Rigid Body Dynamics

Pongpat Sombatpanich

August, 2014

Master of Science,
Computer Animation and Visual Effects

Contents

Chapter 1

Introduction

The study of rigid body dynamics has a long history in the field of computer graphics. Its application can be seen extensively in games and computer animation these days as the concept itself is an important factor to create a believable virtual world. The simulation of rigid body dynamics, which can be refered as a physics engine, will calculate the approximated movements of objects in the system, including collision detection and gravity. This creates an illusion that objects in the virtual world are also subjected to the law of physics, which greatly enchances user experience.

As the name implied, rigid body system focuses on non-deformable objects. By representing objects as rigid, we can greatly reduce the amount of calculation we have to deal with, which is important in real-time application.

As mentioned, a physics engine is a fundamental system in many application in computer graphics field. By developing one, the developer can learn a great deal from it, which can further extend to the other areas of study.

Chapter 2

Technical Background

A physics engine can be divided into four subsystems: Unconstrained motion, Collision detection, Collision response and lastly, constraint resolution which is not in the scope of this project.

2.1 Unconstrained motion

The motion of rigid body can be explained with Newton's second law of motion, which can be extended to explain the relation between position x, velocity v and acceleration a

$$\dot{x} = v(t)$$

$$\dot{v} = a(t)$$

Newtonion Dynamics also give us

$$P = mv$$

$$L = I\omega$$

which represents linear momentum P where m refer to mass of rigid body. Similarly, angular momentum L, is the product of angular velocity ω and inertia tensor I which will be described below.

For simplicity, we can just think of inertia tensor as an angular version of mass. However, inertia tensor is subjected to the body rotation, so we need to recalculate it every time the rotation changes from the original local inertia by

$$I = RI_{body}R^T$$

where R represent a rotational matrix of rigid body. R^T is its transpose.

As we will use ordinary differential equation to describe unconstrained motion, it is easier if represent the motion of rigid body as a state vector.

$$X(t) = \begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix}$$

Together with the above equations, we can explained the derivative of $X(t)$ as

$$\dot{X}(t) = \begin{bmatrix} \dot{x}(t) \\ \dot{R}(t) \\ \dot{P}(t) \\ \dot{L}(t) \end{bmatrix} = \begin{bmatrix} \frac{P}{m} \\ skew(I^{-1}L)R \\ \dot{P}(t) \\ \dot{L}(t) \end{bmatrix}$$

However, it is recommended that the rotation of rigid body can also be represented as quaternion, which is proved to have less numerical error in real implementation.

2.2 Collision Detection

This topic covers the techniques that have been used to sort and query a set of rigid bodies, as well as pinpoint the colliding location. Collision detection generally can be divided into phases such as broadphase and narrowphase. Broadphase involves quickly determining whether the two bodies have potential to collide, so that to avoid having to expensively compare every objects in the scene. The technique includes using some types of spatial structure such as grid and dynamic tree to organize rigid bodies, as well as using cheap and less accurate representation of the actual object such as bounding sphere, or axis aligned bounding box.

On the other hand, narrowphase is where we process the actual detection. Depending on the type of collision shapes, the simple primitive shapes, which are easier to implement, need to be implemented with special-case detection algorithms per each shape combination e.g. sphere-sphere, sphere-box, and box-box. There are also algorithms that are independent from shape type like the Gilbert-Johnson-Keerthi or GJK which handles convex shapes by using set operations between collided pair. Regardless of methods chosen, the algorithms suppose to accurately determine collisions and return collision details for response solver.

2.3 Collision Response

After detect that objects collide, A physics engine must decide on how to resolve the penetration. One approach is described as to force the nonpenetration constraint to the objects. The concept is that in order to prevent objects from further interpenetration the solver will generate impulsive forces along the contact normal, which will abruptly change the objects velocity. In order to do this, we need contact these information from collision detection system.

- Contact point
- Contact normal
- Penetration depth
- Contact type : vertex-face or edge-edge

The need of these information can be varied depending on implementation, but contact point and normal are essential in any case. The following section will describe different method in details.

