determined by 6 orbital elements: Longitude of the Ascending Node (*N)*, Inclination (*i)*, Argument of perihelion (*w)*, semi-major axis (*a)*, eccentricity (*e)*, and mean anomaly *(M)*.

3.2.1 Shape

The orbital elements involved in the shape of the orbit are the semi-major axis *a* and the eccentricity *e* of the orbit . The semi-major axis is a general parameter used in creating an ellipse and is the longest semidiameter (ie. given the 2 diameters of an ellipse, the semi-major is the larger one). Effectively, it is the largest of 2 axial radii. Eccentricity is a non-negative real number that uniquely characterizes the orbit's shape. The semi-major axis and semi-minor axis length are related through the eccentricity which can be broken down into a few value ranges where *e*=0 represents a circular orbit, *e*=(0,1) is elliptical, *e*=1 is parabolic, and e > 1 is hyperbolic. In order to fully see the orbit, eccentricity is limited to range [0,1].

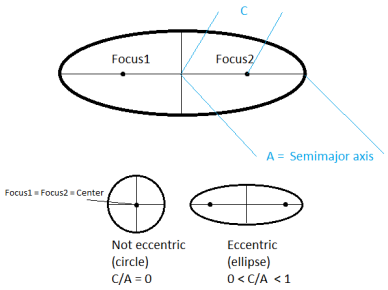

Figure 2: *Representation of an ellipse showing eccentricity and the semimajor axis*

3.2.2 Orientation

The orbital elements involved in the orientation of the orbit are the inclination *i*, the longitude of the ascending node denoted as *N*, and the argument of periapsis denoted as *w*. The inclination is the angle between the plane of orbit (in our case the ecliptic) and the reference plane (the plane the celestial body is orbiting in). The longitude of the ascending node is the location where the orbit of our celestial body crosses the reference plan from below to above it. Effectively it is one of two intersection points between the reference plane and the ecliptic. The argument of periapsis is the angle between the ascending node and the periapsis (within the reference plane).

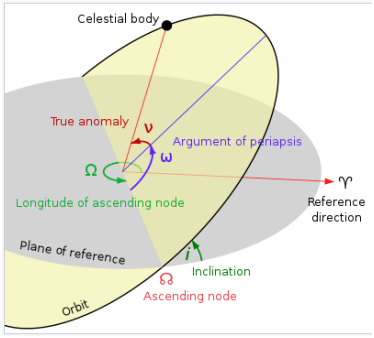

Figure 3: *Diagram showing all major angular parameters for elliptical orbits around a reference plane*

3.3.3 Implementation

Initially, my implementation for orbital positions was done in real-time causing my system to bottleneck and in addition would limit future expansion of the simulation to allow for orbit paths to be shown. As such, the implementation of elliptical orbit paths starts with the generation of a planet object. Each planet object is given the input parameter of orbit duration in years. The mean anomaly is thus determined as the full 360-degree orbit divided by the orbit duration. As an example, the mean anomaly constant value for earth is 360 / 365. The mean anomaly is then multiplied by the current day in the animation to get the location of the orbit. All angle based orbital elements (mean anomaly included) are reduced to be between 0 and 360 for viability for each iteration of an orbit location being generated.

Once the orbital elements are normalized to be between 0 and 360 degrees, we want to calculate 2 of the 3 major angular parameters that define a position along an elliptic Kepler orbit. The first of which, the eccentric anomaly, is defined iteratively. In some cases where the eccentricity of an orbit is large (near 1), the eccentric anomaly will need multiple iterations to converge at an adequate number. It is defined as follows:

```
E0 = M + (180_deg/pi) * e * sin(M)
     * (1 + e*cos(M))
while abs(E1 - E0) > 0.005:
    E = E1
    E1 = E0 -
         - (E0 - (180_deg/pi) * sin(E0) - M)
         / (1 - e*cos(E0))
```

The eccentric anomaly allows for the orbit's coordaintes to be extracted within its own reference plane of orbit. This means we still need to re-orient the plane to be based on the ecliptic orbit. A good next step is to transition to rectangular coordinates using the relationship between the eccentric anomaly and the rectangular x and y coordiantes in the reference plane:

```
x = a * (cos(E) - e)
y = a * sqrt(1 - e*e) * sin(E)
```

From here, we convert to distance and true anomaly (the second major angular parameter):

```
r = sqrt(x*x + y*x)
v = atan2(y, x)
```

Using the true anomaly, the orbit position in reference to the ecliptic plane is extracted:

```
L = v+w
xe = r * ( cos(N) * cos(L) - sin(N) * sin(L) * cos(i) )
ye = r * ( sin(N) * cos(L) - cos(N) * sin(L) * clos(i) )
ze  = r * sin(L) * sin(i)
```

These equations will provide the position of the celestial bodies in astronomical units. For purposes of demonstration I have provided a scale within the planet object class. This scale factor allows for all of the planets to be and in general provides a better visual effect. Once a position is determined it is stored for later use.

