

Solar System Survey

A COS 426 project by Zak Dasaro and Jason Kim

Abstract

The goal of this project was to create a graphical representation of the solar system, including its planets, major moons, and other prominent objects like dwarf planets, asteroids, and comets. In this program, planets are accurately positioned in the simulation according to their orbital data, so the user can see an accurate spatial layout of the solar system. Celestial bodies are rendered with great visual accuracy, using available textures and carefully tuned reflection models. The user may move the view throughout the solar system, zoom in on different objects, and look around using mouse controls.

Introduction

Goal

The primary goal of this project was to graphically model the solar system in a way that's physically accurate in scale, size, appearance, and location of all the celestial bodies. In addition, the goal was to create something visually appealing and visually accurate. The benefit is for users to learn about the positions and behavior of celestial bodies, the orbits that they take through the solar system, and appreciate the enormous scale of everything in the solar system.

Previous Work

A COS 426 project from last year, <https://shamailah.github.io/COS426-final-project/>, produced a representation of the solar system and simulated the newtonian forces of gravity. We saw an opportunity to greatly expand upon the features of that project, by improving photorealism, adding better camera controls, and improving the realism of the scale of the celestial bodies. In addition, based on the relatively slow and limited performance of this solar system model, we aimed to create a simulation that is much more computationally efficient and able to render all the major solar system objects and their movement while maintaining an acceptably high FPS.

A 3D graphics studio known as Plus 360 Degrees created a photorealistic rendering of Earth, <http://earth.plus360degrees.com/>, complete with the sun in the background and a backdrop of distant stars. We were inspired by this realistic look to replicate it for the rest of the solar system.

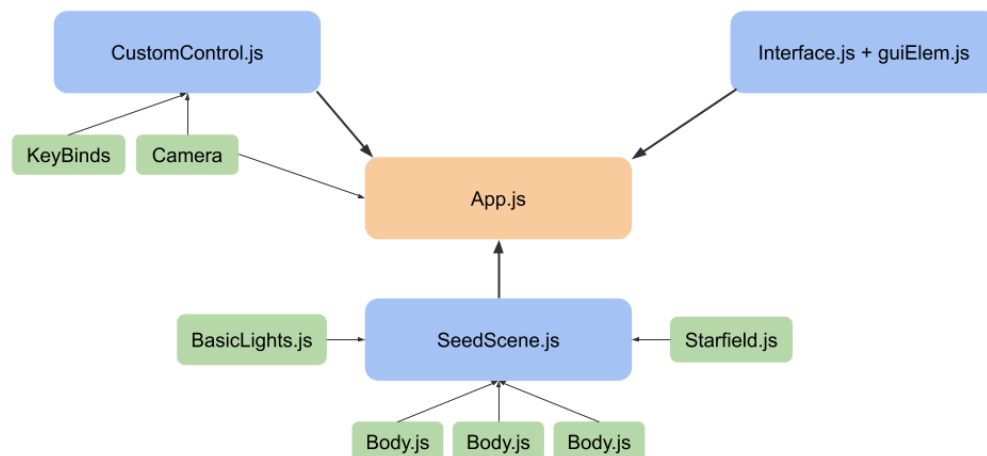
A desktop application named Celestia (<https://celestia.space>), is an impressively accurate and efficient representation of not only the solar system, but the galaxy. We want to be able to replicate the accuracy of the sizes/positioning of objects in Celestia in addition to the relatively

efficient performance of the application while improving the intuitive nature of the camera controls, accuracy of the planet/moon models, and the accuracy of the background starfield.

Approach

We decided to implement our Solar System simulation in the form of a ThreeJS web application.

The structure of the application centers around `app.js` which provides a loading screen while the scene and its objects are loaded. In addition, `app.js` initializes the camera and set of custom controls (these custom controls are used to allow the camera to re-center on any object in the solar system and not just the origin). The scene, which is implemented in `SeedScene.js` which loads in all the celestial objects. Every object is implemented through `Body.js`. `Body.js` is responsible for calculating and storing the orbit of an object and also contains helpful methods which control the position, textures, orbital path and other features. `SeedScene.js` loads an accurate starfield, manages which objects are rendered and which objects are not rendered, and also provides a GUI for manipulating time (pausing, speeding up, slowing down, going back in time, choosing an exact date, etc).

