Attention:

Chris Hecker
definition six, incorporated
checker@d6.com
http://www.d6.com/users/checker

PS. The email address at the end of the article is incorrect. Please use checker@d6.com for any correspondence.

Physics, Part 3: Collision Response

Chris Hecker

Once a collision has occured between objects, careful modeling of the physics involved can impart realistic velocities and rotations to the objects.

Ask anyone who's experienced it before, and they'll tell you not to get in a car with me when I'm driving. For some reason, cars and I just don't get along very well. Or maybe I should say the front end of my car gets along very well indeed with the rear ends — and various additional parts — of other cars.

My driving skills notwithstanding, the topic for today is not how to avoid collisions (a topic about which I'm clearly not qualified to write), but rather "collision response" — what to do once we already know there is a collision.

You can probably guess that in the context of our series on game physics, the term "collision response" doesn't refer to calling an ambulance (in contrast with the context of my daily commute). The term refers to the second half of the collision process in a physical simulator, the first half of which is "collision detection." While in the real world, the sound of smashing glass is all the collision detection we need, the same is not true of our simulator, where we need code to explicitly check our geometry for collisions. Collision detection itself is worth a series of columns. Still, it's much more a geometric problem than a physical one, so for this column, we're going to assume you already have a way to detect collisions (we might return to the collision detection problem in a later column). The physics simulator requires certain information from the collision detector; we'll identify this information as we develop the collision response formulas and summarize the requirements at the end of the column.

Once we've detected a collision, the fun physics math starts, as we try to decide which directions the objects move in response to the impact. While we're going to restrict our scope to collisions between rigid bodies (so we won't be able to model all the crumpling and buckling that goes on when I run into an unsuspecting motorist), we'll still do better than you've probably seen before. Most current games do simple vector reflections, or maybe even take the objects' masses into account. However, in keeping with our goal for this series, we're going to do more accurate (and interesting) collision response. Our objects will spin and tumble as they collide, with heavy objects tossing lighter objects aside, imparting rotation to each other when they hit off-center. So, insurance premiums be damned: Full speed ahead!

Impulsive Behavior

To begin understanding the collision process, let's imagine we have two objects, labelled A and B, that are about to collide at a point P. Coincidentally, Figure 1 shows these very objects. There's actually a point P on both objects, so I've labeled the vector from the center of mass of object A to its point P as $\mathbf{r}^{AP}$, and likewise with $\mathbf{r}^{BP}$ for B. Let's also denote the velocities of the Ps as $\mathbf{v}^{AP}$ and $\mathbf{v}^{BP}$. A moment's thought convinces us that even though the Ps will be in the same exact position at the instant of collision (or there wouldn't be a collision at P), their velocities at that instant can be quite different — if one object is stationary, for example. Given the velocities of the Ps, we can define their relative velocity as $\mathbf{v}^{AB}$.

$$\mathbf{v}^{AB} = \mathbf{v}^{AP} - \mathbf{v}^{BP} \quad \text{(Eq. 1)}$$

More importantly, if our collision detector supplies us with a "normal vector" for the collision (denoted by **n**, and pointing toward body A by convention), we can define the "relative normal velocity" as the component of the relative velocity in the direction of the collision normal.

$$\mathbf{v}^{AB} \cdot \mathbf{n} = \left(\mathbf{v}^{AP} - \mathbf{v}^{BP}\right) \cdot \mathbf{n} \quad \text{(Eq. 2)}$$

Choosing a normal vector can be tricky, as we'll discuss below. But in the case of a vertex/edge collision — as in Figure 1 — it's pretty obvious that the normal should be perpendicular to the edge. Eq. 2 allows us to define the criterion for a collision:

A collision occurs when a point on one body touches a point on another body with a negative relative normal velocity.

This statement says Eq. 2 must be negative at the contact point or there's no collision. Consider the following three cases: If Eq. 2 is greater than 0, then the points are leaving each other, and we can ignore them. If it's equal to 0, the points are neither colliding nor separating — a situation called contact — and we'll have to deal with that problem in a future column. Finally, if Eq. 2 is less than 0, then the points are smashing into each other, and we need to do something to stop them from penetrating. That something is the collision response.

