4.3 Real-Time Game Physics

In This Chapter

- Overview
- Rewind: A Fresh Look at Basic Physics
- Introduction to Numerical Rigid-body Simulation
- Generalized Translational Motion
- Rotational Motion
- Collision Response Revisited
- Bringing It All Together
- Commercial and Freeware Physics Engines
- Summary
- Exercises
- References

Overview

Physics is part of our life experience. Our brains are conditioned through life to recognize physically based motion as being correct motion. It makes sense, then, that game players become more immersed in some types of games when objects move in a realistic manner. When appropriate, there are a number of ways to create realistic motion for a game. One common approach is for artists to author keyframe animations that give the appearance of being physically based. Another common approach, which is popular for character animation in particular, is to use motion capture technology to record real-life motions and then apply those recorded motions to game models. Both of these approaches are extremely labor intensive, and expensive.

Simulation of physics represents a third approach to generating realistic motion for games. Physics simulation can provide at least two benefits that are significant to
publishers, developers, and game players. The first benefit is a cost savings to developers and publishers. Physics simulation has the potential to be far less expensive than keyframe or motion capture animation, since (ideally) the artist only needs to configure the physical properties of a game model. The simulation rather than an artist or actor determines the actual motion, and does so without charging an hourly rate!

In game physics simulation, there is the possibility of simulating an object's motion within digital content creation software, thus creating a preprocessed solution that is fixed at runtime, just as with keyframe or motion capture animation. There is also the possibility of simulating motion at runtime. The latter case provides the second benefit of physics simulation. By simulating physics at runtime, the game engine can create emergent behavior, leading to a richer game experience for the game player.

This chapter contains a whirlwind introduction to physics simulation, with a focus on techniques that can run in real time. The concepts presented herein are not comprehensive, but are intended to provide sufficient information to implement a wide variety of realistic behaviors. The entire chapter can be useful in generating physics simulation within a runtime game engine, as well as within digital content creation tools.

Rewind: A Fresh Look at Basic Physics

Let's begin slowly. Chances are good that you've already attended a science course, perhaps in high school or during the first year at a university, that introduced you to the fundamentals of physics. It makes sense to reflect back on what you may already know. Those basic equations are easy to code, and can actually be useful in real games. Further, they are a building block upon which we will develop more generalized rigid body physics that can make simulated 3D worlds come to life.

The Importance of Consistent Units

Throughout this chapter, as new variables or quantities are introduced, we will identify units in which the quantity can be measured, from the International System of Units (SI). For example, position can be measured in meters, and time can be measured in seconds. You can choose to use a different system of units, such as the English System of Units, if you wish. However, make no mistake about this: it is necessary that you use consistent units in your equations. Equations will produce the wrong results if you use values with inconsistent units! An example of using inconsistent units would be to use a force measured in pounds (English Units) and an object mass measured in kilograms (SI Units) in the same equation. To correct the inconsistency, convert the force to Newtons, which is consistent with a mass measured in kilograms. Conversion factors are readily available on the Internet, and can be found using any search engine.

Particle Kinematics

Every introduction to physics begins by defining several fundamental properties of motion. It is likely that you already understand these properties implicitly, and find
them to be second nature. Nevertheless, we begin our review by formally defining the properties of particle motion, or *particle kinematics*. They are the basis for everything that follows.

From the point of view of theoretical physics, a particle is an object that has no volume; for example, a particle is a mass concentrated into an infinitely small sphere. For our purposes, however, we define a particle to be a perfectly smooth, frictionless sphere with a finite radius. We choose this definition solely to avoid the need to consider rotational motion, an advanced topic, until later. Perfectly smooth, frictionless spheres will never begin to rotate due to normal interactions with other objects. Figure 4.3.1 illustrates a particle in motion along a curved path.

At any moment in time, \( t \), the particle is located at a position, \( \mathbf{p} \), measured in an *inertial reference frame*. We won’t worry about the formal definition of an inertial reference frame (it has to do with the relative position of the stars in space). For games, we simply choose a game’s world coordinate system to be the inertial reference frame. The position, \( \mathbf{p} \), is the location of the particle, measured in world space, \( \mathbf{p} = (p_x, p_y, p_z) \). Position is measured in units of type *distance* or *length*. The SI units for position are meters (m).

If the particle is moving, its position is a function of time, \( \mathbf{p}(t) \). Figure 4.3.2 illustrates the particle at time \( t \) and at a later time, \( t + \Delta t \). The symbol, \( \Delta \), is the Greek letter Delta, commonly used to indicate a change in value. The quantity, \( \Delta t \), indicates an incremental change in time. *The SI units for time are seconds.*

The vector quantity, velocity, is defined to be the change in position over time. The magnitude of velocity is the particle's speed. *Velocity and speed are measured in units of type distance over time.* The SI units for velocity and speed are meters per second (m/s).

Given the position of a particle at two different times, the average velocity between the start and end time can be computed as 

\[
\mathbf{v}_{\text{avg}} = \frac{(\mathbf{p}(t + \Delta t) - \mathbf{p}(t))}{\Delta t}.
\]

It is possible that the particle’s velocity changes significantly between time \( t \) and \( t + \Delta t \). The average velocity is not necessarily the particle velocity at time \( t \). From calculus, the true, *instantaneous velocity* at time \( t \) is the time derivative of position, shown in Equation 4.3.1. Note also that the inverse is true: position is the integral of velocity over time. (If you are unfamiliar with calculus, and the meaning of the terms derivative and integral, please refer to any introductory calculus text, such as [Munem78].)

---

**FIGURE 4.3.1** Position of a particle in a world coordinate system.

**FIGURE 4.3.2** Time-dependent position of a particle moving along a path.
\[ V(t) = \lim_{\Delta t \to 0} \frac{p(t+\Delta t) - p(t)}{\Delta t} = \frac{d}{dt} p(t) \quad (4.3.1) \]

There is one more fundamental property of particle kinematics that we require. The vector quantity, \textit{acceleration}, defined in Equation 4.3.2, is the time derivative of velocity. Note that the inverse is also true: velocity is the integral of acceleration over time. Acceleration can also be defined as the second time derivative of position. \textit{Acceleration is measured in units of type distance over the square of time. The SI units for acceleration are meters per second squared (m/s²).}

\[ a(t) = \frac{d}{dt} V(t) = \frac{d^2}{dt^2} p(t) \quad (4.3.2) \]

\noindent**Newton's Famous Laws**

We begin our review of basic physics by considering Sir Isaac Newton's First and Second Laws of Motion. \textit{Newton's First Law of Motion}, paraphrased, states that an object will move at a constant velocity until compelled to change its velocity by forces imposed upon the object. From this, we can make a profound observation from the history of video games: the classic arcade and console game, Atari's \textit{Breakout}, actually had a realistic physics model.

\textit{Newton's Second Law of Motion} is important to us, as we can use it to implement physics simulations in the presence of forces that cause an object's velocity and position to change in interesting ways. This law, paraphrased, states that an object's change in velocity is proportional to an applied force. Stated in equation form, the law is the infamous \( \mathbf{F} = m\mathbf{a} \), shown in vector form as Equation 4.3.3 (assumes mass is constant).

\[ \mathbf{F}(t) = m\mathbf{a}(t) \quad (4.3.3) \]

Here, the force, \( \mathbf{F}(t) \), may change over time, resulting in an acceleration that changes over time. The quantity, \( m \), is called the \textit{mass} of the object. \textit{The SI units for mass are kilograms (kg). Force is measured in units of type mass times distance over the square of time. The SI units for force are called Newton's (N).} By definition, 1 Newton is equal to 1 kg·m/s² (note the consistency). Note that mass is not the same as the object's weight (which is a force)!

\noindent**The Cycle of Motion**

You should be able to see a cycle of motion in Equations 4.3.1 through 4.3.3. A force causes acceleration. Acceleration causes a change in velocity. Velocity causes a change in the position of a particle. By integrating the equations in reverse order from 4.3.3 to 4.3.1, we can determine the motion of the particle. This is what physics simulation is all about.
The Effect of a Constant Force on Particle Motion

As described by Newton's First Law of Motion, the simplest possible motion is that of a particle experiencing no force and therefore continuing to move at a constant velocity (zero for a particle at rest). We continue our review by analyzing a slightly more complex case: a particle experiencing a constant force. Consider Equation 4.3.3, when \( F \) is constant. Since the right-hand side is constant in time, acceleration also is a constant, \( a \). Constants are easy to integrate (even vector constants), and by integrating Equation 4.3.3, we derive Equation 4.3.4, a closed-form equation for the velocity of a particle experiencing a constant force. Note that the integration was performed using a change of variable, \( \tau = t - t_{\text{init}} \). The equation is valid for a force that is applied to the particle beginning at time \( t_{\text{init}} \), but not before. The velocity, \( V_{\text{init}} \), is the velocity of the particle at time \( t_{\text{init}} \).

\[
V(t) = \int m d\tau = \int \frac{F}{m} d\tau = V_{\text{init}} + \frac{F}{m}(t - t_{\text{init}}) \quad (4.3.4)
\]

From here, we can derive a closed-form equation for position by integrating Equation 4.3.4. This is shown in Equation 4.3.5, in which \( p_{\text{init}} \) is the initial position of the particle, at time \( t_{\text{init}} \).

\[
p(t) = \int V(\tau) d\tau = \int \left( V_{\text{init}} + \frac{F}{m}(\tau) \right) d\tau = p_{\text{init}} + V_{\text{init}}(t - t_{\text{init}}) + \frac{F}{2m}(t - t_{\text{init}})^2 \quad (4.3.5)
\]

It is important that you recognize one fact about Equations 4.3.4 and 4.3.5. This equation is exact and will produce a correct, realistic result for any time \( t > t_{\text{init}} \) as long as the applied force remains a constant. Further, you can use these equations in a piecewise fashion, applying them as long as the applied force remains constant, and then restarting them when the force changes to a different, constant value by updating the values \( t_{\text{init}}, p_{\text{init}}, \) and \( V_{\text{init}} \) to be the current values of \( t, p(t), \) and \( V(t), \) respectively, at the moment the force changes. It is only valid to restart the equations when \( F \) remains constant for some time after every change.

Consistency of Units, Still Important

Let us revisit, for a moment, the important issue of consistency of units. It is insufficient to simply ensure all of the values used in an equation are measured in a consistent system of units such as SI. It is also crucial that all of the terms in an equation measure the same type of unit. For example, as presented previously, velocity is measured in units of the type distance over time. You can verify the consistency of unit types in the rightmost term of Equation 4.3.4, \( \frac{F}{m}(t - t_{\text{init}}) \), by substituting the unit types for the variables and simplifying algebraically, as though the unit types were variables. This verification is shown in Equation 4.3.6, which proves that the units of the rightmost term are, correctly, the unit type for velocity. Note that the difference of two values of
time, \((t - t_{\text{init}})\), does not result in a cancellation of the units of time. The value represented by \((t - t_{\text{init}})\) is a change in time, which also has units of type time. If you have reason to derive new equations yourself, make it your standard practice to analyze the unit types in your derived equation to be sure they are consistent. If you discover that one or more of the terms in your equation are inconsistent, you will know that there is an error in your equation.

\[
\text{Units of } \frac{F}{m} (t - t_{\text{init}}) = \frac{(\text{mass})(\text{dist})}{(\text{mass})(\text{time}^2)} \text{ time} = \frac{(\text{mass})(\text{dist})}{(\text{mass})(\text{time}^2)} \text{ time} = \frac{\text{dist}}{\text{time}} \tag{4.3.6}
\]

**Projectile Motion**

It is actually possible to use Equation 4.3.5 to achieve meaningful physics simulations in games. We continue the review by looking at simple projectile motion in 3D. One approximately constant force acts on all real objects near the surface of a planet. That force is, of course, the force due to gravity. This force is the weight of the object, equal to the object's mass times a constant acceleration due to gravity, acting in the direction of a vector from the object's position toward the center of the planet. The acceleration due to gravity is commonly represented by the variable \(g\). In SI units, the value of \(g\) on Earth is 9.81 \(\text{m/s}^2\) toward the center of the Earth, or the vector, \(g = (0, 0, 0, -9.81)\). Here, we've chosen the up direction to be the positive \(z\)-axis in world space. We can rewrite Equation 4.3.5 for a simple projectile on Earth, as Equation 4.3.7.

\[
p(t) = p_{\text{init}} + v_{\text{init}} (t - t_{\text{init}}) + \frac{1}{2} g (t - t_{\text{init}})^2 \tag{4.3.7}
\]

If \(v_{\text{init}}\) is not parallel to \(g\), the path will be parabolic, as illustrated in Figure 4.3.3.

