Tricks of the Windows Game Programming Gurus

Copyright © 2002 by Sams Publishing

International Standard Book Number: 0-672-32369-9

Library of Congress Catalog Card Number: 2001097517

Printed in the United States of America

First Printing: June 2002

05 04 03 02 4 3 2 1

Trademarks

Warning and Disclaimer

Executive Editor
Michael Stephens

Acquisitions Editor
Kim Spilker

Development Editor
Mark Renfrow

Managing Editor
Charlotte Clapp

Project Editor
George E. Nedeff

Copy Editor
Sean Medlock
Aaron Black
Howard Jones

Indexer
Erika Millen

Proofreader
Mike Henry

Technical Editor
David Franson

Team Coordinator
Lynn Williams

Multimedia Developer
Dan Scherf

Interior Designer
Gary Adair

Cover Designer
Gary Adair

Page Layout
Becky Harmon

TRICKS OF THE WINDOWS GAME PROGRAMMING GURUS

SECOND EDITION

André LaMothe

SAMS

201 West 103rd Street,
Indianapolis, Indiana 46290

CHAPTER 13

Playing God: Basic Physics Modeling

"Follow the white rabbit."
—Morpheus, The Matrix

There wasn't a whole lot of physics involved in the video games in the '70s and '80s. For the most part, games were shoot-'em-ups, search-and-destroy games, adventure games, and so on. However, beginning with the '90s and the "3D era," physics modeling became much more important. You simply can't get away with having objects in games move in non-realistic ways—the motion of the objects has to at least approximate what you'd expect in the real world. This chapter covers fundamental non-calculus-based physics modeling. Then, in the second volume I'll cover more rigid, calculus-based 2D and 3D modeling. Here are the topics you'll see in this chapter:

- Fundamental laws of physics
- Gravity
- Friction
- Collision response

- Forward kinematics
- Particle systems
- Playing God

If the universe is just a simulation in some unbelievably advanced computer, God is one heck of a programmer! The point is, the laws of physics work perfectly at all levels, from the quantum level to the cosmological level. The beauty of physics is that there aren't that many laws that govern the whole universe. Granted, our knowledge of physics and mathematics is that of a Cabbage Patch doll, but we do know enough to create computer simulations that can fool just about anyone.

Most computer simulations and games that use physics models use models based on standard Newtonian physics—a class of physics that works reasonably well on motion and objects that are within reasonable limits of size and mass. That is, speeds much less than the speed of light, and objects that are much bigger than a single atom, but much smaller than a galaxy. However, even modeling reality with basic Newtonian-based physics can take a lot of computing power. A simple simulation like rainfall or a pool table (if done correctly) would bring a Pentium IV to its knees.

Nonetheless, we have all seen rainfall and pool games on everything from the Apple II to the PC. So how did they do it? The programmers of these games understand the physics that they're trying to model and within the budget of the system they're programming on, create models that are close enough to what the player would expect in real life. This programming consists of a lot of tricks, optimizations, and most of all assumptions and simplifications about the systems that are being modeled. For example, it's a lot easier figuring out how two spheres will react after collision than it is to calculate the result of the collision of two irregular asteroids, thus a programmer might approximate all the asteroids in a game with simple spheres (as far as the physics calculations go).

In a state-of-the-art game, physics would take about 10,000 pages because it's not only the physics, but the math that needs to be learned, so I'm just going to cover some of the most fundamental physics models. From them you should be able to model everything you need in your first 2D/3D games. Most of this stuff should be more than familiar from High School physics—or Junior High School!

Fundamental Laws of Physics

Let's begin our physics journey by covering some of the basic concepts of physics and the properties of time, space, and matter. These fundamental concepts will give you a vocabulary to understand the more advanced topics that follow.

WARNING

Everything that I am about to say is not entirely true at the quantum or the cosmological level. However, the statements I'm going to make are true enough for our discussions. In addition, I'm going to lean toward the metric system since the English system is about 200 years antiquated and the only people that use it are the general population of the United States. The scientific community and the rest of the world all use the metric system. Seriously; 12 inches to a foot, 3 feet to a yard, 2 yards to a fathom, 5,280 feet to a mile. Was somebody smoking crack or what?

Mass (m)

All matter has mass. Mass is a measure of how much matter or actual atomic mass units. It doesn't have anything to do with weight. Many people have mass and weight confused. For example, they might incorrectly say that they weigh 75 kilograms (165 pounds) on Earth. First, kilograms (kg) are a metric measure of mass, that is, how much matter an object has. Pounds is a measure of force, or, more loosely, weight (mass in a gravity field). The measure of weight or force in the English system is a *pound (lb.)* and in the metric system is called a *Newton (N)*. Matter has no weight per se; it only can be acted upon by a gravitational field to produce what we refer to as weight. Hence, the concept of mass is a much more pure idea than weight (which changes planet to planet).

In games, the concept of mass is only used abstractly (in most cases) as a relative quantity. For example, I might set the spaceship equal to 100 mass units and the asteroid equal to 10,000. I could use kilograms, but unless I'm doing a real physics simulation it really doesn't matter. All I need to know is an object that has a mass of 100 has twice as much matter as an object that has 50 mass units. I'll revisit mass in a bit when I talk about force and gravity. Mass is the measure of how much matter an object is made of and is measured in kilograms in the metric system or—ready for this—*slugs* in the English system.

NOTE

Mass is also thought of as a measure of the resistance an object has to change in its velocity[md]Newton's First law. Basically, Newton's First law states that an object at rest remains at rest, and an object in motion remains in motion (at a constant velocity) until an exterior force acts on the object.

Time (t)

Time is an abstract concept. Think about it. How would you explain time without using time itself in the explanation? Time is definitely an impossible concept to convey without using circular definitions and a lot of hand waving. Luckily, everyone knows what time is, so I won't go into it, but I do want to talk about how it relates to time in a game.

Time in real life is usually measured in seconds, minutes, hours, and so forth. Or if you need to be really accurate then it's measured in milliseconds (ms 10^{-3} seconds), microseconds (µs 10^{-6}), nano (10^{-9}), pico (10^{-12}), femto (10^{-15}), etc. However, in a video game (most games), there isn't a really close correlation to real-time. Algorithms are designed more around the frame rate than real time and seconds (except for time modeled games). For example, most games consider one frame to be one virtual second, or in other words, the smallest amount of time that can transpire. Thus, most of the time, you won't use real seconds in your games and your physics models, but virtual seconds based on a single frame as the fundamental time step.

On the other hand, if you're creating a really sophisticated 3D game then you probably will use real time. All the algorithms in the game track in real time, and invariant of the frame rate, they adjust the motion of the objects so that a tank can move at, say, 100 feet per second even if the frame rate slows down to 2 frames per second or runs at 60 frames per second. Modeling time at this level of accuracy is challenging, but absolutely necessary if you want to have ultra-realistic motion and physical events that are independent of frame rate changes. In any case, we'll measure time in seconds (s) in the examples or in virtual seconds, which simply means a single frame.

Position (s)

Every object has an (x,y,z) position in 3D space or an (x,y) position in 2D space or an x in 1D or linear space (also sometimes referred to as an s). Figure 13.1 shows examples of all these dimensional cases. However, sometimes it's not clear what the position of an object is even if you know where it is. For example, if you had to pick one single point that locates the position of a sphere then you would probably pick its center as shown in Figure 13.2. But what about a hammer? A hammer is an irregular shape, so most physicists would use its *center of mass,* or balancing point, as the position to locate it, as shown in Figure 13.3.

FIGURE 13.1
The concept of position.

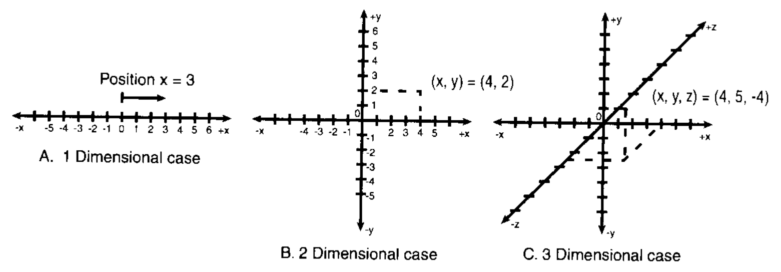


FIGURE 13.2
Picking the center.

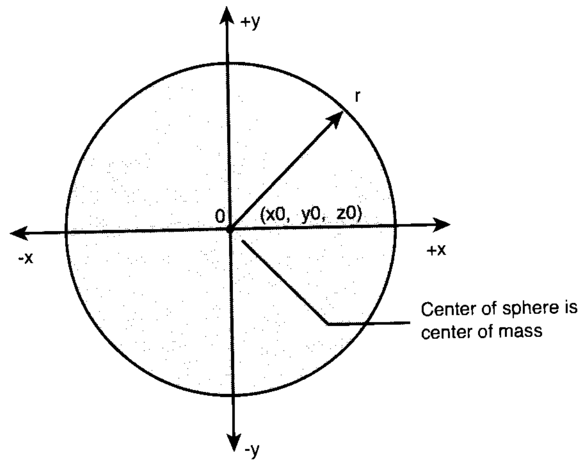
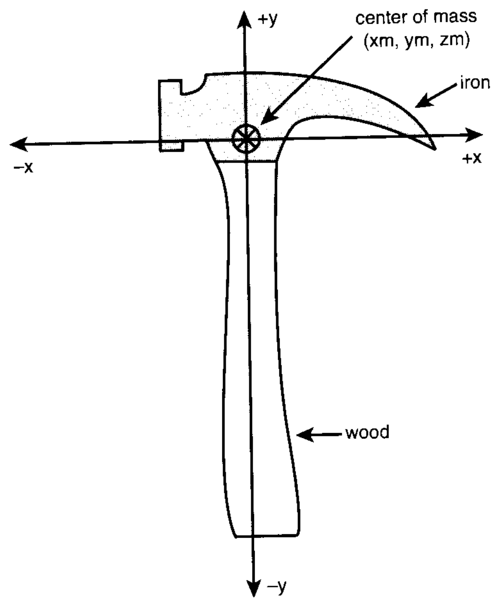


FIGURE 13.3
Picking the center of an irregular object.



The concept of position and the physically correct location of the position when it comes to games is usually rather lax. Most game programmers place a bounding box, circle, or sphere around all the game objects as shown in Figure 13.4 and simply use the center of the bounding entity as the center of the object. This works for most cases, where most of the mass of the object is located at the center of the object by luck, but if that's not the case then any physics calculations that use this artificial center will be incorrect.

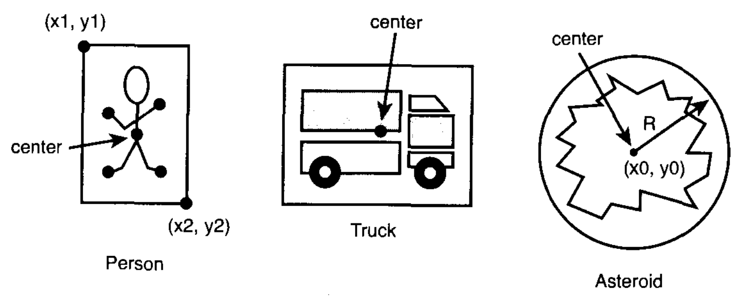


FIGURE 13.4
Collision contour shapes.

The only way to solve the problem is to pick a better center that takes the virtual mass of the object into consideration. For example, you could create an algorithm that scanned the pixels making up the object and the more pixels that were in an area, the more that area would be weighted to be the center. Or if the object is a polygon-based object then you could attach a weight to each vertex and compute the real center of mass of the object. Assuming that there are n vertices and each vertex position is labeled by (x_i,y_i) with a mass of m_I, then the center of mass is

$$Xc = \frac{\sum_{i=0}^{n} x_i * m_i}{\sum_{i=0}^{n} m_i}$$

$$Yc = \frac{\sum_{i=0}^{n} y_i * m_i}{\sum_{i=0}^{n} m_i}$$

MATH

The Σf_i symbol just means "sum of." It's like a for loop that sums the values f_i for each value of *i*.

Velocity (v)

Velocity is the instantaneous rate of speed of an object and is usually measured in meters per second (m/s) or in the case of the automobile, miles per hour, or mph. Whatever units you prefer, velocity is the change in position per change in time. Stated mathematically in a 1 dimensional case, this reads:

Velocity = v = ds/dt.

In other words, the instantaneous change in position (ds) with respect to time (dt). As an example, say you are driving down the road and you just drove 100 miles in one hour, then your average velocity would be

v = ds/dt = 100 miles/1 hour = 100 mph.

In a video game the concept of velocity is used all the time, but again the units are arbitrary and relative. For example, in a number of the demos I have written I usually move objects at 4 units in the x- or y-axis per frame with code something like the following:

```
x_position = x_position + x_velocity;
y_position = y_position + y_velocity;
```

That translates to 4 pixels/frame. But frames aren't time, are they? Actually, they are as long as the frame rate stays constant. In the case of 30 fps, which is equal to 1/30th of a second per frame, the 4 pixels per frame translate to:

```
Virtual Velocity = 4 pixel / (1/30) seconds
                 = 120 pixels per second.
```

Hence, the objects in our game have been moving with velocities measured in pixels/second. If you wanted to get crazy then you could estimate how many virtual meters were in one pixel in your game world and perform the computation in meters/second in cyberspace. In either case, now you know how to gauge where an object will be at any given time or frame if you know the velocity. For example, if an object was currently at position x_0 and it was moving at 4 pixels/frame, and 30 frames go by, the object would be at

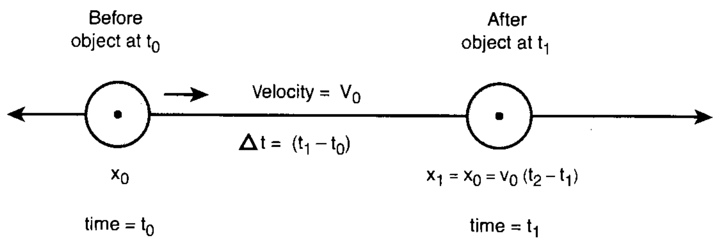
```
new position = x0 + 4 * 30 = x0 + 120 pixels.
```

This leads us to our first important basic formula for motion:

```
New Position = Old Position + Velocity * Time
             = xt = x0 + v*t.
```

This formula states that an object moving with velocity *v* that starts at location x_0 and moves for *t* seconds will move to a position equal to its original position plus the velocity times the time. Take a look at Figure 13.5 to see this more clearly. As an example of constant velocity I have created a demo `DEMO13_1.CPP|EXE (16-bit version, DEMO13_1_16B.CPP|EXE)` that moves a fleet of choppers from left to right on the screen.

FIGURE 13.5
Constant velocity motion.



TRICK

I always amaze my friends by telling them how long it will take to get to an off-ramp or some other location when we're in the car. The trick is simple; just look at the speed and use the fact that at 60 mph it takes 1 minute to go 1 mile. So if the driver is driving 60 and the off ramp is in 2 miles then it will take 2 minutes. On the other hand, if the on ramp is in 3.5 miles then it would take 3 minutes and 30 seconds. And if the driver isn't driving 60 mph then use the closest plus or minus 30 mph. For example, if they're going 80 then do you calculations with 90 mph (1.5 miles per minute) and then shrink your answer a bit.

Acceleration (a)

Acceleration is similar to velocity, but it is the measure of the rate of change of velocity rather than the velocity itself. Take a look at Figure 13.6; it illustrates an object moving with a constant velocity and one with a changing velocity. The object moving with a constant velocity has a flat line (slope of 0) for its velocity as a function of time, but the accelerating object has a slope of non-zero because its velocity is changing as a function of time.

FIGURE 13.6
Velocity vs. acceleration.

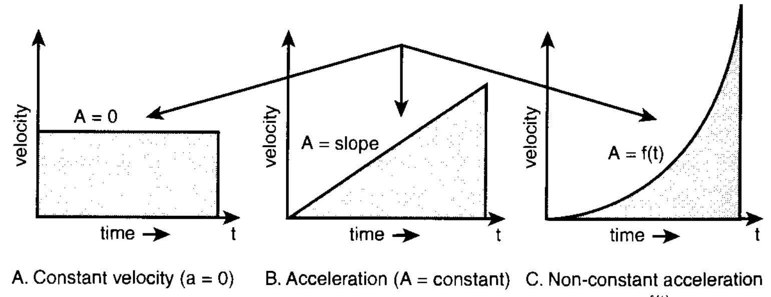


Figure 13.6 illustrates constant acceleration. There is also non-constant acceleration. In this case the line would be a curve in part C of Figure 13.6. Pressing the accelerator in your car will give you the feeling of non-constant acceleration and jumping off a

cliff will give you the feeling of constant acceleration. Mathematically, acceleration is the rate of change of velocity with respect to time:

```
Acceleration = a = dv/dt.
```

The units of acceleration are a little weird. Since velocity is already in units of distance per second, acceleration is in units of distance per second*second, or in the metric system, m/s^2. If you think about this it makes sense because acceleration is the change of velocity (m/s) per second. Furthermore, our second motion law relates the velocity, time, and acceleration, which states that the new velocity at some time, *t*, in the future equals the starting velocity plus the acceleration times the amount of time the object has been accelerating for:

```
New Velocity  = Old Velocity + Acceleration * Time
              = vt = v0 + a*t.
```

Acceleration is a fairly simple concept and can be modeled in a number of ways, but let's take a look at a simple example. Imagine that an object is located at (0,0) and it has a starting velocity of 0. If you were to accelerate it at a constant velocity of 2 m/s, you could figure out the new velocity each second simply by adding the acceleration to the last velocity, as shown in Table 13.1.

TABLE 13.1 Velocity as a Function of Time for Acceleration 2 m/s^2

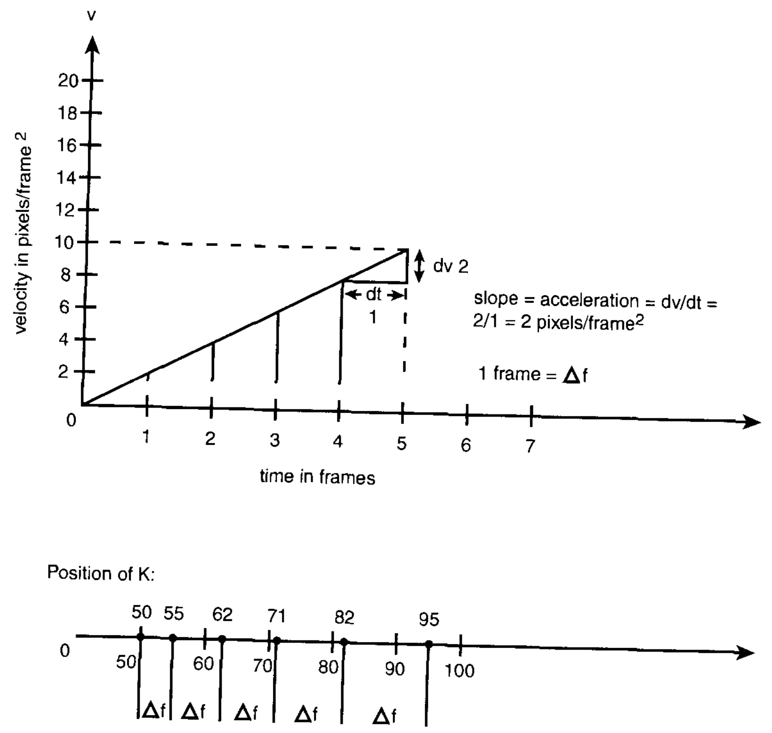
Time (t = s)	*Acceleration (v = m/s)*	*Velocity (a = m/s²)*
0	2	0
1	2	2
2	2	4
3	2	6
4	2	8
5	2	10

Taking the data in the table into consideration, the next step is to figure out the relationship between position, velocity, acceleration, and time. Unfortunately, this takes a bit of calculus, so I'll just give you the results in terms of position at some time t:

```
xt = x0 + v0*t + 1/2*a*t2
```

This equation states that the position of an object at some time t is equal to its initial position, plus its initial velocity, times time, plus one half the acceleration, times time squared. The $1/2*a*t^2$ term is basically the time integral of the velocity. Let's see if you can use the equation in your game world of pixels and frames. Refer to Figure 13.7.

FIGURE 13.7
An acceleration in pixels/frame^2.



Assume these initial conditions: The object is at x=50 pixels, the initial velocity is 4 pixels/frame, and the acceleration is 2 pixels/frame^2. Finally, assume that these are the conditions at frame 0. To find the position of the object at any time in C/C++, use the following:

```
x = 50 + 4*t + (0.5)*2*t*t;
```

Where t is simply the frame number. Table 13.2 lists some examples for t = 0,1,2...5.

TABLE 13.2 An Object Moving with Constant Acceleration

Time/Frame (t)	*Position (x)*	*Delta (x)=x_t-x_{t-1}*
0	50	0
1	50+4*1+(0.5)*2*1^2 = 55	5
2	50+4*2+(0.5)*2*2^2 = 62	7
3	50+4*3+(0.5)*2*3^2 = 71	9
4	50+4*4+(0.5)*2*4^2 = 82	11
5	50+4*5+(0.5)*2*5^2 = 95	13

There's a lot of interesting data in Table 13.2, but maybe the most interesting data is the change in position each time the frame is constant and equal to 2. Now this doesn't mean that the object moves 2 pixels per frame, it means that the change in motion each frame gets larger or increases by 2 pixels. Thus on the first frame the object moves 5 pixels, then on the next frame it moves 7, then 9, 11, then 13, and so on. And the delta between each change in motion is 2 pixels, which is simply the acceleration!

