Physics - Collision Response

Introduction

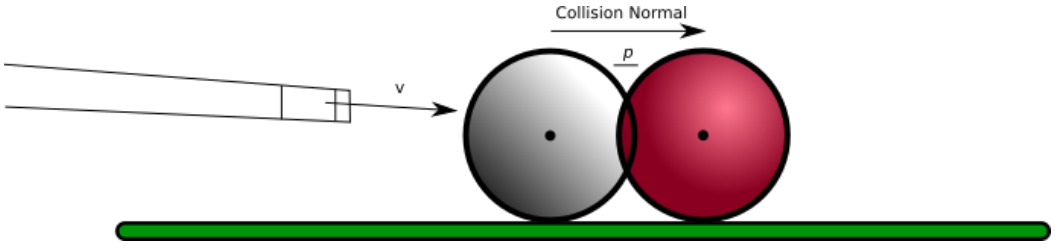
We now know methods by which to calculate intersections between some simple shapes, and determine the contact point, collision normal, and intersection distance. That's only half of the solution, though - as well as detecting the collisions between our physics bodies, we must then calculate the correct collision response to those collisions, to maintain the physical accuracy of our simulation. Two physical objects can't occupy the same point in space at the same time, so part of collision response is to separate out the objects. As well as this, we also have to determine whether the objects impart forces upon each other - to reuse the snooker example from earlier tutorials, if the cue ball hits a another ball, we first *detect* that the two balls have hit, and then determine the correct physical *response* to the objects colliding, by calculating the amount of energy to transfer from one ball to the other.

Collision Response

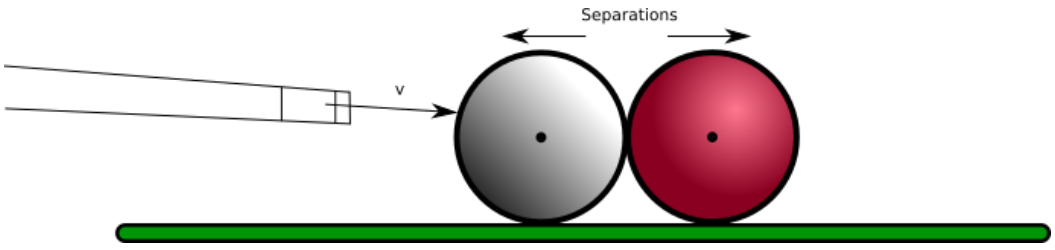
As the objects in our simulation move around due to forces imparted upon them, they will at some point intersect - something we can now detect for spheres and cubes. To maintain the consistency of the simulation, as well as detecting the collision, we must resolve it, by moving the objects such that they stop intersecting, and if necessary, changes the momentum of each object in a realistic way. To do this, we must in some way change the position of the colliding objects over time. We've seen that we can change an object's position directly, but also by changing the derivatives of position - velocity and acceleration. Each of these is a valid solution to our collision response, and so there are methods which use all three in some capacity.

The Projection Method

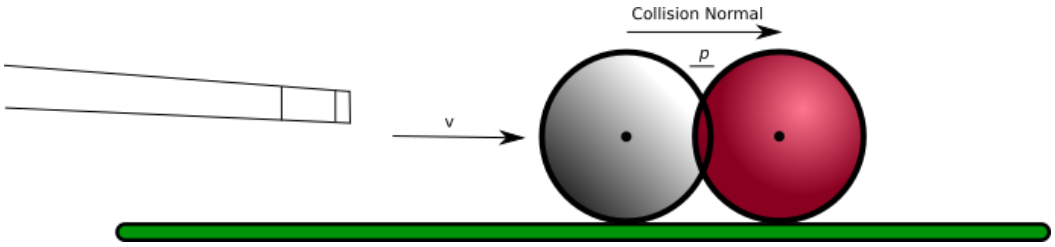
Perhaps the simplest and most intuitive method of resolving a collision would be to simply separate them out by changing their position, in opposite directions along the collision normal. This is known as the *projection method*, and while simple, does work to some extent. It does have some drawbacks, though. If we again consider the case of the snooker game, and imagine the cue ball has been hit, and has traveled in a straight line, until it collides with another ball:



We can then 'solve' the collision by separating them out along the contact normal:



That provides some sort of response to the collision, for one frame. What then happens in the next frame, though? The cue ball is still moving at its previous velocity, and the collided with ball has no current velocity, as no energy was transferred from the cue ball to the other, we just modified their position vectors. In the next frame, then, they collide again, as the cue ball moves forward and collides with the other ball. This just keeps happening every frame, and the cue ball 'pushes' the other ball along the collision normal until they no longer collide - if the collision normal and cue ball velocity direction match up exactly, though (the dot product between them is 0), then the ball is just pushed along forever, or at least until friction and damping reduce the cue ball's velocity to zero:



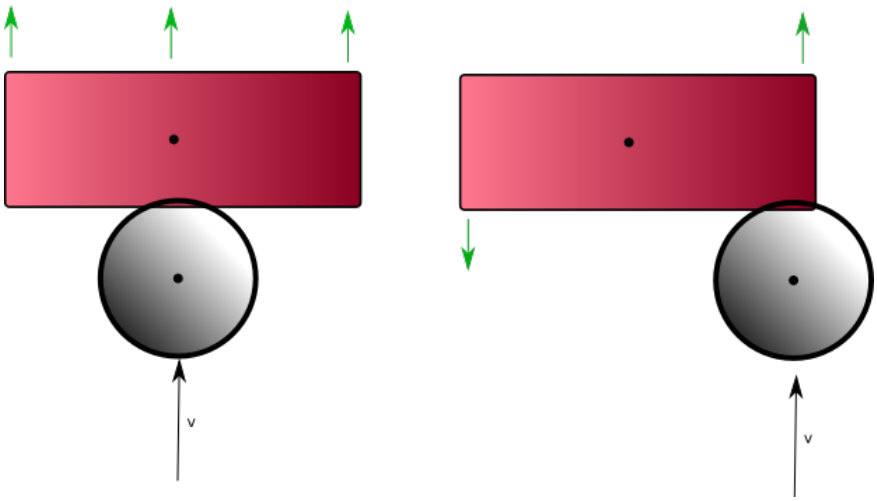
As the projection method doesn't take into account the relative velocities, or masses, between the objects, its not particularly suitable on its own. Consider if instead the cue ball hit a *planet* - the projection method would just move both the planet and the ball along the collision normal by the same amount, and the physical accuracy of the simulation would be lost.

We can improve this separation concept by, instead of just separating the objects out by an equal amount, instead separating them out by an amount proportional to their mass - so if our cue ball above was somehow twice as heavy as the red ball, it would be moved along the collision normal by half as much as the red ball. This helps for our ball hitting a planet example, but won't stop the simple Snooker example from not 'feeling' right as the cue ball still doesn't lose any velocity, so it keeps going, and keeps pushing the red ball.

The Impulse Method

Rather than directly modify the position of an object, its possible to instead resolve the collision by modifying the *first derivative* of position; that is, the *velocities* of the colliding objects. This can be achieved by calculating a new velocity for each object that will move them apart in a realistic manner, taking into consideration their relative mass, and current velocity (that is, each object's *momentum*). An instantaneous change in velocity, without first integrating from acceleration with regard to time, is known as an *impulse*, and so collision response using these is known as the *impulse method*.

To correctly apply an impulse, we must take into consideration both the linear, and angular velocities of the objects, and modify them accordingly. To understand why, here's a couple of simple cases of objects colliding:



In the left example, the sphere is hitting the cuboid straight on, and it seems intuitive enough that the cuboid will be pushed away. In the right example, the sphere is hitting the corner of the cuboid, and the 'correct' thing that would occur would be that while neither the sphere or the cuboid had any angular motion at the point of collision, that the cuboid should 'twist' under the impact - these expected results are indicated using green arrows in the example.

To see how an impulse J applied over the course of time t relates to a change in momentum, we can break it down into the individual equations. Generally we have a t of a single frame's time delta, as we want to 'fire and forget' our impulse and have it do the right thing. As to why J is the standard mathematical notation for an impulse...no-one can really remember, but may be related to not being able to use 'i' as it was already commonly being used to denote *inertia* in physics equations, where it could cause confusion.

We start with the simple definition of our impulse J being defined as some force F applied over time t:

$$J = F\Delta t$$

Newton's second law states that $F = ma$, (that is, mass times acceleration), or alternatively the rate of change in velocity (v)over time period t:

$$\begin{gathered} J = F\Delta t \\ J = ma\Delta t \\ J = m\tfrac{v}{\Delta t}\Delta t \\ J = m\Delta v \end{gathered}$$

Now that we can see that an impulse **J** is just a change in velocity, what do we do with it, and how do we actually determine what to plug into our equation to calculate the correct **J** for a given collision? The purpose of the impulse is to in some way change the velocity of each of our colliding objects, in such a way that the *momentum* is conserved, and to therefore take into consideration velocity and mass:

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

If we calculate the correct J, then each object should move by an amount proportional to its mass; while the overall *momentum* of the collision must be conserved, that doesn't mean that the two objects should move by an equal amount, and an object with a larger mass should move less in a collision than one with less mass (to conserve the momentum of our colliding objects, that just means that the heavier object adds more velocity to the lighter object than vice-versa).