![Particle undergoing simple projectile motion.](image)

Listing 4.3.1 is a fragment of pseudocode that can be used to simulate a particle undergoing simple projectile motion on Earth, using Equation 4.3.5.
Listing 4.3.1  Pseudo-code for simulating projectile motion of a particle on Earth.

```c
void main()
{
  // Initialize variables used in the simulation.
  Vector3D V_init(10.0, 0.0, 10.0);
  Vector3D p_init(0.0, 0.0, 100.0), p = p_init;
  Vector3D g(0.0, 0.0, -9.81);
  float t_init = particle launch time;

  // The game simulation/rendering loop
  while (game simulation is running)
  {
    float t = current game time;
    if (t > t_init)
    {
      float tmti = t - t_init;
      p = p_init + (V_init * tmti);
      p = p + 0.5 * g * (tmti * tmti);
    }
    render particle at location p;
  }
}
```

Complete Exercise 1 to translate this pseudocode into a simple targeting game.

**Frictionless Collision Response**

We conclude our review of basic physics with a detailed analysis of classical particle-collision response. First, a definition. Linear momentum is defined to be the vector quantity mass times velocity, or \( m \mathbf{V} \). Linear momentum is measured in units of mass times distance over time. The SI units of linear momentum are kilogram-meters per second (kg·m/s).

Linear momentum is related to the force being applied to an object. In fact, its relationship with force is more fundamental than the relationship between force and acceleration shown in Equation 4.3.3. Equation 4.3.3 is actually an approximation to Equation 4.3.8, a more general relationship that defines the first time derivative of linear momentum as being equal to the net force applied to an object.

\[
\frac{d}{dt}(m \mathbf{V}(t)) = \mathbf{F}(t)
\]

(4.3.8)

For most objects, mass is constant, and this enables us to derive Equation 4.3.3 by noting \( \frac{d}{dt}(m \mathbf{V}(t)) = m \frac{d}{dt} \mathbf{V}(t) \) when mass is constant. Equation 4.3.8 is called the Newtonian Equation of Motion, since when integrated over time it determines the motion of an object. By integrating the force applied to an object over time, we obtain the change in linear momentum and velocity over time.
Consider two colliding particles, 1 and 2. For the duration of the collision, each particle exerts a force on the other. The duration of most collisions is an extremely short period of time, and yet the change in velocity of the objects is often dramatic. For example, think about the collision response of billiard balls. Large, nearly instantaneous changes in velocity can only occur if the collision forces are large. Collision forces are usually so large that they dominate during the collision. It is usually acceptable to ignore other forces entirely, assuming their effect is negligible for the duration of the collision. By integrating Equation 4.3.8 over the duration of the collision, we obtain the linear impulse-momentum equation, given as Equation 4.3.9.

\[ m_1 V_1^- = m_1 V_1^+ + \Lambda \]  \hspace{1cm} (4.3.9)

Here, \( m_1 V_1^- \) is the linear momentum of particle 1 just before the collision, and \( m_1 V_1^+ \) is the linear momentum just after the collision. The superscripts, – and +, indicate quantities before and after collisions, respectively. The vector \( \Lambda \) is called the linear impulse, defined to be the integral of the collision force over the duration of the collision.

Newton's Third Law of Motion states that for every action there is an equal but opposite reaction. This law tells us that the collision forces, and the impulses, on the two objects are equal in magnitude but opposite in direction. From this result, we can immediately write the linear impulse-momentum equation for particle 2, as Equation 4.3.10. Note that impulse is negated on the second object—equal but opposite.

\[ m_2 V_2^- = m_2 V_2^+ - \Lambda \]  \hspace{1cm} (4.3.10)

Our goal is to find the velocities of the two objects after the collision response is complete. We can solve Equations 4.3.9 and 4.3.10 for the after-collision velocities if we are able to compute the impulse. The fact that we are assuming a frictionless collision allows us to simplify the situation somewhat. Without friction, the impulse will always act purely along the unit surface normal vector at the point of contact. In this case, \( \Lambda \) can be defined by Equation 4.3.11, where \( \Lambda \) is the scalar value (positive or negative) of the impulse and \( \hat{n} \) is the unit surface normal vector. A collision detection algorithm determines \( \hat{n} \) and the point of contact, as detailed in Chapter 4.2, "Collision Detection and Resolution."

\[ \Lambda = \Lambda \hat{n} \]  \hspace{1cm} (4.3.11)

By substituting Equation 4.3.11 into Equations 4.3.9 and 4.3.10, we obtain two vector equations that together contain three unknowns: the two vectors \( V_1^+ \), \( V_2^+ \), and one scalar \( \Lambda \). We require a third equation before we can solve for all three unknown values. We can generate the third equation by observing the physical behavior of real objects during collisions. Observe the behavior when two objects in real life collide, as shown in Figure 4.3.4. Here, just prior to impact, the objects exhibit their natural
geometric shapes. During the initial impact, both objects experience a *period of deformation* in which their shapes change in response to the collision force. After the initial impact, the objects experience a *period of restitution*, in which they are restored to their natural shapes and accelerate to their after-collision velocities.

![Diagram: Period of Deformation and Period of Restitution](image)

**FIGURE 4.3.4** *A realistic view of collision response.*

The third equation is chosen to be an approximation to the material response of real objects during a collision. Before we present the equation, observe that objects do not always collide while traveling toward each other along collinear paths. The more general situation is illustrated in Figure 4.3.5.

![Diagram: Frictionless collision of two spherical particles](image)

**FIGURE 4.3.5** *Frictionless collision of two spherical particles.*

Equations 4.3.9 and 4.3.10, being vector equations, can be represented in a variety of coordinate systems. One valid choice is the world coordinate system; however, it is more convenient if we consider them in a coordinate system that includes \( \hat{\mathbf{n}} \) and the contact plane that is tangent to the object surfaces at the point of contact. In the general case, with friction, a collision will affect the object velocity components parallel to the contact plane and the component parallel to the contact normal direction. For a frictionless collision, however, the velocity components in the contact plane do not change. Our third equation, given as Equation 4.3.12, reflects this fact and defines the relationship between the normal components of the velocities of the objects, before and after a collision.
\[
(V_1^+ - V_2^+) \cdot \hat{n} = -\epsilon (V_1^- - V_2^-) \cdot \hat{n}
\] (4.3.12)

Here, the scalar \( \epsilon \) is called the coefficient of restitution. This coefficient is related to the conservation or loss of kinetic energy during a collision. Due to space constraints, we will not discuss the concept of energy in detail here; however, you should know that total energy, equal to kinetic energy plus potential energy, is a physical quantity that is conserved similar to linear momentum. If \( \epsilon \) is equal to 1, the collision is perfectly elastic, representing objects that rebound fully so that the sum of the particles' kinetic energies is the same before and after the collision. If \( \epsilon \) is equal to 0, the collision is perfectly plastic, representing objects that experience no period of restitution and a maximum loss of kinetic energy. In real life, \( \epsilon \) is a function of the material properties of the objects involved in the collision. For example, the coefficient of restitution for a collision between a tennis ball and tennis racket is approximately 0.85, and that for a deflated basketball colliding with the court surface is nearly 0.

Using Equations 4.3.9 through 4.3.12, we can solve for the linear impulse, given in Equation 4.3.13. To compute the after-collision velocities, apply the result of Equation 4.3.13 into Equations 4.3.9 and 4.3.10, and divide by \( m_1 \) or \( m_2 \), respectively, to find the after-collision velocities.

\[
\Lambda = -\left( \frac{m_1 m_2 (1 + \epsilon) (V_1^- - V_2^-) \cdot \hat{n}}{m_1 + m_2} \right) \hat{n}
\] (4.3.13)

**The Story So Far**

To this point, we have revisited a few basic concepts in kinematics and physics that you may have seen before. The concepts have been generalized to three dimensions and are ready to use in certain types of games. Depending on the game you are developing, you may not need to read any further. Listing 4.3.2 is a fragment of pseudocode that you can use to simulate a collection of \( N \) spherical particles experiencing gravitational acceleration in a game, with occasional frictionless collisions.

**Listing 4.3.2** Pseudocode for simulating a collection of \( N \) spherical particles under gravity with frictionless collisions.

```c
void main()
{
    // Initialize variables needed by the simulation.
    Vector3D V_init[N] = initial velocities;
    Vector3D p_init[N] = initial center positions;
    Vector3D g(0.0, 0.0, -9.81);
    float mass[N] = particle masses;
    float time_init[N] = per particle start times;
    float eps = coefficient of restitution;
```
// Main game simulation/rendering loop.  
while (game simulation is running) 
{ 
    float t = current game time; 
    detect collisions and collision times; 
    
    // Resolve collisions. 
    for (each colliding pair i, j) 
    { 
        // Calc before collision position and 
        // velocity of obj i (Equations 4 and 5). 
        float tmi = time_collision - time_init[i]; 
        pi = p_init[i] + (V_init[i] * tmi); 
        pi = p + 0.5 * g * (tmi * tmi); 
        vi = V_init[i] + g * tmi; 
        
        // Calc before collision position and 
        // velocity of obj j (Equations 4 and 5). 
        tmj = time_collision - time_init[j]; 
        pj = p_init[j] + (V_init[j] * tmj); 
        pj = p + 0.5 * g * (tmj * tmj); 
        vj = V_init[j] + g * tmj; 
        
        // For spherical particles, surface normal 
        // is just the vector joining their centers. 
        normal = Normalize(pj - pi); 
        
        // Compute the impulse (Equation 13). 
        impulse = normal; 
        impulse *= -(1 + eps) * mass[i] * mass[j]; 
        impulse *= normal.DotProduct(vi - vj); 
        impulse /= (mass[i] + mass[j]); 
        
        // Restart particles i and j immediately 
        // after the collision. (Equation 9 and 10). 
        // Since the collision occurs instantaneously, 
        // the after-collision positions are pi, pj, 
        // the same as the before-collision positions. 
        V_init[i] += impulse / mass[i]; 
        V_init[j] -= impulse / mass[j]; 
        p_init[i] = pi; 
        p_init[j] = pj; 
        
        // Reset the start times since we updated 
        // initial velocities. 
        time_init[i] = time_collision; 
        time_init[j] = time_collision; 
    } 
    
    // Update particle positions (Equation 5) and 
    // render particles 
    for (k = 0; k < N; k++) 
    { 

Complete Exercise 5 to determine how to handle particle collisions with an immovable object (terrain or a wall).

**Introduction to Numerical Rigid-Body Simulation**

The presentation in the prior section provides you with simple equations that you can use to put projectile physics into real games. If your game needs only to simulate spherical objects that do not rotate and that experience only gravity plus an occasional frictionless collision, you can use the pseudocode in Listing 4.3.2 as a basis for your physics system.

In many games, as in real life, the most interesting motion involves forces other than constant and collision impulse forces. Unfortunately, for the general case, closed-form solutions, such as those presented in the prior section as Equations 4.3.4 and 4.3.5, rarely exist. We assign the term **numerical simulation** to a series of techniques that allow us to approximate the motion of objects for which there is no closed-form solution. This section provides a brief introduction to numerical simulation, along with some implementation suggestions.

**Numerical Integration of the Newtonian Equation of Motion**

We will consider a family of numerical simulation techniques called **finite difference methods**. Most finite difference methods are derived using the **Taylor series** expansion of the properties we are interested in. We begin in a generic way. Equation 4.3.14 shows the Taylor Series expansion of a vector property, \( \mathbf{S}(t) \). Equation 4.3.14 is valid, and exact, when \( \mathbf{S}(t) \) is continuous and differentiable on the closed domain, \([t, t + \Delta t]\).

