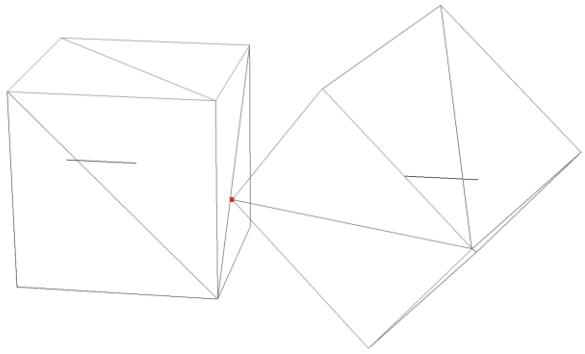
Rigid Body Dynamics Simulator Sandbox



Patrick Phipps, Sriram Narayanan

Abstract

The goal of this project was to create a physics simulation environment to impose realistic models of translations, rotations and collision detection on rigid bodies. The collision detection includes methods for edge, vertex and face to face collisions for convex polygons. Freely moving mesh objects can be given an initial velocity and initial rotation, and if a collision occurs it will be resolved based on these implemented methods. The resolution incorporates both elastic and inelastic collisions. This simulator can be used to model the physics of various objects, and scenes can be created for specific simulation purposes.

Introduction

Modeling physical simulations of rigid bodies has been an ongoing problem in the graphics community, with many methods of implementation discussed over the past few decades. An important aspect of creating realistic simulations is collision detection, and how objects interact with each other. Computation of collision detection algorithms can be expensive if they are not efficient, as it generally requires checking many surface points on two or more distinct objects. The paper "Collision Detection and Response for Computer Animation," written by Matthew Moore and Jane Wilhelms presents algorithms for detecting face to face, vertex and edge collisions, and responding to them by predicting behavior according to physics. This requires giving models specific velocities and rotations that are updated correctly based on the result of the collision.

System Overview

For the simulator, a mesh object can be loaded and is stored within an "actor," which is used describe an object. The actor is given a mass, and the center of mass is calculated by adding all of the vertices in the mesh and dividing by the total number of vertices. This yields a fast solution to requiring a center of mass, but is easily skewed by models created without this in mind.

The physics class is structured with a specific hierarchy to facilitate collision detection and resolution. The class contains the actors that have been created, which in turn contain the bounding boxes. The order of operations performed is as follows. First, a rough bounding box check is performed at each step to gauge whether two actors are close enough to consider a collision check. This reduces the computational load because it is much faster than running collision checks between meshes first. The bounding box check is performed by taking a vector from the center of each box to a corner and getting half the distance. Then the distance is compared to the distance between the two actors.

If this distance is less than both of the vectors for each bounding box, the actors are in close enough proximity for the next step. This involves the collision detection algorithm which checks for face, vertex and edge collisions. If a collision is detected, the collision resolution algorithm is initiated. The velocity and angular rotation are integrated to yield the new velocity and angular rotation for each actor. Each step will now be covered in more detail.

Algorithm Overview

```
Bounding box check distance
  if (true):
    Face collision detection
      if (hit):
        collision resolution
    Vertex collision detection
      if (hit):
        collision resolution
    Edge collision detection
      if (hit):
        collision resolution
```

Collision Detection

The full collision detection algorithm is broken into three main parts and produces a collision point and collision plane or Boolean value for false if there is no

collision. The algorithm utilizes these methods in the following order: Face collisions, vertex collisions and edge collisions.

The face to face collision works on actors “A” and “B”. In the implementation the faces of A are looped over and divided into triangles. The vertices of the triangle are averaged, which yields a point in the middle of the face. These generated points are then fed into the point in mesh function. This function takes a point and a mesh and checks to see if the point is in the mesh. This is done by taking the dot product of the point with every face normal in the mesh. If all dot products yielded are negative, the point is in the mesh. If any dot product is not negative the point is not in the mesh. If any of these generated centroids are found to be in the mesh then a face to face collision is detected.

The above method yields very geometry dependent results, such as if a flat face is made up of several triangles. To mitigate this, all points found to be in the mesh are averaged together to yield a better collision point. The collision plane generated by a face to face collision is the normal of any of the collision triangles of the face. By convention this is the normal pointing from B to A. This algorithm is O (number of triangles of A * number of vertices of B).

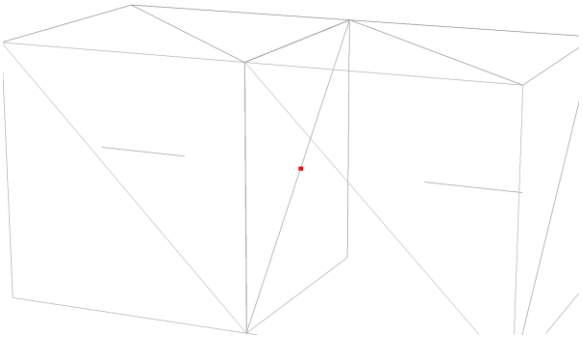


Figure 1: Face to face collision

The second collision test is to check if any of the vertices of A are in B. This is a simple task since the function point in mesh is required for face to face collisions. When point in mesh is run on vertex A and mesh B, the first point to be found in the mesh is returned as the collision point. The collision plane is defined by taking the closest face of B to the collision point and using its normal. This algorithm is O (number of vertices in A * number of faces in B).

