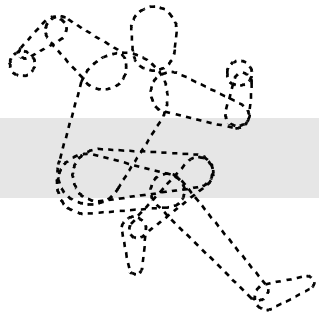

3 The Laws of Motion

Physics engines are based on Newton's laws of motion. In later sections we will begin to use results that were added to Newton's original work, but the fundamentals are his.

Newton created three laws of motion that describe with great accuracy how a point mass behaves. A *point mass* is what we call a "particle," but shouldn't be confused with particle physics, which studies tiny particles such as electrons or photons that definitely do not conform to Newton's laws. For this book we'll use *particle* rather than *point mass*.

Beyond particles we need the physics of rotating, which introduces additional complications that were added to Newton's laws. Even in these cases, however, the point-mass laws still can be seen at work.

Before we look at the laws themselves, and how they are implemented, we need to look at what a particle is within our engine and how it is built in code.

3.1 A Particle

A particle has a position, but no orientation. In other words, we can't tell in what direction a particle is pointing: it either doesn't matter or doesn't make sense. In the former category are bullets: in a game we don't really care in what direction a bullet is pointing; we just care in what direction it is traveling and whether it hits the target. In the second category are pinpricks of light, from an explosion, for example: the light is a dot of light, and it doesn't make sense to ask in which direction a spot of light is pointing.

For each particle we'll need to keep track of various properties: we'll need its current position, its velocity, and its acceleration. We will add additional properties to the particle as we go. The position, velocity, and acceleration are all vectors.

The particle can be implemented with the following structure:

Excerpt from include/cyclone/particle.h

```
/**
 * A particle is the simplest object that can be simulated in the
 * physics system.
 */
class Particle
{
public:
    /**
     * Holds the linear position of the particle in
     * world space.
     */
    Vector3 position;

    /**
     * Holds the linear velocity of the particle in
     * world space.
     */
    Vector3 velocity;

    /**
     * Holds the acceleration of the particle. This value
     * can be used to set acceleration due to gravity (its primary
     * use) or any other constant acceleration.
     */
    Vector3 acceleration;

};
```

Using this structure we can apply some basic physics to create our first physics engine.

3.2 The First Two Laws

There are three law of motion posited by Newton, but for now we will need only the first two. They deal with the way an object behaves in the presence and absence of forces. The first two laws of motion are:

1. An object continues with a constant velocity unless a force acts upon it.

2. A force acting on an object produces acceleration that is proportional to the object's mass.

3.2.1 The First Law

The first law (Newton 1) tells us what happens if there are no forces around. The object will continue to move with a constant velocity. In other words, the velocity of the particle will never change, and its position will keep on being updated based on the velocity. This may not be intuitive: moving objects we see in the real world will slow and come to a stop eventually if they aren't being constantly forced along. In this case the object is experiencing a force—the force of drag (or friction if it is sliding along). In the real world we can't get away from forces acting on a body; the nearest we can imagine is the movement of objects in space. What Newton 1 tells us is that if we could remove all forces, then objects would continue to move at the same speed forever.

In our physics engine we could simply assume that there are no forces at work and use Newton 1 directly. To simulate drag we could add special drag forces. This is fine for the simple engine we are building in this part of the book, but it can cause problems with more complex systems. The problem arises because the processor that performs the physics calculations isn't completely accurate. This inaccuracy can lead to objects getting faster of their own accord.

A better solution is to incorporate a rough approximation of drag directly into the engine. This allows us to make sure objects aren't being accelerated by numerical inaccuracy, and it can allow us to simulate some kinds of drag. If we need complicated drag (such as aerodynamic drag in a flight simulator or racing game), we can still do it the long way by creating a special drag force. We call the simple form of drag "damping" to avoid confusion.

