DH2323 Computer Graphics and Interaction
Rigid Body Simulation Using Particles

Marcus Andersson*
KTH Royal Institute of Technology

Abstract

This article presents the implementation of a rigid body simulation. This was built using C++ and OpenGL. The simulation runs entirely on the CPU, however, the graphics is visualized via GPU rendering. The solution allows for flexibility to change parameters that change how the physics affect the environment. The system comes with a movable camera that enables interaction with the scene in real-time. A set of particles used for collision detection can be created for any closed-shaped OBJ-file such that it fills up a resembling 3D shape. The amount of particles can be adjusted to account for different hardware capabilities.

1 Introduction

In computer games, physics engines are commonly used to let players interact with an environment. The perception of realism is increased when a user can move objects however they want and have them act like in the real world. To simulate an environment one needs two things: A way to detect collisions, and physics equations that decide what happens during and after a collision occur. It is not always desirable to have a perfectly realistic simulation, as the more realism that gets included, the higher the computational cost will be to run the simulation.

There are several variants for creating simulations. The solution presented here make use of the discrete element method (DEM). Each object used is transformed into a set of particles that fills up the inside of the object. When collisions on particles happen, the force of all particles are accumulated for the whole object.

2 Related Work

In this section I will list sources that helped or inspired the outline of this project. Other research of interest, that are helpful in a broader sense are also mentioned here.

All main ideas came from GPU Gems 3, chapter 29, which describes how to make a rigid body simulation [4]. This book gives a proper overview on how to implement it, but does not go into the fine details. One possible error was found in the equations provided (equation 18), although it could be implementation specific.

The book "Essential Mathematics for Games and Interactive Applications" [6] was useful for gaining insight into the topic and optimization ideas. It explains mathematical equations thoroughly that can be used for physics simulations in games.

Yiyu Cai and Sui Lin Goei explains in their book "Simulations, Serious Games and Their Applications" about rigid body simulations [1]. They bring up actual research made about games and statistical data. This provides an empirical perspective on studies about games.

Michele V., Ludovic H., Luigi B. and Carol O. conducted a perceptual study. They show findings about what humans find realistic and non-realistic when a model throws a ball in various ways [7]. The study shows that certain types of movements can appear unnatural. Their methodology could be mimicked in future perceptual studies, as well as in studies involving simulations.

*and8@kth.se

3 Implementation and Results

In this section the details of the project are shown and discussed. Difficulties and how problems were solved is mentioned here. A blog that was made during the development process is available on the internet [2], and can provide further insights.

3.1 Generating Particles

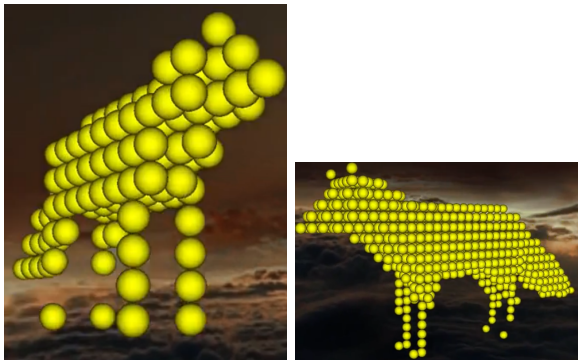
To create a set of particles one needs to have information about an object. In this case we load an OBJ-file and extract the vertex coordinates for an enclosed shape. The shape must be closed and not flat, because rays are used to detect if coordinates are inside or not. In short, triangles are retrieved by taking 3 vertices at a time, and then the triangles are used as an input for a function that outputs a set of particles.