The positions previously calculated are referenced at every animation timestep where the day counter is incremented by 1. The centerpoint of the mesh is set to the new position and the vertices are realigned to the new center. If the realignment step is skipped, the planet will scatter based on the new center.

Figure 4**:** *A planet without realigned vertices deviating from its orbit*

3.3 Day-Night Cycles

Day night cycles for the planet meshes involve rotating the spherical planet object around an axis which is based on the planet's tilt. Fortunately, the matrix transformation for a spinning body around an arbitrary axis is something already derived and is known as the Rodrigues Formula. My initial exposure to the formula was in the first lecture of this class on and was brought up when discussing matrix transformations (namely rotation) for points in a 3d grid system. The Rodrigues Formula is as follows.

$$= \begin{vmatrix} \cos\theta + \omega_x^2(1-\cos\theta) & \omega_x\omega_y(1-\cos\theta) - \omega_z\sin\theta & \omega_y\sin\theta + \omega_x\omega_z(1-\cos\theta) \\ \omega_z\sin\theta + \omega_x\omega_y(1-\cos\theta) & \cos\theta + \omega_y^2(1-\cos\theta) & -\omega_x\sin\theta + \omega_y\omega_z(1-\cos\theta) \\ -\omega_y\sin\theta + \omega_x\omega_z(1-\cos\theta) & \omega_x\sin\theta + \omega_y\omega_z(1-\cos\theta) & \cos\theta + \omega_z^2(1-\cos\theta) \end{vmatrix}$$

Initially when implementing the Rodrigues Formula, I was multiplying by the wrong axis value in the center column, resulting in some of my planets looking like pancakes.

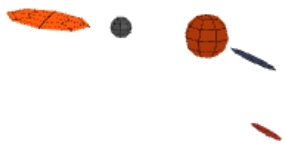

***Figure 5**: Deflated planets due to failed Rodrigues Formula*

I choose to increment the day-night cycles using the same counter as the day cycles for the orbit. This choice was made with the knowledge this would not accurately simulate the real day-night cycles, however, for the purposes of this project I wanted them to be visible. If day-night cycles were on a different parameter than orbit, either the orbit would have to be slowed down so much it was no longer apparent or the planet meshes would update so frequently no spinning would occur.

The actual implementation of Rodrigues was based off the input parameters of tilt and cycle. The axis of rotation was formed as

```
1   Vec3f u(sin(tilt), cos(tilt), 0);
```

From here a translation needed to be applied so that the planet would not orbit a central axis, but rather spin around an axis. The translation was based on the center of the planet mesh. With a translation and axis chosen, the Rodrigues formula was applied to all vertices.

3.4 Showing the Orbit Paths

The solar system would not be complete without a visualization of orbit paths. Initially, when I was trying to implement the paths, I was utilizing the planet position vectors and OpenGL rendering via GL_LINE_LOOP, however after countless hours of work, I deemed this methodology a failure and looked for other avenues of presenting orbit paths.

This is where the RayTree data structure comes in. The RayTree data structure allows for visualization of lines through meshes. As previously discussed, upon creation of a planet object, its orbit locations are stored in a 3d position vector. Upon pressing the 't' key these positions are transferred to the RayTree segment vector and subsequently, all segments (which are based on the collected 3d planet position vector) for each planet are displayed to the screen. Effectively a segment is a connection between 2 points creating a line which is drawn in the form of a 3d box.

4. RESULTS

I tested my simulation by inputting parameters from real-world celestial bodies. Initially, I performed tests on small systems such as the 4 closest planets (Mercury, Venus, Earth, Mars) and expanded to the popular comets (Encke, Halley). The general idea was to test their spins, orbit, and orbit paths and see if there were any errors. For small systems with small orbit paths, there was little to no bottleneck, but with larger orbits (such as Halley) the simulation would slow until the orbit paths were removed.

After both of these were working, I decided to implement the entire solar system including all 8 planets (Mercury, Venus, Earth, Mars, Jupiter, Saturn, Neptune, Uranus) and the 2 most comets I previously implemented (Encke, Halley). Naturally, the simulation would run slow when the orbit paths were showing, however, without orbit paths the simulation ran surprisingly fast, effectively displaying the solar systems changes over time. As such the primary goals for this project were met.