In this context, \( \Delta t \) is called the **time step**, and it represents the incremental intervals at which we will update \( \mathbf{S}(t) \) over time.

\[
\mathbf{S}(t + \Delta t) = \mathbf{S}(t) + \Delta t \frac{d}{dt} \mathbf{S}(t) + \frac{(\Delta t)^2}{2!} \frac{d^2}{dt^2} \mathbf{S}(t) + \sum_{n=3}^{\infty} \frac{(\Delta t)^n}{k!} \frac{d^n}{dt^n} \mathbf{S}(t) \quad (4.3.14)
\]

In general, we will not know the values of any of the higher order time derivatives of a property. We can convert Equation 4.3.14 into a truncated Taylor series, shown in Equation 4.3.15, by simply removing the terms involving \( \frac{d^2 \mathbf{S}}{dt^2} \) and higher order...
derivatives. In some cases, the truncation may occur beyond \( \frac{d^2S}{dt^2} \), but it is always the higher order terms that are eliminated.

\[
S(t + \Delta t) = S(t) + \Delta t \frac{d}{dt} S(t) + O(\Delta t^2)
\]  
(4.3.15)

By removing the higher order terms, we introduce numerical error into the equation that is equal in magnitude, opposite in sign to the higher order terms. This error is called the truncation error. The largest component of truncation error is usually the term with the smallest exponent on the time step. The last term in Equation 4.3.15, \( O(\Delta t^2) \), indicates the order of magnitude of the truncation error, based on the time-step exponent of the largest error term, in “big-O” notation. The truncation error term is never evaluated, and there is never any need to verify its units. It is simply used to annotate the accuracy of the equation. Since the truncation error of Equation 4.3.15 is second order in time, the truncated series must be accurate to something less than second order; for example, this truncated series is first-order accurate. Formally, this is determined by solving the truncated series for \( \frac{dS}{dt} \) and algebraically showing that the truncation error of the derivative equation is \( O(\Delta t) \). (Note that \( O(\Delta t^2)/\Delta t = O(\Delta t) \).)

Equation 4.3.15 is our first example of a finite difference equation that can be used for numerical simulation. The process of updating properties using this particular truncated Taylor series is called “simple” or “explicit” Euler integration. Explicit Euler integration is called a one-point method, because we can solve it using properties stored at exactly one point in time, \( t \), which is prior to the update time, \( t + \Delta t \). It is called an explicit method since the property \( S(t + \Delta t) \) is the only unknown value, which we can explicitly update without solving a system of simultaneous equations. An important characteristic of explicit Euler integration is that every term of the right-hand side of the equation is evaluated at time \( t \), the time step immediately prior to the new time \( t + \Delta t \).

Let’s take a closer look at Equation 4.3.15. We take the view that the variable that we are numerically integrating, \( S \), is the state of an object, and that \( \frac{dS}{dt} \) is the state derivative.

\[
\underbrace{S(t + \Delta t)}_{\text{new state}} = \underbrace{S(t)}_{\text{prior state}} + \Delta t \underbrace{\frac{d}{dt} S(t)}_{\text{state derivative}}
\]  
(4.3.16)

This view enables us to conveniently write a numerical integrator that can integrate arbitrary properties as they change over time. We can, in fact, write an integrator that can integrate a collection of properties for a collection of objects in a single function call. Listing 4.3.3 is a fragment of pseudocode for an explicit Euler integrator that integrates a state vector of arbitrary length.
Listing 4.3.3  Pseudocode for an explicit Euler integrator that integrates a state vector of arbitrary length. Here, \(N\) is the number of states, \(\text{new}_S\) is \(S(t + \Delta t)\), \(\text{prior}_S\) is \(S(t)\), \(S\_derivs\) is \(dS/dt\), and \(\Delta t\) is \(\Delta t\).

```c
void ExplicitEuler(N, new_S, prior_S, S_derivs, delta_t)
{
    for (i = 0; i < N; i++)
    {
        new_S[i] = prior_S[i] + delta_t * S_derivs[i];
    }
}
```

For a single particle, one choice (but not the only choice) for the state vector is quite simply \(S = \langle mV, p \rangle\). The corresponding vector of state derivatives follows directly from Equations 4.3.1 and 4.3.8: \(dS/dt = \langle F, V \rangle\). Note that for three-dimensional motion, the state and state derivative vectors contain six real values each, since the linear momentum, position, force, and velocity are all three-component vectors. For a collection of \(N\) particles, we can expand these states to include properties for all of the particles.

\[
S(t) = \langle m_1V_1, p_1, m_2V_2, p_2, \ldots, m_NV_N, p_N \rangle \tag{4.3.17}
\]

\[
\frac{d}{dt}S(t) = \langle F_1, V_1, F_2, V_2, \ldots, F_N, V_N \rangle \tag{4.3.18}
\]

### Using Numerical Integration to Simulate a Collection of Particles

Listing 4.3.4 is pseudocode for implementing explicit Euler integration for a collection of particles. This code is a straightforward implementation of the state vector shown in Equation 4.3.17 and the state derivative vector shown in Equation 4.3.18, and makes use of the integrator shown in Listing 4.3.3. Note that the `calcForce` function called by the pseudocode is a placeholder for a function that can determine the force applied to a given particle. For example, if the only force applied is that due to gravity, `calcForce` will simply return the weight of the given particle, since the particle's weight is the force due to gravity.

Listing 4.3.4  Pseudocode for explicit Euler integration for a collection of \(N\) particles that move without colliding.

```c
void main()
{
    // Initialize variables needed by the simulation.
    Vector3D cur_S[2*N];  // S(t+\Delta t)
```
Vector3D prior_S[2*N];  // S(t)
Vector3D S_derivs[2*N];  // dS/dt at time t
float mass[N];           // Mass of particles
float t;                 // Current simulation time
float delta_t;           // Physics time step

// Set current state to initial conditions.
for (i = 0; i < N; i++)
{
    mass[i] = mass of particle i;
    cur_S[2*i] = particle i initial linear momentum;
    cur_S[2*i+1] = particle i initial position;
}

// Game simulation/rendering loop
while (game simulation is running)
{
    DoPhysicsSimulationStep(delta_t);
    for (i = 0; i < N; i++)
        Render particle i at position cur_S[2*i+1];
}

// Update the physics
void DoPhysicsSimulationStep(delta_t)
{
    copy cur_S to prior_S

    // Calculate the state derivative vector.
    for (i = 0; i < N; i++)
    {
        S_derivs[2*i] = CalcForce(i);
        S_derivs[2*i+1] = prior_S[2*i] / mass[i];
    }

    // Integrate the equations of motion.
    ExplicitEuler(2*N, cur_S, prior_S, S_derivs, delta_t);

    // By integrating the equations of motion, we have
    // effectively moved simulation time forward by
    // delta_t.
    t = t + delta_t;
}

Collision Response in the Simulation Loop

You can adapt the code in Listing 4.3.4 to a real game without modification as long as there are no collisions; however, in general the code must be modified somewhat to handle collisions.

In theory, if collisions all happen to occur at the beginning of a time step (e.g., at time \( t \) before the equations of motion are integrated), the modification is quite simple. The collisions should be resolved before copying \( \text{cur}_S \) to \( \text{prior}_S \) at the top of
DoPhysicsSimulationStep. For each colliding pair, simply use Equation 4.3.13 to compute the impulse, and then compute the after-collision linear momentums using Equations 4.3.9 and 4.3.10. The after-collision linear momentums should replace the corresponding values in cur_S. When the algorithm continues, these after-collision momentums are copied into prior_S, and after_S derive is set up, the call to ExplicitEuler will use the after-collision velocities to update the positions of the particles.

In practice, collisions will rarely occur at the beginning of a time step. It is far more likely that collisions will occur at some time, \( t_c \), between \( t \) and \( t + \Delta t \). Further, the collision time \( t_c \) is likely to be different for every pair of colliding objects. Observe what must happen in the simulation in this case, if we are to be strictly correct in our handling of collision response using the impulse-momentum approach. For a pair of colliding objects, we know that from time \( t \) until time \( t_c \), the two objects obey the equations of motion. Then, at time \( t_c \), the objects exhibit an instantaneous change in velocity due to the collision event. Finally, from time \( t_c \) until time \( t + \Delta t \), the objects again obey the equations of motion. We must, then, integrate the equations of motion twice in a given physics update for objects in collision: once to obtain the object states at \( t_c \), and again to obtain the state at the end of the time step following the collision, \( t + \Delta t \).

Listing 4.3.5 illustrates a modification to the DoPhysicsSimulationStep function to allow for collisions and the two-step integration. Note that this modified function has been written for clarity, not efficiency. For example, although not shown here, the code will run must faster if you batch objects not involved in collisions and integrate their states in a single call to ExplicitEuler, as illustrated in Listing 4.3.4.

**Listing 4.3.5** Modification to the DoPhysicsSimulationStep function to support collisions

```c
// Update the physics
void DoPhysicsSimulationStep(delta_t)
{
    // Detect collisions using methods described in
    // the Collision Detection and Resolution Chapter 4.2.

    ...

    // Integrate the equations of motion to update
    // the physics.
    for (each object i NOT involved in a collision)
    {
        copy cur_S[2*i] and cur_S[2*i+1] to
        prior_S[2*i] and prior_S[2*i+1]
    }
```
// Calculate the state derivative vector.
S_derivs[2*i] = CalcForce(i);
S_derivs[2*i+1] = prior_S[2*i] / mass[i];

// Integrate equations of motion, for object i only.
ExplicitEuler(2, &current_S[2*i],
             &prior_S[2*i], &S_derivs[2*i], delta_t);
}

for (each pair of objects i, j, involved in a collision)
{
    // tc is the time of the collision, which
    // occurs between t and t + delta_t.

    // Copy current state to prior state.
copy cur_S[2*i] and cur_S[2*i+1] to
prior_S[2*i] and prior_S[2*i+1]

copy cur_S[2*j] and cur_S[2*j+1] to
prior_S[2*j] and prior_S[2*j+1]

    // Calculate the state derivative vector.
S_derivs[2*i] = CalcForce(i);
S_derivs[2*i+1] = prior_S[2*i] / mass[i];

S_derivs[2*j] = CalcForce(j);

    // First integration: prior to collision.
ExplicitEuler(2, &cur_S[2*i], &prior_S[2*i],
             &S_derivs[2*i], tc - t);
ExplicitEuler(2, &cur_S[2*j], &prior_S[2*j],
             &S_derivs[2*j], tc - t);

    // current_S now has the position and velocity of
    // objects i and j at tc, the time of collision.

    // Use Equations 13, 9, and 10 to compute the
    // after-collision linear momentums. The updated
    // momentums should be placed in cur_S[2*i] and
    // cur_S[2*j] for objects i and j, respectively.

    // Once cur_S reflects the after-collision
    // momentums, perform the after-collision
    // integration step, using the pseudo-code below.

copy cur_S[2*i] and cur_S[2*i+1] to
prior_S[2*i] and prior_S[2*i+1]

copy cur_S[2*j] and cur_S[2*j+1] to
prior_S[2*j] and prior_S[2*j+1]
// Calculate the state derivative vector.
S_derivs[2*i] = CalcForce(i);
S_derivs[2*i+1] = prior_S[2*i] / mass[i];

S_derivs[2*j] = CalcForce(j);

// Second integration: after collision.
ExplicitEuler(2, &cur_S[2*i], &prior_S[2*i],
               &S_derivs[2*i], t + delta_t - tc);

ExplicitEuler(2, &cur_S[2*j], &prior_S[2*j],
               &S_derivs[2*j], t + delta_t - tc);

// Now all objects, including those in collision, have
// been updated to time t + delta_t.

// By integrating the equations of motion, we have
// effectively moved simulation time forward by
// delta_t.
t = t + delta_t;

To be strictly correct, the algorithm used to discover the time of collision \( t_c \) should use the physics integrator to advance the motion from \( t \) to \( t_c \) until a converged solution for \( t_c \) is found. For example, the bisection method discussed in Chapter 4.2 should use the physics integrator to compute the position of objects for each estimate of \( t_c \) until the bisection iterations are stopped. However, when object accelerations are zero or moderate and the physics time step is small, depending on the game, it may be perfectly acceptable to assume velocity is constant from \( t \) until \( t_c \) during collision discovery. If you make this assumption, the position of the objects at time \( t_c \) is trivially determined using Equation 4.3.5, with \( \mathbf{V}_{init} \) set to the velocity at the start of the time step, and \( \mathbf{p}_{init} \) set to the position at the start of the time step, and dropping the gravitation term. Note that swept-volume intersection tests are usually based on this assumption of constant velocity from time \( t \) to \( t_c \) before a collision. The swept-sphere intersection test discussed in Chapter 4.2, for example, makes this assumption, and only produces the exact result for \( t_c \) when the velocity of both spheres is constant prior to the collision.

Complete Exercise 6 to create a simple marbles game that implements frictionless particle collision response.

Complete Exercise 7 to create a projectile particle system based on explicit Euler integration.