The last collision test is the edge test. This test loops over every edge in A and every triangle in B. The first step of this algorithm is to calculate the perpendicular distance from the ends of the edge from the triangle's plane.

$$d_i = (v_i - u_{k1}) \cdot n$$

$$d_j = (v_j - u_{k1}) \cdot n$$

This test is used to check if there is a chance that this edge has intersected this face. If the signs of the distances differ then the test can proceed because the edge intersects the triangle. This point can be calculated via the following equation:

$$t = \frac{|d_i|}{|d_i| + |d_j|}$$

$$P = v_i + t(v_j - v_i)$$

After completing this process with every face for the given edge a collection of points is generated along the edge. This collection of points, t, is then sorted. Every pair of these points, including the pair consisting of the start and end vertex, are averaged to generate a point on the line. This point is then run through the point in mesh function, and the first point to be found in the mesh is returned as the collision point. The collision plane for this collision is given by the cross product of the edge with the plane it is intersecting. This algorithm is O (edges of A * faces of B).

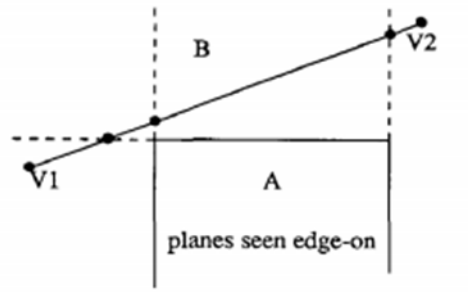


Figure 2: Edge subdivision

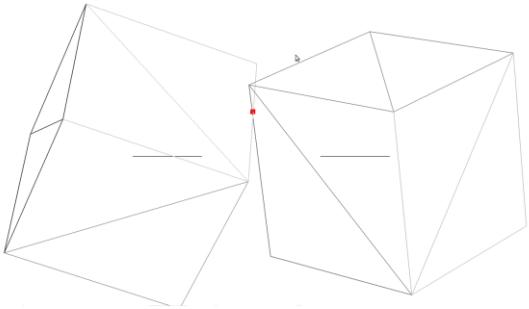


Figure 3: Edge collision

Translations

For modeling the movement and rotation of objects a simple Euler integrator is used. Actors can be given an initial velocity vector, which will translate the geometry according to the velocity. The actor can also be given an initial rotation vector which is given in degrees. Using the rotation vector, a rotation matrix is created using the following principles outlined in the Matrix class:

$$R_x(\theta)\begin{bmatrix}1 & 0 & 0\\ 0 & cos(\theta) & -sin(\theta)\\ 0 & sin(\theta) & cos(\theta)\end{bmatrix}$$

$$R_y(\theta)\begin{bmatrix}cos(\theta) & 0 & sin(\theta)\\ 0 & 1 & 0\\ -sin(\theta) & 0 & cos(\theta)\end{bmatrix}$$

$$R_z(\theta)\begin{bmatrix}cos(\theta) & -sin(\theta) & 0\\ sin(\theta) & cos(\theta) & 0\\ 0 & 0 & 1\end{bmatrix}$$

The center of mass is rotated on each axis by the angular velocity which gives the new rotation of the object. The center of mass is then shifted by adding the velocity to itself.

Collision Resolution

Collision resolution for velocity and rotation, which is the primary focus of this simulation since friction is omitted, is straightforward with the right parameters. The parameters are as follows:

v_a = velocity of A
v_b = velocity of B
$n =$ the normal to the plane of collision
r_a = a vector from the center of mass of A to the collision point
r_b = a vector from the center of mass of B to the collision point
ω_a = angular velocity of A
ω_b = angular velocity of B
m_a, m_b = the mass of A and B
I_a, I_b = inverse inertial tensor of A and B
j = impulse of the collison

