Title: An Interactive 2D Solar System

List of Group Members and Contributions:

Group No: 07

Name	ID
MD MAHRAB HASAN CHOWDHURY	21-45877-3
FUAD HASAN	23-51947-2
ANIS MAHMUD RATUL	23-51922-2
SHAFAYAT KHAN SHAAN	23-52134-2

MD MAHRAB HASAN CHOWDHURY: The Architect (Core Engine & Rendering Pipeline)

- **Contribution:** This member was responsible for setting up the main program structure in main.cpp. They managed the core rendering loop in render.cpp, making sure all the different parts of the project were drawn to the screen in the correct order. They also handled the creation of the 2D "camera" using the gluOrtho2D function.

FUAD HASAN: The Navigator (Camera, Interaction & HUD)

- **Contribution:** This member focused on how the user interacts with the solar system, handling all keyboard and mouse inputs in camera.cpp. They implemented the controls for panning, zooming, and changing the simulation speed. They were also in charge of creating the on-screen Heads-Up Display (HUD) that shows useful information to the user.

ANIS MAHMUD RATUL: The Astronomer (Planets, Moons & Orbits)

- **Contribution:** This member brought the solar system to life. They were responsible for the logic in planets.cpp, which handles the movement and drawing of the sun, all the planets, and their moons. This involved using math to calculate their orbital paths and rendering them as 2D shapes, including the beautiful rings of Saturn.

SHAFAYAT KHAN SHAAN: The Universe Designer (Data Structures & Environment)

- **Contribution:** This member was responsible for the project's blueprint and background. They managed the common.h file, which holds the shared data structures like the Planet and Star structs that connect all parts of the project. They also created the starfield environment in stars.cpp, which generates and draws the thousands of stars that give the project a sense of a vast universe.

Introduction:

The universe has always been a source of wonder for humans. From ancient astronomers charting the stars to modern space exploration, our desire to understand the cosmos is endless. However, the vastness of space and the complex movements of celestial bodies can be difficult to grasp from textbooks alone. Static images and diagrams can only show a single moment in time and fail to capture the dynamic, ever-changing nature of our solar system. This is where computer graphics comes in as a powerful tool for education and exploration. By creating a visual and interactive model, we can make learning about space more engaging, intuitive, and fun.

This project, "An Interactive 2D Solar System," is our attempt to harness the power of computer graphics to create an educational tool. Our main goal is to build a simple, easy-to-use simulation of our solar system. Instead of aiming for a complex 3D model, which can be difficult to navigate, we chose a 2D top-down view. This makes it easy for anyone to see the entire solar system at a glance, understand the orbits of the planets, and appreciate the different scales involved. The project is not just a passive animation; it is an interactive experience. Users can take control of the simulation in many ways. They can pan across the vastness of space, zoom in to get a closer look at a specific planet, or zoom out to see the entire system. They can pause time to study a particular arrangement of planets or speed it up to watch years fly by in seconds. This level of interaction turns the user from a simple observer into an explorer.

To build this project, we used the C++ programming language, which is known for its performance and power, making it a great choice for graphics applications. For the graphics themselves, we used the OpenGL library. OpenGL is a widely-used, cross-platform API (Application Programming Interface) for rendering 2D and 3D graphics. It gives us direct control over the graphics hardware, allowing us to draw shapes, set colors, and create the visual elements of our solar system. To make working with OpenGL easier, we also used the GLUT (OpenGL Utility Toolkit). GLUT helps with the simple but tedious tasks of creating a window, managing user input from the keyboard and mouse, and setting up the main loop that keeps our animation running smoothly. The combination of C++, OpenGL, and GLUT provides a solid foundation for creating a simple yet effective computer graphics project like this one.

This report will walk you through our entire project. We will start by looking at the "Background of the Problem," where we discuss why such a project is useful and the challenges involved in visualizing something as massive as a solar system. Next, in the "Methods Used" section, we will dive into the technical details of our implementation. We will explain how we structured our code, how we used mathematical formulas to calculate the planetary orbits, and how we handled user input to make the simulation interactive. Following that, the "Feature Set" section will provide a complete list of all the things a user can do with our project, from basic navigation to controlling the flow of time. We will then look to the future in our "Future Directions" section, where we will brainstorm ideas for how

this project could be expanded and improved. Finally, we will wrap everything up in the "Conclusion," where we will reflect on what we have achieved and what we learned during the process of creating this interactive solar system. We hope that this project serves as a clear demonstration of our understanding of computer graphics principles and as a useful tool for anyone curious about the celestial neighborhood we call home.

Background of the Problem:

The challenge of understanding our solar system is as old as human curiosity itself. For thousands of years, people have looked to the skies and tried to make sense of the patterns they saw. Early models were simple, often placing the Earth at the center of the universe. These models were based on what could be seen with the naked eye and fit the philosophical beliefs of the time. However, as our tools and understanding grew, so did the complexity of our models. The invention of the telescope was a turning point, allowing astronomers like Galileo to see that the planets were not just wandering stars, but worlds in their own right, with their own moons. This led to the acceptance of the Sun-centered, or heliocentric, model of the solar system, which is the one we use today.

While this model is accurate, it presents a significant challenge when it comes to visualization. The main problem is one of **scale**. The distances and sizes in our solar system are almost too large to imagine. For example, the Sun is so big that you could fit over a million Earths inside it. The distance from the Earth to the Sun is about 150 million kilometers, but the distance to Neptune, the farthest planet, is about 4.5 billion kilometers. If you were to create a physical model where the Earth was the size of a small marble, the Sun would have to be about 1.5 meters wide, and Neptune would need to be placed more than 5 kilometers away! This makes creating accurate physical models in a classroom or museum nearly impossible. Most models we see have to make compromises, either by shrinking the distances between planets or by making the planets much larger than they should be relative to their orbits. These compromises can give people a wrong idea about how empty space really is and how far apart the planets are.

Another problem is **motion**. The solar system is not static; it is in constant motion. Each planet is traveling at a different speed in its orbit around the Sun. Mercury, the closest planet, completes its orbit in just 88 Earth days, while Neptune takes about 165 years. Trying to understand these different orbital periods and how the planets align over time is very difficult with static pictures in a book. A book can show you a snapshot of the solar system, but it cannot show you the dance of the planets over months, years, or centuries. This is a dynamic problem, and it requires a dynamic solution. We need a way to see how the planets move in relation to each other and to appreciate the rhythm of the cosmos.