To support damping, we add another property to the particle class:

Excerpt from include/cyclone/particle.h

```
class Particle
{
    // ... Other Particle code as before ...

    /**
     * Holds the amount of damping applied to linear
     * motion. Damping is required to remove energy added
     * through numerical instability in the integrator.
     */
    real damping;
};
```

When we come to perform the integration, we will remove a proportion of the object's velocity at each update. The damping parameter controls how velocity is left

after the update. If the damping is zero, then the velocity will be reduced to nothing: this would mean that the object couldn't sustain any motion without a force and would look odd to the player. A value of 1 means that the object keeps all its velocity (equivalent to no damping). If you don't want the object to look like it is experiencing drag, then values near but less than 1 are optimal—0.995, for example.

3.2.2 The Second Law

The second law (Newton 2) gives us the mechanism by which forces alter the motion of an object. A force is something that changes the *acceleration* of an object (i.e., the rate of change of velocity). An implication of this law is that we cannot do anything to an object to directly change its position or velocity; we can only do that indirectly by applying a force to change the acceleration and wait until the object reaches our target position or velocity. We'll see later in the book that physics engines need to abuse this law to look good, but for now we'll keep it intact.

Because of Newton 2, we will treat the acceleration of the particle different from velocity and position. Both velocity and position keep track of a quantity from frame to frame during the game. They change, but not directly, only by the influence of accelerations. Acceleration, by contrast, can be different from one moment to another. We can simply set the acceleration of an object as we see fit (although we'll use the force equations in Section 3.2.3), and the behavior of the object will look fine. If we directly set the velocity or position, the particle will appear to jolt or jump. Because of this the position and velocity properties will only be altered by the integrator and should not be manually altered (other than setting up the initial position and velocity for an object, of course). The acceleration property can be set at any time, and it will be left alone by the integrator.

3.2.3 The Force Equations

The second part of Newton 2 tells us how force is related to the acceleration. For the same force, two objects will experience different accelerations depending on their mass. The formula relating the force to the acceleration is the famous

$$f = ma = m\ddot{p} \tag{3.1}$$

Where we are trying to find the acceleration, we have

$$\ddot{p} = \frac{1}{m}f \tag{3.2}$$

where f is the force and m is the mass.

In a physics engine we typically find the forces applying to each object and then use equation 3.2 to find the acceleration, which can then be applied to the object by the integrator. For the engine we are creating in this part of the book, we can find

the acceleration in advance using this equation, without having to repeat it at every update. In the remainder of the book, however, it will be a crucial step.

So far all the equations have been given in their mathematics textbook form, applied to scalar values. As we saw in the last chapter on calculus, we can convert them easily to use vectors:

$$\ddot{\boldsymbol{p}} = \frac{1}{m}\boldsymbol{f}$$

so the force is a vector, as well as acceleration.

3.2.4 Adding Mass to Particles

Added to the position and velocity we have stored per particle, we need to store its mass so that we can correctly calculate its response to forces. Many physics engines simply add a scalar mass value for each object. There is a better way to get the same effect, however.

First there is an important thing to notice about equation 3.2. If the mass of an object is zero, then the acceleration will be infinite, as long as the force is not zero. This is not a situation that should ever occur: no particle that we can simulate should ever have zero mass. If we try to simulate a zero mass particle, it will cause divide-by-zero errors in the code.

It is often useful, however, to simulate infinite masses. These are objects that no force of any magnitude can move. They are very useful for immovable objects in a game: the walls or floor, for example, cannot be moved during the game. If we feed an infinite mass into equation 3.2, then the acceleration will be zero, as we expect. Unfortunately we cannot represent a true infinity in most computer languages, and the optimized mathematics instructions on all common game processors do not cope well with infinities. We have to get slightly creative. Ideally we want a solution where it is easy to get infinite masses but impossible to get zero masses.