The next step is to model acceleration with C/C++ code. Basically, here's the trick: You set up an acceleration constant and then with each frame you add it to your velocity. This way you don't have to use the long equation shown earlier—you simply translate your object with the given velocity. Here's an example:

```
int acceleration = 2, // 2 pixels per frame
    velocity     = 0, // start velocity off at 0
    x            = 0; // start x position of at 0 also
// ...
// then you would execute this code each
// cycle to move your object
// with a constant acceleration

// update velocity
velocity+=acceleration;

// update position
x+=velocity;
```

NOTE

Of course this example is one-dimensional. You can upgrade to two dimensions simply by adding a y position (and y velocity and acceleration if you wish).

To see acceleration in action, I have created a demo named DEMO13_2.CPP|EXE (16-bit version, DEMO13_2_16B.CPP|EXE) that allows you to fire a missile that accelerates as it moves forward. Press the spacebar to fire the missile, the up and down arrow keys to increase and decrease the acceleration factor, and the A key to toggle the acceleration on and off. Look at the difference acceleration makes to the motion of the missile and how acceleration gives the missile a sense of "mass."

Force (F)

One of the most important concepts in physics is *force*. Figure 13.8 depicts one way to think of force. If an object with mass *m* is sitting on a table with gravity pulling it toward the center of the Earth, the acceleration is a=g (force of gravity). This gives the mass *m* weight and if you try to pick it up you will feel a pain in your lower back.

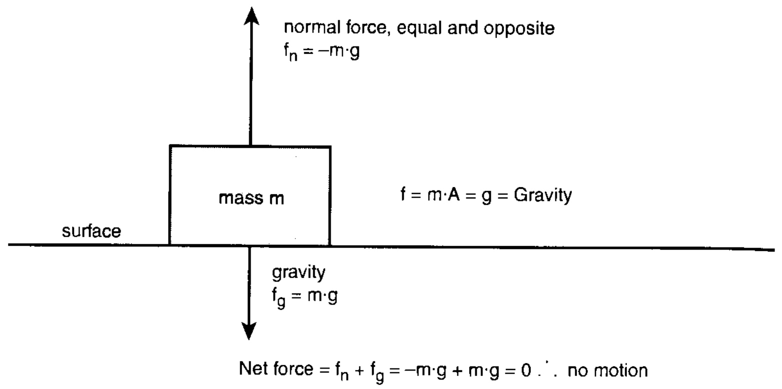

normal force, equal and opposite
$f_n = -m \cdot g$
mass m
$f = m \cdot A = g$ = Gravity
surface
gravity
$f_g = m \cdot g$
Net force = $f_n + f_g = -m \cdot g + m \cdot g = 0$ ∴ no motion

FIGURE 13.8
Force and weight.

The relationship between force, mass, and acceleration is Newton's Second Law:

F=m*a

In other words, the force exerted on an object is equal to its mass times the acceleration of the object. Or, rearranging terms:

a = F/m

This states that an object will accelerate an amount equal to the force you place on it divided by its mass. Now let's talk about the units of measure. But instead of just blurting it out, let's see where it comes from in the metric system. Force is equal to mass times acceleration or kilograms multiplied by m/s^2 (m stands for meters, not mass). Hence, a possible unit of force is

F = kg*m/s^2—kilogram-meters per second squared

This is a bit long, so Newton just called it—a *Newton (N)*. As an example, imagine that a mass m equal to 100 kg is accelerating at a rate of 2 m/s^2. The force that is being applied to the mass is exactly equal to F = m*a = 100 kg * 2 m/s^2 = 200N.

This gives you a bit of a feel for a Newton. A 100 kg mass is roughly equivalent to the force of 220 lbs. on Earth, and 1 m/s^2 is a good accelerating run.

In a video game the concept of force is used for many reasons, but a few that come to mind are

- You want to apply artificial forces like explosions to an object and compute the resulting acceleration.
- Two objects collide and you want to compute the forces on each.
- A game weapon only has a certain force, but it can fire different virtual mass shells and you want to find the acceleration the shells would experience when fired.

Forces in Higher Dimensions

Of course forces can act in all three dimensions, not just in a straight line. For example, take a look at Figure 13.9, which depicts three forces acting on a particle in a 2D plane. The resulting force that the particle p "feels" is simply the sum of the forces that are acting on it. However, in this case it's not as simple as adding scalar numbers together since the forces are vectors. Nevertheless, vectors can be decomposed into their x, y, and z components and then the forces acting in each axis can be computed. The result is the sum of the forces acting on the particle.

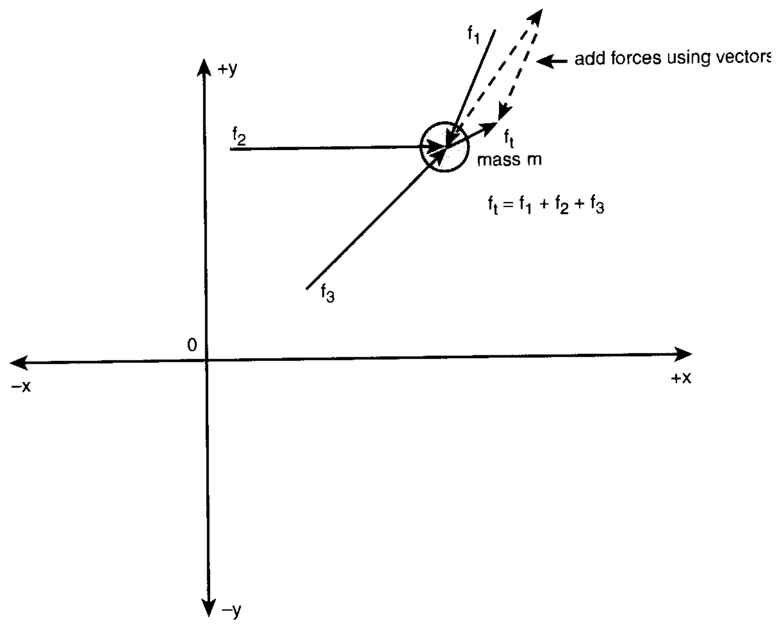


FIGURE 13.9
Forces acting on a particle in 2D.

In the example shown in Figure 13.9 there are three forces; F_1, F_2, and F_3. The final force F_{final}=<fx, fy> that object p feels is simply the sum of these forces:

```
fx = f1x + f2x + f3x
fy = f1y + f2y + f3y
```

Plugging in the values from the example in the figure, you get

```
fx = (x+x+x) = x.x
fy = (y+y+y) = y.y
```

With that in mind, it doesn't take much to deduce that in general the final force F on an object is just the vector sum of forces, or mathematically:

$F_{final} = F_1+F_2+\ldots+F_n$

Where each force F_i can have 1, 2, or 3 components, that is, each vector can be a 1D (scalar), 2D, or 3D.

Momentum (P)

Momentum is one of those quantities that's hard to define verbally. It's basically the property that objects in motion have. Momentum was invented as a measure of both the velocity and mass of an object. Momentum is defined as the product of mass (m) and the velocity (v) of an object:

P = m*v

And the units of measure are kg*m/s, kilogram-meters per second. Now the cool thing about momentum is its relationship to force—watch this:

F = m*a

or substituting, p for m:

F = (p*a)/v

But, a = dv/dt, thus:

$$F = \frac{p*dv/dt}{v} = \frac{d(p)*v}{dt*v} = dp/dt$$

Or in English, force is the time rate change of momentum per unit time. Hmmm... interesting. That means if the momentum of an object changes a lot then so must the force acting on the object. Now here's the clincher. A pea can have as much momentum as a train—how? A pea may have mass of 0.001 kg and a train have a mass of 1,000,000 kg. But if the train is going 1 m/s and the pea is going 100,000,000,000 m/s (that's one fast pea) then the pea will have more momentum:

$m_{pea}*v_{pea}$ = 0.001 kg * 10,000,000,000 m/s = 10,000,000 kg*m/s
$m_{train}*v_{train}$ = 1,000,000 kg * 1 m/s = 1,000,000 kg*m/s

And thus if either of these objects came to an abrupt stop, hit something for example, that object would feel a whole lot of force! That's why a bee hitting you on a motorcycle is so dangerous. It's not the mass of the bee, but the velocity of the bee that gets you in this example. The resulting momentum is huge and can literally throw a 200-pound guy off the bike.

This brings us to *conservation of momentum* and *momentum transfer*.

NOTE

I was on an FZR600 one time, going about 155 mph, and a bee hit my visor. Not only did it crack the visor, but it felt like someone threw a baseball at me! Lesson to be learned—only speed in designated bee-free areas!

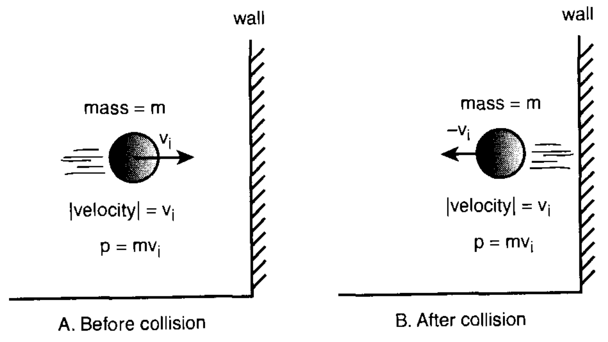
The Physics of Linear Momentum: Conservation and Transfer

Now that you have an idea about what momentum is, let's talk briefly about some of the physics involved when an object strikes another. Later I will go into true collision response in more depth, but for now let's keep it simple.

Remember in the game *DOOM* when you shot a barrel, it would explode and cause the barrels and/or bad guys in the area to move and/or explode? Wasn't that cool splattering a bad guy against a wall! That was just momentum transfer, but believe me, doing it correctly is no picnic!

In general, if two objects collide there are two things that can happen: a perfectly elastic collision and a not so perfectly elastic collision. In a perfectly elastic collision, as shown in Figure 13.10, a ball hits a wall with velocity $v_{i,}$ and when it bounces off it still has velocity $v_{i.}$. Thus, the momentum was conserved. Therefore, the collision was totally elastic. However, in real life this isn't usually the case. Most collisions aren't elastic, they are less than perfect. When collisions that are less than perfectly elastic occur, some energy is converted into heat, work to deform the objects colliding, etc. Thus the resulting momentum of the object(s) after collision is less than when the collision started.

FIGURE 13.10
A perfectly elastic collision of a ball and wall.



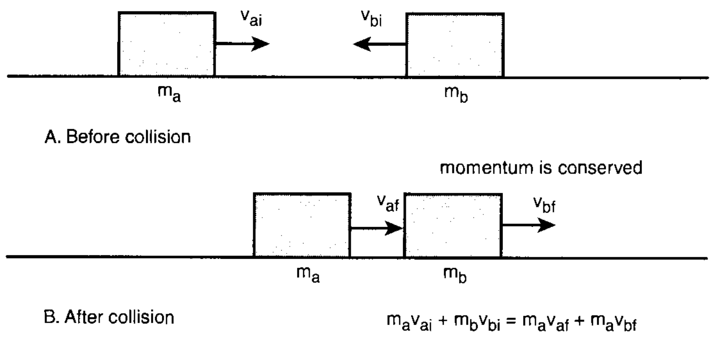
However, I'm not interested in this kind of imperfect world. Since we are gods of the virtual world, we might as well make things easy. Hence, I'm going to talk about perfectly elastic collisions in 1-dimension right now, then later we'll do it in 2D and get medieval with the math! Let's begin.

Figure 13.11 has two block objects A, B with mass m_a and m_b and velocity v_{ai} and $v_{bi,}$ respectively. The question is what happens after they hit, assuming no friction (we'll get to that later) and a perfectly elastic collision? Well, let's start with the *conservation of momentum.* It states that the total momentum before the collision will be the same as after the collision. Or mathematically:

Equation 1: Conservation of momentum

$$m_a*v_{ai} + m_b*v_{bi} = m_a*v_{af} + m_b*v_{bf}$$

FIGURE 13.11
The collision response of two blocks in 1D.



All right, you know m_a, m_b, v_{ai}, and v_{bi}, but you want the final velocities v_{af} and v_{bf}. The problem is that you only have one equation and two unknowns. This is obviously a bad thing. If you knew the velocity of one of the masses, you could figure the other one out. But, is there a way to figure out both velocities without any further knowledge? The answer is yes! You can use another property of physics to come up with another equation. The property is the *conservation of kinetic energy.*

Kinetic energy is like momentum, but is independent of direction. It's like a magnitude of sorts that gauges the amount of total energy that a system has. Now, energy is the ability to do work and that's all I'm going to say, we are getting a little too quantum here. However, computing kinetic energy is trivial, the formula is

Equation 2: Kinetic energy

$$ke = {}^1/_2*m*v^2$$

Momentum was just m*v, so you should see that kinetic energy is very similar, but it's always positive and measured in $kg*m^2/s^2$, which in the Meter-Kilogram-Second system we just call Joules (J). The cool part is that the kinetic energy of any system is always the same before and after a collision, elastic, or not. Of course you would have to compute the energies lost due to deformation, heat, etc. to account for all the energy, but when assuming a perfectly elastic collision, the kinetic energy before and after can be computed by just knowing the velocities of the objects:

Equation 3: Total kinetic energy of collision

$$*m_a*v_{ai}^2 + {}^1/_2*m_b*v_{bi}^2 = {}^1/_2*ma*v_{af}^2 + {}^1/_2*m_b*v_{bf}^2$$

Combining this with equation 1 results in:

$$m_a*v_{ai} + m_b*v_{bi} = m_a*v_{af} + m_b*v_{bf}$$
$$*m_a*v_{ai}^2 + \_*m_b*v_{bi}^2 = {}^1/_2*ma*v_{af}^2 + {}^1/_2*m_b*v_{bf}^2$$

At this point, you have two equations and two unknowns and both v_{af} and v_{bf} can be computed; however, the math is rather complex, so I will just give you the results:

Equation 4: The final velocities of each ball

$$v_{af} = (2*m_b*v_{bi} + v_{ai}*(m_a - m_b))/(m_a + m_b)$$
$$v_{bf} = (2*m_a*v_{ai} - v_{bi}*(m_a - m_b))/(m_a + m_b)$$

Finally, referring back to Figure 13.11, you can figure out the final velocities after the collision of the blocks:

$$m_a = 2 \text{ kg}$$
$$m_b = 3 \text{ kg}$$

$$v_{ai} = 4 \text{ m/s}$$
$$v_{bi} = -2 \text{ m/s}$$

Therefore,

$$\begin{aligned}
v_{af}& \\
&= (2*m_b*v_{bi} + v_{ai}*(m_a - m_b))/(m_a + m_b)\\
&= (2*3*(-2) + 4*(2 - 3))/(2 + 3)\\
&= -3.2 \text{ m/s}
\end{aligned}$$

$$\begin{aligned}
v_{bf} &= (2*m_a*v_{ai} - v_{bi}*(m_a - m_b))/(m_a + m_b)\\
&= (2*2*4 - (-2)*(2 - 3))/(2 + 3)\\
&= 2.4 \text{ m/s}
\end{aligned}$$

Interestingly enough, object A had so much momentum it turned object B around and they both went off in the positive X direction as shown in part B of Figure 13.11.

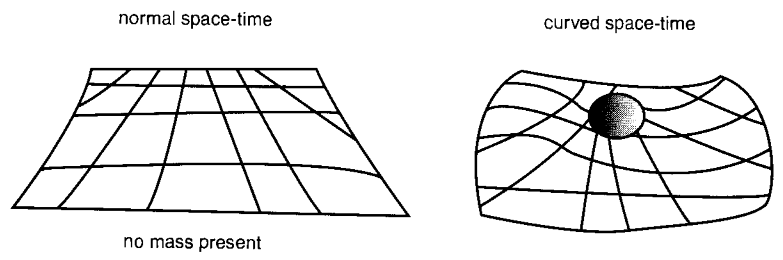
What you just did shows how to use momentum and kinetic energy to help solve kinetic/dynamic problems. However, they get much more complex in two and three dimensions. The study of such collisions is called *collision response*, and it's covered later in the chapter, along with the complete 2D results for perfect and imperfect collisions! For now, though, just think about momentum.

Modeling Gravity Effects

One of the most common effects that a game programmer needs to model in a game is that of gravity. Gravity is the force that attracts every object in the universe to every other. It is an invisible force and unlike magnetic fields can't be blocked.

In reality, gravity isn't really a force. That's simply how we perceive it. Gravity is caused by the curvature of space. When any object is positioned in space it creates a bending of the surrounding space, as shown in Figure 13.12. This bending creates a potential energy difference and hence any object near the gravity well "falls down" toward the object—weird, huh? That's really what gravity is. It's a manifestation of the bending of the space-time fabric.

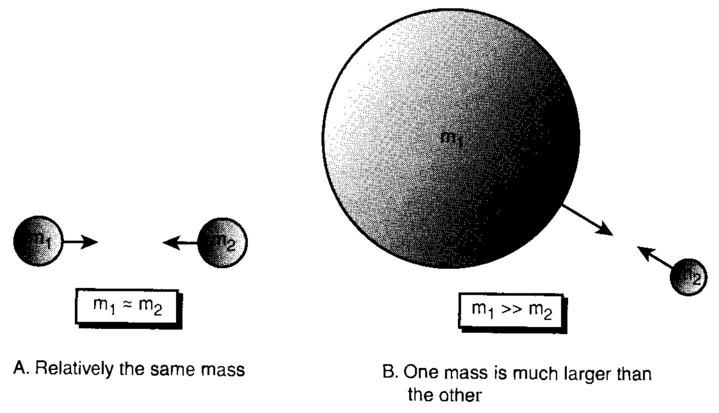
FIGURE 13.12
Gravity and space-time.



You won't need to worry about space-time curvature and what gravity really is; you just want to model it. There are really two cases that you need to consider when modeling gravity, as shown in Figure 13.13:

- Case 1: Two or more objects with relatively the same mass.
- Case 2: Two objects where the mass of one object is much greater than the other.

FIGURE 13.13
The two general cases of gravity.


m_1
m_2
$m_1 \approx m_2$
m_1
m_2
$m_1 >> m_2$
A. Relatively the same mass
B. One mass is much larger than the other

Case 2 is really a sub-case of Case 1. For example, in school you may have learned that if you drop a baseball and a refrigerator off a building they both fall at the same rate. The truth of the matter is they don't, but the difference is so infinitesimal (on the order 10^{-24}) that you could never see the difference. Of course, there are other forces that might make a difference, like wind shear and friction, hence a baseball is going to fall faster than a piece of paper because the paper is going to feel a lot of wind resistance.

Now that you know a little bit about what gravity is, let's take a look at the math behind it. The gravitational force between any two objects with mass m_1 and m_2 is

$F = G*m_1*m_2 / r^2$.

where G is the gravitational constant of the universe equal to $6.67x10^{-11}$ $N*m^2 * kg^{-2}$. Also, the masses must be in kilograms and the distance r in meters. Say that you want to find out what the gravitational attraction is between two average sized people of 70 kg (155 lbs.) at a distance of 1 meter:

$F = 6.67x10^{-11}*70kg*70kg/(1 m)^2 = 3.26x10^{-7}$ N.

That's not much is it? However, let's try the same experiment with a person and the planet Earth at 1 meter given that the Earth has a mass of $5.98x10^{24}$ kg:

$F = 6.67x10^{-11}*70 kg*5.98x10^{24} kg/(1 m)^2 = 2.79x10^{16}$ N.

Obviously, 10^{16} Newtons would crush you into a pancake, so you must be doing something wrong. The problem is that you're assuming that the Earth is a point mass that is 1.0 meters away. A better approximation would be to use the radius of the Earth (the center of mass) as the distance, which is $6.38x10^6$ m:

MATH

You may assume that any spherical mass of radius r is a point mass as long as the matter the sphere is made of is homogenous and any calculations must place the other object at a distance greater than or equal to r.

$$F = 6.67x10^{-11} * 70 kg * 5.98x10^{24} kg / (6.38x10^6 m)^2$$
$$= 685.93 N.$$

Now that seems more reasonable. As a sanity check, on Earth 1 lb. is equal to 4.45 N, so converting the force to lbs. produces

```
685.93 N / (4.45 N / 1 lb.) = 155 lbs.
```

And this was the starting weight! Anyway, now that you know how to compute the force between two objects you can use this simple model in games. Or course, you don't have to use the real gravity constant $G = 6.67x10^{-11}$, you can use whatever you like—remember, you are god. The only thing that's important is the form of the equation that states that the gravity between two objects is proportional—to a constant times the product of their masses divided by the distance squared between the objects' centers.

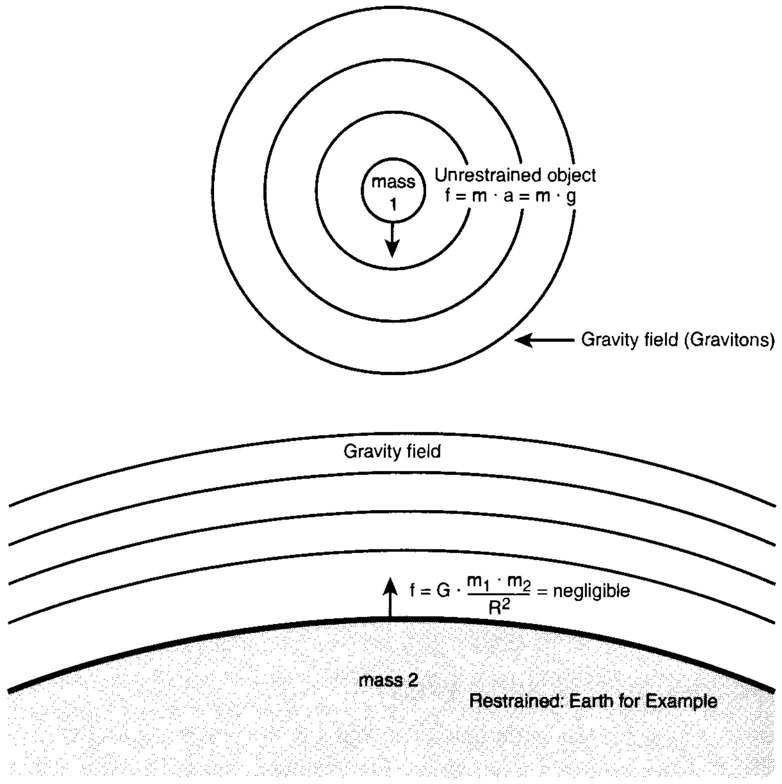
Modeling a Gravity Well

By using the formulation explained in the preceding section, you might, say, model a black hole in a space game. For example, you might have a ship that is flying around on the screen near a black hole, and want the ship to get sucked in if it gets too close. Using the gravitational equation is a snap. You would make up a constant G that worked well in the virtual game world (based on screen resolution, frame rate, etc.) and then simply set an arbitrary mass for the ship and one for the black hole that was much larger. Then you would figure out the force and convert the force to acceleration with F=m*a. You would simply vector or fly the ship directly toward the black hole each frame. As the ship got closer the force would increase until the player couldn't get free!

