VR Astronomy Simulation

Final Report

Andres Diaz Ferrera, Abdil Kaldan, Ryan Milligan

Abstract:

For this project, we developed an educational Virtual Reality (VR) experience that will expose students to an immersive simulation of planetary motion. While there are many Solar System simulations available online, none of them focus on visualizing apparent retrograde motion in an educational setting. This VR experience is intended to fill that gap.

The objective of this research project was to develop and implement an efficient and accurate simulation of Kepler orbits in VR for educational settings with a focus on the visualization of apparent retrograde motion. In particular, our main focus is on the ability to view the apparent retrograde motion of planets from the perspective of planets other than Earth.

Our VR application is created using Unreal Engine 5, with code written in C++, and is designed to be used with the Meta Quest 2 VR headset. To the best of our knowledge, this is one of the first attempts in post-secondary education at creating an immersive experience for planetary motion in VR with a focus on apparent retrograde motion.

Table of Contents:

Introduction:

The application was created in Unreal Engine, as it provided a range of graphical and virtual reality development tools. However, rather than using Unreal's built-in physics engine, we opted to program the logic of the physics simulation ourselves. The application consists of two major components, an overview of the Solar System from space, and a surface view from the point-of-view of the surface of each of the planets, where the positions and motion of the other planets are visible in the sky.

Requirements:

The primary requirement of the application is the ability to view the apparent retrograde motion of planets from the point of view of planets other than earth (i.e. viewing the movement of earth from the perspective of mars while mars is in apparent retrograde motion relative to earth). Additionally, the application is required to be able to run on Meta Quest 2 VR headsets.

Summary:

Our research has involved exploring different methodologies for accurate and efficient implementation of planetary motions. In computer programming, accuracy and efficiency are often competing objectives. This is especially true for our project due to the limited amount of computational power available on a typical VR headset such as the Meta Quest 2. Furthermore, our VR experience must allow students to view apparent retrograde motion. For these reasons, our research has focused on implementing an efficient fixed-step time-marching algorithm to enforce framerate independence between drawing updates in the simulation. This also involves using Bezier spline meshes to model the predicted elliptical paths of orbiting bodies and to trace the motion of one body relative to another. In addition, multiple methods for modeling an elliptical orbit based on different starting parameters have been studied and implemented.

Future Extensions:

- Implementation of the UI for VR.
- Interactive Orbit Editor that will allow the user to experiment with changing orbital elements and seeing how the shape of the orbit changes.
- Incorporation of dwarf planets such as Pluto, Ceres, and Eris, as well as major moons such as the Galilean Moons, and Earth's moon into the model of the Solar System.
- Incorporation of elevation data into the landscape of the surface view.

Conclusions:

Our decision to program the physics simulations ourselves rather than using Unreal's built-in physics engine gave us more direct control over the simulation. It also meant that we didn't need to worry about all of the additional features that Unreal's physics engine provides, which we had no need for. Furthermore, it allows for the physics logic of the simulation to be easily reproducible in other graphics engines such as Unity, OpenGL, or WebGL. We tested implementing the same logic in the Unity game engine to confirm this, and it worked just as well there, both in 2D and in 3D. The physics logic can also be easily modified for use in particle simulations.