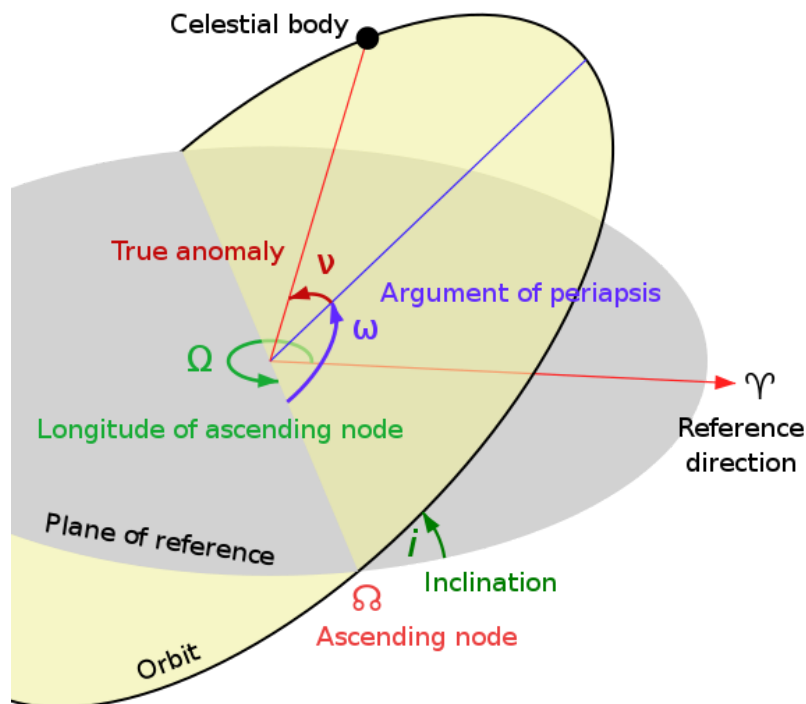


Methodology

The Physics Simulation

Any orbit can be uniquely characterized by six parameters or “orbital elements” that describe the shape and orientation of the ellipse that makes up the orbit. The parameters are: eccentricity, semimajor axis, inclination, longitude of ascending node, argument of periapsis, and mean anomaly. The eccentricity determines the shape of the ellipse (how elliptical or circular the

ellipse is) and is usually a parameter between 0 and 1. The semimajor axis, usually expressed in astronomical units (1 Astronomical Unit (AU) = 1.496e11 meters) determines the size of the orbit and is equal to half of the longer axis of the ellipse. The inclination is an angle that determines the vertical tilt of the ellipse with respect to some reference plane. Usually this plane will be the ecliptic (the plane determined by Earth's orbit around the sun) which is usually assumed as the "plane of the solar system". For moons however, this plane usually refers to the plane perpendicular to the planet's axis. The longitude of ascending node horizontally orients the ellipse. The ascending node refers to the intersection between the ellipse and its reference plane where the celestial body's movement is "ascending" above the plane. The longitude of the ascending node refers to the angle between this node and the direction of the Vernal Equinox. The argument of periapsis (or argument of perihelion when talking about objects orbiting the Sun), defines the orientation of the ellipse within its orbital plane. Specifically it refers to the angle between the ascending node to the periapsis (perihelion), which is the closest point in the orbit between the object and the object it orbits. Mean anomaly is a "mathematically fictitious angle" which varies linearly with time. This parameter helps us to calculate the exact position of the object along its orbit at any time by converting it to True anomaly. True anomaly defines the angle of the object along its orbit at any time from periapsis. (cite: https://en.wikipedia.org/wiki/Orbital_elements)



Based on these parameters, when initializing any celestial body, we pre-calculate the positions of the object along its orbit with set time-steps. We store all these positions as Vector3 objects in an array so that we don't have to ever re-compute these positions. We also implemented a

function that takes in any time (expressed in Gregorian Days) and fetches a position from the positions array that corresponds to the object's accurate position along its orbit at that time. This was a major step in making sure that our application was computationally efficient. True newtonian gravitational simulation is usually a n -squared algorithm, but with this optimization, we have simplified the algorithm to a constant time array access operation.

Accurately Rendering Celestial Bodies

Each celestial body orbiting the sun can be represented as a Phong material. However, different parameters are appropriate for different objects, due to differences in physical properties. For instance, many moons tend to be very reflective, and gas planets generally have less intense shine. To attain the most physically accurate representation, the reflection properties of each object were manually tuned so that the render matched actual photographs as closely as possible.