The space containing the object is discretized in a 3D grid. The side length of a location in the grid decides the size of the particles being created. The smallest x, y, and z coordinate for all vertices are found and their absolute values added to avoid using any coordinate with negative values. Rays are then cast into the grid for each x-y grid index, where the rays go parallel with the z-axis. If the amount of intersections are odd, then the ray must be inside the object, and a particle can be created. This is done for all grid locations when the particles are generated. The following psuedocode displays the main part of process:

Algorithm 1 Creating Particles

```
1: triangles ← getTriangles()
2: XY ← get2DArray()
3: procedure RayIntersections
4:     for Triangle t in triangles do
5:         for Ray r in nearbyRays(t) do
6:             if r interect t then
7:                 XY[r.x][r.y].Add(r.intersect(t).z)
```

The code is $O(n^3)$, since all rays in the x and y axis must be considered for intersections with all triangles, but in practice it becomes $O(n)$. Because nearbyRays(t) returns only rays that have x and y coordinates which can possibly result in an intersection, resulting in about one ray per triangle. The XY array of arrays is then sorted such that one can calculate if any coordinate has an odd number of z-intersections before it or not. Figure 1 shows how it can look like when particles are created for a fox shape.



(a) Few particles. (b) Many particles.

Figure 1: The same object, shown with few and many particles.

3.2 Optimizing Collision Detection

One detail that can cause DEM to be ineffective is the fact that collisions must be checked between all particles. If done naively, it's $O(n^2)$. This can be achieved in $O(n)$ instead by using data structures. Here, a virtual uniform grid is created to group particles into voxels. Each voxel occupied by a particle and it's adjacent voxels are checked once. In total, every particle does $3 \times 3 \times 3 = 27$ collision checks in average.

To make collision detection between particles an $O(n)$ process, inspiration from GPU-gems was taken. However, there was one technique which was learned from the internet, but the link to the source was lost. It proposed using a hashmap to store voxels instead of a large matrix. This showed to save a lot of memory, which is an advantage when doing collision detection on the CPU instead of the GPU. Unless the GPU supports that type of data structures.

What is done in the code is that a uniform grid is created by indexing coordinates to voxels. In this implementation a red-black tree, "std::map", is used for indexing. std::map can be very slow if a code project is run in debug mode, which is something to be aware of. A hashmap could also have been used, but it's less reliable when it comes to efficiency. The particles are grouped into voxels, where each voxel can have at most 4 particles. There were some problems with figuring out proper voxel sizes. It turned out to be rather simple. If there is to be 1 particle per voxel, then the voxel length should be the same as the diameter of a particle. Lastly, each particle coordinate is divided by the voxel length, and stored in the red-black tree. Each voxel is iterated once and used to check for collisions within that voxel and it's adjacent voxels. This results in about 27 collision-checks per particle.

3.3 Applying Physics

The physics in the code was divided into two sections. One part of the calculations was done for each whole object, once per object, per iteration. The second part of the calculations was done on each individual particle that an object consisted of. The particle physics calculations was only changed for a particle when a collision happened with that particle.

3.3.1 Object Physics

A unit quaternion is used instead of a matrix to describe the objects rotation. This is because a matrix stores other information that just rotation which eventually accumulates errors. When a matrix must be used in calculations, the quaternion is transformed to a rotation matrix with help of the OpenGL mathematics library, "glm". An inertia tensor matrix is created once by iterating each particle belonging to the object and adding values to a matrix, where the value depends on the particle mass and relative position to the center of mass. The contents of an inertia matrix must vary depending on the objects shape [5], Michel van Biezen explains in his online lectures how to create a tensor for an object of particles. The inertia matrix decides how much an object will rotate, when given a rotational force.

The linear force and torque was calculated using equation 1 and 2. These forces were used along with the inertia matrix and quaternion in a few physics calculations, as described in GPU-gems, in order to retrieve the velocity and momentum for an object. By using a math library with the code, it was mostly "plug and play" to solve advanced math equations. If one wish to slow down the speed that an object moves with, a quick hack is to divide the Newtonian forces by a constant.

3.3.2 Particle Physics

All particles adds torque and a linear force according to the following equations:

$$F_c = \sum_{i \in RigiBbody} (f_{i,s} + f_{i,d} + f_{i,t}) \quad (1)$$

$$T_c = \sum_{i \in RigiBbody} (r \times (f_{i,s} + f_{i,d} + f_{i,t})) \quad (2)$$

Here r is the relative position of a particle to the center of mass. f_s, f_d, f_t are the spring, dampening, and tangential force. F_c the linear force and T_c the torque.

