CHAPTER 7

Dynamics of Particles

This chapter presents the core of classical physics: the movements of point particles. This topic covers a lot of ground and allows for some pretty good effects, but it's also the starting point for generalizing to the rigid and deformable bodies that are going to become a significant threat to our sanity later on. In addition, this chapter introduces a few forces that you might run into and some others that can really jack up the realism in your game. Oh, and it covers calculus as well. Hold on tight!

Point Particles

The objects we're going to look at in this chapter are smaller than the distances over which they move. Think of a car driving down a freeway. If you look up close, as in Figure 7.1, the car has some orientation and length, people move around inside, and maybe the hubcap falls off. As the car drives down the road, the tires turn. Many forces act on the car, including air resistance, winds blowing the car toward one side or the other, friction between the tires and the road, and so on.

All of that is a lot to try and model. Physicists, and programmers simulating physics, generally start by simplifying the situation. The first step is to back off a bit. In Figure 7.2, we see the big picture. The car looks like a point. All the local complication has been lost to distance. Instantaneously, the whole situation can be described by one vector that gives the position of the car relative to some coordinate system. It's simple.

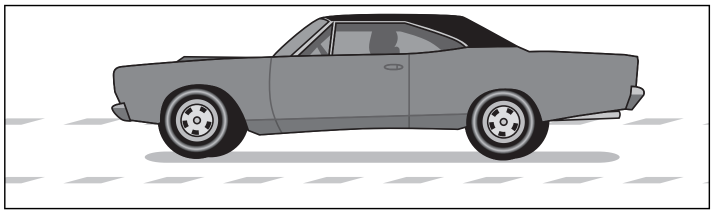
Figure 7.1
A car driving down the freeway looks complicated up close.

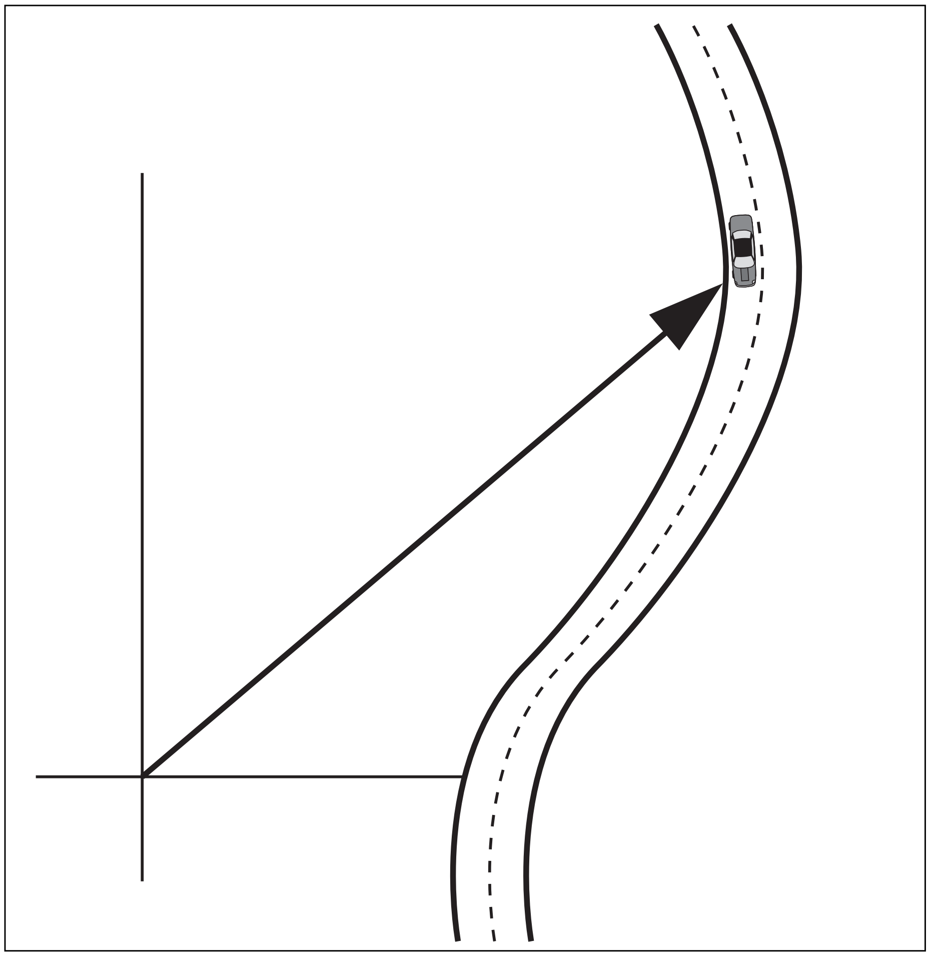

Figure 7.2
The same car, as seen from far away, can be described completely with one vector in a given instant.

When I talk about a point particle, this is what I mean: Ignore the orientation, ignore anything happening internally, and concentrate only on those properties that affect the position.

tip

The key to good physics modeling (and fast code) is knowing what you can ignore in a calculation.

1-D Kinematics

Kinematics is the study of motion in the absence of forces. Let's start our study of kinematics by considering point particles in one dimension. These particles are constrained to move on a line. Particles that are moving along lines with no forces might not seem like the most fascinating thing to look at, but it's a good start, and the subtleties can sometimes be pretty slippery. Let's take a look.

Velocity

The study of kinematics usually starts with velocity. *Velocity* is the amount of distance traveled over some period of time. You can write the equation for velocity like this:

$$v = \frac{\Delta x}{\Delta t}$$

where v is the average velocity, Δx is the distance traveled, and Δt is the time it took to travel the distance.

note

Velocity is measured in miles per hour (mph) in the U.S. and in meters per second (m/s) everywhere else. One meter per second is about two miles per hour.

Imagine that our car has passed the "Welcome to Kansas" sign and is now traveling along a long, straight piece of road at a prodigious velocity (see Figure 7.3). For those who've never been to Kansas, it's very flat, so it makes a good setting for our example. A police officer, sitting in her car behind a bush 500 meters from the state border, starts her stopwatch when the car passes the border and stops it when the car passes the bush. She sees that 10 seconds have passed, so she can compute the average velocity of the car over that distance.

