The Inner Planets Simulation

Developed by Damian Lall, Albert Sun, and Allen Wang

In our program, we simulated a Sun and Inner planet system. Our project was written in Python and used the PyGame library to handle drawing objects and the time system. The main physics and math concepts we employed were finding the angles between objects, the universal law of gravitation, and equations for calculating energy.

By inputting values found in reality, the simulation can accurately calculate variables that match reality, making this simulation a close replica to the inner planets of our solar system. However, this simulation only uses the Sun, Mercury, Venus, Mars, and Earth in the system, meaning other celestial bodies are excluded in the system calculations.

Physics and Math Concepts

Universal Law of Gravitation

```
def update_acceleration(self):
    for planet in planetsList:
        if self.id != planet.id:
            dx = denormalize_distance(self.position.x - planet.position.x)*1000 #pixels to 500000km to meters
            dy = denormalize_distance(self.position.y - planet.position.y)*1000 #pixels to 500000km to meters
            self.angle = atan2(dy, dx)  # Calculate angle between planets
            d = sqrt(pow(dx, 2) + pow(dy, 2))  # Calculate distance
            self.f = (
                normalize_distance(GRAVITATIONAL_CONSTANT * planet.mass / pow(d,2))/1000 #meters to km to pixels
            )  # Calculate gravitational force
```

```
def update(self, time_constant):
    print(time_constant)
    self.velocity.x += (-cos(self.angle) * self.f) / time_constant
    self.velocity.y += (sin(self.angle) * self.f) / time_constant
    self.velocity.x += sun_gravity(self.position).x / time_constant
    self.velocity.y += sun_gravity(self.position).y / time_constant
    self.position += self.velocity / time_constant
```

The following code is part of a method executed in a nested for loop; each celestial body has the equation applied with respect to every other celestial body, reflecting Newton 3rd Law of Motion. At the end, the gravity force of the Sun is then applied to the rest of the planets. However, the sun itself does not have this equation applied due to the opposite reaction force being too small to affect the Sun. Using the formula

$$m_{planet}a = GM_{sun/planet}m_{planet}/r^2$$

$$a_{planet} = GM_{sun/planet}/r^2$$

and applying it to every planet to each other, we can have a simulation accurate to that in reality. Because the gravitational constant uses meters for distance, pixel distances must be adapted. Through normalization/denormalization, we can convert to meters for calculations. **Normalization and Denormalization will be explained in a later section.** However, the gravitational pull from the planets relative to the sun would be so small, the simulation would run around the same way without it. We decided to keep it for accuracy sake if there were certain orbits that depend on it.

Change in Angle Between Bodies

```
dx = denormalize_distance(self.position.x - planet.position.x)*1000 #pixels to 500000km to meters
dy = denormalize_distance(self.position.y - planet.position.y)*1000 #pixels to 500000km to meters
self.angle = atan2(dy, dx)  # Calculate angle between planets
```

This code finds the angle between two bodies, which is then used to find the proper change in velocity and its direction. First converting the pixel units into meters for calculations and utilizing python's atan2() function, we find the change in y and change in x to find the dy/dx and eventually the slope. The slope then returns an angle in radians, giving us the direction of acceleration. Note that the number in the atan2 function does not signify a square, but denotes it as the second version of the method, as the original did not take positive and negative signs into account.

```
self.velocity.x += (-cos(self.angle) * self.f) / time_constant
self.velocity.y += (sin(self.angle) * self.f) / time_constant
```

Applying the acceleration vector to each component based on angle helps alter the direction of the velocity vector, keeping both vectors perpendicular to each other. **Dividing by time_constant will be explained in a later section**, but applying acceleration (or change in velocity) in a constant time frame gives us a realistic change in velocity and eventually distance.

