Physics Tutorial 6: Collision Response – Impulse Methods

Summary

Collision response, the final step of our physics engine, is introduced and an impulse based solution is implemented.

New Concepts

Collision response, Penalty method, Impulse method, Projection method, Impulses.

Introduction

Our physics engine is almost complete. So far we can move objects around the environment in a believable manner, and we can detect when two objects intersect. In this tutorial we discuss how to proceed when we have identified that two objects have collided in order to move them apart. As you might expect, the solution is to give each intersecting object a nudge in the direction away from the collision.

In the previous tutorials we have discussed how to identify when an intersection has occurred, and how to calculate the data required to resolve an intersection. The data which we have calculated consists of:

- The contact point where the intersection was detected - this is typically inside one or more of the objects.
- The contact normal - i.e. the direction vector along which the intersecting object must move to resolve the collision.
- The penetration depth - i.e. the distance along the contact normal that the intersecting object must move so that it is no longer intersecting.

Remember that, at this point, the physics engine is still dealing with all the simulated objects - the new physical state of the simulated objects is not made available to the render loop until after the physics update is complete, so the player will not see any of the intersections between objects, if they

are successfully resolved.

The question which is addressed in this tutorial is: how do we use this collision data to move the intersecting objects apart?

A simple solution would be to simply move the objects along the collision normal by a distance equal to the penetration depth, by directly changing the position vectors. This is known as the *projection method*, and while it will suffice for a simple simulation, it has some fairly obvious drawbacks related to the objects' velocities. If the objects are moved apart without changing their velocities then they will just continue along the same path during the next physics update and are likely to intersect again; alternatively if the objects are moved apart and the velocity then set to zero then the simulation feels very unrealistic, as objects tend to bounce off one another rather than stop dead on first contact. We clearly need a solution which affects the velocities and/or accelerations of the objects rather than directly affecting the positions.

Algorithms which directly affect the velocities of the intersecting objects are known as *Impulse Methods*, whereas algorithms which directly affect the acceleration of the bodies are known as *Penalty Methods*. Penalty methods use spring forces to, in effect, pull the objects away from each other by affecting the acceleration through Newton's second law ($F = ma$). Impulse methods use instantaneous nudges, or impulses, to push the objects apart by directly controlling their velocities. In this tutorial we will concentrate on impulse methods, and in the next tutorial we will discuss penalty methods.

In summary:

- **Projection methods -** control the position of the intersecting objects directly.
- **Impulse methods -** control the velocity of the objects, i.e. the first derivative.
- **Penalty methods -** control the acceleration of the objects, i.e. the second derivative.

Impulses

The impulse method allows us to directly affect the velocities of the simulated objects which have intersected. This is achieved through the application of an impulse, which can be thought of as an immediate transfer of momentum between the two bodies. Impulse is a term defined by classical physics as the accumulated force applied to a body over a specific amount of time (it is therefore measured in Newton seconds Ns). The impulse J is defined in terms of force F and time period Δt as

$$J = F\Delta t$$

We know from Newton's second law, that $F = ma$, and we can also write the acceleration a as the rate of change of velocity v. Substituting these values into the equation for impulse gives us:

$$J = F\Delta t = ma\Delta t = m\frac{\Delta v}{\Delta t}\Delta t = m\Delta v$$

We also know that momentum is equal to the product of mass and velocity, so an impulse is equivalent to the change in momentum.

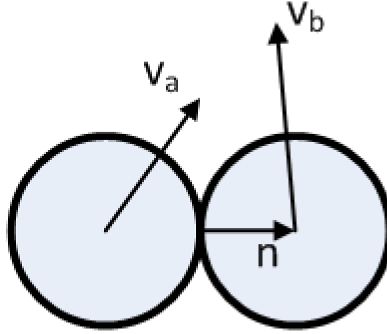
Our plan then is to give colliding objects a nudge, by changing their velocity by an amount equal to

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

The question then, is how to calculate the impulse J generated when two bodies collide.

Calculation of Linear Impulse

First, we will discuss what we would like to happen to the velocities of two colliding objects - namely, we want the bodies to bounce off one another. We will consider the simple case of two spheres colliding, as shown in the figure – sphere a is moving with velocity v_a, while sphere b has velocity v_b; the collision



normal is n. We want to calculate the impulse J.

The impulse is generated by the velocity at which the two spheres have collided so we are interested in the relative velocity of the two objects, which we will label v_{ab}. The component of the relative velocity which caused the collision is along the normal vector, so we calculate the dot product of the relative velocity and the normal:

$$v_{ab} = v_a - v_b$$

$$v_n = v_{ab} \cdot n$$

The velocity along the normal after collision is dependent on the *coefficient of elasticity* ε. A coefficient of 1 means the collision will be purely elastic, so all the velocity is transferred, whereas a coefficient of zero is purely non-elastic, so no velocity is transferred. A purely non-elastic collision will result in the two bodies staying together (i.e. no bounce); a purely elastic collision is a perfect bounce so no damping or slowing down occurs. Your simulation is likely to require a figure somewhere in between. Quite often different object types in a game will have different coefficients of elasticity which are stored as a member of the object class, in the same way as the mass is.