**Numerical Stability Issues and Alternatives to Explicit Euler Integration**

Truncation error is always present in numerical integration. Since the result of one numerical integration step feeds the next numerical integration step, the truncation
error at each step accumulates into a total error in the state vector that may grow or shrink over time. A critical goal in numerical simulation is to ensure that the total error is bounded; that is, the total error does not grow large without limits. A numerical simulation in which the total error is bounded for all time is said to be numerically stable. Unfortunately, in some circumstances the truncation error can interact with the properties that drive the motion in such a way that the simulation is numerically unstable. In this case, the total error is unbounded and will eventually grow as large as possible, ultimately resulting in floating-point overflow. Numerical integration techniques are said to be conditionally stable if they can be made stable by reducing the time step, $\Delta t$, below some threshold, a stability bound. The references [Rhodes01], [Eberly04], and [Anderson95] provide a more detailed introduction to these concepts.

It happens that explicit Euler integration, while simple, even intuitive, is one of the worst possible choices for numerical rigid-body physics simulation. It is conditionally stable at best, and unconditionally unstable when used to simulate physical systems that include spring forces (described in the next section), unless damping forces are added.

One alternative to explicit Euler integration, which is often a better choice for rigid-body physics simulation is Verlet integration. There are several variations, and we present the velocityless version here, as Equation 4.3.19, without proof. It is called “velocityless” since the first time derivative of the state, the velocity of state, does not appear. The references [Porcino04], [Eberly04], and [Jakobsen03] provide a more in-depth look at the derivation of the Verlet integrator. Note that in this case, you must track the state vector for two prior time steps and that the state derivative is actually the second time derivative. Listing 4.3.6 provides pseudocode for a velocityless Verlet integrator.

$$S(t + \Delta t) = 2 \frac{S(t)}{\text{new state}} - \frac{S(t - \Delta t)}{\text{prior state 1}} + \left(\frac{\Delta t}{\text{prior state 2}}\right)^2 \frac{d^2 S(t)}{dt^2}$$  

(4.3.19)

**Listing 4.3.6** Pseudocode for a velocityless Verlet integrator

```c
void VelocityLessVerlet(N, new_S, prior_S1, prior_S2, 
S_2nd_derivs, delta_t)
{
    for (i = 0; i < N; i++)
    {
        new_S[i] = (2.0 * prior_S1[i]) - prior_S2[i] + 
                    (delta_t * delta_t * S_2nd_derivs[i]);
    }
}
```

In the case of velocityless Verlet integration, the natural choice for the state vector of a particle is simply $S = \langle p_1, p_2, \ldots, p_N \rangle$, with the corresponding second state derivative being $d^2 S / dt^2 = \langle F_1 / m_1, F_2 / m_2, \ldots, F_N / m_N \rangle$. 
In some cases, we shall see, the force applied to an object depends on velocity. In this case, you will need to integrate velocity in addition to position. It is often acceptable to use a mixture of different techniques; for example, explicit Euler integration to update the velocity states needed to compute forces that are dependent on velocity, and velocityless Verlet integration to update the position states.

There are a great many alternatives to explicit Euler and Verlet, which can be derived by manipulating truncated Taylor series expansions at different offsets from time \( t \). Each method exhibits its own precision and stability characteristics. In terms of game physics, the Runge-Kutta series of integrators is quite popular, with the fourth-order Runge-Kutta method being a robust general-purpose, explicit integrator.

The most stable numerical integration methods are called implicit methods. These require solving a system of linear equations for each time step. If you must implement a general-purpose physics engine, it will be well worth your time to explore implicit methods. The so-called \( A \)-Stable implicit methods are stable for any time step size, \( \Delta t \), meaning you will never have to reduce your physics time-step size to achieve a stable simulation, although you may need to adjust \( \Delta t \) to achieve good accuracy. (There is an exception to this. \( A \)-Stable and other implicit methods usually are at best conditionally stable when simulating situations that are physically unstable. See [Rhodes01] or [Eberly04] for more details on physical vs. numerical instability.)

There are many, many resources on numerical integration. Details on implementing the Runge-Kutta and implicit methods can be readily found in a library.

Complete Exercise 8 to demonstrate the difference between velocityless Verlet integration and explicit Euler integration.

**The Importance of Frame-Rate Independence**

Given the tendency of numerical simulation to be sensitive to the time step, it is rather important that you strive to create a physics engine that is frame-rate independent. By implementing a frame-rate independent system, you gain two significant benefits. First, your results will be repeatable, every time you run a simulation with the same inputs, regardless of computer CPU or GPU performance. Second, you will have maximum control over the stability of your simulation. Listing 4.3.7 is a fragment of pseudocode that illustrates a simple way to ensure that your physics simulation is updated using a constant time step that is independent of your actual game frame rate.

**Listing 4.3.7** Pseudocode for updating the physics simulation at fixed time steps.

```c
void main()
{
    float delta_t = 0.02;    // Physics time step, seconds
    float game_time;        // Current game time, seconds
```
float prev_game_time;  // Game time at previous frame
float physics_lag_time = 0.0;  // Time since last update

// Simulation/rendering loop
while (main game loop)
{
    update game_time;
    physics_lag_time += (game_time - prev_game_time);
    while (physics_lag_time > delta_t)
    {
        DoPhysicsSimulationStep(delta_t);
        physics_lag_time = physics_lag_time - delta_t;
    }
    prev_game_time = game_time;

    render scene;
}

### Generalized Translational Motion

With the basics of numerical physics simulation in place, we will now consider a variety of nonconstant forces that contribute to the generalized motion of an object. Any combination of these forces might be acting on an object at a given time. You can obtain the net force acting on an object, $F_{\text{net}}$, by simply adding all applied forces together. $F_{\text{net}}$ is exactly the value of $F$ to be used in the state derivative vector for numerical integration.

If $F_{\text{net}}$ has zero magnitude, the object has zero acceleration and is said to be in translational equilibrium, although it may still be moving at a nonzero velocity.

### General Rigid Bodies versus Spherical Particles

We will now generalize our discussion to include arbitrary sized and shaped rigid bodies, rather than perfectly smooth spherical particles. There is exactly one reason for the smooth sphere restriction so far: we were able to completely avoid considering the issue of rotational motion. This is important: all of the equations presented previously are perfectly valid for rigid bodies of any shape that are of finite size.

Despite the fact that the previous equations support arbitrarily shaped objects, they describe the translational physics of a single point on the object (e.g., the center of a sphere). Rigid body objects fill a volume in space, and so we must find an appropriate point on the object for use in a simulation. It is standard practice to choose the object's center of mass to be the reference point for translational motion. The reason for this choice is that it removes inertial coupling that would otherwise make the translational equations dependent on rotation and the entire system more difficult to solve.

Equation 4.3.20 defines the location of the center of mass for a rigid body.

$$
\mathbf{p}_{\text{center-of-mass}} = \frac{1}{mass} \iiint_{\text{vol}} \rho \mathbf{r} \, d\text{x} \, d\text{y} \, d\text{z}
$$  \hspace{1cm} (4.3.20)
The density, $\rho$ is the mass per unit volume, with units of type mass per the cube of distance. The SI units for density are kilograms per meter cubed (kg/m$^3$). The variable, $\mathbf{r}$, is the vector from a known reference point to the location of a differential element of the object’s mass. The resulting center-of-mass location is calculated relative to the same reference point. Note that density might be a constant, but in general it might also vary with the differential element position, $\mathbf{r}$. For example, if you are computing the center of mass of an object made partly of steel and partly of plastic, the integration over the plastic parts would use a different density from the integration over the steel parts. For arbitrarily shaped objects, Equation 4.3.20 can be difficult to evaluate. Brian Mirtich [Mirtich96] and David Eberly [Eberly03] have documented robust techniques for evaluating the center of mass of triangle mesh objects, which are extremely useful for game development.

Figure 4.3.6 illustrates a rigid body and its center of mass, and shows the standard symbol for center of mass. For physics simulation, we normally choose a local object-aligned coordinate system with its origin located at the center of mass. This same local coordinate system can be used for collision detection and rendering, as well as physics simulation.

**Figure 4.3.6 The center of mass of an object.**

### Linear Springs

The first generalized force that we will consider is due to a spring connecting two objects. Figure 4.3.7 illustrates a spring that is connecting two objects.

The spring is connected to one of the objects at location $\mathbf{p}_{e_1}$, and to the other object at location $\mathbf{p}_{e_2}$, the endpoints of the spring. The length of the spring is simply the Euclidian distance between the two endpoints. A spring has a so-called *rest length*, $l_{rest}$, which defines the length of the spring when it is neither compressed nor stretched. The spring exerts zero force when its length is its rest length. When the spring is stretched to be longer than $l_{rest}$, it applies an attraction force to each of the objects. When the spring is compressed to be shorter than $l_{rest}$, it applies a repulsion force to each of the objects. Equation 4.3.21 presents the simplest realistic model of
spring force, *Hooke's Law*, in which the force is a linear function of the displacement from $l_{\text{rest}}$:

$$F_{\text{spring}} = k(l - l_{\text{rest}}) \dot{d}$$  \hspace{1cm} (4.3.21)

The variable, $k$, is the *spring stiffness*, a measure of the strength of the spring. *The stiffness is measured in units of type force per unit length. The SI units for spring stiffness are Newtons per meter.* The variable, $l$, is the current length of the spring, and the vector variable, $\dot{d}$, is a unit length vector in the direction from $p_{e1}$ to $p_{e2}$. The spring force, $F_{\text{spring}}$, is applied to object 1 at location $p_{e1}$. An equal but opposite force, $-F_{\text{spring}}$, is applied to object 2 at location $p_{e2}$.

**Viscous Damping**

Viscous damping is a dissipative force (one that reduces kinetic energy) acting on objects moving at low speeds through fluids such as air, water, and oil. Mechanical damping devices called *dashpots* generate viscous damping forces, and are often used to reduce vibrations in machines, vehicle suspension systems, and so forth. Dashpots apply a damping force to the objects to which they are connected, as shown in Figure 4.3.8.

**FIGURE 4.3.8** *The dashpot damper. If objects approach each other along the line between the points where the damper is connected, the damping force is repulsive.*
The viscous damper applies forces along the damper axis. The magnitude of the forces is related to the relative velocity of the objects along the damper axis. Equation 4.3.22 defines the force. The parameter, \( c \), is the damping coefficient, measured in units of type mass per unit time. The SI units for damping coefficient are kilograms per second. The parameter, \( \hat{d} \), is a unit vector in the direction from \( p_{el} \) to \( p_{e2} \).

\[
F_{damping} = c \left( (V_{ep2} - V_{ep1}) \cdot \hat{d} \right) \hat{d} \tag{4.3.22}
\]

The damping force, \( F_{damping} \), is applied to object 1 at location \( p_{el} \). An equal but opposite force, \(-F_{damping}\), is applied to object 2 at location \( p_{e2} \).

**Aerodynamic Drag**

An object traveling through a fluid, such as air or water, experiences a drag force that acts in the opposite of the object’s velocity through the fluid. Equation 4.3.23 provides a simple approximation for this aerodynamic drag. Here, \( C_D \) is the drag coefficient, which has no units. Typical values for nonstreamlined objects range from 0.1 to 0.4. The variable \( S_{ref} \) is a representative front-projected area of the object. For game objects, other than aircraft, choose \( S_{ref} \) to be the cross-section area of a bounding sphere for the object, and assume the drag force acts at the center of mass of the object. In Equation 4.3.23, the variable \( \rho \) is the mass density of the fluid through which the object is traveling. [Rhodes05] provides a comprehensive overview of aerodynamic forces.

\[
F_{drag} = -\frac{1}{2} \rho |V|^2 C_D S_{ref} \frac{V}{|V|} \tag{4.3.23}
\]

**Surface Friction**

When two objects make contact, either during a collision, while in resting contact, or during sliding contact, the objects potentially exert a force on each other within the contact plane. This tangential force is called friction. The behavior of the friction force is rather complex. Observe that if you apply a horizontal force to an object at rest on a surface, the object does not begin moving unless the force exceeds a threshold. Once the force exceeds the threshold, the object begins moving, often abruptly. When the object is moving, the force required to keep the object moving is less than the force required to cause the initial motion. This observation illustrates the presence of a variable static friction when the object is at rest, and a dynamic friction when the object is in motion.

