Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Mini Project

3D Simulation of Solar System

[Code No: COMP 342]

Submitted by

Prasiddhi Dahal (08)

Submitted to:

Mr. Dhiraj Shrestha

Department of Computer Science and Engineering

Submission Date: 14/1/2024

Introduction

The Sun and every other celestial body that orbits it, including planets, moons, asteroids, comets, and other celestial bodies, make up the Solar System. The estimated age of the Solar System, which is housed within the Milky Way galaxy, is 4.6 billion years.

The eight planets in the Solar System, in order from the Sun, are Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, and Neptune and also Pluto. These planets all rotate around the Sun in elliptical orbits, meaning their distance from the Sun changes throughout their orbit. The speeds of rotation of theplanets vary, with the inner planets (Mercury, Venus, Earth, and Mars) generally rotating faster than the outer planets (Jupiter, Saturn, Uranus, Neptune and Pluto). For example, Jupiter rotates once every 9.9 hours, while Neptune takes about 16 hours to complete one rotation.

The planets also have different positions in the Solar System. The four inner planets are located closer to the Sun and are rocky, while the four outer planets are located farther from the Sun and are mostly made of gas and ice. Overall, the Solar System is a fascinating and complex system that continues to inspire exploration and discovery.

With the help of modern technology and computer graphics, we can create stunning simulations of the Solar System. In this project, I will be using Python and PyOpenGL to simulate the Solar System which will include The Sun, the eight planets and the sparkling stars in the galaxy, all in 3D graphics. PyOpenGL is a powerful library that provides bindings to the OpenGL API, which is widely used for creating computer graphics applications. The simulation will allow us to explore the Solar System from different perspectives and learn about the unique characteristics of the planets.

Project Description with Snapshots

This solar system simulation project utilizes Python, Pygame, and OpenGL to create an interactive and visually engaging representation of the solar system. The main objective is to offer users an immersive experience where they can observe and learn about the motions and characteristics of celestial bodies.

The simulation includes key components of the solar system, such as the Sun and several planets: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune and Pluto. Each planet is represented as a distinct sphere with a defined radius and color, accurately reflecting its appearance. These planets are positioned at appropriate distances from the Sun, creating a scaled-down model of their orbital paths. The motion of the planets is simulated by applying rotational transformations around the z-axis, allowing them to move along their respective circular orbits. The speed of rotation of each planet is also taken into account, ensuring realistic movement within the simulation. By observing the planets' orbits and rotations, users can gain a better understanding of their relative positions and movements in the solar system.

To enhance the visual experience, the simulation incorporates a backdrop of stars. These stars are randomly positioned in 3D space, generating a realistic representation of the vastness of the universe. The brightness of the stars is varied to add depth and visual interest to the scene, creating a visually stunning environment for the solar system simulation. The explanation of different parts of the source code is shown below:

```
import math
import numpy as np
import pygame
from OpenGL.GL import *
from OpenGL.GLU import *
```

The above section imports the math and numpy modules for mathematical calculations,pygame module for graphical display, and required OpenGL functions.

```
class Planet:
    def __init__(self, name, color, radius, distance, speed):
        self.name = name
        self.color = color
        self.radius = radius
        self.distance = distance
        self.speed = speed
        self.angle = 0.0

    def draw(self):
        num_segments = 100
        glColor3f(0.5, 0.5, 0.5)
        glBegin(GL_LINE_LOOP)
        for i in range(num_segments):
            theta = 2.0 * math.pi * float(i) / num_segments
            dx = self.distance * math.cos(theta)
            dy = self.distance * math.sin(theta)
            glVertex3f(dx, dy, 0.0)
        glEnd()
        glPushMatrix()
        glColor3f(self.color[0], self.color[1], self.color[2])
        glRotatef(self.angle, 0.0, 0.0, 1.0)
        glTranslatef(self.distance, 0.0, 0.0)
        gluSphere(gluNewQuadric(), self.radius, 32, 32)

        # Draw rings if the planet is Saturn
        if self.name == "Saturn":
            glColor3f(0.8, 0.8, 0.6)  # Color of the rings
            for i in np.linspace(self.radius + 0.1, self.radius + 0.3, 2):  # Two
rings
                glBegin(GL_LINE_LOOP)
                for theta in np.linspace(0, 2 * np.pi, 100):
                    x = i * np.cos(theta)
                    y = i * np.sin(theta)
                    glVertex3f(x, y, 0.0)
                glEnd()

        glPopMatrix()

    def update_position(self):
        self.angle += self.speed
```