The coefficient of elasticity is the factor by which the velocity before the collision is multiplied to calculate the velocity after the collision. Hence:

$$v_n^+ = -\varepsilon v_n^-$$

or, substituting for v_n:

$$(v_a^+ - v_b^+) \cdot n = -\varepsilon(v_a^- - v_b^-) \cdot n$$

Note the introduction of $^-$ and $^+$ nomenclature to denote the state of the bodies before and after the collision respectively. Also note the negation of the velocity - remember we want to push the two bodies back apart in the opposite direction to their colliding velocity.

We also need to think about the momentum of the two bodies. You will remember from the first tutorial in this series on Newtonian mechanics that the total momentum must remain constant in any collision. However our plan to resolve the collision is to "inject" some momentum into the system. Hence we need to ensure that the overall additional momentum is equal to zero, which is achieved by making the momentum used to nudge the second body, the exact opposite of that used to nudge the first. This is shown in the equations below, showing the relationship between momentum before and after the collision for each body, where J is the injected impulse along the normal vector n.

$$m_a v_a^+ = m_a v_a^- + Jn$$

$$m_b v_b^+ = m_b v_b^- - Jn$$

Note that it is the injected *momentum* which is equal and opposite, not the *velocity* as that will be affected by the mass of the object and therefore unequal for differently sized objects. Combining the last three equations, allows us to solve for the impulse J

$$J = \frac{-(1+\varepsilon)v_{ab} \cdot n}{n \cdot n(\frac{1}{m_a} + \frac{1}{m_b})}$$

which in turn allows us to calculate the velocities of the two bodies after the collision:

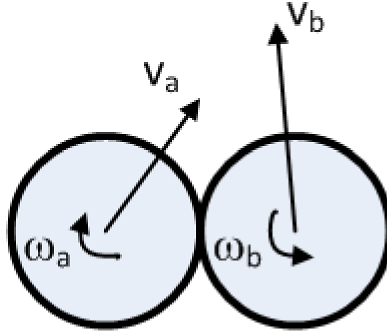
$$v_a^+ = v_a^- + \frac{J}{m_a}n$$

$$v_b^+ = v_b^- - \frac{J}{m_b}n$$

So we can now calculate the velocity at which two colliding bodies should move away from one another in a manner which feels believable, as it is based on Newtonian mechanics, and which incorporates a "bounciness" factor for different types of object in the form of the coefficient of elasticity. Remember, as ever, that the velocities we are discussing are three dimensional vectors – the bodies can move along all three axes of the simulated world. They can also *rotate* around all three axes, so next we need to consider how to account for rotational elements in collision response.

Calculation of Angular Impulse

Let's add some spin to our two colliding spheres. The angular velocity of each is ω_a, ω_b and the radius is r_a, r_b respectively



In order to realistically add angular motion to a collision response, we need some more information about the actual contact point between the two objects – specifically the velocity of that point on each of the objects. This is the sum of the object's linear velocity, and the extra tangential velocity v_t created by the fact that the point is rotating around the object's centre. The velocity at a point which is distance r from the centre on an object turning with angular velocity ω is

$$v_t = \omega r$$

Remember that rotational velocity is measured in radians, and there are 2π radians in a full revolution; similarly the distance around that revolution is $2\pi r$.

So the velocity of the contact point v_C on each object is:

$$v_{Ca} = v_a + \omega_a r_a$$

$$v_{Cb} = v_b + \omega_b r_b$$

Again we need to ensure that angular momentum is conserved, so:

$$I_a\omega_a^+ = I_a\omega_a^- + r_a \times Jn$$

$$I_b\omega_b^+ = I_b\omega_b^- - r_b \times Jn$$

Remember that the angular momentum of a body is the product of the angular velocity ω and the inertia tensor I. J is the impulse which we are calculating and n is the collision normal. Solving our equations for J, taking into account the angular momentum, leads to a much more complicated looking calculation for the impulse:

$$J = \frac{-(1+\varepsilon)v_{ab} \cdot n}{n \cdot n(\frac{1}{m_a} + \frac{1}{m_b}) + [(I_a^{-1}(r_a \times n)) \times r_a + (I_b^{-1}(r_b \times n)) \times r_b] \cdot n}$$

which again allows us to calculate the velocities of the two bodies after the collision, as well as the angular velocities:

$$v_a^+ = v_a^- + \frac{J}{m_a} n$$

$$v_b^+ = v_b^- - \frac{J}{m_b} n$$

$$\omega_a^+ = \omega_a^- + \frac{r_a \times Jn}{I_a}$$

$$\omega_b^+ = \omega_b^- + \frac{r_b \times Jn}{I_b}$$

These equations allow us to write an algorithm in C++ to believably simulate two objects colliding and bouncing off one another using Newtonian mechanics. Both linear and angular movement are accounted for, and the elasticity of the collision is also incorporated.