Notice in equation 3.2 that we use 1 over the mass each time. Because we never use the 3.1 form of the equation, we can speed up our calculations by not storing the mass for each object, but 1 over the mass. We call this the "inverse mass." This solves our problem for representing objects of zero or infinite mass: infinite mass objects have a zero inverse mass, which is easy to set. Objects of zero mass would have an infinite inverse mass, which cannot be specified in most programming languages.

We update our particle class to include the inverse mass in this way:

Excerpt from include/cyclone/particle.h

```
class Particle
{
    // ... Other Particle code as before ...

    /**
     * Holds the inverse of the mass of the particle. It
     * is more useful to hold the inverse mass because
```

```
     * integration is simpler and because in real-time
     * simulation it is more useful to have objects with
     * infinite mass (immovable) than zero mass
     * (completely unstable in numerical simulation).
     */
    real inverseMass;
};
```

It is really important to remember that you are dealing with the inverse mass, and not the mass. It is quite easy to set the mass of the particle directly, without remembering, only to see it perform completely inappropriate behavior on screen—barely movable barrels zooming off at the slightest tap, for example.

To help with this, I've made the `inverseMass` data field protected in the `Particle` class on the CD. To set the inverse mass you need to use an accessor function. I have provided functions for `setInverseMass` and `setMass`. Most of the time it is more convenient to use the latter, unless we are trying to set an infinite mass.

3.2.5 Momentum and Velocity

Although Newton 1 is often introduced in terms of velocity, that is a misrepresentation. It is not velocity that is constant in the absence of any forces, but momentum.

Momentum is the product of velocity and mass. Since mass is normally constant, we can assume that velocity is therefore constant by Newton 1. In the event that a traveling object were changing mass, then its velocity would also be changing, even with no forces.

We don't need to worry about this for our physics engine because we'll not deal with any situation where mass changes. It will be an important distinction when we come to look at rotations later, however, because rotating objects can change the way their mass is distributed. Under the rotational form of Newton 1, that means a change in rotational speed, with no other forces acting.

3.2.6 The Force of Gravity

The force of gravity is the most important force in a physics engine. Gravity applies between every pair of objects: attracting them together with a force depends on their mass and their distance apart. It was Newton who also discovered this fact, and along with the three laws of motion he used it to explain the motion of planets and moons with a new level of accuracy.

The formula he gave us is the "law of universal gravitation":

$$f = G\frac{m_1 m_2}{r^2} \qquad [3.3]$$

where m_1 and m_2 are the masses of the two objects, r is the distance between their centers, f is the resulting force, and G is the "universal gravitational constant"—a scaling factor derived from observation of planetary motion.

The effects of gravity between two objects the size of a planet are significant; the effects between (relatively) small objects such as a car, or even a building, are small. On earth our experience of gravity is completely dominated by the earth itself. We notice the pull of the moon in the way our tides work, but other than that we only experience gravity pulling us down to the planet's surface.

Because we are only interested in the pull of the earth, we can simplify equation 3.3. First we can assume that m_1 is always constant. Second, and less obviously, we can assume that r is also constant. This is due to the huge distances involved. The distance from the surface of the earth to its center is so huge (6,400 km) that there is almost no difference in gravity between standing at sea level and standing on the top of a mountain. For the accuracy we need in a game, we can therefore assume the r parameter is constant.

Equation 3.3 simplifies to

$$f = mg \qquad [3.4]$$

where m is the mass of the object we are simulating; f is the force, as before; and g is a constant that includes the universal gravitational constant, the mass of the earth, and its radius:

$$g = G\frac{m_{\text{earth}}}{r^2}$$

The constant, g, is an acceleration, which we measure in meters per second per second. On earth this g constant has a value of around 10 m/s^2. (Scientists sometimes use a value of 9.807 m/s^2, although because of the variations in r and other effects, this is a global average rather than a measured value.)