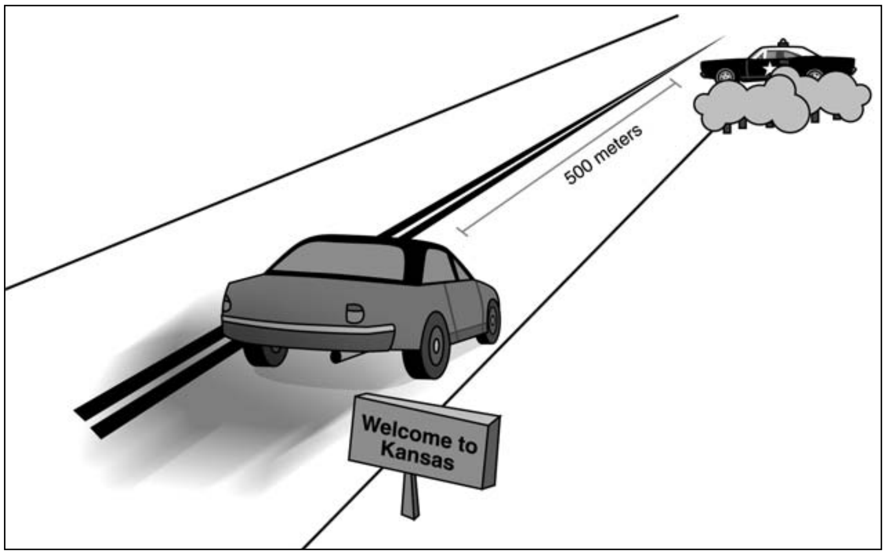

Figure 7.3
A car speeds down the road in Kansas.

$$v = \frac{\Delta x}{\Delta t} = \frac{500 \text{ meters}}{10 \text{ seconds}} = 50 \frac{\text{meters}}{\text{second}}$$

50 meters per second is more than 100 mph, which is somewhat faster than the speed limit in Kansas. The officer feels justified in pulling the car over.

Velocity as a Derivative

The police officer adjusts her sunglasses and saunters up to the car window. The driver of the car sighs and rolls down his window to begin the ritual. On cue, the cop says, "Do you know how fast you were going?" The driver shakes his head, and the cop continues. "I clocked you going more than 100 miles per hour." The driver says, "That's impossible! I was only going to drive for 20 minutes! How could I drive 100 miles in an hour when I wasn't driving for an hour?"

How could the police officer answer this argument? Well, a real traffic cop wouldn't have to, but maybe this cop is a physics student trying to supplement her stipend. The policewoman says, "What I mean is that if you kept going for an hour, you'd have driven 100 miles." The driver replies, "But I was braking and slowing down, so if I'd have kept going, I wouldn't have made 100 miles."

note

My little story about the cop and driver was adapted from *The Feynman Lectures on Physics* by Richard Feynman (the Great Feynman), Robert Leighton, and Matthew Sands. Feynman is one of the greatest physicists of all time and one of its greatest teachers. His *Lectures*, a three-volume set, might be the best book ever written on introductory physics.

The point here is that our formula for velocity, $v = \frac{\Delta x}{\Delta t}$, is good only for computing an average velocity over some distance. This average velocity isn't what we usually mean by velocity; we mean some instantaneous velocity—the velocity at some moment.

Look at Figure 7.4, a graph of the braking car. The curving line represents the distance that the car has traveled at any given time. Because the vertical axis is distance and the horizontal axis is time, the slope of any line on this graph is the speed. Velocity is a vector. It has magnitude and direction. Speed is the magnitude of the velocity vector. It is a scalar quantity that is always positive.

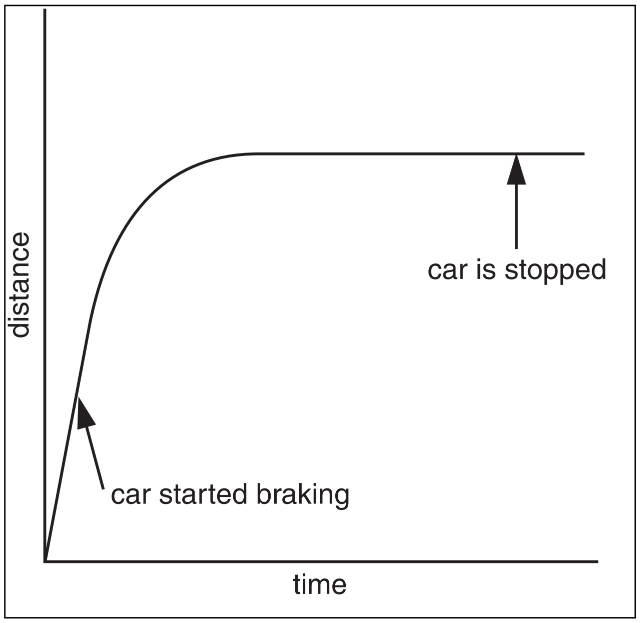

Figure 7.4
Distance versus time for a braking car.

Notice from this figure that as the car slows down, the distance it covers becomes smaller. When the car is stopped, it doesn't matter how much time goes by—the distance traveled doesn't increase.

Let's try to measure the speed at some instant in time. You can get a first estimate for the speed by choosing a Δt centered around the time you would like to know the speed, as in Figure 7.5. Using the graph, you can find the distance measurements at the beginning and end of the time span, Δt. That gives you Δx. The average speed over that time is then Δx / Δt.

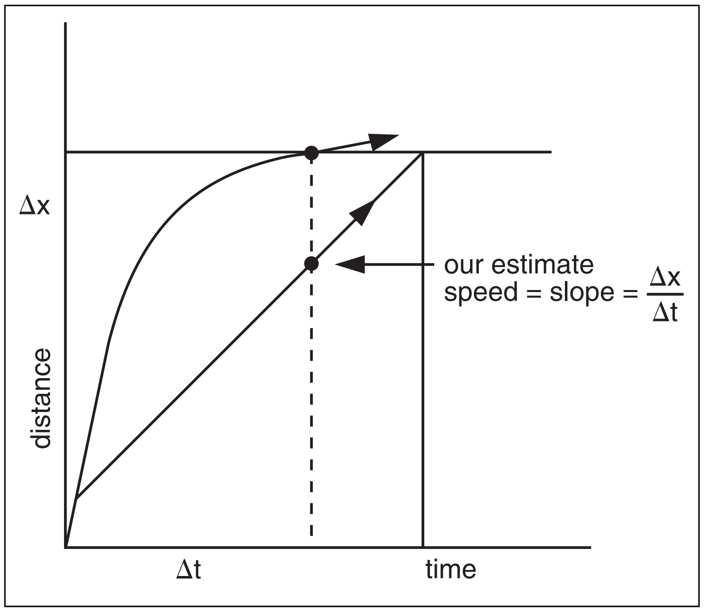

Figure 7.5
Estimating the speed.

