Orbital Gravity Simulation

An implementation of newtonian mechanics in Scala and OpenGL

Freddie Poser

2016

An implementation of Newtonian mechanics designed to simulate gravity and interactions between particles in two dimensions.

Contents

List of Figures

1 Setup

I am building this project in Scala, a JVM programming language. The advantage of Scala is that it is multi-paradigm, allowing me to use OOP and functional programming. I am using LWJGL, an OpenGL wrapper for Java, which allows me to do proper graphics for the simulation and watch the particles in real-time.

1.1 Universe

The Universe class manages the overall simulation. The key part of Universe is its list of all the particles in existence. The Universe also contains the GraphicsManager which controls all of the rendering and adds a layer of abstraction between the simulation and the OpenGL bindings.

1.2 Particle

The Particle class represents every single object in the system. The particles all have a ParticleType which defines their intrinsic properties: radius, mass, colour as well as a position vector. Each particle is able to render itself which involves drawing the appropriate circle at its location, the path that it has been on and all of the forces acting on it.

The key physics is contained in the methods interact(that: Particle) and runTick():

interact Interact is called by the universe class before every tick is run. For every unique pair of particles in system interact is called once. It takes in the particle to interact with and returns a list of forces generated by the interaction.

These forces are then taken by the universe class and applied to their target particles, storing them up for the next time runTick is called.

runTick RunTick is called on every particle each time the simulation updates after all of the interactions are computed. It returns no value but instead computes the effects of the forces and updates the particles position and velocity accordingly.

The effective input to runTick is a list of forces and it then performs the following calculations (t = 1 in this case):

$$\vec{F_R} = \vec{F_1} + \vec{F_2}...\vec{F_n} \tag{1}$$

$$\vec{a} = \frac{\vec{F_R}}{m} \tag{2}$$

$$\vec{v} = \vec{u} + \vec{a}t \tag{3}$$

$$\vec{r} = \vec{r} + \vec{v}t \tag{4}$$

Figure 1: Movement equations used in Particle class

1.3 Rendering

Rendering uses code that I originally wrote for a simple game engine (Cotton Game)[1]. This code does a number of things:

GraphicsManager The graphics manager class controls the basic screen setup and the standard OpenGL commands that need to be called to simply get rendering working.

Sync The sync class is a utility that allows the programmer to control how often a given method is run in a non-blocking fashion. In this project it is used in three places:

- Render Sync - locks the render loop at 60FPS
- Update Sync - allows the user to choose how fast to run the simulation
- UX Sync - separates the input polling from the update sync so that even if the game is paused the user can still control it

Render The Render object is a static collection of utilities that make rendering easier to work with, they abstract away from the details of OpenGL so that code is cleaner and easier to debug

[1] `https://github.com/vogon101/CottonGame`

2 Gravity

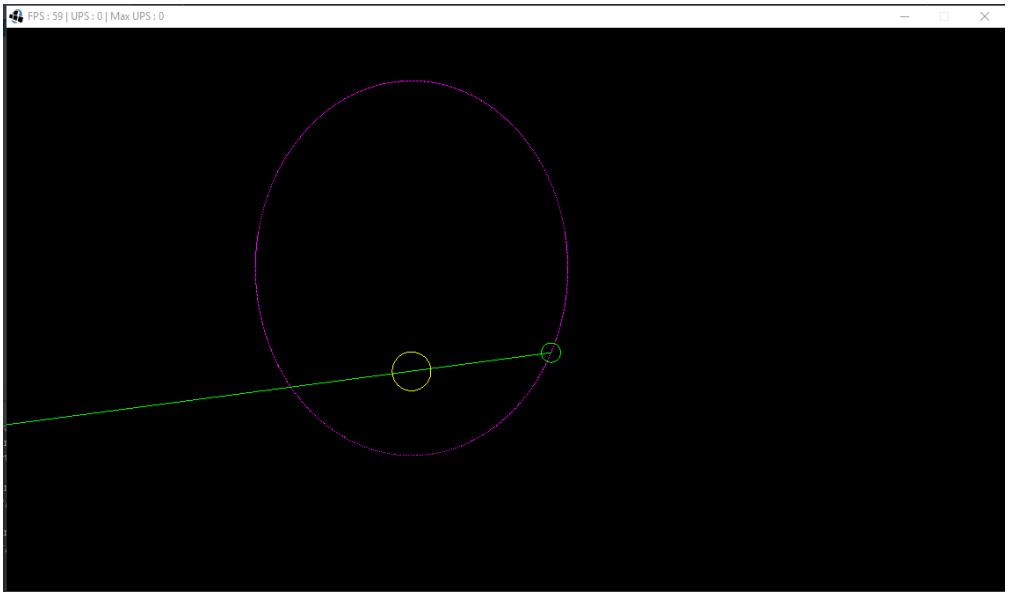
For gravity I use the Newtonian equation which provides the magnitude of the force between two bodies. I calculate this during the interact function so the force is calculated between every pair of particles in the universe.

