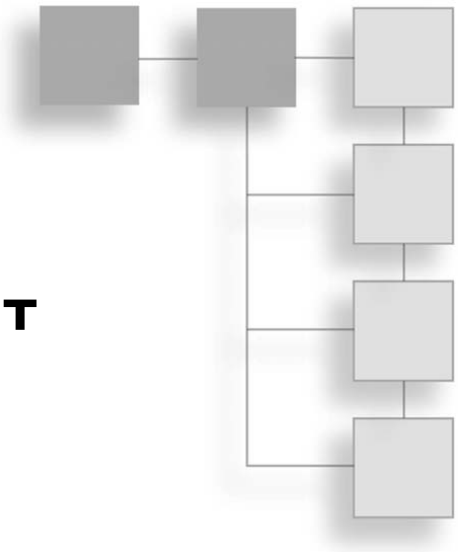
CHAPTER 8



Collisions of Point Particles

So far, the point masses created in the previous chapter move around under any force you might care to give them. We could keep adding forces, making the model more realistic (and complicated!), but we would still be lacking something fundamental because the balls don't interact. In this chapter, we'll do something about that.

However, before we dive into collisions, it's important to understand that simulating collisions is really a combination of two distinct problems. The first is collision detection. To be able to react to collisions, games have to know that a collision has occurred. It sounds simple, but it's not. Collision detection is complex.

In any case, collision detection is a programming problem, not a physics problem. It's definitely not the focus of this book. Therefore, I'll give an overview of collision detection and move on to the second problem associated with collisions: collision response.

Collision response *is* a physics problem. When we talk about collision response, we're speaking of how objects react when they collide. That's determined strictly by physics. After the overview of collision detection, we'll look at collision response for point masses.

Collision Detection

Computer Science departments are full of professors and grad students trying to work out newer, faster, better collision algorithms. Consequently, there's a tremendous bulk of literature on the subject that you could spend the rest of your life sifting through. In computer games, simple solutions are usually the best, so we'll look at the most common methods of collision detection.

Bounding Spheres

This is the easiest type of collision detection. Fit a sphere around your object, and shrink the radius until it fits as tightly as possible, as in Figure 8.1. If anything comes within this sphere, you assume that the object has collided with it.

For example, you can use this method when testing for collisions between objects and a ground plane. Here's the algorithm:

1. First check to see if the object is above the plane. If it is, proceed to step 2.
2. Test to determine whether the distance from the center of the point mass to the ground is less than the radius of the bounding sphere. If it is, proceed to step 3. Otherwise, skip step 3 and repeat this algorithm for the next object in the scene.
3. The object has collided with the ground. React to the collision.

This method works just as well with any planar surface, such as walls. Figure 8.2 applies this algorithm to collisions with walls.

The objects in question do not have to be balls. They can be any shape that you can fit into a sphere.

Your game applies a variation of this algorithm to collisions with other point masses. Two objects are said to have collided if their bounding spheres overlap, as in Figure 8.3.

To detect whether a collision has taken place, you need the radii of the two bounding spheres and the position vectors of the objects. You then compare the distance between the bounding spheres of two objects to the sum of their radii, as in Figure 8.4. If the separation between the two objects is less than the sum of the radii, a collision has taken place.

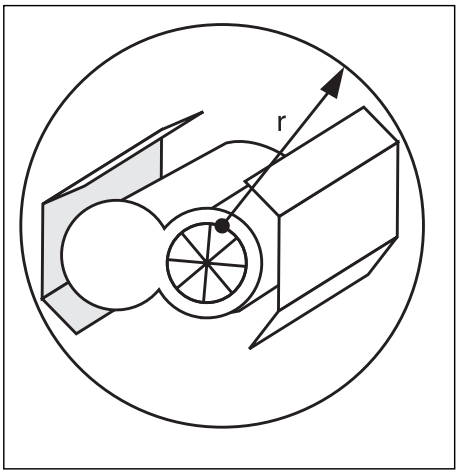


Figure 8.1
An object with a bounding sphere.

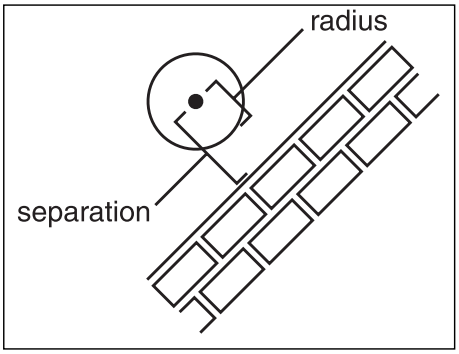


Figure 8.2
Detecting a ball colliding with a wall.

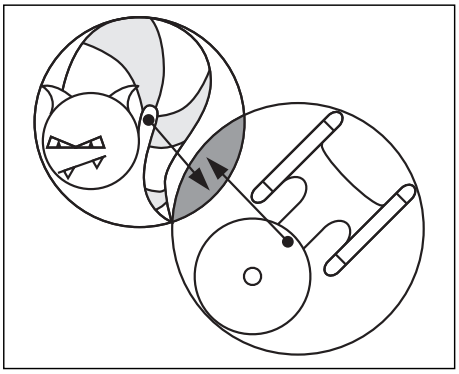


Figure 8.3
Two objects with bounding spheres colliding.

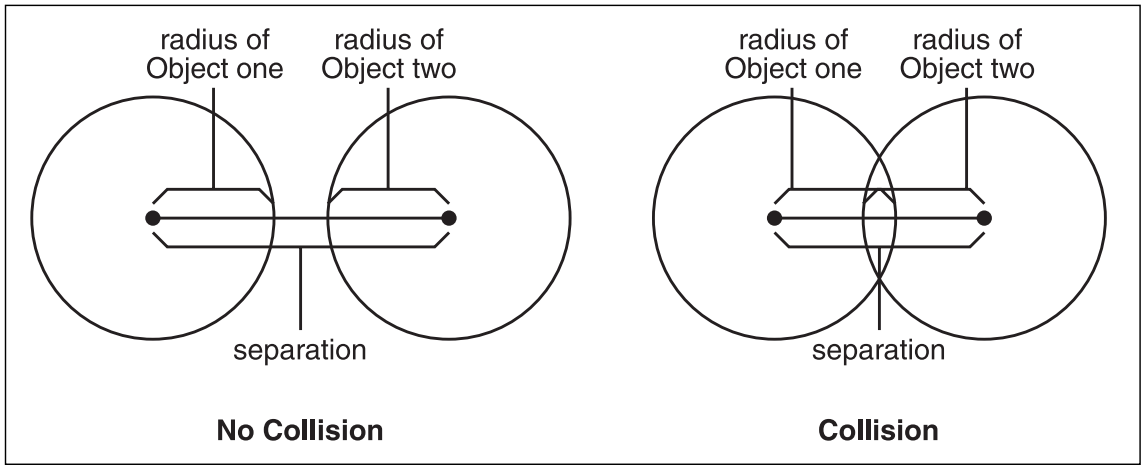


Figure 8.4
Comparing the separation between two objects to the sum of their radii.

Using vectors makes these calculations easy. Suppose that you're creating a pool game, and you want to write a function for detecting whether two balls have collided. If you have the positions of the centers of the balls as vectors, you can compute the distance between them by subtracting the vectors. Figure 8.5 illustrates this.

