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

1 Plan 3

2 Setup 4

2.1 Universe 4

2.2 Particle 4

2.3 Rendering 5

3 Gravity 6

4 Momentum and Collisions 8

5 User Control 11

6 Evaluation 13

List of Figures

1 Movement equations used in Particle class 5

2 Newton's law of Universal Gravitation 6

3 The gravity calculations from the Particle class 6

4 Examples of gravity with one and two bodies in orbit around a fixed star 7

a One body in orbit, the orbit is perfectly stable in this case. The green line is the force on the planet. 7

b Two bodies in orbit, one is on a more stable orbit whilst the outer one moves more erratically 7

5 Conservation of momentum and kinetic energy equations 8

6 Equations to find the normal and tangent components of the velocity . . . 8
7 Equation to find the final velocity of particle 1 in the normal direction . . 9
8 Combination of the normal and tangent components of the velocity into a final vector 9
9 The code that calculates the result of collisions 10
10 Key bindings for the simulation 11
11 Commands for the simulation 11
12 Information about the simulation displayed in the top left corner 12

1 Plan

The plan for this project is to simulate the motion of round particles in two dimensions. The particles will exert Newtonian gravity on each other and will be able to collide. This will all be implemented using Newtonian mechanics, namely Newton's second law and the kinematic laws of motion.

For the motion I will assume that acceleration is constant for 1 second and so the simulation will update in steps, each representing 1s. It may be possible in the future to decrease this for more accurate simulations (or increase it for faster ones).

I am going to build this in with Scala, a JVM programming language. The advantage of Scala is that it is multi-paradigm, allowing me to use OOP and functional programming concepts. For the graphics I will use an OpenGL wrapper library called LWJGL[1].

Using this will allow me to watch the simulation in real-time rather than look at the data it outputs after the fact. I may implement a way of getting the raw data out of the simulation as well so that I can simulate real situations with greater ease.

[1]`https://www.lwjgl.org/`

2 Setup

All of the source code is available at `https://github.com/vogon101/NewtonianMechanics`

Below is a list of a few of the core classes in the simulation

2.1 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.

2.2 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 = length of tick in this case):

These equations are based first on Newton's second law of motion which gives that the net force on an object is equal to its mass times its acceleration ($\vec{F_R} = m\vec{a}$). With the acceleration I find the new velocity by multiplying it by the time it acts for and adding it to the old velocity. This model assumes constant acceleration and velocity for the duration of each tick. This means that the accuracy of the model is linked to the length of the tick, with the shorter they are, the better the simulation. The position is then found by applying the velocity in the same way.

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

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

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

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

Figure 1: Movement equations used in Particle class

2.3 Rendering

Rendering uses code that I originally wrote for a simple game engine (Cotton Game)[2]. 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

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

3 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. [3]

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

Figure 2: Newton's law of Universal Gravitation

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.

```
val distance = that.position.distance(position)
val gravForce = (GRAVITATIONAL_CONSTANT * mass * that.mass) / Math
    ↪ .pow(distance, 2)

//Final forces from gravity
List(
  Force(that, gravForce, (this.position - that.position).theta),
  Force(this, gravForce, (that.position - this.position).theta)
)
```

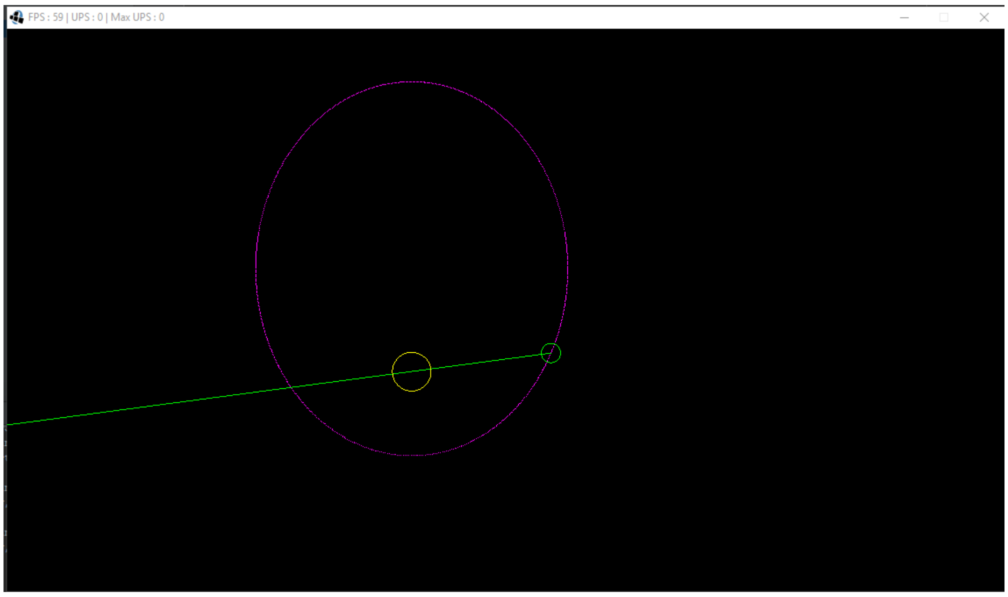
Figure 3: The gravity calculations from the Particle class

This creates two vector forces, one on each object which are used during the run tick phase.

