CHAPTER 14
Physics Engines

A *physics engine* is the part of your game that contains all the code required for whatever you're trying to simulate using physics-based techniques. For many game programmers, a physics engine is a real-time, rigid-body simulator such as the sort we've discussed earlier in this book. The open source and licensable physics engines available to you are typically of the rigid-body-simulator variety. Some physics engines are rather generic and are useful for general rigid bodies and particles; others include various connectors and constraints, enabling ragdoll simulation. Still others focus on soft bodies and fluids. Fewer actually focus on the physics of some specific thing, like a car or a boat. A simple Internet search on the phrase "game physics engine" will generate many links to potential options for your use. That said, you could always write your own physics engine.

Building Your Own Physics Engine

We're advocates of using physics where you need it. Sure, you can write a general-purpose physics engine for a game, but if you're creating a game that doesn't require a general-purpose physics engine, then don't write one. That may sound obvious, but sometimes we are compelled to do more than what we need just so we can say we did it. Aside from the effort involved, a general-purpose physics engine will probably be less efficient than a purpose-built physics engine. By *purpose-built*, we mean designing the physics engine specifically to suit what you're trying to simulate. For example, a general-purpose physics engine would surely include particles, rigid bodies, connectors, other force effectors, and who knows what else—fluids, perhaps—and be fully 3D. But if you're writing a 2D side-scrolling game for a smartphone, you certainly won't need 3D with the associated complexities involved in dealing with rotation and collisions in 3D; and if your game simply involves throwing a ball of fuzz at some arbitrary junk, then you may not even need to deal with rigid bodies at all. We're being somewhat facetious here, but the point is, unless you *must* write a general-purpose physics engine —say, if you plan to license it as a middleware product or use it in a variety of game

types—then don't write one. Instead, write one specifically optimized for the game you're working on.

Let's consider a few examples. Let's say you're writing a 3D first-person shooter and you want to use physics to simulate how wooden barrels and crates blow apart when shot. Typically, such an effect would show pieces of wood flying off in different directions while falling under the influence of gravity. You could simulate such an effect in 3D using rigid bodies and you wouldn't even need to consider collisions, unless you wanted the pieces to bounce off each other or other objects. Ignoring these aspects greatly simplifies the underlying physics engine. Consider another example. Let's say you're working on a game involving flying an airplane. You can use physics to simulate the flight dynamics, as we explain in this book, without the need for particles, connectors, or even collision response.

The point of all this discussion is that you should consider which aspects of your game will really benefit from physics and write your physics engine to deal specifically with those aspects.

Another thing to consider is whether or not you need real-time physics. You might expect, after reading the available game physics literature, that your game must include real-time simulations if it is to incorporate physics. However, there are many ways to include physics in a game without having to solve the physics via real-time simulations. We show you an example in Chapter 19 whereby a golf swing is simulated in order to determine club head velocity at the time of club-ball impact. In this case, given specific initial parameters, we can solve the swing quickly, almost instantaneously, to determine the club speed, which can then be used as an initial condition for the ball flight. The ball's flight can be solved quickly as well and not necessarily in real time. It really depends on how you want to present the result to the player. If your game involves following the flight of the ball as it soars through the air, then you might want to simulate its flight in real time so you can realistically move the ball and camera. If, however, you simply want to show where the ball ends up, then you need not perform the simulation in real time. For such a simple problem, you can solve for the final ball location quicker than real time. Sometimes, the action you're simulating may happen so fast in real life that you'll want to slow it down for your game. Following a golf ball's flight in real time might have the camera moving so fast that your player won't be able to enjoy the beautifully rendered bird's-eye view of the course. In this case, you rapidly solve the flight path, save the data, and then animate the scene at a more enjoyable pace of your choosing.

We don't want to come across as trying to talk you out of writing a physics engine if you so choose. The point of our discussion so far is that you simply don't have to write a generic, real-time physics engine in order to use physics in your games. You have other options as we've just explained.

Assuming that, after all these considerations, you need to write a physics engine, then we have the following to offer.

