Orbital Gravity Simulation

An implementation of newtonian mechanics in Scala and OpenGL

Freddie Poser

2017

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

Contents

List of Figures

1 Plan

This project aims 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.

Below is an overview of the features that I implemented in the program. They are covered in the order that I implemented them.

[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 the core classes in the simulation with an explanation of their function within 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 and colour. They also have a position vector and initial velocity. 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. It also contains all of the key physics in the simulation.

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 This 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 makes rendering code easier to work with, it abstracts 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.

```
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  )
```

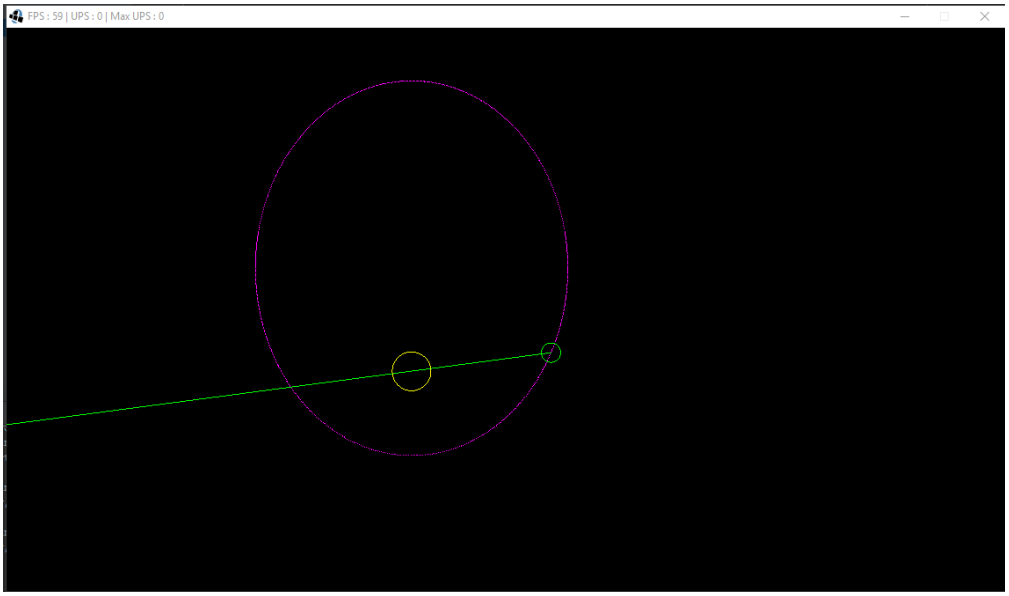
Figure 3: The gravity calculations from the Particle class

This creates two vector forces, one on each object which are used during the `runTick` 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 easier to implement.[4]

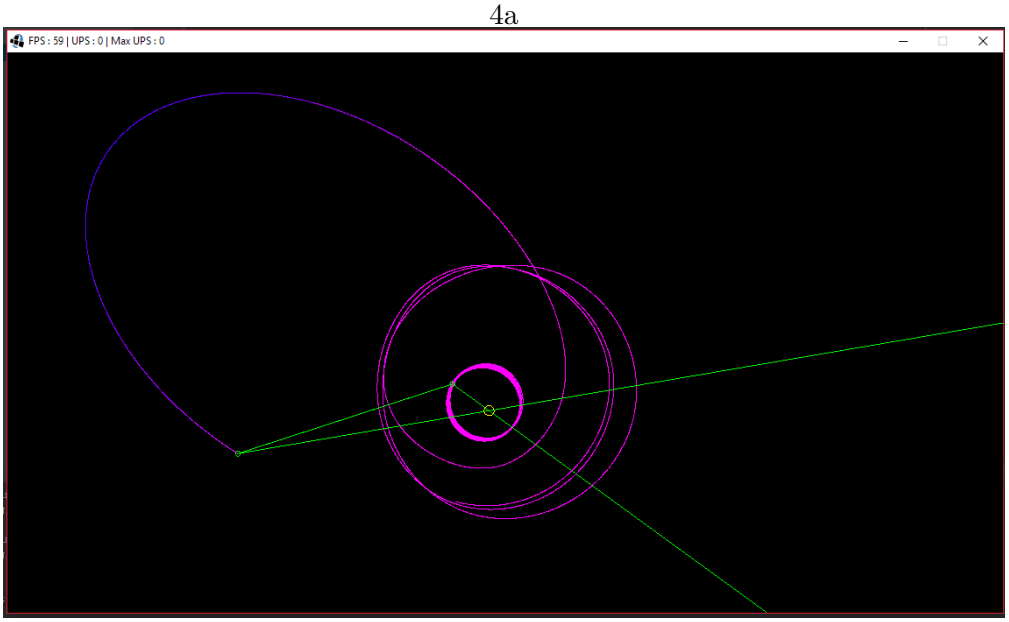
[3] https://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation

[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

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

I found online[5] 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).

[5]Source: `http://vobarian.com/collisions/2dcollisions2.pdf`

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

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:

```
/**
* 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 focus of this project is the physics, I thought it was important that to be able to interact with the simulation in order to watch it more effectively. To this end I 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

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 real time graphics.

- 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)
- noCap → Set the max UPS to -1, effectively removing the cap on update speed
- scale <scale> → Set the zoom so that one of the scale = 1 pixel (options: mm, cm, m, km)
- zoom <zoom> → Set the zoom to a specific number $zoom \in (0, \sim 1.7 \times 10^{308})$
- pauseCollision → Pause the simulation on every collision
- exit → Exit the simulation
- run <simulation> → run a specific simulation
- simulations → List the simulations available
- restart → Restart then current simulation

Figure 11: Commands for the simulation

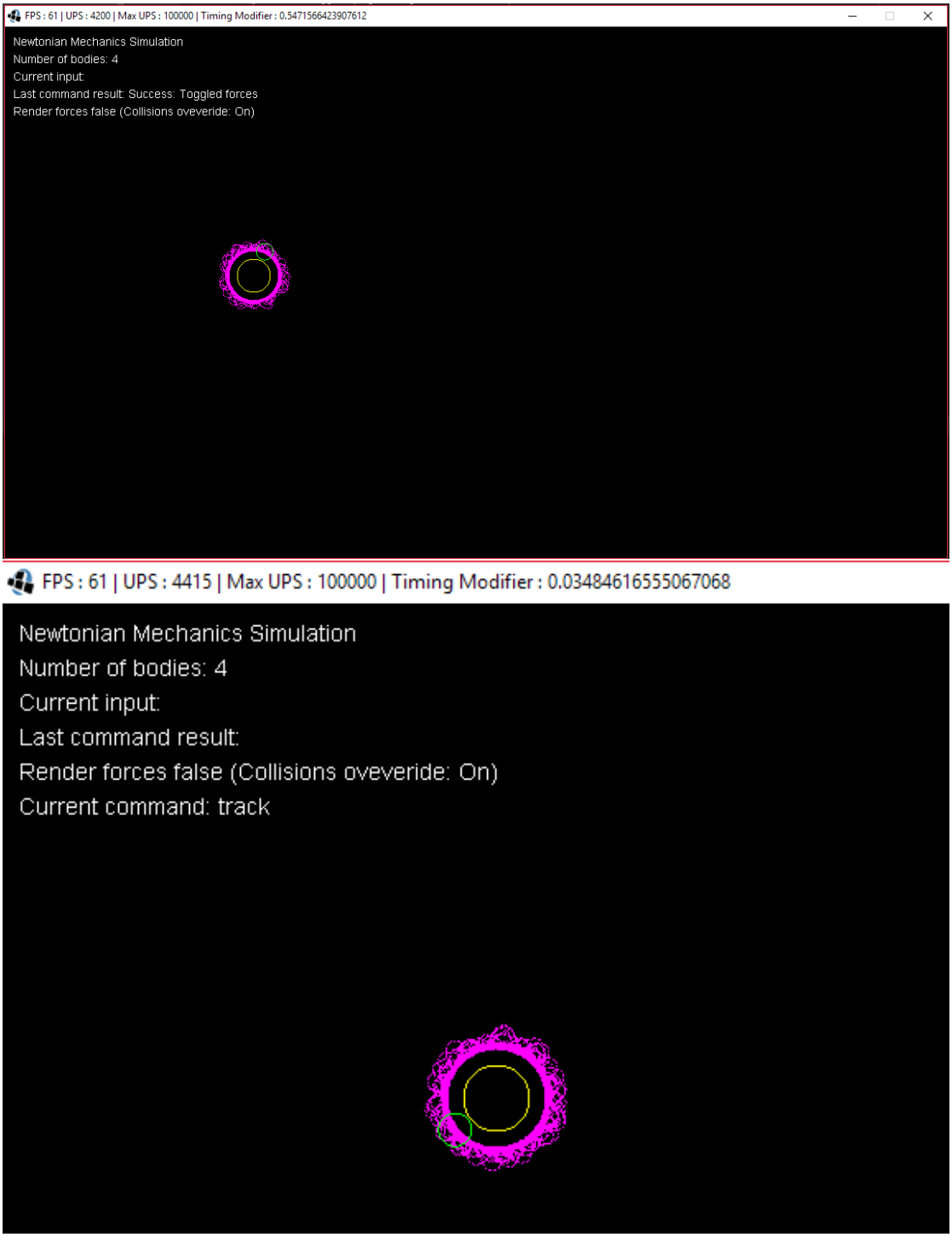


Figure 12: Information about the simulation displayed in the top left corner

6 Diary

