Virtual Reality: Visualizing Data in Astronomy

A Standalone Visualization Tool for NEO Observation

Niles Dorn

A thesis presented for the degree of
Bachelor of Arts

Physics and Astronomy Department
Pomona College
United States of America
14 December 2022

Virtual Reality: Visualizing Data in Astronomy

A Standalone Visualization Tool for NEO Observation

Niles Dorn

Abstract

In an age of exponentially increasing data set size, modern visualization tools are necessary to convey information in a digestible manner. Contributing to the solution to this problem, this thesis covers the development of a virtual reality-based application using the Unity game engine to effectively visualize orbits of near-Earth objects relative to the Sun and planets.

Contents

Virtual Reality: Visualizing Data in Astronomy

Niles Dorn

14 December 2022

Advised by Professor Philip Choi

1 Introduction

It is difficult to discuss the field of modern astronomy without also addressing what many call "the era of big data." As the technology behind research equipment continues to develop, the amount of information astronomers can gather through research increases. For example, the Vera C. Rubin Observatory began conducting its 10-year Legacy Survey of Space and Time (LSST) in 2022. This survey is estimated to produce a 500-petabyte data set, generating 20 terabytes of data per night. For reference, many modern consumer-grade computers can hold anywhere from 256 gigabytes to one or two terabytes of data, meaning the LSST will produce about ten MacBooks worth of information per night.

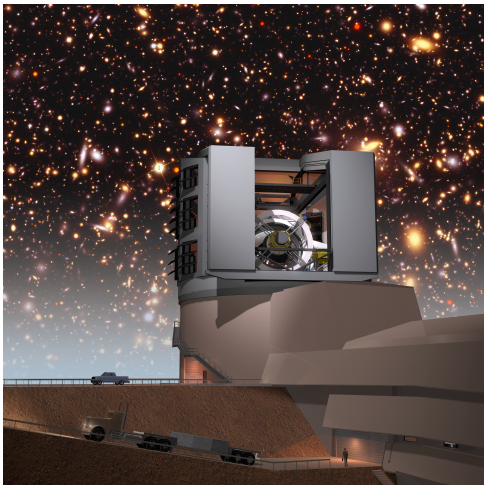

Figure 1: The Vera C. Rubin Observatory, located in the Elqui Province of Chile. Construction finished and data collection began in 2022.[1]

At the undergraduate level, this much data can be daunting for students to process, especially when they are still learning the guiding principles of astronomy. Traditionally, converting numerical data into two-dimensional graphs has been an exercise for students around the world, with the intent of strengthening their understanding of specific topics in astronomy. For example, as shown in Figure 2, in Pomona College's Advanced Introductory Astronomy course (ASTR 051) students use a spreadsheet with period,

luminosity, and magnitude measurements of Cepheid Variable Stars to create graphs for absolute magnitude vs. log of the period and luminosity vs. period. These graphs are intended to demonstrate how these stars' magnitudes and luminosities change over time. While effective, creating graphs does not always lead to comprehension with undergraduates. Over time, it will become more and more difficult to communicate to students the full extent of information gathered from modern astronomy research with standard graphs and charts. They will need new methods and tools to visualize and understand any given data set.

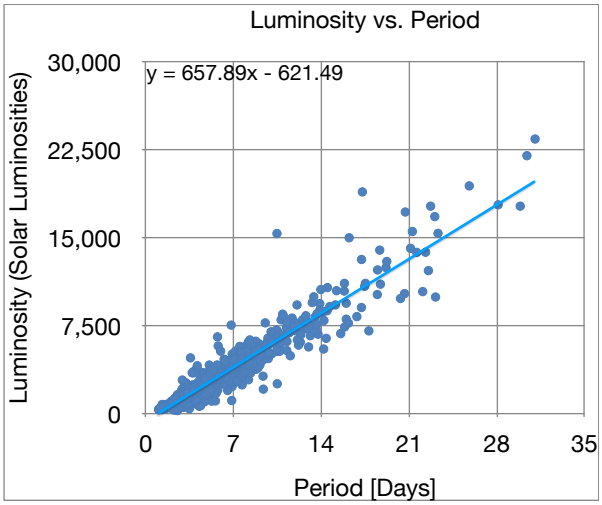

Figure 2: Cepheid Variable luminosity vs. period from the Astronomy 051 course at Pomona College. Credit: the Author.

At graduate and professional levels, overall comprehension is less of an issue, as those working in the field have years of experience. However, other problems arise. Due to the file size of many data sets, algorithms and machine learning are often employed to analyze and prune information.[2] While effective, the nature of constructing an algorithm results in not every error being caught, leading scientists to manually process data as well. This is a tedious process and, without a strong visualization tool, errors still go unnoticed.

1.1 Methods of Data Visualization

Over the years, a number of tools have been experimented with to effectively visualize data pertaining to astronomy. Two notable examples are the orrery and the planetarium. Orreries, as shown in Figure 3, are physical models of the Solar System that mechanically reflect the movement of planets and moons. These devices allowed early astronomers to visualize how certain bodies in the solar system move in relation to others. They generally consist of a clock-like mechanism with planets or moons on the clock's "arms" centered around a model Sun. The first orrery of this design was constructed by Giovanni Dondi in 1348, and these artifacts remained a common visualization tool in astronomy until the advent of the planetarium. Planetaria are similar to orreries in that they visualize the movement of bodies in the night sky, though they are often much larger than orreries and place more emphasis on communicating astronomical data to a non-scientific audience. Early planetariums resembled oversized

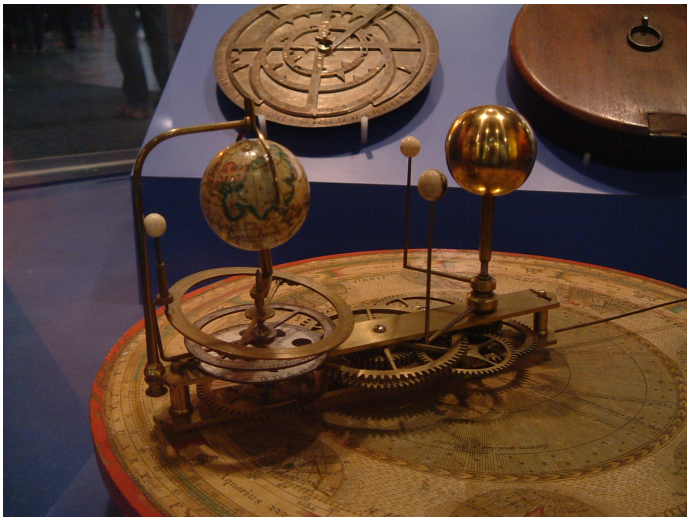

Figure 3: An example of an orrery, an early method of data visualization. Orreries are used to predict the relative position and motion of celestial bodies in the solar system. This particular orrery depicts the Earth, moon, and inner planets.[3]

orreries; by definition an orrery is a type of planetarium, though the words orrery and planetarium came to classify these devices long after orreries were first invented. The first recognizable computerized planetarium was built in 1983. Its design has become standard for many science museums and colleges. Through planetaria, audiences are able to view dynamic movement of a vast range of stellar objects, while planetarium showings can be adapted for a significant number of astronomical topics. They have often been used as data visualization tools in research, as projectors allow a large amount of data to be displayed at once and planetarium software can be utilized to view much more than just stellar movement.

These tools, while useful, do not fully solve the problem at hand. Orreries in the era of computer systems and big data are rendered almost entirely obsolete. Meanwhile, planetaria lack interactivity, limiting their efficiency as data visualization tools. Enter virtual reality (VR), a technology pioneered by the gaming industry to immerse players in virtual worlds and stories. Using a VR headset, users interact with a digital environment similar to the way they interact with the world around them. As the world of VR has developed, uses for VR outside of gaming have been investigated, leading to novel possibilities in a number of fields. Astronomy, in particular, would benefit greatly from the integration of VR into the data visualization process.

To visualize data in virtual reality, numerical data must first be translated into three-dimensional models. Once models are generated, they must be instantiated (a game development term analagous to the word *placed*) into a 3D environment to be viewed with a VR headset. The outcome of this process is shown in Figure 4. This, on its own, is not an easy task but more is necessary— interactivity is key for strong comprehension at an undergraduate level and for more efficient processing at graduate and professional levels. Once viewing the generated digital environment, users will need to have access to individual data as well as the corresponding graphs and charts traditionally used for visualization. In this way, undergraduate students will be able to connect the principles conveyed by traditional visualization to a visual model of the investigated space and objects, allowing for greater comprehension both numerically and visually. Graduate

students and professionals in the field will be able to identify errors visually, with the numerical data necessary for error confirmation readily available to them.

Figure 4: A researcher at the European Space Agency wearing a virtual reality headset and demonstrating how VR could be used to train astronauts.[4]

For all its benefits, VR, just like every developing technology, has its weaknesses. Its usefulness is limited by human biology: visual blind spots, central vision, field of view limitations, even conditions such as colorblindness and scotoma, can all make using VR difficult for some. Additionally, when unaccustomed to using a VR setup, many users report developing headaches or feeling nauseous after extended periods of time. The design of a software visualization tool must take these issues into account in order to minimize their effects. Users, for instance, might activate a "tunnel vision" filter, removing objects in the user's peripheral view to reduce motion sickness. Additionally, selecting a particular piece of data could isolate it by default, allowing the user to inspect the piece individually without interruption by the rest of the data.

A visualization tool such as this is not a wistful dream for the far future (as many things in astronomy tend to be). The technologies needed already exist and have been rapidly developing since the early 2010s. Oculus[1] (now owned by Meta), one of the first producers of consumer-grade VR equipment, was founded in 2012 and released the first consumer-grade headset, the Oculus Rift CV1, in 2016. The headset itself featured two OLED displays that provided a total field of view of 110° along with gyroscopic motion tracking to record the position and orientation of the user's head. Additionally, two handheld motion controllers allowed the user to interact with the digital environment. Each controller simulated a hand within the environment and different buttons on the controller allowed the user to perform a variety of gestures and actions. Altogether, even the very first consumer VR headset possessed the features necessary for an in-depth

[1]See: https://www.meta.com/quest/

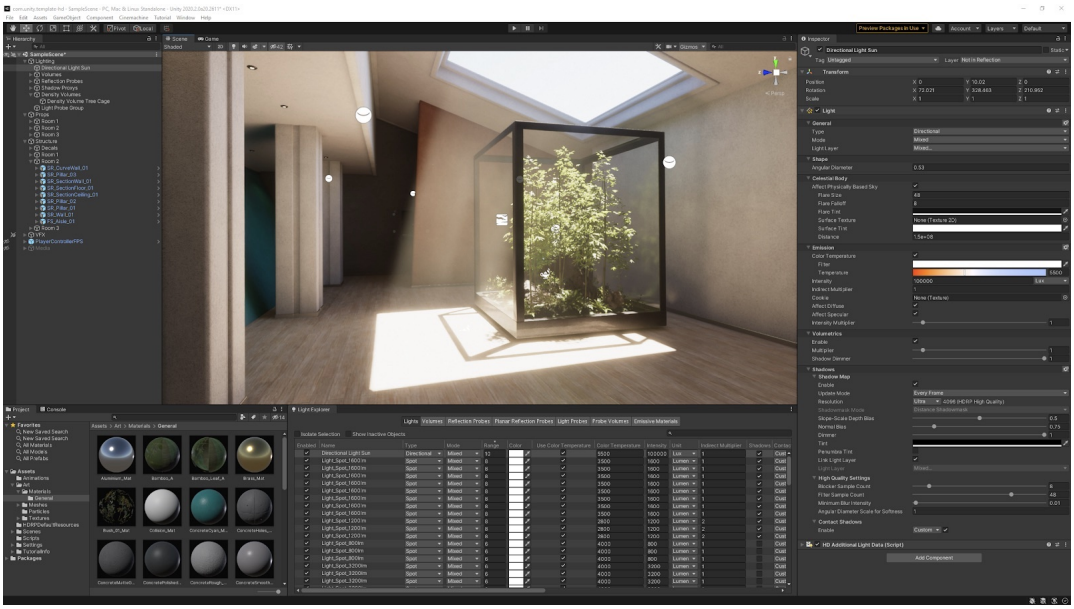

Figure 5: An example of Unity's graphic user interface (GUI). Here, the software is being used to create an interior scene. Unity's variety of materials and shaders lets users create incredibly realistic digital 3D environments.[5]

data visualization tool. Since its release, all of the above-mentioned features have become industry standard. Furthermore, modern game development engines allow for the creation of the required 3D environments and provide many of the development tools needed for interactivity features. The Unity engine[2], shown in Figure 5 and used for development on this visualization project, is available for public use by consumers and comes with an entire suite of VR-specific development tools. The use of Unity allows for relatively efficient, user-friendly software development and greatly streamlines many aspects of VR data visualization.

