Physics for Games

Report

Maciej Legas

Student number: s4922675

Overview

The physics simulation by default has been set to have 4 slopes, which are rotated planes with a bounciness factor determining how much of the ball's velocity remains after bouncing off it, above which there are 5 balls. The camera is moved by user input.

When a user presses the X key, a ball drops on each of the planes and bounces off it, colliding with a second ball in the air and dropping to the ground. As a side event, a fifth ball drops from a higher distance and bounces off the slope, going a far distance with a high angular velocity.

The simulation has been set in this way to showcase a few events:

- an impulse based collision response, both linear and angular, with the rotated planes (bounce from a slope)
- an impulse based collision response, both linear and angular, with the spheres colliding amongst themselves
- rotation due to angular momentum (high velocity drop)

I include a small diagram showing the main operations the program is supposed to do.

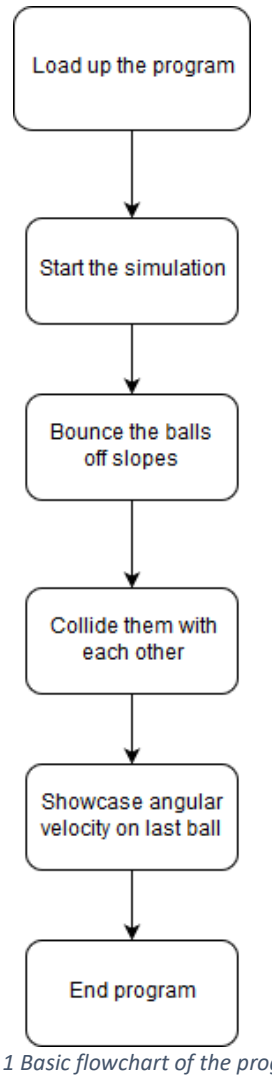

Figure 1 Basic flowchart of the program.

Research

For my research, I based mostly on the material from the Physics for Games course. To implement my project, which has sphere-sphere collisions, sphere-plane collisions, dynamic movement and rotation, I needed:

- a numerical solution for ordinary differential equations
- broad and narrow phase collision detection methods for spheres and planes
- projectile movement

In the end I decided to implement:

- RK4 (Runge-Kutta), RK2 and Euler methods for ODEs
- Manhattan distance based broad phase collision detection
- Vector-based narrow phase collision detection
- Impulse linear and angular movement

I implemented three integration methods mostly to check if my program will work fine with all three of them during debugging:

- The Euler method is one of the easiest, and most error-prone in the long run methods for integration. It works by simply adding the change of velocity happening in the next time step to the current velocity:

  $$v(t_0 + \Delta t) = v_0 + a\Delta t$$

  This, however, only uses the derivative information at the beginning of the time interval, leading to incorrect information in the long run (Dawkins, 2018).

- Runge-Kutta2 is an improvement compared to Euler, as it takes a step at the midpoint of the interval $\Delta t/2$, calculating the value, and then using it for computing the whole value of the interval Δt (Tang, 2018).

- Runge-Kutta4 expands on Runge-Kutta2, evaluating derivatives 4 times, using an initial time step, two midpoints, and an endpoint. It calculates the final values by calculating a weighted sum of the values from these steps (Tang, 2018).

For broad phase collision detection, I decided to use a Manhattan distance based check. Here is an example:

```
distance = abs(x1 - x2) + abs(y1 – y2) + abs(z1 - z2);
```

This function calculates the Manhattan distance (Decoret, 2006) between two positions (preferably the centre of a model), which allows us to create a broad phase collision detection from it, based on whether the distance between two positions is small enough for them to collide. This optimizes the code, as it is faster than checking vector collisions.

As for narrow phase collision detection, I used the sphere to sphere collision method and sphere to plane collision method. The sphere to sphere collision method works by finding a contact point from the radii of two spheres, giving the normal of the collision as well, while

the sphere to plane collision checks if the sphere collides with the plane in the next time step, retrieving the contact position as well (Tang, 2018).

I decided to use impulse collision responses in my program as it allows for accurate dynamic motion calculations, using of course impulses for linear motion, but also using torque, angular motion, angular velocity and inertia to calculate angular motion.