Coulomb developed the most common model of friction in the year 1781. You may be familiar with Coulomb friction from your prior studies. Using the Coulomb model, the magnitude of static friction is equal to the component of an external force, \( F_{applied} \), applied between the objects in the contact plane up to a maximum magnitude of...
of $\mu_s |\mathbf{F}_n|$, where $\mu_s$ is a static friction coefficient, and $\mathbf{F}_n$ is the component of the applied force parallel to $\mathbf{\hat{n}}$, given by $\mathbf{F}_n = \mathbf{\hat{n}} \cdot (\mathbf{F}_{\text{applied}} \cdot \mathbf{\hat{n}})$. The magnitude of dynamic friction, generated when there is relative motion in the contact plane between the two objects, is given by $\mu_d |\mathbf{F}_n|$. Here, $\mu_d$ is the dynamic friction coefficient. The friction coefficients are functions of the material properties of the two objects that are in contact. For example, the value of $\mu_s$ between two objects made of wood ranges from around 0.2 to around 0.75. The value of $\mu_d$ is usually smaller than $\mu_s$. The difference between the coefficients leads to a discontinuity in the magnitude of friction force at the moment the objects begin to slide past one another, and this discontinuity can cause a difficulty in numerical simulations. The friction coefficients have no units.

There are three basic scenarios for two objects in contact with each other, shown in Figure 4.3.9. The following are the conventions used in the figure and equations to follow: $\mathbf{F}_{\text{applied}}$ is the total force, less friction, applied by object 1 onto object 2; $\mathbf{V}_t$ is the tangential component of the relative velocity of object 1 moving past object 2; $\mathbf{\hat{n}}$ is the contact normal measured outward from object 2; the resulting friction force, $\mathbf{F}_{\text{friction}}$ as calculated is applied on object 1, so that the net force on object 1 becomes $\mathbf{F}_{\text{net}} = \mathbf{F}_{\text{applied}} + \mathbf{F}_{\text{friction}}$. By Newton's Third Law of Motion, $-\mathbf{F}_{\text{friction}}$ is applied on object 2, so there is no need to calculate the friction force twice. The tangential relative velocity, $\mathbf{V}_t$, is given by $\mathbf{V}_t = (\mathbf{V}_1 - \mathbf{V}_2) - \mathbf{\hat{n}}((\mathbf{V}_1 - \mathbf{V}_2) \cdot \mathbf{\hat{n}})$.

![Figure 4.3.9 Friction acting on object 1 at a contact point for three scenarios. The force, $\mathbf{F}_{\text{applied}}$, is applied by object 1 onto object 2.](image)

An intuitive example of an external force, $\mathbf{F}_{\text{applied}}$, applied by one object onto another is simply the weight of an object resting on a horizontal surface. The resting object applies a force equal to its weight on the surface in the direction $-\mathbf{\hat{n}}$, and from Newton's Third Law of Motion, the surface applies an equal but opposite force back on the object. If you exert an additional horizontal force on the object, attempting to slide the object, $\mathbf{F}_{\text{applied}}$ would be the sum of the object's weight plus the additional horizontal force.
If \( \mathbf{V}_r \) is zero, the friction force is given by Equation 4.3.24. Note that this equation guarantees that the magnitude of static friction never exceeds the Coulomb maximum of \( \mu_s |\mathbf{F}_n| \). The tangential component of the applied force, \( \mathbf{F}_t \), is given by

\[
\mathbf{F}_t = \mathbf{F}_\text{applied} - \mathbf{F}_n.
\]

\[
\mathbf{F}_{\text{friction}} = \frac{-\mathbf{F}_t}{|\mathbf{F}_t|} \min(\mu_s |\mathbf{F}_n|, |\mathbf{F}_t|)
\]

(4.3.24)

If \( \mathbf{V}_r \) is zero, and \( \mathbf{F}_t \) exceeds \( \mathbf{F}_{\text{friction}} \) in magnitude, the objects will begin to accelerate tangentially past one another. For the case when \( \mathbf{V}_r \) is nonzero, the friction force is given by Equation 4.3.25. Note that here, \( \mathbf{F}_\text{applied} \cdot \hat{n} \) is always negative, and so friction acts in a direction opposite \( \mathbf{V}_r \). Friction is a dissipative force when the objects are in relative motion, and acts to reduce the kinetic energy of the two objects.

\[
\mathbf{F}_{\text{friction}} = \frac{-\mathbf{V}_r}{|\mathbf{V}_r|} \mu_d |\mathbf{F}_n|
\]

(4.3.25)

It is interesting to note that it is possible for dynamic friction to act in the same direction as the tangential applied force, rather than always against it. For example, consider a passenger train headed on a deadly course toward a bridge recently destroyed by a villain. Our hero has managed to set the brakes so that the wheels are no longer turning. Sliding, Coulomb friction acts to slow the train down. However, friction is insufficient to stop the train in time. Our hero might tie one end of a chain to the train, and then hold the other end of the chain while bracing herself against a building, or a perhaps a mountain. The force that the chain applies to the train acts opposite the train's velocity, in the same direction as the friction force. Thus, the applied force and the friction force act in the same direction, both contributing to the deceleration of the train and saving of lives.

**A Simple Spring-Mass-Damper Soft-Body Dynamics System**

To understand better how you might go about using these various forces, consider a fun example. Using the results of this section and the prior section, you can construct a simple soft-body dynamics simulator. Simply create a polygon mesh with an interesting shape. You can create the mesh in code or use a digital content creation modeling package. For this system, you will use physics to update the position of the vertices of the mesh. For the physics system, create a particle at the location of each vertex of the mesh, and assign a mass to the particle. Then, create a spring and a damper between unique pairs of particles. The spring rest lengths should be equal to the initial distance between particles. Figure 4.3.10 illustrates this configuration, for a 2D model; the arrangement extends naturally to 3D. It is important that you include
springs that connect particles on opposite sides of the mesh, to prevent the shape from collapsing; however, for complex meshes you don’t necessarily have to have springs between every unique pair.

![Diagram showing geometric mesh for rendering and spring-mass-damper physics model with notes: Each dashed line represents 1 spring + 1 damper.]

**FIGURE 4.3.10** A simple soft-body model of a mesh, represented by a collection of particles connected by springs and dampers.

As a way of experimenting with this type of simple soft-body model, consider the pseudocode in Listing 4.3.8, which initializes the object in midair, with an initial velocity of zero. The forces acting on the particles include gravity, as well as the spring and damper forces.

**Listing 4.3.8** A simple spring-mass-damper soft-body dynamics system.

```
void main()
{
    N = number of particles;  // # verts in visual model
    Vector3D cur_S[2*N];     // S(t+delta_t)
    Vector3D prior_S[2*N];   // S(t)
    Vector3D S_derivs[2*N];  // dS/dt
    Vector3D g(0.0,0.0,-9.81); // Gravity
    float mass[N];           // Mass of particles
    float k[N][N];           // Spring constant between particles
    float lrest[N][N];       // Spring rest lengths
    float c[N][N];           // Damper constant between particles
    float delta_t = 0.02;    // Physics time step, seconds
    float game_time;         // Current game time, seconds
    float prev_game_time;    // Game time at previous frame
    float physics_lag_time=0.0; // Time since last update
    float init_height;       // Initial height above the ground
```
// Initialize the particles.
for (i = 0; i < N; i++)
{
    mass[i] = 1.0f; // mass = 1 kg

    // Set initial linear momentum to be zero.
    cur_S[2*i] = Vector3D(0,0,0);

    // Assign initial position from visual model.
    cur_S[2*i+1] = position of model vertex i;

    // Update the initial position to reflect init_height.
    cur_S[2*i+1].z += init_height;
}

// Initialize a spring and damper between every pair
// of particles.
for (i = 0; i < N; i++)
{
    for (j = 0; j < N; j++)
    {
        // Configure the spring.
        k[i][j] = 10.0f; // stiffness = 10 N/m

        // Configure the damper.
        c[i][j] = 0.1f; // damping coef = 0.1 kg/s

        // Configure the rest length.
        lrest[i][j] = length of vector
                       (cur_S[2*j+1] - cur_S[2*i+1]);
    }
}

// Runtime loop, derived from Listing 6.
while (main game loop)
{
    update game_time;
    physics_lag_time += (game_time - prev_game_time);
    while (physics_lag_time > delta_t)
    {
        DoPhysicsSimulationStep(delta_t);
        physics_lag_time = physics_lag_time - delta_t;
    }
    prev_game_time = game_time;

    // Once physics has updated the particle
    // positions, we must transfer these to the
    // visual model for rendering.
    for (i = 0; i < N; i++)
    {

update visual model vertex i position to be cur_S[2*i + 1];
}
}

render the visual model;
}

void DoPhysicsSimulationStep(delta_t)
{
    Use Listing 5.

    For better stability, modify Listing 5 to use velocity-less Verlet integration for the position updates and explicit Euler to update particle velocities.

    Vector3D CalcForce(i)
    {
        Vector3D d, SForce, DForce, RelativeVel;
        Vector3D Force_Net = 0.0f;

        // Initialize the net force by calculating the force due to gravity, the particle's weight.
        Force_Net += mass[i] * g;

        // Compute the spring and damper forces.
        for (j = 0; j < N; j++)
        {
            // Compute unit vector from particle i to particle j, and the current length of the spring.
            d = cur_S[2*j+1] - cur_S[2*i+1];
            length = d.Length();

            d.Normalize();  // Make d unit length.

            // Compute the spring force using Equation 20. i is attracted to j if the current length is greater than the rest length, repelled from j if the current length is less than rest length.
            SForce = k[i][j] * (length - lrest[i][j]) * d;

            // Compute the damping force. First we need the relative velocity.
            RelativeVel = (cur_S[2*j]/mass[j]) - (cur_S[2*i]/mass[i]);

            // From here, we calc the damping force using Equation 21. If object j is moving away from object i, the force on object j draws i
// towards j, otherwise the force repels i away
// from j.
DForce = c[i][j] * RelativeVel.DotProduct(d) * d;

// Increment the net force.
Force_Net += SForce;
Force_Net += DForce;

return(Force_Net);
}

Complete Exercise 10 to implement a soft-body simulation based on Listing 4.3.7. Note that for this exercise, you should initially implement collision detection and response between the particles and a solid, immovable ground surface. Use the results of Exercise 5 to determine the results of an impulsive collision with an immovable object. Jeff Lander [Lander99] has created a simple demo program based on the approach outlined in Listing 4.3.7. Jeff’s code is available for download on the Internet at www.gdmag.com/srclmar99.zip. This demo can serve as a nice reference for Exercise 10.

Rotational Motion

The physics of rotational motion are analogous to the kinematics of particle or center-of-mass translational motion. Torque, the analog of force, causes an angular acceleration. Angular acceleration causes a change in angular velocity. Angular velocity causes a change in orientation, the rotational analog of position. We begin our analysis of rotational motion with Equation 4.3.26, the rotational analog to Equation 4.3.8.

\[
\frac{d}{dt} \mathbf{L}(t) = \tau(t)
\]  \hspace{1cm} (4.3.26)

Here, the vector \( \tau(t) \) is the torque, sometimes called the moment, or moment of force. Torque is measured in units of type force times distance. The SI units for torque are Newton-meters. Torque is calculated at a point about which an object is expected to rotate, and is related to a force applied to the object. Torque is nonzero when the force acts along a line that does not intersect the point where torque is being calculated. Equation 4.3.27 gives the mathematical definition of torque, where \( \mathbf{r} \) is the vector from the point about which torque is being calculated to the point where the force causing torque is being applied. From the equation, note that torque has a direction that is perpendicular to the force vector and \( \mathbf{r} \).

\[
\tau = \mathbf{r} \times \mathbf{F}
\]  \hspace{1cm} (4.3.27)
4.3 Real-Time Game Physics

![Diagram showing the relationship between force and torque. The arc arrow indicates that the torque due to F causes a counterclockwise rotation, (e.g., a positive rotation about a vector pointing out of the page, the direction of the torque vector).](image)

**FIGURE 4.3.11** The relationship between force and torque. The arc arrow indicates that the torque due to $F$ causes a counterclockwise rotation, (e.g., a positive rotation about a vector pointing out of the page, the direction of the torque vector).

Figure 4.3.11 illustrates the generation of torque about the center of mass of a rigid body, due to a force, $F$, applied at a point, $P$, on the body. Here, $r$ is the vector from the center of mass to the point $P$.