6.1 Dec 2016

In the session before the Christmas break I decided that I wanted to create a simulation of Gravity. I decided on this because it seemed like a do-able project which would encompass a number of physical concepts (Newtonian gravity, constant acceleration formulae, etc) and some interesting programming concepts.

6.2 9th Jan 2017

In this session I fleshed out my idea. I decided to use OpenGL for rendering the simulation. I also implemented the core physics of the simulation: the gravity. I did this by first creating the Particle class and then implementing forces and movement. I then had every Particle exert a gravitational force on every other Particle.

6.3 16th Jan

I started work on the rendering system so that I could visualise my simulation. I used OpenGL for this. In this session I only got as far as simply rendering the circles for the particles

6.4 23rd Jan

By this session I had completed the core rendering with particles, forces and paths all being rendered to the screen. I then decided to try and implement elastic collisions. This ended up being more difficult than I had thought so I worked on this for a while.

6.5 30th Jan

In this session I continued working on the collisions but I also implemented a command-line system so that the user could control the simulation. This included all the commands listed in the **User Control** section.

6.6 6th Feb

In this session I implemented some sort of working collisions. Whilst they are not as good as I would have liked they work. I also created in this session a way of outputting the path of a particle over time to a `.csv` file.

7 Evaluation

7.1 Gravity

Overall I am happy with how this project turned out. The gravity portion works very well. To test this I set up a simulation of earth's orbit using real values for the mass and relative velocity of earth and the sun along with the distance between them.

Because of the scale difference I exported the data from the simulation to a `.csv` and created a graph of earth's position over 365 simulated days.

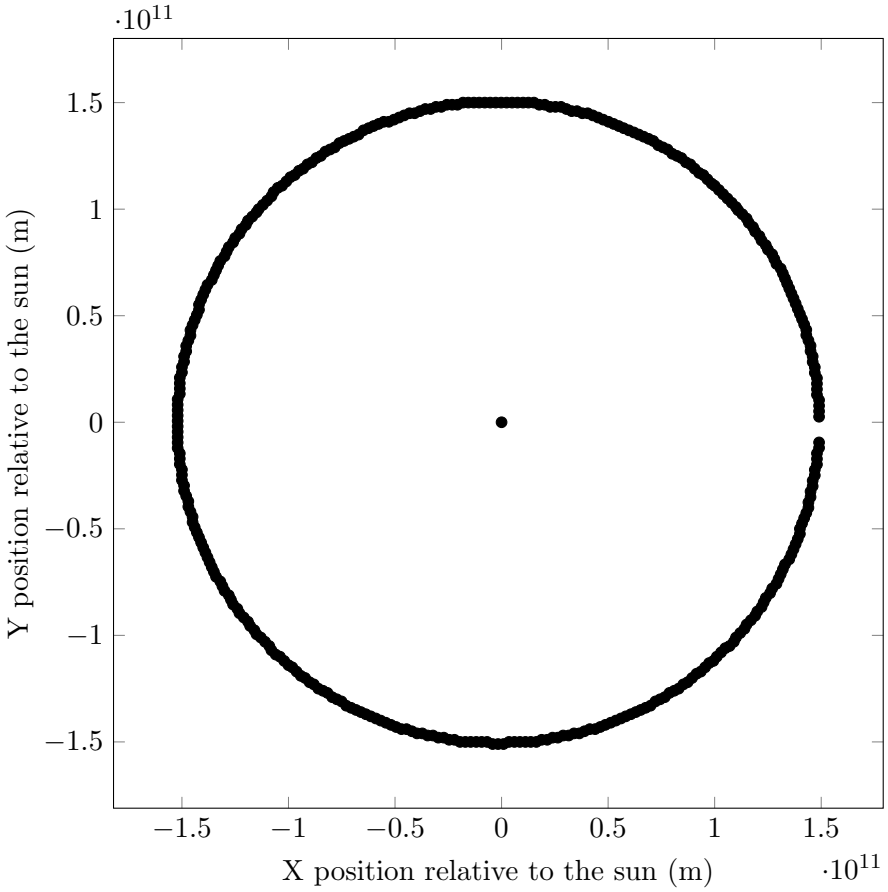


Figure 13: Simulated orbit of earth over 365 days

As you can see in Figure 13, the simulation correctly simulates the orbital period of the earth to be about 365 days. This is very close to the actual period of 365.256 days. The differences are probably down to the precision of the data I used to set up the simulation and the nature of that data being averages.

I also simulated the orbit of the moon around the earth. As you can see (Figure 15) it correctly simulates the orbital period to be just over 27 days (the true value is 27.323).

The data used are listed in the table below

Particle	Mass (kg)	Distance (m)	Speed (m/s)
Earth	5.9724×10^{24}	149×10^{9} (from sun)	30×10^{3} (relative to sun)
Sun	1.989×10^{30}	–	–
Moon	7.346×10^{22}	385×10^{6}	1022 (relative to earth)