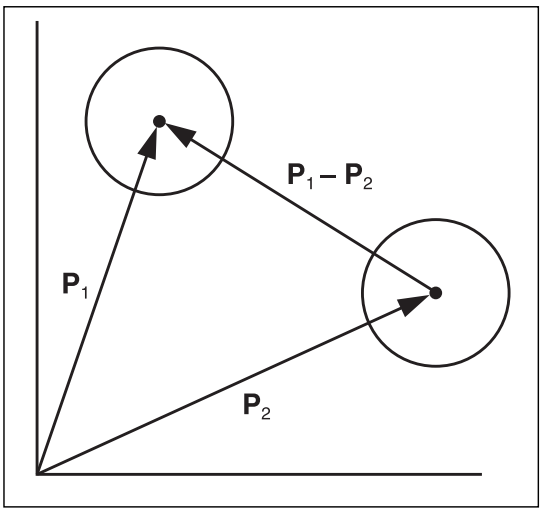


Figure 8.5
Computing the separation vector between two objects.

Subtracting P_2 from P_1 in Figure 8.5 gives a vector specifying the distance between the two centers of mass. Now your program can find the magnitude of the distance vector. Compare it to the sum of the radii of the bounding spheres. If the distance is less than the sum of the radii, the balls have collided. If not, there is no collision.

To make this a little clearer, look at the following code fragment. It takes two `d3d_point_mass` objects—`ball1` and `ball2`—calculates the separation vector, and compares the length of the separation vector with the sum of the radii.

```
// Calculate the vector between the spheres.
vector_3d distance = ball1. Location() - ball2.Location();
// Compute the actual separation distance.
scalar magnitude = distance.Norm();
```

```
    // Calculate the distance at which a collision occurs.
    scalar minDistance = (ball1.Radius() + ball2.Radius());
    if (magnitude < minDistance)
    {
        // A collision occurred, so we have to handle it.
    }
```

This algorithm works great, except that it's incredibly slow! Remember that you're going to have to calculate this for every pair of particles in the region. It's the square root in the norm calculation that kills you: It takes about 70 times as long as multiplying floats. Here's a look at that `vector_3d::Norm()` method I used to calculate the norm:

```
inline scalar vector_3d::Norm(void)
{
    return(sqrtf(x*x + y*y + z*z));
}
```

You could speed this up in several ways, including taking fast numerical approximations of the square root. Clever readers might have noticed an even faster optimization: not taking the square root at all!

If I have two numbers x and y, if x is greater than y, then x^2 had better be greater than y^2. The same concept holds here: Instead of comparing the separation distance to the sum of the radii, I'll compare the *square* of the separation with the *square* of the sum of the radii.

Here's some code that uses this optimization:

```
    // Calculate the vector between the spheres.
    vector_3d distance = ball1. Location() - ball2.Location();
    // Compute the actual separation distance.
    scalar magnitude = distance. NormSquared();
    // Calculate the distance at which a collision occurs.
    scalar minDistance = (ball1.Radius() + ball2.Radius());
    // Square the minimum distance.
    minDistance*=minDistance;
    if (magnitude < minDistance)
    {
        // A collision occurred, so we have to handle it.
    }
```

In this case, we used the `vector_3d::NormSquared()` method. It's exactly like the `vector_3d::Norm` function, except that it doesn't compute that square root.

```
inline scalar vector_3d::NormSquared(void)
{
    return(x*x + y*y + z*z);
}
```

We do have to calculate an extra floating-point multiplication this way, when we square the sum of the radii. However, because that calculation is lightning fast compared to finding the square root, it's definitely worth it.

Using bounding spheres is often the best way to detect collisions. It's simple, it's fast, and it gives great results for many applications. Even when you want to use more sophisticated collision detection algorithms, it's usually worth it to try a bounding sphere first to see whether it's even worth the CPU cycles to try the more complicated method.

Bounding Cylinders

Instead of using bounding spheres, you can try some more complicated shapes. Bounding cylinders work great for games in which most objects are oriented about the same way toward some surface. Good examples are *Doom*-type games where most characters manage to stay standing upright even under a barrage of bullets. Every character stays oriented in the same direction with respect to the floor. Figure 8.6 shows such a character bounded by a cylinder.

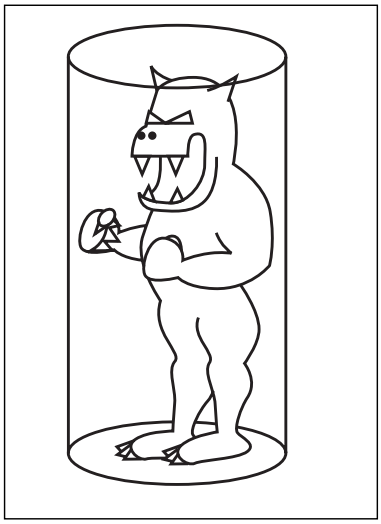


Figure 8.6
An object in a bounding cylinder.

To detect collisions, you have to check the top and bottom of the cylinder, as well as its radius. The first step to putting this in code is to add the dimensions to the `d3d_point_mass` class. Figure 8.7 shows those dimensions in relation to the position of the object.

Testing for collision between bounding cylinders is a two-step process. In the first step, you assume that the bounding cylinders are always oriented as shown in Figures 8.6 and 8.7. That enables you to reduce this to a 2-D problem rather than a 3-D problem. The radius of the bounding cylinder is always in the xz plane. The first step is to treat the objects as circles in the xz plane. Find the distance between the centers of the circles. If the distance is greater than the sum of the radii, there is no collision. However, if the distance is less, there might be a collision.

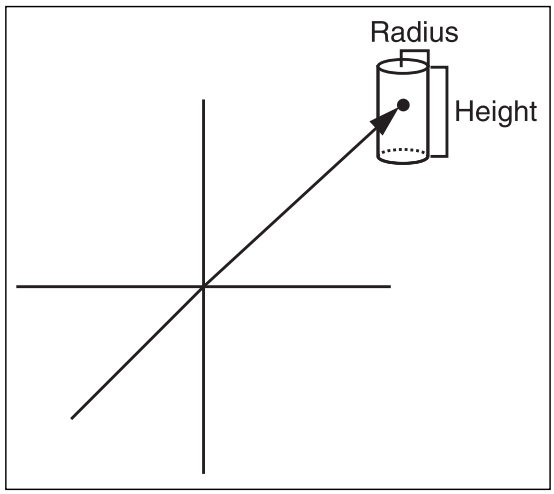


Figure 8.7
Dimensions of the bounding cylinder.

The next step to determining whether there has been a collision is to check whether the vertical distances overlap, as in Figure 8.8.

What matters is whether the top of one cylinder is vertically between the top and bottom of the other cylinder. If it is, the program must handle the collision. If not, there is no collision even though the radii overlap. This occurs, for instance, when the characters in a multiplayer game are on different floors of a building and one walks over the top of the other.