As an example of a black hole simulation (which is nothing more than two masses, one much larger than another) take a look at `DEMO13_3.CPP|EXE (16-bit version, DEMO13_1_16B.CPP|EXE)`. It's a space simulator that allows you to navigate a ship around the screen, but there's a black hole in the middle that you have to deal with! Use the arrows keys to control the ship. Try to see if you can get into an orbit!

The next use of gravity in games is to simply make things fall from the sky or off buildings at the proper rate. This is really the special case that we talked about before, that is, one object has a mass much greater than the other. However, there's one more constraint and that is that one object is fixed—the ground. Take a look at Figure 13.14; it depicts the situation that I'm describing.

FIGURE 13.14
Gravitational attraction.


mass 1
Unrestrained object
f = m · a = m · g
Gravity field (Gravitons)
Gravity field
f = G · m1 · m2 / R2 = negligible
mass 2
Restrained: Earth for Example

In this case, there are a number of assumptions that we can make that will make the math work out simpler. The first is that the acceleration due to gravity is constant for the mass that is being dropped, which is equal to 9.8 m/s^2 or 32 ft/s^2. Of course, this isn't really true, but is true to about 23 decimal places. If we know that the acceleration of any object is simply 9.8 m/s^2 then we can just plug that into our old motion equation for velocity or position. Thus, the formula for velocity as a function of time with Earth gravity is

$V(t) = v_0 + 9.8\ m/s^2 * t.$

And position is

$y(t) = y_0 + v_0 * t + 1/2 * 9.8 m/s^2 * t^2.$

In the case of a ball falling off a building we can let the initial position x_0 be equal to 0 and the initial velocity v_0 also equal 0. This simplifies the falling object model to

$y(t) = 1/2 * 9.8 m/s^2 * t^2.$

Furthermore, you are free to change the constant 9.8 to anything you like and t represents the frame number (virtual time) in a game. Taking all that into consideration, here's how you would make a ball fall from the top of the screen:

```
int y_pos      = 0, // top of screen
    y_velocity = 0, // initial y velocity
    gravity    = 1; // do want to fall too fast

// do  gravity loop until object hits
// bottom of screen at SCREEN_BOTTOM
while(y_pos < SCREEN_BOTTOM)
     {
     // update position
     y_pos+=y_velocity;

     // update velocity
     y_velocity+=gravity;
     } // end while
```

TIP

I used the velocity to modify the position rather than modifying the position directly with the position formula. This is simpler.

You may be asking how to make the object fall with a curved trajectory. This is simple—just move the x position at a constant rate each cycle and the object will seem like it was thrown off rather then just dropped. The code to do this follows:

```
int y_pos      = 0, // top of screen
    y_velocity = 0, // initial y velocity
    x_velocity = 2, // constant x velocity
    gravity    = 1; // do want to fall too fast
// do  gravity loop until object hits
// bottom of screen at SCREEN_BOTTOM
while(y_pos < SCREEN_BOTTOM)
     {
     // update position
     x_pos+=x_velocity;
     y_pos+=y_velocity;
// update velocity
y_velocity+=gravity;
} // end while
```

Modeling Projectile Trajectories

Falling objects are fairly exciting, but let's see if we can do something a little more appropriate for video game programming! How about computing trajectory paths? Take a look at Figure 13.15, which shows the general setup for the problem. We have a ground plane, call it y=0, and a tank located at x=0, y=0, with a barrel pointed at an angle of inclination θ (theta) with the x-axis. The question is, if we fire a projectile with mass m at a velocity v_I, what will happen?

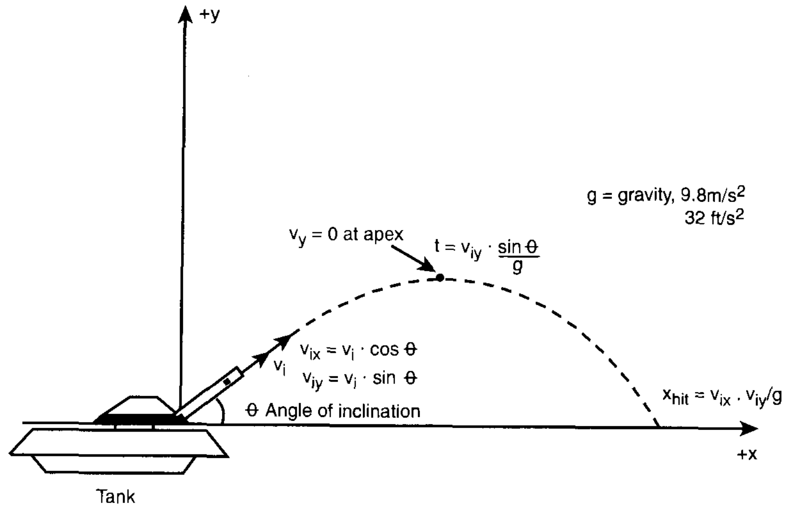

+y
g = gravity, 9.8m/s2
32 ft/s2
v_y = 0 at apex
$t = v_{iy} \cdot \frac{\sin\theta}{g}$
$v_{ix} = v_i \cdot \cos\theta$
v_i
$v_{iy} = v_i \cdot \sin\theta$
θ Angle of inclination
$x_{hit} = v_{ix} \cdot v_{iy}/g$
+x
Tank

FIGURE 13.15
The trajectory problem.

We can solve the problem by breaking it up into its x,y components. First, let's break the velocity into an (x,y) vector:

V_{ix} = V*cos θ
V_{iy} = V*sin θ

Trust me.

Okay, now forget about the x part for a minute, and think about the problem. The projectile is going to go up and down and hit the ground. How long will this take? Take a look at our previous gravity equations:

$V(t) = v_0 + 9.8\ m/s^2 * t.$

The position for the y-axis is

$y(t) = y_0 + v_0 * t + 1/2 * 9.8m/s^2 * t^2.$

The first one tells us the velocity relative to time. That's what we need. We know that when the projectile reaches its maximum height, the velocity will be equal to 0. Furthermore, the amount of time that the projectile takes to reach this height will be the same amount of time it takes to fall to the ground again. Take a look at Figure 13.15. Plugging in our values for initial y velocity of our projectile and solving for time t, we have

$V_y(t) = V_{iy} - 9.8\ m/s^2 * t$

Note that I flipped the sign of the acceleration due to gravity because down is negative and matters in this case, and in general, when the velocity equals 0:

$0 = V * \sin\theta - a * t$ (a is just the acceleration)

Solving for time t, we get

$t = V_{iy} * (\sin\theta)/a$

Now the total time of flight is just time up + time down which equals t+t=2*t since the projectile must go up, then down. Therefore, we can revisit the x component now. We know that the total flight time is 2*t and we can compute t from (V_{iy} * (sin θ)/a). Therefore, the distance that the projectile travels in the x-axis is just

$X(t) = V_{ix} * t$

Plugging in our values, this results in

$x_{hit} = (V*\cos\theta) * (V*(\sin\theta)/a)$

or

$x_{hit} = V_{ix} * V_{iy}/a$

Neat, huh?

MATH

Note that I replaced the 9.8 value of acceleration with a. I did this to reenforce that the acceleration is just a number, and you can make it whatever you wish.

That's the physics behind everything, but how do you model it in a program? Well, all you do is apply constant x-axis velocity to the projectile and gravity in the y-axis and test for when the projectile hits the ground or something else. Of course, in real life the X and Y velocities would diminish due to air resistance, but throwing that out the algorithm I just described works great. Here's the code to do it:

```
// Inputs
float x_pos      = 0, // starting point of projectile
      y_pos      = SCREEN_BOTTOM, // bottom of screen
      y_velocity = 0, // initial y velocity
      x_velocity = 0, // constant x velocity
      gravity    = 1, // do want to fall too fast
```

```
        velocity   = INITIAL_VEL, // whatever
        angle      = INITIAL_ANGLE; // whatever, must be in radians

// compute velocities in x,y
x_velocity = velocity*cos(angle);
y_velocity = velocity*sin(angle);


// do projectile loop until object hits
// bottom of screen at SCREEN_BOTTOM
while(y_pos < SCREEN_BOTTOM)
     {
     // update position
     x_pos+=x_velocity;
     y_pos+=y_velocity;

     // update velocity
     y_velocity+=gravity;
     } // end while
```

That's all there is to it! If you want to add a wind force, just model it as a small acceleration in the direction opposing the X-motion, and assume that the wind force creates a constant acceleration against the projectile. As a result, you simply need to add this line of code in the projectile loop:

```
x_velocity-=wind_factor;
```

Where `wind_factor` would be something like 0.01—something fairly small.

As a demo of all this trajectory stuff, check out `DEMO13_4.CPP|EXE (16-bit version, DEMO13_1_16B.CPP|EXE)` on the CD. A screen shot is shown in Figure 13.16. The demo allows you to aim a virtual cannon and fire a projectile.

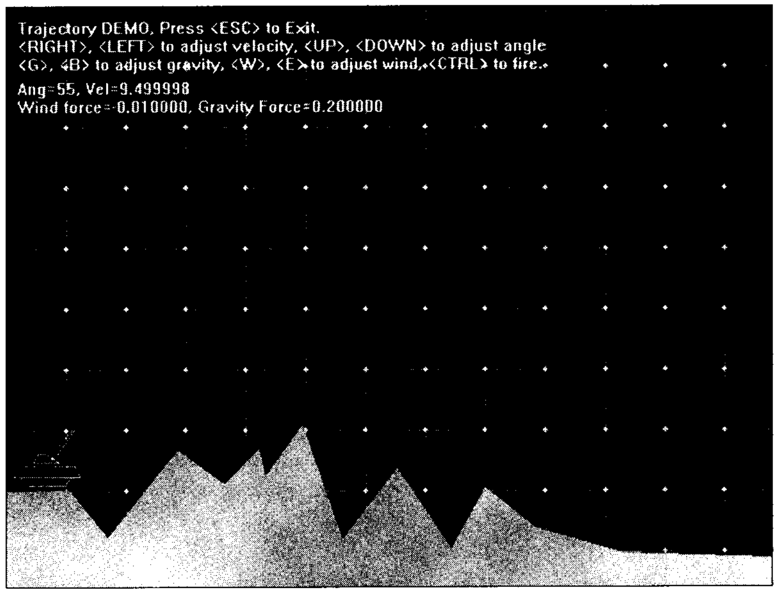
Here are the controls:

Key	*Action*
Up, down	Controls the angle of the tank's cannon.
Right, left	Controls the velocity of the projectile.
G, B	Controls the gravity.
W, E	Controls the wind speed.
Ctrl	Fires the cannon!

The Evil Head of Friction

The next topic of discussion is friction. Friction is any force that retards or consumes energy from another system. For example, automobiles use internal combustion to operate; however, a whopping 30–40 percent of the energy that is produced is eaten up by

thermal conversion or mechanical friction. On the other hand, a bicycle is about 80–90 percent efficient and is probably the most efficient mode of transportation in existence.



```
Trajectory DEMO, Press <ESC> to Exit.
<RIGHT>, <LEFT> to adjust velocity, <UP>, <DOWN> to adjust angle
<G>, <B> to adjust gravity, <W>, <E> to adjust wind, <CTRL> to fire.
Ang=55, Vel=9.499998
Wind force=-0.010000, Gravity Force=0.200000
```

FIGURE 13.16
The projectile demo.

Basic Friction Concepts

Friction is basically resistance in the opposite direction of motion and hence can be modeled with a force usually referred to as the frictional force. Take a look at Figure 13.17; it depicts the standard frictional model of a mass m on a flat plane.

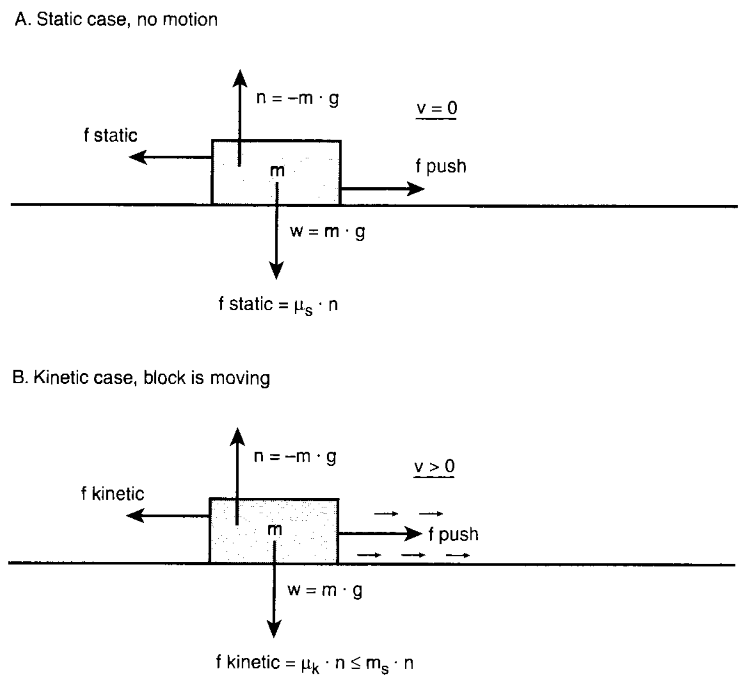
If you try to push the mass in a direction parallel to the plane you will encounter a resistance or frictional force that pushes back against you. This force is defined mathematically as

$F_{fstatic} = m*g*\mu_s$.

where m is the mass of the object, g is the gravitational constant (9.8 m/s^2) and $_s$ is the static frictional coefficient of the system that depends on the conditions and materials of the mass and the plane. If the force F you apply to the object is greater than F_f, then the object will begin to move. Once the object is in motion its frictional coefficient usually decreases to another value, which is referred to as the coefficient of kinetic friction, μ_k.

$F_{fkinetic} = m*g*\mu_k$.

FIGURE 13.17
Basic friction model.



When you release the force, the object slowly decelerates and comes to rest because friction is always present.

To model friction on a flat surface all you need do is apply a constant negative velocity to all your objects that is proportional to the friction that you want. Mathematically, this is

```
Velocity New = Velocity Old - friction.
```

The result is objects that slow down at a constant rate once you stop moving them. Of course, you have to watch out for letting the sign of the velocity go negative or in the other direction, but that's just a detail. Here's an example of an object that is moved to the right with an initial velocity of 16 pixels per frame and then slowed down at a rate of 1 pixel per frame due to virtual friction:

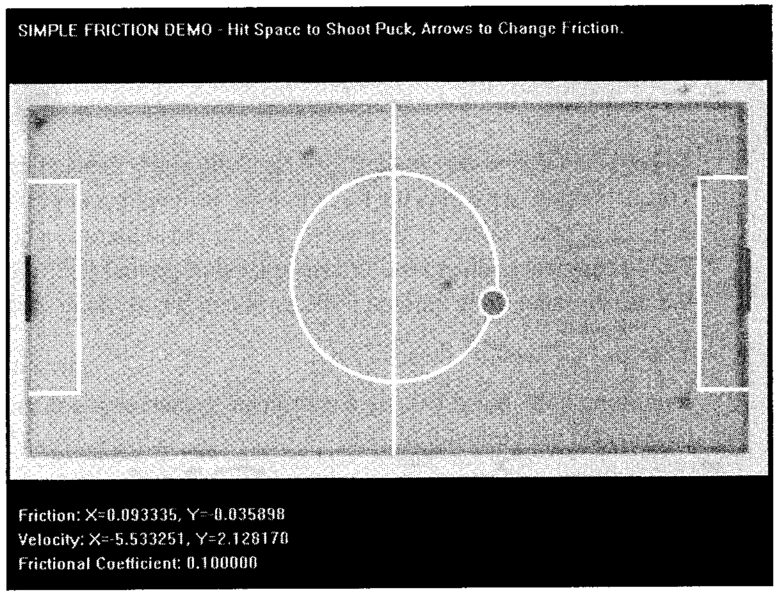
```
int x_pos      = 0,  // starting position
    x_velocity = 16, // starting velocity
    friction   = -1; // frictional value

// move object until velocity <= 0
while(x_velocity > 0)
     {
     // move object
     x_pos+=x_velocity;
```

```
     // apply friction
     x_velocity+=friction;
     } // end while
```

The first thing you should notice is how similar the model for friction is to gravity. They are almost identical. The truth is that both gravitational forces and frictional forces act in the same way. In fact, all forces in the universe can be modeled in the exact same way. Also, you can apply as many frictional forces to an object as you want. Just sum them up.

As an example of friction I have written a little air hockey demo named `DEMO13_5.CPP|EXE (16-bit version, DEMO13_5_16B.CPP|EXE)`, shown in Figure 13.18. The program lets you fire a hockey puck on a virtual air hockey table in a random direction every time you press the spacebar. The puck then bounces around off the borders of the table until it comes to rest due to friction. If you want to change the frictional coefficient of the table, use the arrow keys. See if you can add a paddle and a computer controlled opponent to the simulation!

FIGURE 13.18
The hockey demo.


```
SIMPLE FRICTION DEMO - Hit Space to Shoot Puck, Arrows to Change Friction.
```

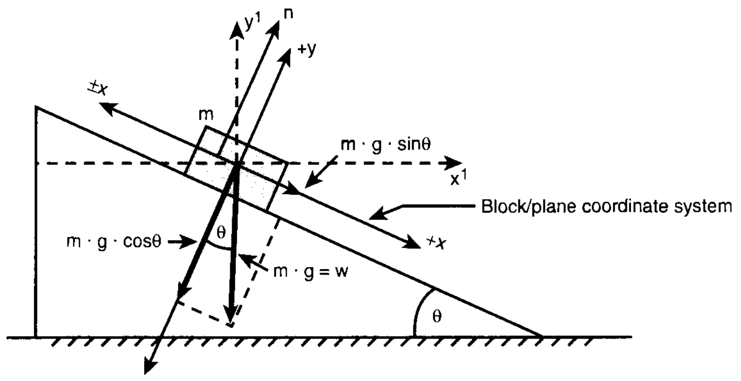
```
Friction: X=0.093335, Y=-0.035898
Velocity: X=-5.533251, Y=2.128170
Frictional Coefficient: 0.100000
```

Friction on an Inclined Plane (Advanced)

That wasn't too bad huh? The bottom line is that friction can be modeled as a simple resistive force or a negative velocity on an object each cycle. However, I want to show you the math and derivation of friction on an inclined plane since this will give you the tools you need to analyze much more complex problems. Be warned, though: I'm going to use a lot of vectors, so if you're still rusty or having trouble then take a look back when I talked about them in Chapter 8, "Vector Rasterization and 2D Transformations," or pick up a good linear algebra book.

Figure 13.19 shows the problem we're trying to solve. Basically, there is a mass m on an inclined plane. The plane has frictional coefficients μ_s and μ_k for the static and kinetic (moving) cases respectively. The first thing we need to do is write the formulas that describe the mass in its equilibrium position, that is, not moving. In this case, the sum of the forces in the x-axis are 0 and the sum of the forces in the y-axis are 0.

FIGURE 13.19
The inclined plane problem.



To begin the derivation we must first touch on a new concept called the *normal force*. This is nothing more than the force that the inclined plane pushes the object back with, or in other words, if you weigh 200 lbs., then there is a normal force of –200 lbs. pushing back (due to the surface tension of ground you're standing on) at you. We usually refer to the normal force as η, and it is equal in magnitude to

$\eta = m*g.$

Interesting huh? But if you lay a coordinate system down, then the gravity force must be opposite the normal force, or

$\eta - m*g = 0.$

This is why everything doesn't get sucked into the ground. Okay, now that we know that, let's derive the motion equations of this block mass. First, we lay down an x,y coordinate system on the incline plane with +X parallel to the plane and in the downward sliding direction; this helps the math. Then we write the equilibrium equations for the x and y axes. For the x-axis we know that the component of gravity pushing the block is

$\text{force due to gravity} = m*g*\sin\theta.$

And the force due to friction holding the block from sliding is

$\text{force due to friction} = -\eta*\mu_s.$

The negative sign is because the force acts in the opposite direction. When the object isn't sliding we know that the sum of these forces are equal to 0. Mathematically, we have

`force due to gravity + force due to friction = 0`

Or, the sum of forces in the x-axis is

$\Sigma F_x = m*g*\sin\theta - \eta*\mu_s = 0.$

MATH

Note that I use sine and cosine to resolve the x,y components of the force. I'm basically just breaking the force vectors into components, nothing more.