Total Energy Conversion

```
#energy calculation
self.tEnergy = 0
for planet in planetsList:
    if self.id != planet.id:
        dx = denormalize_distance(self.position.x - planet.position.x) * 1000  # pixels to 500000km to meters
        dy = denormalize_distance(self.position.y - planet.position.y) * 1000
        velSquare = pow(denormalize_distance(self.velocity.x) * 1000, 2) + pow(denormalize_distance(self.velocity.y) * 1000, 2)
        self.tEnergy -= (
                (GRAVITATIONAL_CONSTANT*planet.mass*self.mass*(1/sqrt(dx**2+dy**2)))
        )
dx = denormalize_distance(self.position.x - SUN_POS.x) * 1000
dy = denormalize_distance(self.position.y - SUN_POS.y) * 1000
velSquare = pow(denormalize_distance(self.velocity.x)*1000,2)+pow(denormalize_distance(self.velocity.y)*1000,2)
self.tEnergy += (
    ((self.mass / 2) * velSquare) - ((GRAVITATIONAL_CONSTANT*SUN_MASS*self.mass)*(1/sqrt(dx**2+dy**2)))
)
```

This code finds the total energy of the entire system, showing that within a closed system, energy is conserved in elliptical orbits when running. To prevent each update from

stacking on each other, the energy calculation is reset for every update, recalculating the energy from the entire system again.

We used this formula to determine the energy between two planets:

$$E = KE + PE = 1/2 m_{self} v^2 - GM_{planet\ 1} m_{self}/r^2 - \ldots - GM_{planet\ n} m_{self}/r^2$$

For each planet, we looped through and added the energy with every other planet and the sun, calculating total energy. Taking the distance between them, calculating the planet's current velocity, and taking each looped planet's mass, we can find the potential energy from each celestial body. Adding it back to the kinetic energy of the planet and running the simulation, we can see that energy is conserved as it remains a constant value.

Game Concepts

Distance Normalization

In order to use accurate values in the simulation, while still displaying the system onscreen, we determine a normalization constant that we use to scale the real-life distances down to pixel drawing distances that fit on the screen. The current normalization constant is 1 pixel per every 500,000 kilometers. However, due to the comparative sizes between planet radius and distance, we decided to keep the radius not to scale.

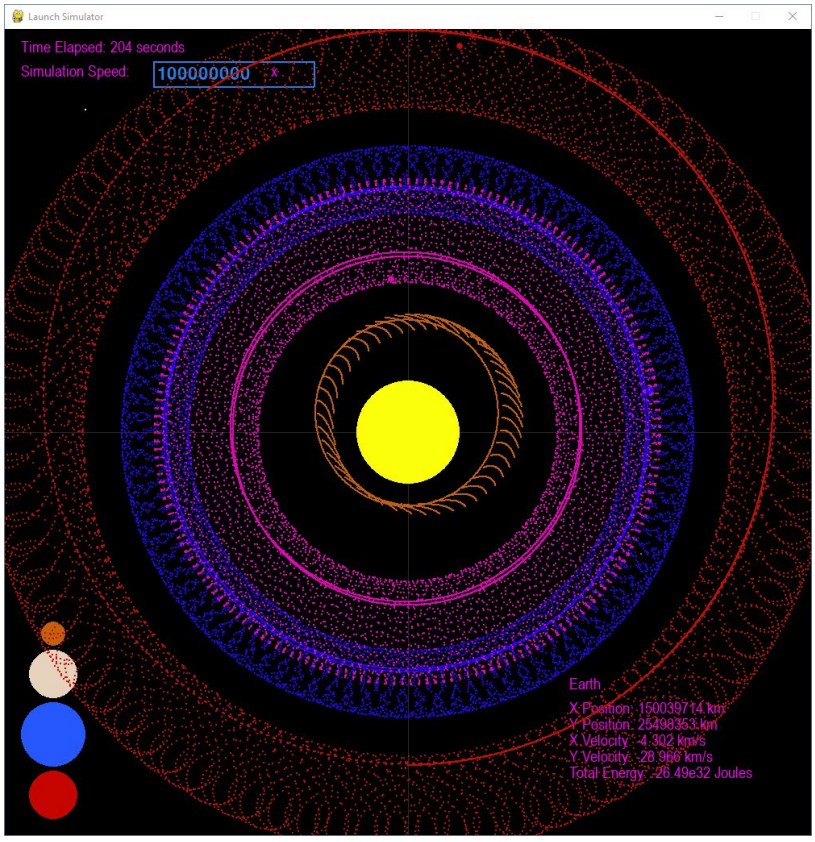
Game Ticks

Having a frame rate of sixty frames per second, every one-sixtieth of a second, the game goes through its main game loop. The main loop executes everything necessary to keep the game going: calculating angles, acceleration, velocity, position, and everything else you could imagine. The game also repaints the screen with the updated position values. Because of this, a naive implementation would inadvertently calculate and increase the speed of the simulation in accordance with a higher frame rate. To account for this, we divide every change to movement by a time constant, equivalent to the number of frames per second. For example, if we wished to apply a velocity of five meters per second, a naive implementation would add the velocity of five sixty times. However, divided by the time constant, we would only add one twelfth of a meter every tick, which would sum to five over the course of sixty ticks in a second.