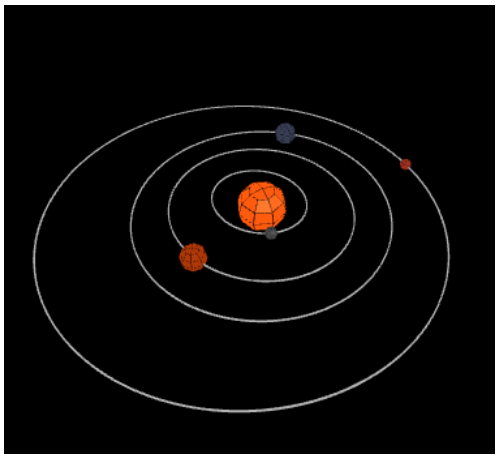

***Figure 6:** Rendering of 4 closest planets (Mercury, Venus, Earth, Mars) and their orbit paths*

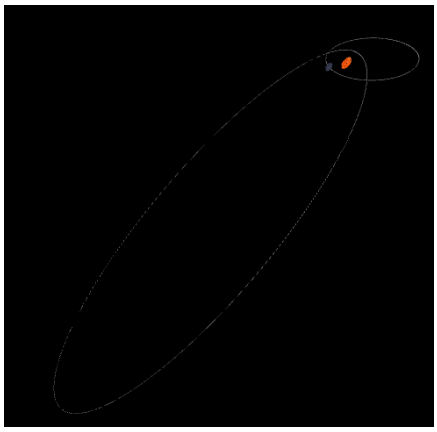

Figure 7: *Rendering of 2 comets (Encke, Halley) and their orbit paths*

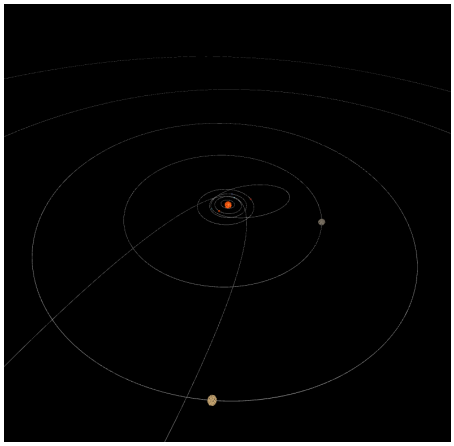

Figure 8: *Rendering of entire solar system and all planet and comet orbit paths (zoomed in)*

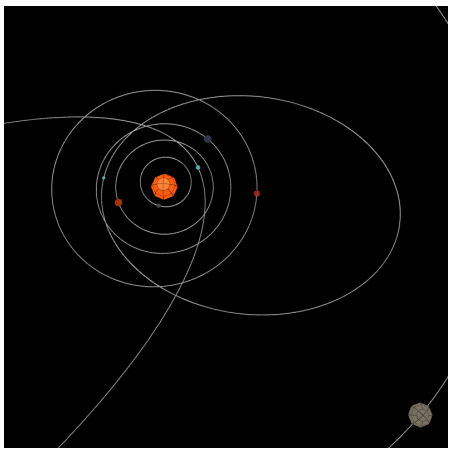

Figure 9: *Entire solar system top view (zoomed in)*

5. LIMITATIONS & BUGS

A more prominent limitation of my simulation involves the rendering of orbit paths. I noticed the issue when trying to render my entire solar system model which includes Halley's Comet, Jupiter, Neptune, and Saturn, all of which have rather large orbits spanning over a large time period. Due to the length of the orbit, in creating position vectors, there could be upwards of thousands to go through. In the case of Halley's Comet, the orbital period in years is 27,375 days. For every iteration, a minimum of 27,375 *12 mesh triangles would need to be generated to show just this object's orbit path.

Upon running the simulation, the meshes size becomes all too apparent as the simulation moves radically slower than before. This limitation does not appear to bottleneck the simulation for smaller orbit paths such as in the small system test I presented.

6. FUTURE WORK

In the future, I would like to solve the issue of orbit paths bottlenecking the simulation. One possibility could be to find a better way to render elliptical geometry within OpenGL directly as opposed to using the RayTree data structure. Another optimization, would like to expand the simulation to be capable of handling hyperbolic phenomenon.

Beyond simple optimizations, it would be interesting to add more phenomenon to the system such as bodies being able to orbit more than just the sun. For example, implementing moons orbiting specific planets (while the planets orbit the sun) could be a good expansion. In addition, adding capabilities to allow rings such as those for Saturn and actually mapping textures as opposed to averaging texture color values are put into consideration. A final rendering touch would be to add trails to comets or make a new body comet in general as opposed to rendering them as planets. This way they could be better differentiated from the planets.

REFERENCES

"JPL Solar System Dynamics." *NASA*. NASA. Web. 04 May 2021.

Schlyter, Paul. "Computing Planetary Positions - a Tutorial with Worked Examples." Web. 04 May 2021.

Pramod Kumar, Bhatt, Yogesh C., Shishodiya, Yogendra S., Rajmal Jain. "Simulation of Earth orbit around Sun by Computational Method." *Jagan Nath Institute of Engineering*

and Technology. N.p: n.p., n.d. Print.

Rasala, Richard. “The Rodrigues Formula and Polynomial Differential Operators.” *Journal of Mathematical Analysis and Applications*. N.p.: n.p., n.d. Print.