Virtual reality has major applications in the world of astronomy, specifically pertaining to data visualization. Software with the capability to take in large data sets, autonomously generate three-dimensional environments and models from that data, and display those environments and models with supplemental information in VR provides a modern solution to the issue of big data. Its applications impact undergraduate, graduate, and professional levels of astronomy, leading to greater comprehension and more efficient data processing. The technology and tools necessary for the creation of this software are readily available to consumers and, given that the software's design takes into account a few biological limitations, its practical application is apparent. Over time, as educators and researchers in the field of astronomy adapt this software to their needs, the presence of a VR headset in an observatory could become just as commonplace as that of an oscilloscope in a physics lab.

1.2 Near-Earth Objects

Before delving into the process of mathematically building orbits or developing the subject visualization tool of this thesis, it is important to cover the astronomical concept of near-Earth objects, otherwise known as NEOs. NASA JPL's Center for Near Earth

[2]See: https://unity.com

Object Studies (CNEOS) defines NEOs as follows:

"Near-Earth Objects (NEOs) are comets and asteroids that have been nudged by the gravitational attraction of nearby planets into orbits that allow them to enter the Earth's neighborhood. Composed mostly of water ice with embedded dust particles, comets originally formed in the cold outer planetary system while most of the rocky asteroids formed in the warmer inner solar system between the orbits of Mars and Jupiter."[6]

NEOs are of particular scientific interest for a number of reasons. First, NEOs are solar system bodies primarily left over from the formation of planets within our solar system around 4.6 billion years ago.[6] Tracking their movement, along with analyzing their chemical compositions, provides astronomers and astrophysicists with clues and insight into the formation of our solar system. Furthermore, from what some would consider a more practical perspective, determining the orbits of these objects will allow us to predict when and where these gravitationally-influenced orbiting bodies will pass by or collide with and impact the Earth, providing humanity with a sort of early-warning system for a potentially apocalyptic event. In fact, NASA's DART mission,[3] recently completed, proved that the trajectory of a NEO-like object could be altered by smashing a satellite into it before it reaches Earth.

From a physical standpoint, NEOs are defined as comets or asteroids whose perihelion distance (the point at which a body's orbit takes it closest to the Sun) falls below 1.3 AU, where 1 AU=149,597,870.7 km, defined as the average distance between the Earth and Sun. Furthermore, the broad category of "NEO" is divided into subgroups defined as follows:

- **NECs**
  - An abbreviation for Near-Earth Comets, NECs are a subgroup of NEOs containing only short-period comets.
  - These objects are defined to have perihelion distance $q < 1.3$ AU and period $P < 200$ years.
- **NEAs**
  - An abbreviation for Near-Earth Asteroids, NEAs make up the vast majority of known NEOs.
  - These objects are defined as non-comet NEOs with perihelion distance $q < 1.3$ AU.
- **Atrias**
  - Named after asteroid 163693 Atria, these objects are NEAs with orbits completely within the orbit of Earth.
  - These NEAs are defined to have aphelion distance $Q < 0.983$ AU and semi-major axis $a < 1$ AU.
- **Atens**

[3]See: https://www.nasa.gov/press-release/nasa-s-dart-mission-hits-asteroid-in-first-ever-planetary-defense-test

- Named after asteroid 2062 Aten, these Earth-crossing NEAs have semi-major axes smaller than that of Earth.
- These NEAs are defined to have aphelion distance $Q > 0.983$ AU and semi-major axis $a < 1$ AU.

- **Apollos**
  - Named after asteroid 1862 Apollo, these Earth-crossing NEAs have semi-major axes larger than that of Earth.
  - These NEAs are defined to have perihelion distance $q < 1.017$ AU and semi-major axis $a > 1$ AU.

- **Amors**
  - Named after asteroid 1221 Amor, these NEAs approach and have semi-major axes exterior to Earth but interior to Mars.
  - These NEAs are defined to have perihelion distance $1.017 < q < 1.3$ AU and semi-major axis $a > 1$ AU.

- **PHAs**
  - An abbreviation for Potentially Hazardous Asteroids, these NEAs are defined by their Minimum Orbit Intersection Distance (MOID) and absolute magnitude values.
  - These NEAs are defined to have minimum orbit intersection distance $MOID \leq 0.05$ AU and absolute magnitude $H \leq 22.0$.

1.3 Application Use Cases

This project's visualization tool was built, primarily, with the interests of the Pomona College-NASA JPL joint venture to observe, catalogue, and image near-Earth objects in mind. Therefore, its features focus specifically on NEO visualization.

1.3.1 Student Intuition

While working as observers on Pomona's NEO imaging project, students remotely access the Table Mountain Observatory to track and observe near-Earth objects. Speaking from personal experience, though in a sense students see images of the objects they are observing, these images (often viewed though color filters and independent of other orbiting bodies) do not directly translate to an inherent understanding of where these objects are in space. Furthermore, though student observers must understand the motion of these objects respective to their position on Earth in order to effectively track and image them, it is difficult to reconcile the apparent motion of objects from an observer's perspective with their own definite motion around the Sun.

The primary and most straightforward use case of this visualization tool is as follows: when student observers track, image, and catalogue the movement of any given NEO on any given night, upon returning the next night they will be able to enter said NEO's

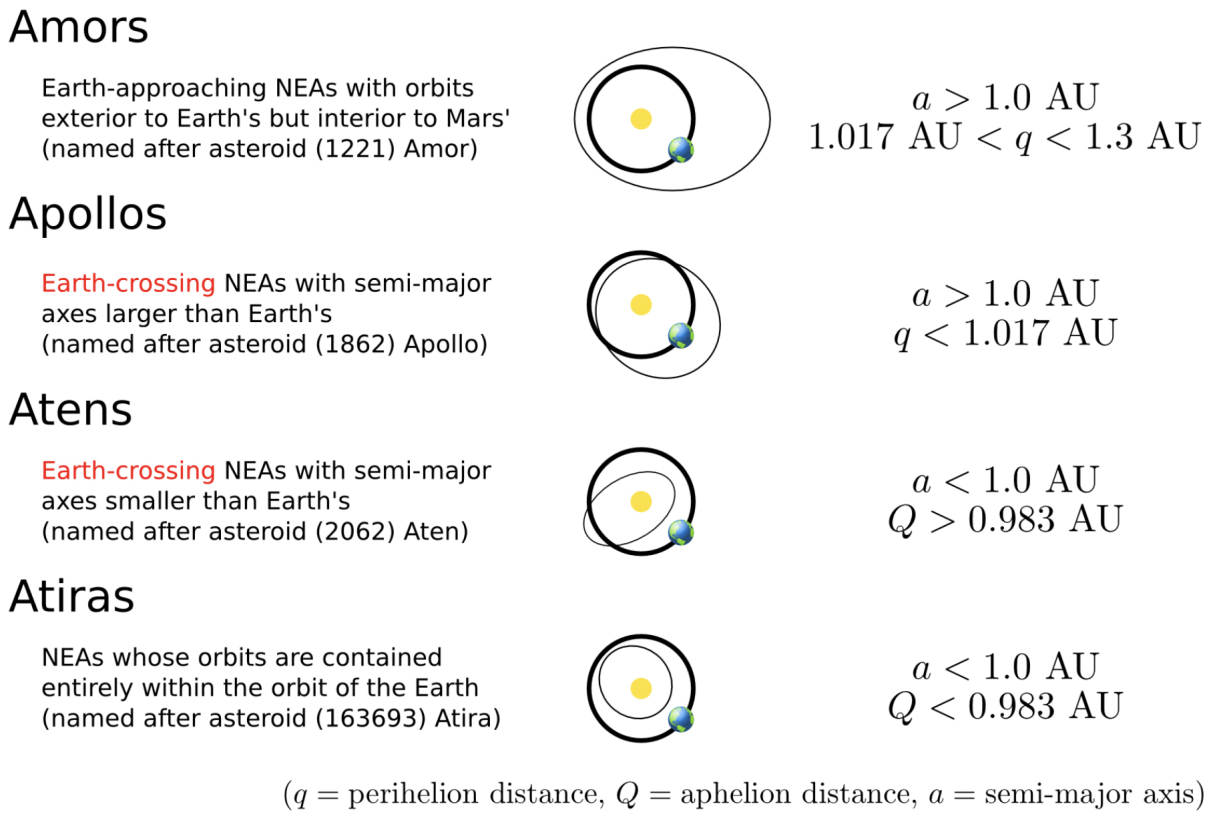

Figure 6: This graphic, Courtesy NASA/JPL-Caltech, visualizes the orbits of four types of Near-Earth Asteroids (NEAs).[7]

designation and view its position and orbital progression relative to known quantities within the solar system.

This use case, while simple and straightforward, serves an important purpose. By strengthening student intuition, observers will have a much greater sense of the hands-on work in which they are participating. As stated in Chapter 1, it can be difficult to reconcile the numerical data presented to student observers while carrying out observations with the actual science being performed. Without deep understanding and significant experience, beginner observers can easily get lost "in the numbers" and lose track of the higher-level concepts. In viewing an up-to-date visual model of their work's results, they will be able to much more quickly develop an intuition for what results are to be expected.

1.3.2 Planet 9

After Neptune was discovered in 1846,[9] scientists turned to the possibility that there may be another planet orbiting the Sun beyond Neptune. In 2014, two astronomers proposed that, based on the perturbed orbits of two small bodies orbiting exterior to Neptune, the possibility of the existence of a proposed Planet 9 was strengthened.[10]

This NEO visualization tool provides the groundwork for identifying possible positions of near-extrasolar objects. By specifying possible times at which orbits may be perturbed and batch loading a broad range of objects from NASA JPL's small-body database, individual objects can be isolated to sustain further research on for the

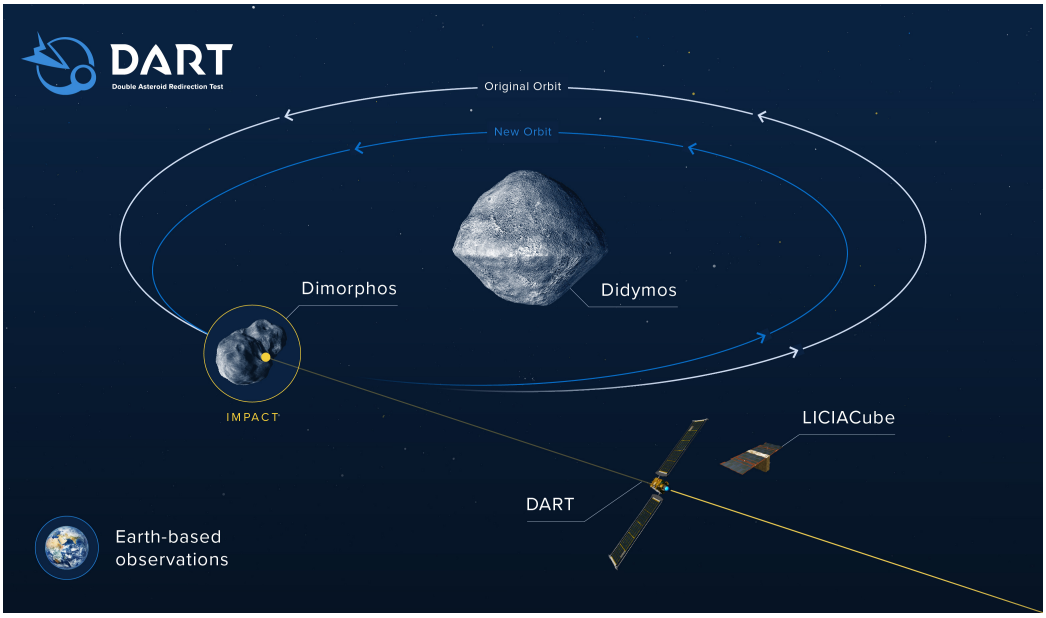

Figure 7: A diagram detailing and visualizing NASA's DART mission and its successful attempt to alter the orbit of a solar system orbiting body.[8]

possibility of discovering previously-unknown solar system bodies.

1.4 Project Goals

The broad-scale goal of this project is to produce an application capable of visualizing the orbits of near-Earth objects in the NASA JPL Horizons database for use by individuals involved with Pomona College-NASA JPL's joint venture to observe, catalogue, and track near-Earth objects. To meet the needs of the eventual end users of this project, the application needed to have a number of core features. These aspects, outlined below, were easily defined by breaking down the above-stated goal of the project.