This is where computer graphics becomes an invaluable tool. A computer simulation can overcome the problems of scale and motion in a way that physical models and static images cannot. With a computer program, we can represent the vast distances of the solar system on a single screen. We can use mathematical formulas to model the orbits of the planets with great accuracy. Most importantly, we can introduce the element of time. A simulation allows

us to play, pause, and speed up time, making it possible to observe long-term cosmic events in a matter of seconds.

The goal of our "Interactive 2D Solar System" project is to provide a solution to these problems. We aimed to create a tool that is both educational and engaging. By choosing a 2D top-down perspective, we make the entire solar system immediately understandable. There is no complex 3D camera to get lost with; the user can see everything at once. This perspective is excellent for showing the layout of the planets and their orbits. To solve the problem of scale, we implemented zoom functionality. The user can zoom out to appreciate the vast distances between the outer planets or zoom in to see the details of the inner solar system, including Earth and its moon. To solve the problem of motion, our project animates the orbits in real-time. The planets move at their relative speeds, giving the user an intuitive feel for their different orbital periods. The ability to control the speed of time is perhaps the most powerful feature. A student can speed up the simulation to see how many times Mercury orbits the Sun in the time it takes Jupiter to complete just one orbit.

In summary, the problem we are addressing is the difficulty of accurately and intuitively visualizing the solar system. The vast scales, the constant motion, and the long time periods involved make it a perfect challenge for a computer graphics solution. Our project tackles this challenge by creating an interactive, 2D simulation that allows users to explore our solar system in a way that is simply not possible with traditional educational materials. It aims to turn an abstract concept into a tangible, explorable virtual world.

Methods Used:

To build our interactive solar system, we used a combination of programming techniques and graphics libraries. Our approach was to break the project down into smaller, manageable parts: the core rendering engine, the planet simulation, the background environment, and the user interaction system. All these parts work together to create the final application.

1. Core Engine and Rendering Pipeline: The foundation of our project is built in C++ using the OpenGL and GLUT libraries. The main entry point of our application is in the `main.cpp` file. This file is responsible for initializing GLUT, setting up the display mode (we use `GLUT_DOUBLE` for smooth, flicker-free animation), and creating the window where our solar system will be drawn. Once the window is created, we register several "callback" functions. These are functions that GLUT will call automatically when certain events happen. For example, `glutDisplayFunc(display)` tells GLUT to call our `display` function whenever the screen needs to be redrawn. Similarly, we register functions for handling keyboard and mouse input. The final call in `main` is `glutMainLoop()`, which starts the application and continuously listens for events.

The heart of the rendering is the `display()` function, located in `render.cpp`. This function is called many times per second. Its first job is to clear the screen to a dark blue color, representing the void of space, using `glClear()`. It then calculates the time that has

passed since the last frame (`dt`), which is crucial for making our animations smooth and independent of the computer's speed. The most important part of this function is setting up the 2D view. We use `gluOrtho2D()` to create a 2D coordinate system. This function defines the visible area of our world, and we dynamically adjust it based on the `zoomFactor` and the `viewCenterX`/`viewCenterY` variables, which allows the user to pan and zoom. After setting up the view, the `display()` function calls the drawing functions for each part of the scene in order: `drawStars()`, `drawSun()`, `drawPlanets()`, and `drawHUD()`. This layered approach ensures that the stars are in the back, the planets are in the middle, and the user interface is on top.

2. Data Structures and Shared Information: To keep our code organized, we created a central header file called `common.h`. This file is the blueprint for our universe. It defines the main data structures, `Star` and `Planet`. The `Planet` struct is particularly important; it holds all the information about a planet, such as its name, radius, orbit radius, color, and speed, and even information about its moon if it has one. This file also declares all the global variables that need to be shared across different files. For example, the `timeScale` variable, which controls the speed of the simulation, is declared in `common.h` so that both the user input code in `camera.cpp` and the animation code in `planets.cpp` can access and modify it. This central file prevents us from having to pass many variables between functions and makes the project easier to manage.

3. Simulating and Drawing the Planets: The logic for the celestial bodies is in `planets.cpp`. The file starts by defining the data for all eight planets in an array of `Planet` structs. The main function here is `drawPlanets()`. Inside this function, we loop through every planet in the array. For each planet, we first update its position. If the simulation is not paused, we increase its `orbitAngle` based on its `orbitSpeed` and the time passed (`dt`). The actual position of the planet is then calculated using basic trigonometry. The x-coordinate is found using `p.orbitRadius * cosf(deg2rad(p.orbitAngle))`, and the y-coordinate is found using `p.orbitRadius * sinf(deg2rad(p.orbitAngle))`. This is the classic mathematical method for finding a point on a circle.

Once we have the position, we use OpenGL's transformation functions (`glTranslatef`) to move our drawing cursor to that spot. Then, we draw the planet itself. To draw a filled 2D circle, we wrote a helper function called `drawCircle()`. This function uses the `GL_TRIANGLE_FAN` primitive in OpenGL, starting from a center point and drawing triangles out to a series of points along the circle's edge. This is an efficient way to draw a solid circle. We also draw the orbit of each planet as a thin line using the `GL_LINE_LOOP` primitive, again using trigonometry to calculate the points around the circle. The logic for drawing moons is very similar, but it is done within the transformation of its parent planet.

4. User Interaction and Camera Control: All the code for handling user input is located in `camera.cpp`. The `keyboard()` function is set up to handle key presses for things like pausing the simulation (' '), zooming ('W' and 'S'), and panning ('A', 'D', 'Q', 'E'). The `mouse()` function detects mouse clicks, which is used to start panning, and it also handles the mouse wheel for zooming. The `motion()` function is called when the mouse is moved while a button is held down. Inside this function, we calculate how far the mouse has moved since the last frame and update the `viewCenterX` and `viewCenterY` variables. This gives the effect of dragging the entire scene. A key feature here is the `focusIndex`. When the user pans with the mouse or keyboard, we set `focusIndex` to a special value (`FOCUS_MANUAL_PAN`). In the `display()` function, if the focus is on a planet, the view center is automatically updated to that planet's position. But if the focus is set to manual, the view center is not updated, allowing the user's panning to take effect.

5. Creating the Environment: To avoid having our solar system float in a plain, dark background, we created a starfield. The code for this is in `stars.cpp`. The `initStars()` function is called once at the beginning of the program. It loops `STAR_COUNT` times (which is set to 8000 in `common.h`) and generates a random x and z coordinate for each star. These coordinates are spread out over a large area. The `drawStars()` function then loops through this array of stars and draws each one as a single point using OpenGL's `GL_POINTS` primitive. This is a very simple and efficient way to create a convincing background that adds a great sense of depth and scale to the scene.