The **Planet class** represents a celestial body in the solar system simulation. Each planet is defined by its name, color, radius, distance, and speed attributes. The speed attribute determines the rate of rotation for the planet. A higher speed value causes the planet to rotate more quickly, while a lower value results in a slower rotation. The angle attribute is used to track the current rotational position of the planet. It is updated based on the planet's speed, allowing for continuous rotation over time. The draw method is responsible for rendering the planet and its orbit path.

Drawing the Orbit Path:

num_segments = 100 sets the number of line segments to define the circular path. glColor3f(0.5, 0.5, 0.5) sets the color to gray. glBegin(GL_LINE_LOOP) initiates the drawing process for a connected line loop. The for loop runs through num_segments, calculating dx and dy coordinates for each point along the circular path using trigonometric functions (cos and sin). glVertex3f(dx, dy, 0.0) plots each vertex of the circular path in 3D space (dx and dy for x and y, while z remains constant at 0.0). glEnd() finalizes the drawing of the orbit path.

Drawing the Planet:

glPushMatrix() saves the current matrix state to allow transformations specific to the planet without affecting other elements. glColor3f(self.color[0], self.color[1], self.color[2]) sets the color of the planet using the RGB values provided during initialization. glRotatef(self.angle, 0.0, 0.0, 1.0) applies a rotation transformation around the z-axis (1.0 indicates the z-axis) based on the planet's current angle. glTranslatef(self.distance, 0.0, 0.0) translates the planet to its orbit distance along the x-axis. gluSphere(gluNewQuadric(), self.radius, 32, 32) creates a sphere representing the planet using OpenGL's quadric object. It's rendered at the planet's position, with the specified radius and mesh details (32x32 segments).

Saturn's Rings:

If the planet is "Saturn," additional code inside the conditional block will draw its rings. It creates two concentric rings around Saturn using nested loops and GL_LINE_LOOP, varying radii to illustrate the rings' appearance. glPopMatrix():Restores the previous transformation state, ensuring that the transformations applied for this planet do not affect subsequent objects. The planet's position is updated by incrementing the angle attribute based on its speed. The update_position method is responsible for updating the planet's position in each frame of the simulation. By incrementing the angle attribute based on the planet's speed, the planet appears to rotate smoothly over time. The code provides a simple and scalable implementation for simulating multiple planets with different attributes.

```
class Moon:
    def __init__(self, planet, distance, radius, speed, color):
        self.planet = planet
        self.distance = distance
        self.radius = radius
        self.speed = speed
        self.color = color
        self.angle = 0

    def update_position(self):
        self.angle += self.speed
        if self.angle > 360:
            self.angle -= 360

    def draw(self):
        glPushMatrix()
        glColor3f(self.color[0], self.color[1], self.color[2])
        glRotatef(self.planet.angle, 0.0, 0.0, 1.0)
        glTranslatef(self.planet.distance, 0.0, 0.0)
        glRotatef(self.angle, 0.0, 0.0, 1.0)
        glTranslatef(self.distance, 0.0, 0.0)
        gluSphere(gluNewQuadric(), self.radius, 32, 32)
        glPopMatrix()
```

The **Moon class** represents a celestial body orbiting around a given planet within a three-dimensional space rendered using OpenGL. Upon instantiation, a Moon object is initialized with properties such as the parent planet it orbits, the distance from the planet, its own radius, orbital speed, color, and an initial orbital angle set to 0.