The obvious thing to do for collision response is to apply a force to both objects, but that doesn't actually do the job for rigid bodies. A force won't stop the bodies from interpenetrating because a force can't instantaneously change a velocity. That is, a force takes time to change a velocity — it can only do so via integration over time, as we learned in previous columns. Yet our objects are already touching, so we don't have any extra time to allow the force to do its work and counteract the negative relative normal velocity. We must change their velocities immediately or our objects will move inside each other. How can we affect this discontinous velocity change?

Think about the physics we've learned so far. Nowhere did velocities, either linear or angular, change instantly. Both are changed only by forces and torques through integration, which by definition means the velocity changes are continuous. In the case of a rigid body collision, however, we must change the velocities instantaneously. That calls for a new quantity: the "impulse."

We shouldn't feel bad about introducing yet another quantity at this point. After all, it was our idealization of inpenetrable rigid bodies that got us into this discontinuous velocity mess in the first place; it should come as no surprise that we have to idealize a little more to get ourselves out of it.

In a real-world collision, a lot of complicated atomic things happen that we can't hope to simulate directly. Thus, in the same way that we're approximating real-world objects with rigid bodies, we need to approximate the real-world collision process with an idealized model. Impulses are part of this model.

An impulse can change velocities directly, without waiting — the way a force must — for integration to do it. You can think of an impulse as a really huge force integrated over a really short period of time. The force is so large and the amount of time so small that we're no longer dealing with an almost infinite force over an infinitesimal period of time, but with a perfectly finite impulse. And, as force changes the momentum over time (remember $\mathbf{F} = \dot{\mathbf{p}}$), our impulse changes the momentum instantaneously, which in turn changes our velocity (by the definition of momentum as mass times velocity). We can calculate and apply impulses at the point and instant of collision, and these impulses will change the bodies' velocities and prevent them from interpenetrating.

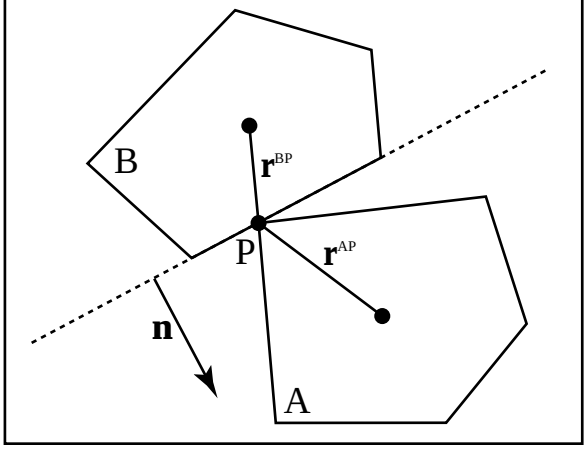
But *how* do we calculate the impulses to apply? This is the central problem of collision response. There are many ways to calculate the impulse's magnitude and direction, depending on how realistic you want to be. In the interest of space, we're going to go with a relatively simple model, but one that will still give us the interesting angular collision behavior we want. Later in the series, when we're more comfortable with the mathematics, we might try a more complex approximation.

The collision model we'll use is called "Newton's Law of Restitution for Instantaneous Collisions with No Friction." The easiest part of this model to understand is the "instantaneous" part. The model assumes the collision process takes no time. Since "no time" is a very small amount of time, all of our regular noncollision forces go away during the collision, and only the collision impulses are calculated. Thus, noncollision forces such as gravity are not taken into account during the collision, although they're in effect as usual before and after the collision.

Newton's Law of Restitution introduces yet another new quantity, the "coefficient of restitution" (usually denoted by an e or an ε, lowercase epsilon). The coefficient of restitution models the complicated compression and restitution of impacting bodies with a single scalar, which relates the contact point's incoming and outgoing relative normal velocities.

$$\mathbf{v}_2^{AB} \cdot \mathbf{n} = -e\,\mathbf{v}_1^{AB} \cdot \mathbf{n} \quad \text{(Eq. 3)}$$

Figure 1. Objects A and B colliding.