- The application must take and record user input so as to effectively visualize the orbits of intended objects at any given user-specified time. Valid user input is defined as object names/specifications (IAU number, designation, name, or SPK-ID) and timestamps given as a Julian date.
- From the given user input, the application must retrieve relevant information on the user's intended object (or objects) necessary for modeling and visualizing orbits. In general, this information consists of previously-defined orbital element values.
- The application must use this information to calculate and model an intended object's orbit in terms of its orbital elements. It must then translate this data to be usable within a position-based three-dimensional Cartesian coordinate system.
- Finally, the application must instantiate a 3D model to visually represent the intended object. After being instantiated, the position of this model must be continuously updated so as to accurately display the object's position in its orbit.

2 Orbital Elements

Classically called Keplerian elements, the term "orbital elements" refers to a range of measurements and values that help astronomers describe the orbits of celestial bodies. Given that the use of a cartesian coordinate system is not always appropriate in space, orbital elements define orbits in terms of other orbital elements. A variety of formulae and transformations allow astronomers to easily calculate desired orbital elements from known orbital element values, solidifying the use of orbital elements as a strong system to define and describe the orbits of bodies within our solar system. Furthermore, astronomers have derived transformations that permit the use of orbits modeled with orbital elements in cartesian, vector-based, and other coordinate systems. These transformations will prove useful in regards to the development of this visualization project.

2.1 Describing Orbits with Keplerian Elements

Published between 1609 and 1619, Johannes Kepler's laws of planetary motion replaced Nicolaus Copernicus' heliocentric model as the scientific standard for describing planetary motion around the Sun. Formulated by Kepler's analysis of Tycho Brahe's observations, the three laws are as follows:

- Planets orbit the Sun in an ellipse, with the Sun stationed at one of the ellipse's two foci.
- Regardless of the planet's position in its orbit, the line segment connecting the Sun to the orbiting planet will sweep out equal areas given equal amounts of time.
- By squaring the planet's orbital period, you get a value proportional to the cube of the semi-major axis of the planet's orbit.

These laws, arising from Isaac Newton's gravitational law, allowed contemporary astronomers to revise and more accurately model the motions of planets around the Sun. However, while these laws describe the motion of the planets within their orbits, they cannot define orbits themselves relative to celestial landmarks, i.e. the Sun or ecliptic plane. To do so, we must turn to orbital elements. Modeling a planet's orbit involves incorporating six degrees of freedom, namely the three spacial dimensions and the orbiting planet's velocity in each dimension. In a three-dimensional Cartesian coordinate system, similar to the positional system used within Unity, these three spacial dimensions are commonly notated as (x, y, z). Therefore, following from Kepler's derivations of the laws governing planetary motion, there are six central orbital elements (formally known as Keplerian elements) that arise. These six main elements and their general abbreviations are:

- Eccentricity, e
- Semi-Major Axis, a
- Inclination, i

- Longitude of the Ascending Node, Ω
- Argument of Periapsis, ω
- True Anomaly, ν

Pertaining to the body's orbit, the eccentricity and semi-major axis are used to describe its size and shape, the inclination, longitude of the ascending node, and argument of periapsis are used to describe its orientation within the orbital plane, and the true anomaly is used to describe the body's position within its orbit at a given point in time.

Though these six elements are widely considered the "standard" orbital elements, alternate parameterizations are often employed under specific circumstances. For example, the semi-major axis is commonly omitted in favor of the orbit's period, as the semi-major axis can be calculated from the period using the standard gravitational parameter:

$$\mu = \frac{4\pi^2 a^3}{T^2}. \tag{1}$$

Furthermore, elemental terminology differs in regards to the orbiting body in question; when orbiting the Earth, periapsis is known as perigee.

2.2 Classical Keplerian Element Definitions

2.2.1 Eccentricity

Abbreviation(s): e
Units: unitless
JPL Horizons Definition: "An orbital parameter describing the eccentricity of the orbit ellipse. Eccentricity e is the ratio of half the distance between the foci c to the semi-major axis a: $e = c/a$. For example, an orbit with $e = 0$ is circular, $e = 1$ is parabolic, and e between 0 and 1 is elliptic."[11]
Heuristic Terms: The eccentricity of an elliptical orbit is a dimensionless parameter that determines the amount by which its shape deviates from a perfect circle. The value of an orbit's eccentricity can range from $0 < e < 1$, with $e = 0$ describing a circular orbit and $e = 1$ describing a parabolic trajectory.

2.2.2 Semi-Major Axis

Abbreviation(s): a
Units: AU (generally)
JPL Horizons Definition: "One half of the major axis of the elliptical orbit; also the mean distance from the Sun."[11]
Heuristic Terms: The semi-major axis defines the distance from the center of an ellipse to one of the two equidistant points on the perimeter furthest from the center. Therefore, the semi-major axis intersects one of the two foci of the ellipse. The name "semi-major axis" comes from the fact that it is one half of the ellipse's major axis, its longest diameter passing through both foci and the center, with ends at the two most widely separated points of the perimeter.

Notes: Along with the semi-major axis, the semi-minor axis of an ellipse can be useful in describing orbits. Following from the logic of the semi-major axis, the semi-minor axis defines the distance from the center of an ellipse to one of the two equidistant closest points on its perimeter.

2.2.3 Inclination

Abbreviation(s): i
Units: degrees
JPL Horizons Definition: "Angle between the orbit plane and the ecliptic plane."[11]
Heuristic Terms: The inclination of an orbit measures the tilt of the orbit relative to the ecliptic plane. Therefore, for a body orbiting in the ecliptic plane, its orbit would have an inclination of 0°. Furthermore, for a body orbiting perpendicular to the ecliptic plane, its orbit would have an inclination of 90°.
Notes: More generally, the inclination of an orbit describes its tilt relative to any given reference plane. For simplicity's sake, however, orbits within our solar system often use the ecliptic plane as a reference plane.

2.2.4 Longitude of the Ascending Node

Abbreviation(s): Ω, node
Units: degrees
JPL Horizons Definition: "Angle in the ecliptic plane between the inertial-frame x-axis[4] and the line through the ascending node."[11]
Heuristic Terms: The longitude of the ascending node defines the swivel of an orbit. The ascending node of an orbit is the point at which the orbiting body ascends (passes from below to above) through the reference plane. Therefore, the longitude of the ascending node is the angle from a given reference direction to the ascending node, measured within the reference plane.
Notes: For bodies orbiting the Sun, the ecliptic plane is generally used as the reference plane while the First Point of Ares (the location of the vernal equinox at J2000) is used as the reference direction.

2.2.5 Argument of Periapsis

Abbreviation(s): ω, peri
Units: degrees
JPL Horizons Definition: "Angle in the orbit plane between the ascending node and the perihelion point."[11]
Heuristic Terms: The argument of periapsis defines an orbit's rotation about its center. It is the angle between the ascending node and the periapsis of the orbit, measured within the orbital plane.
Notes: Though they all refer to the same measurement, you may encounter this element masquerading under a range of different names depending on the type of orbit. For

[4]The inertial frame x-axis refers to a previously-defined direction within a non-accelerating reference frame. In many cases, a heliocentric reference frame is used with the x-axis defined to be in the direction of the vernal equinox.

example, the above JPL Horizons definition is actually a definition for the argument of perihelion, referring specifically to bodies orbiting the Sun. Alternate names include: argument of pedigree (for bodies orbiting the Earth), and argument of periastron (for bodies orbiting stars). The term periapsis refers to the farthest point in the orbit of a planetary body about its primary body, making it the most general term for the measurement and hence why it is used above.

2.2.6 True Anomaly

Abbreviation(s): ν, θ, f
Units: degrees
JPL Horizons Definition: "Angle in the orbit plane between the perihelion point and the position of the orbiting object."[11]
Heuristic Terms: The true anomaly of an orbit defines the location of an orbiting body within its orbit. It is the angle between the direction of periapsis and the orbiting body's current position, as seen from the orbit's main focus (the point around which the body orbits).

2.3 Additional Orbital Elements

These six orbital elements can be used by themselves to model and define the orbit of a celestial body, prompting their grouping as the six "Classical Keplerian Elements." However, in many cases not all of these elements are known values. Therefore, a number of additional orbital elements exist allowing astronomers to calculate their desired Keplerian elements. The five most prominent of these additional elements are defined below.

2.3.1 True Anomaly at Epoch

Abbreviation(s): ν_{t_0}, θ_{t_0}, f_{t_0}
Units: degrees
JPL Horizons Definition: See section 2.2.6
Heuristic Terms: Due to the inherent nature of orbiting bodies, an orbit's true anomaly changes with the body's motion through its orbit. Therefore, the true anomaly is time-dependent and, for an orbiting body, an epoch must be specified for any location besides its current position. This brings us to the true anomaly at epoch, referring to the true anomaly of an orbit at a give point in time.

2.3.2 Mean Anomaly

Abbreviation(s): M
Units: degrees
JPL Horizons Definition: "The product of an orbiting body's mean motion and time past perihelion passage."[11]
Heuristic Terms: The mean anomaly describes where an orbiting body would be in its orbit if it moved at a constant speed in a circular orbit with the same period (and consequently the same semi-major axis) as the body's actual elliptical orbit. Therefore, it is the fraction of an elliptical orbit's period that has elapsed since the orbiting body

passed periapsis (called perihelion for bodies orbiting the Sun and perigee for bodies orbiting the Earth), expressed as an angle.
Notes: The mean anomaly is, fundamentally, a measure of the area swept out by an orbiting body over a given time. The area swept out per unit time by a body in an elliptical orbit is the same as the area swept out by an imaginary body orbiting at a constant speed in a circular orbit with the same period. Therefore, by finding the area swept out by this imaginary body, we can determine the area swept out by the actual body without taking the variable speed of elliptical orbits into account.

2.3.3 Eccentric Anomaly

Abbreviation(s): E
Units: degrees
JPL Horizons Definition: N/A
Heuristic Terms: The eccentric anomaly is somewhat of a mix between the true anomaly and mean anomaly. Similar to the mean anomaly, it describes the position of an orbiting body if its orbit were circular. However, it also takes the variations in speed present in the body's actual elliptical orbit into account, likening it to the true anomaly. Nevertheless, in contrast to the true anomaly, the eccentric anomaly is expressed as the angle between periapsis and the orbiting body's current position as seen from the center of the orbit, rather than the main focus of the orbit.
Notes: The eccentric anomaly is employed somewhat less frequently than the other anomaly parameters, but it can be used in tandem with an orbits eccentricity to determine the mean anomaly with the equation $M = E - e\sin(E)$. This equation must be solved using numerical methods, i.e. Newton's Method.

2.3.4 Longitude of the Periapsis

Abbreviation(s): ϖ
Units: degrees
JPL Horizons Definition: "The sum of the longitude of the ascending node and the argument of perihelion."[11]
Heuristic Terms: The longitude of the periapsis (also called the longitude of the perihelion for the motions of a planet around the sun or longitude of pericenter) of an orbiting body is the longitude at which the periapsis would occur if the body's orbit inclination were zero. It can be calculated using the equation $\varpi = \Omega + \omega$.

2.3.5 Mean Longitude

Abbreviation(s): I
Units: degrees
JPL Horizons Definition: "The sum of the mean anomaly and the longitude of perihelion."[11]
Heuristic Terms: The mean longitude describes the ecliptic longitude at which an orbiting body could be found if its orbit were circular and free of perturbations. While similar to the mean anomaly, the mean longitude is the angle between a given reference direction and the body's position in the ecliptic plane rather than the orbital plane. It can be calculated using the equation $I = \varpi + M$.

2.4 Building Planetary Models

When building planetary orbital models, NASA's planetary fact sheets provide useful values for common orbital elements. These sheets can be found at: https://nssdc.gsfc.nasa.gov/planetary/factsheet/index.html.

For each planet in our solar system, these sheets provide six Mean orbital elements for the epoch J2000: semi-major axis, eccentricity, inclination, longitude of the ascending node, longitude of perihelion, and mean longitude. These six elements, however, only include four of the six Keplerian elements necessary for modeling an orbit; instead of the argument of periapsis and true anomaly, they list the longitude of perihelion and the mean longitude.[5] To complete these planetary models, we can first calculate the argument of periapsis using the equation $\varpi = \Omega + \omega$. Then, using the equation $I = \varpi + M$, we can find the mean anomaly which in turn can be used to determine the eccentric anomaly. To do so, we use Newton's Method (described below) on the equation $M = E - e\sin(E)$. Finally, we can calculate the true anomaly with the equation

$$\cos(\nu) = \frac{\cos(E) - e}{1 - e\cos(E)}, \tag{2}$$

allowing us to complete and visualize our model.