Coefficient of restitution

Before we get into the full calculations for calculating impulses, its worth having a brief think about what the colliding objects are made of, and how that might affect the response to collisions between them. If we were to drop two balls, each weighing 1KG, onto the floor from the same height, we would expect both of them to react the same way. But what if one ball were made of rubber, and one were made of steel? The rubber ball would probably bounce up in the air, while the steel ball would bounce far less; to put it another way, after colliding with the floor, the rubber ball retains more of its velocity than the steel ball.

The ratio between an object's velocity before and after a collision is known as the *coefficient of restitution* (usually denoted e), and represents loss of kinetic energy due to it being converted to heat, or the object's material being deformed. This coefficient ranges between 0 and 1 for most combinations of material, with a value of 1 being said to be a perfectly *elastic* collision, with less than 1 being an *inelastic* collision. A value of above 1 would *gain* kinetic energy, meaning that mass, heat, or light was in some way being transformed into a change in velocity - perhaps the material is exploding!

Really, the coefficient of restitution relates to the expected change in velocity after a particular *pair* of collision objects have collided, so the coefficient would be different for two steel balls colliding than it would for a steel ball and a rubber ball colliding. While we could use a lookup table of some sort to store the exact coefficients for a given object 'material', in code its common to approximate the coefficient by storing a restitution value per object, and then multiplying the two restitutions together to give us a value which determines how much kinetic energy is lost.

Calculation of Linear Impulse

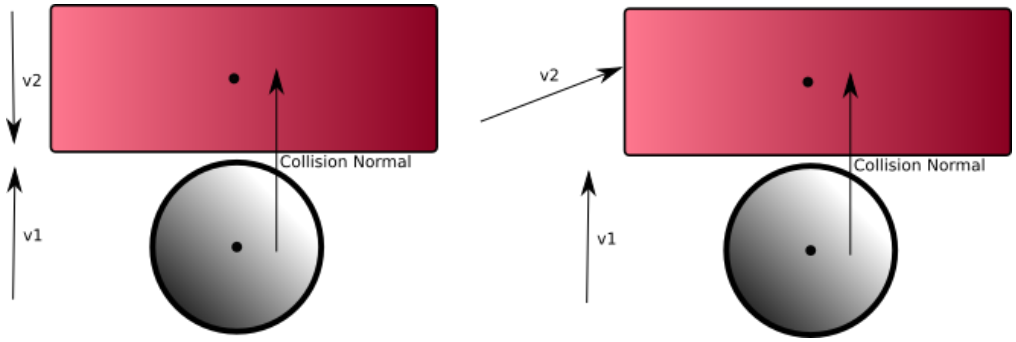
To see how to build up the correct calculation for our impulse response J, we are going to consider a simple collision between two rigid bodies, each with a velocity v_x, an inverse mass m_x^{-1}, and an inverse inertia tensor I_x^{-1}, where x in each case will be 1 for the first object, and 2 for the second. If you can't remember why we're using *inverse* mass and inertia, remember that it helps us turn some equations from divides into multiplies, and lets us easily represent objects with 'infinite' mass and so should never move.

When calculating the value of the linear component of the impulse response, we need to take into consideration the *relative* velocity between the two objects:

$$v_r = v_1 - v_2$$

It doesn't matter whether our object's are moving forward at 1m/s and 2m/s or 1001 m/s and 1002 m/s - the response to the collision between them should be the same.

We also need to know about how much of that relative velocity is in the direction of the collision normal. To understand why, consider the following two cases of objects moving at the same two speeds, but different directions:



In both cases the relative velocity between the objects is the same, but in the right case the objects are only narrowly colliding in the direction they are traveling in, while on the left the collision normal matches up to the direction being traveled in, and so should result in more energy being transferred.