The **update_position** method is responsible for incrementing the moon's orbital angle based on its orbital speed. Additionally, it ensures that the angle remains within the range [0, 360) degrees by resetting it to the difference between its current angle and 360 when it exceeds 360 degrees.

The draw method utilizes OpenGL transformations to render the moon in its orbit around the parent planet. Within the OpenGL modelview matrix, the method first pushes the current matrix onto the stack (glPushMatrix()). It then sets the drawing color to the specified RGB color of the moon using glColor3f. The subsequent transformations involve rotating the coordinate system around the z-axis by the current angle of the parent planet (glRotatef(self.planet.angle, 0.0, 0.0, 1.0)), translating to the distance from the planet along the x-axis (glTranslatef(self.planet.distance, 0.0, 0.0)), rotating the coordinate system around its own axis by the current angle of the moon (glRotatef(self.angle, 0.0, 0.0, 1.0)), and finally translating to the moon's position along its orbit (glTranslatef(self.distance, 0.0, 0.0)). The moon is then rendered as a sphere using gluSphere(gluNewQuadric(), self.radius, 32, 32). Finally, the method pops the previous matrix

from the stack (glPopMatrix()), restoring the original coordinate system.

```
class Meteor:
    def __init__(self, color, radius, speed):
        self.color = color
        self.radius = radius
        self.speed = speed
        self.angle = np.random.uniform(0.0, 360.0)
        self.distance = np.random.uniform(15.0, 20.0)

    def draw(self):
        glColor3f(self.color[0], self.color[1], self.color[2])
        glPushMatrix()
        glRotatef(self.angle, 0.0, 0.0, 1.0)
        glTranslatef(self.distance, 0.0, 0.0)
        gluSphere(gluNewQuadric(), self.radius, 32, 32)
        glPopMatrix()

    def update_position(self):
        self.angle += self.speed
```

The **Meteor class** represents a meteor moving through three-dimensional space in an OpenGL-rendered solar system. Upon instantiation, a Meteor object is initialized with properties including its color represented as an RGB tuple, radius specifying its size, speed determining its angular speed of rotation, a random initial angle of rotation (in degrees), and a random initial distance from the origin along the x-axis.

The **draw method** is responsible for rendering the meteor. It sets the drawing color to the specified RGB color using glColor3f(self.color[0], self.color[1], self.color[2]). Inside the OpenGL modelview matrix, it pushes the current matrix onto the stack (glPushMatrix()), applies a rotation around the z-axis by the meteor's current angle of rotation (glRotatef(self.angle, 0.0, 0.0, 1.0)), translates to the meteor's position along its orbit (glTranslatef(self.distance, 0.0, 0.0)), and finally renders the meteor as a sphere using gluSphere(gluNewQuadric(), self.radius, 32, 32). It then pops the previous matrix from the stack (glPopMatrix()), restoring the original coordinate system.