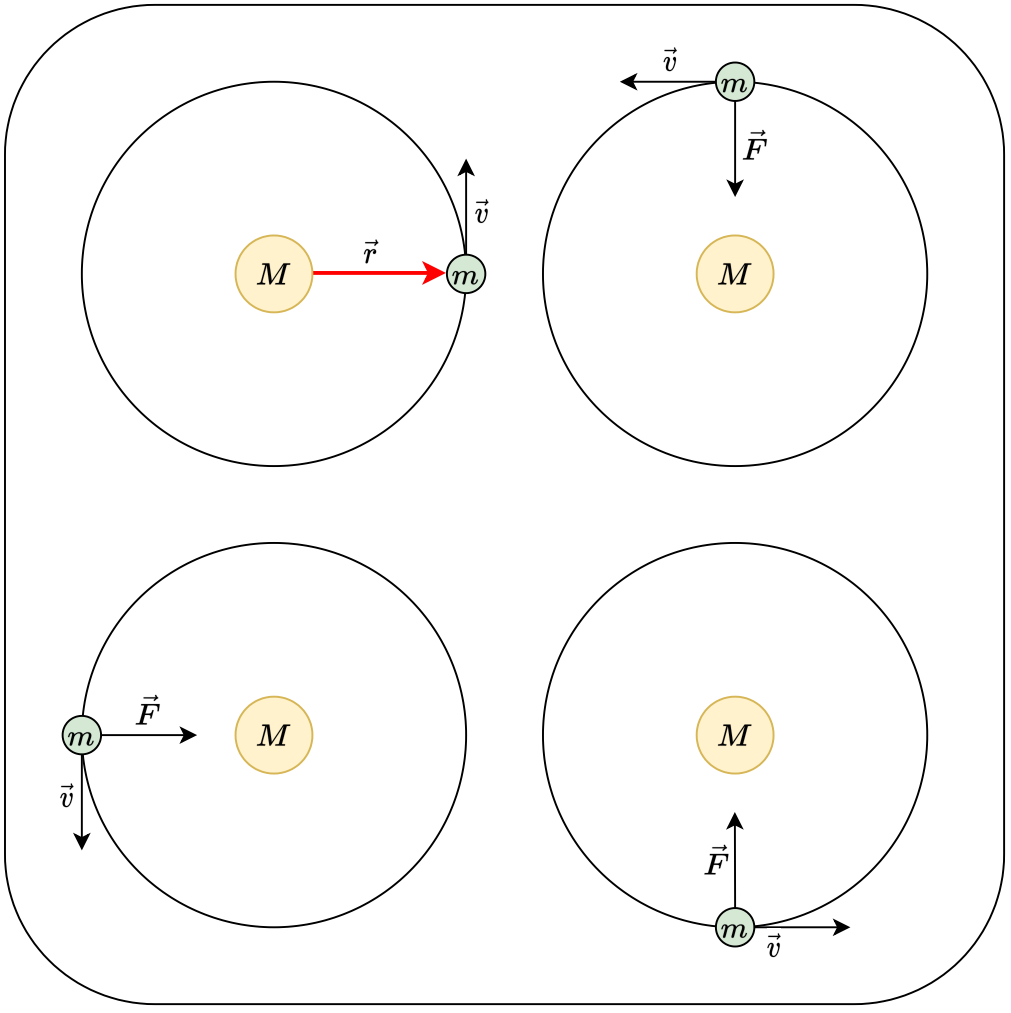
The application is intended to be used as an educational tool by the Douglas College Physics and Astronomy Department in teaching about apparent retrograde motion. Our aim is to deploy the application in classes as early as the Summer 2023 semester, so we can gather and incorporate feedback from course instructors for subsequent versions of the application.

VR Astronomy Simulation - Implementation

Simulating Orbital Motion:

To simulate the motion of the orbiting body, we calculate the orbiting body's position ($\vec{r}$) at each time step based on its velocity ($\vec{v}$), and the gravitational force exerted on the orbiting body ($\vec{F}$).

Variables	
M	Mass of central body
m	Mass of orbiting body
$\vec{v}$	Velocity vector
$\vec{r}$	Position vector
$\vec{F}$	Gravitational force



We used the Euler Method to calculate the position of the orbiting body at each time step. With it, we determine the new position ($\vec{r}_{n+1}$) by adding the current velocity ($\vec{v}_n$) multiplied with the difference in time between steps (Δt) to the current position ($\vec{r}_n$). The new acceleration is then calculated using the new position to determine the gravitational force exerted on the orbiting body. The acceleration is multiplied by the delta time and added to the velocity to get the new velocity for the next interval.
($\vec{d}$ is the direction vector pointing from the orbiting body to the central body.)

$$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_n \Delta t$$

$$\vec{d}_n = -\frac{\vec{r}_n}{|\vec{r}_n|}$$

$$\vec{a}_n = \frac{GM}{|\vec{r}_n|^2}\vec{d}_n$$

$$\vec{v}_{n+1} = \vec{v}_n + \vec{a}_n \Delta t$$

Fixed Time Step:
For the simulation to be deterministic, the difference in time between each update to the simulation must be constant. As the framerate in Unreal Engine is variable, meaning the time between frame updates varies, and because we wanted to be able to modify the time-scale of the simulation during runtime, to speed it up and slow it down, we needed to design an algorithm to enforce a framerate-independent fixed time step between updates to the simulation.