To calculate how much of the relative velocity is in the direction of the collision normal $\hat{n}$, we can simply use the dot product operator. This, combined with the coefficient of restitution of the colliding objects, allows us to determine the total velocity of the collision:

$$v_j = -(1+e)\mathbf{v}_r \cdot \hat{\mathbf{n}}$$

We aren't quite done! We still need to take into consideration the conservation of momentum. Remember that momentum is mass times velocity, and that we want the overall momentum of our colliding objects to be the same after they have collided as it was before (which is not to say that velocity directions or magnitudes won't change, its just that between the two objects it will equate to the same). To conserve the momentum, we divide the velocity of the impulse by the sum of the inverse masses of the objects - this essentially scales the velocity so that when we apply it on each object it gets the correct proportion of the total energy of the collision (remember that $\mathbf{J} = m\Delta v$; we've calculated the total of the change in v, it just then needs to be scaled by the m^{-1} of each object).

$$\mathrm{J} = \frac{v_j}{(m_1^{-1}+m_2^{-1})}$$

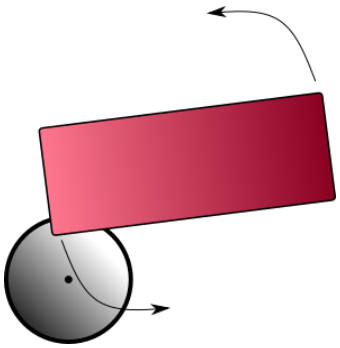
Once we have the impulse $\mathbf{J}$, we can adjust the velocities of each of the colliding objects, such that they react to the collision, and conserve the momentum of the system:

$$v_1^+ = v_1 + m_1^{-1} J\hat{n}$$
$$v_2^+ = v_2 - m_2^{-1} J\hat{n}$$

Each object will therefore end up gaining some velocity along the collision normal (either forwards or backwards) proportionally to their inverse mass, moving the objects apart by a varying amount. Now, if our cue ball hits another ball, it will stop, while the other ball will be knocked forward, and allowing us to play a game of Snooker.

Calculation of Angular Impulse

That's not all, though. As well as linear velocity, we need to take into consideration the *angular* velocity of our objects, too. To understand why, here's a simple example:



Imagine that neither the sphere or the box are moving in space (their *linear* velocity is zero), but that the red box is spinning around. At some point the box will collide with the sphere, generating a contact point on the edge of the box. Clearly, the sphere must be knocked downwards by the spinning box, but as so far we've only been considering linear velocity in our impulse **J** will in this case have a magnitude of zero, and nothing will happen.

Clearly then, we must augment the impulse equation to take into consideration angular velocity. If you recall back in the angular motion tutorial, we were introduced to *inertia tensors*, and the idea that we apply a *torque* to our objects when a force is applied on their surface, with the magnitude of the torque being proportional to the distance from the centre of mass. From this concept, it stands to reason that angular velocity can *create* a force, with a magnitude proportional to the distance from the centre of mass - Newton's third law states that every action has an equal and opposite reaction, after all!

To take into consideration the added force that angular velocity may apply, we need to further scale the impulse velocity, not just by the inverse mass of the objects, but by their inverse inertia tensors, too. We must calculate the amount of angular velocity each object has at the collision point by taking the cross product between the relative collision point and the collision normal, and then cross *that* by the relative position on the object once again - this effectively scales the angular velocity by how much it is moving in the direction of the collision normal:

$$\theta_1 = {\mathbf{I}_1}^{-1}(\mathbf{r}_1 \times \hat{\mathbf{n}}) \times \mathbf{r}_1$$
$$\theta_2 = {\mathbf{I}_2}^{-1}(\mathbf{r}_2 \times \hat{\mathbf{n}}) \times \mathbf{r}_2$$

We can then modify the *angular* velocity of the colliding objects by getting the cross product between the local collision position and the applied impulse force, and scaling the result by the inverse inertia tensor - much in the same way as we did when we applied forces using raycasts.

$$av_1^+ = av_1 + {\mathbf{I}_1}^{-1}(r_1 \times J\hat{n})$$
$$av_2^+ = av_2 - {\mathbf{I}_2}^{-1}(r_2 \times J\hat{n})$$

Objects receiving a change in angular velocity due to collision resolution is why we set the relative collision point on AABBs to a zero length vector - we don't' want our AABBs to twist at all (or they wouldn't be AABBs!), so we give them a collision point that would result in an angular change of 0 (the results of the cross products will have a magnitude of zero).

