

# Orbits 5: A Graphical N-body Gravitational Simulator

Leonardo Carnovale

April 29, 2019

# Contents

<b>1</b>	<b>Simulator and Graphics API</b>	<b>2</b>
1.1	Relationship between simulator and API . . . . .	2
1.2	Function of the Simulator . . . . .	2
1.3	Function of the API . . . . .	2
<b>2</b>	<b>Simulation Algorithms</b>	<b>4</b>
2.1	Matrix Mechanics . . . . .	4

# Chapter 1

## Simulator and Graphics API

### 1.1 Relationship between simulator and API

The graphical display should not be a design specific to the operation of the simulator, so as to allow significant improvement and modification in future versions. This is to say that the simulator and the graphics API should operate independently, and communication or interaction should mainly be from the simulator to the camera, then to the API. The simulator will need to prioritise simulation, and spend all its time on this task. It should not try to optimize output for the API, this can be done within the API.

### 1.2 Function of the Simulator

The job of the simulator is simply to calculate the position of all objects after a time step interval. The main program will be frequently interacting with it. The simulator should have the following methods:

- Add an object to the simulation.
- Remove an object from the simulation.
- Change the default time step interval.
- Perform a step.
- Perform  $n$  steps.

There are multiple ways to perform this simulation, and the current preference is the leapfrog method. Other methods should be available to be chosen at creation of the simulator object.

The above methods will likely not need to have a return value. All information about the simulator such as the current time step, number of particles, or the information about a particular particle, should be available via additional ‘get’ methods.

### 1.3 Function of the API

The API must have the following functionality as a minimum:

- Display a coloured pixel at a given point.
- Display a coloured circle or ellipse at a given location, filled or unfilled.
- Display a coloured regular or irregular polygon at a given location, filled or unfilled.

- Display coloured text at any position.
- Change the colour of the background/canvas.
- Clear the screen of all drawings instantly.

An extra ‘bonus’ feature would be to save the image drawn to file. Graphics were achieved in previous versions using Python’s Turtle library. This was very effective for small simple simulations. However, when displaying flares on stares or when drawing lots of particles the performance drops rapidly. Ideally, an additional API such as OpenGL should be used.

The picture to be drawn by the API will be provided by the camera. Like the simulator, the camera should only have to do the bare minimum work before handing over to the graphics API. The only information provided to the api from the camera will be:

- Type of shape.
- Screen position of shape.
- Shape parameters. (If required)
- Shape fill options. (If required)

Potentially, the brightness or light intensity of the object will also be given.

## Chapter 2

# Simulation Algorithms

### 2.1 Matrix Mechanics

Generally using a static n-dimensional array as a matrix will be faster than using a dynamic n-dimensional list. This is especially the case with Numpy's arrays compared to the native python lists.

To make calculations we require the following arrays to be known at all times:

- $\mathbf{P}_N \in \mathbb{R}^{N \times 3} \rightarrow$  Array of position vectors.
- $\mathbf{R} \in \mathbb{R}^N \rightarrow$  Array of radii of particles.
- $\mathbf{M} \in \mathbb{R}^N \rightarrow$  Array of masses of particles.

Where  $N$  is the number of particles. For a calculation such as standard gravitational attraction, we would perform the following particle for each particle. Here the individual particle we are simulating for is as position  $\mathbf{P}$ , with mass  $M$  and radius  $R$ .

Find the distance to each other particle:

$$\mathbf{D}_N = \mathbf{P}_N - \{\mathbf{P}\}$$

(The braces indicate the subtraction is for each element in the matrix  $\mathbf{P}_N$ )

The absolute distances are given by:

$$\mathbf{D} = |\mathbf{D}_N| = \{|\mathbf{x}| \mid \forall \mathbf{x} \in \mathbf{D}_N\}$$

For the force:

$$\mathbf{F}_N = GM \odot \mathbf{M} \odot (\mathbf{D}_N) \oslash (\mathbf{D}^3)$$

Where  $\odot, \oslash$  are element wise multiply and divide respectively. Also note that cubic power at the right is element wise aswell.

Now we sum the forces to get the net force:

$$\mathbf{F} = \sum_{n=0}^N \mathbf{F}_{N_n}$$

The acceleration is simply derived from  $\mathbf{F} = m\mathbf{a}$ , however it would be fine to simply remove the mass of the particle at the start in this case to directly calculate acceleration instead of force.

We also want to check for collisions. Lets consider a vector  $\mathbf{A}$  containing the ‘altitude’ of each particle, ie the surface-surface distance.

$$\mathbf{A} = \mathbf{D} - \mathbf{R} - \{R\}$$

$$\mathbf{B}_{\text{collision}} = \{\mathbf{A} < 0\}$$

Where  $\mathbf{B}$  is a boolean vector where a value of True indicates that our current particle is colliding with the particle at that index.