Eq. 3 uses a subscripted 1 and 2 to indicate the incoming and outgoing velocities, respectively. The coefficient of restitution *e* is a scalar that tells us how much of the incoming energy is dissipated during the collision. It can range from a totally elastic collision at *e*=1 (a superball), to a totally plastic collision at *e*=0 (a lump of clay landing on the floor).

Our collision model makes the final simplifying assumption that there is no friction at the point of collision. Thus, the impulse generated by the collision is entirely in the normal direction **n** (there's no tangential impulse at all). We can express the impulse with a single scalar *j* times the normal, giving us *j***n**. Newton's Third Law of equal and opposite forces says that the impulse felt by A is *j***n**, while the impulse felt by B is simply –*j***n**, the equal and opposite impulse. Now we're ready to derive the collision response equations.

Hit Me

For starters, we'll derive the collision response equations for objects that cannot rotate, then we'll go all the way and calculate the angular impact equations, as well. This is going to get a bit hairy, so you should probably get a piece of paper. The first equations we write relate the incoming and outgoing Center of Mass (CM) velocities under the influence of the (currently unknown) impulse.

$$\mathbf{v}_2^A = \mathbf{v}_1^A + \frac{j}{M^A}\mathbf{n} \qquad \text{(Eq. 4a)}$$

$$\mathbf{v}_2^B = \mathbf{v}_1^B - \frac{j}{M^B}\mathbf{n} \qquad \text{(Eq. 4b)}$$

I was able to write Eqs. 4a and 4b by keeping in mind that the impulse is a change in momentum, and I divided through by each object's mass to convert from a momentum equation to one in terms of velocity. Since the objects can't rotate yet, the velocities of the CMs ($\mathbf{v}^A$ and $\mathbf{v}^B$) are the velocities of all the points on the respective bodies; we can replace $\mathbf{v}^{AP}$ with $\mathbf{v}^A$ in Eq. 1 and make a similar exchange for B. Next, we use Eq. 3 to relate the incoming and outgoing relative velocities with the coefficient of restitution, and substitute in Eq. 1 for the definition of relative velocity. Substituting in Eqs. 4a and 4b and distributing the dot product, we get

$$\left(\mathbf{v}_2^A - \mathbf{v}_2^B\right)\cdot\mathbf{n} = -e\left(\mathbf{v}_1^A - \mathbf{v}_1^B\right)\cdot\mathbf{n}$$

$$\mathbf{v}_1^A\cdot\mathbf{n} + \frac{j}{M^A}\mathbf{n}\cdot\mathbf{n} - \mathbf{v}_1^B\cdot\mathbf{n} + \frac{j}{M^B}\mathbf{n}\cdot\mathbf{n} = -e\mathbf{v}_1^{AB}\cdot\mathbf{n} \qquad \text{(Eq. 5)}$$

We can simplify Eq. 5 by noting that the **v** terms on the left-hand side make up the relative normal velocity from Eq. 2 (modified by our assumption that the object can't rotate). We then solve for the scalar *j* and find (notice all the terms on the right-hand side are known at the time of collision)

$$j = \frac{-(1+e)\mathbf{v}_1^{AB}\cdot\mathbf{n}}{\mathbf{n}\cdot\mathbf{n}\left(\frac{1}{M^A} + \frac{1}{M^B}\right)} \qquad \text{(Eq. 6)}$$

Now that we know the impulse magnitude, we can plug it back into Eqs. 4a and 4b to find the new linear velocities of our objects. The collision is resolved!

Let's note a few things about Eqs. 4 and 6. First, you should notice that **n** doesn't have to be a unit-length vector for the collision response equations to work; the various dot products will cancel out any nonunit magnitude for **n** without forcing you to explicitly normalize it (thus avoiding normalization's accompanying square root). Of course, if you know **n** is unit length, you can avoid some multiplies in the denominator of Eq. 6.