Feature Set:

Our Interactive 2D Solar System project is equipped with a rich set of features designed to provide an engaging and educational user experience. These features allow the user to explore and control the simulation in various ways, transforming a simple animation into an interactive virtual laboratory. The features can be broadly categorized into visualization, simulation control, and camera interaction.

1. 2D Solar System Visualization: The primary feature of the project is the clear and intuitive 2D top-down visualization of our solar system.

- **Accurate Planetary Layout:** The project displays the Sun and all eight planets of our solar system: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune. The planets are placed in their correct order from the Sun, and their orbits are drawn to scale relative to each other.
- **Sun, Planets, and Moon:** The Sun is rendered as a large, bright yellow circle at the center of the system. Each planet is drawn as a colored circle with a size relative to the other planets. The Earth is specially featured with its own moon, which orbits the planet.

- **Saturn's Rings:** To add more visual detail, Saturn is realistically depicted with its iconic ring system. We created a special function, `drawRing()`, to render the rings as a flat, 2D object around the planet.
- **Toggleable Orbits:** The orbital paths of the planets are drawn as grey circles, making it easy to see the path they follow. The user can toggle these orbits on and off by pressing the 'O' key, allowing for a clearer view of the planets themselves if desired.

2. Dynamic Animation and Simulation Control: The project is not a static image but a dynamic, running simulation of the planets' movements.

- **Real-Time Orbital Animation:** All planets orbit the Sun at their own relative speeds, as defined in the `planets.cpp` file. This provides a visual representation of how inner planets move much faster than the outer ones. The Earth's moon also orbits the Earth in real-time.
- **Simulation Speed Control:** The user has full control over the flow of time in the simulation. By pressing the **Up Arrow key**, the user can speed up time, and by pressing the **Down Arrow key**, they can slow it down. The current speed multiplier is shown on the HUD. This feature is powerful for observing long-term astronomical patterns.
- **Pause and Play:** The simulation can be paused at any moment by pressing the **Spacebar**. This freezes the entire system, allowing the user to examine the current positions of all planets. Pressing the Spacebar again will resume the simulation. The HUD clearly indicates whether the simulation is "Running" or "Paused".

3. Interactive Camera and Navigation: To make the solar system feel explorable, we implemented a flexible and user-friendly camera system.

- **Zoom Functionality:** The user can zoom in and out to change the scale of the view. This can be done by using the **W and S keys** or by **scrolling the mouse wheel**. This allows the user to get a close-up view of the inner planets or zoom out to see the entire solar system, all the way to Neptune's orbit.
- **Panning (Manual Camera Movement):** The user can freely move the camera's view around. This can be done in two ways:
    - **Mouse Drag:** By **clicking and dragging with the left mouse button**, the user can pan the view around the scene.
    - **Keyboard Panning:** The **A, D, Q, and E keys**, as well as the **Left and Right Arrow keys**, can be used to pan the camera horizontally and vertically.
- **Focus-Locking:** This feature allows the camera to automatically follow a selected celestial body. The user can press the number keys **'1' through '8'** to lock the focus onto the corresponding planet (1 for Mercury, 2 for Venus, etc.). Pressing **'0'** locks the focus onto the Sun. When the camera is locked on a planet, the view will automatically move to keep that planet in the center of the screen. Panning with the mouse or keyboard will break this focus and switch the camera to "Manual Pan" mode.

4. Environment and User Interface: To enhance the user experience, we added environmental details and an informative on-screen display.

- **Procedural Starfield:** The background is not just a solid color. It is filled with 8,000 stars that are randomly generated every time the program starts. This creates a beautiful and dynamic backdrop that makes the scene feel more like deep space.
- **Toggleable Stars:** Just like the orbits, the starfield can be turned on or off by pressing the **'N' key**. This can be useful if the user wants to focus solely on the planets without the distraction of the background.
- **Informative Heads-Up Display (HUD):** A semi-transparent information panel is displayed in the top-left corner of the screen. This HUD provides essential information at a glance, including:
    - The current simulation speed.
    - Whether the simulation is paused or running.
    - The current camera focus (e.g., "Focus: Earth" or "Focus: Manual Pan").
    - A handy list of all the keyboard and mouse controls.
- **Reset Functionality:** If the user ever gets lost or wants to return to the starting view, they can simply press the **'R' key**. This will reset the zoom, pan, simulation speed, and planet positions back to their original state.

Future Directions:

While we are proud of the interactive 2D solar system we have created, we also recognize that there are many exciting ways this project could be improved and expanded in the future. Our current project serves as a strong foundation, and here are some of the future directions we have considered.

First, the most obvious and ambitious improvement would be to **transition from a 2D to a 3D simulation**. While our 2D view is great for understanding orbits from a top-down perspective, a 3D model would provide a much more immersive and realistic experience. In a 3D environment, we could represent the slight inclinations of the planets' orbits, which are not all on the same flat plane. Users could rotate the camera to view the solar system from any angle, whether it's from above, from the side, or even from the perspective of a spaceship flying through the system. This would also make the self-rotation of the planets and the sun visible, adding another layer of realism. Moving to 3D would require learning more advanced OpenGL concepts, such as 3D transformations, lighting, and camera management, making it a challenging but very rewarding next step.

Another major area for improvement is in **visual fidelity**. Currently, our planets are simple, single-colored circles. A great future enhancement would be to **add textures to the celestial bodies**. We could map images of the surfaces of the Sun, Mercury, Earth, and Mars onto our 2D circles (or 3D spheres). This would make the planets instantly recognizable and visually stunning. For the Sun, we could even add a glowing effect using blending techniques to make it look more like a star. For Saturn, instead of a simple 2D ring, we could create a more

detailed ring system made of thousands of small particles, which would look much more realistic.

The simulation's accuracy could also be enhanced. Our current model uses perfect circles for orbits, which is a good approximation but not entirely accurate. In reality, the planets follow slightly elliptical (oval-shaped) paths. Modifying the code to simulate **elliptical orbits** would be a great way to incorporate more real-world physics into the project. We could also **add more celestial bodies**. This could include dwarf planets like Pluto, major asteroids from the asteroid belt between Mars and Jupiter, or even famous comets with their highly elliptical orbits and visible tails.

Finally, we could greatly increase the **educational value** of the project. We could implement a feature where clicking on a planet brings up a window with interesting facts and figures about it, such as its diameter, mass, temperature, and number of moons. This would turn the project from a purely visual simulation into a rich, interactive encyclopedia of the solar system. We could also add a "scenario" mode, where the simulation could automatically demonstrate interesting astronomical events, such as a solar eclipse or a planetary alignment.