Combined Impulse Calculation

To turn the angular velocities of our two objects back into scalar values suitable for adding into our impulse function, we take the dot product of them versus the collision normal direction. We can then add this into the divisor of our impulse vector, along with the inverse masses, giving us a final impulse equation of:

$$J = \frac{-(1+e)\mathbf{v}_r \cdot \hat{\mathbf{n}}}{{m_1}^{-1} + {m_2}^{-1} + ({\mathbf{I}_1}^{-1}(\mathbf{r}_1 \times \hat{\mathbf{n}}) \times \mathbf{r}_1 + {\mathbf{I}_2}^{-1}(\mathbf{r}_2 \times \hat{\mathbf{n}}) \times \mathbf{r}_2) \cdot \hat{\mathbf{n}}}$$

It *looks* pretty nasty, but it boils down to a single scalar value, and gives us everything we need to determine the collision force, and allow us to conserve momentum (both linear and angular). Once we calculate the total sum of velocity for the impulse, we then adjust the linear and angular velocity of the objects, and in doing so resolve the collision.

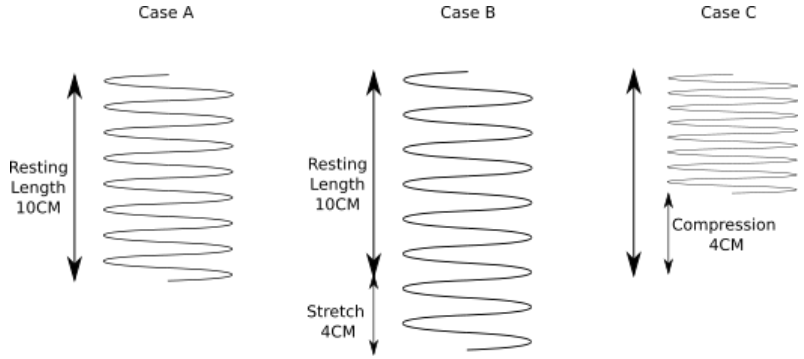
The Penalty Method

We've seen how we can resolve collisions by directly modifying the position of the colliding objects (via the projection method), and via the first derivative of position (the impulse method). It stands to reason that we could also adjust the second derivative of position (the object's acceleration) to move them apart, and thus resolve the collision. Methods of collision resolution which adjust acceleration are known as *penalty* methods. We saw in the linear motion tutorial that we change the acceleration of an object by adding a force, so penalty methods calculate a force to apply that will, over time, resolve the collision. This is generally done by applying spring calculations to the colliding objects - the more the objects penetrate, the more the 'spring' that connects them wants to snap back to its resting length.

We can model a spring by using *Hooke's Law*, which calculates the force applied at either end of a spring as it is either compressed or extended. The law can be represented as a single equation:

$$F = -kx$$

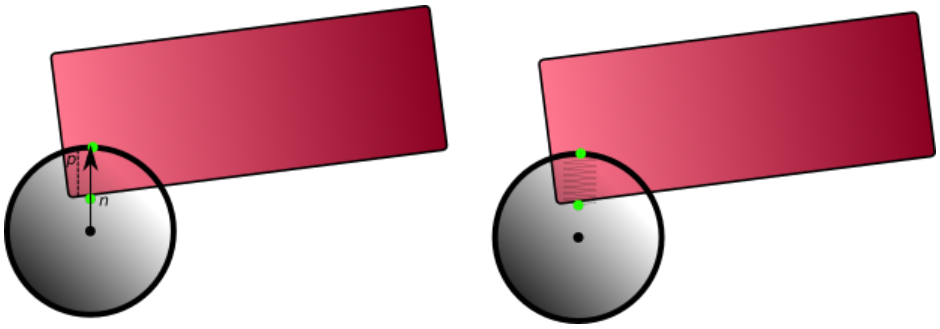
This states that the spring force F is the result of the difference in spring length x from the 'resting' length, multiplied by spring constant k, which represents how 'snappy' the spring is, so the larger k is, the more force is applied, and the quicker the spring tries to return to its resting length. Here's some examples of springs, and how Hooke's law determines how much force they apply:



In this case, the resting length of the spring is 10cm - if we either try to stretch the spring out by 4cm, or 'squish' it by 4cm, we get the same resulting force magnitude (we'll assume the k constant is 1.0 for now), just with opposite directions:

$$F_a = 0 = -(10 - 10)$$
$$F_b = 4 = -(10 - 14)$$
$$F_c = -4 = -(10 - 4)$$

From this, we can see that stretching and compressing a spring produces the same amount of force, and that if we added in that spring constant, we'd just scale the resulting force. Springs sound interesting, but how exactly do we use them to resolve collision detection? We can use this spring equation by modeling a spring with a resting length of zero, attached to the collision points of the collision:



We can state that the temporary spring has a resting length of *zero*, and that it is being extended along the collision normal n by the penetration distance p. From this, we can apply a force proportional to the penetration distance, at the collision point on each object, in the direction of the collision normal (and thus giving us both an acceleration and a torque for that frame), much the same as when we applied forces at a specific point when we performed raycasting. At this point, it is worth recalling that the resulting acceleration of a force is $a = Fm^{-1}$, which means that we still get some 'realism' from our collision resolution in that lighter objects will move more than heavier objects. As the objects start to move apart due to the force of the spring, it will in subsequent frames apply *less* force (the penetration distance is decreasing, and thus so is x in Hooke's law). If the objects continue to move together, the spring force will get greater and greater, modifying their velocity until they start to separate.

Game response to collisions

Sometimes it's not just the physics engine that needs to know about a collision - the gameplay code might need to as well. In an FPS game, if you hit another player with your rocket, they might get knocked back (the physics response to the collision), but will also lose some health or armour (the gameplay response to the collision). This is usually done in games by tagging certain physics object types as being collideable with each other, and then having some way of informing the game that a collision has started (or sometimes even ended - maybe the player stops losing health once they stop colliding with the 'lava' tiles). For example, in games built with the *Unity* game engine, programmers have access to the **OnCollisionEnter** and **OnCollisionExit** methods, which are called by the physics engine to inform the gameplay code of any object implementing those methods in its gameplay script that it has hit another object.

Implementing some sort of method by which the physics engine can tell your game that collisions are occurring is important to creating an interactive experience. There's a variety of different ways this could be achieved, and there are no hard and fast rules, although some ways may have performance implications. The codebase you have been provided with has *OnCollisionBegin* and *OnCollisionEnd* end virtual functions, allowing custom gameplay interactions can be created - the game will be told when collisions occur, and you can then add some gameplay code in a number of subclasses to represent the rockets / cars / people/ robots that make up your game. However, calling it in this fashion is harmful to the efficiency of your physics engine, due to the instruction cache of the CPU. Computers work best when they're doing the same thing over and over again, on an ordered set of data - all of the code will be sitting in the instruction cache making it fast to run, and the data access is predicable, helping the CPU to precache the correct data, so it doesn't have to stop and wait for data to be loaded from main memory. If we decide to call a virtual function in the middle of our physics calculations, the CPU then stops what it was doing, and starts executing your virtual function, which might then make the 'physics' code drop out of cache, so that when the virtual function ends, the CPU must wait a while before it can run the next bit of collision detection code. What's worse, is that if the physics engine runs multiple collision detection calculations per 'game frame', or is multithreaded, then those functions could be called at the wrong time, on the wrong core, the wrong number of times per game update!

A better solution may be to add collisions to a list, which can then be iterated through once all of the collision detection has been completed, or maybe to even tag 'special' physics objects that have gameplay side effects (like those rockets in the earlier example), and only add those objects to a list.

Tutorial Code

To get collision response going in our physics engine, we're going to implement a combination of the projection method, and the impulse method. This lets us separate objects out quickly (the remove the object penetration that frame), and then move the objects away in a realistic manner. We only need to fill out one more method in the **PhysicsSystem** class, *ImpulseResolveCollision*. As the section on the impulse method may have suggested, the code is going to be a fairly long, and involve quite a lot of vector operations, but we only need to calculate it once, and we don't need to consider the different collision shapes any more, as any of the collisions reduce down to the collision normal, penetration

distance, and collision point, each of which we'll be using at different parts in this code. Here's the first part of the method, which will take in a pair of **GameObjects**, and a collision point, and then apply the correct forces to the objects, based on the collision information:

```
1  void PhysicsSystem::ImpulseResolveCollision(
2     GameObject& a, GameObject& b,
3     CollisionDetection::ContactPoint& p) const {
4     PhysicsObject* physA = a.GetPhysicsObject();
5     PhysicsObject* physB = b.GetPhysicsObject();
6
7     Transform& transformA = a.GetTransform();
8     Transform& transformB = b.GetTransform();
9
10    float totalMass = physA->GetInverseMass()+physB->GetInverseMass();
11
12 //Separate them out using projection
13    transformA.SetWorldPosition(transformA.GetWorldPosition() -
14     (p.normal * p.penetration*(physA->GetInverseMass() / totalMass)));
15
16    transformB.SetWorldPosition(transformB.GetWorldPosition() +
17     (p.normal * p.penetration*(physB->GetInverseMass() / totalMass)));
```

PhysicsSystem::ImpulseResolveCollision method

To make things a bit less error-prone (and easier to read!), the first thing we're going to do is get the physics objects of our two colliding objects, and their transforms, and store them in some variables (lines 4 - 8). We're also going to determine the total inverse mass of the two objects, as we're going to use this later on to calculate impulse **J**, but also for our simple object projection.