For sphere to sphere collisions, I used these equations:

$$J_{linear} = \frac{-(1+e)(v_i - v_j) \cdot n_c}{\frac{1}{m_i} + \frac{1}{m_j}}$$

$$J_{angular} = -(1+e)(v_i - v_j) \cdot n_c [n_c \cdot \left((r_1 \times n_c) I_{b1_{inverse}} \times r_1\right) + n_c \cdot \left((r_2 \times n_c) I_{b2_{inverse}} \times r_2\right)]$$

And for sphere to plane collisions, I used these (Tang, 2018):

$$J_{linear} = \frac{-(1+e)(v_i) \cdot n_c}{\frac{1}{m_i}}$$

$$J_{angular} = -(1+e)(v_i) \cdot n_c [n_c \cdot \left((r_1 \times n_c) I_{b1_{inverse}} \times r_1\right)]$$

Implementation

I made the program read the initial simulation parameters from a text file while the Scene is being constructed, so that it knows how many spheres and planes are supposed to be in the scene, as well as their positions and other specifications. After the scene is constructed, its update function gets called by the application, where the camera gets updated, and if the simulation has been started, each of the spheres and planes gets updated as well. Afterwards, all of the objects get drawn. Here is the updated flowchart of the simulation:

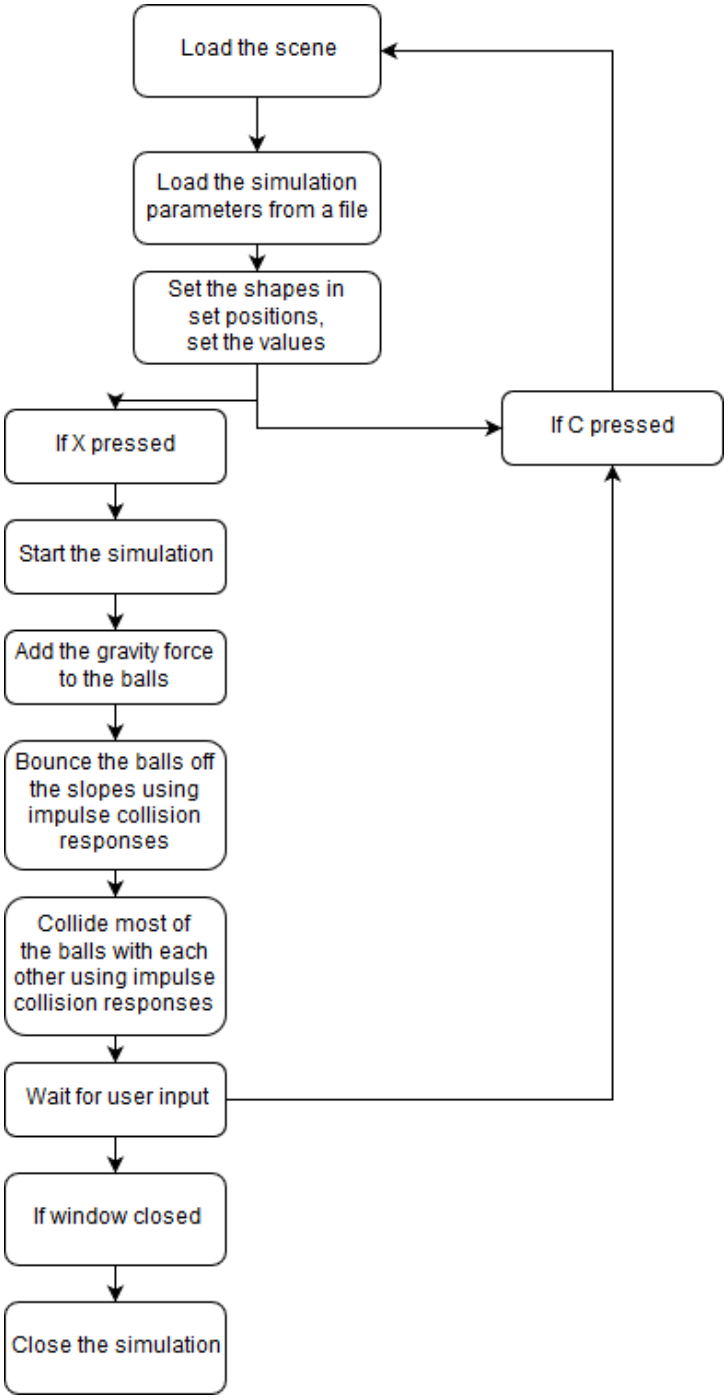

Figure 2 Updated flowchart of the simulation.