Notice that the force depends on the mass of the object. If we work out the acceleration using equation 3.2, then we get

$$\ddot{p} = \frac{1}{m}mg = g$$

In other words, no matter what mass the object has, it will always accelerate at the same rate due to gravity. As the legend goes, Galileo dropped heavy and light objects from the Tower of Pisa and showed that they hit the ground at the same time.

What this means for our engine is that the most significant force we need to apply can be applied directly as an acceleration. There is no point using equation 3.4 to calculate a force and then using equation 3.2 to convert it back into an acceleration. In this iteration of the engine we will introduce gravity as the sole force at work on particles, and it will be applied directly as an acceleration.

The Value of g

It is worth noting that, although the acceleration due to gravity on earth is about 10 m/s^2, this doesn't look very convincing on screen. Games are intended to be more exciting than the real world: things happen more quickly and at a higher intensity.

Creating simulations with a g value of 10 m/s^2 can look dull and insipid. Most developers use higher values, from around 15 m/s^2 for shooters (to avoid projectiles being accelerated into the ground too quickly) to 20 m/s^2, which is typical of driving games. Some developers go further and incorporate the facility to tune the g value on an object-by-object basis. Our engine will include this facility.

Gravity typically acts in the down direction, unless you are going for a special effect. In most games the Y axis represents up and down in the game world; and almost exclusively the positive Y axis points up.

The acceleration due to gravity can therefore be represented as a vector with the form

$$\boldsymbol{g} = \begin{bmatrix} 0 \\ -g \\ 0 \end{bmatrix}$$

where g is the value we discussed before, and $\boldsymbol{g}$ is the acceleration vector we will use to update the particle in the next section.

3.3 The Integrator

We now have all the equations and background needed to finish the first implementation of the engine. At each frame, the engine needs to look at each object in turn, work out its acceleration, and perform the integration step. The calculation of the acceleration in this case will be trivial: we will use the acceleration due to gravity only.

The integrator consists of two parts: one to update the position of the object, and the other to update its velocity. The position will depend on the velocity and acceleration, while the velocity will depend only on the acceleration.

Integration requires a time interval over which to update the position and velocity: because we update every frame, we use the time interval between frames as the update time. If your engine is running on a console that has a consistent frame rate, then you can hard-code this value into your code (although it is wise not to because in the same console different territories can have different frame rates). If you are running on a PC with a variable frame-rate, then you probably need to time the duration of the frame.

Typically developers will time a frame and then use that value to update the next frame. This can cause noticeable jolts if frame durations are dramatically inconsistent, but the game is unlikely to feel smooth in this case anyway, so it is a common rule of thumb.

In the code on the CD I use a central timing system that calculates the duration of each frame. The physics code we will develop here simply takes in a time parameter when it updates and doesn't care how this value was calculated.

3.3.1 The Update Equations

We need to update both position and velocity; each is handled slightly differently.

Position Update

In chapter 2 we saw that integrating the acceleration twice gives us this equation for the position update:

$$p' = p + \dot{p}t + \frac{1}{2}\ddot{p}t^2$$

This is a well-known equation seen in high school and undergraduate textbooks on applied mathematics.

We could use this equation to perform the position update in the engine, with code something like

```
object.position += object.velocity * time +
                   object.acceleration * time * time * 0.5;
```

or

```
object.position.addScaledVector(object.velocity, time);
object.position.addScaledVector(object.acceleration, time * time * 0.5);
```

In fact, if we are running the update every frame, then the time interval will be very small (typically 0.033 s for a 30-frames-per-second game). If we look at the acceleration part of this equation, we are taking half of the squared time (which gives 0.0005). This is such a small value that it is unlikely the acceleration will have much of an impact on the change in position of an object.

For this reason we typically ignore the acceleration entirely in the position update and use the simpler form

$$p' = p + \dot{p}t$$

This is the equation we will use in the integrator throughout this book.