We have to do the same for the y-axis, but this is fairly easy because the only forces are the weight of gravity and the normal force:

$\Sigma F_y = \eta - m*g*\cos\theta$

All right, so all together we have

$\Sigma F_x = m*g*\sin\theta - \eta*\mu_s = 0.$

$\Sigma F_y = \eta - m*g*\cos\theta = 0.$

But, what is η? From ΣF_y, we once again see that:

$\eta - m*g*\cos\theta = 0.$

Hence,

$\eta = m*g*\cos\theta,$

therefore we can write:

$\Sigma F_x = m*g*\sin\theta - (m*g*\cos\theta)*\mu_s = 0.$

This is what we need. From it we can derive the following results:

$m*g*\sin\theta = (m*g*\cos\theta)*\mu_s$

$\mu_s = (m*g*\sin\theta)/(m*g*\cos\theta) = \tan\theta$

Or canceling out the m*g and replacing sin/cos by tan,

$\theta_{critical} = \tan^{-1}\mu_s$

Now listen carefully. This tells us that there is an angle called the *critical angle* ($\theta_{critical}$) at which the mass starts to slide. It is equal to the inverse tangent of the static coefficient of friction. If we didn't know the frictional coefficient of a mass and some incline plane, we could find it this way by tilting the plane until the mass starts to move. But this doesn't help us with the x-axis, or does it? The equation tells us that the mass won't slide until the angle $\theta_{critical}$ is reached. When it is reached the mass will slide governed by:

$$\Sigma F_x = m*g*\sin\theta - (m*g*\cos\theta)*\mu_s$$

Well, almost… There is one detail. When the mass starts to slide, the difference is $m*g*\sin\theta - (m*g*\cos\theta)*_{s} > 0$, but in addition we need to change the frictional coefficient to $_{k}$ (the coefficient of kinetic friction) to be totally correct!

$$F_x = m*g*\sin\theta - (m*g*\cos\theta)*\mu_k$$

TRICK

You can just average μ_k and μ_s and use that value in all the calculations. Because you're making video games and not real simulations, it doesn't matter if you oversimplify the two frictional cases into one, but if you want to be correct, you should use both frictional constants at the appropriate times.

With all that in mind let's compute the final force along the x-axis. We know that F=m*a, therefore:

$$F_x = m*a = m*g*\sin\theta - (m*g*\cos\theta)*\mu_k$$

And dividing by m we get

$$a = g*\sin\theta - (g*\cos\theta)*\mu_k$$
$$a = g*(\sin\theta - \mu_k*\cos\theta)$$

You can use this exact model to move the block mass, that is, each cycle you can increase the velocity of the block in the positive X-direction by $g*(\sin\theta - \mu_k*\cos\theta)$. There's only one problem: This solution is in our rotated coordinate system! There's a trick to getting around this: You know the angle of the plane, and hence you can figure out a vector along the downward angle of the plane:

$$x_{plane} = \cos\theta$$
$$y_{plane} = -\sin\theta$$

$$Slide\_Vector = (\cos\theta, -\sin\theta)$$

The minus sign is on the Y-component because we know it's in the –Y direction. With this vector we can then move the object in the correct direction each cycle—this is a hack, but it works. Here's the code to perform the translation and velocity tracking:

```
// Inputs
float x_pos      = SX, // starting point of mass on plane
      y_pos      = SY,
      y_velocity = 0,  // initial y velocity
      x_velocity = 0,  // initial x velocity
      x_plane    = 0,  // sliding vector
      y_plane    = 0,
```

```
      gravity    = 1,  // do want to fall too fast
      velocity   = INITIAL_VEL, // whatever

     // must be in radians and it must be greater
     // than the critical angle
     angle      = PLANE_ANGLE, // compute velocities in x,y

     frictionk   = 0.1; // frictional value

// compute trajectory vector
x_plane = cos(angle);
y_plane = sin(angle); // no negative since +y is down

// do slide loop until object hits
// bottom of screen at SCREEN_BOTTOM
while(y_pos < SCREEN_BOTTOM)
     {
     // update position
     x_pos+=x_velocity;
y_pos+=y_velocity;

     // update velocity
      x_vel+=x_plane*gravity*(sin(angle) - frictionk *cos(angle));
      y_vel+=y_plane*gravity*(sin(angle) - frictionk *cos(angle));

     } // end while
```

The point of physics modeling sometimes is just to understand what the underlying physics are so you can model them in a somewhat convincing manner. In the case of the incline plane, basically all that math just boiled down to the concept that acceleration is a function of the angle (we knew this from common sense). However, in Volume II of the book I'm going to cover much more realistic physics using numerical integration, and in those cases, you need to know the real models and real forces on everything.

Basic Ad Hoc Collision Response

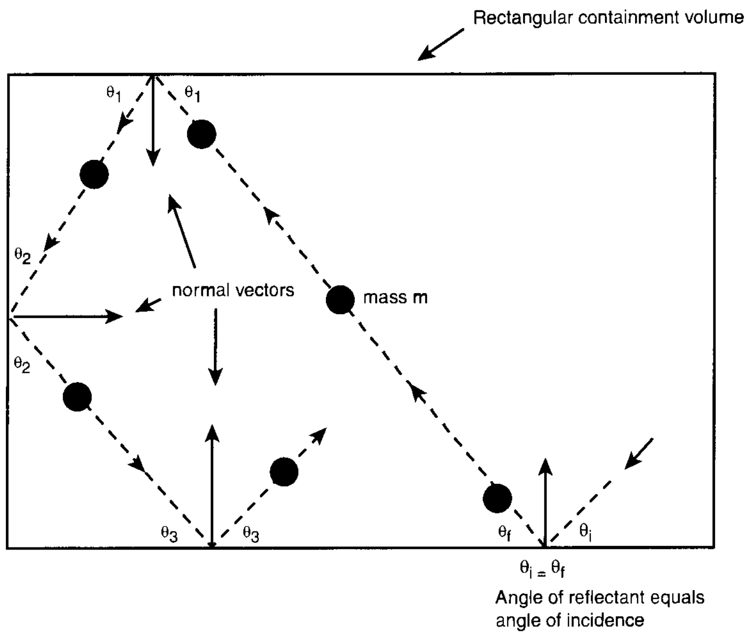
As I explained earlier in the chapter, two kinds of collisions exist: *elastic* and *non-elastic*. Elastic collisions are collisions where both kinetic energy and momentum are conserved in the colliding objects while non-elastic collisions don't conserve these values and energy is converted to heat and/or used for mechanical deformations.

Most video games don't even try to mess with non-elastic collisions and stick to simplified elastic collisions since they themselves are hard enough to compute. Before I show you the real way to do it let's use the other side of our brains. Game programmers that don't know anything about elastic or inelastic collisions have been faking collisions for years and we can do the same.

Simple x,y Bounce Physics

Take a look at Figure 13.20. It depicts a fairly common collision problem in games, that is, bouncing an object off the boundaries of the screen. Given the object has initial velocity (xv,yv), the object can hit any of the four sides of the screen. If one object collides with another object that has mass much greater than the colliding object, then the collision is much simplified since we only need to figure out what happened to the single object that's doing the colliding rather than two objects. A pool table is a good example of this. The balls have very small mass in comparison to the pool table.

FIGURE 13.20
The bouncing ball.



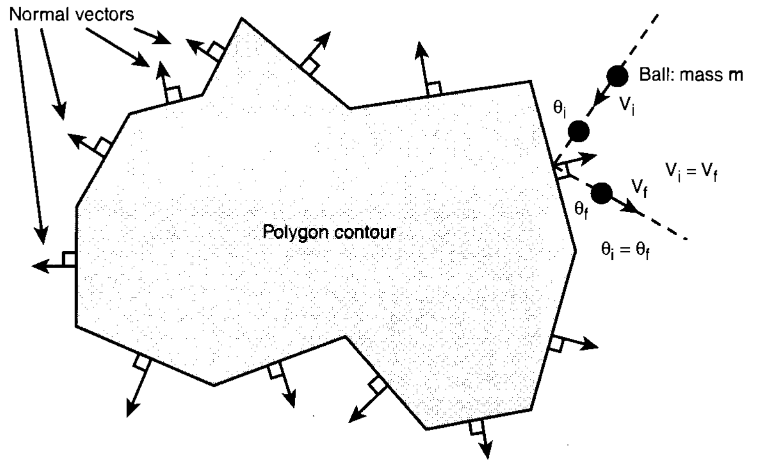
When a ball hits one of the sides then it always reflects off the side at an angle equal and opposite to its initial trajectory, as shown in Figure 13.20. Thus all we need to do to bounce an object off a pool table-like environment that consists of hard edges that have large mass is to compute the normal vector direction, the direction that the object struck at, and then reflect the object at the same angle as shown in Figure 13.21.

Although this isn't as complex as the general elastic collision, it still takes a bit of trigonometry, so there's got to be a simpler way! And of course there is. The trick is to understand the physics model that you're trying to model. Then the idea is to see if you can solve the problem in some other way since you have exact knowledge of all

the conditions. Here's the trick: Instead of thinking in terms of angles and all that, think in terms of results. The bottom line is if the object hits a wall to the east or west then you want to reverse its X velocity while leaving its Y velocity alone. And similarly on the north and south walls, you want to reverse the Y velocity and leave the X velocity alone. Here's the code:

```
// given the object is at x,y with a velocity if xv,yv
// test for east and west wall collisions
if (x >= EAST_EDGE || x <= WEST_EDGE)
   xv=-xv; // reverse x velocity

// now test for north and south wall collisions
if (y >= SOUTH_EDGE || y <= NORTH_EDGE)
   yv=-yv; // reverse y velocity
```

FIGURE 13.21
Bouncing a ball off an irregular object with flat facets.



And amazingly the object will bounce off the walls. Of course, this simplification only works well for horizontal and vertical barriers. You'll have to use the more general angle calculation for walls or barriers that aren't co-linear with the x and y axes.

TRICK

If you want to use the preceding technique as a quick cheat to make objects bounce off of each other, simply assume that each object is a bounding rectangle from the other object's point of view. Enact the collision and then re-compute the velocities. Figure 13.22 illustrates this.

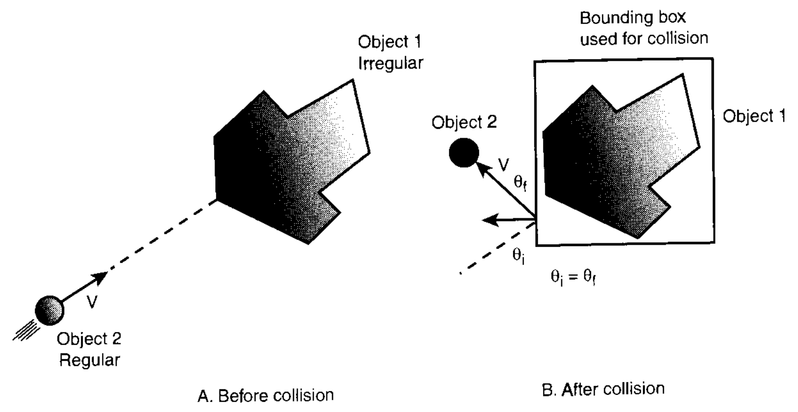


FIGURE 13.22
A simple cheat for object-to-object collisions.

As an example of using these techniques, I have created a demo named `DEMO13_6.CPP|EXE (16-bit version, DEMO13_6_16B.CPP|EXE)` that models a pool table with balls that never stop bouncing. Figure 13.23 shows a screen shot of the game in action. Note that when running the simulation the balls don't hit each other, just the edges of the pool table.

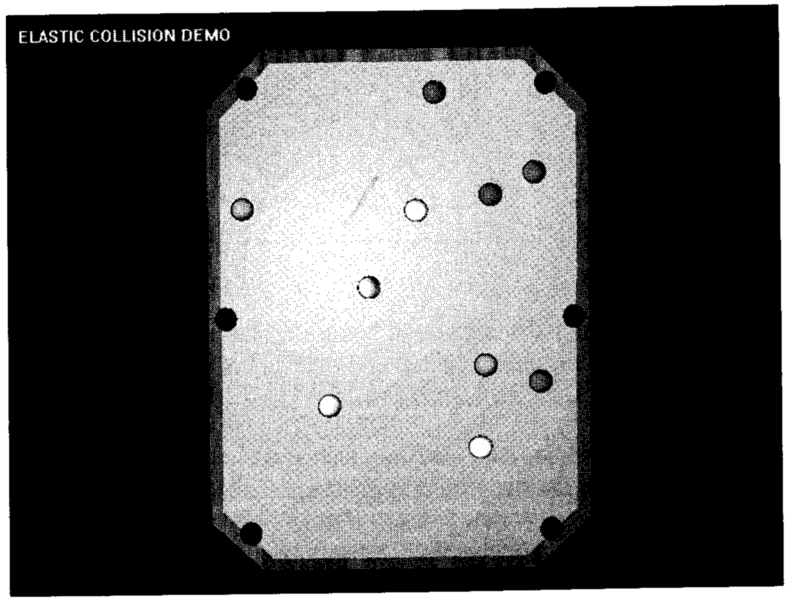


FIGURE 13.23
Simple collision pool table model.

Computing the Collision Response with Planes of Any Orientation

Using rectangles as bounding collision volumes works okay if you write pong games, but this is the 21st century, baby, and we need just a little bit more! What we need to do is derive the reflection equations for a vector reflecting off a flat plane. This is

shown in part A of Figure 13.24 in 2D (the 3D case is the same). To solve the problem, first we have to make an assumption about what will happen when an object that's very small and perfectly elastic hits a wall. I think we can already come to the conclusion that it will bounce off the wall at the same angle it arrived at. Therefore, the *angle of reflection* (the angle of departure by the object after the collision) equals the *angle of incidence* (the incoming angle before the collision) relative to the *normal*, or perpendicular to the wall. Now, let's see the math…

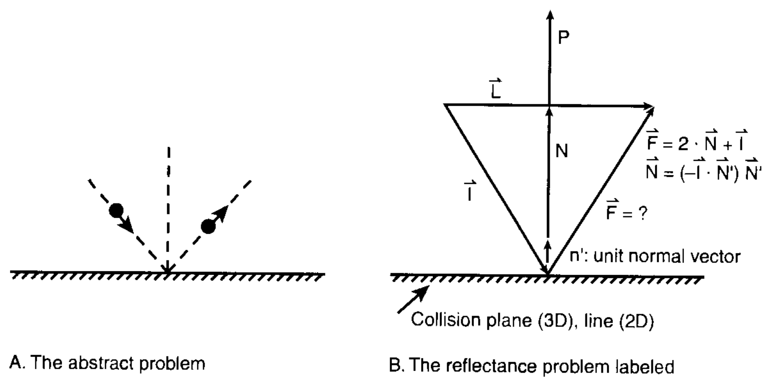

P
$\vec{L}$
N
$\vec{F} = 2 \cdot \vec{N} + \vec{I}$
$\vec{N} = (-\vec{I} \cdot \vec{N}')\,\vec{N}'$
$\vec{I}$
$\vec{F} = ?$
n': unit normal vector
Collision plane (3D), line (2D)
A. The abstract problem
B. The reflectance problem labeled

FIGURE 13.24
The vector reflection problem illustrated.

Solving the problem requires nothing more than a vector geometry construction, but it's not totally trivial.

TIP

If you ever try to get a game programming job, I can almost guarantee they will ask you this question on the interview because it's deceptively complex. Luckily, you can just read this section and blurt out the answer, and they'll think you're a genius! Let's get started.

Part B of Figure 13.24 depicts the problem. Note that there isn't an x,y axis. It's unnecessary since we're using vectors and I want the problem to be totally general.

The problem can be stated like this:

Given the initial vector direction **I** and the normal to the plane **N'**, find **F**.

Before we get crazy, let's talk about the normal vector. The normal vector **N'** is just the normalized version of **P**, but what is **P**? **N** is just the perpendicular to the plane or line that the ball is hitting that we want the ball to bounce off of. We can determine **P** in a number of ways; we might have pre-computed it and stored it in a data structure or we can figure it out on-the-fly.

There are a number of ways to do this depending on the representation of the "wall." If the wall is a plane in 3D then we can extract **P** based on the point-normal form of a plane:

$n_x(x - x_0) + n_y(y - y_0) + n_z(z - z_0) = 0$

The normal vector is just **P**=<n_x, n_y, n_z>. To make sure the normal is normalized or a unit vector then you divide each element by the total length—remember?

$N' = <n_x, n_y, n_z>/|P|$

Where |**P**| is the length and is computed like this:

$|P| = \text{sqrt}(n_x^2 + n_y^2 + n_z^2)$

In general, the length of a vector is the square root of the sum of squares of its components.

On the other hand if your collision line is a line in 2D or a segment then you can compute the normal or perpendicular by finding any vector that is perpendicular to the line. Thus if the line is in the form of two endpoints in 2D as shown in Figure 13.25 like this:

Given: $p_1(x_1,y_1)$, $p_2(x_2,y_2)$

$V_{12} = <x_2 - x_1, y_2 - y_1> = <v_x, v_y>$

The perpendicular can be found with this trick:

$P_{12} = <n_x, n_y> = <-v_y, v_x>$

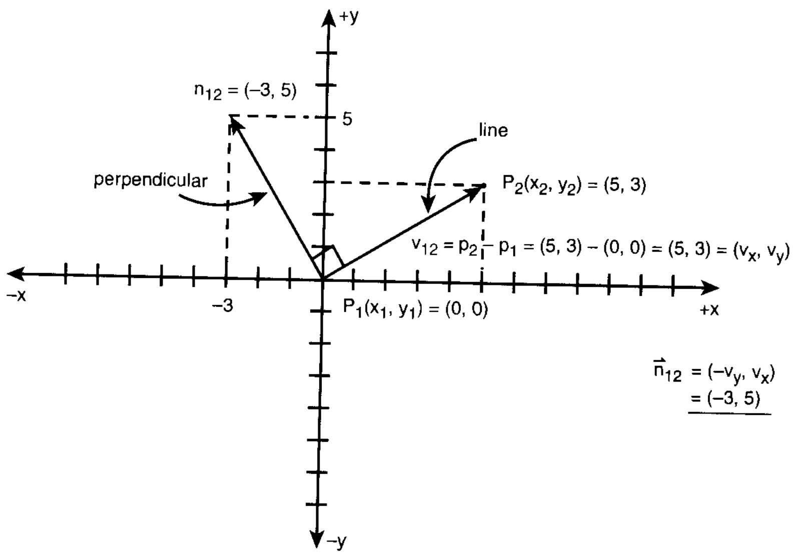

+y
$n_{12} = (-3, 5)$
5
line
perpendicular
$P_2(x_2, y_2) = (5, 3)$
$V_{12} = P_2 - P_1 = (5, 3) - (0, 0) = (5, 3) = (v_x, v_y)$
–x
–3
$P_1(x_1, y_1) = (0, 0)$
+x
$\vec{n}_{12} = (-v_y, v_x) = (-3, 5)$
–y

FIGURE 13.25
Computing the perpendicular to a line.

The vector from p1 to p2 (remember endpoint minus initial) is

This trick is based on the definition of dot product, which states that a vector dotted with its normal equals 0, thus:

$V_{12} \cdot N_{12} = 0$
$<v_x, v_y> . <n_x, n_y> = 0$

or

$v_x*n_x + v_y*n_y = 0$

And what makes this true is $n_x = -v_y$, $n_y = v_x$:

$v_x*(-v_y) + v_y*(v_x) = -v_x*v_y+v_x*v_y = 0$

Nice, huh?

All right, so you know how to get the normal vector and of course you need to normalize it and make sure it has length 1.0, so **N'**, = **P**/|**P**| which is equal to:

$N' = <-v_y, v_x>/\text{sqrt}((-v_y)^2 + v_x^2)$

Now back to the derivation... At this point we have the normal vector **N'**, which shouldn't be confused with N in the figure since **N** is along **N'**, but not related to the length of **P** in any way. **N** *is the projection of* **I** *along* **N'**. Okay, that sounds like voodoo—and it is. A projection is like a shadow. If I were to shine light from the left side of the figure in the left to right direction then **N** would be the shadow or projection that **I** casts on the **N'** axis. This projection is the **N** we need. Once we have **N** then we can find **F** with a little vector geometry. First, here's **N**:

$N = (-I . N')*N'$

This states that **N** (which is the vector projection of I on **N'**) is equal to the dot product of **–I** and **N'** and then multiplied by the vector **N'**. Let's take this apart in two chunks. First the term (**-I** . **N'**) is just a scalar length like 5; it's not a vector. This is a handy thing about dot products: If you want to find the shadow of one vector (projection) then you can dot it with the unit version of the vector in question, thus you can resolve the components of a vector into any direction you want. Basically, you can ask, "What's **R** in the **V'** direction?" Where **V'** is normalized. Therefore, the first part (**-I** . **N'**) gives you a number (the –1 is just to flip the direction of **I**). But, you need a vector **N**, so all you do is multiply the number by the unit vector **N'** (vector multiplication) and, whammo, you have **N**.

Once you have **N** it's all bedrock, baby, just do some vector geometry and you can find **F**:

$L = N + I$

and

```
F = N + L
```

substituting **L** into **F**,

```
F = N + (N + I)
```

Therefore,

```
F = 2*N + I
```

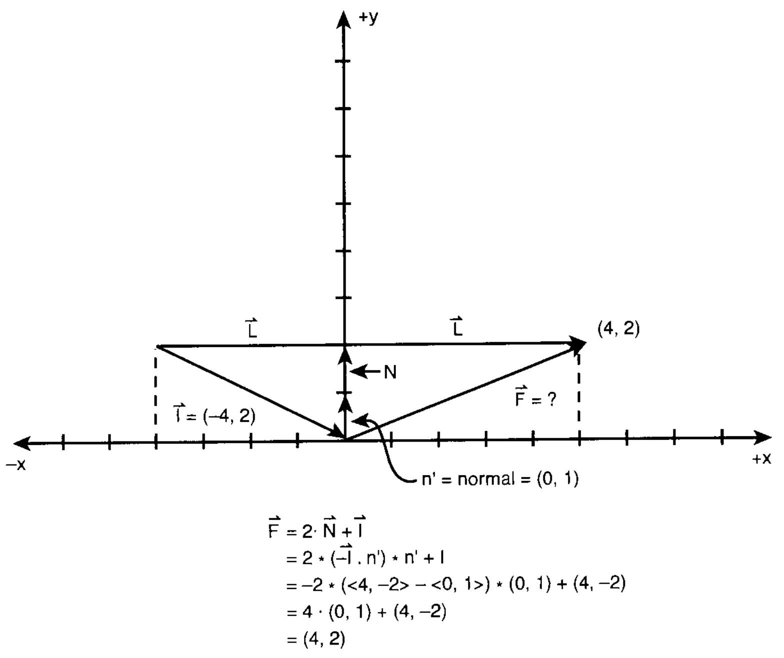
Burn that last line into your skull. It could be the difference between Burger King and DreamWorks Interactive—right, Rich?