You can see that there's quite a difference between the average speed and the actual speed at the time the car was clocked if you compare the slope of the original line to the slope of our estimate. The slope of a line equals the *rise* (change in the vertical axis) of a line divided by the *run* (change in the horizontal axis). A horizontal line has a slope of 0 because the rise is 0. A line running at a 45^{o} angle has a slope of 1. A vertical line has an undefined slope because the run is 0 (slope = [rise] / 0 = undefined). In this case, the distance is on the vertical axis, so the rise is Δx. The horizontal axis is time, so Δt is the run.

You might have already figured out that we can get a better estimate by making Δt smaller. This works really well; look at Figure 7.6.

You can keep repeating this process, making the Δt smaller and smaller and getting better and better estimates for the speed at that moment. This is a method we'll use a lot for our physics models. We can make the solutions more exact by taking smaller time steps.

You can even ask what happens in the limit as Δt goes to 0.

$$v = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t}$$

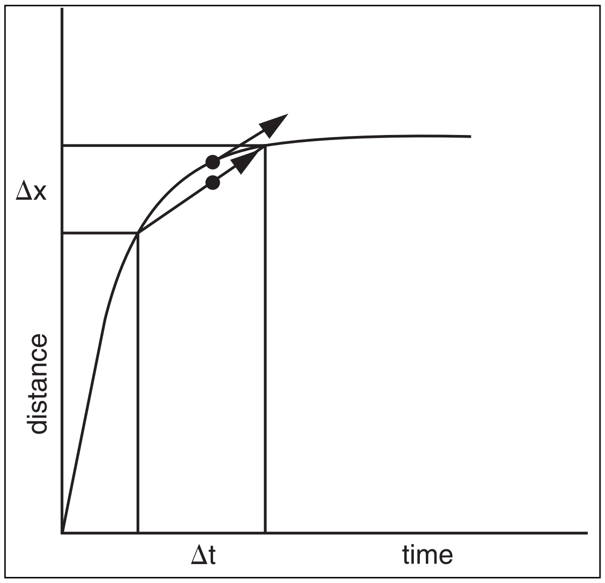

Figure 7.6
Get a better estimate of the speed by using a smaller Δt.

This is the instantaneous speed we've been looking for. This infinitely small Δt is written dt, and the infinitely small Δx that results is written dx. Using this notation, you can write the previous equation like this:

$$v = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} = \frac{dx}{dt}$$

The quantity dx / dt is called "the derivative of x with respect to t." The process of getting the value of the derivative is called *differentiation*.

note

This is an *extremely* fast and loose introduction to differential calculus. "Nonrigorous" doesn't even begin to describe it. For a more mathematical introduction, check out any introductory text in calculus. A nice one is Gerald Bradley and Karl Smith's book, descriptively titled *Calculus*. For a more rigorous treatment (and a certain amount of masochistic pleasure), check out an introduction to analysis. My favorite is *Yet Another Introduction to Analysis* by Victor Bryant.

Often, we can ignore differentiation because we can use the computer to make good numerical approximations for us.

Acceleration

Remember the braking car? A braking car is slowing down; that is, the speed of the car is changing with time. *Acceleration* is the rate of change of the speed with respect to time, just as *speed* is the rate of change of distance with respect to time.

You can write the average acceleration just like you might expect:

$$a = \frac{\Delta v}{\Delta t}$$

where a is the average acceleration, Δs is the change in speed, and Δt is the time period it underwent that change. The units of acceleration are m/s^2.

Here's an example. Let's say that you can go from 0 to 32 m/s in 4 seconds on your new motorcycle. Your average acceleration over that time is as follows:

$$a = \frac{32\frac{m}{s}}{4s} = 8\frac{m}{s^2}$$

Now let's try it in reverse. Say that your acceleration is 4 m/s^2 for 5 seconds. You can compute your speed by rearranging the equation for acceleration:

$$a = \frac{\Delta v}{\Delta t}$$

$$v = a\Delta t = (4\frac{m}{s^2})(5s) = 20\frac{m}{s}$$

You can get the instantaneous acceleration by taking the limit as Δt goes to 0.

$$a = \lim_{\Delta t \to 0} \frac{\Delta v}{\Delta t} = \frac{dv}{dt}$$

The instantaneous acceleration shows the acceleration at any given moment.

Forces

Most of the ancients in the West thought that moving things naturally come to a stop. This seems to be borne out by experience. Anyone who's ever tried to move heavy furniture will vouch that getting it moving is no guarantee that it will keep moving. However, the ancients had some problems with their theory. Great thinkers throughout history observed these problems and tried to sort them out.

Galileo came up with a successful alternative theory called the *principle of inertia*. His theory states that something moving at a constant speed will continue to move at that speed unless acted upon. This took a certain amount of imagination because almost everything in this world does stop unless you keep pushing it, but Galileo realized that this slowing was due to friction between surfaces rather than the inherent nature of things.

Sir Isaac Newton later formulated this same idea in a more precise way. He called it his first law of motion. Newton's second law asks the next logical question. If objects that aren't acted upon continue to travel at the same speed forever, what happens if objects *are* acted upon? The answer is that the force is equal to the mass of an object times its acceleration:

$$F = ma$$

Most objects have some mass, which is a measure of how hard it is to change the velocity of an object; you have to push a more massive object harder than a less massive object to get it to change speed. This formula assumes that the mass of an object does not change over time.

For example, we can use this formula to calculate the forces acting on a car as it travels down the road. But nothing must fall off the car. You might point out that the mass of the car decreases over time because it burns up its fuel. You're right. However, the change in the car's mass is small enough to be ignored in 3-D simulations.

The formula F=ma is the way most people remember the second law for Newtonian mechanics because it's so useful. If I give you the force, you can use the second law to find the acceleration, use the acceleration to find the velocity, and use the velocity to find the distance traveled.

note

You might have heard that the mass isn't constant in special relativity. Don't worry about it. Unless you're specifically trying to simulate special relativity (I don't know of any games that do), you can assume that mass is constant.

The only trick is to find the forces. In modern physics, there are really only four forces: gravitational, electromagnetic, weak nuclear, and strong nuclear. In theory, you should be able to calculate any problem by properly treating these four forces. In reality, doing this is much too complicated.