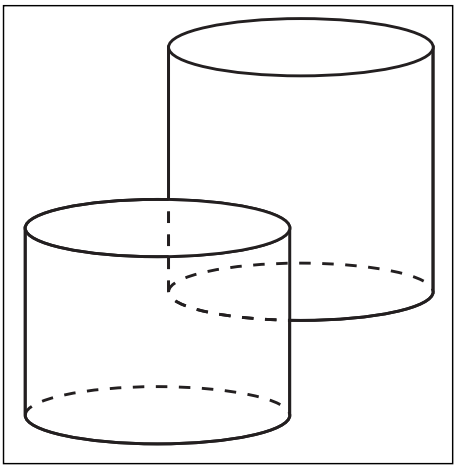


Figure 8.8
Checking on the overlap of bounding cylinders in the vertical direction.

Bounding Boxes

Now that we've done bounding spheres and bounding cylinders, it probably won't surprise you that we can also detect collisions using bounding boxes, as shown in Figure 8.9. You can use bounding boxes in two and three dimensions.

When you use a bounding sphere or a bounding cylinder, you're assuming that the object has that shape, more or less. That might be just fine, but sometimes it can create problems. Look at Figure 8.10, which shows a 2-D polygon. Although a circular bounding area works, it encloses an awful lot that isn't part of the polygon. In this case, a bounding box works much better.

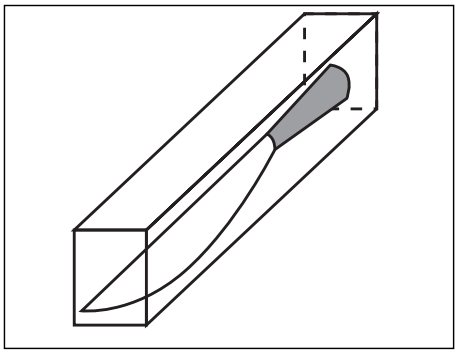


Figure 8.9
A bounding box.

Here's how you check whether two boxes have collided. Pick a vertex on one box. Then see if that vertex is inside the other bounding box. If it is, the boxes have collided, as in Figure 8.11. If it isn't, check the next vertex until you run out of vertices. There are eight vertices in a box.

Note that you'll have to check all the vertices for *both* boxes to avoid errors. In Figure 8.11, if you check only the vertices of the box on the left, you won't find a collision.

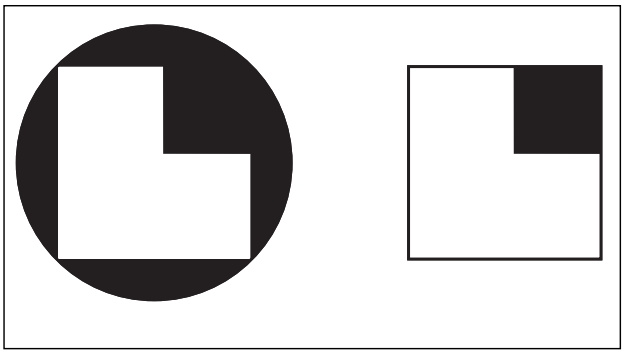


Figure 8.10
Choose the shape of your bounding region to fit the object.

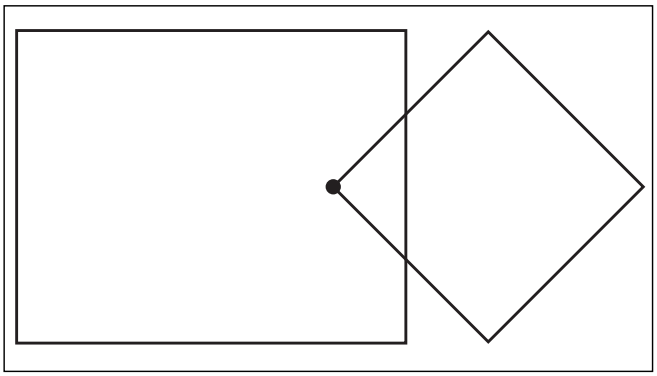


Figure 8.11
Two bounding boxes are colliding if the vertex of one is inside the other.

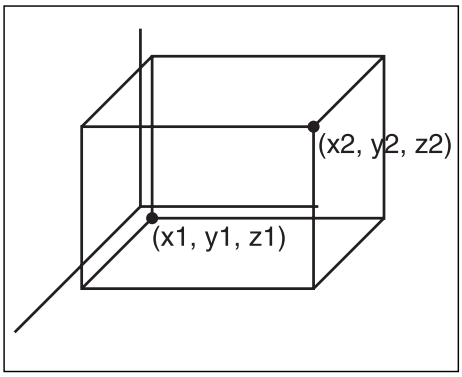


Figure 8.12
A box sitting flat in a Cartesian plane can be defined with two points.

You can define a box with two vertices in a Cartesian coordinate system, as shown in Figure 8.12, as long as the box is constrained to sit flat. I'll call these vertices (x1, y1, z1) and (x2, y2, z2). Checking whether a vertex (x, y, z) is in this box is pretty easy. It's much like the check you just did on the top and bottom of the cylinder, but in all three directions.

```
if( x >= x1 && x <= x2) // Contained on the x axis
    if( y >= y1 && y <= y2) // Contained on the y axis
        if( z >= z1 && z <= z2) // Contained on the z axis
            ( /* The Point is Contained */ )
```

Bounding boxes take quite a bit more CPU time to calculate than bounding spheres or cylinders, but they're often helpful. One of these three methods will serve most of the time.

note

A bounding box that is constrained to sit flat is called an axis-aligned bounding box (AABB). You'll see this term a lot in game programming literature.

Optimization with Spatial Partitioning

You can calculate the number of possible collisions, N_c, you have to deal with in each frame with this formula, where n is the number of objects:

$$N_c = \frac{n!}{2!(n-2)!}$$

For large n, this number is approximately equal to half the square of n. That is $N_c \approx n^2 / 2$.

Factorials

The ! sign is a *factorial*. This is a little hard to explain in words, so let me give you a formula and some examples.

$$n! = \prod_{k=1}^{n} k$$

The factorial of zero is defined to be one (0! = 1). Here are a few examples:

$$3! = 3 \times 2 \times 1 = 6$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$6!/4! = (6 \times 5 \times 4 \times 3 \times 2 \times 1)/(4 \times 3 \times 2 \times 1) = 6 \times 5 = 30$$

Table 8.1 lists the number of collisions you need to check for several values of n. You can see from the table that the number grows large quickly as you increase the objects. 10,000 objects wouldn't be ridiculous for a game once the bullets started flying. That's almost 50 million possible collisions! This makes collision algorithms a good place to optimize.

Table 8.1 Number of Possible Collisions

Number of Objects	Number of Possible Collisions
2	1
4	6
10	45
20	190
100	4950
1,000	499,500
10,000	49,995,000

Still, making faster collisions isn't going to solve the fundamental problem here. The only way to do that is to reduce the number of collisions you have to deal with.

One way to do this is called *spatial partitioning*. The idea is to divide the universe into cells, as in Figure 8.13. You only need to check for collisions in those regions near the player.