Accuracy

A perfectly accurate simulation would be very boring, as the time it takes for the earth to revolve around the sun would take a full year, just like in real life! To account for this, we can divide the time constant by a desired speed multiplier. The speed multiplier is equivalent to how many times faster the simulation is to reality, which can make a year in the simulation go by in seconds. The more ticks, the smaller and more precise the calculations will be, as there will be less changes in movement between shorter lengths of time. The opposite also holds true, with great of changes in movement between ticks, even recalculated dozens of times in a second, causing the simulation to lose accuracy. This can be observed by increasing the simulation speed to values greater than about one million, afterwhich the orbits will begin to drift out of their orbits. This can be remedied by increasing the frame rate; however, one cannot increase the frame rate infinitely due to hardware constraints. As simulating every frame involves calculating the complicated physics equations described above, an excessively high rate would start to skip frames, especially on less powerful hardware, potentially causing more inaccuracies.

An example of the inaccuracies that arise from a high speed multiplier. It may look pretty, but we'd have some pretty extreme seasons if this reflected reality.



Observations

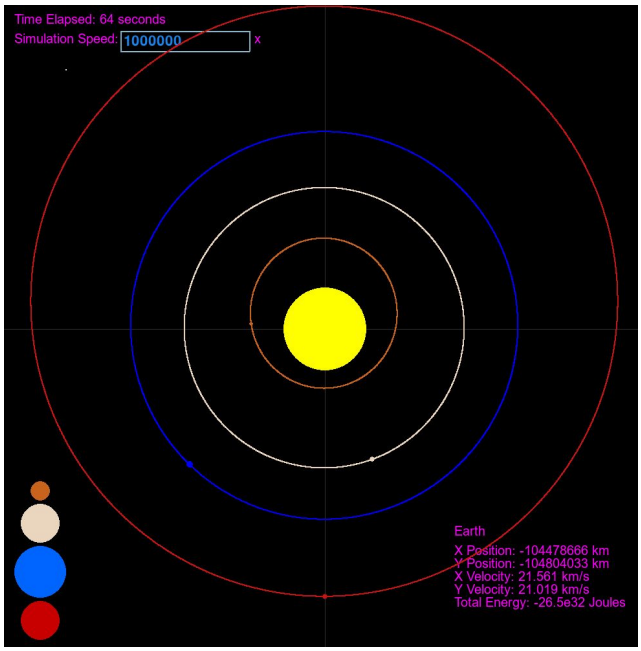
Applying Values from Reality

To test the simulation, we applied values from real life. This may sound difficult, but once we had proven the accuracy of our simulation, it was the easiest way to simulate other objects in space, as what works in real life should work in an accurate simulation. Of course, we would still have to change the scale depending on the sizes of the orbits.

Referring to a NASA fact sheet about each planet, we decided to use the perihelion of each planet's orbit as the starting point. This also means that each planet will be going at their maximum velocity, given by the NASA fact sheet once again.

Waiting for each planet's orbit to finish, we stopped the simulation with the **Spacebar** key when the planet reaches halfway through its orbit when it reaches its aphelion/minimum velocity. Comparing its value to that of the NASA fact sheet, we calculated the error to see how close our simulation is.

NASA Fact Sheet for Real Data



Planet	**True Aphelion (10^6 km)**	**Sim Aphelion (10^6 km)**	**Aphelion Percent Error**	**True Min. Vel. (km/s)**	**Sim Min. Vel (km/s)**	**Min. Vel Percent Error**
Mercury	69.817	69.531	**0.41%**	38.86	39.024	**0.42%**
Venus	108.939	108.878	**0.06%**	34.79	34.806	**0.05%**
Earth	152.099	152.072	**0.02%**	29.29	29.299	**0.03%**
Mars	249.229	248.927	**0.12%**	21.97	21.99	**0.11%**