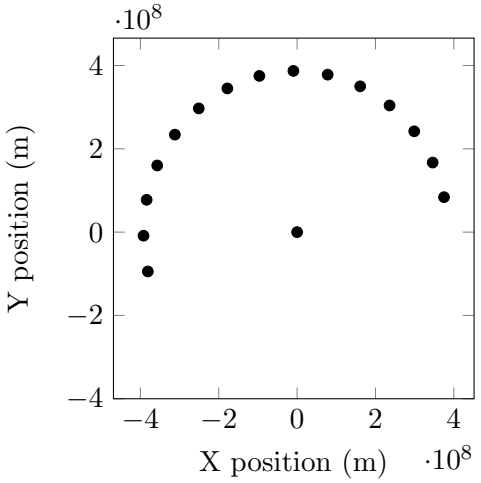
Figure 14: Data used in simulations of real-world orbits

These examples show that gravity is simulated correctly and with an acceptable degree of accuracy.

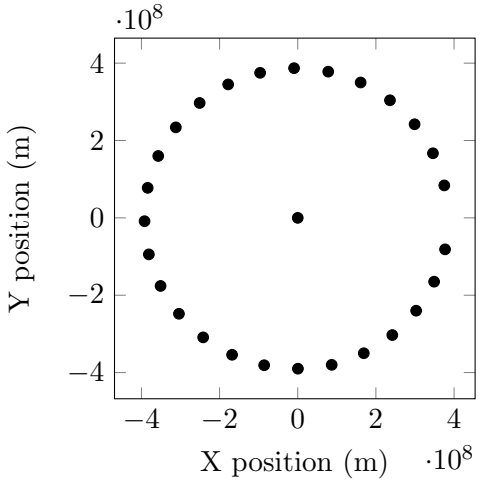
7.2 Collisions

The collisions I implemented on the other hand were less successful. I was unable to find a simple solution to elastic collisions in two dimensions. To try and overcome this I used the method I discussed in section 4. This gives what seem like plausible results when watched in the simulation and it successfully conserves momentum.

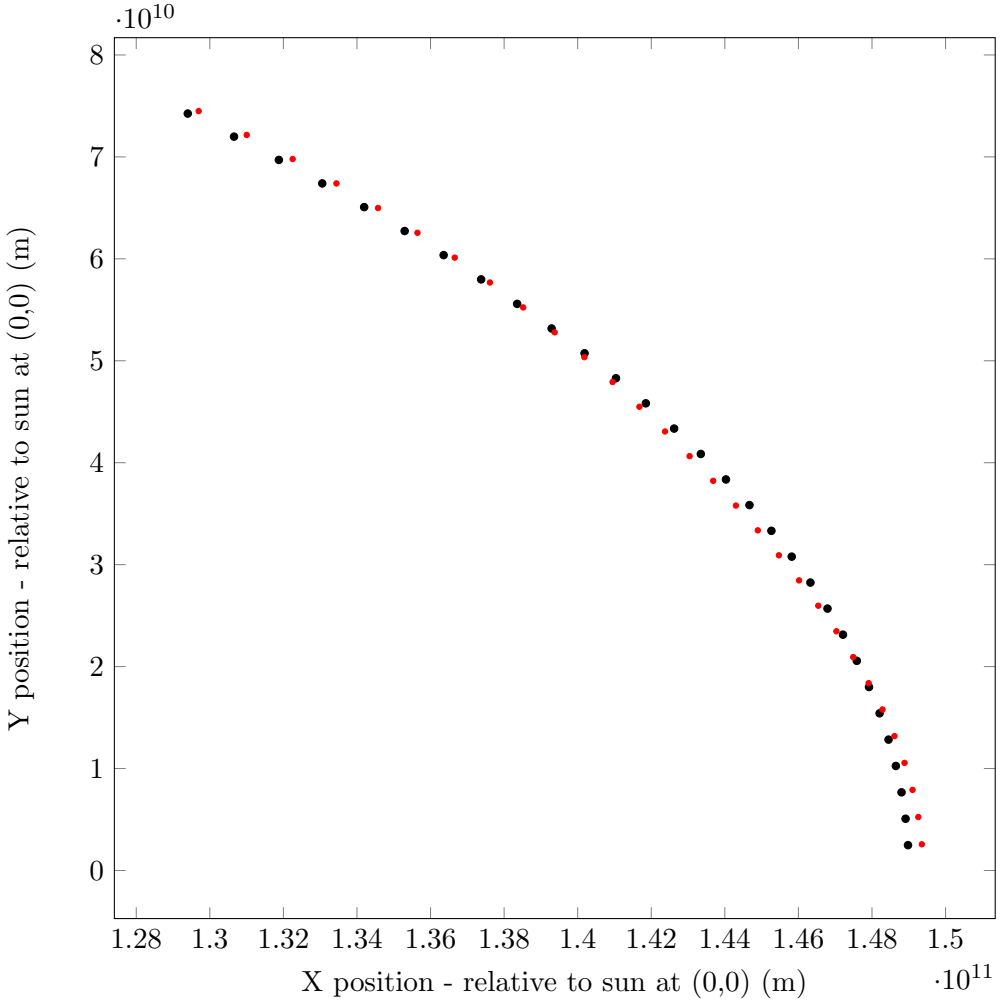
The problem is that the collisions do not preserve kinetic energy, infact they create it. If I had more time I would spend longer and derive equations myself. As this project was meant to be about gravity this is not too much of a problem. Figures 16 and 17 show some tests of the collisions



(a) Simulated orbit of moon over 15 days

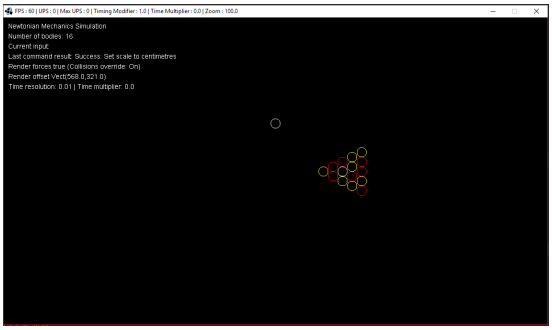


(b) Simulated orbit of moon over 27 days

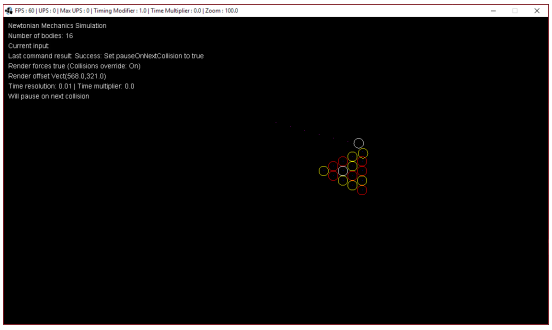


(c) Simulated orbit of moon (red) and earth (black)

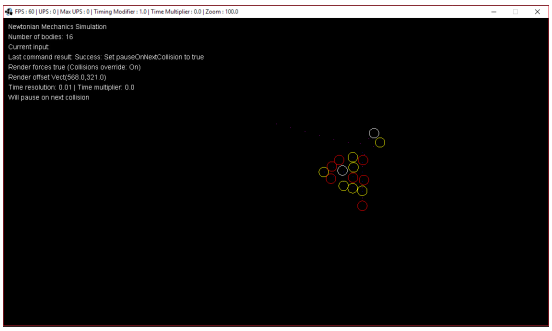
Figure 15: Simulated orbit of the moon and earth



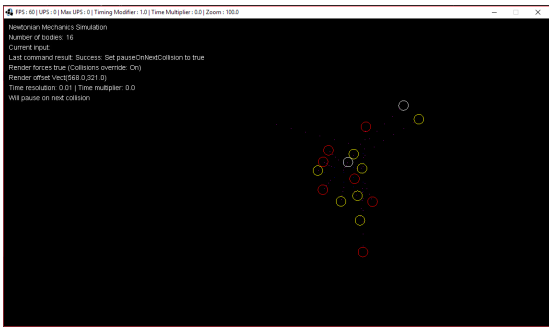
(a) Pool simulation before any collisions



(b) First impact of white ball on the triangle

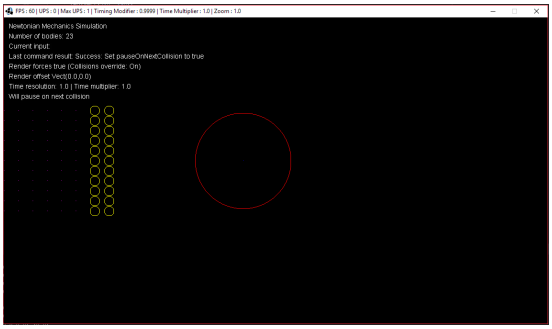