$$F = G\frac{m_1 m_2}{r^2} \tag{5}$$

Once I have the magnitude I find its direction by finding the angle between the two position vectors and it is applied to each body.

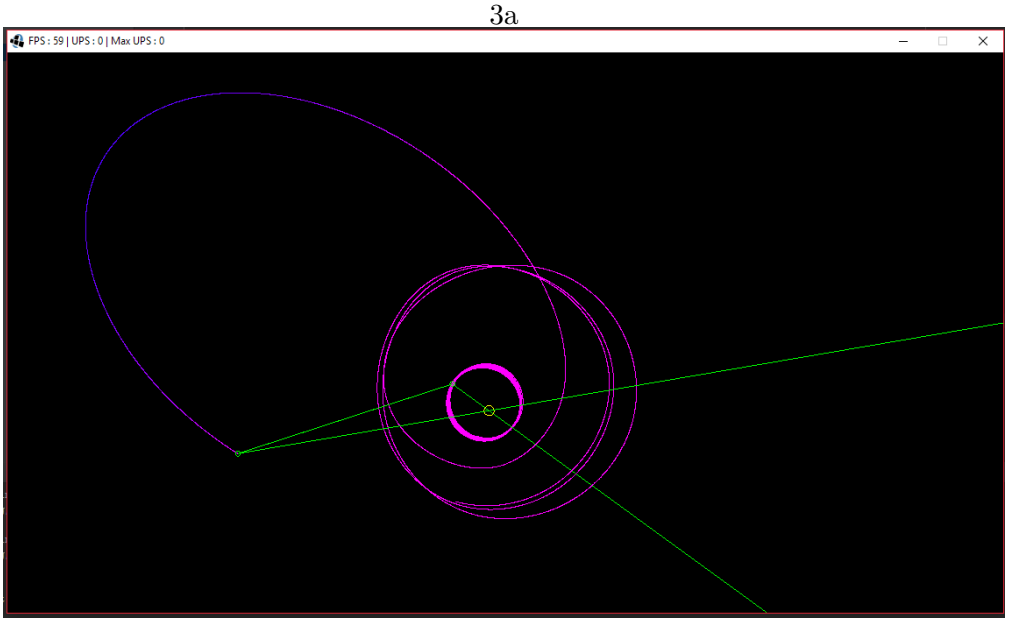
```
1 val distance = that.position.distance(position)
2 val gravForce = (GRAVITATIONAL_CONSTANT * mass * that.mass) / Math
    ↪ .pow(distance, 2)
3
4 //Final forces from gravity
5 List(
6   Force(that, gravForce, (this.position - that.position).theta),
7   Force(this, gravForce, (that.position - this.position).theta)
8 )
```

Figure 2: The gravity calculations from the Particle class



(a) One body in orbit, the orbit is perfectly stable in this case. The green line is the force on the planet.

3a



(b) Two bodies in orbit, one is on a more stable orbit whilst the outer one moves more erratically

3b

Figure 3: Examples of gravity with one and two bodies in orbit around a fixed star

3 Momentum and Collisions

Once I had implemented gravity I decided to add in simple 2D collisions to make the simulation slightly more realistic.