An Example of Vector Reflection

When I used to study mathematics, I used to read things like "R is a closed ring with an isomorphism on Q's kernel." I wouldn't be as nutty as I am today if they just gave me an example once in a while! Alas, I don't want you to end up running naked in the streets with a cape—one naked super hero game programmer is enough, so let's try a real example.

Figure 13.26 shows the setup of the problem. I have made the bounce plane co-linear with the x-axis to make things easier.

FIGURE 13.26
A numerical example of vector reflection.



The initial velocity vector of our object is **I**=<4,-2>, **N**'=<0,1>, and we want to find **F**. Let's plug everything into our equation:

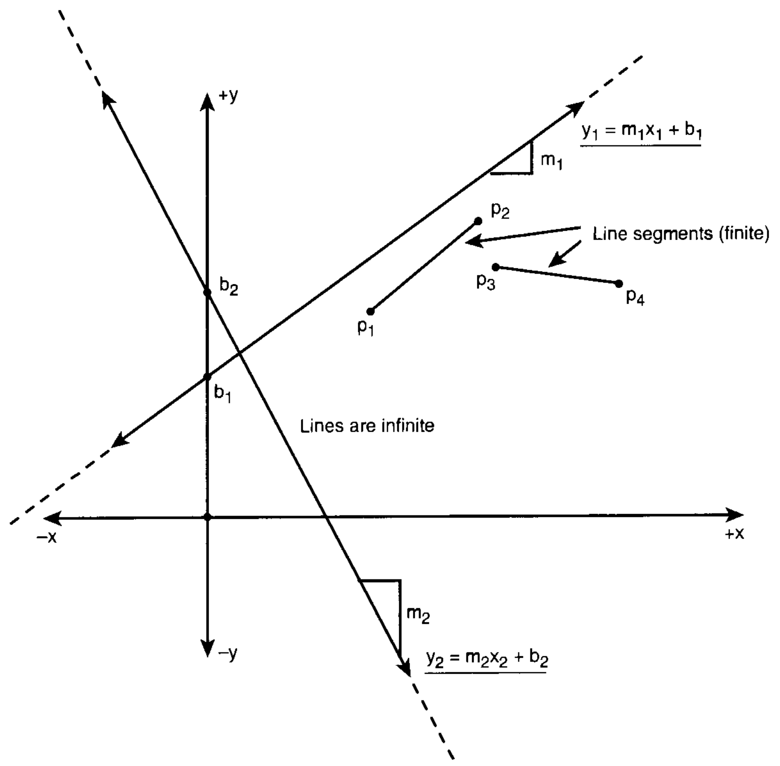
```
F =  2*N + I
  =  2*(-I . N')*N' + I
  = -2*(<4,-2> . <0,1>)*<0,1> + <4,-2>
  = -2*(4*0 + -2*1)*<0,1> + <4,-2>
  = 4*<0,1> + <4,-2>
  = <0,4> + <4,-2>
  = <4,2>
```

Lo and behold, if you look at Figure 13.26, that's the correct answer! Now, there's only one detail that we've left out of all this: determining when the ball or object actually hits the plane or line.

Intersection of Line Segments

You could probably figure this one out, but I'll give you a hand. The problem is basically a line intersection computation. But the surprise is that we are intersecting line segments, not lines; there's a difference. A line goes to infinity in both directions, while a line segment is finite, as shown in Figure 13.27.

FIGURE 13.27
Lines and line segments are very different.



The problem boils down to this: As an object moves with some velocity vector V_i, we want to detect whether the vector pierces through the collision plane or line. Why? Well, if an object is moving at velocity V_i then one frame or time click later it will be located at its current position $(x_0,y_0)+V_i$, or in terms of components:

```
x1 = x0 + vix
y1 = y0 + viy
```

Therefore, you can think of the velocity vector as a line segment that leads the way to wherever the object we are drawing is going. In other words, we want to determine whether there is an intersection point (x,y) of the line segments. Here's the setup:

Object Vector Segment: $\mathbf{S}_1 = <p_1(x_1,y_1) - p_0(x_0,y_0)>$

Boundary Line Segment: $\mathbf{S}_2 = <p_3(x_3,y_3) - p_2(x_2,y_2)>$

You need the exact intersection point (x,y), so that when you compute the reflection vector **F** you start its initial position at (x,y). This is shown in Figure 13.28. The problem seems simple enough, but it's not as easy as you think. Intersecting two lines is as easy as solving a system of 2 equations, but determining if two line segments intersect is a little harder. That's because that although the segments are lines, they are finite, so even if the lines that the segments run along intersect, the segments may not. This is shown in Figure 13.29. Therefore, you need to not only determine where the lines that the segments define intersect, you need to see if this point is within both segments! This is the hard part.

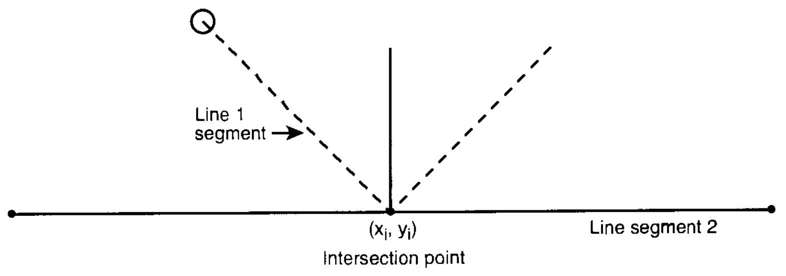


FIGURE 13.28
Intersection and reflection.

The trick to solving the problem is using a *parametric* representation of each line segment. I'll call **U** the position vector of any point on $\mathbf{S}_1$ and **V** the position vector of any point on S_2:

Equation 1: $\mathbf{U} = p_0 + t^*\mathbf{S}_1$

Equation 2: $\mathbf{V} = p_2 + s^*\mathbf{S}_2$

with the constraint that (0 <= t <= 1), (0 <= s <= 1).

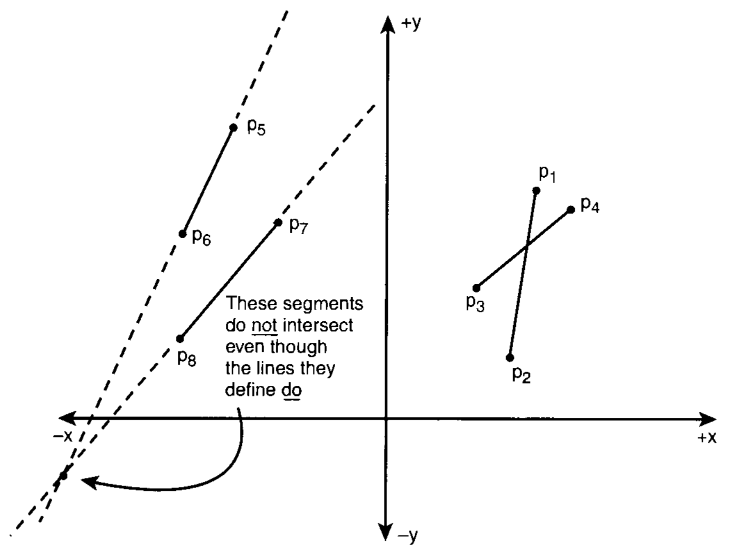

+y
p_5
p_6
p_7
p_8
These segments do not intersect even though the lines they define do
p_1
p_4
p_3
p_2
-x
+x
-y

FIGURE 13.29
Intersecting and non-intersecting segments.

Figure 13.30 shows what these two equations represent.

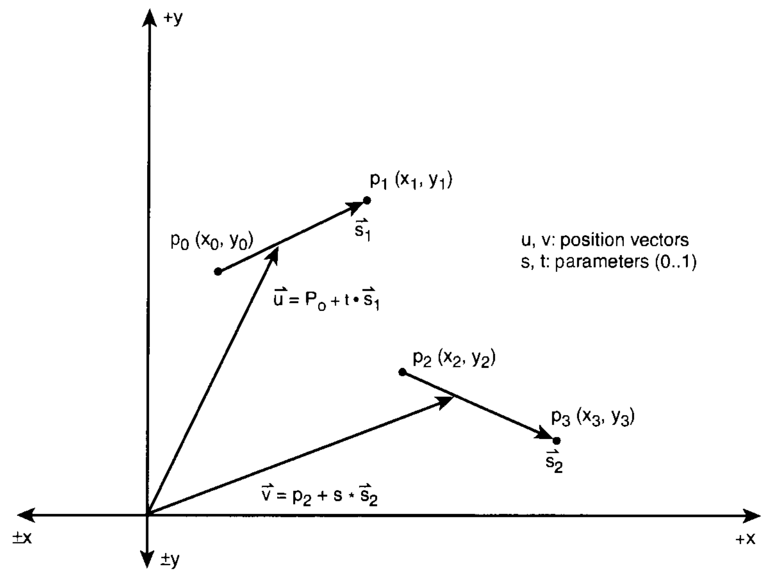


FIGURE 13.30
The parametric representation of U and V.

Referring to the figure, we see that as t varies from 0 to 1 the line segment from p_0 to p_1 is traced out and similarly as s varies from 0 to 1, the line segment from p_2 to p_3 is traced out. Now we have all we need to solve the problem. We solve equations 1 and 2 for s,t and then plug the values back in to either equation and find the (x,y) of the intersection. Moreover, when we find (s,t) if either of them is not in the range (0 to 1) then we know that the intersection was not within the segments. Pretty neat, huh? I'm not going to every detail of the entire derivation since this is in about 100,000 math books, but I'll show you the important stuff:

Given:

$$\mathbf{U} = p_0 + t*\mathbf{S}_1$$
$$\mathbf{V} = p_2 + s*S_2$$

solve for (s,t) when **U** = **V**,

$$p_0 + t*\mathbf{S}_1 = p_2 + s*\mathbf{S_2}$$

$$s*\mathbf{S_2} - t*\mathbf{S}_1 = p_0 - p_2$$

Breaking the last equation into (x,y) components,

$$s*S_{2x} - t*S_{1x} = p_{0x} - p_{2x}$$
$$s*S_{2y} - t*S_{1y} = p_{0y} - p_{2y}$$

We have two equations, two unknowns, push into a matrix and solve for (s,t):

$$\underset{\mathbf{A}}{\begin{vmatrix} S_{2x} & -S_{1x} \\ S_{2y} & -S_{1y} \end{vmatrix}} \underset{\mathbf{X}}{\begin{vmatrix} s \\ t \end{vmatrix}} \underset{=}{=} \underset{\mathbf{B}}{\begin{vmatrix} (p_{0x} - p_{2x}) \\ (p_{0y} - p_{2y}) \end{vmatrix}}$$

Using Cramer's Rule we have the following:

$$s = \frac{\text{Det}\begin{vmatrix} (p_{0x}-p_{2x}) & -S_{1x} \\ (p_{0y}-p_{2y}) & -S_{1y} \end{vmatrix}}{\text{Det}\begin{vmatrix} S_{2x} & -S_{1x} \\ S_{2y} & -S_{1y} \end{vmatrix}} \quad t = \frac{\text{Det}\begin{vmatrix} S_{2x} & (p_{0x}-p_{2x}) \\ S_{2y} & (p_{0y}-p_{2y}) \end{vmatrix}}{\text{Det}\begin{vmatrix} S_{2x} & -S_{1x} \\ S_{2y} & -S_{1y} \end{vmatrix}}$$

MATH

Cramer's rule states that you can solve a system of equations **AX=B**, by computing xi = Det($\mathbf{A}_i$)/Det(**A**). Where $\mathbf{A}_i$ is the matrix formed by replacing the ith column of **A** with **B**.

MATH

The determinate (Det) of a matrix in general is rather complex, but for a 2x2 or 3x3 it is very simple to remember. Given a 2x2 matrix the determinate can be computed as follows:

```
A = |a b|  Det(A) = (a*d - c*b)
    |c d|
```

Multiplying all this stuff out, you get

$$s = (-S_{1y}*(p_{0x}-p_{2x}) + S_{1x}*(p_{0y}-p_{2y}))/(-S_{2x}*S_{1y} + S_{1x}*S_{2y})$$
$$t = (S_{2x}*(p_{0y}-p_{2y}) - S_{2y}*(p_{0x}-p_{2x}))/(-S_{2x}*S_{1y} + S_{1x}*S_{2y})$$

Then once you have found (s,t), you can plug either of them into

$$\mathbf{U} = p_0 + t*S_1$$
$$\mathbf{V} = p_2 + s*S_2$$

and solve for **U**(x,y) or **V**(x,y). However, for s,t to be valid both of them must be in the range of (0..1). If either of them is out of range then there is NO intersection. Referring to the worked example in Figure 13.31, let's see if the math works:

TRICK

There's no need to test for intersections of two lines segments if their bounding boxes don't overlap.

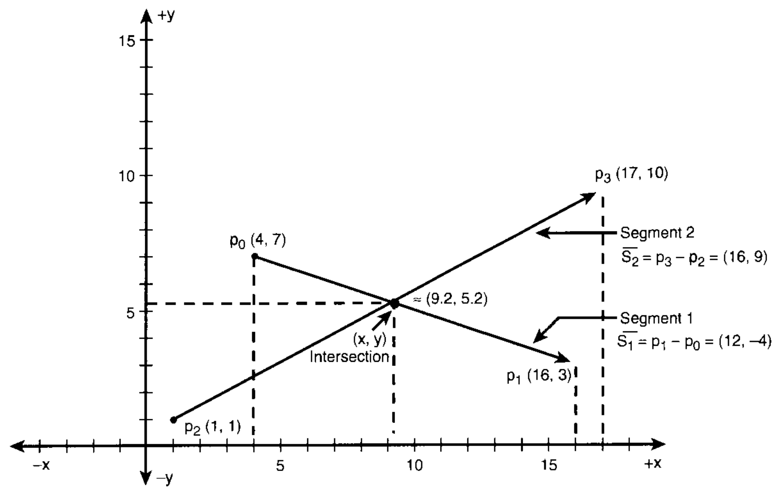


FIGURE 13.31
A worked line segment intersection example.

Math:

$$\bar{u} = \overset{p_0}{p_0} + t \cdot \overline{s_1} = (4, 7) + t \cdot (\overset{\overline{s_1}}{12, -4})$$
$$\bar{v} = \overset{p_2}{p_2} + s \cdot \overline{s_2} = (1, 1) + s \cdot (\overset{\overline{s_2}}{17, 10})$$

plugging in and solving for s:
$s = (-s_{1y} \cdot (p_{0x} - p_{2x}) + s_{1x} \cdot (p_{0x} - p_{2y})) / (-s_{2x} \cdot s_{1y} + s_{1x} \cdot s_{2y})$
$= [4 . (4-1) + 12 \cdot (7-1)] / (17 \cdot 4 + 12 \cdot 10)$
$= 844/.88 = \underline{.44}$
similarly t = $\underline{.383}$
plugging in (s · t) into u, v we get (x,y) = ($\underline{9.28, 5.24}$)

$$p_0=(4,7),\ p_1=(16,3),\ S_1=p_1-p_0=<12,-4>$$
$$p_2=(1,1),\ p_3=(17,10),\ S_2=p_3-p_2=<16,9>$$

And we know that

$$s = (-S_{1y}*(p_{0x}-p_{2x}) + S_{1x}*(p_{0y}-p_{2y}))/(-S_{2x}*S_{1y} + S_{1x}*S_{2y})$$
$$t = (S_{2x}*(p_{0y}-p_{2y}) - S_{2y}*(p_{0x}-p_{2x}))/(-S_{2x}*S_{1y} + S_{1x}*S_{2y})$$

Plugging in all the values we get

```
s = (4*(4-1) + 12*(7-1))/(17*4 + 12*10)   = 0.44
t = (17*(7-1) - 10*(4-1))/ (17*4 + 12*10) = 0.383
```

Since both 0<=(s,t)<=1, we know that we have a valid intersection and thus either s or t can be used to find the intersection point (x,y). Let's use t, shall we?

```
U(x,y) = p0 + t*S1
       = <7,7> + t*<12,-4>
```

Plugging in t=.44, we get

```
       = <7,7> + 0.44*<12,-4> = (9.28, 5.24)
```

which is indeed the intersection. Isn't math fun?

As for using all this technology, I have created a demo of a ball bouncing off the interior of an irregularly shaped polygonal 2D object. Take a look at `DEMO13_7.CPP|EXE` (`16-bit version, DEMO13_7_16B.CPP|EXE`). A screen shot is shown in Figure 13.32. Try editing the code and changing the shape of the polygon.

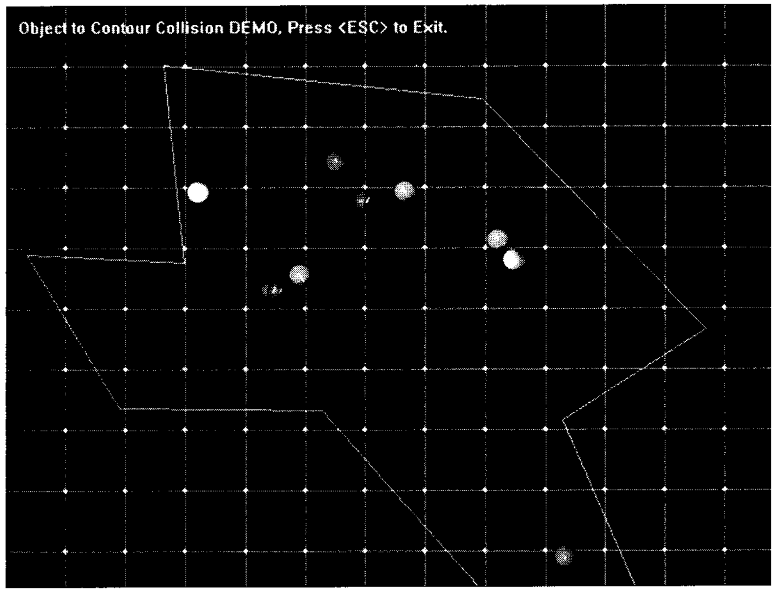

Object to Contour Collision DEMO, Press <ESC> to Exit.

FIGURE 13.32
A bouncing ball trapped in an irregularly shaped polygon demo.

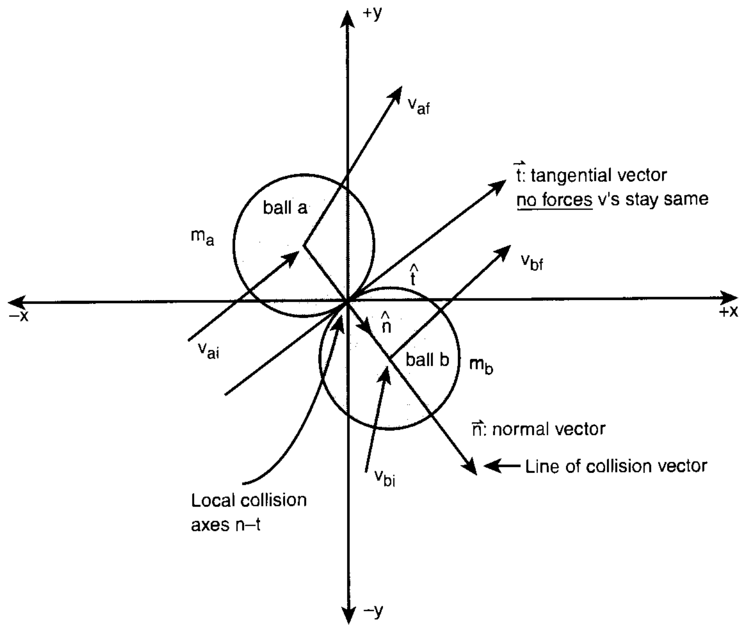
Finally, you may want to try another heuristic when finding a collision trajectory vector. In the previous example we used the velocity vector as one test segment. However, it may make more sense to create a vector that is the length of the radius of the ball and then drop it from the center perpendicular to the edge being tested. This way you catch scathing collisions, but it's a bit more complex and I'll just leave it as an exercise.

Real 2D Object-to-Object Collision Response (Advanced)

I put this section off a bit and moved it down here because I wanted you to really get a handle on momentum and collision and the mathematics needed to work with both. But like Dr. Brown said in *Back to the Future*, "Roads? Where we're going, we don't need roads..." Alas, object-to-object collisions with a fairly realistic collision response aren't the easiest thing in the world to figure out. The final results aren't bad, but coming up with them is no picnic. Anyway, let's get started.

Figure 13.33 depicts the general problem that we want to solve. There are two objects modeled by 2D circles or 3D spheres, and each has a mass and an initial trajectory. When they make contact we want to compute the final trajectory or velocity after the collision. We've touched on this already in the section "The Physics of Linear Momentum: Conservation and Transfer" when we came up with the following equations:

FIGURE 13.33
The central impact of two masses problem.