Calculations

%Error = |expected-actual|/expected * 100

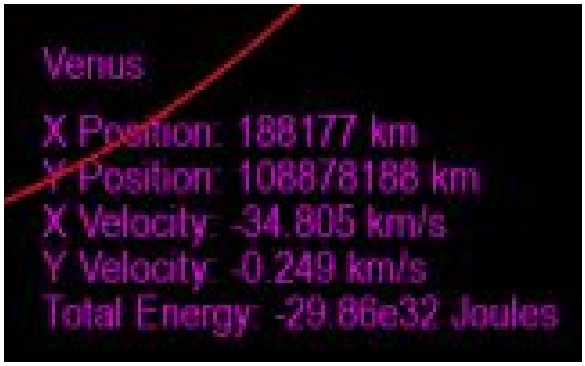
Mercury

	Mercury
Semimajor axis (10^6 km)	57.909
Sidereal orbit period (days)	87.969
Tropical orbit period (days)	87.968
Perihelion (10^6 km)	46.002
Aphelion (10^6 km)	69.817
Synodic period (days)	115.88
Mean orbital velocity (km/s)	47.36
Max. orbital velocity (km/s)	58.98
Min. orbital velocity (km/s)	38.86

Aphelion: |69.817-sqrt(69.526^2+.871^2)|/69.817 * 100 = 0.4% error
Min. orbital vel: |38.86-sqrt(39.024^2+.104^2)|/38.86 * 100 = 0.4% error

Venus

	Venus
Semimajor axis (10^6 km)	108.209
Sidereal orbit period (days)	224.701
Tropical orbit period (days)	224.695
Perihelion (10^6 km)	107.476
Aphelion (10^6 km)	108.939
Synodic period (days)	583.92
Mean orbital velocity (km/s)	35.02
Max. orbital velocity (km/s)	35.26
Min. orbital velocity (km/s)	34.79



Aphelion: |108.939-sqrt(.188177^2+108.878^2)|/108.939 * 100 = 0.056% error
Min. orbital vel: |34.79-sqrt(34.805^2+.249^2)|/34.79 * 100 = 0.046% error

Earth

Orbital parameters

Semimajor axis (10^6 km)	149.596
Sidereal orbit period (days)	365.256
Tropical orbit period (days)	365.242
Perihelion (10^6 km)	147.092
Aphelion (10^6 km)	152.099
Mean orbital velocity (km/s)	29.78
Max. orbital velocity (km/s)	30.29
Min. orbital velocity (km/s)	29.29

Aphelion: |152.099-sqrt(152.07^2+.825^2)|/152.099 * 100 = 0.017% error
Min. orbital vel: |29.29-sqrt(29.298^2+.261^2)|/29.29 * 100 = 0.031% error

Mars

	Mars
Semimajor axis (10^6 km)	227.923
Sidereal orbit period (days)	686.980
Tropical orbit period (days)	686.973
Perihelion (10^6 km)	206.617
Aphelion (10^6 km)	249.229
Synodic period (days)	779.94
Mean orbital velocity (km/s)	24.07
Max. orbital velocity (km/s)	26.50
Min. orbital velocity (km/s)	21.97

Aphelion: |249.229-sqrt(248.927^2+.042^2)|/249.229 * 100 = 0.12% error
Min. orbital vel: |21.97-sqrt(21.994^2+.048^2)|/21.97 * 100 = 0.11% error

Running

Environment

The sole dependency required is the PyGame library. The specific runtime environment on which the project was tested was PyGame version 2.0.0 on Python 3.8.6 on Windows 10 and Ubuntu 20.04, although any reasonably recent version of Python and operating system for which PyGame has official support should work, as we did not use any experimental features or OS-specific code. This means this simulation should be compatible for all computers as long they have the recent PyGame and Python version.

Execution

The simplest way of running the program is to open a terminal prompt, navigate to the folder containing the simulation, install PyGame with pip (`pip install pygame`), and run the main entry file with `python main.py`. The simulation will begin paused, and can be started by double-tapping the space key. Pressing the space key again will pause and resume the simulation, which can be useful for noting the exact movement values of a planet at a certain time, especially at higher speeds. The simulation speed can be altered with the input box in the upper left-hand corner; we have found one million to be a comfortable value. Moving planets have a colored trail to help visualize their orbits, although these dissipate after a certain amount of time in order to lessen the impact on performance. To view a planet's information, click the planet in the bottom left corner.

Demo Video