The calculation of the inertial tensor is given by:

$$I = \begin{bmatrix}I_{11} & I_{12} & I_{13}\\ I_{21} & I_{22} & I_{23}\\ I_{31} & I_{32} & I_{33}\end{bmatrix}$$

$$I_{11} = I_{xx} \overset{\text{def}}{=} \sum_{k=1}^{N} m_k(y_k^2 + z_k^2)$$

$$I_{22} = I_{yy} \overset{\text{def}}{=} \sum_{k=1}^{N} m_k(x_k^2 + z_k^2)$$

$$I_{33} = I_{zz} \overset{\text{def}}{=} \sum_{k=1}^{N} m_k(x_k^2 + y_k^2)$$

$$I_{12} = I_{xy} \overset{\text{def}}{=} -\sum_{k=1}^{N} m_k x_k y_k$$

$$I_{13} = I_{xz} \overset{\text{def}}{=} -\sum_{k=1}^{N} m_k x_k z_k$$

$$I_{23} = I_{yz} \overset{\text{def}}{=} -\sum_{k=1}^{N} m_k y_k z_k$$

$$I_{12} = I_{21} \quad I_{13} = I_{31} \quad I_{23} = I_{32}$$

The new velocity of A and B as well as the new angular velocity of A and B are as follows:

$$\overrightarrow{v_{af}} = V_{ai} - \frac{J}{m_a}n$$

$$\overrightarrow{v_{bf}} = V_{bi} - \frac{J}{m_b}n$$

$$\omega_{af} = \omega_{ai} - [I_a]^{-1}(J \times r_a)$$

$$\omega_{bf} = \omega_{bi} - [I_b]^{-1}(J \times r_b)$$

Using these values the impulse j, generated from a collision, can calculated as follows:

$$\frac{-(1+e)(v_a - v_b)\cdot n + (r_a \times n)\cdot\omega_a - (r_b \times n)\cdot\omega_b}{\frac{1}{m_a} + \frac{1}{m_b} + (r_a \times n)\cdot\left([I_a]^{-1}(r_a \times n)\right) + (r_b \times n)\cdot\left([I_b]^{-1}(r_b \times n)\right)}$$

Tests Run

Testing the simulations was made easier with the addition of a scene loader, which reads in text files of objects with set positions, velocities and rotations. This expedited running tests for face, edge and vertex collisions by setting predetermined paths for objects to collide. It also allows setting up multiple collisions between several actors quickly.

For the face to face collision, two boxes were set apart and given velocities so they would collide directly on one side. This allowed for the testing of elastic collisions. For the edge test, a box was rotated and collided with another box by giving it a velocity down. This test enabled the correction of an issue where face

to face collisions caused cubes to rotate due to collision vectors being slightly offset.

The vertex collision involved launching a box corner at the face of another box and analyzing the resolution. This allowed testing elastic collisions as well as rotation during resolution. Tests were modified by giving actors different weights and sizes to see if the response was modeled accurately.

Results

Following the description from [3] correct collision detection between faces, edges and vertices on faces was able to be implemented. Following the equations from [2] and [4] the resolutions of these collisions are modeled with relative accuracy. The bounding box check lowers the computational cost by only performing collision detection if two actors are relatively close together. With these systems in place multiple actors are able to collide. Collisions between two objects with high polygon counts cannot be simulated.

Future Work

While this simulation adds a working collision detection and resolution system, there are many areas which can be improved upon. A rudimentary bounding box check was implemented because the boxes are not given correctly triangulated geometry. A more robust check would feature collision detection for properly computed geometry of the bounding box.

Another flaw in the system is that it is very obvious when the bounding box check passes and collision detection ensues. This effect is even more obvious when there are many (or about to be many) collisions happening simultaneously. This hints at an adaptive time stepper, or a more advanced method of integration from the simple Euler integration method used. The center of mass calculation is, as mentioned before, less than stellar. It does not have proper mass summation since all of the mass is on the vertexes this causes the center of mass to be skewed to where the highest number of vertices is.

The current implementation of collision detection is very inefficient. While the methods that are described in this paper suffice the implementation is not true to the described algorithm. The collision detection algorithm should furnish a point and a plane upon collision. The current implementation only produces the collision point. Instead, another round of processing must occur to determine the collision plane. The impulse based collision response is also an area of improvement. Impulse based response can handle elastic or perfectly elastic collisions, but once the objects are moving they will not come to rest.

Another piece that would be required in a feature complete physics simulator is resting contact between objects, which would require the implementation of friction. The addition of constraints would allow an object to rotate freely while not translating. An improved simulation interface would allow quick loading of objects without the use of the command line.

References

[1] Hecker, Chris. "Physics Part 3: Collision Response." *Game Development Magazine* Mar 1997: 11-18. Web. 4 May 2011. <http://chrishecker.com/images/e/e7/Gdmphys3.pdf>

[2] Hecker, Chris. "Physics Part 4: The Third Dimension." *Game Development Magazine* June 1997: 15-26. Web. 4 May 2011. <http://chrishecker.com/images/b/bb/Gdmphys4.pdf>.

[3] Moore, Matthew, and Wilhelms Jane. "Collision Detection and Response for Computer Animation." *SIGGRAPH '88*. 22.4 (1988): 9. Print. <http://graphics.stanford.edu/courses/cs448-01-spring/papers/moore.pdf>

[4] Baker, Martin. "Euclidean Space." *Euclidean Space*. N.p., n.d. Web. 4 May 2011. <http://www.euclideanspace.com/physics/>.