Before continuing, let’s consider the effect of torque on an object. We would like the concept to be somewhat intuitive. The effect of torque is fairly intuitive when it acts on an object that initially is not rotating, and so we will consider that case here. Consider the classic children’s outdoor play toy, the seesaw. A seesaw in play rotates back and forth about the center fulcrum. If you approach a seesaw that is not in use, it will be resting, not rotating. When you sit on the end of the seesaw, your weight acts straight down near one end, and the vector $r$ from the fulcrum to your body points in a generally horizontal direction. Your weight results in a torque about the center of the fulcrum that acts to one side, parallel to the ground and perpendicular to the seesaw and your weight vector. It happens that the torque applied is exactly parallel to the axis of rotation about the fulcrum. This, in a nutshell, is the effect of torque on an object that is not initially rotating; the torque causes the object to begin rotating about the torque axis. For objects that are rotating with fairly small angular velocities, the result is similar: large torques change the axis of rotation to be approximately the torque axis. Objects with large angular momentum exhibit behavior that is less intuitive, called gyroscopic precession, in response to an applied torque, and even respond in a nonintuitive fashion when no torque is being applied (torque-free precession). We will not discuss these high angular momentum behaviors here.

If an object is moving freely in space, torque is calculated at the center of mass, and rotation is about the center of mass. If an object is constrained in some way,
torque should be calculated about some other point that represents the possible axis of rotation. For example, if an object is constrained by a hinge, torque should be measured at a point along the hinge axis. In this chapter, we are only concerned with torque about the center of mass.

The vector, \( \mathbf{L}(t) \), is called **angular momentum**, the rotational analog to linear momentum. Equation 4.3.28 states the mathematical definition of angular momentum. **Angular momentum is measured in units of type mass times distance squared per unit time. The SI units for angular momentum are kilogram-meters-squared per second (kg-m²/s).**

\[
\mathbf{L} = \mathbf{J}\omega
\]  
(4.3.28)

The variable \( \mathbf{J} \) is a symmetric \( 3 \times 3 \) matrix called the **inertia tensor**, the analog of mass. The terms of the inertia tensor describe the distribution of mass throughout the volume of a rigid body. (\( \mathbf{J} \) is represented by the variable \( \mathbf{I} \) in many texts; however, we use \( \mathbf{J} \) here to avoid confusion with the identity matrix.) Equation 4.3.29 defines the inertia tensor, which is measured in world coordinates.

\[
\mathbf{J} = \begin{bmatrix}
J_{xx} & J_{xy} & J_{xz} \\
J_{xy} & J_{yy} & J_{yz} \\
J_{xz} & J_{yz} & J_{zz}
\end{bmatrix}
\]  
(4.3.29)

The diagonal components of the inertia tensor are called **moments of inertia**, and the off-diagonal elements are called **products of inertia**. The inertia tensor is measured in units of type mass times distance squared. **The SI units for the inertia tensor are kilogram-meters-squared (kg-m²).** Equations 4.3.30 and 4.3.31 define the moments of inertia and products of inertia, respectively. Here, the variables \( r_x, r_y, \) and \( r_z \) are the components of a vector \( \mathbf{r} \) from the object’s center of mass to a differential element of mass in the object.

\[
J_{xx} = \iiint_{\text{Vol}} \rho \left( r_y^2 + r_z^2 \right) dxdydz ;
J_{yy} = \iiint_{\text{Vol}} \rho \left( r_x^2 + r_z^2 \right) dxdydz ;
J_{zz} = \iiint_{\text{Vol}} \rho \left( r_x^2 + r_y^2 \right) dxdydz
\]  
(4.3.30)

\[
J_{xy} = \iiint_{\text{Vol}} \rho r_x r_y dxdydz ;
J_{xz} = \iiint_{\text{Vol}} \rho r_x r_z dxdydz ;
J_{yz} = \iiint_{\text{Vol}} \rho r_y r_z dxdydz
\]  
(4.3.31)

As with center of mass, for general-shaped objects these equations can be tedious to evaluate. Table 4.3.1 provides equations for the inertia tensor elements for a few simple shapes. The products of inertia for these objects are zero. [Mirtich96] and [Eberly03] provide methods for robustly evaluating the inertia tensor and center of mass for the arbitrary triangle meshes that are common to game development.
Table 4.3.1 Inertia Tensor Values and Center-of-Mass Locations for Primitive Shapes with Constant Density

<table>
<thead>
<tr>
<th>Cylinder</th>
<th>Rectanguloid</th>
<th>Cone</th>
<th>Sphere</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Cylinder Diagram" /></td>
<td><img src="image" alt="Rectanguloid Diagram" /></td>
<td><img src="image" alt="Cone Diagram" /></td>
<td><img src="image" alt="Sphere Diagram" /></td>
</tr>
<tr>
<td>$r = \text{radius}$</td>
<td>$d = \text{depth}$</td>
<td>$J_{xx} = \frac{3}{20} m \left( r^2 + h^2 \right)$</td>
<td>$J_{xx} = J_{yy} = J_{zz} = \frac{2}{5} mr^2$</td>
</tr>
<tr>
<td>$J_{xx} = \frac{1}{12} m \left( 3r^2 + h^2 \right)$</td>
<td>$J_{xx} = \frac{1}{12} m \left( d^2 + h^2 \right)$</td>
<td>$J_{yy} = J_{xx}$</td>
<td>$J_{xx} = J_{yy} = J_{zz} = \frac{2}{5} mr^2$</td>
</tr>
<tr>
<td>$J_{yy} = J_{xx}$</td>
<td>$J_{yy} = \frac{1}{12} m \left( w^2 + h^2 \right)$</td>
<td>$J_{zz} = \frac{3}{10} mr^2$</td>
<td>$J_{xx} = J_{yy} = J_{zz} = \frac{2}{5} mr^2$</td>
</tr>
<tr>
<td>$J_{zz} = 0.5mr^2$</td>
<td>$J_{zz} = \frac{1}{12} m \left( d^2 + w^2 \right)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For physics simulation, as presented herein, the inertia tensor must be represented in the inertial reference frame, or the game's world coordinate system. If the object is rotating, the inertia tensor in this coordinate system will change as the object rotates. To avoid an expensive recomputation for every numerical integration step, it is best to compute the inertia tensor, and its inverse, $J^{-1}$, in the object's local coordinate system, and then transform that tensor into world space at every integration step using Equation 4.3.32. Note that the same transformation is applied to both the tensor and its inverse, where $R$ is the $3 \times 3$ object-to-world rotation matrix.

$$J = RJ_{\text{object\_space}} R^T ; \quad J^{-1} = RJ_{\text{object\_space}}^{-1} R^T$$ (4.3.32)

The orientation of an object can be represented by either $R$ or by a unit quaternion, $q$. Each of these is an analog to position. The angular velocity of an object, $\omega = \left\{ \omega_x, \omega_y, \omega_z \right\}$, measured in world space, is the analog of velocity. The direction of angular velocity is the direction of the object's rotation, and its magnitude is the rate of rotation. Angular velocity is measured in units of angle per unit time (angle/time). The SI units for angular velocity are radians per second (rad/s).

We can use Equations 4.3.26 through 4.3.32 to perform the rotational portion of our numerical physics simulation. For a single rigid body, the state vector for explicit Euler integration, with both translational and rotational states included, can be $S_i = \left\{ m_i \mathbf{V}_i, \mathbf{p}_i, \mathbf{L}_i, \mathbf{R}_i \right\}$, or $S_i = \left\{ m_i \mathbf{V}_i, \mathbf{p}_i, \mathbf{L}_i, q_i \right\}$ if you choose a unit quaternion to represent the object's orientation state. The state derivative entries for $m_i \mathbf{V}_i$ and $\mathbf{p}_i$ are the same as for the no-rotation case. The state derivative entry for $\mathbf{L}$ is given by Equa-
tion 4.3.26; it is the net torque, \( \tau_{\text{net}} \), calculated as the sum of torques due to the applied forces in Equation 4.3.33.

\[
\tau_{\text{net}} = \left( \sum_{i=1}^{N_{\text{springs}}} \mathbf{r}_{\text{spring},i} \times \mathbf{F}_{\text{spring},i} \right) + \left( \sum_{i=1}^{N_{\text{dampers}}} \mathbf{r}_{\text{damp},i} \times \mathbf{F}_{\text{damp},i} \right) + \mathbf{r}_{\text{contact}} \times \mathbf{F}_{\text{friction}} + \cdots
\] (4.3.33)

Equation 4.3.34 defines the value of the state derivative of \( \mathbf{R} \) when the orientation state is represented as an object-to-world rotation matrix, and Equation 4.3.35 defines the state derivative of \( q \) when the orientation state is represented by a unit quaternion. Take care to note the special representation of the angular velocity as a quaternion with its real component equal to 0.0.

\[
\frac{d}{dt} \mathbf{R}(t) = \begin{bmatrix}
0 & -\omega_z & \omega_y \\
\omega_z & 0 & -\omega_x \\
-\omega_y & \omega_x & 0
\end{bmatrix} \mathbf{R}(t) \tag{4.3.34}
\]

\[
\frac{d}{dt} q(t) = \frac{1}{2} \omega(t) q(t), \quad \text{with} \quad \omega = 0 + i\omega_x + j\omega_y + k\omega_z \tag{4.3.35}
\]

With the state vector and state derivative vector in place, you can use your favorite numerical integrator to solve for the updated state vector.

There are two postprocessing steps that you must perform to properly simulate rotation. First, since the state derivative of orientation, given by Equation 4.3.34 or 4.3.35, requires that we know the angular velocity, it is necessary to compute the angular velocity after the integration step, by solving Equation 4.3.28 for the angular velocity and transforming the object space inertia tensor into world space using the second part of Equation 4.3.32. This step is given here as Equation 4.3.36.

\[
\omega(t + \Delta t) = \mathbf{R}(t + \Delta t) \mathbf{J}^{-1}_{\text{object space}} \mathbf{R}^T(t + \Delta t) \mathbf{L}(t + \Delta t) \tag{4.3.36}
\]

Once the updated angular velocity is known, you will be able to prepare for the next integration step.

Second, a correction step must be applied to \( \mathbf{R} \) or \( q \) every few frames. When simulating rotation, \( \mathbf{R} \) is expected to be orthogonal and \( q \) is expected to be unit length. Floating-point round off and truncation error will cause these to drift over time. Every few frames, you must ensure that \( \mathbf{R} \) is orthogonal by performing a Gram-Schmidt orthonormalization ([Eberly04], [Golub96]) or perform a Euclidian normalization of \( q \).

**The Simulation Loop with Support for Rigid Body Rotation**

As discussed previously, the simulation loop for rigid body dynamics with rotational motion is somewhat more complex than pure translational motion. Listing 4.3.9 out-
lines the steps required to configure such a system, and illustrates the simulation loop without collision detection and response. As in prior code listings, comments within the code clarify the process. Note that in practice, collisions must be handled in a manner similar to Listing 4.3.5.

**Listing 4.3.9  Simulation loop with rotation**

```c
void main()
{
    N = number of rigid bodies
    Matrix33 JObj[N];         // Inertia tensors in object space
    Matrix33 JObjInv[N];      // Inverse inertia tensors in
                               // object space
    Matrix33 J;               // Temporary inertia tensor
                               // in world space
    float mass[N];            // Rigid body masses
    Vector3D cur_S[3*N];      // Velocity, position, and
                               // angular momentum states
    Quaternion cur_q[N];      // Orientation states as
                               // quaternions
    Vector3D prior_S[3*N];    // Prior vel, position,
                               // angular momentum states
    Quaternion prior_q[N];    // Prior orientation state
    Vector3D S_derivs[3*N];   // dS/dt for vel, pos,
                               // angular mom
    Quaternion q_derivs[N];   // dS/dt for orientation
    Vector3D cur_W[N];        // Current angular velocities
    Matrix33 R;               // Temporary rotation matrix
    int iCounter = 0;          // Counter to tell us when to
                               // re-normalize the
                               // orientation quaternion

    // Initialize the rigid bodies
    for (i = 0; i < N; i++)
    {
        mass[i] = mass of rigid body i;

        // The initial center-of-mass position and
        // linear momentum are the same as the
        // translational only case.
        cur_S[3*i] = initial linear momentum of i
        cur_S[3*i+1] = initial position of i's center-of-mass;

        // The rotational state variables are dependent
        // on the inertia tensor, so we calculate that here.
        JObj[i] = compute inertia tensor of rigid body i,
                   in the local object space of i;

        // Since we need it later, compute and store the
        // inverse inertia tensors.
    }
}
```

JObjInv[i] = JObj[i].Inverse();

// Set the initial orientation of object i. This
// is a quaternion represented in world space.

cur_q[i] = current orientation as a unit quaternion;

// The initial angular velocity is here assumed
// to be zero, but it might be nonzero.

cur_w[i] = Vector3D(0,0,0);

// Compute initial angular momentum given angular
// velocity using Equation 28 and 32. This
// requires that we first compute a 3x3 rotation
// matrix cur_q[i].