These future directions offer a clear roadmap for how this project could continue to grow. Each of these ideas would build upon the foundation we have already created, making the simulation more realistic, more visually appealing, and more educational.

Conclusion:

In conclusion, our "Interactive 2D Solar System" project successfully achieves the goals we set out to accomplish. We have created a stable, engaging, and educational tool that visually demonstrates the layout and motion of the planets in our solar system. By leveraging the power of C++ and the OpenGL graphics library, we were able to build a simulation that is both interactive and informative. The project serves as a practical application of the core concepts of computer graphics, including 2D transformations, animation loops, user input handling, and procedural content generation.

Throughout the development of this project, our team gained valuable hands-on experience. We learned how to structure a graphics application from the ground up, starting with the main application loop and building up layers of complexity. We applied mathematical principles, particularly trigonometry, to solve the real-world problem of simulating planetary orbits. We also learned the importance of good software design, such as separating different parts of the code into logical files (`render`, `camera`, `planets`, `stars`) and using a common header file (`common.h`) to manage shared data. This modular approach made it much easier for us to work as a team, with each member taking ownership of a specific part of the project.

The final result is a feature-rich application that goes beyond a simple animation. The user is given a great deal of control, allowing them to navigate the virtual space, manipulate time, and focus on the aspects of the solar system that interest them most. The interactive camera, with its pan and zoom capabilities, and the flexible simulation controls, such as pause and

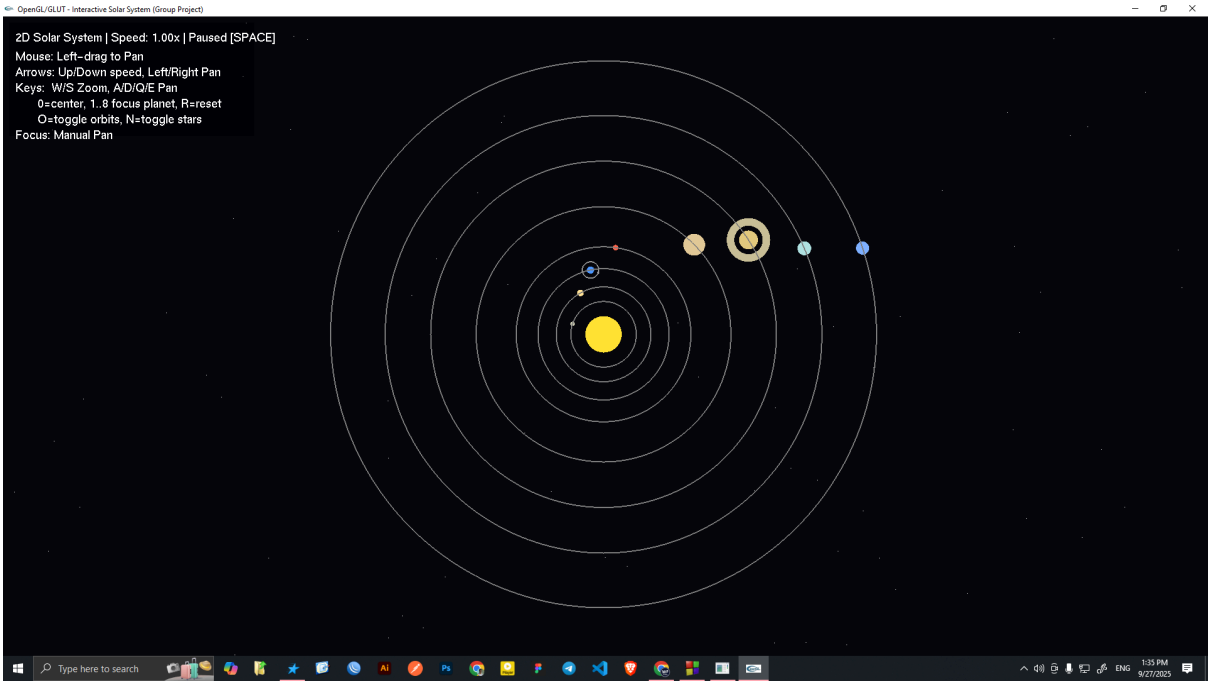
speed adjustment, make the project a truly dynamic experience. The inclusion of an informative HUD and a procedurally generated starfield adds a final layer of polish and completeness to the application.

Reflecting on the project, we are proud of the final product and the collaborative effort that went into it. We successfully tackled the challenge of representing the immense scale and complex motion of the solar system in an intuitive and accessible way. This project has not only solidified our understanding of computer graphics but has also sparked our interest in the potential of using technology for education and scientific visualization. We believe that the skills and knowledge we have gained will be invaluable in our future studies and careers. The project stands as a testament to what can be achieved with a clear vision, good teamwork, and the powerful tools of computer graphics.

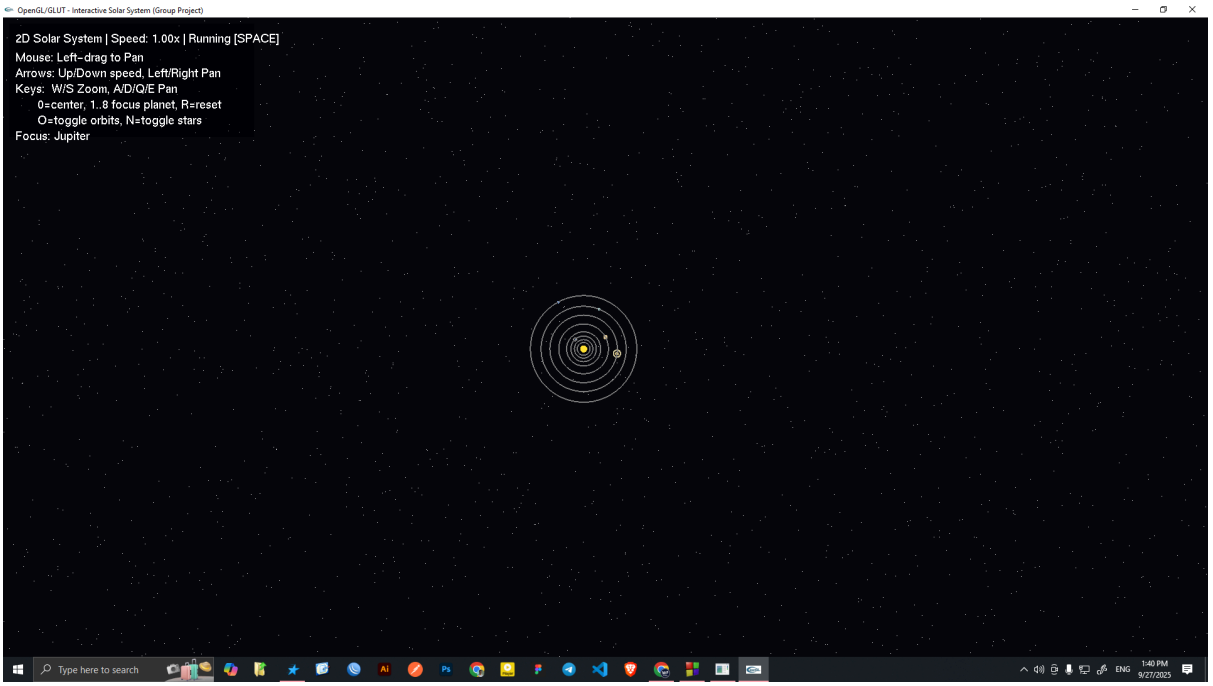
Implementation Code:

https://github.com/fuadhasandipro/InteractiveSolarSystem

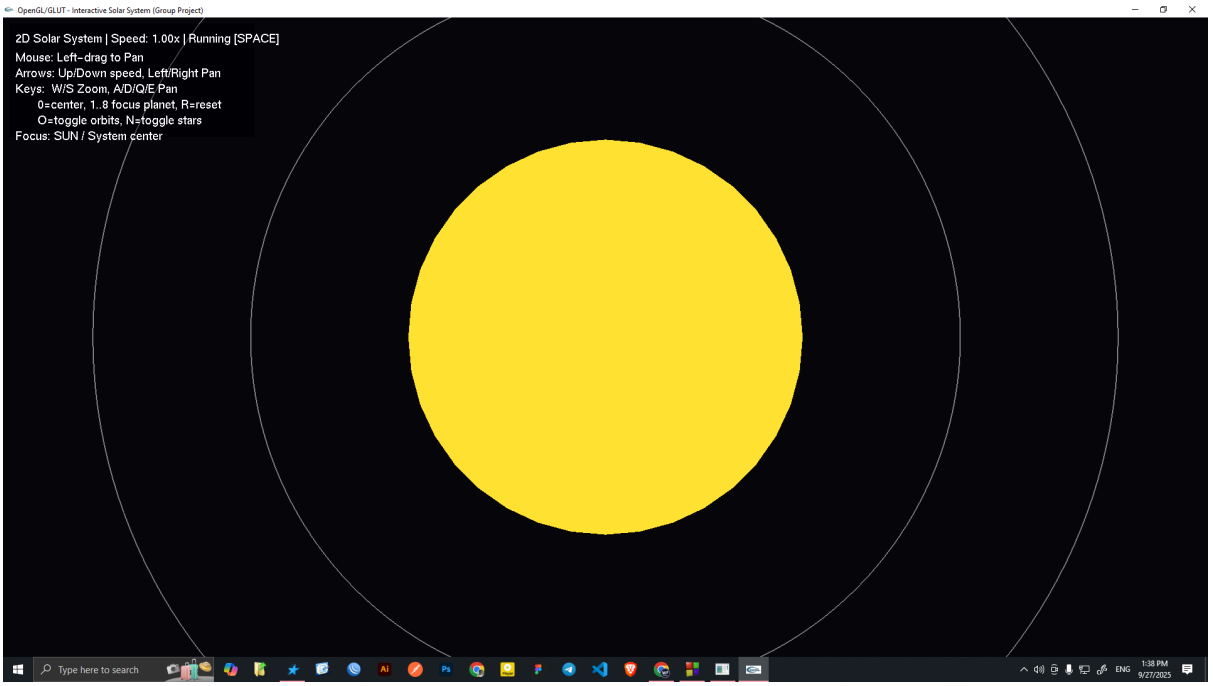
Screenshots of the Project:



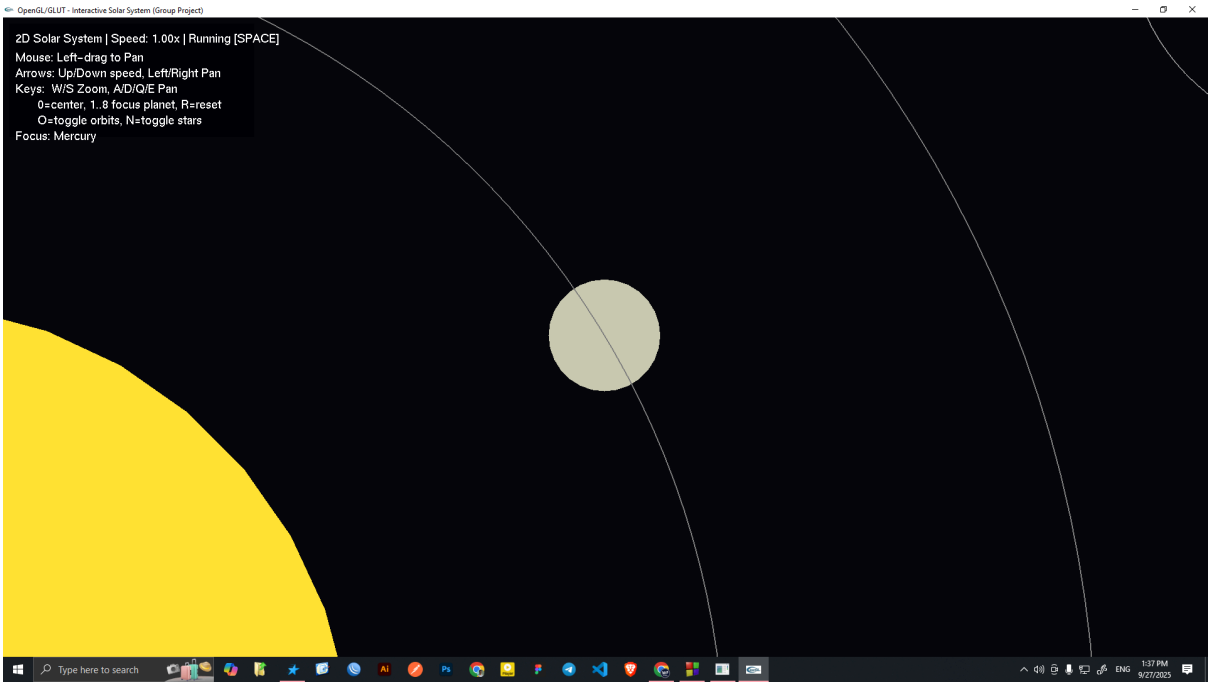
Scenario 1: Full view of the solar system