Conservation of linear momentum:

$$m_a*v_{ai} + m_b*v_{bi} = m_a*v_{af} + m_b*v_{bf}$$

Conservation of kinetic energy:

$$^1/_2*m_a*v_{ai}^2 + {}^1/_2*m_b*v_{bi}^2 = {}^1/_2*m_a*v_{af}^2 + {}^1/_2*m_b*v_{bf}^2$$

After combining them and solving for the final velocities, we get

$$v_{af} = (2*m_b*v_{bi} + v_{ai}*(m_a - m_b))/(m_a + m_b)$$
$$v_{bf} = (2*m_a*v_{ai} - v_{bi}*(m_a - m_b))/(m_a + m_b)$$

These equations are true for perfectly elastic collisions. However, there's a little problem—as they stand they are only 1-dimensional. What we need to do is come up with the 2D solution to the problem (something like a pool table) and this is a bit more complex. Let's start with what we know.

We know that each ball (2D representation) has some mass m; furthermore, the balls are made of the same material throughout, so the center of mass is at the center of the ball body. Next, we know that when real balls hit each other, the balls deform for a moment, some of the kinetic energy is converted to heat, and mechanical work to deform the balls, then the balls separate. This is called the impact event, which is shown in Figure 13.34.

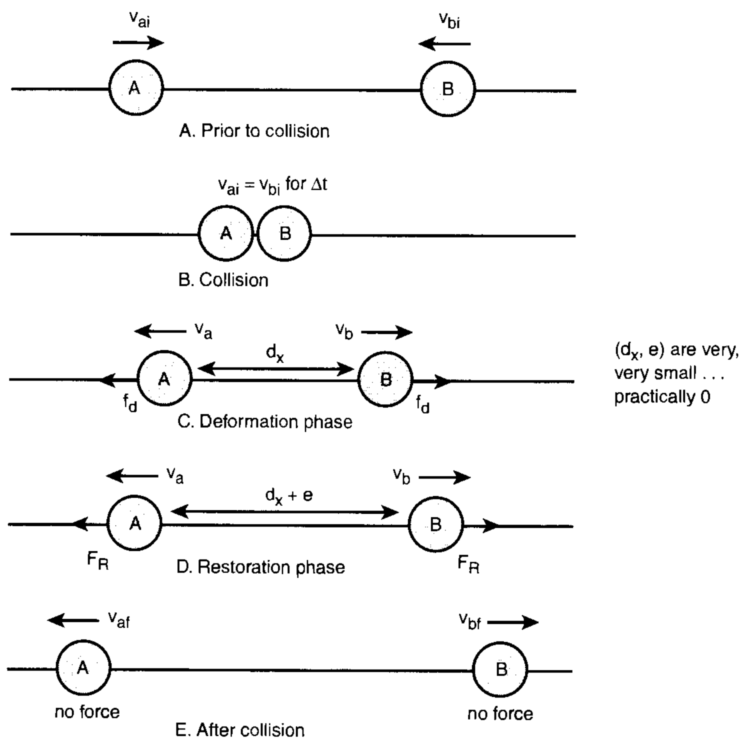


FIGURE 13.34
The phases of the impact event.13.34.

The impact event consists of two separate phases. The first phase: *Deformation*, occurs when the balls make first contact and the balls move at the same velocity. At the end of the deformation phase the *restoration* phase begins and continues until the balls separate. The bottom line is that during the collision event a lot of really complex physics happen that we can't possibly model with a computer, so we have to make some assumptions about the collision. The cool thing is that even with the assumptions, when you see the simulation it will look pretty real! The assumptions are the following:

1. The time it takes for the collision event is very small; call it *dt*.
2. During the collision the positions of the balls do not change.
3. *The velocity of the balls may change significantly.*
4. There are no frictional forces acting during the collision.

Assumption 3 is the only one I need to clarify since I think the others are easy to swallow. For assumption 3 to be true, a force has to be applied during the collision that is almost instantaneous. This type of force is called an *impulse force*. This is the key to solving the problem. When the balls hit there will be a very large, short timed force that is created—an impulse. We can compute the impulses and from them come up with another equation to help solve the problem in 2D. The math is advanced and calculus based, so I will forgo it. The results are the generation of a coefficient that models all the physics during the impact event:

Equation 1: Coefficient of restitution

$$e = \frac{v_{bf} - v_{af}}{v_{bi} - v_{ai}}$$

Equation 1 is referred to by e and called the *coefficient of restitution*. It models the velocity before and after the collision and the loss of kinetic energy. If you set e=1 then the model is a perfectly elastic collision. On the other hand, e < 1 models a less than perfect collision and the velocity of each ball after the collision and the linear momentum will be less. Now where do you get e? e is something you set or look up. The interesting thing is that if you combine the equation for e along with the conservation of momentum equation:

$$m_a*v_{ai} + m_b*v_{bi} = m_a*v_{af} + m_b*v_{bf}$$

you get the following results:

Equation 2: Final velocities

$$v_{af} = ((e+1)*m_b*v_{bi} + v_{ai}*(m_a - e*m_b))/(m_a + m_b)$$
$$v_{bf} = ((e+1)*m_a*v_{ai} - v_{bi}*(m_a - e*m_b))/(m_a + m_b)$$

Isn't that interesting? It's almost identical to the formulas we got when we combined the kinetic energy equations with the linear momentum equations. And in fact the assumption we made when we combined the kinetic energy equations with the linear momentum equations was that kinetic energy was conserved. If we assume that now then we set e=1 and we get

$$v_{af} = ((1+1)*m_b*v_{bi} + v_{ai}*(m_a - 1*m_b))/(m_a + m_b)$$
$$v_{bf} = ((1+1)*m_a*v_{ai} - v_{bi}*(m_a - 1*m_b))/(m_a + m_b)$$

or

$$v_{af} = (2*m_b*v_{bi} + v_{ai}*(m_a - m_b))/(m_a + m_b)$$
$$v_{bf} = (2*m_a*v_{ai} - v_{bi}*(m_a - m_b))/(m_a + m_b)$$

These indeed are the equations with both kinetic energy and linear momentum conserved! So it looks like we're on the right track. We have equations 1 and 2, so we should be able to solve the problem. But there's a catch; the equations are still in 1D, so we need to write them in 2D and then find a solution.

Referring back to Figure 13.33, you see there are two extra axes labeled—the **n** and **t** axes. The **n** axis is in the direction of the line of collision and the **t** axis or tangential axis is perpendicular to **n**. Assuming we have computed the vectors representing these axes (I'll show how a little later) then we can write some equations.

The first set of equations we're going to write relates the tangential component of the velocities before and after the collision. Since there are no frictional forces and no impulsive forces acting tangentially to the line of collision (trust me), the tangential linear momentum (and therefore the velocities) must be the same before and after, right? If there are no forces then this must be true, thus we can write

Equation 3: Relationship between initial and final tangential momentum/velocities

$$m_a*(v_{ai})\mathbf{t} = ma(v_{af})\mathbf{t}$$
$$m_b*(v_{bi})\mathbf{t} = mb(v_{bf})\mathbf{t}$$

And if you wish you can combine them like this:

$$m_a*(v_{ai})\mathbf{t} + m_b*(v_{bi})\mathbf{t} = m_a(v_{af})\mathbf{t} + m_b(v_{bf})\mathbf{t}$$

MATH

My notation is simple; (a,b) refers to the ball, (i,f) refers to initial or final, and (n,t) refers to the component along the n or **t** axes.

Since the masses are the same before and after the collision we can deduce that the velocities are the same by dividing the masses out:

Equation 4: Velocities after collision are equal in tangential direction

$$(v_{ai})t = (v_{af})t$$
$$(v_{bi})t = (v_{bf})t$$

Cool. Now that we have half the problem solved we know the final velocities of the tangential components. Let's find the final velocities of the normal components, or the velocities in the line of collision **n**. We know that linear momentum is conserved always because there are no outside forces acting on the balls when they hit, so we can write:

Equation 5: Linear momentum is conserved in the **n** axis or line of collision

$$m_a*(v_{ai})\mathbf{n} + m_b*(v_{bi})\mathbf{n} = m_a*(v_{af})\mathbf{n} + m_b*(v_{bf})\mathbf{n}$$

And we can also write e in terms of the **n** axis:

Equation 6: The coefficient of restitution in the **n** axis

$$e = \frac{(\mathbf{v_{bf}})n - (\mathbf{v_{af}})n}{(v_{bi})n - (v_{ai})n}$$

Now let's take a look at what we have. If you look at equations 5 and 6, I have highlighted the variables that we don't have: $(\mathbf{v_{af}})\mathbf{n}$, and $(\mathbf{v_{bf}})\mathbf{n}$. Just the normal components of the final velocity. Bingo! We have two equations and two unknowns, so we can solve for them. But we already have the answer! Equation 2 still holds for any particular axis, so I can rewrite it for the component along n:

Equation 7: Final velocities in the normal direction

$$v_{af} = ((e+1)*m_b*(v_{bi})\mathbf{n} + (v_{ai})\mathbf{n}*(m_a - e*m_b))/(m_a + m_b)$$
$$v_{bf} = ((e+1)*m_a*(v_{ai})\mathbf{n} - (v_{bi})\mathbf{n}*(m_a - e*m_b))/(m_a + m_b)$$

That's it!

Resolving the n-t Coordinate System

Now that we have the final collision response, we need to figure out how to get the initial values for $(v_{ai})\mathbf{n}$, $(v_{ai})\mathbf{t}$, $(v_{bi})\mathbf{n}$, $(v_{bi})\mathbf{t}$ and then when the problem is solved we have to convert the values in the **n-t** axes back into values in the x,y axes. Let's begin by first finding the vectors **n** and **t**.

To find **n** we want a vector that is unit length (length equal to 1.0) and along the line from the center of ball $A(x_{a0},y_{a0})$ to the center of ball $B(x_{b0},y_{b0})$. Let's begin by finding a vector from ball A to B, calling it **N**, and then normalizing it:

Equation 8: Computation of **n** and **t**

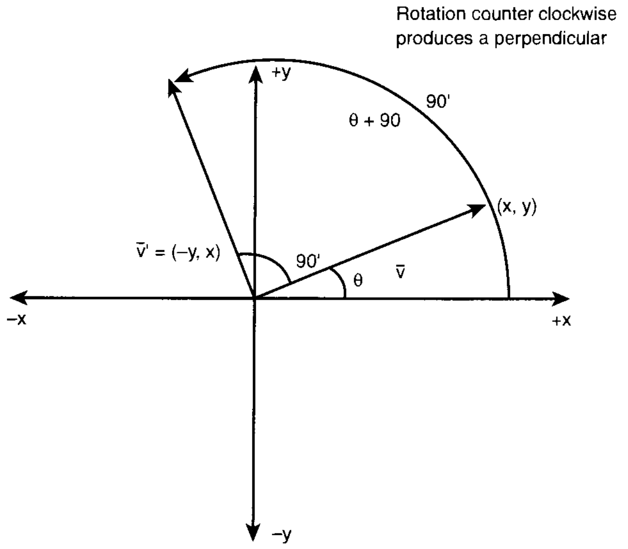
$$N = B - A = \langle x_{b0} - x_{a0},\ y_{b0} - y_{a0}\rangle$$

Normalizing **N** to find **n**, we get

$$n = \mathbf{N}/|\mathbf{N}| = \langle n_x, n_y\rangle$$

Now we need the tangential axis **t** which is perpendicular to **n**. We could find it again using vector geometry, but there's a trick we can use; if we rotate n 90 degrees clockwise, that's what we want. However, when a 2D vector <x,y> is rotated in a plane 90 degrees clockwise, the rotated vector is just **t**=<-y, x>=<t_x, t_y>. Take a look at Figure 13.35.

FIGURE 13.35
Rotating a vector 90 degrees to find its perpendicular.



Now that we have both n and t and they are both unit vectors (since **n** was unit, t is unit also) we're ready to go. We want to resolve the initial velocities of each ball into terms of n and t for ball A and B, respectively:

v_{ai}=<x_{vai}, y_{vai}>
v_{bi}=<x_{vbi}, y_{vbi}>

This is nothing more than a dot product. To find (v_{ai})**n**, the component of the initial velocity of ball A along axis **n**, here's what you do:

```
(vai)n = vai . n = <xvai, yvai> . <nx,ny>
       = (xvai*nx + yvai* ny).
```

Note the result is a scalar, as it should be. Computing the other initial velocities is the same as shown in Equation 9.

Equation 9: Components of v_{ai} along **n** and **t**:

```
(vai)n = vai . n = <xvai, yvai> . <nx,ny>
 (xvai*nx + yvai* ny)
=
(vai)t = vai . t = <xvai, yvai> . <tx,ty>
       = (xvai*tx + yvai* ty)
```

Components of v_{bi} along **n** and **t**:

```
(vbi)n = vbi . n = <xvbi, yvbi> . <nx,ny>
       = (xvbi*nx + yvbi* ny)

(vbi)t = vbi . t = <xvbi, yvbi> . <tx,ty>
       = (xvbi*tx + yvbi* ty)
```

Now we're ready to solve the problem completely. Here are the steps:

1. Compute **n** and **t** (use equation 8).
2. Resolve all the components of v_{ai} and v_{bi} into magnitudes along n and t (use equation 9).
3. Plug values into final velocity shown in equation 7 and remember the tangential components of the final velocities are the same as the initial.
4. The results to the problem are in terms of the coordinate axes **n** and **t**, so you must transform back into x,y.

I'll leave step 4 up to you. Now let's talk about tensors. Just kidding, just kidding. Let's finish this bad boy off. At this point we have the final velocities:

Final velocity for Ball A in terms of **n,t**:

```
(vaf)nt=<(vaf)n,(vaf)t>
```

Final velocity for Ball B in terms of **n,t**:

```
(vbf)nt=<(vbf)n,(vbf)t>
```

Now, let's forget about collisions and think about vector geometry. Take a look at Figure 13.36; it illustrates the problem we have.

Stated in plain Vulcan, we have a vector in one coordinate system **n-t** that we want to resolve into x,y. But how? Again, we are going to use dot products. Take a look at the vector (v_{af})$_{\mathbf{nt}}$ in Figure 13.36, forgetting about the **n,t** axes. We just want (v_{af})$_{\mathbf{nt}}$ in terms of the axis x,y. To compute this we're going to need the contribution of (v_{af})$_{nt}$ along both the x- and y-axes. This can be found with dot products. All we need to do are the following dot products:

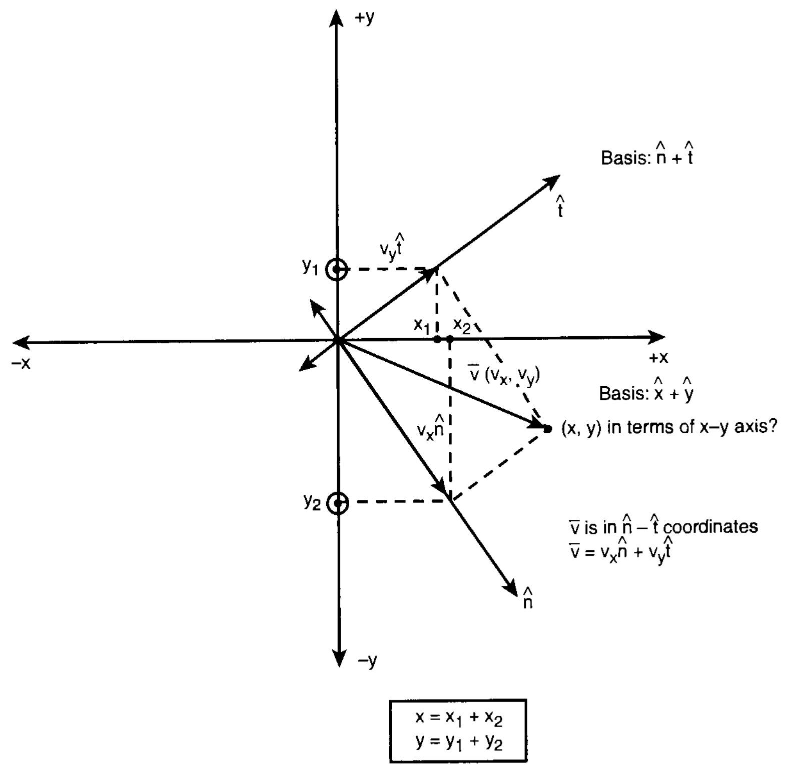
For Ball A, v_a in terms of **n,t** is

```
va = (vaf)n * n + (vaf)t * t
     (vaf)n * <nx,ny> + (vaf)t * <tx,ty>
```

Therefore writing with dot products,

```
xaf = <nx,0> . (vaf)n + <tx,0> . (vaf)t
    = nx*(vaf)n + tx*(vaf)t

yaf = <0,ny> . (vaf)n + <0,ty> . (vaf)t
    = ny*(vaf)n + ty*(vaf)t
```

FIGURE 13.36
Transforming a vector from basis to basis.


+y
Basis: $\hat{n} + \hat{t}$
$\hat{t}$
$v_y\hat{t}$
y_1
x_1
x_2
–x
+x
$\bar{v}\ (v_x, v_y)$
Basis: $\hat{x} + \hat{y}$
(x, y) in terms of x–y axis?
$v_x\hat{n}$
y_2
$\bar{v}$ is in $\hat{n} - \hat{t}$ coordinates
$\bar{v} = v_x\hat{n} + v_y\hat{t}$
$\hat{n}$
–y
$x = x_1 + x_2$
$y = y_1 + y_2$

For Ball B:

$$v_b = (v_{bf})\mathbf{n} * \mathbf{n} + (v_{bf})\mathbf{t} * \mathbf{t}$$
$$(v_{bf})\mathbf{n} * <n_x,n_y> + (v_{bf})\mathbf{t} * <t_x,t_y>$$

Therefore,

$$x_{bf} = <n_x,0> \;.\; (v_{bf})\mathbf{n} + <t_x,0> \;.\; (v_{bf})\mathbf{t}$$
$$= n_x*(v_{bf})\mathbf{n} + t_x*(v_{bf})\mathbf{t}$$

$$y_{bf} = <0,ny> \;.\; (v_{bf})\mathbf{n} + <0,t_y> \;.\; (v_{bf})\mathbf{t}$$
$$= n_y*(v_{bf})\mathbf{n} + t_y*(v_{bf})\mathbf{t}$$

Send the balls off with the above velocities and you're done! Wow, that was a whopper, huh? Now, since I think that the code is a lot easier to understand than the math, I have listed the collision algorithm here from an upcoming demo:

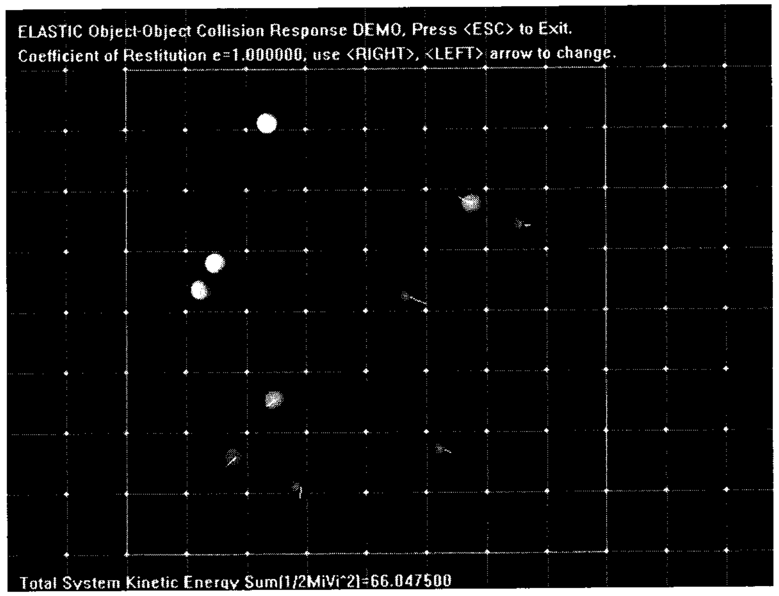
```
void Collision_Response(void)
{
// this function does all the "real" physics to determine if there has
// been a collision between any ball and any other ball; if there is a
// collision, the function uses the mass of each ball along with the
// initial velocities to compute the resulting velocities
```

```
// from the book we know that in general
// va2 = (e+1)*mb*vb1+va1(ma - e*mb)/(ma+mb)
// vb2 = (e+1)*ma*va1+vb1(ma - e*mb)/(ma+mb)

// and the objects will have direction vectors co-linear to the normal
// of the point of collision, but since we are using spheres here as the
// objects, we know that the normal to the point of collision is just
// the vector from the centers of each object, thus the resulting
// velocity vector of each ball will be along this normal vector direction

// step 1: test each object against each other object and test for a
// collision; there are better ways to do this other than a double nested
// loop, but since there are a small number of objects this is fine;
// also we want to somewhat model if two or more balls hit simultaneously

for (int ball_a = 0; ball_a < NUM_BALLS; ball_a++)
     {
     for (int ball_b = ball_a+1; ball_b < NUM_BALLS; ball_b++)
         {
         if (ball_a == ball_b)
            continue;

         // compute the normal vector from a->b
         float nabx = (balls[ball_b].varsF[INDEX_X] -
                       balls[ball_a].varsF[INDEX_X] );
         float naby = (balls[ball_b].varsF[INDEX_Y] -
                       balls[ball_a].varsF[INDEX_Y] );
         float length = sqrt(nabx*nabx + naby*naby);

         // is there a collision?
         if (length <= 2.0*(BALL_RADIUS*.75))
            {
            // the balls have made contact, compute response

            // compute the response coordinate system axes
            // normalize normal vector
            nabx/=length;
            naby/=length;

            // compute the tangential vector perpendicular to normal,
            // simply rotate vector 90
            float tabx =  -naby;
            float taby =  nabx;

            // draw collision
            DDraw_Lock_Primary_Surface();

            // blue is normal
            Draw_Clip_Line(balls[ball_a].varsF[INDEX_X]+0.5,
               balls[ball_a].varsF[INDEX_Y]+0.5,
               balls[ball_a].varsF[INDEX_X]+20*nabx+0.5,
               balls[ball_a].varsF[INDEX_Y]+20*naby+0.5,
               252, primary_buffer, primary_lpitch);
```

```
                // yellow is tangential
                Draw_Clip_Line(balls[ball_a].varsF[INDEX_X]+0.5,
                   balls[ball_a].varsF[INDEX_Y]+0.5,
                   balls[ball_a].varsF[INDEX_X]+20*tabx+0.5,
                   balls[ball_a].varsF[INDEX_Y]+20*taby+0.5,
                   251, primary_buffer, primary_lpitch);

                 DDraw_Unlock_Primary_Surface();

                // tangential is also normalized since
                // it's just a rotated normal vector

                // step 2: compute all the initial velocities
                // notation ball: (a,b) initial: i, final: f,
                // n: normal direction, t: tangential direction

                float vait = DOT_PRODUCT(balls[ball_a].varsF[INDEX_XV],
                                         balls[ball_a].varsF[INDEX_YV],
                                         tabx, taby);

                float vain = DOT_PRODUCT(balls[ball_a].varsF[INDEX_XV],
                                         balls[ball_a].varsF[INDEX_YV],
                                         nabx, naby);

                float vbit = DOT_PRODUCT(balls[ball_b].varsF[INDEX_XV],
                                         balls[ball_b].varsF[INDEX_YV],
                                         tabx, taby);

                float vbin = DOT_PRODUCT(balls[ball_b].varsF[INDEX_XV],
                                         balls[ball_b].varsF[INDEX_YV],
                                         nabx, naby);


                // now we have all the initial velocities
                // in terms of the n and t axes
                // step 3: compute final velocities after
                // collision, from book we have
                // note: all this code can be optimized, but I want you
    // to see what's happening :)

                float ma = balls[ball_a].varsF[INDEX_MASS];
                float mb = balls[ball_b].varsF[INDEX_MASS];

                float vafn = (mb*vbin*(cof_E+1) + vain*(ma - cof_E*mb))
                             / (ma + mb);
                float vbfn = (ma*vain*(cof_E+1) - vbin*(ma - cof_E*mb))
                             / (ma + mb);

                // now luckily the tangential components
                // are the same before and after, so
                float vaft = vait;
                float vbft = vbit;
```

```
                // and that's that baby!
                // the velocity vectors are:
                // object a (vafn, vaft)
                // object b (vbfn, vbft)

                // the only problem is that we are in the wrong coordinate
                // system! we need to
               // translate back to the original x,y
                // coordinate system; basically we need to
                // compute the sum of the x components relative to
                // the n,t axes and the sum of
                // the y components relative to the n,t axis,
                // since n,t may both have x,y
                // components in the original x,y coordinate system

                float xfa = vafn*nabx + vaft*tabx;
                float yfa = vafn*naby + vaft*taby;

                float xfb = vbfn*nabx + vbft*tabx;
                float yfb = vbfn*naby + vbft*taby;

                // store results
                balls[ball_a].varsF[INDEX_XV] = xfa;
                balls[ball_a].varsF[INDEX_YV] = yfa;

                balls[ball_b].varsF[INDEX_XV] = xfb;
                balls[ball_b].varsF[INDEX_YV] = yfb;

                // update position
               balls[ball_a].varsF[INDEX_X]+=
                        balls[ball_a].varsF[INDEX_XV];
                balls[ball_a].varsF[INDEX_Y]+=
                        balls[ball_a].varsF[INDEX_YV];

                balls[ball_b].varsF[INDEX_X]+=
                        balls[ball_b].varsF[INDEX_XV];
                balls[ball_b].varsF[INDEX_Y]+=
                        balls[ball_b].varsF[INDEX_YV];

                } // end if

             } // end for ball2

         } // end for ball1

    } // end Collision_Response
```

The code follows the algorithm almost identically. However, the code is from a demo that simulates a pool table system, so added loops test all collision pairs. Once inside the loop the code follows the math. To see the algorithm in action check out `DEMO13_8.CPP|EXE (16-bit version, DEMO13_8_16B.CPP|EXE)`. A screen shot from it is shown in Figure 13.37. The demo starts up with a number of randomly moving balls and then the physics takes over. At the bottom of the screen the total kinetic energy is displayed. Try changing the coefficient of restitution with the right and left arrows keys and watch what happens. For values less than 1, the system loses energy, for values equal to 1, the system maintains energy, and for values greater than 1, the system gains energy—I wish my bank account did that!



```
ELASTIC Object-Object Collision Response DEMO, Press <ESC> to Exit.
Coefficient of Restitution e=1.000000, use <RIGHT>, <LEFT> arrow to change.
Total System Kinetic Energy Sum[1/2MiVi^2]=66.047500
```

FIGURE 13.37
The hyper-realistic collision response model.

Simple Kinematics

The term *kinematics* means a lot of things. To a 3D artist it means one thing, to the 3D game programmer it means another, and to a physicist it means yet another. However, in this section of the book it means the mechanics of moving linked chains of rigid bodies. In computer animation there are two kinematic problems. The first is called *forward kinematics* and the second is *inverse kinematics*. The forward kinematic problem is shown in Figure 13.38; here you see a 2D serially linked chain of rigid bodies (straight arms). Each joint can freely rotate in the plane, thus there are 2 degrees of freedom in this example, θ_1, and θ_2. In addition, each arm has length l_1 and l_2, respectively. The forward kinematic problem can be stated as follows:

Given θ_1, θ_2, l_1, and l_2, find the position of p_2.

Why are we interested in this? Well, if you are going to write a 2D or 3D game and want to have real-time models that have links that move around, then you better know how to do this. For example, 3D animation is accomplished in two ways. The quick and dirty method is to have a set of meshes that represent the 3D animation of an

object. The more flexible method is to have a single 3D mesh that has a number of joints and arms and then to "play" motion data through the 3D model. However, to do this you must understand the physics/mechanics of how to move a hand in relation to the wrist, in relation to the elbow, in relation to the shoulder, in relation to the hips, and so forth—see the problem?

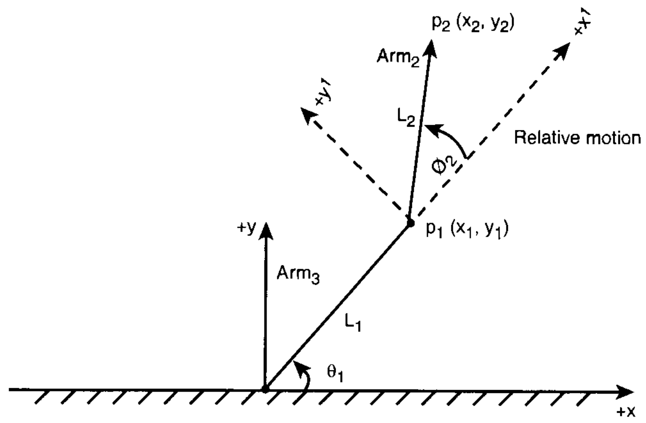


FIGURE 13.38
The forward kinematic problem.

The second kinematic problem is the converse of the first:

Given the position of p_2, find values θ_1, θ_2 that satisfy all the constraints l_1 and l_2 of the physical model. This is much harder than you can imagine. Figure 13.39 shows an example of why this is true.

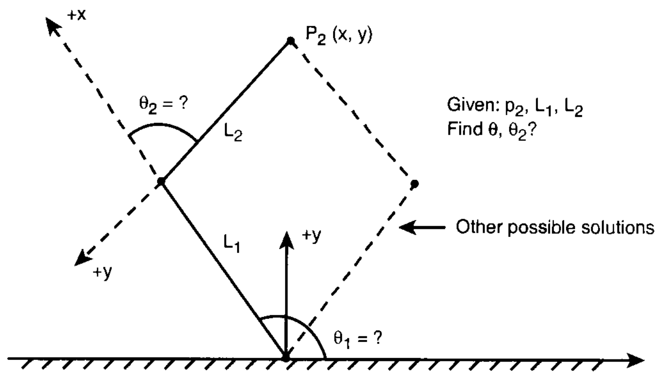


FIGURE 13.39
The inverse kinematic problem.

Referring to the figure, you see that there are two possible solutions that satisfy all the constraints. I'm not going to tackle this problem in general since in most cases we never need to solve it, and the math is gnarly, but I will give an example later in the section of how to go about it.

Solving the Forward Kinematic Problem

What I want to do is show you how to solve the forward kinematic problem because it's almost trivial. Referring back to Figure 13.38, the problem is nothing more than relative motions. If you look at the problem from joint 2, then locating p_2 is nothing more than a translation of l_2 and a rotation of θ_2. However, the point p_2 itself is just located as a translation from p_1 of l_1 and a rotation of θ_1. Thus, the solution of the problem is nothing more than a series of translations and rotations from frame to frame or link to link. Let's solve the problem in pieces.

Forget about the first arm and just focus on the second, that is, let's work our way backward. The starting point is p_1 and we want to move a distance l_2 out on the x-axis and then rotate an angle θ_2 around the x,y plane (or the z-axis in 3D) to locate the point p_2. This is easy—all we need to do is the following transformation from p_1:

```
p2 = p1*Tl2*Rθ2
```

But we don't have p_1? That's okay—we assume that we do for the derivation. Anyway, $\mathbf{T}_{l2}$ and $\mathbf{R}\theta_2$ are the standard 2D translation and rotation matrices you learned about in Chapter 8, "Vector Rasterization and 2D Transformations." Therefore, we have

```
      |1  0  0|        | cos θ2 sin θ2   0|
Tl2 = |0  1  0|  Rθ2 = |-sin θ2 cos θ2   0|
      |l1 0  1|        |  0       0      1|
```

Therefore, p_2 is the product:

```
          |1  0  0|   | cos θ2 sin θ2  0|
p2 = p1 * |0  1  0| * |-sin θ2 cos θ2  0|
          |l2 0  1|   |  0       0     1|
```

Okay, if we can compute p_2 from p_1, then p_1 should be p_0 transformed in the same way, that is, translated by l_1 and rotated by θ_1. Or mathematically:

```
p2 = p0*Tl2*Rθ2*Tl1*Rθ1
```

where $p_0 = [0,0,1]$, that is, the origin in homogenous 2D coordinates (we could use any point if we wanted, but basically this represents the base of the kinematic chain). The reason for three components in a 2D system is so we can use homogenous transforms and accomplish translation with matrices, hence that last 1.0 is a place holder. All points are in the form (x,y,1). Furthermore, $\mathbf{T}_{l1}$ and $\mathbf{R}\theta_1$ are of the same form as T_{l2} and $R\theta_2$, but with different values. Note the order of multiplication—since we're working backward, we must first transform p_0 by $\mathbf{T}_{l2}$*$\mathbf{R}\theta_2$ then by $\mathbf{T}_{l1}$*$\mathbf{R}\theta_1$, so order counts!

With all that in mind, we see that the point p_2 is really just the starting point p_0 multiplied by the matrices ($\mathbf{T}_{l2}$*$\mathbf{R}_{\theta 2}$) * ($\mathbf{T}_{l1}$*$\mathbf{R}_{\theta 1}$). This holds for as many links as needed, or in general:

```
pn = p0 * Tn*Rn * Tn-1*Rn-1 * Tn-2*Rn-2 *...* T1*R1
```

for n links.

This works because each matrix multiplication pair T*R transforms the coordinate system relative to the link, hence, the products of these transforms is like a sequence of changing coordinate systems that you can use to locate the end point. As an example, let's see if this mumbo jumbo works. Figure 13.40 depicts a carefully worked out version of the problem on graph paper.

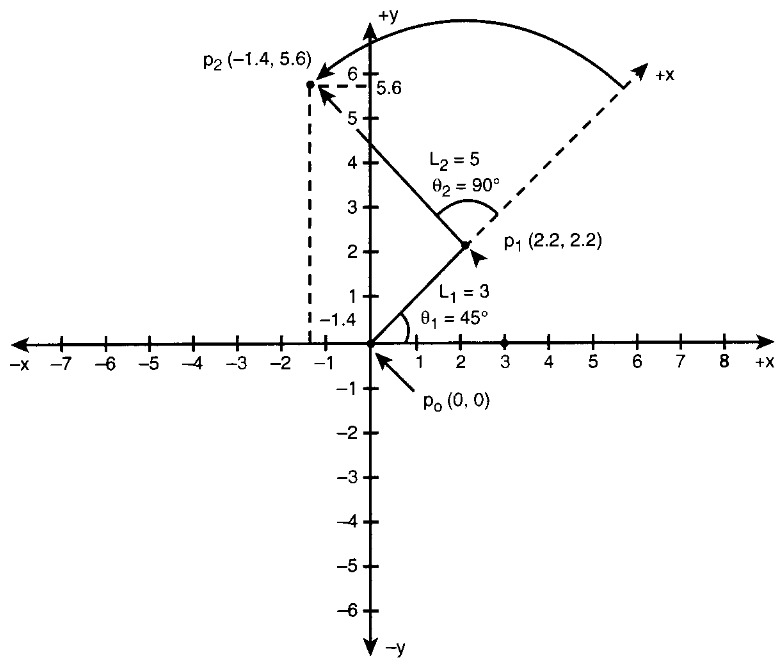


FIGURE 13.40
A kinematic chain worked out on paper.

I have labeled the points, angles, and so on, and using a compass and ruler computed the position of p_2 given the input values:

```
l1 = 3, l2 = 5
θ1 = 45, θ2 = 90
p0 = (0,0)
```

I roughly estimate from the figure that

```
p2 = (-1.4,5.6)
```

Now, let's see if the math gives us the same answer.

```
                |1 0 0| | 0 1 0|  |1 0 0| | .707 .707 0|
p2 = [0,0,1]*|0 1 0|*|-1 0 0|* |0 1 0|*|-.707 .707 0|
                |5 0 1| | 0 0 1|  |3 0 1| |  0    0   1|
      p0        T12      R_2       T11        R_1
```

```
                | 0  1  0|  * | .707  .707  0|
   = [0 0 1]*|-1  0  0|  * |-.707  .707  0|
                | 0  5  1|  * |2.121 2.121  1|
       p0          T12*Rθ2          T11*Rθ1

                |-.707    .707   0|
   = [0 0 1]*|.707     .707   0|
                |-1.414 5.656   1|
       p0           T12*Rθ2*T11*Rθ1

p2 =[-1.414, 5.656,1]
```

Discarding the 1.0 since [x,y,1] really means, x' = x/1, y'=y/1, or x'=x, y'=y, we have:

`p2 =(-1.414, 5.656).`

If you look at Figure 13.40, it looks pretty close! That's all there is to forward kinematics in 2D. Of course, doing it in 3D is a bit more complex due to the z-axis, but as long as you pick a rotation convention then it all works out. I created DEMO13_9.CPP|EXE (16-bit version, DEMO13_9_16B.CPP|EXE) shown in Figure 13.41, as an example of forward kinematics. It lets you change the angle of the two links and then computes the positions p1, p2 and displays them. The keys A, S, D, and F control the angles of link 1 and link 2, respectively. See if you can add a restraint to the program, so the end effector at p_2 can't drop below the y=0 axis, shown by the green line.

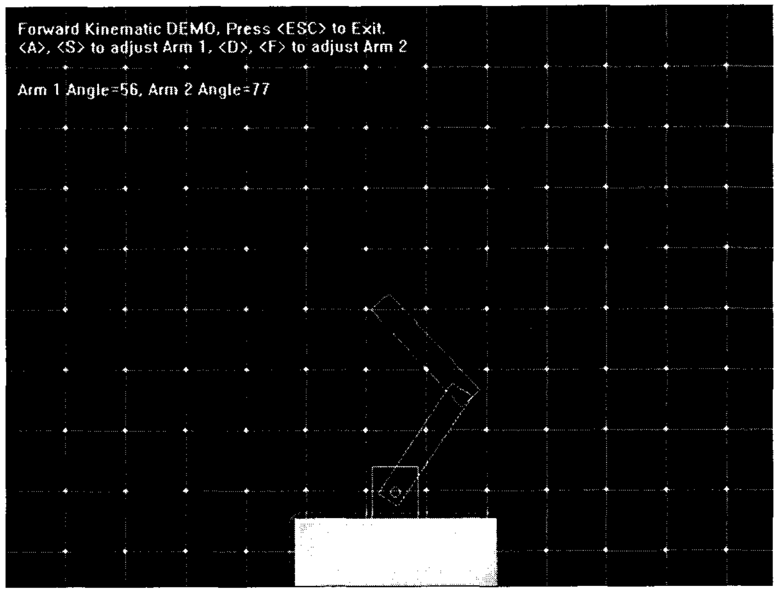

```
Forward Kinematic DEMO, Press <ESC> to Exit.
<A>, <S> to adjust Arm 1, <D>, <F> to adjust Arm 2

Arm 1 Angle=56, Arm 2 Angle=77
```

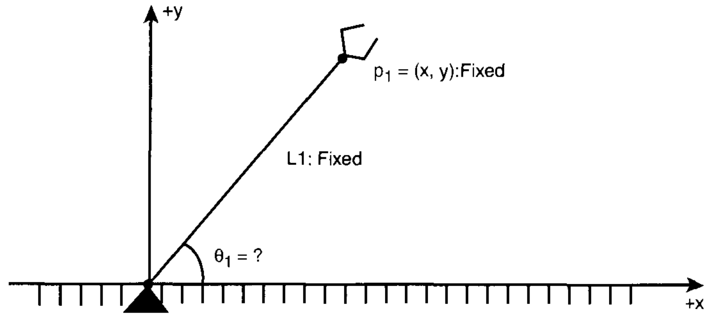
FIGURE 13.41
The kinematic chain demo.

Solving the Inverse Kinematic Problem

Solving inverse kinematics is rather complex in general, but I want to give you a taste of it so you can at least know where to start. The previous section solved for p_2 knowing p_0, l_1, l_2, θ_1, θ_2. But what if you didn't know θ_1, and θ_2, but knew p_2? The solution of the kinematic problem can be found by setting up a system of restraint equations and then solving for the unknown angles. The problem is that you may have an underdetermined system, meaning that there is more than one solution. Thus, you must add other heuristic or constraints to find the solution you want.

As an example, let's try a simpler problem with only one link, so you can see the process. Figure 13.42 shows one link l_1 making an angle θ_1 with the x-axis. Given $p_1(x_1,y_1)$, what is θ_1?

FIGURE 13.42
A single link inverse kinematic problem.



We can use the forward kinematic matrices to solve the problem like this:

```
p1 = p0*T11*Rθ1
                    |1  0  0| | cos θ1 sin θ1  0|
p1(x1,y1) =[0 0 1]*|0  1  0|*|-sin θ1 cos θ1  0|
                    |l1 0  1| |  0      0      1|
              p0       T11          Rθ1

                      | cos θ1 sin θ1  0|
           =[l1 0 1]*|-sin θ1 cos θ1  0|
                      |  0      0      1|
            p0*T11          Rθ1

p1(x1,y1)   =  (l1*cos θ1, l1*sin θ1,  1)
```

Therefore,

```
x1 = l1*cos θ1
y1 = l1*sin θ1

θ1 = cos -1 x1/l1
```

or,

```
θ1 = sin -1 y1/l1
```

MATH

I could have kept the entire problem in matrix form, but this is more illustrative.

Okay, this system is overdetermined; in other words, once you select x or y then the other is determined via θ_1. This is interesting, but if you think about it then it makes sense—the arm link l_1 causes us to lose a degree of freedom therefore, you can't locate any point you wish (x,y) anymore, in fact, the only points that are valid anymore are of the form:

$x_1 = l_1 * \cos \theta_1$
$y_1 = l_1 * \sin \theta_1$

If this had two links, then you would see that for any x,y there would be more than one solution set θ_1, θ_2 that satisfied the equations along with a relationship between θ_1, and θ_2.

Particle Systems

This is *the* hot topic. Everyone is always saying, "So, does it have particle systems?" Well, particle systems can be very complex or very simple. Basically, particle systems are physics models that model small particles. They are great for explosions, vapor trails, and general light shows in your game. You have already learned a lot about physics modeling and I'm sure can create your own particle system. However, just to get you started, I'm going to show you how to create a very quick and simple system based on pixel-sized particles.

Let's say that we want to use particles for explosions, and maybe vapor trails. Since a particle system is nothing more than *n* particles, let's just focus on the model of a single particle.

What Every Particle Needs

If you wanted, you could model collision response, momentum transfer, and all that stuff, but most particle systems have extremely simple models. The following are the general features of a garden variety particle:

- Position
- Velocity
- Color/animation
- Life span
- Gravity
- Wind force

When you start a particle, you will want to give it a position, initial velocity, color, and a life span at the very least. Also, the particle might be a glowing cinder, so there might be color animation involved. Additionally, you may want to have some global forces that act on all particles, like gravity and wind. You may want to have functions that create collections of particles with the desired initial conditions that you're looking for, like explosions or vapor trails. And of course, you may want to give particles the ability to bounce off objects with some physical realism. However, most of the time particles just tunnel through everything and no one cares!

Designing a Particle Engine

To design a particle system you need three separate elements:

- The particle data structure.
- The particle engine that processes each particle.
- Functions to generate particular particle initial conditions.