A physics engine is just one component of a game engine. The other components include the graphics engine, audio engine, AI engine, and whatever other engines you may require or whatever other components of a game you may elevate to the status of engine. Whatever the case, the physics engine handles the physics. Depending on whom you talk to, you'll get different ideas on what composes a physics engine. Some will say that the heart of the physics engine is the collision detection module. Well, what if your game doesn't require collision detection, yet it still uses physics to simulate certain behaviors or features? Then collision detection certainly cannot be the heart of your physics engine. Some programmers will certainly take issue with these statements. To them, a physics engine simulates rigid-body motion using Newtonian dynamics while taking care of collision detection and response. To us, a significant component of a physics engine is the model—that is, the idealization of the thing you're trying to simulate in a realistic manner. You cannot realistically simulate the flight characteristics of a specific aircraft by treating it as a generic rigid body. You have to develop a representative model of that aircraft including very specific features; otherwise, it's a hack (which, by the way, we recognize as a valid and long-established approach).

Earlier, in Chapter 7 and Chapter 13, we showed you several example simulations. While simple, these examples include many of the required components of a generic physics engine. There are the particle and rigid-body classes that encapsulate generic object properties and behaviors, physics models that govern object behaviors, collision detection and response systems, and a numerical integrator. Additionally, those examples include interfacing the physics code with user input and visual feedback. These examples also show the basic flow from user input to physics solver to visual feedback.

In summary, the major components of a generic physics engine include:

- Physics models
- Simulated objects manager
- Collision detection engine or interface thereto
- Collision response module
- Force effectors
- Numerical integrator
- Game engine interface

Physics Models

Physics models are the idealizations of the things you're simulating. If your physics engine is a generic rigid-body simulator used to simulate an assortment of solid objects your players can knock around, throw, shoot, and generally interact with in a basic manner, then the physics model will probably be very generic. It's probably safe to as-

sume that each object will be subject to gravity's pull, thus mass will be an important attribute. Size will also be important, not only because you'll need to know how big things are when checking for collisions and handling other interactions, but also because size is related to the distribution of the object's mass. More precisely, each object will have *mass moment of inertia* attributes. The objects will most likely also have some ascribed coefficient of restitution that will be used during collision response handling. Additionally, you might ascribe some friction coefficients that may be used during collision response or in situations where the objects may slide along a floor. As the objects will likely find themselves airborne at some point, you'll probably also include a drag coefficient for each object. All of these parameters will help you differentiate massive objects from lighter ones or compact objects from voluminous ones.

If your simulation involves more than generic rigid bodies, then your physics will be more specific and perhaps far more elaborate. A great example of a more complicated model is flight simulation. No matter how good your generic rigid-body model, it won't fly like any specific aircraft if it flies at all. You must develop a model that captures flight aerodynamics specific to the aircraft you're simulating. Chapter 15 shows how to put together a model for an aircraft that can be used in a real-time flight simulation.

The other chapters in Part IV of this book are meant to give you a taste of modeling aspects for a variety of things you might simulate in a game. Just as you cannot simulate an aircraft with a generic rigid-body model, you cannot simulate a ship with an aircraft model, nor can you simulate a golf ball with a ship model. The point is that you must spend some time designing your physics model specific to what you're going to simulate in your game. Time spent here is just as important to creating a realistic physics engine as time spent on designing a robust integration scheme or collision detection system. We can't overstate the importance of the physical model. The model is what defines the behavior of the thing you're simulating.

Simulated Objects Manager

Your simulated objects manager will be responsible for instantiating, initializing, and disposing of objects. It will also be responsible for maintaining links between object physics and other attributes such as geometry, for example, if in a 3D simulation you use the same polyhedron to render an object and for collision detection and response.

You must have some means of managing the objects in your simulation. One can imagine many different approaches to managing these objects, and unless your simulation uses just a handful of objects or fewer, essentially what you need is a list of objects of whatever class you've defined. You've seen in previous chapters' examples where we use simple arrays of `RigidBody` type objects or `Particle` type objects. If all the objects in your simulation are the same, then you need only a single class capturing all their behavior. However, for more diversity, you should use a list of various classes with each class encapsulating the code required to implement its own physical model. This is

particularly important with respect to the forces acting on the model. For example, you could have some objects representing projectiles with others representing aircraft. These different classes will share some common code (for example, collision detection); however, the way forces are computed on each will vary due to the differences in how they are modeled. With such an approach, each class must have code that implements its particular model. During integration, the entire list will be traversed, calling the force aggregation method for each object, and the particular class will handle the details suitable for the type of object.