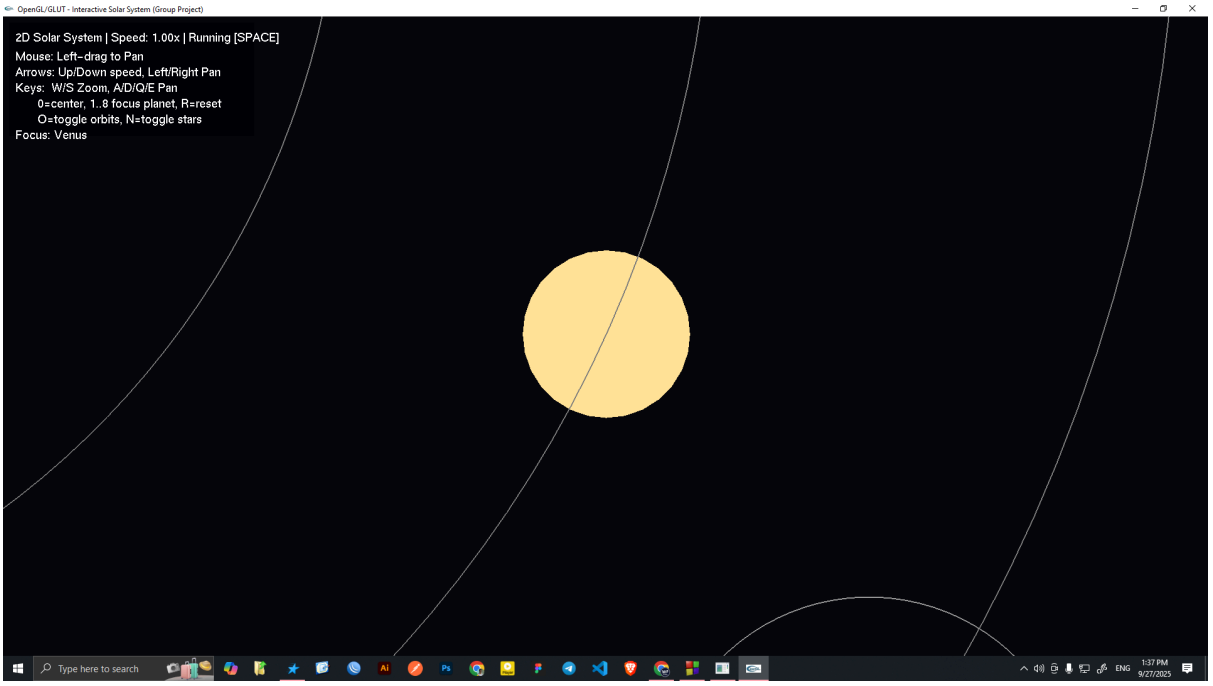
Scenario 2: Zoomed out view with stars



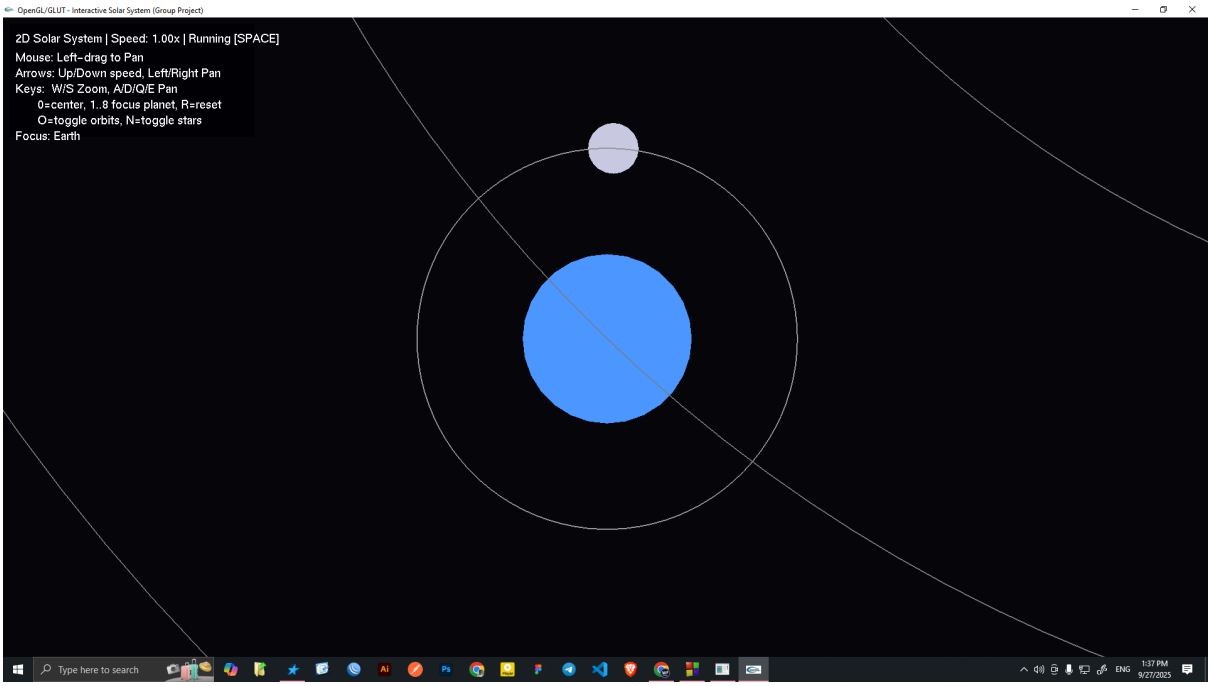
Scenario 3: Zoomed-in view of Sun



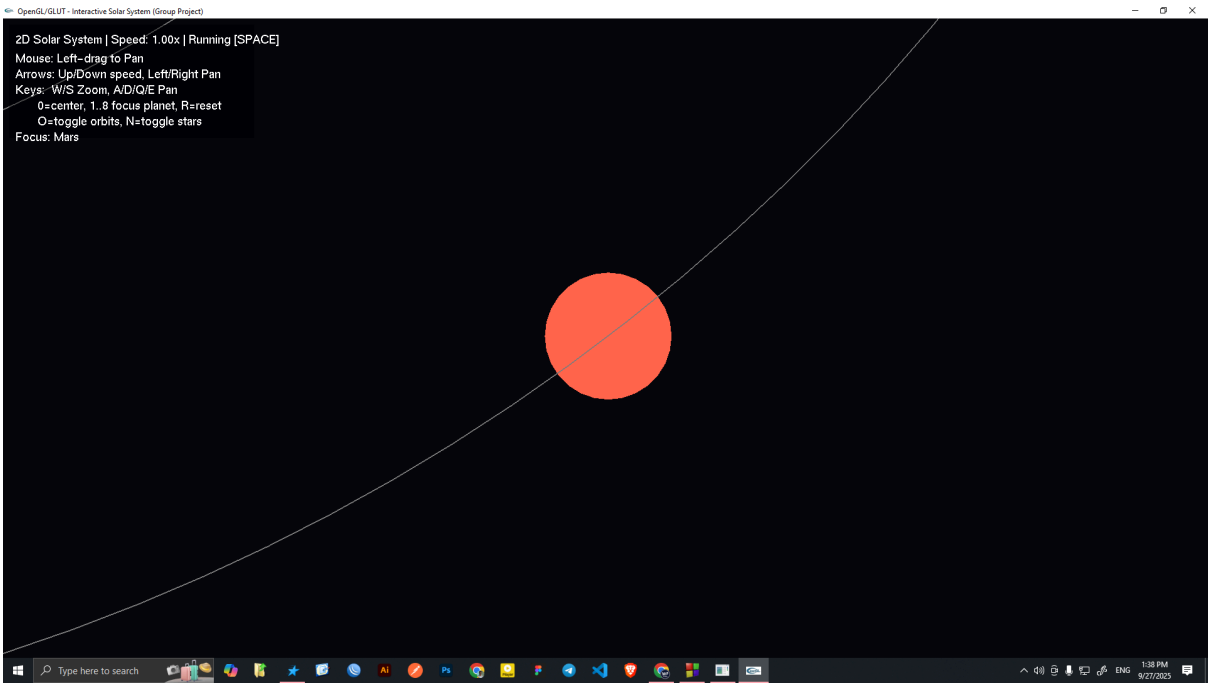