Each spherical body was represented as a texture map applied to a sphere. A common pitfall of this approach is what's known as pole pinch, where the poles of the sphere are visually distorted as if the texture is being pinched to a point. One way to avoid pole pinch would be to implement cube mapping, where 6 textures (one for each side of a cube) are mapped to the surface of the sphere. Thus, there would be no single point of distortion. However, this would require finding cube maps for all the celestial bodies, and it is much more common to find equirectangular texture maps. As a compromise, pole pinching was mitigated by changing the minification filter. By default, THREE.js uses a linear filter, where each point is colored based on a linear interpolation of its 4 neighboring pixels. By changing the filter to a nearest filter, where each point receives the color of its one nearest pixel, the pole pinch is significantly alleviated.

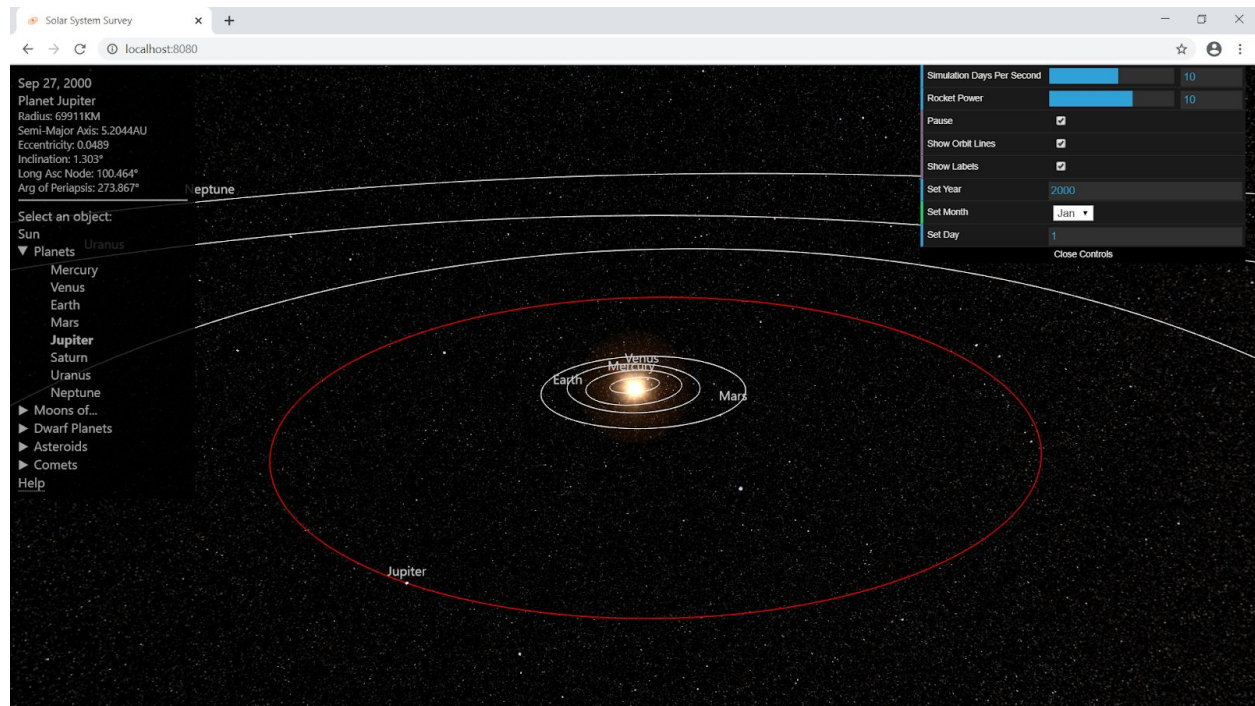
The choice of minification filter is also relevant when rendering the celestial background. With a linear filter, the stars tend to be blurred. However, with a nearest filter, the stars are sharper. The downside is that the rendering may not be consistent across screens of different resolutions, because small stars become invisible on low-resolution displays, but the sharper image was worth the tradeoff.

User Interface

On the left side of the screen is an interface which shows the simulation date and provides a menu for selecting an object to focus on. It also displays orbital parameters for the currently selected object. Upon selecting an object, its orbit will be highlighted in red. Pressing the F key will reposition the camera so that it is centered on the object. There is also a help menu which can be accessed from this interface.