2.4.1 Newton's Method

Newton's Method is a numerical method for approximating the roots of a real-valued function. Given a single-variable function f, defined for a real variable x, the method works by taking an initial guess and subtracting the function with respect to the initial guess divided by the derivative of the function with respect to the initial guess, resulting in a closer approximation of the initial guess. Mathematically, this formula is given by

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{3}$$

This process is repeated, i.e.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{4}$$

until a sufficiently close approximation is reached.

In our case, we will rearrange the equation $M = E - e\sin E$ to define our function $f(E)$ and, due to the similarities between the two elements, we will use our calculated mean anomaly as our initial guess x_0. In this way, we have

$$f(E) = E - (M + e\sin E). \tag{5}$$

Then, taking the derivative of $f(E)$, we get

[5]Note the differences between the longitude of the periapsis, the argument of periapsis, and the longitude of the ascending node. In contrast to the difference between the longitude of the periapsis and the longitude of the perihelion, these three elements are distinct measurements and cannot be used interchangeably.

$$f'(E) = -1 + e\cos E. \tag{6}$$

With this, we are ready to employ Newton's Method in our planetary orbital models:

$$E_{i+1} = E_i - \frac{E_i - (M + e\sin E_i)}{-1 + e\cos E_i} \tag{7}$$

2.5 Example: Mercury

With the following values, Mercury's position within its orbit at J2000 can be calculated.[6]
$a = 0.38709893$ AU
$e = 0.20563069$
$i = 7.00487°$
$\Omega = 48.33167°$
$\varpi = 77.45645°$
$I = 252.25084°$

Argument of Periapsis: for values $\Omega = 48.33167°$and $\varpi = 77.45645°$, we can use the equation $\varpi = \Omega + \omega$ to determine a value for ω:

$$\omega = \varpi - \Omega = 77.45645 - 48.33167 = 29.12478° \tag{8}$$

Mean Anomaly: for values $I = 252.25084°$and $\varpi = 77.45645°$, we can use the equation $I = \varpi + M$ to determine a value for M:

$$M = I - \varpi = 252.25084 - 77.45645 = 174.79439° \tag{9}$$

Eccentric Anomaly: we can now employ Newton's Method to approximate E. Using the mean anomaly M as our initial guess E_0, we get:

$$E_1 = M - \frac{M - (M + e\sin(M))}{-1 + e\cos(M)} = 174.7789043 \tag{10}$$

Repeating this process until we reach E_5, find that $E \approx E_5 = 174.4282903$.

True Anomaly: now that we have all the necessary values, to find the true anomaly ν we simply solve the equation $\cos(\nu) = \frac{\cos(E)-e}{1-e\cos(E)}$ for ν:

$$\nu = \cos^{-1}\left(\frac{\cos(E) - e}{1 - e\cos(E)}\right) = 175.4761384° \tag{11}$$

With this final value, our mathematical orbital model for Mercury at J2000 is complete.

[6]These values were obtained from NASA's planetary fact sheet for Mercury: https://nssdc.gsfc.nasa.gov/planetary/factsheet/mercuryfact.html

3 Project Development

In a contextual sense, developing a game with Unity involves attaching documents of code called scripts to digital assets called game objects. These game objects, with scripts attached, are then placed within Unity's scene, a digital "space" where position is defined using a three-dimensional Cartesian coordinate system. In a general sense, the scene is what the user sees and is therefore made up of game objects that are modified and manipulated by scripts. As an example, Figure 8 displays a screenshot from a game titled *Untitled Goose Game*, made in Unity and released in 2019. Everything visible in this image, ranging from the umbrellas on the clothesline to the goose itself, are game objects that are being manipulated by scripts, all contained within the scene.

Figure 8: A screenshot from House House's 2019 puzzle stealth game, *Untitled Goose Game*. The developers at House House used Unity to produce this game.[12]

It is important to understand the basic workflow of development in Unity, as without this base-level knowledge much of the vernacular and terminology used in the following sections will sound entirely foreign.

3.1 Preliminary Work

A lesson often learned when entering the field of computer science is that seemingly simple tasks are much more complex than they seem. With this lesson in mind, the first step in developing the application at hand was to hard-code a model of the Earth-Sun orbital system so that the difficulty of the overall project could be gauged. To do this, two standard sphere models were manually placed in the scene. For ease of use, one sphere was scaled up and placed at the center of the scene $(0, 0, 0)$ to represent the Sun while the other sphere, representing the Earth, was placed at an arbitrary point with a y-coordinate value of 0 near the Sun model. To rotate the Earth around the Sun, I developed a C# script named Orbit.cs and attached it the model Earth game object. This script defined a "timer" value and incremented it by a set amount each frame, averaging out to about 60 times a second. By setting the x-coordinate of the

Earth model to the cosine of the timer value and the z-coordinate to the sine of the timer value each time the value was incremented, Orbit.cs continuously rotated the model Earth in a circle around the model Sun at a distance specified by a user-defined semi-major axis value. To add more complexity, Orbit.cs was modified so that, given an orbital eccentricity value, it would use the formula $b = a\sqrt{1 - e^2}$ to calculate a value representing the semi-minor axis of the orbit. By multiplying the x-coordinate value by the semi-major axis and the z-coordinate value by the semi-minor axis, the script was capable of replicating an elliptical orbit.

These were promising results, but overall an imitation of the proposed functionality. The application could not stand on its own as semi-major axis values had to be manually entered through the Unity editor, and it was could only replicate the size and eccentricity of the orbit. Figure 9, an image of the Unity editor displaying the project build at hand, documents this state of development. Nevertheless, these results were an effective gauge of difficulty for the project as a whole. The next steps in development involved building a standalone prototype with the functionality to meet the project goals as outlined in the beginning of Chapter 3.

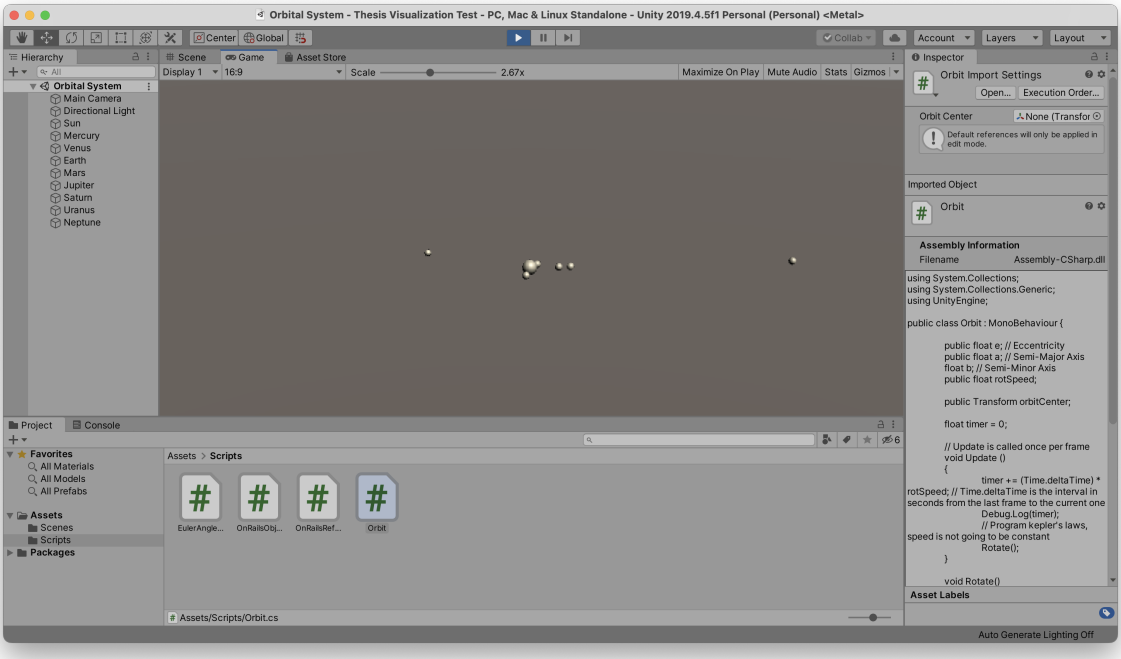

Figure 9: The Unity editor displaying a model solar system. Instances of the script Orbit.cs are attached to each model planet in the scene, individually controlling their motion over time.

3.2 Modeling Orbits in Unity

The first step in prototype development was to implement basic orbital functionality. Within Unity, every game object contains a transform component which defines the game object's position and rotation relative to the scene's origin point. Therefore, to change the position of a game object over time, it is necessary to continuously modify the $x-$, $y-$, and z-position values of the game object's transform. By default, scripts

within Unity contain an Update() function that is called on each frame update. This means that any code included in the Update() function will similarly run once per frame, allowing for continuous position modifications for game objects.

The modeling process outlined in Chapter 2 describes orbits in terms of orbital elements. While these models are capable of standing on their own, to display them accurately in Unity, x-, y-, and z-coordinate values are required to set the orbiting body's position appropriately. These transformations are given by

$$x = r[\cos\Omega\cos(\omega+\nu) - \sin\Omega\sin(\omega+\nu)\cos i], \tag{12}$$

$$y = r[\sin i\sin(\omega+\nu)], \tag{13}$$

$$z = r[\sin\Omega\sin(\omega+\nu) + \cos\Omega\sin(\omega+\nu)\cos i]. \tag{14}$$

It is important to note that, in this case, the y- and z-transformations have been swapped in accordance with Unity's coordinate system. In these transformations, the variable r represents the physical distance between the orbiting body and the object it is orbiting, given by the following formula:

$$r = a(1 - e\cos E). \tag{15}$$

Following this strategy, the desired outcome was achieved by writing a script named OrbitingBody.cs that takes five orbital elements (eccentricity, semi-major axis, inclination, longitude of the ascending node, argument of perihelion) along with two orbital parameters (time of perihelion passage, mean motion) as input, carries out the necessary calculations to determine the remaining orbital elements (mean anomaly, eccentric anomaly, true anomaly), calculates the radius r, and carries out the Cartesian coordinate transformations. The script then sets the x- y- and z-coordinate values of the game object's transform component to the calculated coordinate values. Provided with the relevant input values, repeating this process once a frame using the script's Update()[7] function results in an accurate orbit visualization for the object.

3.3 Retrieving Orbital Element Data from User Input

With basic implementation of orbit visualization complete, the next step in project development was to record user input and retrieve orbital data relevant to said user input. Unity provides built-in assets tailored to recording user input, namely the inputfield UI asset. When an inputfield is present in the scene, users are able to click on its associated in-game textbox to type and enter their input (for example, the designation of an intended NEO target). This information is recorded by the inputfield asset so it can be accessed by scripts.

For any catalogued object, the NASA JPL Horizons small-body database lookup tool (https://ssd.jpl.nasa.gov/tools/sbdb_lookup.html#/) is capable of displaying a

[7]Unity's Update() function, as stated, is called on each frame update allowing coded processes to occur many times a second. However, if thousands of objects are loaded into the scene at once, the increase in required processing power can lead to a decrease in framerate, resulting in a decrease in the rate at which Update() is called. Further development on this project may involve researching Unity's FixedUpdate() function, which could help negate these performance issues.

number of orbital parameters. As shown in Figure 10, the small-body database lookup outputs the following object-specific values:

- Eccentricity
- Semi-major axis
- Perihelion distance
- Inclination
- Longitude of the ascending node
- Argument of perihelion
- Mean anomaly
- Time of perihelion passage
- Sidereal orbital period
- Mean motion
- Aphelion distance

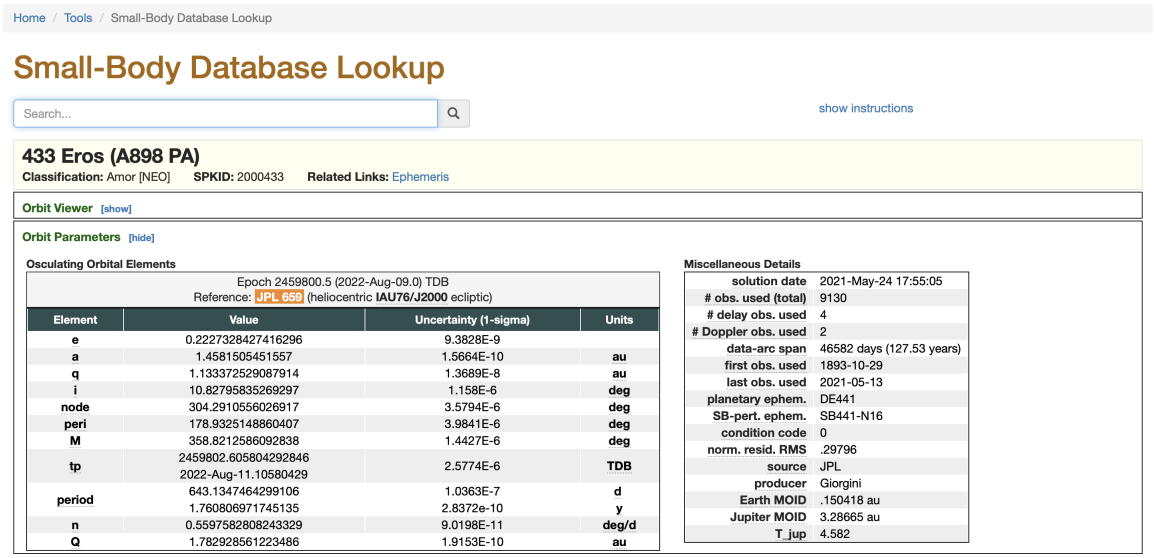

Figure 10: The output of JPL Horizon's Small Body Database Lookup tool for object 433 Eros.[13]

Furthermore, Horizons offers an application programming interface (API) allowing for direct access to the service through scripts. Therefore, to retrieve orbital data on a target entered by the user, a new script named NEOManager.cs was implemented. This script accesses the user input recorded by the inputfield and dynamically builds a URL to contact using Unity's WebRequest feature. Upon contacting the Horizons API through the built URL, the script receives a response containing the requested information. By parsing this response, the script records the relevant data for further use.