The **update_position** method is responsible for updating the meteor's position by incrementing its angle of rotation based on its angular speed (self.angle += self.speed). This method is typically called in the main loop of the simulation to animate the movement of the meteors over time.

```
class SolarSystem:
    def __init__(self):
        earth = Planet("Earth", (0.0, 0.5, 1.0), 0.7, 10.0, 2.5)
        moon = Moon(earth, 2.0, 0.1, 2.0, (1.0, 1.0, 1.0))
        earth.moons = [moon]

        self.planets = [
            Planet("Sun", (1.0, 1.0, 0.0), 1.0, 0.0, 0.1), #color, radius,
distance, speed
            Planet("Mercury", (0.6, 0.3, 0.0), 0.5, 4.0, 4.0),
            Planet("Venus", (0.5, 0.2, 0.5), 0.6, 7.0, 3.0),
            earth,
            Planet("Mars", (1.0, 0.0, 0.0), 0.6, 15.0, 2.0),
            Planet("Jupiter", (0.9, 0.7, 0.5), 1.3, 20.0, 1.3),
            Planet("Saturn", (0.9, 0.7, 0.5), 1.2, 25.0, 1.0),
            Planet("Uranus", (0.0, 0.5, 0.5), 2, 28.0, 0.3),
            Planet("Neptune", (0.0, 0.0, 1.0), 2, 34.0, 0.2),
            Planet("Pluto", (0.5, 0.5, 0.5), 0.4, 36.0, 0.1),
        ]
        self.meteors = [Meteor((0.5, 0.5, 0.5), 0.1, np.random.uniform(0.1, 1.5))
for _ in range(100)]

    def update_positions(self):
        for planet in self.planets:
            planet.update_position()
            if hasattr(planet, 'moons'):
                for moon in planet.moons:
                    moon.update_position()
        for meteor in self.meteors:
            meteor.update_position()

    def draw(self):
        for planet in self.planets:
            planet.draw()
            if hasattr(planet, 'moons'):
                for moon in planet.moons:
                    moon.draw()
        for meteor in self.meteors:
            meteor.draw()
```

The **SolarSystem class** initializes and manages the different celestial bodies, including planets, moons, and meteors, to coordinate the simulation of a celestial system. It starts by seeing Earth as a planet with unique characteristics and connecting it to a moon. The Moon class represents this moon, which orbits Earth according to particular orbital parameters. Other planets make up the solar system; they are all instantiated as instances of the Planet class and are arranged in a hierarchical structure. There are also a lot of meteors are added between jupter and mars, which are

created as instances of the Meteor class.

The update_positions method makes sure that the locations of the celestial bodies move continuously over time. It takes into account that planets and moons move in orbit while meteors move in a straight line. The draw technique shows the solar system as it is right now, with each planet, moon, and meteor depicted in its proper location. An interactive and visually appealing portrayal of a solar system in an OpenGL environment is made possible by this hierarchical and dynamic method.

```
class Star:
    def __init__(self):
        self.color = (np.random.uniform(0.0, 1.0), np.random.uniform(0.0, 1.0),
np.random.uniform(0.0, 1.0))
        self.position = (np.random.uniform(-40.0, 40.0), np.random.uniform(-40.0,
40.0), np.random.uniform(-40.0, 40.0))

    def draw(self):
        glColor3f(*self.color)
        glVertex3f(*self.position)

stars = [Star() for _ in range(500)]  # Generate stars once

def draw_stars():
    glPointSize(0.5)
    glBegin(GL_POINTS)
    for star in stars:  # Draw each star
        star.draw()
    glEnd()
```

The **Star class** uses OpenGL to encapsulate the characteristics and drawing logic for individual stars in three dimensions. Every star has a random RGB color initialized in its __init__ method, with each color component falling between [0.0, 1.0]. In addition, any x, y, and z coordinates within the interval [-40.0, 40.0] are used to set the star's position. By defining the vertex at its 3D position and setting the OpenGL color to the star's color, the draw method renders the star.

The Star class is used by the draw_stars function to create a list of 500 stars with unique colors and locations. The draw method is then called for each star as iteratively going over the list, rendering every star in the designated 3D space. The function creates an aesthetically pleasing depiction of a starfield by using OpenGL functions to set the point size and render each star as a point.

```
def main():
    pygame.init()
    display = (800, 600)
    pygame.display.set_mode(display, pygame.DOUBLEBUF | pygame.OPENGL)
    gluPerspective(500, (display[0] / display[1]), 0.1, 50.0)
    glTranslatef(0.0, 0.0, -5)

    solar_system = SolarSystem()

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        glRotatef(1, 3, 1, 1)
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        draw_stars()
        solar_system.update_positions()
        solar_system.draw()
        pygame.display.flip()
        pygame.time.wait(30)

if __name__ == "__main__":
    main()
```