R = cur_q[i].ConvertToRotationMatrix();

// From here we can compute the initial world
// space inertia tensor using Equation 32.

J = R * JObj[i] * R.Transpose();

// Now we can compute the initial angular
// momentum using Equation 28.

cur_S[3*i+2] = J * cur_w[i];

} // Game simulation/rendering loop
while (game simulation is running)
{
    update game_time;

    // Update the physics.
    physics_lag_time += (game_time - prev_game_time);
    while (physics_lag_time > delta_t)
    {
        DoPhysicsSimulationStep(delta_t);
        physics_lag_time = physics_lag_time - delta_t;
    }
    prev_game_time = game_time;

    // Occasionally renormalize the orientation
    // quaternion.
    if (++iCounter == 5)
    {
        iCounter = 0;
        for (i = 0; i < N; i++)
            cur_q[i].Normalize();
    }
// Render the scene.
for (i = 0; i < N; i++)
{
    render rigid body i at position
    cur_S[3*i+1], and orientation cur_q[i];
}

// Update the physics
void DoPhysicsSimulationStep(delta_t)
{
    copy cur_S to prior_S
    copy cur_q to prior_q

    // temp_w is a temporary quaternion version of the
    // angular velocity, as shown in Equation 35.
    Quaternion temp_w;

    // Calculate the state derivative vectors.
    for (i = 0; i < N; i++)
    {
        // State derivative for translational linear
        // momentum and position are the same as for
        // the non-rotation case.
        S_derivs[3*i] = CalcForce(i);
        S_derivs[3*i+1] = prior_S[3*i] / mass[i];

        // State derivative for angular momentum is the
        // net torque, given by Equations 26 and 33.
        S_derivs[3*i+2] = CalcTorque(i);

        // State derivative for the orientation
        // is given by Equation 35.
        temp_w.Set(0, cur_w[i].x, cur_w[i].y, cur_w[i].z);

        q_derivs[i] = 0.5 * temp_w * cur_q[i];
    }

    // Integrate the equations of motion. Vector states first.
    ExplicitEuler(3*N, cur_S, prior_S, S_derivs, delta_t);

    // Followed by the quaternion orientation state.
    ExplicitEuler(N, cur_q, prior_q, q_derivs, delta_t);

    // We are not done yet. We have the updated state, but
    // we need to compute the new angular velocity using
    // Equation 36. We do this in a loop since it has to
    // be done for all objects.
    for (i = 0; i < N; i++)
    {
        // We need a rotation matrix for time t + delta_t.
        R = cur_q[i].ConvertToRotationMatrix();
// We next compute the inverse inertia tensor
// in world space, which is used in Equation 36.
J = R * JO bjInv[i] * R.Transpose();

// We are now in a position to update the
// angular velocity using Equation 36.
cur_w[i] = J * cur_S[3*i+2];
}

// Now we’re done with the integration!

// By integrating the equations of motion, we have
// effectively moved simulation time forward by delta_t.
t = t + delta_t;

A Brief Word about Integrators and
Different State Variable Types

If you look carefully at Listing 4.3.9, you will see that the call to ExplicitEuler looks identical regardless of whether the state vectors and derivatives contain Vector3D objects or Quaternion objects. In addition, you may wonder how you can actually create a single integrator function that can integrate state variables of different types. There are a couple of ways to accomplish this. One approach is to flatten all state variables (e.g., vectors and quaternions) into an array of floating-point values, and use an integrator that simply integrates an array of floating-point state values. Another approach, using object-oriented programming, is to derive all state variables from a base State class, and ensure that all concrete state classes overload the operators required by the integrator: -, +, and *.

Let’s look at the first approach, in which the state variables are flattened to an array of floats. A Vector3D object can be represented as an array of three floats, and a Quaternion object can be represented as an array of four floats. Listing 4.3.10 shows how you might represent a collection of object states that follow this approach, along with the call to ExplicitEuler. The state variable vector includes mV, p, L, and q in a single floating-point array.

Listing 4.3.10 Object states flattened into an array of floating-point values

```c
float cur_S[13*N];    // Current state
float prior_S[13*N];  // Prior state
float S_derivs[13*N]; // State derivatives

// for object i
cur_S[13*i + 0] = linear momentum x component;
cur_S[13*i + 1] = linear momentum y component;
cur_S[13*i + 2] = linear momentum z component;
cur_S[13*i + 3] = position x component;
```
4.3 Real-Time Game Physics

\[
\begin{align*}
\text{cur}_S[13*i + 4] &= \text{position y component}; \\
\text{cur}_S[13*i + 5] &= \text{position z component}; \\
\text{cur}_S[13*i + 6] &= \text{angular momentum x component}; \\
\text{cur}_S[13*i + 7] &= \text{angular momentum y component}; \\
\text{cur}_S[13*i + 8] &= \text{angular momentum z component}; \\
\text{cur}_S[13*i + 9] &= \text{orientation real component}; \\
\text{cur}_S[13*i + 10] &= \text{orientation imaginary i component}; \\
\text{cur}_S[13*i + 11] &= \text{orientation imaginary j component}; \\
\text{cur}_S[13*i + 12] &= \text{orientation imaginary k component};
\end{align*}
\]

\[\text{prior}_S[13*i + 0] \text{ through prior}_S[13*i + 12] \text{ is similar;}\]

\[
\begin{align*}
\text{S}_\text{derivs}[13*i + 0] &= \text{Net force x component}; \\
\text{S}_\text{derivs}[13*i + 1] &= \text{Net force y component}; \\
\text{S}_\text{derivs}[13*i + 2] &= \text{Net force z component}; \\
\text{S}_\text{derivs}[13*i + 3] &= \text{Velocity x component}; \\
\text{S}_\text{derivs}[13*i + 4] &= \text{Velocity y component}; \\
\text{S}_\text{derivs}[13*i + 5] &= \text{Velocity z component}; \\
\text{S}_\text{derivs}[13*i + 6] &= \text{Net torque x component}; \\
\text{S}_\text{derivs}[13*i + 7] &= \text{Net torque y component}; \\
\text{S}_\text{derivs}[13*i + 8] &= \text{Net torque z component}; \\
\text{S}_\text{derivs}[13*i + 9] &= 0.0; \\
\text{S}_\text{derivs}[13*i + 10] &= \text{angular velocity x component}; \\
\text{S}_\text{derivs}[13*i + 11] &= \text{angular velocity y component}; \\
\text{S}_\text{derivs}[13*i + 12] &= \text{angular velocity z component};
\end{align*}
\]

// The following call integrates the vector values and // quaternion values all in one call. ExplicitEuler(13*N, new_S, prior_S, S_derivs, delta_t);

Using an object-oriented approach, the integrator parameters would be specified as arrays of a base object class type, and then derive all state variable types from the base class. The base class in this case must define pure virtual functions for the basic mathematical operators \(-\), \(+\), and \(*\), and the state variable classes must each provide concrete implementations of those operators. If you choose to use C++, and store the states in STL vectors, the state variable classes will also need to provide assignment operators and copy constructors to make the STL container classes happy. Listing 4.3.11 shows example code in pseudo-C++.

**Listing 4.3.11 Object-oriented state classes and integrator**

```cpp
class State {
    const State &operator+(const State &Other) = 0;
    const State &operator-(const State &Other) = 0;
    const State &operator*(const State &Other) = 0;
    const State &operator*(const float fFactor) = 0;
};
```
class Vector3D : public State
{
    const Vector3D &operator+(const Vector3D &Other);
    const Vector3D &operator-(const Vector3D &Other);
    const Vector3D &operator*(const Vector3D &Other);
    const Vector3D &operator*(const float fFactor);

    float m_fX, m_fY, m_fZ;    // Components of vector
};

class Quaternion : public State
{
    const Quaternion &operator+(const Quaternion &Other);
    const Quaternion &operator-(const Quaternion &Other);
    const Quaternion &operator*(const Quaternion &Other);
    const Quaternion &operator*(const float fFactor);

    float m_fR;       // Real component
    float m_fI, m_fJ, m_fK;    // Imaginary components
};

void main()
{
    ArrayContainer<State> cur_S;
    ArrayContainer<State> prior_S;
    ArrayContainer<State> S_derivs;
    for (i = 0; i < N; i++)
    {
        cur_S.Add(Vector3D(linear momentum of i));
        cur_S.Add(Vector3D(position of i));
        cur_S.Add(Vector3D(angular momentum of i));
        cur_S.Add(Quaternion(orientation of i));

        // prior_S and S_derivs follow similarly.
    }

    // In the simulation loop, just call the integrator by
    // passing references to cur_S, prior_S, and S_derivs.
}

void ExplicitEuler(ArrayContainer<State>& &new_S,
                    const ArrayContainer<State>& &prior_S,
                    const ArrayContainer<State>& &S_derivs,
                    delta_t)
{
    unsigned int N = new_S.size();
    for (i = 0; i < N; i++)
    {
        new_S[i] = prior_S[i] + delta_t * S_derivs[i];
    }
}

There certainly are other ways in which you might build an object-oriented numerical simulator; however, one strong benefit of the approach shown in Listing
4.3.11 is that the numerical integrator method need only exist in one location in your source code. This makes the code fairly easy to maintain.

**Collision Response Revisited**

Now that we understand how to simulate the motion of objects undergoing rotational motion, it is worth revisiting impulse-momentum-based collision response. Figure 4.3.12 illustrates a generalized, frictionless rigid-body collision. Because there is no friction, the collision impulse acts through the point of impact in the direction of a unit normal vector at the point of impact. As shown in Figure 4.3.12, the line of action of the impulse does not necessarily intersect the centers of mass of the objects involved, resulting in an impulsive torque that changes the rotational motion state of the objects.

![Diagram of collision response](image)

**FIGURE 4.3.12** Generalized frictionless collision.

The rotational analog to the linear impulse-momentum equation is the *angular impulse-momentum equation*, given by Equation 4.3.37 for frictionless collisions between two objects.

\[
L^+_i = L^-_i + \Lambda \left( r_i \times \hat{n} \right) \quad ; \quad L^+_2 = L^-_2 - \Lambda \left( r_2 \times \hat{n} \right) \tag{4.3.37}
\]

To compute generalized frictionless collision response, we must solve Equations 4.3.9 through 4.3.12 and 4.3.37 for the impulse. Equation 4.3.38 gives the resulting impulse value, which models both translational and rotational effects. The impulse vector is given by Equation 4.3.11.

\[
\Lambda = \frac{-(1+\epsilon)(\hat{n} \cdot (V^-_1 - V^-_2) + \omega^-_1 \cdot (r_i \times \hat{n}) - \omega^-_2 \cdot (r_2 \times \hat{n}))}{\frac{1}{m_1} + \frac{1}{m_2} + \left( (r_i \times \hat{n})^T J_1^{-1} (r_i \times \hat{n}) + (r_2 \times \hat{n})^T J_2^{-1} (r_2 \times \hat{n}) \right)} \tag{4.3.38}
\]
To compute the postcollision linear and angular momentums, simply apply the impulse from Equations 4.3.38 and 4.3.11 to Equations 4.3.9, 4.3.10, and 4.3.37. Note that Equation 4.3.38, when substituted into Equation 4.3.11, simplifies to Equation 4.3.13 when the line joining the two centers of mass intersects the impact point and is parallel to the contact normal, since all the cross products in Equation 4.3.38 become zero. This is to be expected, since there is no impulsive torque in this case.

A Brief Word on Alternative Collision Response Methods

The impulse-momentum approach to resolving collision response is robust, realistic, and reliable. However, in practice, it is no panacea. Case in point: as illustrated earlier, the numerical integration of state can be difficult to manage efficiently since collisions usually occur between \( t \) and \( t + \Delta t \), requiring a two-part integration—before and after the collision—with time steps that differ from the noncollision time step. Further, as discussed in Chapter 4.2, an expensive iteration may be required to determine the time and position of collision prior to computing the response. In some cases, this can be prohibitively slow when there are many simultaneous collisions between pairs or groups of objects. Alternative methods exist, and we briefly mention some of them here.