More Complex Shapes

The simple case of two colliding spheres has been used to illustrate the algorithms employed in impulse method collision response. The algorithms are perfectly suited to more complex three-dimensional shapes. In fact, you should notice that there are no assumptions about the shapes made in the calculations. When implementing the code, you will see that all calculations are carried out in three dimensions, so for example the distance of a contact point from the centre of an object is a vector containing three values – it does not matter whether that point is on the surface of a sphere or a more complex shape. Similarly the relationship between angular velocity and linear velocity is irrespective of the object shape, it is based purely on the distance of the point of interest from the centre of rotation.

Implementation

The aim of this practical session is to expand your physics engine to react to collisions between simulated objects. We will implement the impulse method for both linear and angular motion. To demonstrate that the collision tests are working, we will add functionality to the project which bounces colliding objects off one another.

```
1
2  static void AddCollisionImpulse( Cube& c0,
3               Cube& c1,
4               const Vector3& hitPoint,
5               const Vector3& normal,
6               float penetration)
7  {
8     // Some simple check code.
9     float invMass0 = (c0.m_mass>1000.0f) ? 0.0f : (1.0f/c0.m_mass);
10    float invMass1 = (c1.m_mass>1000.0f) ? 0.0f : (1.0f/c1.m_mass);
11
12     invMass0 = (!c0.m_awake) ? 0.0f : invMass0;
13    invMass1 = (!c1.m_awake) ? 0.0f : invMass1;
14
15    const Matrix worldInvInertia0 = c0.m_invInertia;
16    const Matrix worldInvInertia1 = c1.m_invInertia;
17
18    // Both objects are non movable
19    if ( (invMass0+invMass1)==0.0 ) return;
20
21    Vector3 r0 = hitPoint - c0.m_c;
22    Vector3 r1 = hitPoint - c1.m_c;
23
24    Vector3 v0 = c0.m_linVelocity + Cross(c0.m_angVelocity, r0);
```

```
25     Vector3 v1 = c1.m_linVelocity + Cross(c1.m_angVelocity, r1);
26 
27     // Relative Velocity
28     Vector3 dv = v0 - v1;
29 
30     // If the objects are moving away from each other
31     // we dont need to apply an impulse
32     float relativeMovement = -Dot(dv, normal);
33      if (relativeMovement < -0.01f)
34     {
35         return;
36     }
37 
38     // NORMAL Impulse
39     {
40        // Coefficient of Restitution
41        float e = 0.0f;
42 
43        float normDiv = Dot(normal, normal) *  ( (invMass0 + invMass1)
44           + Dot( normal, Cross( Transform( Cross(r0, normal),
45           worldInvInertia0), r0)
46           + Cross( Transform( Cross(r1, normal),
47           worldInvInertia1), r1) ) );
48 
49        float jn = -1*(1+e)*Dot(dv, normal) / normDiv;
50 
51        // Hack fix to stop sinking
52        // bias impulse proportional to penetration distance
53        jn = jn + (penetration*1.5f);
54 
55        c0.m_linVelocity += invMass0 * normal * jn;
56        c0.m_angVelocity += Transform(Cross(r0, normal * jn),
57           worldInvInertia0);
58 
59        c1.m_linVelocity -= invMass1 * normal * jn;
60        c1.m_angVelocity -= Transform(Cross(r1, normal * jn),
61           worldInvInertia1);
62      }
63 
64      // TANGENT Impulse Code
65     {
66        // Work out our tangent vector, with is perpendicular
67        // to our collision normal
68        Vector3 tangent =Vector3(0,0,0);
69        tangent = dv - (Dot(dv, normal) * normal);
70        tangent = Normalize(tangent);
71 
72        float tangDiv = invMass0 + invMass1
73           + Dot( tangent, Cross((Cross(r0, tangent)
74           * c0.m_invInertia), r0)
75           + Cross((Cross(r1, tangent) * c1.m_invInertia), r1) );
76 
77        float jt = -1 * Dot(dv, tangent) / tangDiv;
78        // Clamp min/max tangental component
79 
80        // Apply contact impulse
81        c0.m_linVelocity += invMass0 * tangent * jt;
82        c0.m_angVelocity += Transform(Cross(r0, tangent * jt),
```

```
83            worldInvInertia0);
84
85        c1.m_linVelocity -= invMass1 * tangent * jt;
86        c1.m_angVelocity -= Transform(Cross(r1, tangent * jt),
87            worldInvInertia1);
88     } // TANGENT
89 }
```

Impulse Method

Tutorial Summary

In this tutorial we have introduced the concept of collision response with particular focus on impulse methods, i.e. a method which directly affects the velocities of colliding objects in order to resolve that collision. In the next tutorial we will look at penalty methods for collision response.