Explosion Simulator
CS 523: Computer Graphics

Harsh Darji(hid3)	Pawan Alur(pka21)

Introduction

The project we have for CS523 is to simulate a brick being thrown at a wall. This is an application of rigid body dynamics, where we simulate a rigid body and compute it's position and rotation each step. We take into account collision between all bricks and apply Colliding Contact or Resting Contact where needed.
We assume in this problem that each block is a rigid body, and there is no compression. Thus, all collisions happen instantaneously.

Challenges

The challenges faced during this project were :

- Finding a QP solver. As QP solvers are not easy to find, and the ones that we found usually solved a specific QP model rather than what we wanted, we had to write our own solver.
- Lack of a general NMatrix and NVector class. As C# does not have any stl dealing with such classes, we had to code our own. While most of the operations were easy, some operations took a lot of effort(Inverse, Multiply etc.)
- Using both code and user input at the same time. As the code moved the block to it's computed position every frame, getting user input was impossible. This prevented us from making our original plan(Jenga), and we just decided to make a 'throw a brick at the wall' simulation instead.

Implementation:

The implementation has 3 major parts :

1. Rigid Body Physics	2. Collision Detection	3. Collision Response

Rigid body Physics

We simulated a rigid body by considering the following elements to be properties of the block :

- Constants : These values would be initialized at the beginning and would never change
  - Mass : Mass of the body.
  - Mass Inverse : Set as 0 for unmoving objects
  - IBody : The inertia of the body defined in body space. This allows us to calculate the inertia tensor using just the IBody and Rotation.
  - IBodyInverse : It's function is to simplify calculations and is set to 0 for constant objects to avoid rotation
- State Variables : These variables are used to represent the current 'state' of the object completely

    - Position : The current position of the center of the body in world space.
    - Rotation : The current rotation of the body in world space. Stored as a quaternion
    - Linear Momentum : This is just Mass * Velocity. Storing this rather than velocity allows us to directly interact with force.
    - Angular Momentum : This is Inertia * Angular Velocity. It remains constant when no torque is applied and is independent of the current Rotation of the body.
- Derived Variables : These variables are not stored as part of the body, but are derived every frame.
    - Inertia : $I(t) = R(t)*Ibody*R(t)^T$. This is the inertia for the body that depends on the rotation at the current frame.
    - Velocity : This is calculated as Linear Momentum / Mass
    - Angular Velocity : This is calculated as follows :
        - Calculate Inertia at time T
        - Calculate Inverse of Inertia
        - Angular Velocity = InertiaInverse*Angular Momentum
- Other Variables : These are variables that are given by the user/calculated when external forces act on the body :
    - Force
    - Torque

By creating 4 simple ODE's(solved using Forward Euler method) we can simulate a rigid body :

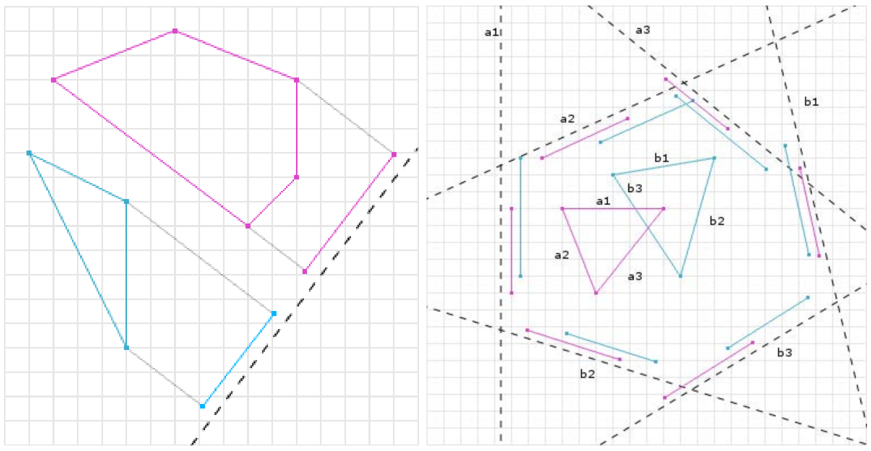
- Position = Derivative of Velocity
- Linear Momentum = Derivative of Force
- Rotation = Derivative of (Rotation*Angular Velocity)
- Angular Momentum = Derivative of Torque

Collision Detection

For collision detection, we use the Separating Axis Theorem(SAT), introduced in [1]. The theorem is a method to determine if two convex shapes are intersecting. It states that,

"If two convex objects are not penetrating, there exists an axis for which the projection of the objects will not overlap."

So, if two convex rigid bodies A and B are not colliding, then there exists at least one hyperplane in world space on which the projections of A and B do not overlap. For the opposite case where there is a collision, there is no hyperplane where the projections of A and B do not overlap. The below images show both the cases.



The problem is that testing all possible hyper planes in the world space for intersections is infeasible. So which hyperplanes to test for? For a 2D case (in the above figure), they are the edge normals of both the objects. For a 3D case, the possible Separating Axes increase to 15, 3 face normals for rigid body A, 3 face normals for rigid body B, and 9 more axes obtained by cross products of face normals of rigid bodies A and B. In the script Controller.cs, for each pair of rigid bodies, below things happen:

- The method AddCollision() calculates the 15 possible separating axes between the two rigid bodies.
- The method IntersectsWhenProjected() determines if the projection of the rigid bodies intersect on the given axis.
- If there is a collision, an object of the struct *Collision* is added to a list.