Scenario 4: Zoomed-in view of Mercury



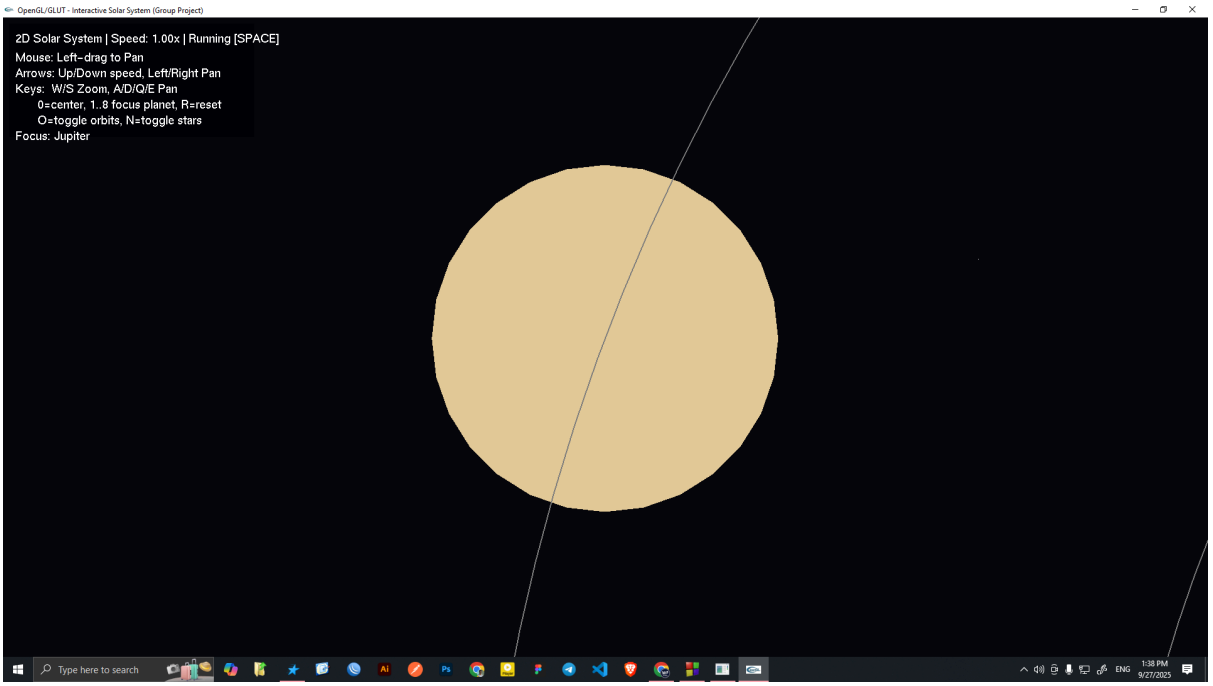
Scenario 5: Zoomed-in view of Venus



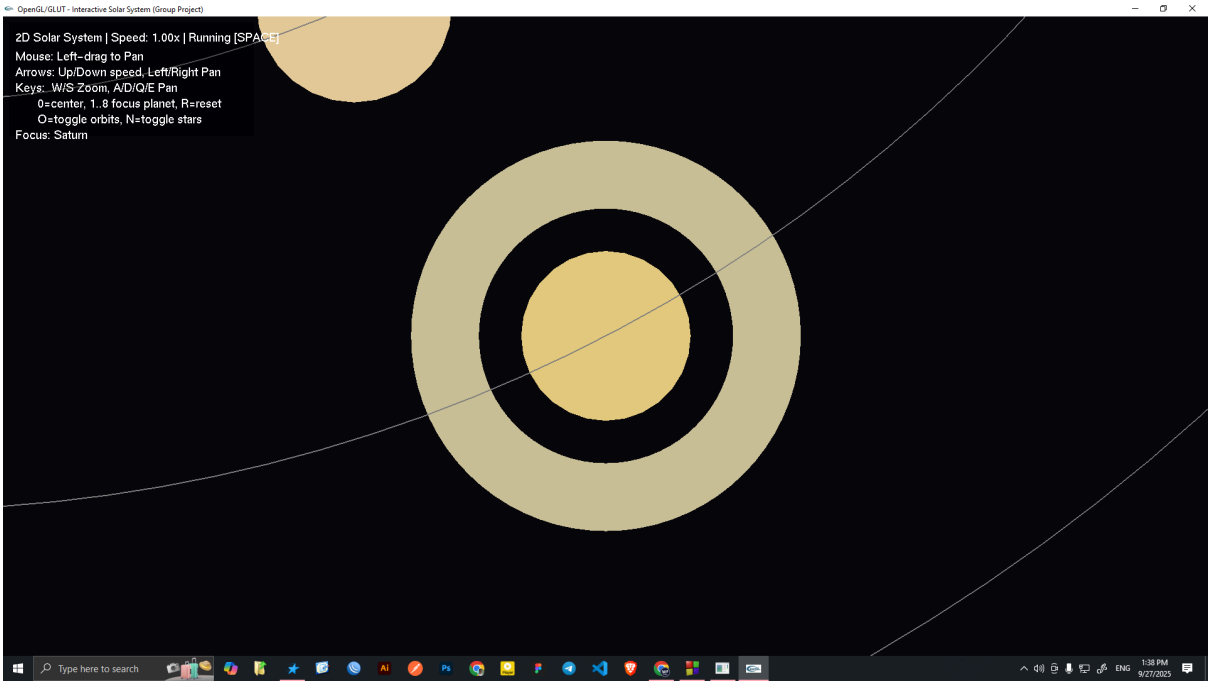
Scenario 6: Zoomed-in view of Earth with orbiting Moon