The second thing to notice is that these same equations can handle a moving rigid body colliding with another rigid body that is supposed to stay fixed, such as a building or the ground. To see this, look at what happens when the mass of one of the objects increases: the effect of the impulse on that object decreases. Take this to the limit of infinite mass, and all the mass reciprocals for that object go to 0. Eq. 6 no longer contains the object's mass, and it degenerates into the equation for collision with a fixed object. Actually, the infinitely massive object doesn't have to be fixed, as its velocity is still present in Eq. 6. If it is moving, however, it will brush aside any dynamically simulated object and not feel so much as a nudge (such an object is called kinematically driven, since it's ignoring the dynamic quantities of mass, force, and impulse).

Finally, if you set A's mass to 1, set B's mass to infinity and its velocity to 0, make the coefficient of restitution 1, and make **n** unit length, you might recognize the equation to reflect a vector ($\mathbf{v}^A$) about a normal.

Spin Out

Now that we're warmed up, we can derive the complete 2D collision response equations, including the terms for angular velocity. To do this, we'll need to use the equation we learned in the last column for calculating the velocity of an arbitrary point on a rotating and translating rigid body.

$$\mathbf{v}_2^{AP} = \mathbf{v}_2^A + \omega_2^A\mathbf{r}_\perp^{AP} \qquad \text{(Eq. 7)}$$

I've written Eq. 7 for the postcollision velocities using the subscript 2, but it holds for the precollision velocities as well if you replace the 2s with 1s.

Next, in the same way we wrote Eqs. 4a and 4b for the change in linear velocity under the influence of an impulse, we can write equations for the changes in both linear and angular velocities when the impulse is applied. Here, I've written the equations for body A:

$$\mathbf{v}_2^A = \mathbf{v}_1^A + \frac{j}{M^A}\mathbf{n} \qquad \text{(Eq. 8a)}$$

$$\omega_2^A = \omega_1^A + \frac{\mathbf{r}_\perp^{AP}\cdot j\mathbf{n}}{I^A} \qquad \text{(Eq. 8b)}$$

Eq. 8a should be familiar from our linear collision example; it matches Eq. 4a. Eq. 8b, on the other hand, is the result of applying the impulse $j\mathbf{n}$ at point P on body A. The last term on the right translates the linear impulse into an angular impulse in exactly the same way that we translated linear force into torque in the last column: using a perp-dot product to the point of application. Since impulse will change the angular momentum, I've divided through by the moment of inertia at the CM to convert Eq. 8b into an equation in the angular velocities.

Eqs. 8a and 8b together show how the collision impulse will affect body A's precollision velocities. The equations for body B are exactly the same when j is replaced by $-j$, since the impulse is equal and opposite. Our remaining task is to solve for j, and then plug it into Eqs. 8a and 8b (and the counterparts for B) to resolve the collision.

Solving for j involves the same sort of algebra as in the previous example. First, start with Eq. 3, replace $\mathbf{v}^{AB}$ with the definition in Eq. 1, and substitute Eq. 7 for $\mathbf{v}^{AP}$ and its twin for $\mathbf{v}^{BP}$. Then, for the unknown postcollision linear and angular velocities, substitute in Eqs. 8a and 8b and their B versions. Gather the terms, being sure to recognize the expression for the precollision relative normal velocity (in the same way we brought it into the numerator in Eq. 6), and solve for j. We end up with

$$j = \frac{-(1+e)\,\mathbf{v}_1^{AB} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}\left(\frac{1}{M^A} + \frac{1}{M^B}\right) + \frac{\left(\mathbf{r}_\perp^{AP} \cdot \mathbf{n}\right)^2}{I^A} + \frac{\left(\mathbf{r}_\perp^{BP} \cdot \mathbf{n}\right)^2}{I^B}}$$

(Eq. 9)

Once we've calculated j, we plug it into Eqs. 8a and 8b (don't forget to negate j and plug it into the equivalent equations for B), and we're done with the collision response. The colliding bodies go flying apart, complete with the correct spin based on their incoming velocities and masses.

A Little Touch Up

Now that you know the collision response equations, let's see how they fit into our overall simulation loop. Listing 1 shows the pseudocode for the simulation loop that supports collision detection and response from the sample application. I changed last issue's step-by-step algorithm to pseudocode because the loop got a bit more complex when it was extended to handle collisions.