On lines 13-17, we push each object along the collision normal, by an amount proportional to the penetration distance, and the object's inverse mass. By dividing each object's mass by the *totalMass* variable, we make it such that between the two calculations, the object's move by a total amount of *p.penetration* away from each other, but a 'heavier' object will move by less, as it makes up less of the 'total mass' (remember, we're dealing with *inverse* mass, so heavier objects have lower values).

That's all we need to do to fulfill collision response via the projection method - if we used just this, objects would separate out, and the boxes and spheres of our test world would happily sit on the floor. The floor would never move, as its inverse mass is 0, and so will be projected by an amount of 0 units along the collision normal, and any object falling on it would move by the full penetration amount. What *won't* happen though, is any conservation of momentum, via a change in the amount of linear or angular velocity the objects. To do that, we need to start building up the variables we need to calculate the impulse value **J**, and where on each object to apply the impulse. To do this, we need the collision points relative to each object's position, something we've been calculating in our collision detection code previously (lines 18 and 19). From these points, we can determine how much angular velocity the object has at that point (the further away from the centre of the object a point is, the faster it moves as the object rotates). We can determine how much the object is moving via the cross product (line 21 and 22), much in the same way as we determine the amount of torque using the cross product when applying angular forces. Now, from a combination of each object's linear and angular velocity, we can determine the velocities at which the two objects are colliding, via simple addition (lines 26 - 27), and thus the relative velocity of the collision (line 29).

```
18    Vector3 relativeA = p.localA;
19    Vector3 relativeB = p.localB;
20
21    Vector3 angVelocityA =
22          Vector3::Cross(physA->GetAngularVelocity(), relativeA);
23    Vector3 angVelocityB =
24          Vector3::Cross(physB->GetAngularVelocity(), relativeB);
25
26    Vector3 fullVelocityA = physA->GetLinearVelocity() + angVelocityA;
27    Vector3 fullVelocityB = physB->GetLinearVelocity() + angVelocityB;
28
29    Vector3 contactVelocity = fullVelocityB - fullVelocityA;
```

PhysicsSystem::ImpulseResolveCollision method

Now to start building up that impulse vector **J**, which for reference, was calculated as follows:

$$J = \frac{\overbrace{(1+\epsilon)\mathbf{v}_c \cdot \hat{\mathbf{n}}}^{\text{impulseForce}}}{\underbrace{m_1^{-1} + m_2^{-1}}_{\text{totalMass}} + \underbrace{(\mathbf{I}_1^{-1}(\mathbf{r}_1 \times \hat{\mathbf{n}}) \times \mathbf{r}_1 + \mathbf{I}_2^{-1}(\mathbf{r}_2 \times \hat{\mathbf{n}}) \times \mathbf{r}_2) \cdot \hat{\mathbf{n}}}_{\text{inertiaA}\quad\text{inertiaB}\quad\text{angularEffect}}}$$

To make this easier to follow, the variable names that store each part of the equation have been noted down, so you can see how the codebase implements each of the bits that make up **J**. For now, we're going to hardcode the coefficient of restitution to something that will dissipate some energy - see what happens to the collisions when this variable has been changed (especially when the value is greater than 1...).

```
30    float impulseForce = Vector3::Dot(contactVelocity, p.normal);
31
32    //now to work out the effect of inertia....
33    Vector3 inertiaA     = Vector3::Cross(physA->GetInertiaTensor() *
34                  Vector3::Cross(relativeA, p.normal), relativeA);
35    Vector3 inertiaB     = Vector3::Cross(physB->GetInertiaTensor() *
36                  Vector3::Cross(relativeB, p.normal), relativeB);
37    float   angularEffect= Vector3::Dot(inertiaA + inertiaB, p.normal);
38
39    float cRestitution = 0.66f; //disperse some kinectic energy
40
41    float j = ( -(1.0f + cRestitution) * impulseForce)  /
42              (totalMass + angularEffect);
43
44    Vector3 fullImpulse = p.normal * j;
```