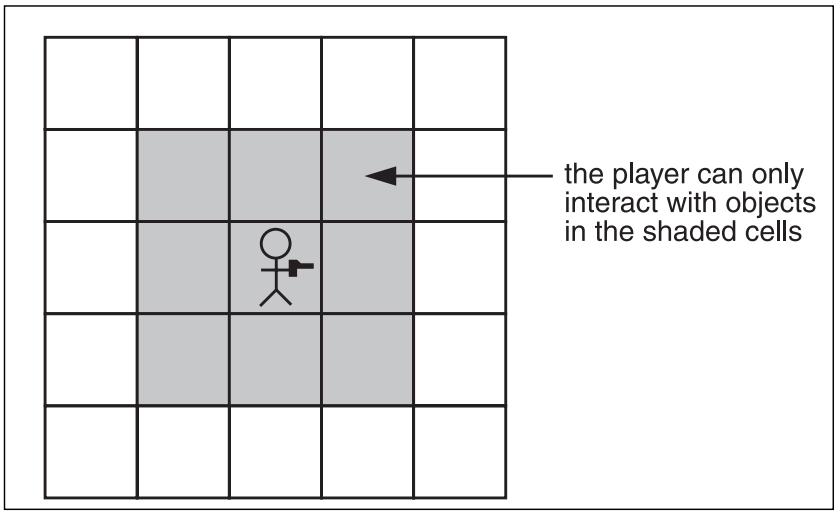


Figure 8.13
Spatial partitioning can speed up collision detection.

tip

You can also use spatial partition to clip graphics. Why calculate and render things the player can't see?

Let's say that you've divided your world into a 10×10×10 grid of cells. That's 1,000 cells. If there are 10,000 objects, each cell will average 10 objects. If the player can only interact with objects in the closest 3×3×3 block of cells, you only have to calculate collisions for these 27 cells. That's about 270 objects, or 270!/2!269! ≈ 36,500 possible collisions. 36,500 is quite an improvement over 50 million!

Collision Response

What happens when the objects actually do collide?

Massive confusion—that's what happens. Things squish together. Cracks form, and chips of material go hurtling away. Sound erupts, and sparks fly. The objects heat up, and the air gets more turbulent.

Can we hope to model all this?

No way.

Here's what we're going to do instead. Consider a one-dimensional collision between two objects, as in Figure 8.14. Two objects with masses m_1 and m_2 fly in with some velocities v_1 and v_2, stuff happens, and the two objects fly out with different velocities v_1' and v_2'. The question becomes this: Given the masses and incoming velocities, what are the outgoing velocities?

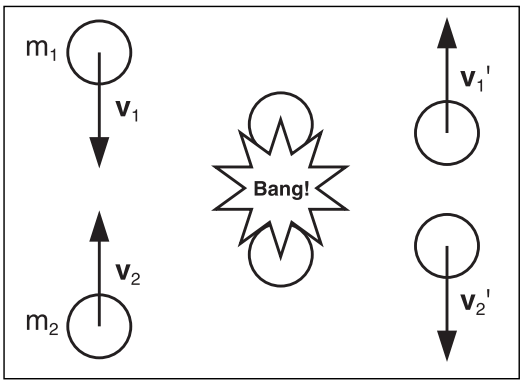


Figure 8.14
A collision between two objects.

We're going to assume that the collision (that is, all of this nasty, hard-to-model stuff in the middle) is short and ask ourselves if there are relationships between the velocities of the objects coming in and their velocities going out. It turns out that there are.

Conservation of Momentum

There's another way we can write Newton's second law. It looks like this:

$$\mathbf{F} = \frac{d\mathbf{p}}{dt}$$

where **F** is the force and **p** is the momentum. What is *momentum*? It's usually just defined in physics books as the product of the mass of an object times its velocity. However, a professor friend of mine says that momentum is inertia in motion. That's the best conceptual definition of momentum that I've ever encountered.

$$\mathbf{p} = m\mathbf{v}$$

Imagine that we have a system of particles, each with its own momentum, all in a big box in space. They're flying past each other, doing whatever, but we aren't applying forces to them from the outside. If we add up the momentum of all the particles, we have the momentum for the system. Now, using Newton's second law,

$$\frac{d\mathbf{p}}{dt} = 0$$

The only solution to this equation is that the momentum is constant. This leads us to a nice general principle that the momentum of this type of system must be constant for all time. That is, if the equation

$$\frac{\Delta \mathbf{p}}{\Delta t} = 0$$

is true for a system, then

$$\Delta \mathbf{p} = 0$$

The reason for this is that dividing by 0 is undefined, so Δt cannot be 0. Therefore, it must be Δ that's 0. This gives the equation:

$$\mathbf{p}' = \mathbf{p} + \Delta \mathbf{p}$$
$$\mathbf{p}' = \mathbf{p}$$

This is a mathematical way of stating the principle of the Conservation of Momentum. Formally, here's the way that the conservation of momentum works: In a system with no outside forces applied, the momentum is constant.

We can apply conservation of momentum to the collision problem. Because the momentum is constant, the sum of the momenta before and after the collision must be equal.

$$\mathbf{p}_1 + \mathbf{p}_2 = \mathbf{p}_1' + \mathbf{p}_2'$$

where $\mathbf{p_1}$ is the momentum of particle one before the collision, $\mathbf{p_1}'$ is the momentum of particle one after the collision, and similarly for the second particle. I'm assuming there aren't little bits flying off that could carry momentum away.

Plugging in $\mathbf{p} = m\mathbf{v}$ for each particle, we get this:

$$m_1\mathbf{v}_1 + m_2\mathbf{v}_2 = m_1\mathbf{v}_1' + m_2\mathbf{v}_2'$$

Because we're doing a one-dimensional collision, we can write this as follows:

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2'$$

This is really a great result. We have a nice relationship between the velocities before the collision and the velocities afterward that's independent of anything that happens during the actual collision.

Still, it's not enough to actually solve for the motion. It leaves us with only one equation, and we would like to be able to solve for both v_1' and v_2'.

Energy

To talk much more about collisions, I'm going to have to say a little something about energy and the related concept of work. If you keep pushing on something with a constant force while it moves over some distance, as in Figure 8.15, it moves faster and faster. This takes some effort. That effort is called *work*, and it's given by the dot product of force and distance.

$$W = \mathbf{F} \bullet \mathbf{x}$$

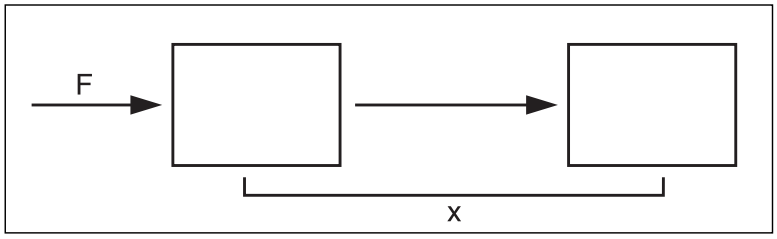


Figure 8.15
Applying a force over a distance.

Let's say that we're accelerating something from rest at a position $\mathbf{x} = \mathbf{0}$, at a constant acceleration to some velocity $\mathbf{v}$ in a time t. In this case, we can use the following equations:

$$\mathbf{x} = (\frac{1}{2})\mathbf{a}t^2$$
$$\mathbf{v} = \mathbf{a}t$$

We also have Newton's second law:

$$\mathbf{F} = m\mathbf{a}$$

Combining all these, we can solve for the work required to accelerate an object of mass m to a speed $\mathbf{v}$.

$$\begin{aligned} W &= \mathbf{F} \bullet \mathbf{x} \\ &= m\mathbf{a} \bullet \mathbf{x} \\ &= m\mathbf{a} \bullet \left(\frac{1}{2}\right)\mathbf{a}t^2 \\ &= \left(\frac{1}{2}\right)m(\mathbf{a} \bullet \mathbf{a})\, t^2 \\ &= \left(\frac{1}{2}\right)m(\mathbf{a}t \bullet \mathbf{a}t) \\ &= \left(\frac{1}{2}\right)m(\mathbf{v} \bullet \mathbf{v}) \\ &= \left(\frac{1}{2}\right)mv^2 \end{aligned}$$

Most of this is just simple substitution. It also uses the fact that a vector dotted with itself gives the square of the norm of the original vector.

warning

Don't forget that the norm of a vector **v** can be written in two ways: v or |**v**|.