One thing that caused some nuisance was that the tutorial on GPU-gems showed an equation that didn't work. The equation 3 must be like in equation 4 instead for it to work. Otherwise the position for the particles gets rotated too much.

$$r_i = Q_j \times r_i^0 \times Q_j^* \quad (3)$$

$$r_i = Q_j \times r_i^0 \quad (4)$$

Here r^0 the initial relative position and Q the quaternion representing a rotation.

Each particle calculates it's own spring, dampening and tangential force. A constant gravitational force is also applied to every particle. How the tangential force is calculated is well defined. On the other hand, how the dampening and spring force is calculated can vary depending on what the goal is with the simulation. Therefore, these were set to arbitrary values. Altough, physical laws were still followed. The spring force works in accordance to Hooke's law. The spring and dampening force emulate if an object elastically collide into other objects or if it acts like a soft-body [3]. In this solution, the dampening ratio was set equal to 1 (critically damped). Since this results in less computations being made. It may be possible however that a user will want to tweak these forces for objects and then more equations would have to be added in the code.

4 Perceptual Study

A perceptual study was not done. Instead this section will contain ideas for how that could be done with this project. As inspiration the author of this document attended two other real perceptual studies.

The first was conducted by the Masters student Maha El Garf, at KTH. She researched if consensus exist among humans in regards to classifying a fictional characters personality traits. A few video clips were shown of talking and moving characters, which then was led by a few questions. The questions asked if the person seemed angry or nice, introvert or extrovert etc. In that way Maha let the user rate different traits in reference to the videos. Also short comments were allowed to be made. How could one apply this technique into a perceptual study about a physics simulation? The answer is to let users see clips of test-cases constructed with the simulation. Then these people would have to rate if the collision reactions seemed correct or not. Questions could be asked about whether rotations appear to happen like in the real world, if objects are moving too fast or slow, and more.

The second study was made by the Masters student, Linneá, at KTH. She let the test-subjects also

see videos of fictional characters talking. The main difference was that the characters had faces constructed from real humans of varying ethnicity. They also didn't move. The questions in the study asked about ones emotional response with scales from high to low. Physics is not directly related to emotions. Games are to some extent about emotions. If the simulation is used in a game one could ask if the simulation made things more fun or not for example.

5 Future Work

The physics calculations could be done on the GPU instead of the CPU. This would allow for utilizing the massive power that is available in graphics cards. More statistical data could be made by coming up with test cases that use the simulation. One such test could be about how many particles that can be rendered on hardware without experience frame-rate drops.

Extra features such as friction and wind could be added. The project could be extended for creating fluid simulations. How the forces are calculated can be done better. As it is now, there is a problem with gravity creating a bigger total force than forces caused by collisions. This causes objects to sink into one another when they are stacked.

References

[1] *Simulations, Serious Games and Their Applications*. Gaming media and social effects. 1st ed. 2014.. edition, 2014.

[2] Marcus Emmanuel Andersson. Blogger. `https://kthcollisionproject.blogspot.com/`, Aug 2019.

[3] R. Cross. *Physics of Baseball & Softball*. SpringerLink : Bücher. Springer New York, 2011.

[4] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 1 edition, 2007.

[5] Michel van Biezen. Inertia tensor tutorial. `https://www.youtube.com/watch?v=Ch-VTxTIt0E&t`.

[6] James M Van Verth and Lars M Bishop. *Essential mathematics for games and interactive applications*. AK Peters/CRC Press, 2015.

[7] Michele Vicovaro, Ludovic Hoyet, Luigi Burigana, and Carol O'sullivan. Perceptual evaluation of motion editing for realistic throwing animations. *ACM Trans. Appl. Percept.*, 11(2):10:1–10:23, June 2014.