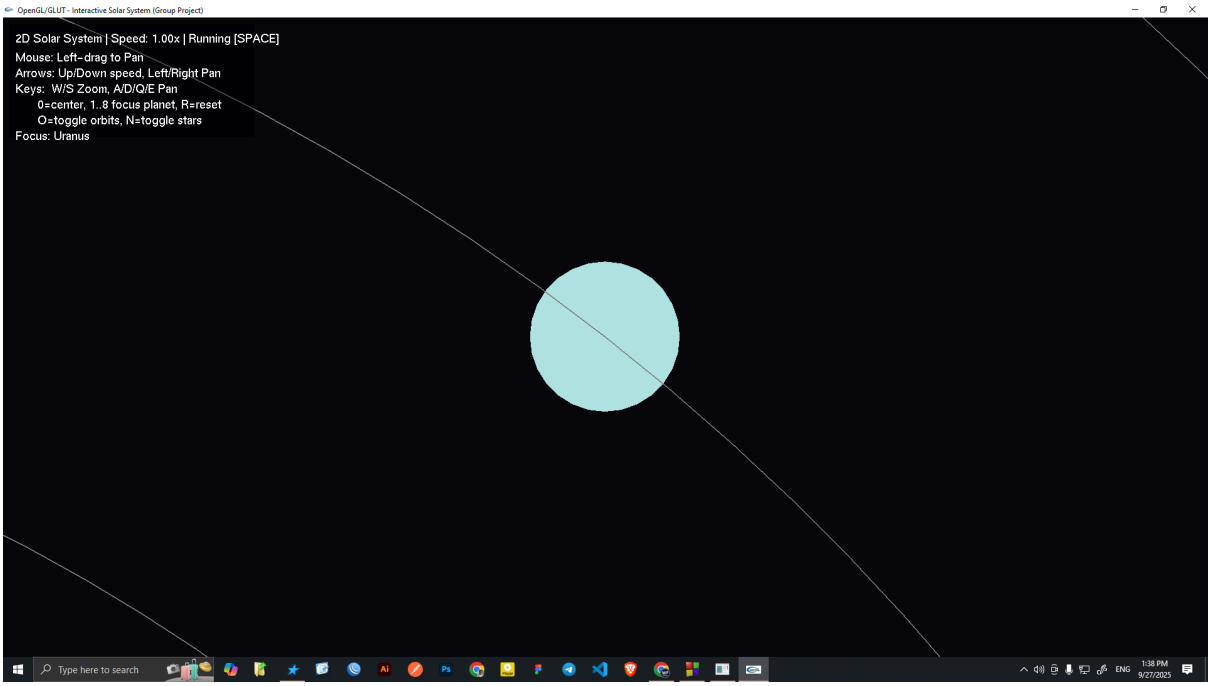
Scenario 7: Zoomed-in view of Mars



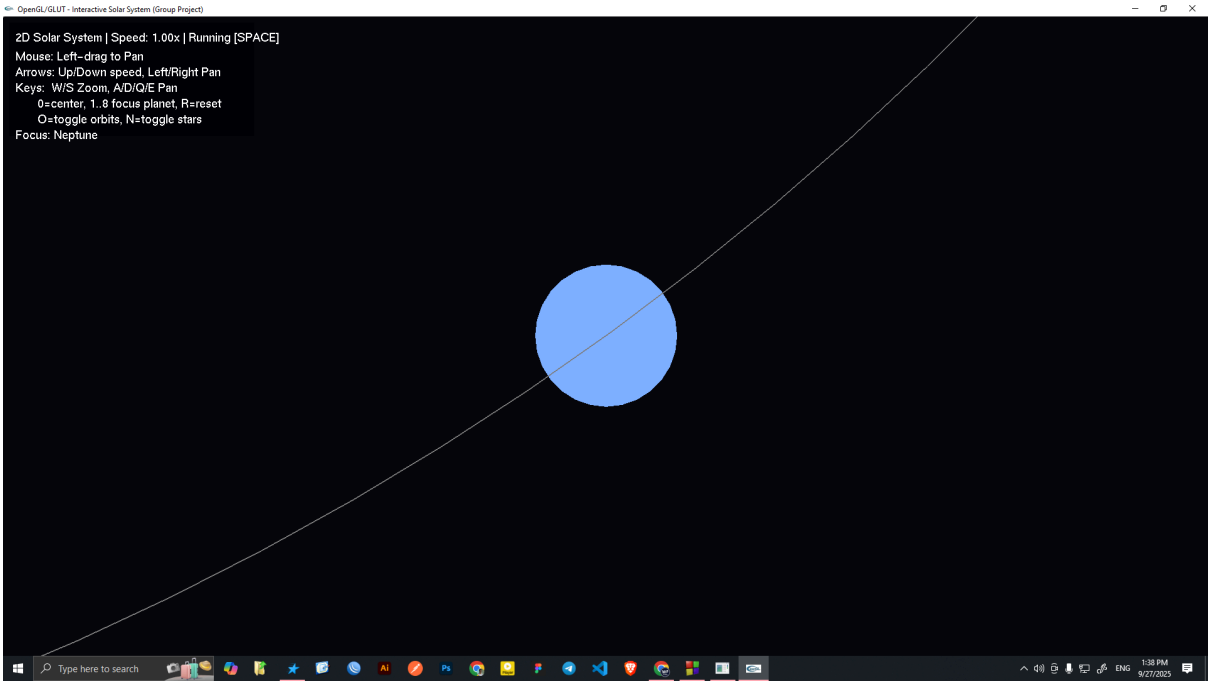
Scenario 8: Zoomed-in view of Jupiter



Scenario 9: Zoomed-in view of Jupiter



Scenario 10: Zoomed-in view of Uranus



Scenario 11: Zoomed-in view of Neptune