Initialization in **main function:**

- pygame.init(): Initializes the Pygame library.
- display = (800, 600): Sets the dimensions of the display window.
- pygame.display.set_mode(display, pygame.DOUBLEBUF | pygame.OPENGL): Creates a Pygame window with OpenGL support.
- gluPerspective(500, (display[0] / display[1]), 0.1, 50.0): Sets the perspective projection for the OpenGL scene.
- glTranslatef(0.0, 0.0, -5): Translates the scene by moving it 5 units away from the camera.

Solar System Initialization:

- solar_system = SolarSystem(): Initializes the SolarSystem object, creating planets and meteors.

Event Handling and Rendering Loop:

- while True:: Starts an infinite loop for continuous rendering.

Event Handling:

- for event in pygame.event.get():: Loops through Pygame events.
- if event.type == pygame.QUIT:: Checks if the quit event (window close) is triggered.
- pygame.quit(): Quits Pygame.
- quit(): Exits the program.

Scene Rendering:

- glRotatef(1, 3, 1, 1): Applies a slight rotation to the scene.
- glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT): Clears the display.
- draw_stars(): Draws stars using the draw_stars() function.
- solar_system.update_positions(): Updates the positions of planets and meteors.
- solar_system.draw(): Draws planets and meteors in the solar system.
- pygame.display.flip(): Updates the Pygame display.
- pygame.time.wait(30): Adds a 30-millisecond delay before the next iteration.

Execution:

if __name__ == "__main__":: Ensures that this code block is executed only when the script is run directly.

main(): Calls the main() function to start the simulation loop

Source Code

Below is the complete source code:

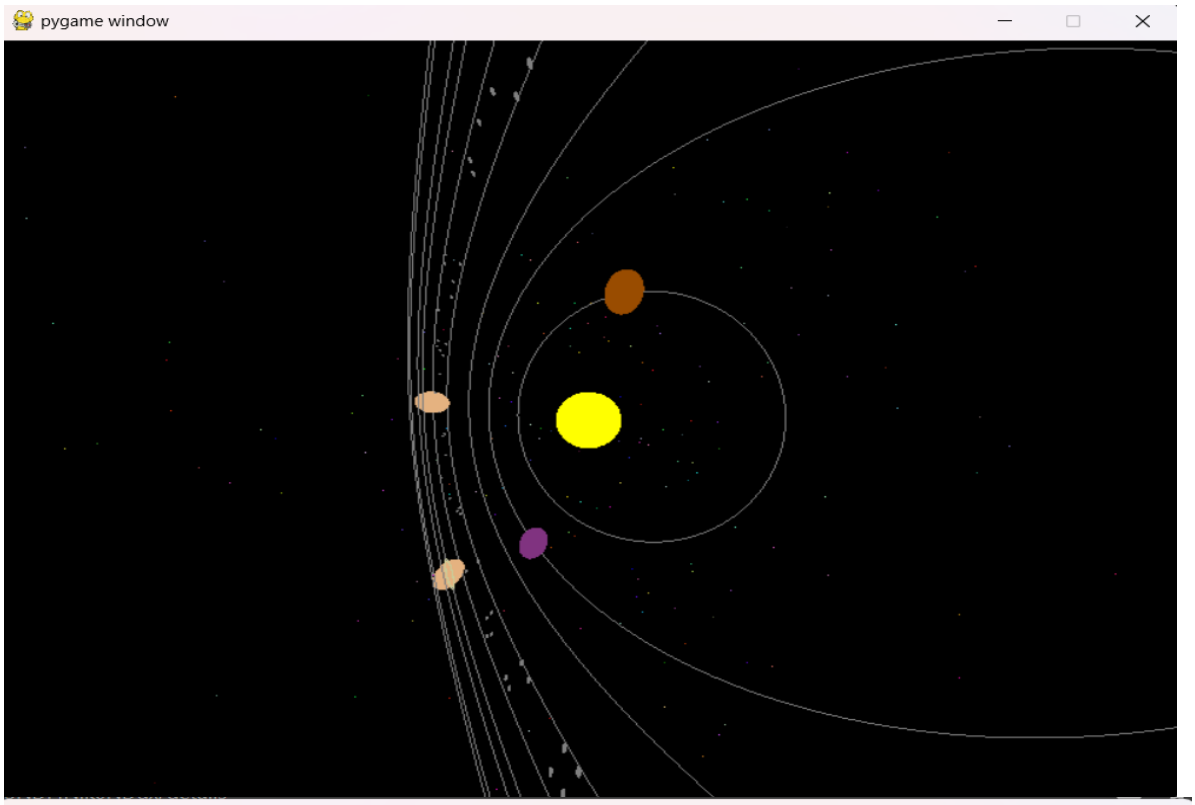
```
import math
import numpy as np
import pygame
from OpenGL.GL import *
from OpenGL.GLU import *

class Planet:
    def __init__(self, name, color, radius, distance, speed):
        self.name = name
        self.color = color
        self.radius = radius
        self.distance = distance
        self.speed = speed
        self.angle = 0.0

    def draw(self):
        num_segments = 1000 # The higher this number, the more circular the planet
will appear
        glColor3f(0.5, 0.5, 0.5)
        glBegin(GL_LINE_LOOP)
        for i in range(num_segments):
            theta = 2.0 * math.pi * float(i) / num_segments
            dx = self.distance * math.cos(theta)
            dy = self.distance * math.sin(theta)
            glVertex3f(dx, dy, 0.0)
        glEnd()
        glPushMatrix()
        glColor3f(self.color[0], self.color[1], self.color[2])
        glRotatef(self.angle, 0.0, 0.0, 1.0) #rotates the obj around z axis
        glTranslatef(self.distance, 0.0, 0.0)
        gluSphere(gluNewQuadric(), self.radius, 32, 32) #draws a sphere

        # Draw rings if the planet is Saturn
        if self.name == "Saturn":
            glColor3f(0.8, 0.8, 0.6)  # Color of the rings
            for i in np.linspace(self.radius + 0.1, self.radius + 0.3, 2):  # Two
rings
                glBegin(GL_LINE_LOOP)
                for theta in np.linspace(0, 2 * np.pi, 100):
                    x = i * np.cos(theta)
                    y = i * np.sin(theta)
                    glVertex3f(x, y, 0.0)
```