The so-called penalty force methods represent one alternative to impulse-based collision response. In the classical penalty force-based collision response, rather than calculate instantaneous changes in velocity and angular velocity in response to a collision, stiff springs are applied between objects that impact each other, with a displacement based on interpenetration depth. As the objects interpenetrate, the spring stretches, creating a force that attempts to reduce the interpenetration depth to approximately zero over a few integration steps. As with the impulse-momentum method, penalty force methods have a basis in reality. The spring force is a model of the real-life forces generated as two colliding objects deform and rebound. The effect of the spring force applied over a few physics integration steps is an approximation to the collision impulse. In a sense, penalty force methods are more realistic than the impulse-momentum method. Whereas the impulse-momentum approach models deformation and restitution as occurring instantaneously, the penalty-force methods approximately model deformation and restitution over a finite period of time, which is, of course, what happens in the real world.

The use of penalty-force methods is often problematic, for two reasons. First, to achieve reasonable-looking collision response, the springs must have a large stiffness, and this greatly increases the likelihood of numerical instability. (In particular, the explicit Euler integrator will often fail dramatically.) Second, if the spring stiffness is relaxed to avoid instability, significant interpenetration is likely, which is both physically and visually inconsistent. Implicit methods are highly recommended if you choose to implement penalty-based collision response, since they will be stable even for stiff penalty forces.
Despite the difficulties, state management is quite elegant with penalty methods, which do not require a two-part time integration for objects in collision. All objects can be updated with a single integration step and no iterative discovery of the exact time and location of collision, although a collision detection technique that can estimate interpenetration depth is required. Many real-world games use penalty force methods because of their elegance and simplicity. David Wu of Pseudo Interactive has demonstrated good success using penalty methods for collision response, using implicit Euler integration (a one-step implicit numerical integrator) and potential functions rather than springs to model the penalty force that keeps colliding objects separated [Wu00].

Jakobsen [Jakobsen03] presents a novel approach to collision response, which he implemented with good success in IO Interactive’s game, Hitman: Codename 47. In this approach, all objects are represented as collections of particles, rigidly connected together via a series of constraints designed to maintain the shape of the object and prevent articulating components from exceeding rotation limits. Rather than handle collision response in a physically based manner, when two or more objects are detected to interpenetrate, Jakobsen projects particles of each object such that they lie on the boundary of the other object involved in the collision. He then adjusts the positions of all of the particles of the objects involved such that the rigid connection constraints are satisfied. This process is performed iteratively, for a few repetitions, until the rigid connection constraints are satisfied and no particles of a given object remain inside the other object(s) involved in the collision. Jakobsen’s method can readily support simulation of rigid bodies, particles, cloth, and characters.

Baraff [Baraff] has written a number of papers, available online, that present various collision-response techniques and issues, including comprehensive discussions of the impulse-momentum approach summarized in this chapter.

**Bringing It All Together**

You now should have a good understanding of the major factors that cause and affect the motion of general rigid bodies, and a basic understanding of the numerical integration techniques that can be used to approximately solve for the motion of a collection of objects over time. Whether you are implementing simple spherical particle physics or general rigid-body physics, your basic process is rather straightforward, as outlined here:

1. Choose a numerical integrator (or integrators), which will determine the state vector and the number of prior states you must track.
2. Choose a representation for the state vector, $S$, and derivative vector, $dS/dt$ and/or $d^2S/dt^2$.
3. Choose a set of initial conditions for your rigid bodies, and assign these into $S$.
4. If your selected numerical integrator requires more than one prior state, use explicit Euler integration to initialize the entire set of prior states. Alternatively, initially set all prior states equal to the initial conditions.
5. During the simulation for each physics step, if collisions are detected and you are using impulse-momentum-based collision response, resolve them using the instantaneous impulse-momentum equations and update the state properties for the objects involved before continuing with the general update of all objects. Alternatively, if you are using penalty-force-based collision response, compute the penalty force on objects involved in collisions based on current interpenetration depths at the time $t$, and add the penalty force to the net force applied to each object. Then, update all objects in one step, including those in collision.

6. During the simulation for each physics step:
   a. Copy the current state $\bar{\mathbf{S}}$ into a temporary state array to be given to the integrator as one of the prior states. Copy other prior states if necessary.
   b. Calculate or copy the state derivative vector for each object.
   c. Call the integrator(s) to update the state vector.
   d. If simulating rotational motion, update the angular velocity every frame, and renormalize the orientation matrix and/or orientation quaternion every few frames.

**Commercial and Freeware Physics Engines**

Currently, there is a handful of ready-made rigid-body physics engines available for games. Many of them include their own collision detection, and many also run on both game consoles and PCs. These engines can save you the trouble of actually writing your own comprehensive and robust physics code; however, you still need a good understanding of physics to use the engines properly. You will not obtain good results, for example, if you do not provide inputs that use consistent units, or if you provide an incorrect inertia tensor.

The following is a list of several physics engines that are available as of 2005.

**Commercial Physics Engines**

- Game Dynamics SDK by Havok.com, Inc. ([www.havok.com](http://www.havok.com))
- Renderware Physics by Renderware, Inc. ([www.renderware.com](http://www.renderware.com))
- NovodeX SDK by NovodeX, Inc. ([www.novodex.com](http://www.novodex.com))

**Freeware/Shareware Physics Engines**

- Open Dynamics Engine (ODE) ([www.ode.org](http://www.ode.org))
- Tokamak Game Physics SDK ([www.tokamakphysics.com](http://www.tokamakphysics.com))
- Newton Game Dynamics SDK ([www.newtondynamics.com](http://www.newtondynamics.com))

**Licensing Issues**

As with any middleware software, whether labeled as commercial, shareware, or freeware, you should carefully read the license agreement terms before selecting products...
to use in your games. The licensing terms of these engines may change over time, and may be different for commercial projects, school projects, and hobby or giveaway projects. Freeware engines, for example, often require that you credit the middleware in any binaries, documentation, advertising, or source code you publish or distribute. Be certain that you are willing and able to comply with the license terms before deciding to use any products or tools that you did not develop yourself.

Summary

Closed-form particle physics are extremely practical for games that require only simple physics. One significant benefit of these equations, if they are suitable, is that they are perfectly stable and will never cause floating-point overflow. In practice, these equations are only useful for spherical particles experiencing occasional collisions and at most a constant acceleration, such as that due to gravity.

Numerical integration techniques remove the restriction that an object experiences only a constant force, making these techniques quite useful for implementing a general-purpose physics engine. These techniques are subject to stability concerns that you must consider carefully. Regardless of the stability considerations, these techniques open up a world of opportunity for physics simulation.

In this chapter, we developed the physics and numerical integration techniques that can be used to implement general rigid-body motion. However, the presentation here is by no means comprehensive. If you want to implement a full, rigid-body physics system, your studies have only just begun. For example, one rather obvious effect that we have deliberately ignored is the presence of surface friction during collisions. Impulsive friction is present during collisions, in the real world, and can affect motion. Friction in collisions can, for example, create a torque causing the objects to rotate, even if the vector joining the two objects' centers of mass intersects the contact point. Some of the other important features that we ignored include:

- Multiple contact points between a given pair of objects
- Simultaneous collisions between more than two bodies
- Articulating rigid body chains, in which two bodies are linked for all time via various hinges and joints, with limits
- The friction of rolling objects
- The details of mechanically applied forces, such as motor forces
- Dealing with resting contact and stacking of objects
- Breakable objects

The material presented here is a great starting point. The exercises that follow will solidify your comprehension of the theory and implementation of basic rigid-body physics. Beyond these rigid-body effects, you may need to implement a deformable or soft-body physics system, which can be used to simulate cloth, hair, and objects that wobble. You may also need to implement a fluid physics system, to simulate the motion of bodies of water, such as a raging river. Some of the references listed, and
many other sources not listed, provide good introductions to these advanced topics should you wish to explore them.

### Exercises

1. Use Equation 4.3.7 to create a simple targeting game in which the player launches a projectile particle at a target particle. You can implement a full 3D game, or a 2D system. Provide the player with the ability to change the launch speed of the particle and the launch direction (thus, setting \( V_{\text{init}} \)). Provide a “perfect launch” feature that allows the game to automatically choose the proper launch velocity to hit a given target. To do this, use Equations 4.3.4 and 4.3.7 together to solve for \( V_{\text{init}} \) and the time of impact, \( t_{\text{impact}} - t_{\text{start}} \), given that the particle position at time of impact is \( p_{\text{particle}}(t_{\text{impact}}) = p_{\text{target}} \).

2. Verify that the units of linear impulse, defined as \( \int_{t_{\text{start}}}^{t_{\text{end}}} F_{\text{collision}} dt \), are the same as the units of linear momentum. Determine what the units of the coefficient of restitution must be, so that Equation 4.3.12 is consistent.

3. Verify that the velocity in the contact plane for the frictionless collision of two spheres is unchanged during the collision response, using Equations 4.3.9 through 4.3.13.

4. Determine whether the direction of the surface normal is important when computing collision response (e.g., is the result still valid if you choose \( \hat{n} = -\hat{n} \)?)

5. Most game worlds, or simulation worlds, include objects that are immovable; for example, the terrain or structures fixed to the terrain. In physics terms, these immovable objects can be considered to have infinite mass; however, infinite mass is not physically feasible from the point of view of classical dynamics. Because of the infeasibility of infinite mass, conservation of linear momentum is not satisfied, and Equation 4.3.13 is invalid. However, Equation 4.3.12 remains valid. After all, there is still an action and reaction, and they can still be related using a coefficient of restitution. Note, though, that the before-collision and after-collision velocities of the immovable object are identical. Using Equation 4.3.12, derive the velocity of a 0.5 kg ball after a collision with the Earth, with Earth considered immovable. Your goal is to derive an equation that can be used for any valid values of \( V^* \) and \( \varepsilon \). Verify that the units of your equation are consistent.

6. Implement a simple top-down view marbles game. The goal of this game should be to allow the player to experiment with different marble collisions. Allow the player to choose two marbles at a time, each with a different mass. The player should be able to launch both marbles toward each other at the same time so that they are moving toward each other. Use Equation
4.3.7 to simulate the motion of the marbles prior to and after the collision, and Equations 4.3.9, 4.3.10, and 4.3.13 to determine the collision response.

7. Implement the projectile system of Exercises 1 and 6 using explicit Euler integration. Ensure that your implementation is frame-rate independent. Compare the results against the exact solutions from the other exercises.

8. Modify your implementation of Exercise 7 to use Verlet integration, and compare the results with the explicit Euler solution.

9. Use Equations 4.3.9 through 4.3.12 to prove Equation 4.3.13. Using Equation 4.3.37 also, prove Equation 4.3.38 (advanced).

10. Linear springs can be used to cheaply approximate soft bodies. Using Listing 4.3.8 as a guideline, create a simulation for a soft-body rectanguloid block (or other shape) being dropped on a hard surface. Use velocityless Verlet integration to update the positions, and use explicit Euler integration to compute velocities for the dashpots. Simulate the block being dropped at a slight angle from some height above an immovable plane, and use the result of Exercise 5 to implement collision of the particles with the immovable plane (choose \(\varepsilon\) to be 0.75). Experiment with different spring stiffness and dashpot damping coefficients. Observe the difference in behavior when the dashpots are enabled versus disabled.

11. Create a simple bumper-cars simulation, with rotational collisions. To do this, treat the cars as simple rectanguloid geometries. Implement a simple propulsion force that causes acceleration in the forward direction and is proportional to a throttle setting that the player can control with a keyboard or joystick. Implement a simple, constant braking force that acts in the rearward direction and is activated with the keyboard. When resolving collisions, take the point of contact to be at the same height as the center of mass of the car bodies. Implement a friction model that uses only dynamic friction.

12. Springs and dashpots can be used to approximate an automobile suspension. Simulate a vehicle moving at a constant speed over a speed bump. To do this, approximate the vehicle as a rectanguloid. Place a spring and damper at each corner, between the rectanguloid and the ground plane, to model the suspension. Represent the speedbump as a simple constant change in height of the ground plane. As each corner of the vehicle moves over the edge of the speed bump, force the bottom endpoint of the corresponding spring to be at the new ground height. Force the lower endpoint of the spring to remain fixed to the ground (e.g., the vehicle does not bounce) and apply the appropriate spring force to each corner of the vehicle. Use the same integration schemes as Exercise 11. How does the behavior of the vehicle change as you increase or decrease the spring stiffness, and if you change the width and length of the vehicle?
References