What you do in practice is measure the type of force in several situations and look for an equation that describes the results well under some subset of conditions. For example, if you stretch a spring, you'll notice that it applies a force against you, as shown in Figure 7.7.

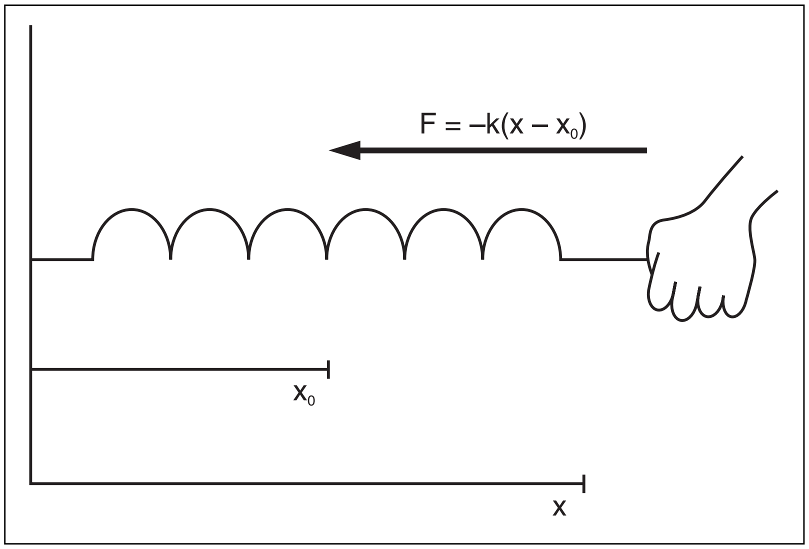

Figure 7.7
Forces on a stretched spring.

You can model this force with this equation:

$$F = -k(x - x_0)$$

where x is the position of the end of the spring and x_o is the position of the spring at equilibrium. k is a constant that's called the spring constant.

This equation says that if you increase the distance between x and x_o, $(x - x_o)$, the force grows larger, which corresponds to our experience with springs. The value of k depends on the stiffness of the spring.

2-D and 3-D Kinematics

All of the equations we've been talking about are extended easily to more dimensions.

Look at the first kinematic equation:

$$v_x = \frac{dx}{dt}$$

If x is the distance along the x axis of a 2-D Cartesian coordinate system, this equation describes the velocity of the particle in the x direction, as shown in Figure 7.8. You can treat the motion along the y axis completely independently, like this:

$$v_y = \frac{dy}{dt}$$

When you're working in three dimensions, the same is true of motion along the z axis.

$$v_z = \frac{dz}{dt}$$

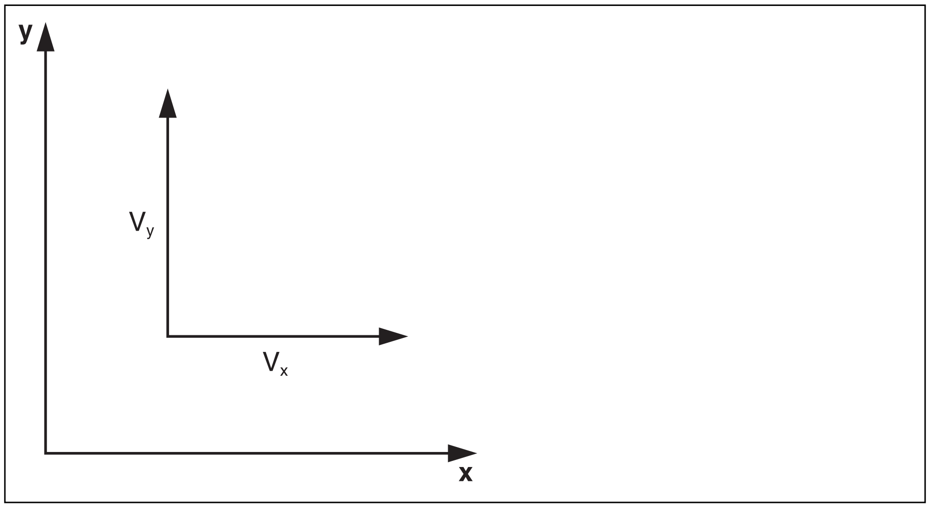

Figure 7.8
Velocities along the x and y axes.

Now, look at this. If **v** is the vector whose components in a 2-D coordinate system are (v_x, v_y), and **x** has components (x, y), you can write these two equations together:

$$\mathbf{v} = \frac{d\mathbf{x}}{dt}$$

In components, that's

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \frac{d\begin{bmatrix} x \\ y \end{bmatrix}}{dt} = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \end{bmatrix}$$

Now that you have this equation in vector form, you can guarantee that it holds in all dimensions and coordinate systems. In three dimensions, then, the equation is still this:

$$\mathbf{v} = \frac{d\mathbf{x}}{dt}$$

and the components are usually written like this:

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \frac{d\begin{bmatrix} x \\ y \\ z \end{bmatrix}}{dt} = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{bmatrix}$$

The velocity vector, being a vector, has both a direction and a magnitude. The direction is the direction the point particle is moving, and the magnitude is the speed, as shown in Figure 7.9:

$$\text{speed} = |\,v\,| = v$$

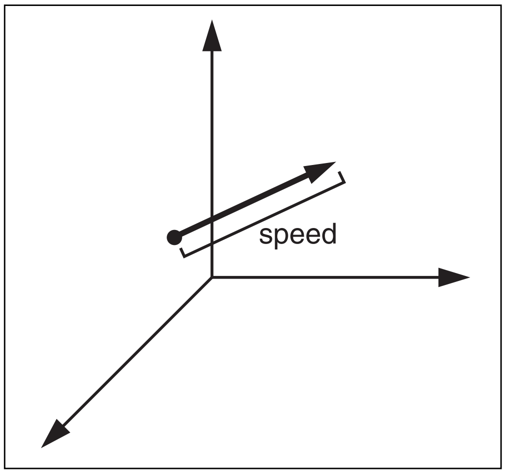

Figure 7.9
The velocity vector of a point particle can be resolved into direction and speed.

You can generalize all the other equations I've mentioned in the same way by expressing them as vectors, as shown here:

$$\mathbf{v} = \frac{d\mathbf{x}}{dt}$$

$$\mathbf{a} = \frac{d\mathbf{v}}{dt}$$

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

These equations express velocity, acceleration, and force as vectors. You might ask, "So what?"