Another key feature of Unity involves the use of prefabs, or developer-created assets that can be used multiple times throughout a scene. For example, this project makes use of an "Orbiting Body" prefab consisting of an model asteroid game object with the a copy of OrbitingBody.cs attached. This prefab can be referenced by script so that, for example, NEOManager.cs can instantiate the entire Orbiting Body prefab into the scene at once instead of instantiating a game object, rendering the asteroid model, and then attaching OrbitingBody.cs as distinct steps.

After recording the orbital data from the Horizons API response, NEOManager.cs instantiates an instance of the Orbiting Body prefab and passes along the values given by Horizons. This allows the instance of OrbitingBody.cs attached to the Orbiting Body prefab to perform the calculations required to update the object's position accordingly. Combined, these two scripts provide the majority of the application's basic functionality, permitting users to visualize the Orbits of catalogued NEOs by entering their designations.

3.4 Time Management

The last step in prototype development was to implement a basic time management system. This system keeps track of the value used in OrbitingBody.cs's mean anomaly calculation, which is in turn used to determine the true anomaly and therefore the position of the orbiting object. Using a script named TimeManager.cs, this system was successfully implemented.

Within the application, time is measured as a Julian date. While it is possible to display orbit visualizations in real time, due to the timescales involved with orbital periods, doing so results in movement imperceptible to the human eye. Therefore, after converting the user's current system time to a Julian date on startup, TimeManager.cs increments this Julian date value by an arbitrary amount each frame to provide the user with a more effective visualization of the movement of instantiated targets.

These three scripts, OrbitingBody.cs, NEOManager.cs, and TimeManager.cs, handle the basic functionality of this application and constitute the majority of the work involved with developing the application prototype. The overarching process carried out by these scripts is displayed in Figure 11, and the prototype itself is shown in Figure 12. Together, they provided a strong foundation on which additional features were developed to improve user experience and application usability, a number of which are discussed in the next chapter.

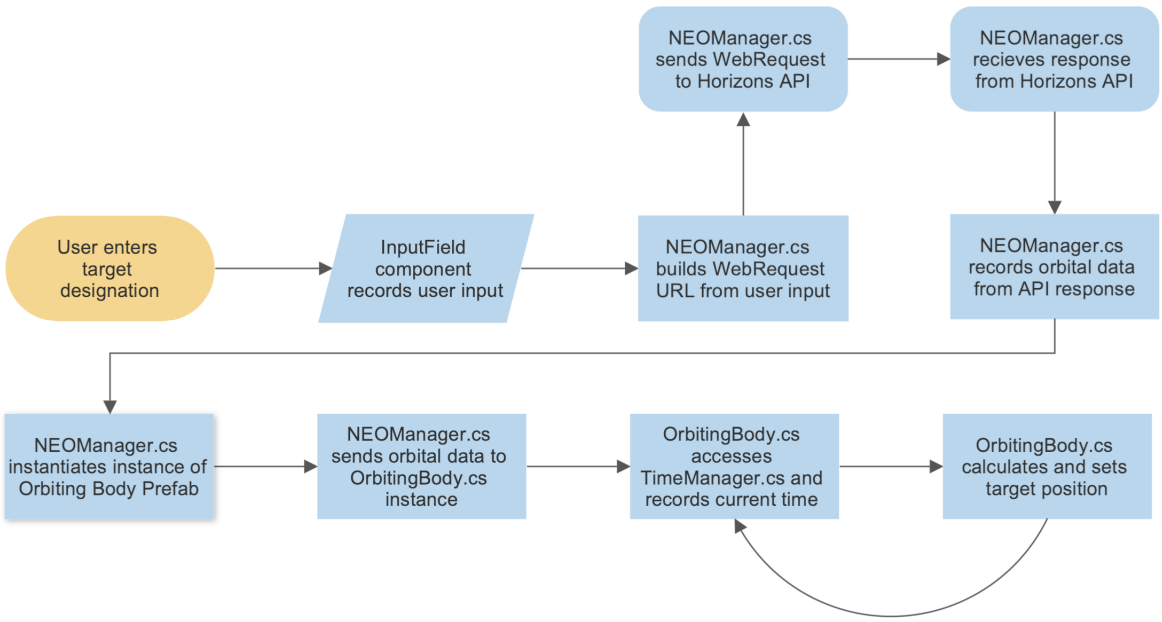

Figure 11: A flowchart diagramming the code structure of the prototype functionality process.

4 User Experience and Virtual Reality

Along with the core aspects discussed in the previous chapter, a number of other features were discussed and implemented to streamline user experience and improve ease of use. Many of these features are visible in the drastically improved UI elements, while others are visible in-scene. Furthermore, given the application was developed with the intention of being used in virtual reality, VR interactivity was implemented along with Oculus Quest 2 compatibility.

4.1 User Experience

Though greatly improved relative to the project build in Figure 9, as shown in Figure 12, the initial application prototype was lacking in overall functionality. To remove previously added targets from the scene, users were forced to restart the application entirely, there was no way to distinguish orbits of NEOs from those of planets, users had no way to see what targets they had added to the scene, and there was no way to manipulate the passage of time. Further project development remedied all of these issues.

4.1.1 User Interface Improvements

From Figure 12, the initial prototype application allowed for the singular action of adding individual targets to the scene. The first step in expanding the user interface involved adding a function to NEOManager.cs allowing for the removal of user-specified targets in the scene. The added function takes user input from the afore-mentioned inputfield component and searches for an in-scene target with a designation matching the user input. If successfully found, this function uses Unity's Object.Destroy() function to completely remove the object and any attached components from the scene.

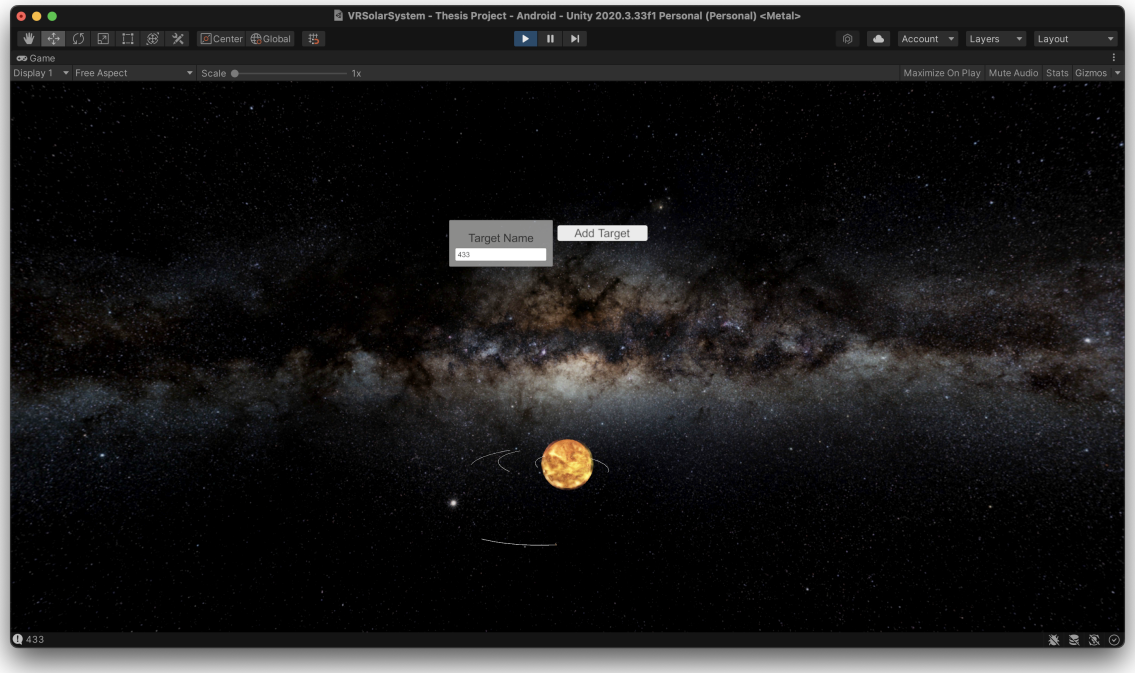

Figure 12: A view of the initial application prototype running in the Unity editor. Mercury, Venus, Earth, and object 433 Eros can be seen orbiting around the model Sun.

This function was then mapped to a UI button labeled "Remove Target," lending users a method of removing specific targets from the scene. This concept was expanded upon with the implementation of a "Remove All Targets" button, similarly mapped to a newly-written function within NEOManager.cs. This function simply calls Object.Destroy() while looping through every object added to the scene, resulting in the removal of all current targets. Further relevant improvements include the addition of two checkbox UI elements, one labeled "Hide Planets" and the other labeled "Hide NEOs." These checkboxes are linked to the parent game objects of the visible planets and current targets, respectively. Interacting with these UI elements enables or disables their linked parent game object as necessary, resulting in the temporary removal of orbiting objects in the scene.

The next UI improvement was the addition of a visible list of current targets. Using Unity's built-in scrollview component (visible in Figure 13), a dynamic list was added next to the original inputfield component. By recording the designation of each object added to the scene, a function written within NEOManager.cs was able to instantiate an instance of a custom TargetMenuItem prefab to display the designations of targets added to the scene within the afore-mentioned scrollview component. Furthermore, whenever a current target is removed from the scene, this function uses Object.Destroy() to remove the corresponding TargetMenuItem prefab instance from the scrollview component. To expand on this feature, a "Target Info" area was added to the UI while an additional component was attached to the TargetMenuItem prefab, transforming it into an element selectable by the user. When the user clicks on a designation in the list of current targets, the clicked TargetMenuItem calls a function within a script named TargetInfoManager.cs. This function accesses the instance of Orbit-

ingBody.cs attached to the object with the clicked designation, takes the orbital data stored in OrbitingBody.cs, and displays it along with the selected object's designation in the Target Info area.

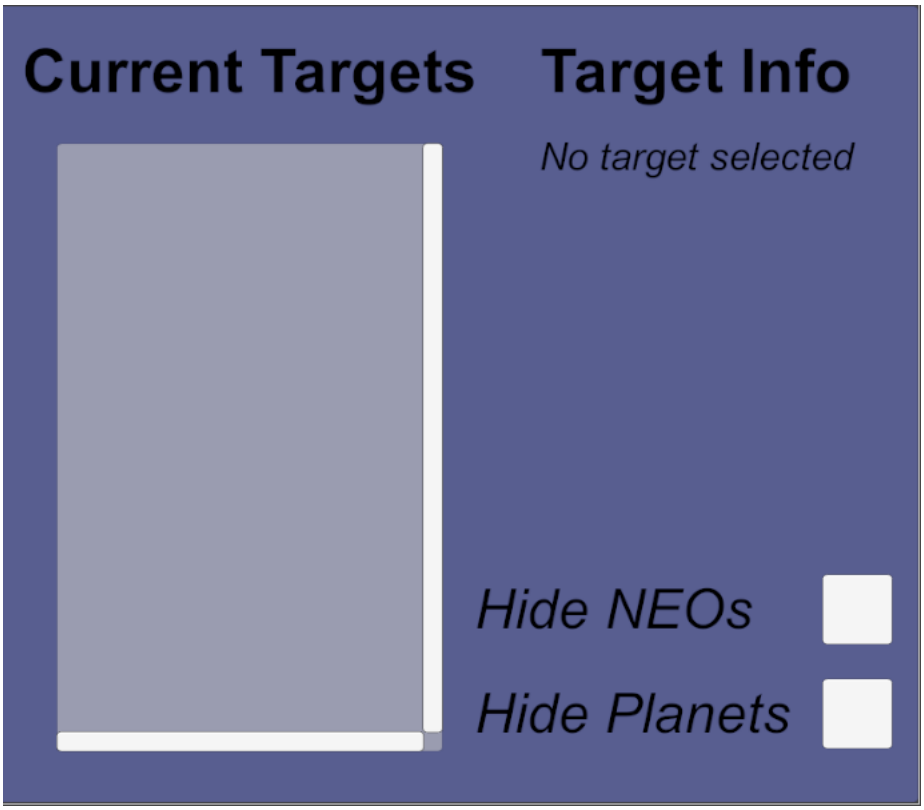

Figure 13: The current target and target info UI elements. These elements display information on in-scene targets to the user. The hide planets and hide NEOs checkbox elements are additionally visible.