If your game regularly uses short bursts of huge accelerations, then you might be better off using the longer form of the equation. If you do intend to use huge accelerations, however, you are likely to get all sorts of other accuracy problems in any case: all physics engines typically become unstable with very large accelerations.

Velocity Update

The velocity update has a similar basic form

$$\dot{p}' = \dot{p} + \ddot{p}t$$

Earlier in the chapter, however, we introduced another factor to alter the velocity: the damping parameter.

The damping parameter is used to remove a bit of velocity at each frame. This is done by simply multiplying the velocity by the damping factor

$$\dot{p}' = \dot{p}d + \ddot{p}t \qquad [3.5]$$

where d is the damping for the object.

This form of the equation hides a problem, however. No matter whether we have a long or a short time interval over which to update, the amount of velocity being removed is the same. If our frame rate suddenly improves, then there will be more updates per second and the object will suddenly appear to have more drag. A more correct version of the equation solves this problem by incorporating the time into the drag part of the equation:

$$\dot{p}' = \dot{p}d^t + \ddot{p}t \qquad [3.6]$$

where the damping parameter d is now the proportion of the velocity retained each second, rather than each frame.

Calculating one floating-point number to the power of another is a slow process on most modern hardware. If you are simulating a huge number of objects, then it is normally best to avoid this step. For a particle physics engine designed to simulate thousands of sparks, for example, use equation 3.5, or even remove damping altogether.

Because we are heading toward an engine designed for simulating a smaller number of rigid bodies, I will use the full form in this book, as given in equation 3.6. A different approach favored by many engine developers is to use equation 3.5 with a damping value very near to 1—so small that it will not be noticeable to the player but big enough to be able to solve the numerical instability problem. In this case a variation in frame rate will not make any visual difference. Drag forces can then be created and applied as explicit forces that will act on each object (as we'll see in chapter 5).

Unfortunately, this simply moves the problem to another part of the code—the part where we calculate the size of the drag force. For this reason I prefer to make the damping parameter more flexible and allow it to be used to simulate visible levels of drag.

3.3.2 The Complete Integrator

We can now implement our integrator unit. The code looks like this:

Excerpt from include/cyclone/precision.h

```
/** Defines the precision of the power operator. */
#define real_pow powf
```

Excerpt from include/cyclone/particle.h

```
class Particle
{
    // ... Other Particle code as before ...

    /**
     * Integrates the particle forward in time by the given amount.
     * This function uses a Newton-Euler integration method, which is a
     * linear approximation of the correct integral. For this reason it
     * may be inaccurate in some cases.
     */
    void integrate(real duration);
};
```

Excerpt from src/particle.cpp

```
#include <assert.h>
#include <cyclone/particle.h>

using namespace cyclone;

void Particle::integrate(real duration)
{
    assert(duration > 0.0);

    // Update linear position.
    position.addScaledVector(velocity, duration);

    // Work out the acceleration from the force.
    Vector3 resultingAcc = acceleration;
    resultingAcc.addScaledVector(forceAccum, inverseMass);

    // Update linear velocity from the acceleration.
    velocity.addScaledVector(resultingAcc, duration);

    // Impose drag.
    velocity *= real_pow(damping, duration);
}
```

I have added the integration method to the `Particle` class because it simply updates the particles' internal data. It takes just a time interval and updates the position and velocity of the particle, returning no data.

3.4 Summary

In two short chapters we've gone from coding vectors to a first complete physics engine.

The laws of motion are elegant, simple, and incredibly powerful. The fundamental connections that Newton discovered drive all the physical simulations in this book. Calculating forces, and integrating position and velocity based on force and integrating position and velocity based on force and time, are the fundamental steps of all physics engines, complex or simple.

Although we now have a physics engine that can actually be used in games (and is equivalent to the systems used in many hundreds of published games), it isn't yet suitable for a wide range of physical applications. In chapter 4 we'll look at some of the applications it can support and some of its limitations.