The answer is that when you're programming 3-D scenes that involve complex systems of moving objects, you can often calculate how the objects behave by treating them as if they were point particles. This generalization provides a simple way to find the linear velocity and acceleration of each object, as well as forces acting on each one. Being able to express velocity, acceleration, and force as vectors means that you can express them as matrices.

Again, you might ask, "So what?"

Recall that 3-D objects in software are defined as meshes that are really a collection of vertices. As an object in a 3-D scene moves, every vertex in the mesh has to move. That means a game must apply the concepts of velocity, acceleration, and force to each vertex in the mesh. Complex objects can involve meshes containing 200,000 triangles with as many as 600,000 vertices!

How can a program possibly calculate the movement of every vertex of an object that complex?

It doesn't. It treats the object as a point and uses vectors representing the forces acting on the object to calculate the velocity and acceleration of the point object. It then expresses velocity and acceleration as matrices. Moving every vertex in the object becomes a matter of doing a matrix multiplication on each vertex. This moves the entire object in a way that realistically simulates the physics of our universe.

The Modeling Point Masses

Now that we have the physics we need, we can write code that models the physics of point masses. This requires that we first create a class for point masses. Next, we'll put the point mass into an environment that is free of gravity and friction. This enables us to see how point masses move with just one external force acting on them. Later chapters demonstrate how to add gravity and friction.

The sample program will display a nice bowling ball moving from left to right across the program's window. Because we're not yet looking at rotational kinematics, the same program doesn't make the ball roll.

Introducing the d3d_point_mass Class

A class that models point masses must store information about its mass, its location, and the forces acting on it. Listing 7.1 presents the definition of the `d3d_point_mass` class.

note

You can find the code for this sample program on the CD-ROM that comes with this book. It's in the folder `Source\Chapter07\PointMass`. If you want to take a quick look at the program in action, you'll find the executable version of it in `Source\Chapter07\Bin`.

Listing 7.1
The d3d_point_mass Class

```
class d3d_point_mass
{
private:
    d3d_mesh objectMesh;

    scalar mass;
    vector_3d centerOfMassLocation;
    vector_3d linearVelocity;
    vector_3d linearAcceleration;
10        vector_3d sumForces;
11
12        D3DXMATRIX worldMatrix;
13
14    public:
15        d3d_point_mass();
16
17        bool LoadMesh(
18            std::string meshFileName);
19
20        void Mass(
21            scalar massValue);
22        scalar Mass(void);
23
24        void Location(
25            vector_3d locationCenterOfMass);
26        vector_3d Location(void);
27
28        void LinearVelocity(
29            vector_3d newVelocity);
30        vector_3d LinearVelocity(void);
31
32        void LinearAcceleration(
33            vector_3d newAcceleration);
34        vector_3d LinearAcceleration(void);
35
36        void Force(
37            vector_3d sumExternalForces);
38        vector_3d Force(void);
39
40        bool Update(
41            scalar changeInTime);
42        bool Render(void);
43    };
```

The `d3d_point_mass` class defines private data members that contain the object's mesh, mass, center of mass, and properties of movement. This class also defines a special data member just for working with Direct3D. If you look on line 12, you'll see a member called `worldMatrix`. The world matrix was explained in Chapter 4, "2-D Transformations and Rendering." Direct3D uses the world matrix to update the position and orientation of objects in 3-D space. If your program has multiple `d3d_point_mass` objects moving through a scene, each one needs a different world matrix to track its position and orientation. The d3d_point_mass class sets its world matrix each time the program calls its Update() function. In its Render() function, the d3d_point_mass class uses the world matrix to animate the point mass.

In addition to the private member data, the d3d_point_mass class defines its public member functions on lines 15–42. Most of them simply set or get the member data. The LoadMesh(), Update(), and Render() functions do the real work for the class. The LoadMesh() function is so simple that its defined as an inline member function in the file PMPointMass.h. It's shown in Listing 7.2.

Listing 7.2
The LoadMesh() Function

```
1       inline bool d3d_point_mass::LoadMesh(
2           std::string meshFileName)
3       {
4           assert(meshFileName.length()>0);
5
6           return (objectMesh.Load(meshFileName));
7       }
```

The LoadMesh() function uses the d3d_mesh::Load() function to load the mesh that gives its appearance.

tip

On line 4 of Listing 7.2, the LoadMesh() function uses an assert() to ensure that the length of the file name is greater than 0. If a program calls this function with an empty file name, the program does not have enough error checking in it. That's a programmer error rather than a runtime error. This function uses the assert() to make sure the programmer catches the error before the software is released. In general, that's how software should guard against programmer errors in parameters.

The Update() function has to work a little harder. It appears in Listing 7.3.

Listing 7.3
The Update() Function

```
bool d3d_point_mass::Update(
    scalar changeInTime)
{
    //
    // Begin calculating linear dynamics.
    //

    // Find the linear acceleration.
9           // a = F/m
10          assert(mass!=0);
11          linearAcceleration = sumForces/mass;
12
13          // Find the linear velocity.
14          linearVelocity += linearAcceleration * changeInTime;
15
16          // Find the new location of the center of mass.
17          centerOfMassLocation += linearVelocity * changeInTime;
18
19          //
20          // End calculating linear dynamics.
21          //
22
23          // Create the translation matrix.
24          D3DXMatrixTranslation(
25              &worldMatrix,
26              centerOfMassLocation.X(),
27              centerOfMassLocation.Y(),
28              centerOfMassLocation.Z());
29
30          return(true);
31      }
```

The `d3d_point_mass` class's `Update()` function calculates the linear dynamics of a point mass. For now, it ignores real-world factors such as rotation, friction, and gravity. However, the implementation here forms a solid basis for implementing those real-world factors.

The `Update()` function begins by asserting that the mass of the object is not zero. This is an important assertion because it catches a common programmer error. The mass should never be zero or negative because that's physically impossible.

warning

In physics simulations, we often ignore the mass of some objects in the system. This helps keep the calculations simple enough to be manageable. If you're using this technique in a game, don't use the `d3d_point_mass` class to simulate a massless objects. That's not what it's for.

On line 11 of Listing 7.3, the `Update()` function rearranges the formula **F**=m**a** into **a**=**F**/m so that it can find the acceleration of the point mass. In each frame of the game, your program has to calculate the forces acting on the point mass. It then calls the `d3d_point_mass::Force()` function to set the sum of all the forces. When the program invokes the `d3d_point_mass::Update()` function, `Update()` uses the force to calculate the object's reaction.