2.3.1 Impulse Method

The impulse method resolves the penetration by manipulating the velocities of objects. Impulse j is a term in physics defined as the accumulated forces F applied to a body over a period of time Δt.

$$j = F\Delta t$$

Playing further with Newton's second law, we get

$$\Delta v = \frac{j}{m}$$

Assuming the system is frictionless, the direction of impulse will be in the normal direction, thus we can define the post-collision velocity v' as

$$v'_A = v_A + \frac{j\hat{n}}{m_A}$$

$$v'_B = v_B + \frac{j\hat{n}}{m_B}$$

Similarly, the post-collision angular velocity ω' is

$$\omega'_A = \omega_A + jI^{-1}(r_A \times \hat{n})$$

$$\omega'_B = \omega_B + jI^{-1}(r_B \times \hat{n})$$

Where r is distance from object center to collision point, then we can find velocity at collision point by

$$V = v + \omega \times r$$

Substitute post-collsion velocity equation for rigid body A, we get

$$V'_A = V_A + j(\frac{\hat{n}}{m_A} + I^{-1}(r_A \times \hat{n}) \times r_A)$$

We apply the same to rigid body B, then subtract both equaion, we get

$$V'_A - V'_B = (V_A - V_B) + j(\frac{\hat{n}}{m_A} + \frac{\hat{n}}{m_B} + I^{-1}(r_A \times \hat{n}) \times r_A + I^{-1}(r_B \times \hat{n}) \times r_B)$$

at the last step, we introduce restitution coefficient to scale down the impulse

$$\hat{n} \cdot (V'_A - V'_B) = \epsilon \hat{n} \cdot (V_A - V_B)$$

$$V_{rel} = V_A - V_B$$

Substitute the above equation, finally, we obtain

$$j = \frac{-(1+\epsilon)V_{rel} \cdot \hat{n}}{\frac{1}{m_A} + \frac{1}{m_B} + (r_A \times \hat{n}) \cdot (I^{-1}(r_A \times \hat{n})) + (r_B \times \hat{n}) \cdot (I^{-1}(r_B \times \hat{n}))}$$

After we achieve the impulse magnitude, we will apply it via rigid body momentum

2.3.1.1 Impulse Based Friction

The impulse reaction will only apply along collision normal. we also need reaction force along the surface. *Coulomb Friction model* is one of the most popular model. The model separates friction coefficients into two: static friction μ_s and dynamic friction μ_d. Imagine pushing a heavy object along the surface, at first the object will resist until we exert force pass certain limit. Suddenly, the body starts to slide easily. This is when it changes coefficient from static to the lighter one, dynamic.

Friction impulse j_f will be applied along a tangent vector $\hat{t}$, which is perpendicular to normal vector $\hat{n}$ can be found by

$$\hat{t} = \begin{cases} \frac{v_r - (v_r \cdot \hat{n})\hat{n}}{|v_r - (v_r \cdot \hat{n})\hat{n}_r|} & v \cdot \hat{n} \neq 0 \\ \frac{f_e - (f_e \cdot \hat{n})\hat{n}}{|f_e - (f_e \cdot \hat{n})\hat{n}_r|} & v \cdot \hat{n} = 0 \; f_e \cdot \hat{n} \neq 0 \\ 0 & v \cdot \hat{n} = 0 \; f_e . \hat{n} = 0 \end{cases}$$

where f_e is external forces affect on the body. Frictional force can be computed f_f as

$$f_f = \begin{cases} -(f_f \cdot \hat{t})\hat{t} & v_r = 0 \; f_e \cdot \hat{t} \leq f_s \\ -f_s \hat{t} & v_r = 0 \; f_e \cdot \hat{t} > f_s \\ -f_d \hat{t} & v_r \neq 0 \end{cases}$$

According to Coulomb Friction cone, the relationship between impulse response and impulse friction is

$$j_s = \mu_s j_r$$

$$j_d = \mu_d j_r$$

Finally, the integration of frictional force yields

$$j_f = \begin{cases} -(mv_r \cdot \hat{t})\hat{t} & v_r = 0 \; mv_r \cdot \hat{t} \leq j_s \\ -j_s \hat{t} & v_r = 0 \; mv_r \cdot \hat{t} > j_s \\ -j_d \hat{t} & v_r \neq 0 \end{cases}$$

Chapter 3

Solution

The project implements a physics engine with special-case collision detection algorithms. For collision response Impulse method has been chosen as the algorithm of choice.