During each frame update, Unreal Engine provides the number of seconds between the last frame update and the current frame update as a “DeltaTime" variable. We specify the number of updates or “steps” that we want to be executed each second as a constant value, which is multiplied with the DeltaTime variable during each frame update to determine the number of steps that need to be executed during this frame. Since the DeltaTime variable is a floating-point value, representing a fraction of a second, the resulting value that we get from the multiplication is also a floating-point value. We must truncate this floating-point value by converting it to an integer, as we cannot perform a fraction of an update. This truncated integer can then be used in the condition of a count-controlled loop to perform the necessary number of updates. However, if we simply discard the fractional part of the value, those discarded fractional parts would accumulate dramatically over time. This would result in the simulation being framerate-dependent, as the fractional part depends on the framerate, and by extension, the simulation would become non-deterministic. Instead, the fractional part can be added to the floating-point value that is calculated during the next frame update. This ensures that the fractional part is always accounted for, and that the necessary number of steps are executed each second.

Modeling an Elliptical Kepler Orbit:

To visualize the trajectories of the orbiting bodies, so that we can see the paths that they will take before they get there, we need to model an ellipse. In order to model an ellipse, we need to know, at minimum, the length of the semi-major axis, the length of the semi-minor axis, and the position of the center of the ellipse.
Starting with the orbital parameters,

- Position vector of the orbiting body at periapsis ($\vec{r}_p$) relative to the central body
- Velocity vector of the orbiting body at periapsis ($\vec{v}_p$) relative to the central body
- Mass of the central body (M)

we first calculate the specific angular momentum ($\vec{h}$) and the specific orbital energy (ε).

$$\vec{h} = \vec{r}_p \times \vec{v}_p$$

$$\varepsilon = \frac{\left|\vec{v}_p\right|^2}{2} - \frac{GM}{\left|\vec{r}_p\right|}$$

We then use these to calculate the length of the semi-major axis (a).

$$a = -\frac{GM}{2\varepsilon}$$

From the position vector at periapsis of the orbiting body relative to the central body, and the length of the semi-major axis, we can determine the position vectors of the apoapsis and the center of the ellipse. Since the central body is at the origin ($\vec{0}$), we can get the direction vector pointing from the orbiting body at periapsis to the central body ($\vec{d}$) by inverting and normalizing the position vector of the orbiting body.

$$\vec{d} = -\frac{\vec{r}_p}{\left|\vec{r}_p\right|}$$

With this direction vector, we can get the position vector at apoapsis ($\vec{r}_a$) and the position vector of the center of the ellipse ($\vec{c}$).

$$\vec{r}_a = \vec{r}_p + 2a\vec{d}$$

$$\vec{c} = \vec{r}_p + a\vec{d}$$

Since we know that the central body, and by extension the origin, is located at one of the foci, we can use the position of the center of the ellipse to calculate the linear eccentricity (c), which is the distance between the center of the ellipse and either of the ellipse's foci.

$$c = \left|\vec{c}\right|$$

From the linear eccentricity and the semi-major axis, we can calculate the eccentricity (e).

$$e = \frac{c}{a}$$

From the eccentricity and the semi-major axis, we can calculate the length of the semi-latus rectum (l).

$$l = a(1 - e^2)$$

Finally, we can calculate the semi-minor axis from the semi-major axis and the semi-latus rectum.

$$b = \sqrt{al}$$

Once we have the lengths of the axes and the position of the center of the ellipse, we can use the parametric equation of an ellipse to determine the points that make up the ellipse.

$$\vec{r}_\theta = \vec{c} + (a\cos\theta,\ b\sin\theta)$$

Surface View:

To visualize the apparent retrograde motion of a planet from the point-of-view of the surface of a reference body, we needed to map the relative position of the other body onto a sphere representing the sky of the reference body. We did this by using the position vectors of the two bodies to define a line in 3-dimensional space between the two bodies. We then calculate the intersection of that line with a sphere which encapsulates the reference body. The point of intersection is used to map the position of the other body as it appears in the sky of the reference body. A Bezier spline is drawn between the points mapped by this process to trace the apparent motion across the sky.