The struct *Collision* has the two rigid bodies obj1 and obj2 ,the collision location, normal direction, edge direction of obj1, edge direction of obj2, and a boolean to determine the type of collision. At the end of collision detection, the list *collisions* contains all the collision between all pairs of rigid bodies.

Collision Response

Once all the collisions are detected the method CollisionType() determines if the collision should be handles with resting contact or colliding contact and separates them into two lists, *restingContact* and *collidingContact*.

Resting Contact

When two bodies do NOT travel towards each other(say one resting on the other). In this scenario, they exert a force on each other that keeps them in rest, but does not move either. This is relatively harder to calculate, because we can have many objects balanced on each other. Thus,

every frame, we need to calculate all forces involved in all resting contact simultaneously and apply them at the same time.
This is done by assuming a force f_{ij} for every pair of objects that are in resting contact. We then have 3 conditions that all forces have to satisfy :

1. The two bodies should NEVER have inter-penetration. This means that all points of body A should be outside body B and vice versa. As at current time, they are in contact with each other, their acceleration should thus be away from each other(>= 0)
2. Our force should be repulsive, and push each object away from the other(force >= 0)
3. If bodies move away from each other(no longer in contact), force must become 0. As the two bodies are currently in contact, this condition is Force * Acceleration = 0(when Force = 0, the bodies have accelerated away from each other. When Acceleration = 0, they are in contact and force is applied).

Along with these three conditions, we can represent Acceleration of body i(a_i) as :

$$a_i = A_{i1}*f_1 + A_{i2}*f_2 + \dots A_{in}*f_n + b_i$$

where (f_1,f_2...f_n) are all forces in resting contact and b_i is a constant.
Solving all the equations for every object in the simulation that is in resting contact, while following the above 3 conditions gives us the value of all forces that we have to apply.
To solve this system of equations, we need a quadratic program solver. Writing a general QP solver is a really hard, but for this specific scenario(no friction), it is doable.

- Resting Contact takes place in the RestingContact() of controller.cs.
- The functions ComputeAMatrix and ComputeBVector calculate their respective matrix and vector.
- QP_Solve() solves the QP given an NMatrix and NVector class object respectively.
- The various helper functions used for all these include MaxStep(), ComputeAij(), VelocityOfPointOnBody(), ComputeNDot(),FDirection() and ComputeCoefficients().

Implementing a QP solver specific to this problem was using the method in [4]. It involves the following steps :

- Set all forces to 0, and focus on them one at a time. Set i = 1.
- Solve for f_i and a_i that satisfies all 3 rules.
- Set i = 2, and repeat the above step.
- As changing f_2 might affect a_1, we must then again modify f_1.
- Thus, we repeat the above steps for all i = 1,2,3..n.
- In general, for a certain step i = k, the 3 conditions are satisfied for all points < k, and f_i = 0 for all i > k, satisfying 2 of the 3 conditions.
- Thus, for every frame, we compute a deltaF(change in force), and a maxstep(how much force can we apply before one of the already solved accelerations violates the first condition and becomes negative). We then apply this change in force.

Colliding Contact

A collision needs to be solved for colliding contact when the relative velocity between the rigid bodies is less than a determined threshold -*v*. All the collision objects in the list *collidingContact* are such collisions. For each collision in the list *collidingcontact*, an impulse j is calculated to be applied to the rigid bodies. This j is calculated based on the below formula,

$$j = \frac{-(1+\epsilon)v_{rel}^{-}}{\frac{1}{M_a} + \frac{1}{M_b} + \hat{n}(t_0) \cdot \left(I_a^{-1}(t_0)\left(r_a \times \hat{n}(t_0)\right)\right) \times r_a + \hat{n}(t_0) \cdot \left(I_b^{-1}(t_0)\left(r_b \times \hat{n}(t_0)\right)\right) \times r_b}.$$

For the above formula the quantities used are,

- $\in$: restitution coefficient
- V_{rel} : Relative velocity between the two rigid bodies at point of contact.
- M_a : Mass of rigid body A.
- M_b : Mass of rigid body B.
- n : Normal vector for the collision.
- I_a^{-1} : Inertia inverse of rigid body A.
- I_b^{-1} : Inertia inverse of rigid body B.
- r_a : Vector from position of rigid body A to point of contact.
- r_b : Vector from position of rigid body B to point of contact.

After calculating j, force is calculated as *j*n*. This force is added to the linear momentum of rigid body A and subtracted from the linear momentum of rigid body B. The torque for rigid body A is calculated as the cross product of r_a and the calculated force. This torque is added to the angular momentum of rigid body A. Similarly, the torque for rigid body B is calculated as the cross product of r_b and the calculated force. This torque is subtracted from the angular momentum of rigid body B.

Individual responsibilities

Harsh : Collision detection and Colliding contact.

Pawan : Rigid body physics and Resting contact.

Instruction for the code

1. Open Unity Editor and select Open.
2. Navigate to project folder.
3. Click on Play to see results.
4. To access code, go to Assets → Scripts Folder.
5. The zip file contains videos from multiple angles.

References

[1] Boyd, Stephen P, Vandenberghe, Lieven (2004). *Convex Optimization* (pdf). Exercise 2.22. Cambridge University Press. ISBN 978-0-521-83378-3.
[2] Baraff, David. (2009). An introduction to physically based modeling: Rigid body simulation I : unconstrained rigid body dynamics.
[3] Baraff, David. (2009). An introduction to physically based modeling: Rigid body simulation II : Non-penetration Constraints
[4] D. Baraff. Fast contact force computation for nonpenetrating rigid bodies. Computer Graphics (Proc. SIGGRAPH), 28:23–34, 1994