Aside from usability in regards to current targets, the initial application prototype lacked any means for the user to manipulate the time value used to calculate orbit positions. To address this issue, a separate collection of UI elements was developed, allowing for time scaling and manipulation by the user. As seen in Figure 14, the first element implementation was that of a "Play/Pause Time" button, linked to a function written within TimeManager.cs. Upon user interaction with this button, the function in TimeManager.cs set the arbitrary incrementation value discussed in section 3.4 to 0, leading to the cessation of motion by all orbiting objects. When clicked again, the incrementation value was reset to its previous value, resuming motion of in-scene objects. Along with the Play/Pause Time button, a "Reverse Time" toggle was added to the UI, similarly linked to a function in TimeManager.cs. When this toggle registers user interaction, however, the function in TimeManager.cs multiplies the incrementation value by -1, leading to a reversal of orbital direction for all orbiting objects in the scene. To provide the user with further time manipulation functionality, a "Scale Time" UI element was implemented using Unity's built-in slider component. The slider component was linked to the time incrementation value so that, upon moving the slider, the incrementation value would increase or decrease, resulting in a change in the speed at which in-scene objects orbit around the Sun.

The next step in expanding time manipulation features took the form of allowing users to visualize in-scene objects at a specified Julian date. The implementation of

this feature followed from the prototype implementation of adding targets to the scene. Using a second inputfield and an "Enter" button, upon user interaction with the Enter button element, a function written in TimeManager.cs accesses the inputfield to retrieve the date entered by the user and simply sets the script's time variable equal to the user input. On the next frame update, in-scene object positions are calculated with the user's Julian date input, moving them to the appropriate positions in their orbit. Subsequently, linking a function in TimeManager.cs that converts the user's current system time to a Julian date and a button labeled "Current Time" provided a method for resetting the in-scene objects to their current, real-world positions was implemented. Finally, two UI elements controlled by TimeManager.cs were developed and added to the interface to display the time value being used to calculate object positions. One of these elements was set to display the value itself while the other was set to display the value converted to a calendar date and time format. Together, these seven elements were successfully implemented to provide users with sufficient control over the passage of time within the application.

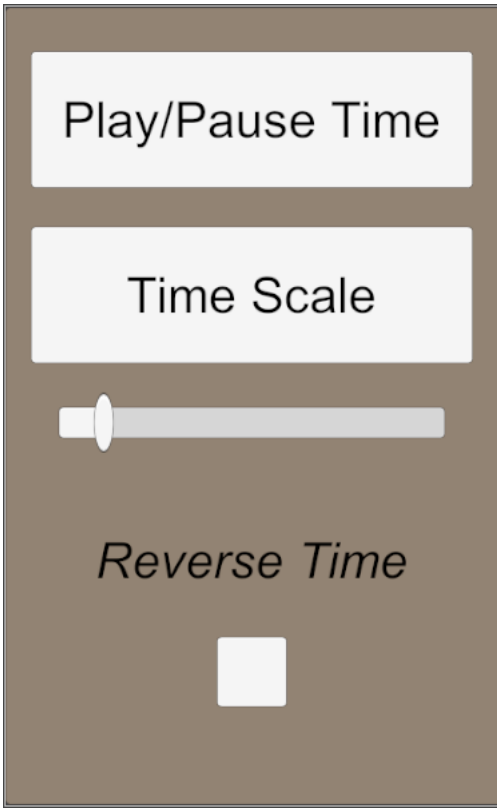

Figure 14: A section of the time manipulation interface, displaying the application's Play/Pause Time, Time Scale, and Reverse Time UI elements.

4.1.2 Batch Loading and In-Scene Improvements

Even with all these UI improvements, a feature requested by members of the the Pomona-JPL joint project was the ability to load an entire data set of NEOs at once instead of adding targets to the scene individually. To implement this feature successfully, access to the user's local files was necessary. Fortunately, in early 2020 GitHub user yasirkula published a public repository containing their Unity Simple File Browser

extension[14], also available for free on the Unity Asset Store. Importing this extension, along with modifications to NEOManager.cs and the addition of a button element labeled "Upload File," allowed for direct access to user files with user permission. The Upload File button was linked to a newly-written function in NEOManager.cs that opened an instance of the file browser. By selecting and loading file from the file browser, the afore-mentioned function recorded the text within the user's selected file and, for each new line, instantiated an object using the target designation from the uploaded file. In this way, users were afforded the capability to load hundreds or thousands of objects at once from a single .csv or .txt file.

To improve overarching in-scene visibility, three steps were taken. First, the size of the model Sun was drastically decreased as to prevent it from blocking objects behind it. During this process, the Sun was scaled down to an accurate size in line with the scale used for orbital positions and distance. Second, both the width and alpha color value of orbit trails rendered behind in-scene objects were lowered. Similarly, the orbit trails rendered behind planets were color coded and the size of the of the model asteroid used to represent NEOs in-scene was increased. Lastly, a nameplate element was added to the Orbiting Body prefab object that could be enabled and disabled through script. Therefore, when an in-scene object was selected from the current targets list by the user, a function written in NEOManager.cs enabled the relevant nameplate above the orbiting object, rendering the nameplate visible to the user.

4.2 Virtual Reality

With the implementation of additional user experience improvements, the final step in product development was to render the application functional in virtual reality. While Unity provides developers with a wide range of tools to develop applications for VR, this application mainly utilized the XR Interaction Toolkit and the OpenXR Plugin. Furthermore, importing the Oculus Integration Package provided the framework necessary to make use of the Oculus Quest 2 system keyboard overlay, allowing for user text input in VR.

The first step in implementing VR interaction capability was to add an XR Origin game object to the scene, and to link Unity's main camera to the XR Origin object. Doing so allows the application to make use of the head tracking features inherent to VR software. Next, two game objects with XR Ray Interactor components were added to the scene and linked to the XR Origin, providing users with visible objects to represent their hands in-scene. By adding Tracked Graphic Raycaster components to the UI panels and in-scene objects, these objects were able to register input from the XR Ray Interactors representing the user's hands. Lastly, the Oculus Integration Package allowed for the creation of a script to open the Oculus System Overlay Keyboard upon user selection of an inputfield.

Though Unity's OpenXR Plugin was not explicitly used over the course of development, it provided the framework necessary for building .apk project files compatible with a variety of consumer-grade VR headsets. Therefore, with the implementation of these interaction tools complete and an .apk project file built using the OpenXR framework, the application was capable of running independently on an Oculus Quest 2 VR headset as intended.

5 Final Product

Figure 15 displays a front-on, zoomed-in view of the standalone application final build visualizing the 426 objects that make up Pomona College's observational NEO catalogue. Object 2022 MM3 has been selected from the current targets list, displaying orbital data to the user on the target UI panel and placing a nameplate above the object's current position to distinguish it from the other in-scene objects. Though the majority of the work completed throughout this project is not explicitly visible to end users, this image is representative of the functionality of the final product. Undeniably a far cry from the first orbital test displayed in Figure 9, the concepts in use remain the same. As in Figure 12, the initial application prototype featured a rudimentary user interface, allowing for the single action of adding individual targets to the scene. With continued development, the prototype's UI was expanded upon and divided into two separate panels to accommodate new features. Furthermore, the code structure provided by Figure 11 still constitutes the main process invoked by the final product. Nevertheless, the additional features developed throughout the project act on this central structure to provide end users with ample capability to visualize the orbits of NEOs relative to the larger bodies present in the solar system.

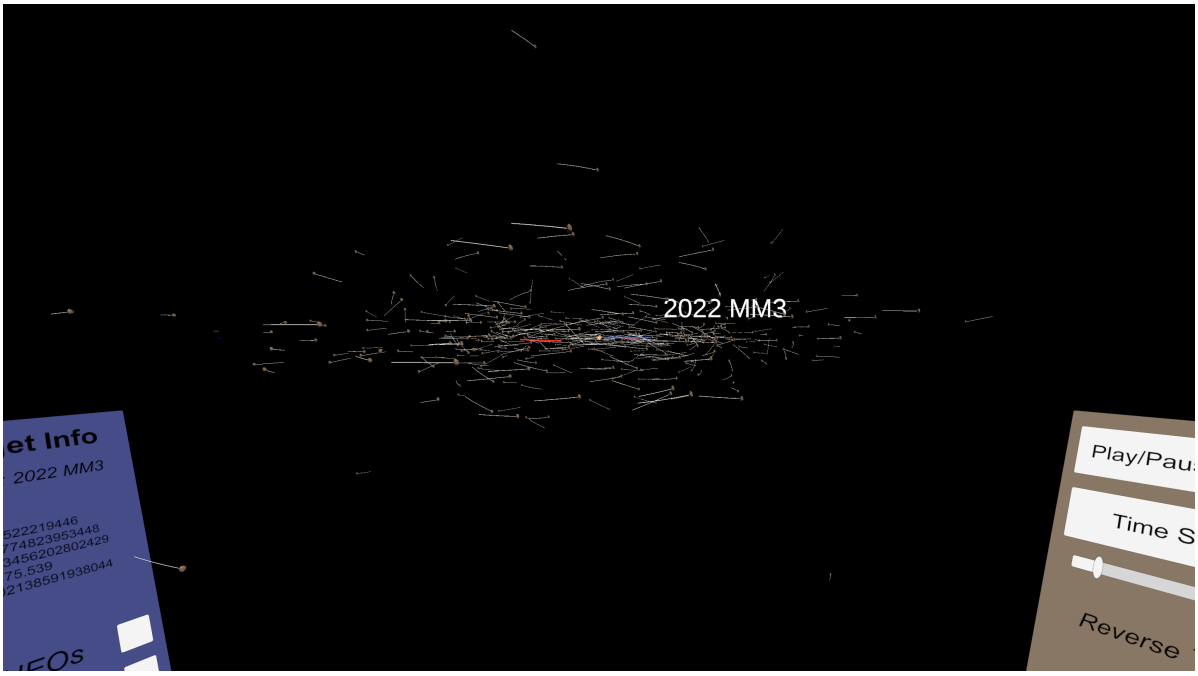

Figure 15: A visualization of Pomona College's 426-object observational catalogue. Object 2022 MM3 has been selected, displaying a nameplate above the object.

To install the application on your own device, the exported .apk file is available for download in the project GitHub repository. This can be found at: https://github.com/ign1ka/XR-NEO-Visualizer. Once downloaded to your personal computer, the .apk file can be transferred to your Oculus device using the SideQuest application available for download at https://sidequestvr.com. Once transferred to your Oculus device, the application can be found in the Unknown Sources section of your Oculus Quest application library. After launching the application, accept its request to access files on your device. At this point, once the application loads, it will be ready for use.

5.1 User's Manual

This section provides an overview of the features available for use in this application.

5.1.1 Application Orbital Scene

Upon startup, the application will display a front-on view of the nine major solar system planets orbiting the Sun. Each planet is distinguishable from the others by the color of their orbit trail; Mercury's orbit trail is grey, Venus' is orange, Earth's is blue, Mars' is red, Jupiter's is dark orange, Saturn's is yellow, Uranus' is light blue, and Neptune's is dark blue. When targets are added to the scene through the user interface, additional objects will appear orbiting around the Sun followed by white orbit trails. Representing the user's hands, in-scene the Oculus controllers take the form of two red rays projecting outward towards the model solar system.