Next, `Update()` uses the acceleration to find the new velocity of the object at the end of the time period specified by the parameter `changeInTime`. On line 17, `Update()` uses the velocity (and the time) to find the new location of the object's center of mass.

The calculations on lines 11, 14, and 17 are fairly straightforward because of the tools we developed in Chapter 3, "Mathematical Tools for Physics and 3-D Programming." The vectors presented there make these calculations easy. The next step is to convert to matrices so that the program can use the transformations presented in Chapters 4 and 5, "3-D Transformations and Rendering." This begins on line 24 of Listing 7.3.

On lines 24–28, the `Update()` function invokes the Direct3D `D3DXMatrixTranslation()` utility function to build a translation matrix. It builds the matrix from the x, y, and z locations of the object's center of mass. The `D3DXMatrixTranslation()` function stores the translation matrix into the `d3d_point_mass` object's world matrix. The program uses the world matrix when it calls the `d3d_point_mass::Render()` function, which is presented in Listing 7.4.

Listing 7.4
The Render() Function

```
bool d3d_point_mass::Render(void)
{
    // Save the world transformation matrix.
    D3DXMATRIX saveWorldMatrix;
    theApp.D3DRenderingDevice()->GetTransform(
        D3DTS_WORLD,
        &saveWorldMatrix);

    // Apply the world transformation matrix for this object.
    theApp.D3DRenderingDevice()->SetTransform(
        D3DTS_WORLD,&worldMatrix);

    // Now render the object with its transformations.
    bool renderedOK=objectMesh.Render();

    // Restore the world transformation matrix.
    theApp.D3DRenderingDevice()->SetTransform(
        D3DTS_WORLD,
        &saveWorldMatrix);

    return (renderedOK);
}
```

To render just one `d3d_point_mass` object, the `Render()` function needs a world transformation matrix for that specific object. It uses that world matrix to position the `d3d_point_mass` object in 3-D space. However, Direct3D allows only one world matrix at a time. Therefore, the `Render()` function must perform the following steps:

1. Save the existing world matrix.
2. Set the `d3d_point_mass` object's world matrix as the current world matrix.
3. Render the `d3d_point_mass` object's mesh.
4. Restore the existing world matrix saved in step 1.

Using these steps, the `Render()` function can position the `d3d_point_mass` object in the world and render it without disturbing the contents of the existing world matrix. If the `Render()` function did not save and then restore the existing world matrix, the translation for the current `d3d_point_mass` object might get applied to other objects in the scene. The results could be undesirable, to say the least.

Listing 7.4 shows that the `Render()` function for the `d3d_point_mass` class follows these steps exactly. The function begins by declaring a temporary variable to save the existing world matrix into. Next, it calls the Direct3D `GetTransform()` function to store the current world matrix in the variable. On lines 10–11, it sets the `d3d_point_mass` object's world matrix as the current world matrix. It renders the mesh by calling the `d3d_mesh::Render()` function on line 14. Lines 17–19 restore the existing world matrix.

Using the d3d_point_mass Class

Okay, so now we have a `d3d_point_mass` class that's just waiting to be used. Let's use it. However, before we do, I want to take a moment to talk briefly about lighting in Direct3D. We'll use some of Direct3D's lighting capabilities to get a decent-looking bowling ball in the sample program.

Enabling Direct3D Lighting

One of the ways we make 3-D objects in our games look better is by using Direct3D's lighting capabilities. At this point, I could launch into a big discussion of how light behaves in nature and how Direct3D simulates it. But I won't. Although light is an aspect of physics, it's not the focus of this book. What we're trying to do here is get things to behave as they would in the real world. Making things look nice is the subject of another book.

note

Color, lighting, and materials are subjects that are closely related to texturing. Games require a knowledge of all of these subjects to look professional. A good introduction to all of them is Mason McCuskey's *Special Effects Game Programming with DirectX* (Premier Press).

For our purposes, all we really need to know is that DirectX does a lot of the work of lighting 3-D scenes for us. It defines several different types of lights. Among these are ambient light and diffuse light. At this point, we'll only worry about diffuse light.

The color that we see on the screen when Direct3D displays a 3-D object depends on the color of the object's material and the color of the light. If the program adds a texture, that factors in as well. However, let's set that aside for now.

The bowling ball used in the sample program is blue. What we want is for the ball to look blue and to look round. Using diffuse lighting can get us that. Let's make some minor changes to the framework so that we can use diffuse lighting.

First, we have to change the `InitD3D()` function in `D3DApp.cpp` so that it uses Direct3D's lighting. Look at the source code on the CD-ROM. Open the file `D3DApp.cpp` in the folder `Source\Chapter07\PointMass` and find the `InitD3D()` function. Right near the end of the function is a line of code that reads

```
theApp.d3dDevice->SetRenderState(D3DRS_LIGHTING,theApp.enableD3DLighting);
```

In previous chapters, we had lighting turned off, so that line read

```
theApp.d3dDevice->SetRenderState(D3DRS_LIGHTING,TRUE);
```

Now we're passing that information to the framework in the `OnAppLoad()` function. Listing 7.5 presents the code for `OnAppLoad()`.

Listing 7.5
The New Version of OnAppLoad()

```
bool OnAppLoad()
{
    window_init_params windowParams;
    windowParams.appWindowTitle = "Point Mass Test";
    windowParams.defaultX=100;
    windowParams.defaultY=100;
    windowParams.defaultWidth = 400;
    windowParams.defaultHeight = 400;

    d3d_init_params d3dParams;
    d3dParams.renderingDeviceClearFlags = D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER;
    d3dParams.surfaceBackgroundColor = D3DCOLOR_XRGB(50,50,50);
    d3dParams.enableAutoDepthStencil = true;
    d3dParams.autoDepthStencilFormat = D3DFMT_D16;
    d3dParams.enableD3DLighting = true;

    // This call must appear in this function.
18        theApp.InitApp(windowParams,d3dParams);
19
20        return (true);
21    }
```

Remember your framework requires the `OnAppLoad()` function. In this sample program, you'll find it in the file `PointMassTest.cpp`.