The root of this new complexity is calculating the "exact" time of collision. Notice we integrate forward by a full time step at first, and if there's interpenetration at the new configuration, we subdivide the time interval and try again. The algorithm amounts to doing a binary search of the time step looking for the time of collision. This is not necessarily the most efficient way to find the collision time, since we throw away all of our previous integration work, but it's very simple and robust. Other solutions to this problem include using the previous integration parameters to help estimate when the collision occured, trying to predict ahead of time where the collision will occur, or even trying to use the interpenetrating coordinates and hoping it doesn't look too bad. Also, this discrete collision routine doesn't catch "tunneling," where fast moving objects can move completely through other objects in a single integration step.

Once a noninterpenetrating configuration is found, we resolve the collision — if present — and update the configuration. Then we loop back up to complete the time step and finally draw the objects.

I've glossed over a few things in this presentation, so let's take the remaining space to get out the Bond-O and fill in some of the holes….

It should be clear from the collision response equations that we need to know four pieces of data about a collision: the time of the collision, the objects participating in the collision, the colliding points on those objects, and the collision normal. Each of these parameters has some subtleties, and we'll go into each in turn.

You'll notice I quoted the word "exact" a few paragraphs back when refering to the first required piece of data: the collision time. The reason is that there's really no such thing as the exact collision time when you're working numerically on a computer. We're forced to use a tolerance value for collision detection, within which we agree to say we're colliding (rather than interpenetrating or not touching). The sample code shows this technique.

The next bit of data — the collection of colliding objects — seems obvious, but note that our current algorithm can only handle a single collision between two bodies. A similar limitation holds for the third parameter, the collision points. It's easy to see that in a 2D collision between convex polygons, you can get an edge/edge collision, which means the collision "manifold" — the space that represents the parts of the objects that are touching — is no longer a point, but a line segment. You can get away with just using the vertices of the line segment for this kind of collision, but even that is beyond the powers of our current collision response routine. It can only handle a single collision point, not multiple simultaneous collision points. Simultaneous collisions are much harder and will have to wait for another time. Things get even worse in 3D, where you can get point, edge, and face collisions with convex polyhedrons, and collision detection and response become a nightmare when you get into curved surfaces. Anyway, the sample application's collision detector currently returns only a single collision point, and although we don't get flat-edged bounces (it always looks like one point hits first), it still looks pretty good.

The collision normal is the final place where ambiguities arise. In 2D, on an edge/edge or a vertex/edge collision, the normal vector is easily obtainable as a vector perpendicular to the edge. However, on a vertex/vertex collision, you need to pick a sensible vector to use for the collision normal. The sample application avoids this problem by treating vertex/vertex collisions as vertex/edge collisions, but that can lead to unrealistic behavior.

The references for this material would just about fill the space of an entire column, so once again, I'm going to put them on my website at *http://ourworld.compuserve.com/homepages/checker*. The derivations I've used here are similar to David Baraff's equations in his SIGGRAPH tutorial on physically based modeling (it's in my references). Like most results in math and physics, there are a bunch of ways of getting to the same equations, including derivations based on the laws of conservation of energy and momentum, and derivations based on things called "generalized coordinates." If you study this stuff seriously, you'll want to work out the equations in a lot of different ways to make sure you understand them. The more practice you get, the better mathematician and physicist you'll become. Now, if only the same principle held for my driving… ■

Chris Hecker's collision response is usually to write a large check to some auto-body shop. Donations are accepted at checker@bix.com.

Listing 1. The Simulation Loop Pseudocode.

```
setup initial conditions

while(simulating) {
   DeltaTime = CurrentTime - LastTime

   while(LastTime < CurrentTime) {
      calculate all forces and torques
      compute linear and angular accelerations
      integrate accelerations and velocities over DeltaTime

      if(objects are interpenetrating) {
         subdivide DeltaTime
      } else {
         if(objects are colliding) {
            resolve collisions using Eqs. 8 and 9
         }

         LastTime = LastTime + DeltaTime
         DeltaTime = CurrentTime - LastTime
         update positions and velocities
      }
   }

   draw objects in current positions
}
```