Orbital Simulator :

It is done in GDscript which is essentially 90% python and 10%C# and is the core programming language used in Godot, a game engine such as Unity. Godot also accepts 100% C# programming however, GDscript is more suitable and thus more efficient.

Focus :

1. To simulate the orbits of bodies around the Sun.
2. To be able to display orbits and information about the orbit and the body (Mass, velocity, vector velocity, angular momentum, apogee, perigee, semi-major axis, semi-minor axis and eccentricity)
3. To be able to add new objects given the mass, the initial velocity and the position.

Further :

1. To be able to corporate orbits with more than a 64bit-integer limit which is why the velocity is capped at 25000
2. To be able to simulate objects orbiting not relative the sun (Moon orbiting around Earth).
3. Simulating hyperbolic orbits and gravity assists.
4. Simulating N-body orbits (changing orbits relative to the position of other masses).

How it gets all the orbital parameters :

1. **Initializer (mass, apogee Velocity, position, name)**

```
func _init(initVel: Vector2, initPos: Vector2, mass: float, name: String):
    self.initVel = initVel
```

```
    self.initPos = initPos
    self.mass = mass
    self.Name = name
    # appending the object to a list of all the objects in the simulator
    Global.objects.append(self)
    self.get_stuff.call_deferred()
```

get_stuff() function :

2. **Main root gravity influence (RGI) :**
    a. Runs the gravity equation on every body and thus gets the main body (GmM/D^2)
    b. Gets the strongest body

```
for object in Global.objects :
    if object != self :
        # Get distance squared whilst converting to kilometre scale
        var distance = ((self.initPos/Global.AU).distance_squared_to(object.initPos/Global.AU))

        # Get Force (F = (G*M*m)/d^2, if the force is stronger than the strongest, that is the new strongest
        var force = (((object.mass*1e+23)*(self.mass*1e+23))*Global.G)/distance

        if force >= strongestForce :
            strongestObject = object
            strongestForce = force

    self.strongestObject = strongestObject
```

3. **Gets the Standard Gravitational Parameter (SGP) for these bodies :**
    a. G(M+m)

```
# Getting Standard gravitational parameter
self.SGP = Global.G*(self.mass*1e+23 + self.strongestObject.mass*1e+23)
```

4. **Getting the apogee (metres) (furthest point from the RGI)**

   a. Distance from the initial position of both the objects (since all the objects start at furthest point)

```
# Getting apogee distance
self.apogee = (self.initPos/Global.AU).distance_to(self.strongestObject.initPos/Global.AU)
```

5. **Getting the angle from the two bodies (degrees) :**
   a. **Some trigonometry (atan(yVector/xVector)) = theta**

```
   # Getting angle
   self.angle = rad_to_deg((self.initPos).angle_to_point(self.strongestObject.initPos))-180
```

6. **Getting the semi-major axis (metres) (half of the distance between the two furthest points in an orbit) :**
   a. **(SGP / (2 (SGP / apogee – velocityAtApogee^2 / 2)))**

```
# Getting semi-major axis
self.semiMajor = self.SGP / (2 * (self.SGP / self.apogee - self.initVel.distance_to(Vector2.ZERO)**2 / 2))
```

7. **Getting the perigee (m) (opposite of apogee) :**
   a. **Absolute value(2 * semimajor - perigee)**

```
# Getting perigee val
self.perigee = abs(2*self.semiMajor-self.apogee)
```

8. **Getting the eccentricity (how much of an ellipse the orbit is)**
   a. **(Apogee-perigee) / (Apogee+perigee)**

```
   # Getting the eccentricity
   self.eccentricity = (self.apogee-self.perigee)/(self.apogee+self.perigee)
```

9. **Getting the semi-minor axis (m) (opposite of the semi-major axis)**
   a. **Semi-major * sqrt(1-eccentricity)**

```
# Getting the semi-minor axis
self.semiMinor = self.semiMajor * sqrt(1-self.eccentricity**2)
```

10. **Getting the angular momentum**

a. **Mass/Velocity/R**

```
# Getting the angular momentum
self.angularMomentum = abs(self.mass*1e+23*(self.initVel.distance_to(Vector2.ZERO)*self.apogee))
```

11. **Pixelization by the scale (1AU = 200 pixels => 1.337e-9 pixel/meter)**

```
# Pixelizing all the values
self.apogee = self.apogee * Global.AU
self.semiMajor = self.semiMajor * Global.AU
self.perigee = self.perigee * Global.AU
self.semiMinor = self.semiMinor * Global.AU
```

12. **Find the foci (the other focus point of the elliptical orbit)**
    a. **apogeePos – perigeePos**

```
self.focii = Vector2(self.initPos-(self.initPos.normalized()*self.perigee))
```

13. **Getting the second Kepler's law constant for correction :**
    a. **Angular momentum / mass**

```
self.secondLaw = (self.angularMomentum/(self.mass*1e+23))
```

How does it simulate the movement :

Mainly, the program uses the gravity equation (depicted in the program as the get_force(self) function) between two objects to simulate this :

GmM/D^2

```
func get_force() :
	var forces: Array
	var netForce: Vector2

	var object = self.strongestObject
	# Get distance squared whilst converting to kilometre scale
	var distance = ((self.pos/Global.AU).distance_squared_to(object.pos/Global.AU))
	# Get Force (F = (G*M*m)/d^2
	var force = (((object.mass*1e+23)*(self.mass*1e+23))*Global.G)/distance

	# Get the vector for the direction of force
	force = (self.pos).direction_to(object.pos)*force
```

```
  # Getting the net accelaration with the formula F = MA thus A = F/M
  var netAccel = force/(self.mass*1e+23)
  # Converting to pixel scale
  netAccel = netAccel * Global.AU
```

```
realAc += (body.get_force() * Global.timeStep**2*delta)
global_position += realAc
```

It also runs some correction algorithms :

1. Kepler's first law (the sum of the distances of the orbiting object and the two foci always equals double the semi-major axis)

```
FLC = (body.focii.distance_to(global_position) + Vector2.ZERO.distance_to(global_position)2*body.semiMajor)*global_position.direction_to(body.strongestObject.pos)
global_position += FLC/2
```

2. Kepler's second law (the of the sector of the orbit is always equal to Darea/Dtime (DA/DT) and also can be derived and used (calculus very useful)

```
SLC = body.secondLaw/((0.5*(global_position.distance_to(Vector2.ZERO)
    /(Global.AU))*((global_position.distance_to(Vector2.ZERO)/(Global.AU))
    *abs(rad_to_deg(global_position.angle_to_point(Vector2.ZERO))-
    (rad_to_deg(oldpos.angle_to_point(Vector2.ZERO))))))/Global.timeStep)/1.103

if abs(SLC-1) < 0.5 :
  body.velocity = realAc*SLC
else :
  body.velocity = realAc

```