This equation might look familiar to you if you've had a physics class. This quantity is called the *kinetic energy*. It's the energy of a body in motion. I'm going to write the kinetic energy as a capital K, like this:

$$K = \left(\frac{1}{2}\right)mv^2$$

Work can then be reinterpreted as the change in energy of a system due to forces on it. If there are no forces on a system, there is no work done, and the energy is constant. This principle is called the *Conservation of Energy*.

Elastic Collisions

Now let's get back to collisions. Recall that applying the conservation of momentum gave this equation:

$$m_1\mathbf{v}_1 + m_2\mathbf{v}_2 = m_1\mathbf{v}_1' + m_2\mathbf{v}_2'$$

In this equation, m_1 and m_2 are the masses, $\mathbf{v}_1$ and $\mathbf{v}_2$ are the velocities before the collisions, and $\mathbf{v}_1'$ and $\mathbf{v}_2'$ are the velocities afterward.

Conservation of energy says the sum of the energies before the collision is equal to the sum of the energies afterward. If I assume that all the energy from before the collision goes into the motion of the balls afterward, I can write this:

$$K_1 + K_2 = K_1' + K_2'$$

where the Ks are the kinetic energies of the objects before and after the collision.

If you plug in the definition for K, $K = (1/2)mv^2$, you get this:

$$\left(\frac{1}{2}\right)m_1v_1^2 + \left(\frac{1}{2}\right)m_2v_2^2 = \left(\frac{1}{2}\right)m_1v_1'^2 + \left(\frac{1}{2}\right)m_2v_2'^2$$

note

The preceding equation works in one, two, or three dimensions.

Now we have two equations and two unknowns, so we can solve for the final velocities. The algebra is easy because we're working with scalars, but it's a little tedious to print in a book, so I'm just going to give you the result:

$$v_1' = \frac{(m_1 - m_2)v_1}{m_1 + m_2} + \frac{2m_2 v_2}{m_1 + m_2}$$

$$v_2' = \frac{(m_2 - m_1)v_2}{m_1 + m_2} + \frac{2m_1 v_1}{m_1 + m_2}$$

Inelastic Collisions

Let's say that the two colliding objects are lumps of clay, as in Figure 8.16. Then, when we throw them together, they're going to stick into one massive blob with mass m_1+m_2.

With this setup, we can write the conservation of momentum like this:

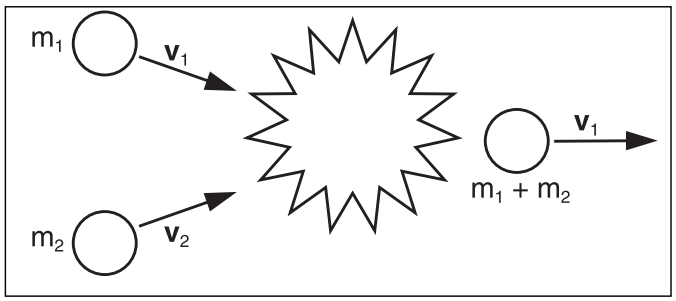


Figure 8.16
An inelastic collision.

$$m_1\mathbf{v}_1 + m_2\mathbf{v}_2 = (m_1+m_2)\mathbf{v}'$$

where $\mathbf{v}'$ is the velocity of the final blob. Rearrange to solve for the final velocity.

$$\mathbf{v}' = m_1\mathbf{v}_1 + \frac{m_2\mathbf{v}_2}{m_1 + m_2}$$

Nice! For an inelastic collision, you can completely solve the collision without talking about the energy at all.

note

Inelastic collisions do come up in games. When you lodge a bullet into an alien abdomen, bullet and alien are joined and go flying off in the same direction.

One thing might seem a little strange. If we got the elastic collision result just by assuming energy and momentum conservation, why do we get a different result for this inelastic collision?

The answer is that in an inelastic collision, not all the energy goes into the final motion of the objects. When the lumps of clay stuck together, they were deformed, and heat was generated inside them. Energy was still conserved, but we didn't track all of it.

The Coefficient of Restitution

Look again at the equations for an elastic collision, the conservation of energy and momentum:

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2'$$
$$\left(\frac{1}{2}\right) m_1 v_1^2 + \left(\frac{1}{2}\right) m_2 v_2^2 = \left(\frac{1}{2}\right) m_1 v_1'^2 + \left(\frac{1}{2}\right) m_2 v_2'^2$$

Let me rearrange these two equations a little:

$$m_1 (v_1 - v_1') = -m_2 (v_2 - v_2')$$
$$m_1 (v_1 - v_1')(v_1 + v_1') = -m_2 (v_2 - v_2')(v_2 + v_2')$$

If you divide the second of these by the first and do a little rearrangement, you get this:

$$\frac{-(v_1' - v_2')}{v_1 - v_2} = 1 \qquad \text{(elastic)}$$

That is, the difference in velocities before the event is equal to and opposite of the difference afterward.

What does this quantity look like for the inelastic collision? In the end, the two particles become one, so $v_1' = v_2'$; therefore, this quantity is 0.

$$\frac{-(v_1' - v_2')}{v_1 - v_2} = 0 \qquad \text{(inelastic)}$$

Collisions in the real world aren't perfectly elastic or inelastic. Most of them take up some energy in the jumble of the collision, but the objects don't remain stuck together. To describe these circumstances, we're going to define the *coefficient of restitution*, e.

$$e = \frac{-(v_1' - v_2')}{v_1 - v_2}$$

If e = 0, it's a completely inelastic collision; if e =1, it's perfectly elastic. We'll usually pick something in between, generally just by our intuition about how something should behave. For billiard balls, you'd pick something near 1; billiard balls don't deform because they don't heat much. For nerf balls, the coefficient of restitution needs to be something closer to 0—maybe 0.2.

note

You can get a fun little effect by setting the coefficient of restitution greater than 1. This system doesn't just conserve energy—every collision makes energy!

Collision Equations

Using the coefficient of restitution and momentum conservation, we can derive the final equations we'll use for calculating collisions in the physics model. Here are the two together:

$$e = \frac{-(v_1' - v_2')}{v_1 - v_2}$$

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2'$$

By rearrangement and substitution, you can solve for v_1' and v_2'.

$$v_1' = \frac{(m_1 - em_2)v_1 + (1 + e)m_2 v_2}{m_1 + m_2}$$

$$v_2' = \frac{(m_2 - em_1)v_2 + (1 + e)m_1 v_1}{m_1 + m_2}$$

This is exactly what we've been looking for: an equation that will describe any linear collision. Try plugging in e = 0 and e = 1 as a check to make sure you get the collision equations for inelastic and elastic collisions.