When either of the controller rays collide with an interactable object, it will turn white. This occurs when pointing at any UI element, added NEO, or the Sun. While UI interaction is discussed in the next section, hovering over an orbiting NEO and pressing the relevant controller's grip button will select the NEO, displaying a nameplate above its model along with orbital parameters in the UI. While hovering over the Sun, pressing and holding the relevant controller's grip button allows the user to move the entire Solar System and any present NEOs to a desired location. Furthermore, while still holding the grip button, pushing the controller's thumb stick forwards or backwards will move the system towards or away from the user and pushing the thumb stick to the left or right rotates the system accordingly. Upon application restart, the system will return to its original position.

5.1.2 User Interface

The application's user interface is primarily divided into two separate control panels; one regarding target management and selection, and the other pertaining to the measurement and passage of time. Through these controls, users are able to add and remove targets to the scene, hide and show the solar system planets, scale and reverse the progression of time, reset the model's time to match the user's system's current time, etc. While interacting with in-scene objects requires the user to utilize the Oculus controller's grip button, UI interaction is achieved using the Oculus controller's trigger. Therefore, in the following sections, the word "clicking" or "click" refers to pressing the trigger.

5.1.3 The Target Panel

The left control panel, shown in Figure 16 and subsequently known as the target panel, allows users to manage targets present in-scene. The target panel is split into three main sections: the Target Name section, the Current Targets section, and the Target info section.

5.1.4 Target Name

The left-most section of the target control panel deals primarily with adding and removing intended targets to the scene. Clicking on the inputfield above the "Add Target"

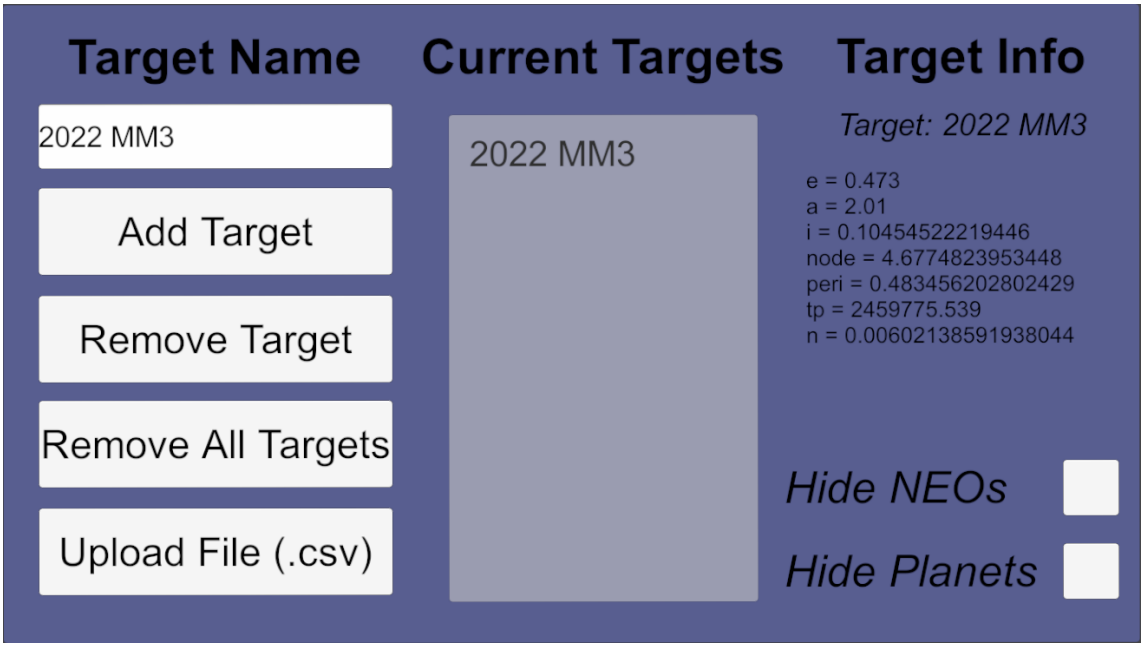

Figure 16: The target control panel, where users manage and discern information on current targets.

button opens the Oculus System Keyboard Overlay, allowing the user to enter the designation of a target they would like to add to the scene. This can be achieved by entering the intended target's IAU number, designation, name, or SPK-ID. Target names are not case-sensitive.

Upon entering the desired target designation, the Add Target button can be clicked to add the specified target to the scene. The "Remove Target" button works in a similar way, where the target matching the designation entered into the input field will be removed from the scene upon click. Clicking "Remove All Targets" will clear the scene of added targets, leaving only the Sun and planets.

Clicking the "Upload File (.csv)" button will open the file browser, allowing the user to search for and select a file containing target designations to load into the scene. Upon locating and selecting the intended file to upload, clicking the "Load" button found towards the bottom right corner of the file browser will automatically begin adding the specified targets to the scene. The easiest way to obtain a dataset formatted in this way is to use the NASA JPL Small-Body Database Query (this can be found at: https://ssd.jpl.nasa.gov/tools/sbdb_query.html) and, after specifying the necessary parameters and clicking "Get Results," clicking the "Download (CSV-Format)" button. This webpage can be accessed from the Oculus Quest's built-in internet browser application, and the downloaded files can be found in the device's "Downloads" directory.

5.1.5 Current Targets and Target Info

The current targets menu displays a scrollable list of all the added targets present in the scene. As the user adds and removes targets to and from the scene, the current targets list will update accordingly. Upon clicking a target present in the list, its nametag will be highlighted to distinguish it from other targets in the scene, and a variety of parameters, along with the target designation, will be displayed under the Target Info

section of the target panel. When there are too many targets in the scene to be displayed by the list at once, a scrollbar will appear on the right-hand side of the list. In this way, the list of current targets can be scrolled through by clicking and dragging the scrollbar up or down.

The "Hide Planets" toggle allows users to toggle view of the solar system planets in the scene. Clicking the checkbox display a check within the box and will temporarily remove the planets from view. The "Hide NEOs" toggle works the same way, but upon user interaction will temporarily remove all current targets from view.

5.1.6 The Time Panel

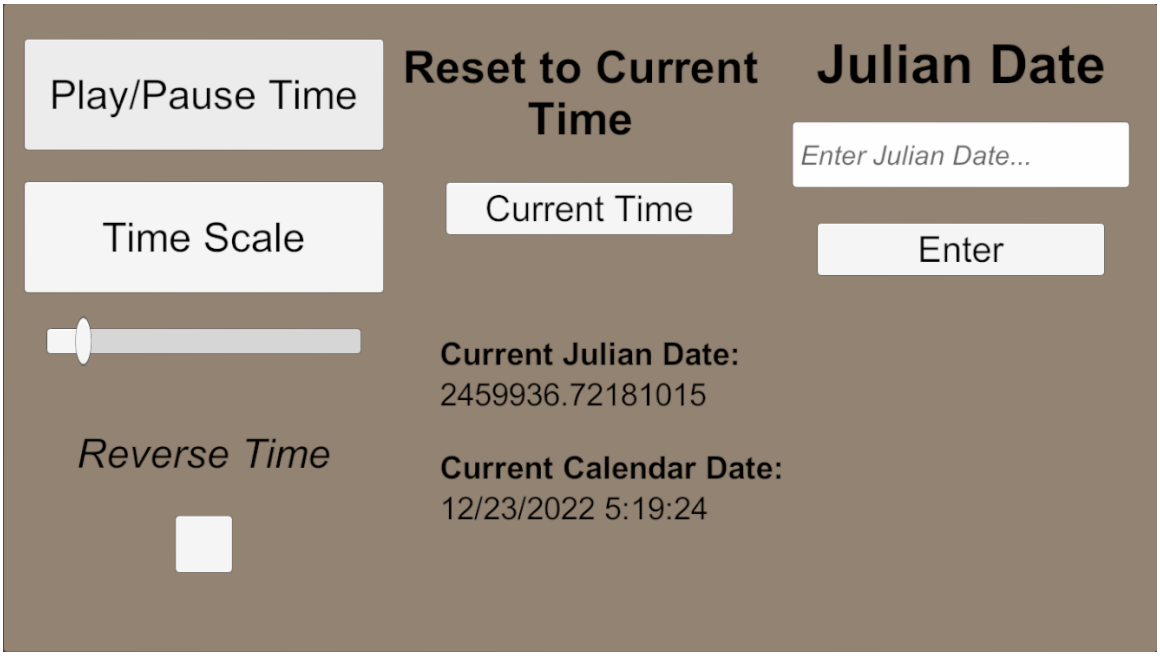

Figure 17: The time control panel, where users manipulate the progression of time within the application.

The right-hand control panel of the NEO visualization tool allows the user to manipulate the time value used to calculate object positions. Displayed in Figure 17, the time panel is split into three main sections: the time manipulation section, the "Reset to Current Time" section, and the "Julian Date" section.

5.1.7 Time Manipulation

This section of the time control panel, found on the left-hand side of the panel, displays controls allowing users to manipulate the passage of time. The "Play/Pause Time" button is relatively self-explanatory, as clicking it will halt or resume the passage of time. The "Time Scale" slider allows the user to speed up or slow down the speed at which time passes in the scene by clicking and dragging the slider to the left or right. When positioned all the way to the left, the objects in scene will progress in real time, which is, for all intents and purposes, imperceptible. The further to the right the slider is positioned, the faster targets will move in the scene. The "Reverse Time" toggle functions similarly to the "Hide Planets" and "Hide NEOs" toggles found on the target

panel. Clicking the checkbox will display a check, and all in-scene objects will begin orbiting in the opposite direction. Clicking the checkbox again will remove the check and in-scene objects will begin orbiting in their original direction.

5.1.8 Reset to Current Time

The reset to current time section contains the "Current Time" button and the "Current Julian Date" and "Current Calendar Date" readouts. Upon clicking the Current Time button, the value used to calculate the position of objects in the scene will be set to the current system time, converted to a Julian date. The current Julian date readout displays the value used to calculate target positions at any given time, and the current calendar date readout displays the same value converted to a calendar date and time format.

5.1.9 Julian Date

The Julian Date section allows users to enter a specific Julian date in order to view the positions of the planets and targets in-scene on that date. This Julian date must be entered as a combined number, i.e. 2459891 for November 7th, 2022. After entering the desired date, clicking the "Enter" button will update all planet and target positions as well as the Current Julian Date and Current Calendar date readouts in accordance with the entered value.

6 Appendix

This Appendix displays the code that handles the basic functionality of the application. It is separated into three sections, each covering one of the three primary application scripts: OrbitingBody.cs, NEOManager.cs, and TimeManager.cs. These scripts, along with all additional scripts and assets used for this application can be found at: https://github.com/ign1ka/XR-NEO-Visualizer.

6.1 OrbitingBody.cs

This script manages and updates the position of objects visible in the scene. Orbital element values, passed in by NEOManager.cs, are used to calculate objects' positions in a three-dimensional cartesian coordinate system.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class OrbitingBody : MonoBehaviour
{
    // Variable definitions

    // Orbital elements given by Horizons
    public double e;    // Eccentricity; unitless
    public double a;    // Semi-major axis; au
    double q;    // Perihelion distance; au
    public double i;    // Inclination; degrees
    public double node; // Longitude of the ascending node; degrees
    public double peri; // Argument of perihelion; degrees
    double mC;   // Mean anomaly [current]; degrees
    public double tp;   // Time of perihelion passage; TDB
    double period;   // Period; days or years
    public double n;    // Mean motion; degrees/day
    double Q;    // Aphelion distance; au

    // Calculated orbital elements
    double M;    // Mean anomaly at time t; degrees
    double E;    // Eccentric anomaly at time t; degrees
    double v;    // True anomaly at time t; degrees
    double r;    // Magnitude of radius at time t; au

    // Other
    GameObject NEOManager;

    private double currentTime;
```

```
// Start is called before the first frame update
void Start()
{
    ConvertToRads();
    NEOManager = GameObject.FindWithTag("NEOManager");
    var timeManager = NEOManager.GetComponent<TimeManager>();
    currentTime = timeManager.time;

}

// Update is called once per frame
void Update()
{
    var timeManager = NEOManager.GetComponent<TimeManager>();
    currentTime = timeManager.time;
    MeanAnomaly();
    EccentricAnomaly();
    TrueAnomaly();
    RadiusMag();
    CoordinateTransformations();
}

// Converts elements given in degrees to radians
void ConvertToRads()
{
    i *= (Math.PI / 180);
    node *= (Math.PI / 180);
    peri *= (Math.PI / 180);
    n *= (Math.PI / 180);
}

// Calculates & normalizes mean anomaly M
void MeanAnomaly()
{
    M = n * (currentTime - tp);
    M %= (2 * Math.PI);
    if (M < 0)
    {
        M += (2 * Math.PI);
    }
}