(c) More collisions lead the triangle to split up

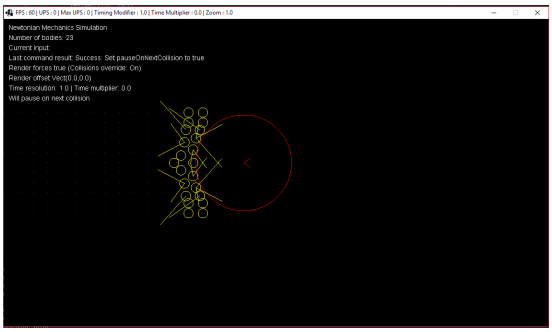


(d) Result of the break

Figure 16: A pool-style break with 1kg balls, with 10cm radii. The white has an initial velocity of 6.17m/s



(a) The set-up for the simulation. The yellow balls are travelling right.



(b) Result of the simulation with forces drawn to show the collisions

Figure 17: A test simulation that demonstrates the conservation of momentum. Momentum before and after is $1100kgm/s$

8 Running the simulation

To run the simulation yourself a runnable build is available at `https://static.vogonjeltz.com/physics/` for windows. To run it: download the zip file and extract it. Then double click on the `.bat` file. The simulation requires java to work.

Once the simulation has started, click on the window to enable control. You can then use the key-bindings and commands listed in the **User Control** section. The default simulation is a pool-like set-up.

To run another simulation use the `run` command. The simulations available are listed below:

Name	**Description**
test	A test of the collision code
collision	A simple demonstration of a 1D collision
pool	A pool-like set-up but with larger balls and higher masses
pool2 (default)	The default pool simulation
orbit	A three-particle orbit simulation
earth	A simulation of earth's orbit that outputs to a `.csv`

Figure 18: The simulations available to run