3.1 Design

The present physics engine was implemented using C++ language and incoperated NGL library, which is a graphic programming support library for students here in NCCA. The application design employs *PhysicWorld* as a main class that wraps around all *RigidBodys* in the system and forward the simulation in time. During each step, *CollisionEngine* and *ImpulseSolver* are the essential classes which handle interaction between objects. The first is responsible for collision detection, while the latter calculate collision responses on collided pairs based on the engine output. GLWindow class is responsible for all rendering works, as well as connecting UI to the lower level application.

RigidBody the class represents an individual rigid body properties including its motion-related variables. It also contains pointer to *CollisionShape* class which represent its bounding shape. The class is inherited from an abstract class *RK4Integrator* which provides ode solving ability.

CollisionShape is inherited by *SphereShape*, *PlaneShape* and *BoxShape*. These classes represent the rigid body bounding shapes which are used in collision detection and rendering. Each contains methods and properties related to its shape for example *BoxShape* knows its extent size and can calculate its vertices position in relative to worldspace.

CollisionEngine is responsible for all collision detection activities both broadphase and narrowphase. It contains several case-specific collision detection methods such as sphereVSsphere, boxVSplane. Each special-case detection method will determine whether the pair collide, if collide it will generate manifold and contact information. The main method is doCollisionDetection() which will decide the appropriate algorithm for each rigid body pair. The method will return manifolds list which will be passed to *ImpulseSolver* later.

ImpulseSolver is responsible for collision response. It employs Impulse method for the resolution. The solve() method takes a list of manifolds as an input and solve each manifold sequentially. After all collisions have been resolved, it will call update() on each rigid body.

3.2 Problem

Unfortunately, the project currently has major bug issues which are listed below.

3.2.1 Major Issue

Currently, the collision detection system break. The application in current state cannot handle any other collision except against the ground plane. This happens after introducing friction to the system. Unfornately, there is simply not enough time to fix.

3.2.2 Minor Issue

While the application works fine under the condition of only one object per scene. Notice that when trying to balance boxshape on one of its corner. The box will sway then topple down, then swing back again. This could happen from excessive angular momentum. By introducing some damping forces to the system, the problem should be fixed.

Chapter 4

Conclusion

Due to confusion and misdirection during the initial of this project, this project has less time to develop to its full potential. Most key features have already been implemented. However, the project has serious bug issue which unfortunately cannot be fix within the deadline.

4.1 Future Work

There are several features which can enchance the current simulation as following

- Implementation of spatial structure such as dynamic aabb tree which greatly improve the efficiency of the system as it allows the simulation to handle more objects
- Develop island processing system which will process interacting rigid bodies together as a group, which is believed will improve the stability especially when bodies stacking.
- Develop a better collision detection algorithm such as GJK
- Implementation of constraint system which enable rigid bodies to interact with each other in interesting ways

Bibliography

[1] Eberly, H, E., 2010. Game physics. Second Edition. New York: Elsevier Science Inc.

[2] Begen, V,D,G., 2004. Collision detection in interactive 3d environments. New York: Elsevier Science Inc.

[3] Vella, C., 2008. Gravitas: An extensible physics engine framework using object- oriented and design pattern-driven software architecture principles. Thesis (Master). University of Malta

[4] Baraff, D., 2001. Physically based modelling: Rigid body simulation. Siggraph 2001 course notes.

[5] Muller, M., 2008. Real time physics. Siggraph 2008 course notes.