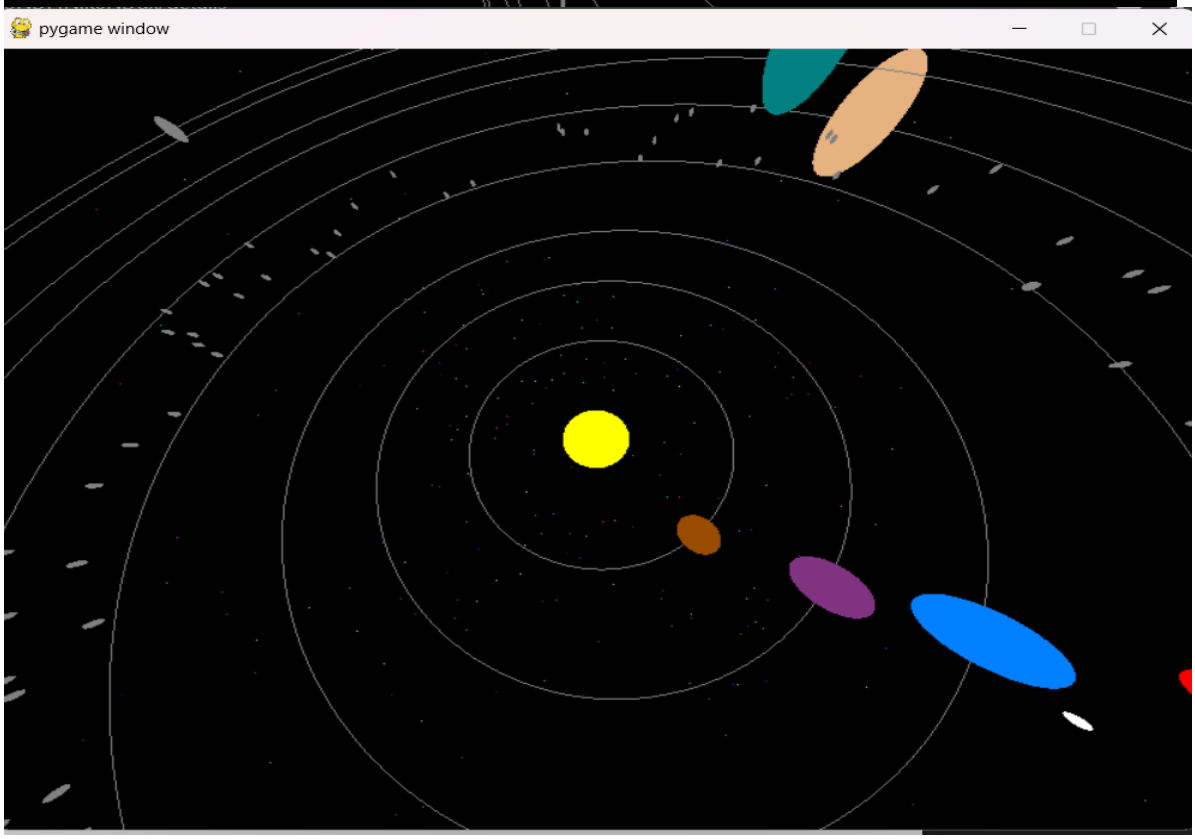
```
                glEnd()

        glPopMatrix()

    def update_position(self):
        self.angle += self.speed
class Moon:
    def __init__(self, planet, distance, radius, speed, color):
        self.planet = planet
        self.distance = distance
        self.radius = radius
        self.speed = speed
        self.color = color
        self.angle = 0

    def update_position(self):
        self.angle += self.speed
        if self.angle > 360:
            self.angle -= 360

    def draw(self):
        glPushMatrix()
        glColor3f(self.color[0], self.color[1], self.color[2])
        glRotatef(self.planet.angle, 0.0, 0.0, 1.0)
        glTranslatef(self.planet.distance, 0.0, 0.0)
        glRotatef(self.angle, 0.0, 0.0, 1.0)
        glTranslatef(self.distance, 0.0, 0.0)
        gluSphere(gluNewQuadric(), self.radius, 32, 32)
        glPopMatrix()
class Meteor:
    def __init__(self, color, radius, speed):
        self.color = color
        self.radius = radius
        self.speed = speed
        self.angle = np.random.uniform(0.0, 360.0)
        self.distance = np.random.uniform(15, 20.0)

    def draw(self):
        glColor3f(self.color[0], self.color[1], self.color[2])
        glPushMatrix()
        glRotatef(self.angle, 0.0, 0.0, 1.0)
        glTranslatef(self.distance, 0.0, 0.0)
        gluSphere(gluNewQuadric(), self.radius, 32, 32)
        glPopMatrix()

    def update_position(self):
        self.angle += self.speed

class SolarSystem:
```

```
    def __init__(self):
        earth = Planet("Earth", (0.0, 0.5, 1.0), 0.7, 10.0, 2.5)
        moon = Moon(earth, 2.0, 0.1, 2.0, (1.0, 1.0, 1.0))
        earth.moons = [moon]

        self.planets = [
            Planet("Sun", (1.0, 1.0, 0.0), 1.0, 0.0, 0.1), #color, radius,
distance, speed
            Planet("Mercury", (0.6, 0.3, 0.0), 0.5, 4.0, 4.0),
            Planet("Venus", (0.5, 0.2, 0.5), 0.6, 7.0, 3.0),
            earth,
            Planet("Mars", (1.0, 0.0, 0.0), 0.6, 15.0, 2.0),
            Planet("Jupiter", (0.9, 0.7, 0.5), 1.3, 20.0, 1.3),
            Planet("Saturn", (0.9, 0.7, 0.5), 1.2, 25.0, 1.0),
            Planet("Uranus", (0.0, 0.5, 0.5), 2, 28.0, 0.3),
            Planet("Neptune", (0.0, 0.0, 1.0), 2, 34.0, 0.2),
            Planet("Pluto", (0.5, 0.5, 0.5), 0.4, 36.0, 0.1),
        ]
        self.meteors = [Meteor((0.5, 0.5, 0.5), 0.1, np.random.uniform(0.1, 1.5))
for _ in range(100)]

    def update_positions(self):
        for planet in self.planets:
            planet.update_position()
            if hasattr(planet, 'moons'):
                for moon in planet.moons:
                    moon.update_position()