Whilst Newtonian gravity has been superseded by general relativity it is still a good approximation in almost all situations where great precision is not required and where the bodies move at low speeds. I have used it in this project because it is much easier to implement[4]

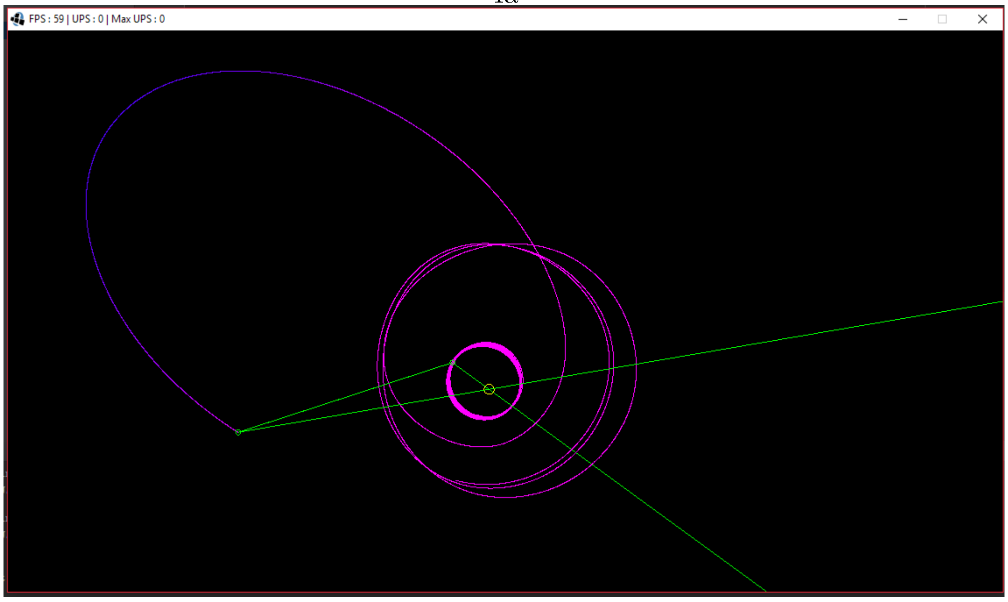
[3]Source: `https://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation`

[4]`https://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation#Problematic_aspects`



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

4a



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

4b

Figure 4: Examples of gravity with one and two bodies in orbit around a fixed star

4 Momentum and Collisions

Once I had implemented gravity I decided to add in simple 2D collisions to make the simulation slightly more realistic. This turned out to be far more difficult than I had expected. The problem was that many of the equations used for this are very complicated when used with vectors in two dimensions.

I found online a set of equations designed for fully elastic vector collisions. These are derived from two equations: Conservation of momentum and of kinetic energy:

$$\frac{1}{2}m_1u_1^2 + \frac{1}{2}m_2u_2^2 = \frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 \tag{6}$$

$$m_1u_1 + m_2u_2 = m_1v_1 + m_2v_2 \tag{7}$$

Figure 5: Conservation of momentum and kinetic energy equations

These equations give two equations that allow me to work out the velocity of the particles after the collision. However, this only works in one-dimension so to find the collision between particles in my simulation I first have to transform the velocity vectors so that the collision is as if it were in 1D.

This is done by taking the unit vectors in the directions normal and tangent to the collision. It is then possible to find the velocities of each particle in these directions by taking the dot product of the vectors. In the following equations $\vec{normal}$ and $\vec{tangent}$ are the unit tangent and normal vectors respectively.