As the simulation starts, each of the spheres will be affected by the gravitional force downwards with each update, also doing a broad check for collisions with each update. As one bounces off a slope, it has its velocity rotated by the normal of the slope, causing it it bounce off it and gain a bit of angular momentum. When the balls collide, they affect on each other with their forces, doing an impulse reaction, and fall down to the ground.

I decided to use two classes for my physics engine: DynamicObject and Plane, as well as a namespace for the collision detection functions. Both of these classes inherit from the GameObject class, which contains the basic functions for a game object, such as rotation, position and scale. The Plane class could be a GameObject, but since we need the normal of

the plane for impulse collision responses, I decided to move it into a separate class instead of having each GameObject have an unnecessary variable to store the normal.

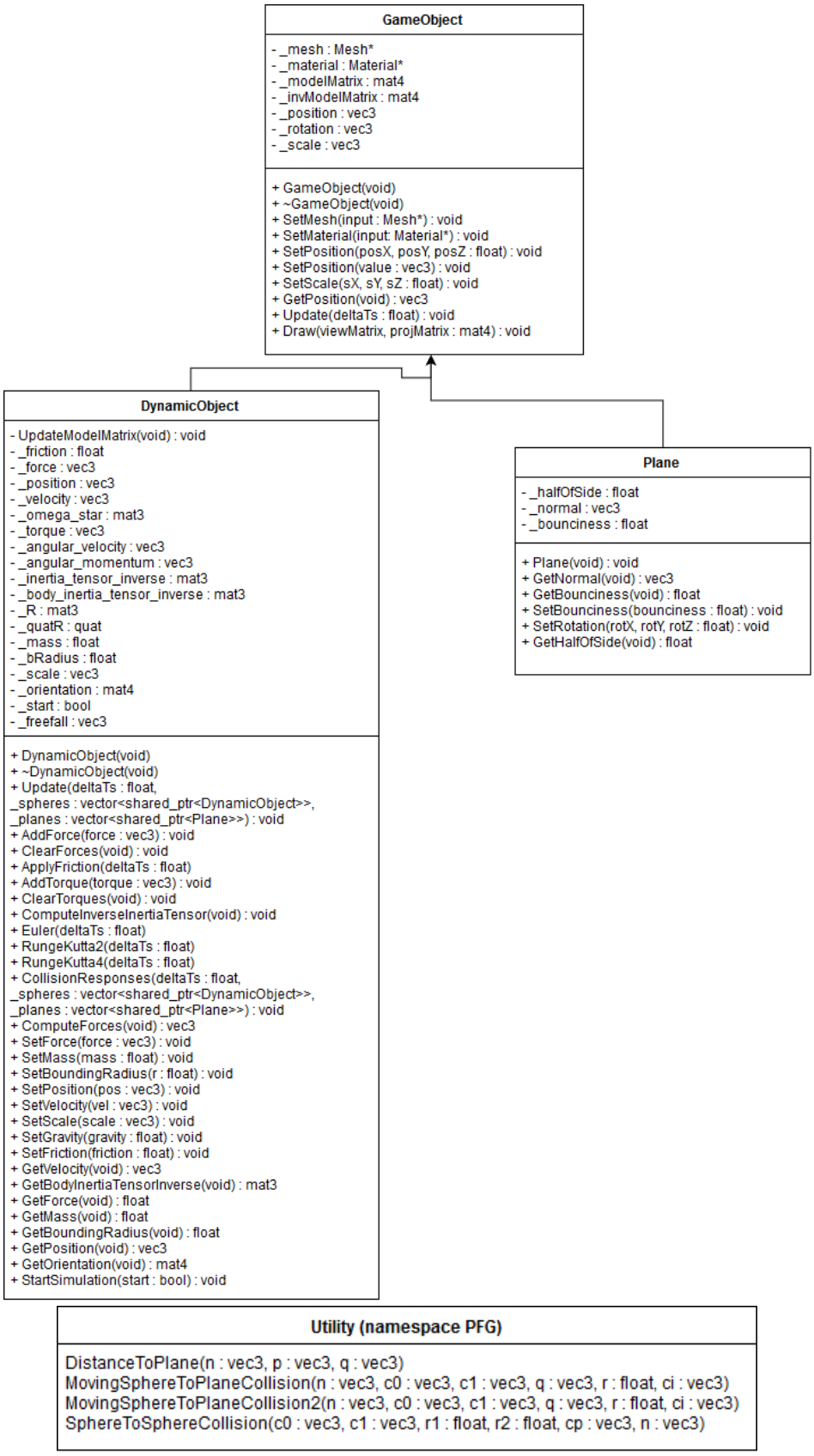

Figure 3 UML flowcharts.