        for meteor in self.meteors:
            meteor.update_position()

    def draw(self):
        for planet in self.planets:
            planet.draw()
            if hasattr(planet, 'moons'):
                for moon in planet.moons:
                    moon.draw()
        for meteor in self.meteors:
            meteor.draw()
class Star:
    def __init__(self):
        self.color = (np.random.uniform(0.0, 1.0), np.random.uniform(0.0, 1.0),
np.random.uniform(0.0, 1.0))
        self.position = (np.random.uniform(-40.0, 40.0), np.random.uniform(-40.0,
40.0), np.random.uniform(-40.0, 40.0))

    def draw(self):
        glColor3f(*self.color)
        glVertex3f(*self.position)
```

```
stars = [Star() for _ in range(500)]  # Generate stars once

def draw_stars():
    glPointSize(0.5)
    glBegin(GL_POINTS)
    for star in stars:  # Draw each star
        star.draw()
    glEnd()

def main():
    pygame.init()
    display = (400, 400)
    pygame.display.set_mode(display, pygame.DOUBLEBUF | pygame.OPENGL)
    gluPerspective(500, (display[0] / display[1]), 0.1, 50.0)
    glTranslatef(0.0, 0.0, -5)

    solar_system = SolarSystem()

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        glRotatef(1, 3, 1, 1)
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        draw_stars()
        solar_system.update_positions()
        solar_system.draw()
        pygame.display.flip()
        pygame.time.wait(30)

if __name__ == "__main__":
    main()
```

Output (Screenshots)





Conclusion

The code combines OpenGL and Pygame to create an interactive solar system simulation. It distinguishes each star as a separate object, adjusting their positions dynamically and coordinating their presentation. While Pygame skillfully handles user interactions, the main loop effectively oversees continuous updates, utilizing OpenGL's power to vividly showcase orbits, planets, and meteors inside a rich 3D space. Users are invited to explore a dynamic portrayal of a solar system brimming with planets, orbiting routes, and celestial features thanks to this seamless integration, which produces an immersive, visually appealing experience. For both enthusiasts and learners, the harmonious fusion of Pygame's interaction with OpenGL's graphics powerhouse creates a fascinating, informative, and exciting astronomical journey.

References

1. *An Introduction to the Solar System – Planetary Sciences, Inc.* (n.d.). https://planetary-science.org/astronomy/solar-system/an-introduction-to-the-solar-system/
2. Khalid, M. J. (2019). Brief Introduction to OpenGL in Python with PyOpenGL. *Stack Abuse*. https://stackabuse.com/brief-introduction-to-opengl-in-python-with-pyopengl/