As mentioned in our approach, we implemented custom camera controls. These allow the user to rotate the camera view by dragging the mouse, zoom in and out with the scroll wheel, and pan and move with the arrow keys and WASD and spin the camera using L and K. There is also an interface on the right which allows the user to manipulate time parameters, as mentioned

earlier. The screenshot below shows the full user interface. Lastly, there is an image capture feature that the user can activate by pressing the I key.



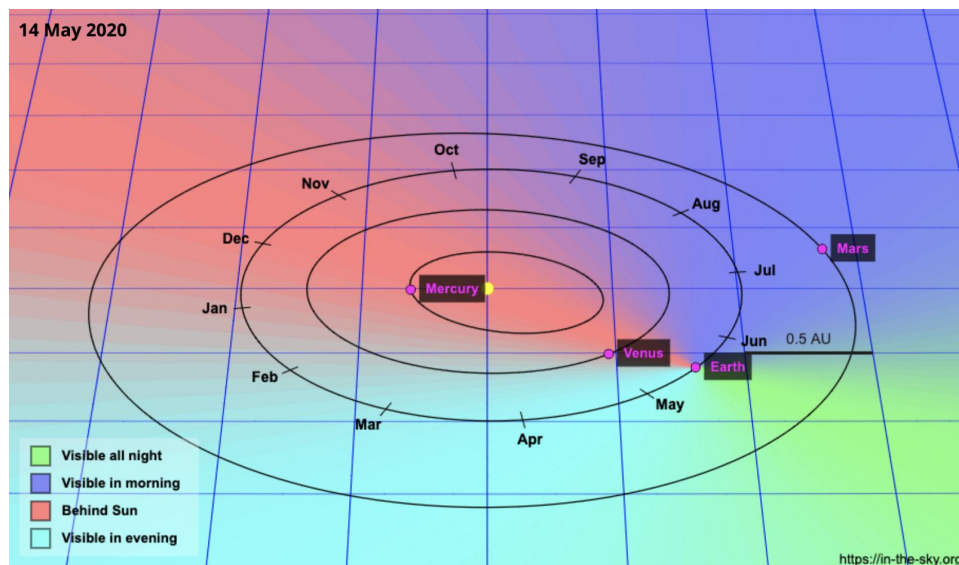
Additional Optimizations:

As we began to realize just how large and massive the solar system is, we quickly determined that users were unlikely to see any objects (except perhaps planets) unless the objects were intentionally focused on. Most objects aside from planets, even relatively large moons or dwarf planets, are so small compared to the scale of the solar system that it is impossible to visually see them unless the user intentionally zooms up on them. Thus, we decided that it was not necessary to actually render all solar system objects at once. Instead, we only render the 8 planets at all times (since these can actually be seen with the naked eye even from relatively far distances). However, other objects like moons, dwarf planets, asteroids, and comets are not rendered or simulated at all unless the user selects that object or, in the case of moons, if the user selects the planet that the moon is orbiting. For example, Jupiter's moons will not be rendered or simulated unless a user selects one of the moons of Jupiter or if a user selects Jupiter. Once a user moves on and selects another object, these previous objects are de-loaded. This helps to improve performance by limiting the actual number of objects being rendered and also limiting the number of objects that are being positionally simulated at one time.