PhysicsSystem::ImpulseResolveCollision method

The last bit of the function is to apply the linear and angular impulses, in opposite directions, and providing a realistic response to our object collisions:

```
45    physA->ApplyLinearImpulse(-fullImpulse);
46    physB->ApplyLinearImpulse( fullImpulse);
47
48    physA->ApplyAngularImpulse(Vector3::Cross(relativeA,-fullImpulse));
49    physB->ApplyAngularImpulse(Vector3::Cross(relativeB, fullImpulse));
50 }
```

PhysicsSystem::ImpulseResolveCollision method

As we've not used the *ApplyLinearImpulse* or *ApplyAngularImpulse* methods before, we'll need to fill them in with the following code:

```
void PhysicsObject::ApplyAngularImpulse(const Vector3& force) {
   angularVelocity += inverseInteriaTensor * force;
}

void PhysicsObject::ApplyLinearImpulse(const Vector3& force) {
   linearVelocity += force * inverseMass;
}
```

PhysicsObject Class

Each one just scales its input by the appropriate inverse mass representation, and adds it to the appropriate velocity vector.

To see where our newly filled in method is being used, check out the *BasicCollisionDetection* method we were introduced to in the previous tutorial. If the collision is detected, we pass the colliding objects on to the new *ImpulseCollisionDetection* method, which will add the impulses to instantaneously change our linear and angular velocity such that the objects will move apart.

```
   if (CollisionDetection::ObjectIntersection(*i, *j, info)) {
      ImpulseResolveCollision(*info.a, *info.b, info.point);
      info.framesLeft = numCollisionFrames;
      allCollisions.insert(info);
   }
```

PhysicsSystem::BasicCollisionDetection method changes

This method also adds the collision pair into an **stl::set**, giving us a unique list of intersections for that particular game frame. Why does it do this? At the end of every frame, the *UpdateCollisionList* method is also called:

```
void PhysicsSystem::UpdateCollisionList() {
   for (std::set<CollisionDetection::CollisionInfo>::iterator
        i = allCollisions.begin(); i != allCollisions.end(); ) {
      if ((*i).framesLeft == numCollisionFrames) {
         i->a->OnCollisionBegin(i->b);
         i->b->OnCollisionBegin(i->a);
      }
      (*i).framesLeft = (*i).framesLeft - 1;
      if ((*i).framesLeft < 0) {
         i->a->OnCollisionEnd(i->b);
         i->b->OnCollisionEnd(i->a);
         i = allCollisions.erase(i);
      }
      else {
         ++i;
      }
   }
}
```

PhysicsSystem::BasicCollisionDetection method changes

The first frame a collision pair is added to the list, the respective **GameObjects** have their *OnCollisionBegin* method called, otherwise a counter is reduced inside the **CollisionInfo** struct - if this becomes less than zero, we assume the objects are no longer colliding, and remove the collision, calling *OnCollisionEnd* on the objects, so that more gameplay interactions can be coded (perhaps the player stops losing health when they leave a lava pool). This allows the 'gameplay' function calls to

be bunched up at the end of a frame, rather than being called in the middle of our physics engine update loop, where it could result in cache usage problems for our code.

Conclusion

If you now create a gameworld using one of the provided functions in the main file, you should now be able to knock objects about using raycasts, or by applying gravity so that they hit the floor, where they should hopefully bounce back up in the air and come to rest, rather than just keep going as they have done in the past.

In this tutorial, we examined methods of resolving the collisions between our objects in the world. We've seen how linear and angular impulses help move colliding objects apart, and how friction and elasticity change collision response. In the next tutorial, we're going to take a look at ways in which collision detection can be made more efficient via the use of *broad phase* and *narrow phase* collision checks.

Further Work

1) In the code for determining collision resolution, we currently have a hard coded coefficient of restitution. Try adding a new variable to the **PhysicsObject** class which stores the elasticity of each object, and then multiply them together into the *cRestitution* variable of the *ImpulseResolveCollision* method. You should then be able to define rubber balls (a high 'elasticity' variable) and steel balls (a low elasticitiy), and see how each responds when falling to the floor from gavity.

2) The current *ImpulseResolveCollision* uses a combination of the projection and impulse methods to maintain the consistency of the objects in the scene. This tutorial also discussed the ideal of using a spring calculation to change the accelerations of the objects, rather than position or velocity. Try making a *ResolveSpringCollision* method which uses Hooke's spring calculations to move the objects - remember that a spring has a resting length, and a spring coefficient to indicate how quickly the spring should 'snap' back to its resting length when stretched or compressed.