I will only consider the first particle as because of Newton's Second Law of motion the force produced by this collision will be the same on both particles. This means that once I find the final velocity of the first particle I can find the change in momentum and thus the magnitude of the force by using $F = \frac{\delta p}{\delta t}$

$$u_{n1} = \vec{normal} \cdot \vec{u_1} \tag{8}$$

$$v_{t1} = \vec{tangent} \cdot \vec{u_1} \tag{9}$$

$$u_{n2} = \vec{normal} \cdot \vec{u_2} \tag{10}$$

Figure 6: Equations to find the normal and tangent components of the velocity

As the velocity in the tangent direction remains the same after the collision, I only need to deal with the normal velocity. It is then possible to calculate the final normal component to the velocity with the following formula (derived from the conservation of momentum and kinetic energy).

$$v_{n1} = \frac{u_{n1}(m_1 - m_2) + 2m_2u_{n2}}{m_1 + m_2} \tag{11}$$

Figure 7: Equation to find the final velocity of particle 1 in the normal direction

Finally I transform the two one-dimensional vectors into a single 2D vector velocity.

$$\vec{v_{n1}} = \vec{normal} * v_{n1} \tag{12}$$

$$\vec{v_{t1}} = \vec{tangent} * v_{t1} \tag{13}$$

$$\vec{v_1} = \vec{v_{n1}} + \vec{v_{t1}} \tag{14}$$

Figure 8: Combination of the normal and tangent components of the velocity into a final vector

As my simulation is built around forces I want to find the forces produced from this. To do this I use the equation above to calculate the magnitude and then create two forces as vectors of that length: One in the direction of $p_1 - p_2$ (where p_1 is the position vector of particle 1) and one in the opposite direction.

The code that calculates this is listed below:

```scala
/**
* Calculate the forces generated by the collision of this particle
    ↪  and another
* @param that The particle to collide with
* @return List of the forces generated
*/
def collide (that: Particle): List[Force] = {

        val normal = (that.position - this.position).normalize
        val tangent = normal.tangent

        val u1 = this.velocity
        val u2 = that.velocity

        val m1 = this.mass
        val m2 = that.mass

        val u1n: Double = u1 dot normal
        val u1t: Double = u1 dot tangent
        val u2n: Double = u2 dot normal

        val v1t = u1t

        val v1n = ( ((m1-m2) * u1n) + (2 * m2 * u2n) ) / (m1 + m2)

        val vect_v1n = normal * v1n
        val vect_v1t = tangent * v1t

        val v1 = vect_v1n + vect_v1t

        val impulse1 = ((v1 * m1) - (u1 * m1)).length / DELTA_TIME

        if (impulse1 < 0.0000000001) {
                List()
        } else {

                val forces = List(
                Force(this, impulse1, (this.position - that.
                    ↪ position).theta, alwaysDraw = true),
                Force(that, impulse1, (that.position - this.
                    ↪ position).theta, alwaysDraw = true)
                )

                forces
        }
}
```

Figure 9: The code that calculates the result of collisions

5 User Control

Whilst the majority of this project is the physics, I think it is important that it is possible to interact with the simulation to be able to watch it more effectively. To this end I have built a command system which allows the user to control parts of the simulation along with certain key bindings.

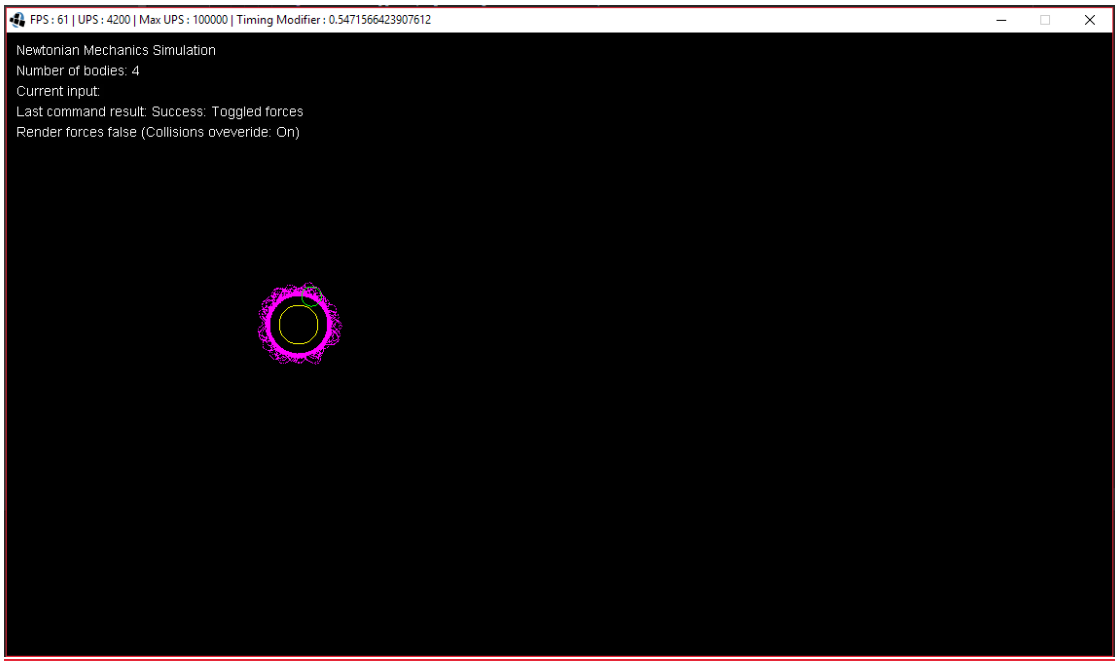
- **+/-** → Speed up/slow down the simulation
- **LSHIFT** + **+/-** → Zoom in/out
- **LEFT/RIGHT** → Pan around the simulation horizontally
- **UP/DOWN** → Pan around the simulation vertically