Let's begin with the data structure. I'll assume an 8-bit display since, in animation, it's easier to work with *bytes* than RGB colors. Also, color effects are easier to implement in 8-bit color. Converting the particle engine to 16-bit isn't bad, but lots of the effects will be lost, so I have decided to keep it 8-bit for illustrative purposes. Anyway, here's a first attempt at a single particle:

```
// a single particle
typedef struct PARTICLE_TYP
        {
        int state;           // state of the particle
        int type;            // type of particle effect
        float x,y;           // world position of particle
        float xv,yv;         // velocity of particle
        int curr_color;      // the current rendering color of particle
        int start_color;     // the start color or range effect
        int end_color;       // the ending color of range effect
        int counter;         // general state transition timer
        int max_count;       // max value for counter

        } PARTICLE, *PARTICLE_PTR;
```

Let's add in some globals to handle external effects such as gravity in the Y direction and wind force in the X direction.

```
float particle_wind = 0;   // assume it operates in the X direction
float particle_gravity = 0; // assume it operates in the Y direction
```

Let's define some useful constants that we might need to accomplish some of the effects:

```
// defines for particle system
#define PARTICLE_STATE_DEAD               0
#define PARTICLE_STATE_ALIVE              1
```

```
// types of particles
#define PARTICLE_TYPE_FLICKER        0
#define PARTICLE_TYPE_FADE           1

// color of particle
#define PARTICLE_COLOR_RED           0
#define PARTICLE_COLOR_GREEN         1
#define PARTICLE_COLOR_BLUE          2
#define PARTICLE_COLOR_WHITE         3

#define MAX_PARTICLES                128

// color ranges (based on my palette)
#define COLOR_RED_START              32
#define COLOR_RED_END                47

#define COLOR_GREEN_START            96
#define COLOR_GREEN_END              111

#define COLOR_BLUE_START             144
#define COLOR_BLUE_END               159

#define COLOR_WHITE_START            16
#define COLOR_WHITE_END              31
```

Hopefully, you can see my thinking here. I want to have particles that are either red, green, blue, or white, so I took a palette and figured out the color indices for the ranges. If you wanted to use 16-bit color then you would have to manually interpolate RGB from the starting value to some ending value—I'll keep it simple. Also, you see that I'm counting on making two types of particles: fading and flickering. The fading particles will just fade away, but the flickering ones will flicker away, like sparks.

Finally, I'm happy with our little particles, so let's create storage for them:

```
PARTICLE particles[MAX_PARTICLES]; // the particles for the particle engine
```

So let's start writing the functions to process each particle.

The Particle Engine Software

We need functions to initialize all the particles, start a particle, process all the particles, and then clean up all the particles when we're done. Let's start with the initialization functions:

```
void Init_Reset_Particles(void)
{
// this function serves as both an init and reset for the particles

// loop thru and reset all the particles to dead
for (int index=0; index<MAX_PARTICLES; index++)
    {
    particles[index].state = PARTICLE_STATE_DEAD;
    particles[index].type  = PARTICLE_TYPE_FADE;
```

```
    particles[index].x     = 0;
    particles[index].y     = 0;
    particles[index].xv    = 0;
    particles[index].yv    = 0;
    particles[index].start_color = 0;
    particles[index].end_color   = 0;
    particles[index].curr_color  = 0;
    particles[index].counter     = 0;
    particles[index].max_count   = 0;
    } // end if

} // end Init_Reset_Particles
```

`Init_Reset_Particles()` just makes all particles zeros and gets them ready for use. If you wanted to do anything special, this would be the place to do it. The next function we need is something to start a particle with a given set of initial conditions. We will worry how to arrive at the initial conditions in a moment, but for now I want to hunt for an available particle, and if found, start it up with the sent data. Here's the function to do that:

```
void Start_Particle(int type, int color, int count,
                    float x, float y, float xv, float yv)
{
// this function starts a single particle

int pindex = -1; // index of particle

// first find open particle
for (int index=0; index < MAX_PARTICLES; index++)
    if (particles[index].state == PARTICLE_STATE_DEAD)
       {
       // set index
       pindex = index;
       break;
       } // end if

// did we find one
if (pindex==-1)
   return;

// set general state info
particles[pindex].state = PARTICLE_STATE_ALIVE;
particles[pindex].type  = type;
particles[pindex].x     = x;
particles[pindex].y     = y;
particles[pindex].xv    = xv;
particles[pindex].yv    = yv;
particles[pindex].counter     = 0;
particles[pindex].max_count   = count;

// set color ranges, always the same
   switch(color)
         {
```

```
        case PARTICLE_COLOR_RED:
             {
             particles[pindex].start_color = COLOR_RED_START;
             particles[pindex].end_color   = COLOR_RED_END;
             } break;

        case PARTICLE_COLOR_GREEN:
             {
             particles[pindex].start_color = COLOR_GREEN_START;
             particles[pindex].end_color   = COLOR_GREEN_END;
             } break;

        case PARTICLE_COLOR_BLUE:
             {
             particles[pindex].start_color = COLOR_BLUE_START;
             particles[pindex].end_color   = COLOR_BLUE_END;
             } break;

        case PARTICLE_COLOR_WHITE:
             {
             particles[pindex].start_color = COLOR_WHITE_START;
             particles[pindex].end_color   = COLOR_WHITE_END;
             } break;

        break;

        } // end switch

// what type of particle is being requested
if (type == PARTICLE_TYPE_FLICKER)
   {
    // set current color
    particles[index].curr_color
    = RAND_RANGE(particles[index].start_color,
                 particles[index].end_color);

   } // end if
else
   {
   // particle is fade type
   // set current color
   particles[index].curr_color  = particles[index].start_color;
   } // end if

} // end Start_Particle
```

NOTE

There is no error detection or even a success/failure sent back. The point is that I don't care; if we can't create one teenie-weenie particle, I think I'll live. However, you might want to add more robust error handling.

To start a particle at (10,20) with an initial velocity of (0, -5) (straight up), a life span of 90 frames, colored a fading green, this is what you would do:

```
Start_Particle(PARTICLE_TYPE_FADE,    // type
               PARTICLE_COLOR_GREEN,  // color
               90,                    // count, lifespan
               10,20,                 // initial position
               0,-5);                 // initial velocity
```

Of course, the particle system has both gravity and wind that are always acting, so you can set them anytime you want and they will globally affect all particles already online as well as new ones. Thus if you want no wind force but a little gravity, you would do this:

```
particle_gravity = 0.1; // positive is downward
particle_wind    = 0.0; // could be +/-
```

Now we have to decide how to move and process the particle. Do we want to wrap them around the screen? Or, when they hit the edges, should we kill them? This depends on the type of game; 2D, 3D, scrolling, and so on. For now let's keep it simple and agree that when a particle goes off a screen edge it's terminated. In addition, the movement function should update the color animation, test if the life counter is expired, and kill particles that are off the screen. Here's the movement function that takes into consideration all that, along with the gravity and wind forces:

```
void Process_Particles(void)
{
// this function moves and animates all particles

for (int index=0; index<MAX_PARTICLES; index++)
    {
    // test if this particle is alive
    if (particles[index].state == PARTICLE_STATE_ALIVE)
       {
       // translate particle
       particles[index].x+=particles[index].xv;
       particles[index].y+=particles[index].yv;

       // update velocity based on gravity and wind
       particles[index].xv+=particle_wind;
       particles[index].yv+=particle_gravity;

       // now based on type of particle perform proper animation
       if (particles[index].type==PARTICLE_TYPE_FLICKER)
          {
          // simply choose a color in the color range and
          // assign it to the current color
          particles[index].curr_color =
            RAND_RANGE(particles[index].start_color,
                       particles[index].end_color);
            // now update counter
            if (++particles[index].counter >= particles[index].max_count)
               {
               // kill the particle
               particles[index].state = PARTICLE_STATE_DEAD;

               } // end if

            } // end if
         else
            {
            // must be a fade, be careful!
            // test if it's time to update color
            if (++particles[index].counter >= particles[index].max_count)
               {
                // reset counter
                particles[index].counter = 0;

               // update color
               if (++particles[index].curr_color >
                                particles[index].end_color)
                  {
                  // transition is complete, terminate particle
                  particles[index].state = PARTICLE_STATE_DEAD;

                  } // end if

               } // end if

            } // end else

        // test if the particle is off the screen?
        if (particles[index].x > screen_width ||
            particles[index].x < 0 ||
            particles[index].y > screen_height ||
            particles[index].y < 0)
           {
           // kill it!
           particles[index].state = PARTICLE_STATE_DEAD;
           } // end if

        } // end if

    } // end for index

} // end Process_Particles
```

The function is self-explanatory—I hope. It translates the particle, applies the external forces, updates the counters and color, tests whether the particle has moved offscreen, and that's it. Next we need to draw the particles. This can be accomplished in a number of ways, but I'm assuming simple pixels and a back buffered display, so here's a function to do that:

```
void Draw_Particles(void)
{
// this function draws all the particles

// lock back surface
DDraw_Lock_Back_Surface();

for (int index=0; index<MAX_PARTICLES; index++)
    {
    // test if particle is alive
    if (particles[index].state==PARTICLE_STATE_ALIVE)
       {
       // render the particle, perform world to screen transform
       int x = particles[index].x;
       int y = particles[index].y;

       // test for clip
       if (x >= screen_width || x < 0 || y >= screen_height || y < 0)
          continue;

       // draw the pixel
       Draw_Pixel(x,y,particles[index].curr_color,
                  back_buffer, back_lpitch);

      } // end if

    } // end for index

// unlock the secondary surface
DDraw_Unlock_Back_Surface();

} // end Draw_Particles
```

Getting exited, huh? Want to try it out, don't you? Well, we're almost done. Now we need some functions to create particle effects like explosions and vapor trails.

Generating the Initial Conditions

Here's the fun part. You can go wild with your imagination. Let's start off with a vapor trail algorithm. Basically, a vapor trail is nothing more than particles that are emitted from a source positioned at (`emit_x`, `emit_y`) with slightly different life spans and starting positions. Here's a possible algorithm:

```
// emit a particle every with a change of 1 in 10
if ((rand()%10) == 1)
{
Start_Particle(PARTICLE_TYPE_FADE,   // type
               PARTICLE_COLOR_GREEN, // color
               RAND_RANGE(90,150),   // count, lifespan
               emit_x+RAND_RANGE(-4,4),  // initial x
               emit_y+RAND_RANGE(-4,4),  // initial y
```

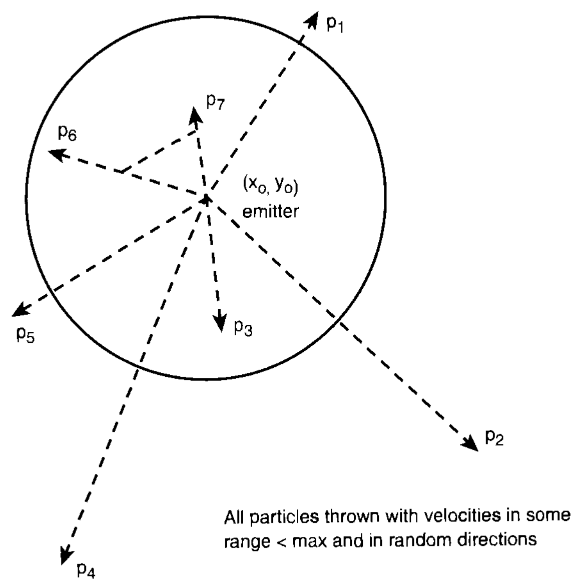
```
                RAND_RANGE(-2,2),    // initial x velocity
                RAND_RANGE(-2,2));   // initial y velocity

} // end if
```

As the emitter moves, so does the emitter source (`emit_x, emit_y`) and therefore a vapor trail is left. If you want to get really real and give the vapor particles an even more realistic physics model you should take into consideration that the emitter could be in motion and thus any particle emitted would have final velocity = emitted velocity + emitter velocity. You would need to know the velocity of the emitter source, (call it (`emit_xv, emit_yv`)) and simply add it to the final particle velocity like this:

```
// emit a particle every with a change of 1 in 10
if ((rand()%10) == 1)
{
Start_Particle(PARTICLE_TYPE_FADE,    // type
               PARTICLE_COLOR_GREEN, // color
               RAND_RANGE(90,150),   // count, lifespan
               emit_x+RAND_RANGE(-4,4),  // initial x
               emit_y+RAND_RANGE(-4,4),  // initial y
               emit_xv+RAND_RANGE(-2,2),   // initial x velocity
               emit_yv+RAND_RANGE(-2,2));  // initial y velocity

} // end if
```

For something a little more exciting, let's model an explosion. An explosion looks something like Figure 13.43. Particles are emitted in a spherical shape in all directions.

FIGURE 13.43
The particles of an explosion.


p1
p7
p6
(x0, y0)
emitter
p5
p3
p2
p4
All particles thrown with velocities in some range < max and in random directions

That's easy enough to model. All we need do is start up a random number of particles from a common point with random velocities that are equally distributed in a circular radius. Then if gravity is on, the particle will fall toward Earth and either go off the screen or die out due to its individual life span. Here's the code to create a particle explosion:

```
void Start_Particle_Explosion(int type, int color, int count,
                              int x, int y, int xv, int yv,
                              int num_particles)
{
// this function starts a particle explosion
// at the given position and velocity
// note the use of look up tables for sin,cos

while(--num_particles >=0)
    {
    // compute random trajectory angle
    int ang = rand()%360;

    // compute random trajectory velocity
    float vel = 2+rand()%4;

    Start_Particle(type,color,count,
                   x+RAND_RANGE(-4,4),y+RAND_RANGE(-4,4),
                   xv+cos_look[ang]*vel, yv+sin_look[ang]*vel);

    } // end while

} // end Start_Particle_Explosion
```

`Start_Particle_Explosion()` takes the type of particle you want (`PARTICLE_TYPE_FADE, PARTICLE_TYPE_FLICKER`), the color of the particles, the desired number of particles, along with the position and velocity of the source. The function then generates all the desired particles.

To create other special effects, just write a function. For example, one of the coolest effects that I like in movies is the ring-shaped shock wave when a spaceship is blown away. Creating this is simple. All you need to do is modify the explosion function to start all the particles out with exactly the same velocity, but at different angles. Here's that code:

```
void Start_Particle_Ring(int type, int color, int count,
                         int x, int y, int xv, int yv,
                         int num_particles)
{
// this function starts a particle explosion at the
// given position and velocity
// note the use of look up tables for sin,cos

// compute random velocity on outside of loop
float vel = 2+rand()%4;
while(--num_particles >=0)
    {
    // compute random trajectory angle
    int ang = rand()%360;

    //start the particle
    Start_Particle(type,color,count,
                   x,y,
                   xv+cos_look[ang]*vel,
                   yv+sin_look[ang]*vel);

    } // end while

} // end Start_Particle_Ring
```

Putting the Particle System Together

You now have everything you need to put together some cool particle effects. Just make a call in the initialize phase of your game to `Init_Reset_Particles()`, then in the main loop make a call to `Process_Particles()`. Each cycle and the engine will do the rest. Of course, you have to call one of the generator functions to create some particles! Lastly, if you want to improve the system you might want to add better memory management so you can have infinite particles, and you might want to add particle-to-particle collision detection and particle-to-environment collision detection—that would be really cool.

As a demo of using the particle system, take a look at `DEMO13_10.CPP|EXE` (16-bit version not available) on the CD. It is a fireworks display based on the tank projectile demo. Basically, the tank from the previous demo fires projectiles now. Also, note in the demo I jacked the number of particles up to 256.

Playing God: Constructing Physics Models for Games

This chapter has given you a lot of information and concepts to sift through. The key is to use the concepts and some of the hard math to make working models that look good. No one will ever know if they accurately simulate reality 100 percent, nor will they care. If you can make an approximation then do it—as long as it's worth it. For example, if you're trying to make a racing game and you want to race on road, ice, and dirt, then you better have some frictional effects, otherwise, your cars will drive like they're on rails!

On the other hand, if you have an asteroid field that the player blows up and each asteroid splits into two or more smaller asteroids then I don't think the player is going to care or know for that matter the exact trajectory that the smaller asteroids would take—just pick them in a deterministic way so they look good.

Data Structures for Physics Modeling

One of the questions that I'm asked continuously (in addition to how to compile DirectX programs with VC++) is what data structures to use for physics modeling. There are no physics-data structures! Most physics models are based on the game objects themselves—you simply need to add enough data elements to your primary data structures to figure the physics out—get it? Nonetheless, you should keep track of the following parameters and values in any physics engine for the universe and objects:

- Position and velocity of the object.
- Angular velocity of the object.
- Mass, frictional coefficient, and any other physical properties of the object.
- Physics engine geometry for the object. This is simply a geometry that can be used for the physics calculations. You may use rectangles, spheres, or whatever, rather than the actual object geometry.
- External universal forces such as wind, gravity, and so on.

Now it's up to you to represent all these values with whatever structures or types are appropriate. For example, the realistic collision response demo used a model something like this:

Each Ball Object

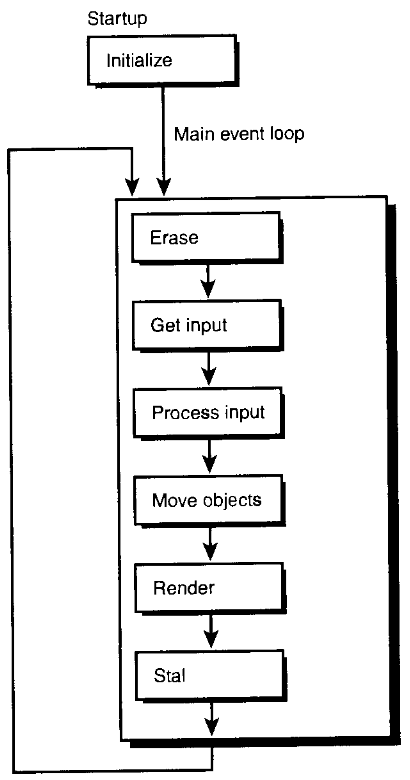
```
float x,y; // position
float xv,yv; // velocity
float radius; // guess?
float coefficient_of_restitution; // just what it says
```

Of course the data was hidden a little in some internal arrays in each of the BOBs (blitter objects) that represented the balls, but the abstract data structure is what we're interested in.

Frame-Based Versus Time-Based Modeling

This is the final topic I want to talk about since it's becoming more and more important in 3D games. Thus far in the book, we have had a game loop that looks like Figure 13.44. We have been assuming that the game will run at a constant rate of R fps. If it doesn't, then no big deal, everything will slow down on the screen. But what if you didn't want things to slow down on the screen? What if you wanted a ship to move from a to b in two seconds no matter what the frame rate was? This is called *time-based modeling*.

FIGURE 13.44
The game loop you've learned to love.



Time-based modeling differs from frame-based modeling in that time t is used in all the kinematic equations that move objects. For example, in a frame-based game if you have the chunk of code:

```
x = 0, y = 0;

x = x + dx;
y = y + dy;
```

and the game runs at 30 fps, then in 30 frames or 1 second, x,y will equal

```
x = 30*dx;
y = 30*dy;
```

If dx=1 and dy=0 then the object would move exactly 30 pixels in the x-direction. And this is fine if you can always guarantee a constant frame rate. But what if the frame rate drops to 10 fps? Then in 1 second, you will have

```
x = 10*dx = 10
y = 10*dy = 0
```

x only changed one-third of what you wanted it to! If visual continuity is what you are going for then this is unacceptable. In addition, this can wreak havoc on a network game. In a network game you can either sync to the slowest machine and stay frame-based, or you can let all the machines run free and use time modeling, which is the more realistic and fair thing to do. I shouldn't have to pay for someone else owning a 486 when I have a PIV 2.4GHz.

To implement time-based motion and kinematics, you have to use time in all of your motion equations. Then, when it's time to move objects, you have to test the difference in time from the last movement and use this as the input into your equations. Thus, as the game slows down a lot, it won't matter because the time parameter will convey this and cause a larger motion. Here's an example game loop:

```
while(1)
{
t0 = Get_Time(); // assume this is in milliseconds

// work, work, work

// move objects
t1 = Get_Time();

// move all the objects
Move_Objects(t1 - t0);

// render
Render();

} // end while
```

With this loop we use the change in time or (t_1-t_0) as an input to the motion code. Normally there would be no input; the motion code would just move. Let's assume that we want our object to move at 30 pixels per second, but since our time base is in milliseconds, or $1\text{x}10^{-3}$ seconds, we need to scale this:

```
dx = 30 pixels/1 sec = .03 pixels/1 millisecond
```

Can you see where I'm going? Let's write the motion equation for x:

```
x = x + dx*t
```

Plugging everything in we get

```
x = x +.03*(t1 - t0)
```

That's it. If a single frame takes one second then (t_1 – t_0) will be 1000 milliseconds and the motion equation will equal

```
x = x + .03*1000 = 30, which is correct!
```

On the other hand, if the time for this frame takes three milliseconds then the motion equation will look like

```
x = x + .03*3 = .09, which is also correct!
```

Obviously you need to use floating-point values for all this to work since you are going to be tracking fractions of pixel motion, but you get the idea. This is so cool because even if your game starts bogging down hardcore due to rendering, the motion will stay the same.

As an example, check out `DEMO13_11.CPP|EXE (16-bit version, DEMO13_11_16B.CPP|EXE)`; it basically moves a little ship (with a shadow!) from left to right and allows you to alter the delay of each frame using the arrow keys to simulate processor load.

Notice that the ship moves at a constant rate. It may jump, but it will always travel a total distance of 50 pixels/sec no matter what the frame rate. As a test, the screen is 640 pixels wide, plus the off-screen overlap of 160 pixels, thus the entire travel is 800 pixels. Since the ship is traveling at 50 pixels/sec that means that it should take 800/50=16 seconds to make one pass. Try changing the delay and note that this is always true. If your game was designed to run at 60 fps and it slows to 15–30 then the jumping won't be that apparent and the game will look the same, but less smooth. Without time modeling your game WILL slow down and look like it's in slow motion—I'm sure you have seen this many times. :)

Summary

I'm sure this chapter has been somewhat enlightening whatever your background is. It gave me a headache writing it! We covered a lot of ground and learned various ways to look at things. For example, we tried an ad hoc collision response algorithm that worked great for a ball bouncing off a rectangle. This technique is perfect for any kind of pong or breakout game. Then we looked more closely at the mathematics of reflection and derived the correct way to do it in general. This is the point of physics modeling in games—and you just learned how to do it.