Point Particle Collisions in Two and Three Dimensions

This section ought to be short. Why? Because, in general, exact solution of collisions in two and three dimensions is too complex. Usually you don't have enough information to solve the two- and three-dimensional equations, except for inelastic collisions. In elastic collisions in two and three dimensions, you need to know additional details about the collision, such as the force law of the interaction between the bodies (for example, Newton's law of gravitation) or the detailed shape of the colliding bodies.

Collisions of Spheres

Because point particle collisions are generally too difficult, we're going to turn our attention to spheres. Suppose that we have two massive, uniform spheres undergoing a frictionless collision, as in Figure 8.17. They come in with velocities $\mathbf{v_1}$ and $\mathbf{v_2}$ and leave with velocities $\mathbf{v_1}'$ and $\mathbf{v_2}'$. Note that the point of contact has to be on a line connecting the centers of the spheres. This line is perpendicular to the surfaces of the spheres.

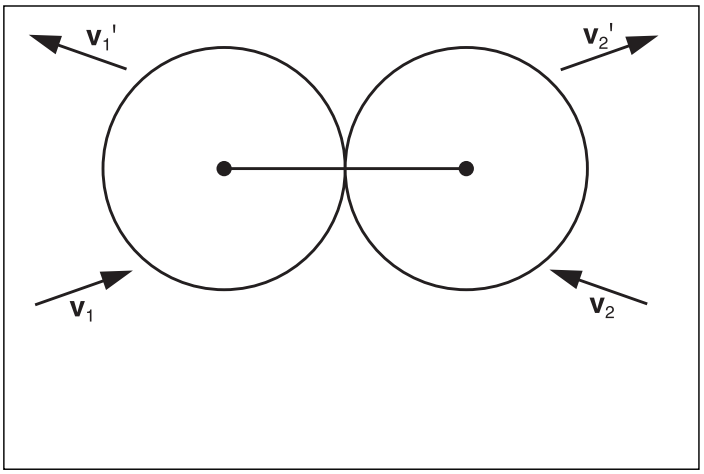


Figure 8.17
Two spheres colliding.

warning

The word *frictionless* was thrown in there because any friction on the surface causes a torque. If you don't know what a torque is, you'll find out in Chapter 9, "Rigid Body Dynamics."

Because this is the only point of contact, the line acts as a line of action for the two spheres. In other words, we can treat the collision problem as one dimensional along this line. The technical reason we can do this is that there is no change in momentum perpendicular to the line of action.

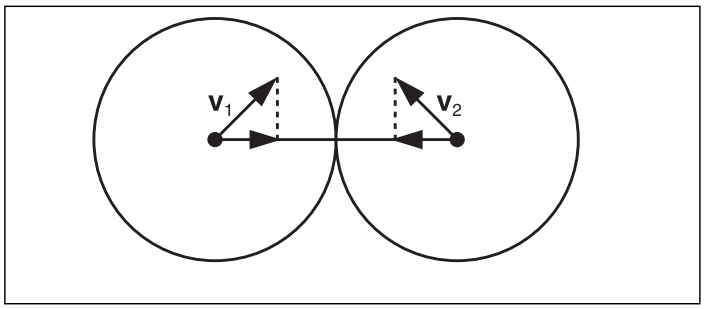


Figure 8.18
To convert this problem to a linear collision, project the initial velocity vectors onto the line of action.

To find the equivalent one-dimensional problem for this collision, all we have to do is project the initial velocities of the spheres onto the line of action, as in Figure 8.18. Using these projected velocities, you solve the problem using the equations we derived for the linear collision.

You can find the unit vector for this line of action by normalizing the displacement vector between the centers of the two spheres. Find the displacement vector by subtracting the position vectors of the spheres, as shown in Figure 8.19.

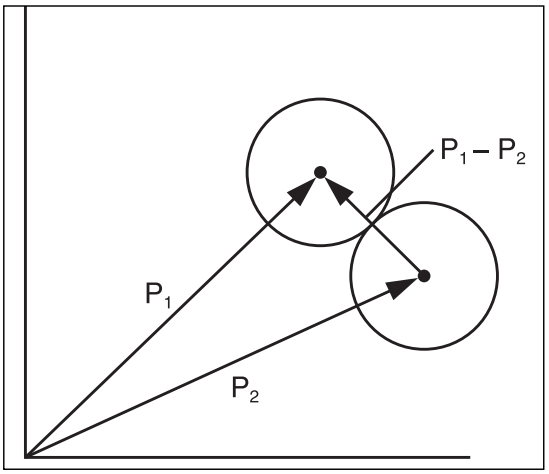


Figure 8.19
The displacement vector between the spheres is the difference of the spheres' position vectors.

After you have the unit vector, which can be represented by $\hat{\mathbf{n}}$, the projection, v_{1p}, of $\mathbf{v_1}$ is $\mathbf{v_1} \cdot \hat{\mathbf{n}}$ and the projection, v_{2p}, of $\mathbf{v_2}$ is $\mathbf{v_2} \cdot \hat{\mathbf{n}}$. These projected velocities are plugged into the linear collision equations.

$$v'_{1p} = \frac{(m_1 - em_2)v_{1p} + (1+e)m_2 v_{2p}}{m_1 + m_2}$$

$$v'_{2p} = \frac{(m_2 - em_1)v_{2p} + (1+e)m_1 v_{1p}}{m_1 + m_2}$$

where v_{1p}' and v_{2p}' are the projections of the velocity vectors after the collision, as shown in Figure 8.20. From these projections, you can calculate the final velocities of the spheres by subtracting the old $\hat{\mathbf{n}}$ component and adding the new one. Calculating the final velocities takes you back into three dimensions, which is what 3-D games require.

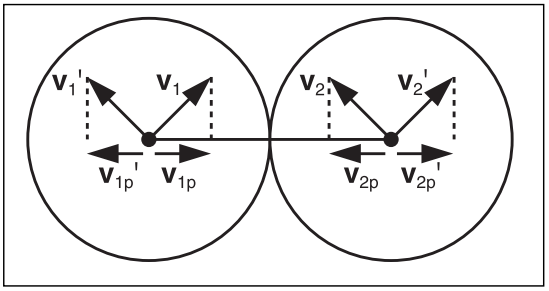


Figure 8.20
The projections of the velocity vectors before and after the collision.

$$\mathbf{v}'_1 = \mathbf{v}_1 + (v'_{1p} - v_{1p})\hat{\mathbf{n}}$$

$$\mathbf{v}'_2 = \mathbf{v}_2 + (v'_{2p} - v_{2p})\hat{\mathbf{n}}$$

Implementation

With all this background, implementation should be a snap.

We're going to treat the collision of two spheres, so we'll use bounding spheres for the collision detection scheme. Everything else follows directly from the equations and line of reasoning that we just completed.

The first task of implementing collision detection and response is to update the `d3d_point_mass` class. Next, we need to set up the simulation. The new aspects of the simulation require that we modify the `UpdateFrame()` function. During each update, the program

must check for collisions and handle them when they occur. Finally, it can render the frame. Let's look at each of these tasks.

note

The code for this chapter's sample program is on the CD-ROM that comes with this book. Look for it in the folder `Source\Chapter08\ParticleBounce`. If you want to view the working program, you can find its `.EXE` file in the folder `Source\Chapter08\Bin`.

Modifying the d3d_point_mass Class

The `d3d_point_mass` class requires only minor modifications for handling collisions. Listing 8.1 gives the new version.