Figure 10: Key bindings for the simulation

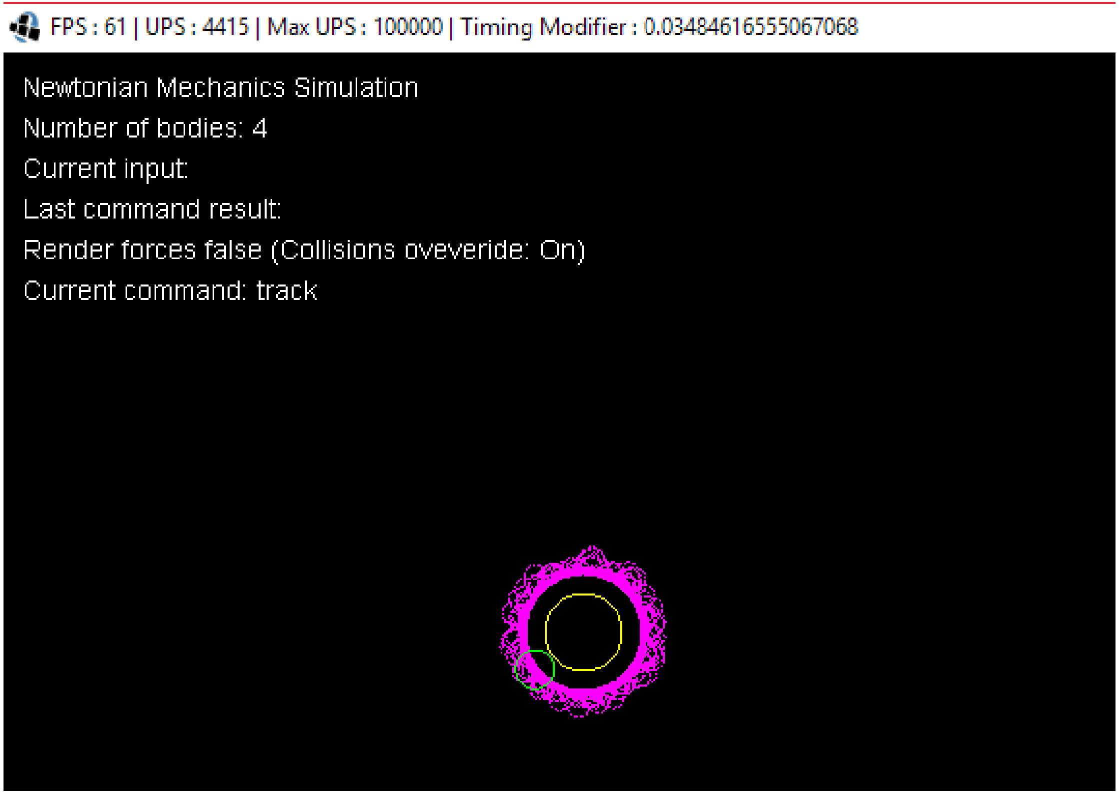
- track <particle num> → Track a certain particle
- cleartrack → Clear the tracker, allow free movement
- show → Print a list of particles to STD Out
- forces → Toggle rendering of forces
- col <on/off> → Render collision forces even if force rendering disabled
- particle <particle num> → Centre view on a particle
- pause → Pause the simulation
- slow → Slow down the simulation to 10 UPS
- 1 → Set the maximum UPS to 1 (real time)
- speed <UPS> → Set the speed of the simulation (sets max UPS)

Figure 11: Commands for the simulation

I also implemented the ability for the user to track a specific particle over the course of the simulation and to export that data to excel in a .csv format. This allows the user to watch simulations where the scales are too big for the realtime graphics.



```
FPS : 61 | UPS : 4200 | Max UPS : 100000 | Timing Modifier : 0.5471566423907612
Newtonian Mechanics Simulation
Number of bodies: 4
Current input:
Last command result: Success: Toggled forces
Render forces false (Collisions oveveride: On)
```



```
FPS : 61 | UPS : 4415 | Max UPS : 100000 | Timing Modifier : 0.03484616555067068
Newtonian Mechanics Simulation
Number of bodies: 4
Current input:
Last command result:
Render forces false (Collisions oveveride: On)
Current command: track
```

Figure 12: Information about the simulation displayed in the top left corner

6 Evaluation