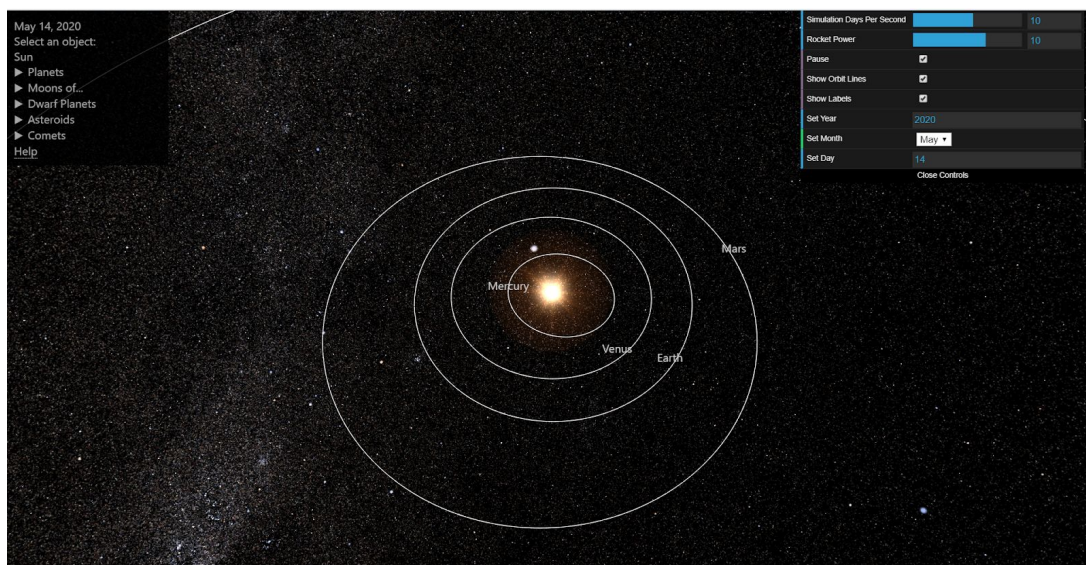
Results

Our primary goal was to make an accurate simulation of the solar system. Thus, we did a few experiments to try to show how accurate our planets/celestial bodies are and how accurate our physical simulation is:

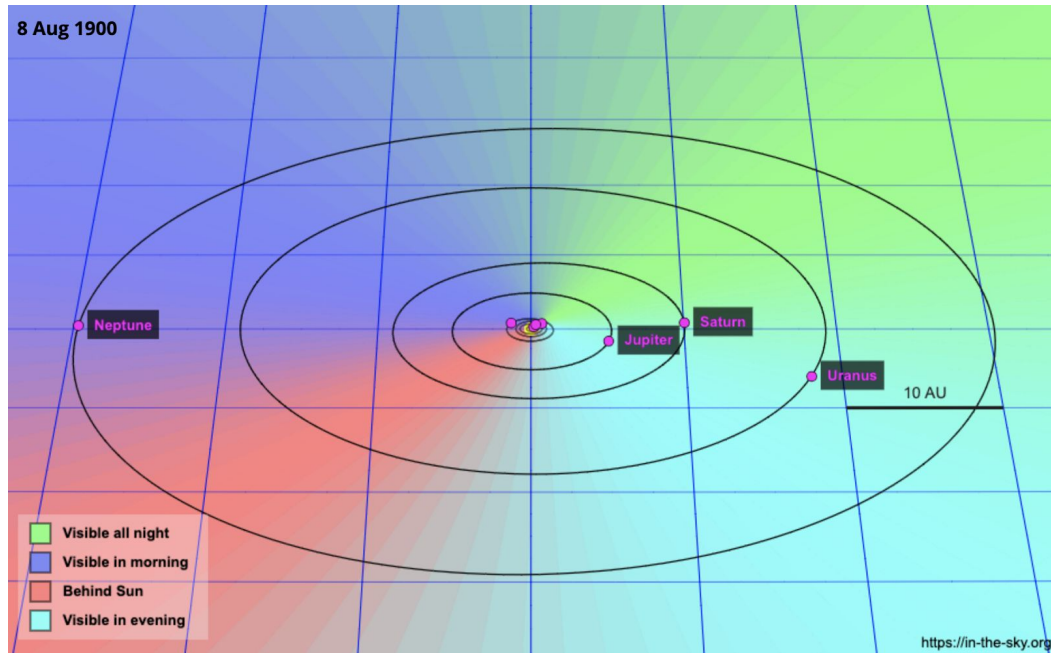
Below is an image from <https://in-the-sky.org/solarsystem.php> showing the accurate positions of the inner planets of the solar system on May 14, 2020:



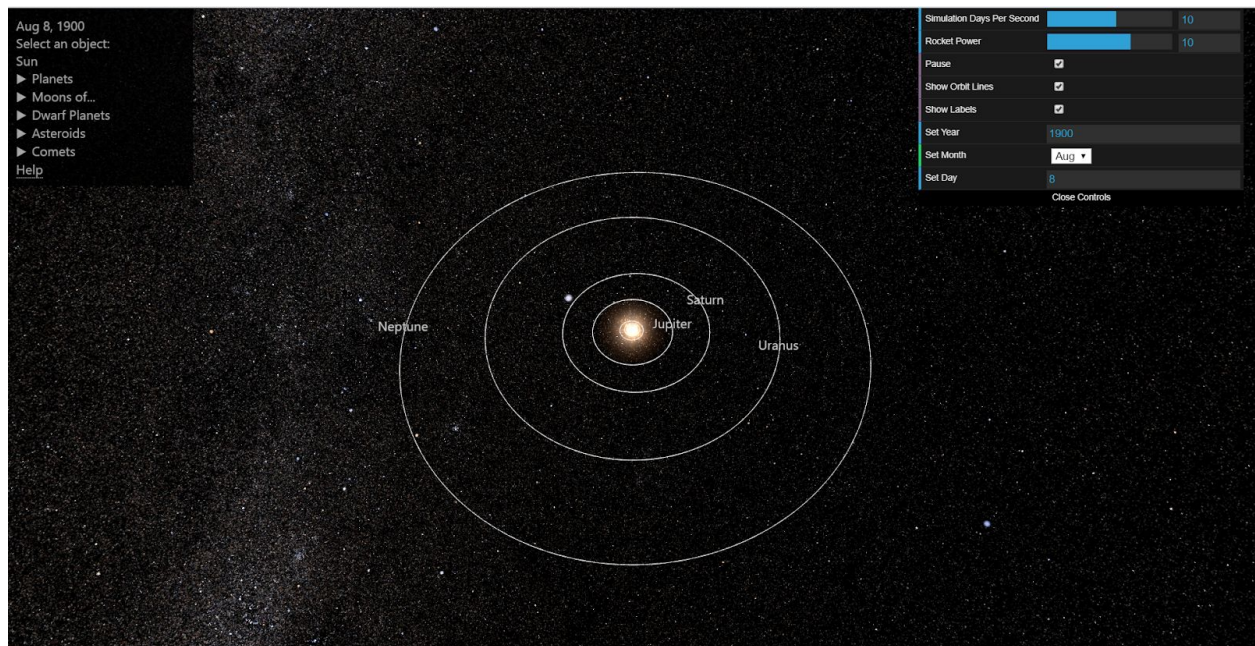
As can be seen in this image, our simulation accurately positions the planets as expected when the time is set May 14, 2020:



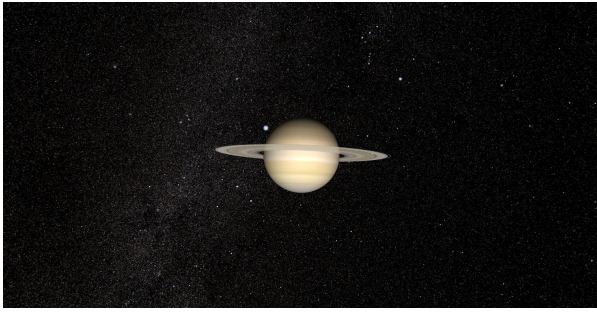
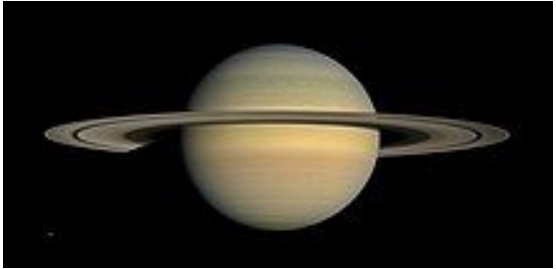
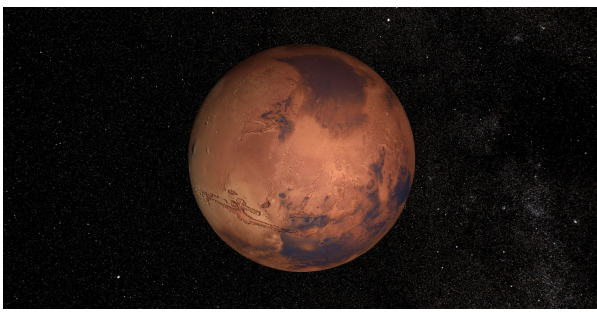

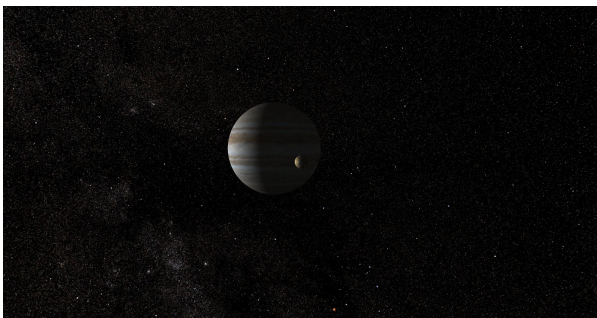
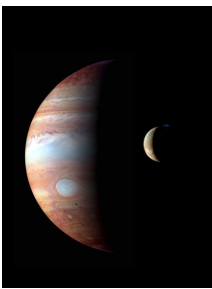
As a second test, we wound the clock back to August 8, 1900 and matched the outer planets:

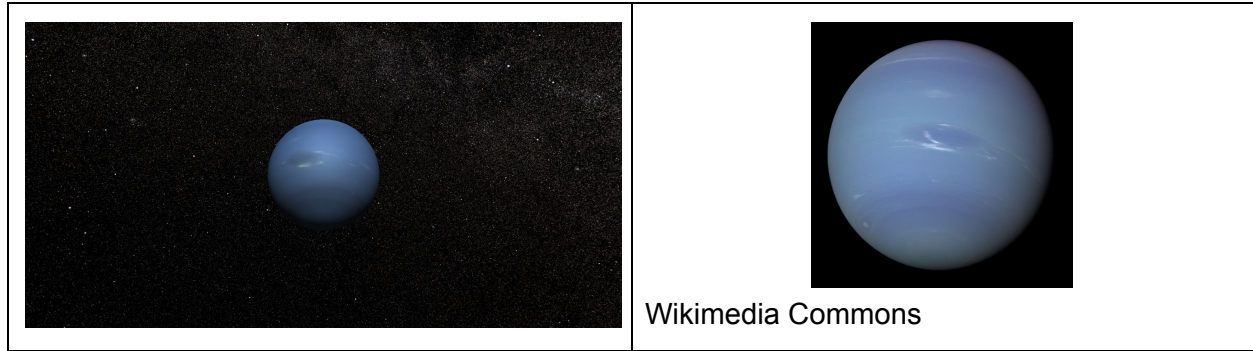


Just as before, our simulation accurately matches the positions of the planets, even more than a century in the past:



To evaluate our rendering results, we compare renderings produced by this program to actual photographs to verify their realism. Below are some of these results.

Renders	Real Photos
	 Wikimedia Commons
	 Wikimedia Commons
	 https://www.space.com/16419-io-facts-about-jupiters-volcanic-moon.html



From top to bottom: Saturn, Mars, Jupiter with Io, and Neptune

Discussion and Conclusion

Overall, our approach was very successful. Our application runs relatively smoothly with high performance while maintaining the accuracy of the simulation. While our approach is sufficient, it is possible that a drastically different approach, where the camera remains stationary and the environment moves to simulate the movement of the camera, could be better if we wanted to achieve even more ambitious aims. Especially in applications like Celestia, which provides a simulation of the entire galaxy, it is impossible to represent an environment in accurate scale at the extreme size of a galaxy. Instead, a limited environment is rendered and moves around the stationary camera.

Again, ultimately, however, our implementation approach was successful for our goals and needs. Some followup work that could be done is the addition of even more minor moons, comets, and asteroids, but this is limited by the 3D models and textures available online. In addition, some optimizations could be made to simplify the rendering of planets far away (though this is not really necessary for performance since we are already limiting the number of objects we render at one time).

Lessons Learned

During this project, we learned a lot about Three.js and how to manipulate the movement of objects in a scene, how cameras and lights work within Three.js and how to use event handlers. In addition, we learned a lot about how to apply textures and manipulate materials to make objects (in this case different planets/moons/etc) look realistic without distortion. In dealing with such large scales, we learned about rendering limitations related to the depth of the scene, namely the precision issue that could result from rendering a scene with such a large depth.

Works Cited

Projects of inspiration:

<https://shamailah.github.io/COS426-final-project/>

<http://earth.plus360degrees.com/>

<https://celestia.space>

Sources for creating orbital data:

https://en.wikipedia.org/wiki/Orbital_elements

<ftp://ssd.jpl.nasa.gov/pub/eph/planets/ioms/ExplSupplChap8.pdf>

https://ssd.jpl.nasa.gov/?sat_elem

Sources of textures and 3D models (as explained in src/img/sources.txt):

<http://www.celestiamotherlode.net/>

<https://www.solarsystemscope.com/textures/>

<https://solarsystem.nasa.gov/resources>

<https://github.com/mrdoob/three.js/tree/master/examples/textures/lensflare>

<https://planet-texture-maps.fandom.com/>

<http://celestia.simulatorlabbs.com/CelSL/textures/medres/>

<https://svs.gsfc.nasa.gov/3572>

<https://3d-asteroids.space/>

<https://in-the-sky.org/solarsystem.php>