When each of the spheres gets updated, it clears all of the forces and torques by calling named functions, adds the gravity force, performs collision detections by calling the STL vectors of spheres and planes as parameters to CollisionResponses(), and using the Utility functions for narrow phase collision detection. Then, it uses by default the Runge-Kutta4 method to calculate its new velocity and rotation, using the impulse forces given from collisions, if any have happened. It also applies friction if it collides with a plane.

To collide with a plane, the simulation firstly checks in the broad phase collision detection whether the distance between the tested sphere and the plane is equal or less than the half of the side of a plane, as all of them are squares. When it checks that they collide in the narrow phase, it does an impulse collision response involving the normal of the plane for the direction of the impulse force, as well as the bounciness factor of the plane.

To collide with a sphere, the simulation simply checks if the distance between the spheres is lower than two, as the spheres might be moving quite fast, so it would be wiser to set the broad phase collision detection a bit looser on detection. Later it does the narrow phase collision detection, and an impulse collision response between the spheres.

Analysis and conclusion

While debugging the code, I noticed a quite regularly happening problem with the rotation matrix of the spheres and the inertia tensor quite often getting the values of infinity or Not a Number. This has been resolved by limiting them both to a value of 1.0. This allows for a realistic simulation of rotation without the simulation breaking, though it could be improved by readjusting the angular momentum, for example slowing down its growth and increasing the limit value.

The broad phase collision detection for the planes has both an advantage and a disadvantage. As the distance to compare has been set to half of the side of the square, there might be a situation where the physics engine will not register a collision at a corner of a plane. This could be resolved with checking if the X and Z values of the sphere's positions are between the plane's sides, however this would mean that each time a sphere flies over a plane the simulation would perform a narrow phase collision detection on it. An ideal solution would perhaps be implementing spatial partitioning with massively divided grids.

To check the performance, I implemented an FPS counter in the window bar. The only time that the amount of frames per second drops is a few moments at the start of the simulation, as it has to perform a lot of operations at once for each of the object, but once it gets past that, it keeps a stable 60 frames per second count, even when the balls are colliding and changing their velocities, which makes it seem that the broad phase collision detection is working great at optimization.

Rotating the planes over 180 degrees at once causes the normals to face the opposite direction, which breaks the correctness of a sphere bounce from the slope, as it bounces in the opposite direction, going through the plane. This could be fixed be either not allowing the user to rotate over 180 degrees, or doing a check if the plane's normals are supposed to be facing the way they're doing during the collision detection.

As for the strengths of my simulation, the major one is that all of the collisions respond with impulse based responses, allowing for smooth and realistic physics reactions. Using a

quaternion for updating the rotation in the integration methods also improves the stability of the rendering of rotation, so that it strays away from “exploding” the sphere meshes.

References

Tang W. (2018). *Lecture 6 – Numerical Integration – Runge-Kutta Method*. [online] Available at: https://brightspace.bournemouth.ac.uk/d2l/le/content/30848/viewContent/193436/View [Accessed 10 Jan 2018].

Tang W. (2018). *Lecture 8 – Rigid Body Simulation – Implementation Details*. [online] Available at: https://brightspace.bournemouth.ac.uk/d2l/le/content/30848/viewContent/201759/View [Accessed 10 Jan 2018].

Tang W. (2018). *Lab 5 – Implementing Collision Detections and Impulse Responses*. [online] Available at: https://brightspace.bournemouth.ac.uk/d2l/le/content/30848/viewContent/219534/View [Accessed 10 Jan 2018].

Decoret X. (2006). *Manhattan distance of a point and a line*, p. 7. [online] Available at: http://artis.imag.fr/~Xavier.Decoret/resources/maths/manhattan/html/ [Accessed 10 Jan 2018].

Dawkins P. (2018). *Euler’s Method.* [online] Available at: http://tutorial.math.lamar.edu/Classes/DE/EulersMethod.aspx [Accessed 10 Jan 2018].