// Calculates f(E)
double F_E(double En)
{
    return (En - (e * Math.Sin(En)) - M);
}
```

```
//Calculates f'(E)
double DF_E(double En)
{
    return (1 - (e * Math.Cos(En)));
}

// Approximates eccentric anomaly E
void EccentricAnomaly()
{
    double En = M;
    double delta = 1f;
    for (int j = 0; delta > 1e-6 && j < 10; j++)
    {
        double En1 = En - (F_E(En) / DF_E(En));
        delta = Math.Abs(En1 - En);
        En = En1;
    }
    E = En;
    // Debug.Log("E = " + E);
}

// Calculates true anomaly v
void TrueAnomaly()
{
    v = 2 * Math.Atan(Math.Sqrt((1 + e) / (1 - e)) *
    Math.Tan(E / 2));
    if (v < 0)
    {
        v += (2 * Math.PI);
    }
}

// Calculates radius magnitude r
void RadiusMag()
{
    r = a * (1 - (e * Math.Cos(E)));
    // Debug.Log("r = " + r);
}

// Calculates Cartesian x-, y-, and z-coordinate components
from given and calculated orbital elements
void CoordinateTransformations()
{
    double x = r * ((Math.Cos(node) * Math.Cos(peri + v)) -
    (Math.Sin(node) * Math.Sin(peri + v) * Math.Cos(i)));
    double y = r * (Math.Sin(i) * Math.Sin(peri + v));
```

```
        double z = r * ((Math.Sin(node) * Math.Cos(peri + v)) +
        (Math.Cos(node) * Math.Sin(peri + v) * Math.Cos(i)));

        transform.localPosition = new Vector3((float) x, (float)
        y, (float) z);
    }
}
```

6.2 NEOManager.cs

This script is responsible for recording user input, querying the JPL Horizons API, and recording orbital element values. After carrying out these processes, NEOManager.cs instantiates a copy of the Orbiting Body prefab and attaches an instance of OrbitingBody.cs to said prefab.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.IO;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Networking;
using SimpleJSON;
using SimpleFileBrowser;
using TMPro;

public class NEOManager : MonoBehaviour
{
    // NEO prefab reference
    public GameObject neoPrefab;
    // Current targets menu item prefab reference
    public GameObject targetMenuItem;
    // Current targets menu reference
    public GameObject targetMenu;
    // Instantiated object parent referennce
    public GameObject neoParent;
    // UI input field reference
    public InputField targetInput;
    // API url
    private string url;
    // Resulting JSON from API request
    public JSONNode jsonResult;
    // Instance
    public static NEOManager instance;
    // GameObject lists
    List<GameObject> currentTargets = new List<GameObject>();
    List<GameObject> currentTargetsUI = new List<GameObject>();
    List<String> fileTargets = new List<String>();
    // Filebrowser reference
    public GameObject browserCanvas;

    void Awake()
    {
        // Set the instance to be this script
        instance = this;
```

```
}

// Start is called before the first frame update
void Start()
{

}

// Update is called once per frame
void Update()
{

}

// sends an API request - returns a JSON file and
instantiates NEO object
IEnumerator CreateNEO(string designation)
{
    // create the web request and download handler
    UnityWebRequest webReq = new UnityWebRequest();
    webReq.downloadHandler = new DownloadHandlerBuffer();

    // build the url and query
    webReq.url = string.Format("https://ssd-
    api.jpl.nasa.gov/sbdb.api?sstr={0}", designation);

    // send the web request and wait for a returning result
    yield return webReq.SendWebRequest();

    // convert the byte array and wait for a returning result
    string rawJson =
    Encoding.Default.GetString(webReq.downloadHandler.data);

    // parse the raw string into a json result we can easily
    read
    jsonResult = JSON.Parse(rawJson);

    // instantiate the object
    GameObject neoController = Instantiate(neoPrefab,
    neoParent.transform);
    neoController.name = designation + "Controller";
    GameObject neo = neoController.GetComponent<Transform>()
    .GetChild(0).gameObject;
    neo.name = designation;
    neo.GetComponent<SelectObjectInScene>().designation =
    designation;
    GameObject nameplate =
```

```
    neo.transform.GetChild(0).gameObject;
    nameplate.GetComponent<TextMeshPro>().text = designation;

    // access the OrbitingBody.cs script and set element
    values
    var orbitingBody = neo.GetComponent<OrbitingBody>();
    orbitingBody.e = jsonResult["orbit"]["elements"][0]
    ["value"];
    orbitingBody.a = jsonResult["orbit"]["elements"][1]
    ["value"];
    orbitingBody.i = jsonResult["orbit"]["elements"][3]
    ["value"];
    orbitingBody.node = jsonResult["orbit"]["elements"][4]
    ["value"];
    orbitingBody.peri = jsonResult["orbit"]["elements"][5]
    ["value"];
    orbitingBody.tp = jsonResult["orbit"]["elements"][7]
    ["value"];
    orbitingBody.n = jsonResult["orbit"]["elements"][9]
    ["value"];

    GameObject menuItem = Instantiate(targetMenuItem,
    targetMenu.transform);
    menuItem.GetComponent<UnityEngine.UI.Text>().text =
    designation;
    menuItem.GetComponent<TargetInfoManager>().designation =
    designation;
    menuItem.name = designation + "UI";
    currentTargets.Add(neoController);
    currentTargetsUI.Add(menuItem);
}

// Calls CreateNEO for batch loading
IEnumerator CreateTargets(string[] targets)
{
    for (int i = 0; i < targets.Length; i++)
    {
        yield return StartCoroutine(CreateNEO(targets[i]));
    }
}

// wrapper for UI interaction
public void AddTargets()
{
    // get the user input for target name
    string userInput = targetInput.text;
    // split user input by line and add to a list
```

```
        string[] targets = userInput.Split (new string[]
        {Environment.NewLine}, StringSplitOptions.None);
        // CreateTargets() call
        StartCoroutine(CreateTargets(targets));
    }

    // Removes instantiated target from user input
    public void RemoveTarget()
    {
        // get the user input for target name
        string designation = targetInput.text;
        GameObject neo = GameObject.Find(designation +
        "Controller");
        currentTargets.Remove(neo);
        Destroy(neo);

        GameObject menuItem = GameObject.Find(designation + "UI");
        currentTargetsUI.Remove(menuItem);
        Destroy(menuItem);
    }

    // Removes all instantiated targets to reset the scene
    public void RemoveAllTargets()
    {
        foreach (GameObject target in currentTargets)
        {
            Destroy(target);
        }

        currentTargets.RemoveAll(s => s == null);

        foreach (GameObject menuItem in currentTargetsUI)
        {
            Destroy(menuItem);
        }

        currentTargetsUI.RemoveAll(s => s == null);
    }

    // Opens file browser instance
    public void UploadFile()
    {
        browserCanvas.GetComponent<Canvas>().enabled = true;

        // Set filters
FileBrowser.SetFilters( true, new FileBrowser.Filter
        ( "Text Files", ".txt", ".csv" ), new FileBrowser.Filter(
```

```
        "Text Files", ".txt", ".pdf" ) );

// Set default filter that is selected when the dialog is shown
FileBrowser.SetDefaultFilter( ".csv" );

// Set excluded file extensions
FileBrowser.SetExcludedExtensions( ".lnk", ".tmp",
        ".zip", ".rar", ".exe" );

// Add a new quick link to the browser (optional)
FileBrowser.AddQuickLink( "Users", "C:\\Users", null );

// Opens file browser instance
StartCoroutine( ShowLoadDialogCoroutine() );
    }

    // Records user file input from file browser reference and
    calls CreateNEO() as necessary
    IEnumerator ShowLoadDialogCoroutine()
{
// Show a load file dialog and wait for a response from
        user
yield return FileBrowser.WaitForLoadDialog(
        FileBrowser.PickMode.FilesAndFolders, true, null, null,
        "Load Files and Folders", "Load" );

// Dialog is closed
        or cancelled the operation (FileBrowser.Success)

if( FileBrowser.Success )
{
// Print paths of the selected files
            (FileBrowser.Result) (null, if FileBrowser.Success is
            false)
for (int i = 0; i < FileBrowser.Result.Length; i++)
            {
                string destinationPath = Path.Combine(
                Application.persistentDataPath,
                FileBrowserHelpers.GetFilename(
                FileBrowser.Result[0] ) );
                FileBrowserHelpers.CopyFile(
                FileBrowser.Result[i], destinationPath );
                StreamReader strReader = new
                StreamReader(destinationPath);

                string line = String.Empty;
```

```
                while ((line = strReader.ReadLine()) != null)
                {
                    fileTargets.Add(line);
                    yield return StartCoroutine(CreateNEO(line));
                }
            }
        }
        browserCanvas.GetComponent<Canvas>().enabled = false;
}
}
```

6.3 TimeManager.cs

This script keeps track of the current time value used to calculate in-scene object positions. It contains functions capable of manipulating both the time value itself and the rate at which said value increases or decreases.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;


public class TimeManager : MonoBehaviour
{
    public InputField dateInput;
    public Text currentJD;
    public Text currentCD;
    double julianDate;
    public bool pauseTime = false;
    public double time;
    public DateTime calendarDate;
    public double scaleTime = 0.2;
    public bool reverseTime = false;


    // Start is called before the first frame update
    void Start()
    {
        GetCurrentTime();
    }

    // Update is called once per frame
    void Update()
    {
        DisplayCurrentJD();
```

```
    DisplayCalendarDate();
    ManageTime();
}

public void DisplayCurrentJD()
{
    currentJD.text = time.ToString();
}

public void DisplayCalendarDate()
{
    double OADate = (time - 2415018.5);
    calendarDate = DateTime.FromOADate(OADate);
    currentCD.text = calendarDate.ToString();
}
public static double ToJulianDate(System.DateTime date)
{
    return date.ToOADate() + 2415018.5;
}

public void GetCurrentTime()
{
    time = ToJulianDate(System.DateTime.Now);
}
public void ManageTime()
{
    if (!pauseTime)
    {
        time += scaleTime;
    }
}

public void PauseTime()
{
    if (!pauseTime)
    {
        pauseTime = true;
    }
    else
    {
        pauseTime = false;
    }
}

public void ReverseTime()
{
```

```
        Toggle timeToggle = GameObject.Find("Reverse Time").GetComponent<Toggle>();

        scaleTime *= -1;
        if (timeToggle.isOn)
        {
            reverseTime = true;
        }
        else
        {
            reverseTime = false;
        }
    }

    // Used by the Scale Time UI Slider
    public void OnValueChanged(float scale)
    {
        if (reverseTime)
        {
            scaleTime = scale * -1;
        }
        else
        {
            scaleTime = scale;
        }
    }

    public void ManualJulianDate()
    {
        julianDate = Double.Parse(dateInput.text);
        Debug.Log(julianDate);
        time = julianDate;

    }
}
```

References

[1] T. Mason, *Vera c. rubin observatory science goals*, https://www.lsst.org/science.

[2] N. M. BALL and R. J. BRUNNER, "DATA MINING AND MACHINE LEARNING IN ASTRONOMY", International Journal of Modern Physics D **19**, 1049–1106 (2010).

[3] K. Kobold, *A small orrery showing earth and the inner planets*, https://www.flickr.com/photos/kaptainkobold/127601212/sizes/m/.

[4] S. Bamfaste, *Reality check*, https://www.esa.int/ESA_Multimedia/Images/2017/07/Reality_check.

[5] *12 must-have tools for android development*, https://lvivity.com/tools-for-android-development.

[6] *Neo basics*, https://cneos.jpl.nasa.gov/about/basics.html.

[7] *Neo basics: neo groups*, https://cneos.jpl.nasa.gov/about/neo_groups.html.

[8] *Didymos—the ideal target for dart's mission*, https://dart.jhuapl.edu/Mission/index.php.

[9] J. Uri, *175 years ago: astronomers discover neptune, the eighth planet*, https://www.nasa.gov/feature/175-years-ago-astronomers-discover-neptune-the-eighth-planet.

[10] C. A. Trujillo, "A sedna-like body with a perihelion of 80 astronomical units", Nature **507**, 471–474 (2014).

[11] *Glossary*, https://ssd.jpl.nasa.gov/glossary/.

[12] P. Lum, *Untitled goose game review - a honking good time*, https://www.theguardian.com/games/2019/sep/23/untitled-goose-game-review-a-honking-good-time.

[13] *Small-body database lookup*, https://ssd.jpl.nasa.gov/tools/sbdb_lookup.html#/?sstr=433.

[14] yasirkula, *Unity simple file browser*, https://github.com/yasirkula/UnitySimpleFileBrowser.