As in Chapter 6, "Meshes and X Files," the programs pass the startup parameters for Windows and Direct3D into the framework using the `d3d_init_params` structure. I added some structure members that enable games to specify the default position and size. I also added a member that lets them turn Direct3D lighting on or off when the game starts. Here's the new structure definition:

```
struct d3d_init_params
{
    DWORD renderingDeviceClearFlags;
    D3DCOLOR surfaceBackgroundColor;
    bool enableAutoDepthStencil;
    D3DFORMAT autoDepthStencilFormat;
    bool enableD3DLighting;
};
```

Line 15 of `OnAppLoad()` in Listing 7.5 sets the `enableD3DLighting` member to `true`. `OnAppLoad()` passes the structure to the framework's `InitApp()` function. `InitApp()` is a member of the `d3d_app` class. It's on the CD-ROM in the file `PMD3DApp.h`, which is in the folder `Source\Chapter07\PointMass`. The code for the `InitApp()` function is also given in Listing 7.6.

Listing 7.6
The New InitApp() Function

```
1     inline bool d3d_app::InitApp(
2         window_init_params windowParams,
3         d3d_init_params d3dParams)
4     {
5         // Set the initial window parameters.
6         windowTitle=windowParams.appWindowTitle;
7         defaultX = windowParams.defaultX;
8         defaultY = windowParams.defaultY;
9         defaultHeight = windowParams.defaultHeight;
10       defaultWidth = windowParams.defaultWidth;
11
12        // Set the initial D3D parameters.
13        deviceClearFlags = d3dParams.renderingDeviceClearFlags;
14        backgroundColor = d3dParams.surfaceBackgroundColor;
```

```
        enableAutoDepthStencil = d3dParams.enableAutoDepthStencil;
        autoDepthStencilFormat = d3dParams.autoDepthStencilFormat;
        enableD3DLighting = d3dParams.enableD3DLighting;

        appInitialized=true;
        return(appInitialized);
    }
```

As you can see from Listing 7.6, the new version of the `InitApp()` function simply copies the information it received in the structure into some new members of the `d3d_app` class. Let's look at Listing 7.7 to see the new definition of the `d3d_app` class.

Listing 7.7
The New d3d_app Class Definition

```
class d3d_app
{
private:
    // App properties
    bool appInitialized;

    // Window properties
    std::string windowTitle;
    int defaultX, defaultY;
    int defaultHeight,defaultWidth;

    // D3D properties
    LPDIRECT3D9             direct3D; // Used to create the D3DDevice
    LPDIRECT3DDEVICE9       d3dDevice; // Our rendering device
    LPDIRECT3DVERTEXBUFFER9 vertexBuffer; // Buffer to hold vertices
    DWORD deviceClearFlags;
    D3DCOLOR backgroundColor;
    bool enableAutoDepthStencil;
    D3DFORMAT autoDepthStencilFormat;
    bool enableD3DLighting;

public:
    d3d_app();
    bool InitApp(
        window_init_params windowParams,
        d3d_init_params d3dParams);

    LPDIRECT3DDEVICE9 D3DRenderingDevice(void);
29
30        LPDIRECT3DVERTEXBUFFER9 D3DVertexBuffer(void);
31        void D3DVertexBuffer(
32            LPDIRECT3DVERTEXBUFFER9 vertexBufferPointer);
33
34        DWORD RenderingDeviceClearFlags(void);
35        D3DCOLOR BackgroundSurfaceColor(void);
36
37        friend INT WINAPI AppMain(
38            HINSTANCE hInst,
39            HINSTANCE,
40            LPSTR,
41            INT);
42        friend HRESULT InitD3D(
43            HWND hWnd);
44        friend VOID CleanupD3D();
45    };
```

As you look at Listing 7.7, notice in particular that there are new members on lines 9–10 and 20. These accommodate the new information being passed in through the `InitApp()` function. As I mentioned earlier, this information is used in the `InitD3D()` function.

Okay, so now we have Direct3D lighting activated, and we're ready to start tossing a bowling ball around. Let's get back to our discussion of how to use the `d3d_point_mass` class for modeling physics.

Initializing a d3d_point_mass Object

If you look at the `GameInitialization()` function in the file `PointMassTest.cpp`, you'll see that it initializes a `d3d_point_mass` object. The declaration of the `d3d_point_mass` object, which looks like this:

```
d3d_point_mass theObject
```

appears near the top of the file. Take a look at Listing 7.8 to see how the `GameInitialization()` function sets up the `d3d_point_mass` object.

Listing 7.8
The GameInitialization() Function

```
1     bool GameInitialization()
2     {
3         // Load the ball's mesh.
4         theObject.LoadMesh("bowlball.x");
5
6         // Set its starting location.
7           theObject.Location(vector_3d(-5.0f,0.0,0.0));
8
9           // Set the mass.
10          theObject.Mass(10);
11
12          D3DLIGHT9 light;
13          ZeroMemory( &light, sizeof(light) );
14          light.Type = D3DLIGHT_DIRECTIONAL;
15
16          D3DXVECTOR3 vecDir;
17          vecDir = D3DXVECTOR3(0.0f, -1.0f, 1.0f);
18          D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDir );
19
20          // Set directional light diffuse color.
21          light.Diffuse.r = 1.0f;
22          light.Diffuse.g = 1.0f;
23          light.Diffuse.b = 1.0f;
24          light.Diffuse.a = 1.0f;
25          theApp.D3DRenderingDevice()->SetLight( 0, &light );
26          theApp.D3DRenderingDevice()->LightEnable( 0, TRUE );
27          theApp.D3DRenderingDevice()->SetRenderState(
28              D3DRS_DIFFUSEMATERIALSOURCE,
29              D3DMCS_MATERIAL);
30
31          return (true);
32      }
```

The `GameInitialization()` function in the sample program for this chapter accomplishes three basic tasks. On line 4 of Listing 7.8, it loads the point mass's mesh. The mesh is a bowling ball that is defined in the file `bowlball.x`. This file is with the sample program in the folder `Source\Chapter07\PointMass`.

The second task that `GameInitialization()` performs is to set the properties of the point mass. It sets the starting location of the bowling ball to the left of the program's window. It isn't displayed initially. `GameInitialization()` also sets the mass of the bowling ball. I've set it to 10 kilograms (22 pounds), which is a fairly hefty weight for a bowling ball.

The `GameInitialization()` function's final task is to set up diffuse lighting for Direct3D. On line 12, it declares a `D3DLIGHT9` variable. It calls the Windows `ZeroMemory()` function to initialize the light. On line 14, it sets the light type as directional. `GameInitialization()` uses a `D3DXVECTOR3` variable on lines 16–18 to define a vector that specifies where the light points. The light color is set to white on lines 21–24. Line 25 tells Direct3D to use the light, and line 26 tells it to turn the light on. Lines 27–29 make Direct3D combine the light color and the ball's material color to get the final color.