Listing 8.1
The d3d_point_mass Class for Handling Collisions

```
1      class d3d_point_mass
2      {
3      private:
4          d3d_mesh objectMesh;
5
6          scalar mass;
7          vector_3d centerOfMassLocation;
8          vector_3d linearVelocity;
9          vector_3d linearAcceleration;
10         vector_3d sumForces;
11
12         scalar radius;
13         scalar coefficientOfRestitution;
14
15         D3DXMATRIX worldMatrix;
16
17     public:
18         d3d_point_mass();
19
20         bool LoadMesh(
21             std::string meshFileName);
22
23         void Mass(
24             scalar massValue);
25         scalar Mass(void);
26
27         void Location(
```

```
28              vector_3d locationCenterOfMass);
29          vector_3d Location(void);
30
31          void LinearVelocity(
32              vector_3d newVelocity);
33          vector_3d LinearVelocity(void);
34
35          void LinearAcceleration(
36              vector_3d newAcceleration);
37          vector_3d LinearAcceleration(void);
38
39          void Force(
40              vector_3d sumExternalForces);
41          vector_3d Force(void);
42
43          void BoundingSphereRadius(
44              scalar sphereRadius);
45          scalar BoundingSphereRadius(void);
46
47          void Elasticity(scalar elasticity);
48          scalar Elasticity(void);
49
50          bool Update(
51              scalar changeInTime);
52          bool Render(void);
53      };
```

Lines 12 and 13 of Listing 8.1 show that the `d3d_point_mass` class now has private data members that store the radius of the mass's bounding sphere and the mass's coefficient of restitution. Lines 43–48 give the prototypes for the member functions that programs use to get and set these values. Other than the addition of these two data members and four member functions, no other changes to the `d3d_point_mass` class are needed.

Setting Up the Simulation

The code that's specific to this simulation is on the CD-ROM in the file `ParticleBounce.cpp`. The `GameInitialization()` function, which is required by the framework, is similar to the version that appeared in Chapter 7, "Dynamics of Particles." The code for this version is presented in Listing 8.2.

Listing 8.2
Initializing the Colliding Bowling Balls

```
1       bool GameInitialization()
2       {
```

```
3         // Load the first ball's mesh.
4         allParticles[0].LoadMesh("bowlball.x");
5
6         // Set the first ball's mass.
7         allParticles[0].Mass(10);
8
9         // Set the first ball's coefficient of restitution.
10        allParticles[0].Elasticity(0.9f);
11
12        // Set the radius of the bounding sphere.
13        allParticles[0].BoundingSphereRadius(0.75f);
14
15        // Let both balls share the same mesh and other properties.
16        allParticles[1]=allParticles[0];
17
18        // Set the first ball's starting location.
19        allParticles[0].Location(vector_3d(-5.0f,0.0,0.0));
20
21        // Set the second ball's starting location.
22        allParticles[1].Location(vector_3d(0.0,-5.0f,0.0));
23
24        // Set the initial forces on the balls.
25        allParticles[0].Force(vector_3d(2.0f,0.0,0.0));
26        allParticles[1].Force(vector_3d(0.0,2.0f,0.0));
27
28        //
29        // Set up a diffuse directional light.
30        //
31        D3DLIGHT9 light;
32        ZeroMemory( &light, sizeof(light) );
33        light.Type = D3DLIGHT_DIRECTIONAL;
34
35        D3DXVECTOR3 vecDir;
36        vecDir = D3DXVECTOR3(0.0f, -1.0f, 1.0f);
37        D3DXVec3Normalize((D3DXVECTOR3*)&light.Direction,&vecDir);
38
39        // Set the directional light diffuse color.
40        light.Diffuse.r = 1.0f;
41        light.Diffuse.g = 1.0f;
42        light.Diffuse.b = 1.0f;
43        light.Diffuse.a = 1.0f;
44        theApp.D3DRenderingDevice()->SetLight( 0, &light );
```

```
45          theApp.D3DRenderingDevice()->LightEnable( 0, TRUE );
46          theApp.D3DRenderingDevice()->SetRenderState(
47              D3DRS_DIFFUSEMATERIALSOURCE,
48              D3DMCS_MATERIAL);
49
50          return (true);
51      }
```

This version of the `GameInitialization()` function begins by loading the mesh for the first point mass. It uses the bowling ball mesh, as in Chapter 7. It sets the mass to 10 kilograms on line 7. On line 10, `GameInitialization()` sets the ball's coefficient of restitution to 0.9. This is much bouncier than a real bowling ball.

tip

I recommend that you try running this program with various values of the coefficient of restitution.

Line 13 of Listing 8.2 sets the bounding sphere for the first ball. Line 16 copies all the initialized data into the second ball. The result is that they share the same mesh and have the same mass, bounding sphere radius, and coefficient of restitution.

Next, the `GameInitialization()` function sets the starting location of the first ball so that it's off the left edge of the program's window. On line 22, it sets the starting location of the second ball so that it's below the bottom of the window. Lines 25–26 set the initial forces on the balls such that they both fly toward the origin (the middle of the window) at the same speed. This guarantees that they'll hit each other.

The rest of the `GameInitialization()` function sets up the lighting, as was done in Chapter 7.

Updating a Frame

The `UpdateFrame()` function must now test for collisions between the flying bowling balls. If a collision occurs, it must calculate the forces to apply to the balls. I've isolated those calculations into a separate function that's discussed in the next section. For now, look at Listing 8.3 to see how `UpdateFrame()` detects collisions.

Listing 8.3
UpdateFrame() Tests for Collisions

```
1       bool UpdateFrame()
2       {
3           // Set up the view matrix, as in previous examples.
4           D3DXVECTOR3 eyePoint(0.0f,3.0f,-5.0f);
5           D3DXVECTOR3 lookatPoint(0.0f,0.0f,0.0f);
6           D3DXVECTOR3 upDirection(0.0f,1.0f,0.0f);
```

```
7         D3DXMATRIXA16 viewMatrix;
8         D3DXMatrixLookAtLH(&viewMatrix,&eyePoint,&lookatPoint,&upDirection);
9         theApp.D3DRenderingDevice()->SetTransform(D3DTS_VIEW,&viewMatrix);
10
11        // Set up the projection matrix, as in previous examples.
12        D3DXMATRIXA16 projectionMatrix;
13        D3DXMatrixPerspectiveFovLH(
14            &projectionMatrix,D3DX_PI/4,1.0f,1.0f,100.0f);
15        theApp.D3DRenderingDevice()->SetTransform(
16            D3DTS_PROJECTION,&projectionMatrix);
17
18        // These initializations are done only once.
19        static bool forceApplied = false;
20        static vector_3d noForce(0.0,0.0,0.0);
21
22        // If the force has not yet been applied...
23        if (!forceApplied)
24        {
25            forceApplied = true;
26        }
27        // Else the force was already applied...
28        else
29        {
30            // Set the forces to 0.
31            allParticles[0].Force(noForce);
32            allParticles[1].Force(noForce);
33        }
34
35        //
36        // Test for a collision.
37        //
38        // Find the distance vector between the balls.
39        vector_3d distance =
40            allParticles[0].Location() - allParticles[1].Location();
41        scalar distanceSquared = distance.NormSquared();
42
43        // Find the square of the sum of the radii of the balls.
44        scalar minDistanceSquared =
45            allParticles[0].BoundingSphereRadius() +
46            allParticles[0].BoundingSphereRadius();
47        minDistanceSquared *= minDistanceSquared;
48
```

```
49          // Set this to a value between 0 and 1 for smoother animation.
50          scalar timeInterval = 1.0;
51
52          // If there is a collision...
53          if (distanceSquared < minDistanceSquared)
54          {
55              // Handle the collision.
56              HandleCollision(distance,timeInterval);
57          }
58
59          allParticles[0].Update(timeInterval);
60          allParticles[1].Update(timeInterval);
61
62          return (true);
63      }
```

In Listing 8.3, the `UpdateFrame()` function takes care of the typical Direct3D overhead on lines 4–14. On line 19, it declares the variable `forceApplied`, which is used in the same manner that it was in Chapter 7. Line 20 declares a static `vector_3d` variable that `UpdateFrame()` uses to initialize the force vectors on lines 28–29. Collision testing begins on line 39.