Well, everything's ready. We have a point mass with a mesh. We have our lights set up. I suppose this is a good time to make some pithy joke that transitions us to the next section that involves the phrase, "Lights, camera, action." Unfortunately, I can't think of one. So please just read the next section.

Updating a d3d_point_mass Object

When the program runs, it updates the point mass for each frame of animation. As presented in previous chapters, the update is done in the required function `UpdateFrame()`. To see the code for `UpdateFrame()`, refer to Listing 7.9.

Listing 7.9
The UpdateFrame() Function

```
bool UpdateFrame()
{
    // Set up the view matrix as in previous examples.
    D3DXVECTOR3 eyePoint(0.0f,3.0f,-5.0f);
    D3DXVECTOR3 lookatPoint(0.0f,0.0f,0.0f);
    D3DXVECTOR3 upDirection(0.0f,1.0f,0.0f);
    D3DXMATRIXA16 viewMatrix;
    D3DXMatrixLookAtLH(&viewMatrix,&eyePoint,&lookatPoint,&upDirection);
    theApp.D3DRenderingDevice()->SetTransform(D3DTS_VIEW,&viewMatrix);

    // Set up the projection matrix as in previous examples.
    D3DXMATRIXA16 projectionMatrix;
    D3DXMatrixPerspectiveFovLH(&projectionMatrix,D3DX_PI/4,1.0f,1.0f,100.0f);
    theApp.D3DRenderingDevice()->SetTransform(D3DTS_PROJECTION,&projectionMatrix);

    //
    // Apply a one-time force to the ball.
    //
    // This initialization is done only once.
    static bool forceApplied = false;

    // If the force has not yet been applied...
    if (!forceApplied)
    {
        // Apply a force.
        theObject.Force(vector_3d(2.0f,0.0,0.0));
        forceApplied = true;
    }
    // Else the force was already applied...
    else
31          {
32              // Set it to zero.
33              theObject.Force(vector_3d(0.0,0.0,0.0));
34          }
35
36          /* Set the parameter to a value between 0 and 1 for smoother
37          animation. */
38          theObject.Update(1);
39
40          return (true);
41      }
```

The `UpdateFrame()` function starts off similarly to the `UpdateFrame()` functions from previous chapters. It first sets up the matrices that Direct3D requires. When that's done, it applies a one-time force to the bowling ball to make it move. To do so, `UpdateFrame()` declares a static variable.

You might know that in C++, static variables in functions are initialized the first time the program calls the function. After that, the initialization is never performed. Using a static variable here helps the program keep track of whether the force has been applied. If it hasn't, the value of `forceApplied` is `false`. That causes the `if` statement, which begins on line 23, to set the force acting on the bowling ball on line 26. It also sets the variable `forceApplied` to `true`.

note

Notice that the force is applied in the positive x direction. That makes the bowling ball move toward the right. Its initial position was set as off-screen to the left. Therefore, when the program runs, the ball moves across the program's window from left to right. After the ball disappears off the right end of the window, close the program. You won't see it do anything else.

The next time that the framework calls the `UpdateFrame()` function, `forceApplied` will still be `true`. When it is, the program executes the `else` clause of the `if` statement. The `else` clause sets the force to zero. The result is that the ball gets a push the first time the `UpdateFrame()` function executes. After that, no other forces act on the ball. The ball just keeps moving into virtual infinity for as long as you let the program run. It's nice to know that our simulation acts in accordance with Newton's laws. It tells us that we're writing the program correctly.

The last thing that the `UpdateFrame()` function does is to call the `d3d_point_mass::Update()` function on the bowling ball. The `d3d_point_mass::Update()` function updates the ball's position based on the force acting on it (if any), the current velocity (if any), and the current acceleration (if any). If you want to look again at the `d3d_point_mass::Update()` function, it's back in Listing 7.3.

Rendering a d3d_point_mass Object

Rendering the bowling ball is really, really tough. If fact, it's excruciating. Are you ready for it? Okay, look at Listing 7.10.

Listing 7.10
Too Tough for Me, Baby

```
bool RenderFrame()
{
    theObject.Render();
    return (true);
}
```

Wow! That was a killer! Okay, so I'm a ham. If you're new to computer graphics, you just can't appreciate how wonderful it is to have something this short that renders a complex object like a bowling ball. I started in graphics more than 20 years ago. Back then, I had to write my own renderer to get anything displayed. It was in Assembly Language. If you don't know what that means, you're very, very lucky. Those were sometimes painful years.

In any case, Listing 7.10 shows that all the `RenderFrame()` function does is call the `d3d_point_mass` class's `Render()` function. That function, in turn, uses the `d3d_mesh` function to apply the world matrix created by rendering the `d3d_point_mass` class's `Update()` function. It also calls the `d3d_mesh::Render()` function to render the point mass's mesh. Because the underlying functions do so much, there's not really a lot for the `RenderFrame()` function to do. In other words, when you're using the `d3d_point_mass` class, you just have to set it up, apply a force, and let it go. There's really little else to do. Nice.

Point Masses in Games

How often do we *really* use point masses in games? Constantly. And there are more uses coming. As processing power increases, point masses will become much more important in games. Let me give you an example.

Suppose that you're writing a game that lets the player shoot a wall. Such games do not use point masses. When the bullet hits, these games apply a texture to the wall to show the bullet hole and discoloration caused by the gunpowder. But the wall's not really damaged. That's okay, but games are getting better.

Some games use what are called particle systems to simulate small fragments of the wall flying around the room. The fragments disappear after a specified period of time. Again, though, the wall is not really damaged.

If you have the processing power, you can simulate the player shooting a hole in the wall. However, the program has to keep track of how much force actually hits the wall. If the player is standing far away, his bullet shouldn't get through the wall. If he's close, the bullet should penetrate and leave a hole. That takes point masses.

More complex objects are often composed of point masses. Any time you're creating an object that's made of heavy things connected by much lighter things, you can use a collection of point masses. If you do, you can blow it up nicely. All the point masses fly apart in a satisfying way. Toss in some nifty explosion effects, and you've got a game.

Summary

That's it for the kinematics of point particles. We've come a long way! Starting with some basic ideas about point particles and speed, we built a system to model them and took it out for a spin. Excellent.

We haven't yet looked at forces like gravity, deflection, drag, and friction, but we will soon. At this point, we have a pretty realistic model of a ball, so let's leave it at that.