To determine whether there has been a collision, the `UpdateFrame()` function calculates the square of the distance between the centers of the two bowling balls. The code for this is on lines 39–41. Next, `UpdateFrame()` adds the radii of the balls' bounding spheres on lines 44–46. It squares that value on line 47. On line 53, it uses these two squared values to test for a collision. If a collision occurs, `UpdateFrame()` calls the `HandleCollision()` function on line 56. As we'll soon see, `HandleCollision()` calculates the forces that are acting on the point masses. When the `d3d_point_mass::Update()` function is called on lines 59–60, the forces of the collision move the point masses.

Handling Collisions

Calculating the forces that result from a collision is the job of the `HandleCollision()` function. This is a helper function that the physics modeling framework does not require. The code for it is in the file `ParticleBounce.cpp`. It's also shown in Listing 8.4.

Listing 8.4
The HandleCollision() Function

```
1       void HandleCollision(
2           vector_3d separationDistance,
3           scalar changeInTime)
4       {
5           //
6           // Find the outgoing velocities.
```

```
7         //
8         /* First, normalize the displacement vector because it's
9         perpendicular to the collision. */
10        vector_3d unitNormal =
11            separationDistance.Normalize(FLOATING_POINT_TOLERANCE);
12
13        /* Compute the projection of the velocities in the direction
14        perpendicular to the collision. */
15        scalar velocity1 =
16            allParticles[0].LinearVelocity().Dot(unitNormal);
17        scalar velocity2 =
18            allParticles[1].LinearVelocity().Dot(unitNormal);
19
20        // Find the average coefficient of restitution.
21        scalar averageE =
22            (allParticles[0].Elasticity()*allParticles[1].Elasticity())/2;
23
24        // Calculate the final velocities.
25        scalar finalVelocity1 =
26            (((allParticles[0].Mass() -
27            (averageE * allParticles[1].Mass())) * velocity1) +
28            ((1 + averageE) * allParticles[1].Mass() * velocity2)) /
29            (allParticles[0].Mass() + allParticles[1].Mass());
30        scalar finalVelocity2 =
31            (((allParticles[1].Mass() -
32            (averageE * allParticles[0].Mass())) * velocity2) +
33            ((1 + averageE) * allParticles[0].Mass() * velocity1)) /
34            (allParticles[0].Mass() + allParticles[1].Mass());
35        allParticles[0].LinearVelocity(
36            (finalVelocity1 - velocity1) * unitNormal +
37            allParticles[0].LinearVelocity());
38        allParticles[1].LinearVelocity(
39            (finalVelocity2 - velocity2) * unitNormal +
40            allParticles[1].LinearVelocity());
41
42        //
43        // Convert the velocities to accelerations.
44        //
45        vector_3d acceleration1 =
46            allParticles[0].LinearVelocity() / changeInTime;
47        vector_3d acceleration2 =
48            allParticles[1].LinearVelocity() / changeInTime;
```

```
49
50          // Find the force on each ball.
51          allParticles[0].Force(
52              acceleration1 * allParticles[0].Mass());
53          allParticles[1].Force(
54              acceleration2 * allParticles[1].Mass());
55      }
```

The problem of calculating the collision forces can be approached in many ways. Earlier in this chapter, we used the conservation of momentum, energy, and work as a means for obtaining formulas for the outgoing velocities of particles after a collision. If we realize that each frame covers a specific amount of time, we can use those formulas to calculate the change in velocity during a period of time. Well, that's the definition of acceleration—the change in velocity divided by the change in time. Knowing the acceleration of a point mass enables us to use the formula **F**=m**a**. We really get a lot of mileage out of that formula.

The `HandleCollision()` function calculates the outgoing velocities of the point masses after the collision using the formulas we derived previously. I've repeated them here for your convenience.

$$\mathbf{v}_1' = \mathbf{v}_1 + (v_{1p}' - v_{1p})\hat{\mathbf{n}}$$
$$\mathbf{v}_2' = \mathbf{v}_2 + (v_{2p}' - v_{2p})\hat{\mathbf{n}}$$

To use these equations, the `HandleCollision()` function must have a unit vector that points along the action of the collision. `HandleCollision()` obtains the unit vector by invoking the `vector_3d::Normalize()` function on lines 10–11. It gets the distance vector through its `separationDistance` parameter from the `UpdateFrame()` function.

After it has the unit normal vector, `HandleCollision()` dots it with the velocity vectors of the particles. This gives the component of the velocity vectors in the direction of action of the collision.

To calculate the final velocity, `HandleCollision()` must use the coefficient of restitution. To enable maximum flexibility, the `d3d_point_mass` class enables each point mass to store its own coefficient of restitution. That allows you to make some objects bouncier than others. However, when a collision occurs, there is only one coefficient of restitution that can be used for the pair of objects. Therefore, `HandleCollision()` averages the coefficient of restitution for the pair of objects and uses that. The code is given on lines 21–22.

tip

For even better results, you could use an average weighted by the masses of the objects involved in the collision.

The final velocities are calculated on lines 24–40. After the `HandleCollision()` function has these velocities, it can calculate the accelerations produced by the collision. That happens on lines 45–48. Using the accelerations, `HandleCollision()` determines the forces acting on the point masses using the formula **F**=m**a**.

Rendering a Frame

Chapter 7 demonstrated how hard rendering a frame *isn't*. If we've modeled all the physics correctly and used the physics to calculate the translation matrix for Direct3D, rendering is easy. In this chapter, rendering requires the addition of only one more line of code. Listing 8.5 shows what it is.

Listing 8.5
Rendering the Colliding Bowling Balls

```
1      bool RenderFrame()
2      {
3          allParticles[0].Render();
4          allParticles[1].Render();
5          return (true);
6      }
```

The only additional work that the `RenderFrame()` function has to do is to call the `d3d_point_mass::Render()` function for two objects instead of just one. It doesn't even break a sweat.

Summary

In this chapter, you learned several ways of detecting collisions between objects. Sure, there's more to learn on the subject, but this is a good start. We talked a lot about the physics of collisions. This understanding is going to be helpful when we're trying to come to grips with rigid body collisions. Finally, we put together a program with two spheres bumping off each other. As we move into the later chapters of the book, you might be surprised at how much you can do with the straightforward physics you learned in this chapter.