In some simple cases, you need not use different object classes if the types of objects you plan to use are not too different. For example, it would be fairly straightforward to implement a single class capable of handling both particles and rigid bodies. The object class could include an object type property used to denote whether the object is a particle or a rigid body, and then the class methods would call the appropriate code. Again, this will work satisfactorily for simple objects with few differences. If you want to simulate more than two types of objects or if they are very different, you're probably better off using different classes specific to each object being simulated.

However you structure your classes or lists, the flow of processing your objects will generally be the same. Every physics *tick*—that is, every time step in the physics simulation—you must check for object collisions, resolve those collisions, aggregate the usual forces on each object, integrate the equations of motion for each object, and then update each object's state.

As we said, this is the general flow at every physics tick, or time step, which may not be the same as your rendering steps. For example, for accuracy in your simulation you may have to take small steps around a millisecond or so. You wouldn't want to update the graphics every millisecond when you need only about a third as many graphics updates per second. Thus, your objects manager will have to be integrated with your overall game engine, and your game engine must be responsible for making sure the physics and graphics are updated appropriately.

Collision Detection

If collisions are an important part of your game, then a robust collision detection system is required.

Your collision detection system is distinct from the collision response system or module, though the two go hand in hand. Collision detection is the computational geometry problem of determining if objects collide and, if so, what points are making contact. These points are sometimes called the *contact manifold*. They're just the points that are touching, which could be a line or surface, though for simplicity usually the point, end points, or points defining the contact surface boundary are all that are included in the contact manifold.

The collision detection system's role is very specific: determine which objects are colliding, what points on each object are involved in the collision, and the velocities of those points. It sounds straightforward, but actual implementation can get quite complex. There are situations where fast-moving objects may go right though other objects, especially thin ones, over a single time step, making the collision detection system miss the collision if it relies solely on checking the separation distance between objects and their relative velocity (i.e., it detects a collision if the objects are within some collision tolerance and are also moving toward each other). A robust collision detection system will capture this situation and respond accordingly. In Chapter 8 we simply check if particles moved past the ground over the course of a single time step, for example, and then reset their position to that of the ground plane level. We can handle many situations using such simple techniques, especially when dealing with objects passing through floors or walls; however, other situations may require more complex algorithms to predict if a collision will occur sometime in the near future depending on how fast objects are moving relative to each other. This latter case is called *continuous* collision detection, and it is covered in many Internet, book, and technical paper sources. Many commercial and open source physics engines advertise their capability to handle continuous collision detection.

Another challenge associated with collision detection is the fact that it can be very time-consuming if you have a large or even moderate number of objects in your simulation. There are various techniques to deal with this. First, the game space is partitioned in some coarse grid-like manner, and this grid is used to organize objects depending on which cell they occupy. Then, in the second phase of collision detection, only those objects occupying adjacent cells are checked against each other to see if they are colliding. Without this grid partitioning, pairwise checks of every object against every other object would be very computationally expensive. The second phase of collision detection is often a broad approach using bounding spheres or bounding boxes, which may be axis- or body-aligned. If the bounding spheres or boxes of each object are found to collide, then the objects likewise may be colliding and further checks will be required; otherwise, we can infer that the objects are not colliding. In the case of a potential collision, these further checks become more complex depending on the geometry of the objects. This phase generally involves polygon- and vertex-level checks; there are well-established techniques for performing such checks that we won't get into here. Again, there's a wealth of literature on collision detection available online.

Collision Response

Once the collision detection system does its job, it's time for the collision response system to deal with the colliding objects. Earlier in this book, we showed you how to implement an impulse-moment collision response method. Recall that this method assumes that at the instant of collision, the most significant forces acting on the objects are the collision forces, so all other forces can be ignored for that instant. The method then com-

putes the resulting velocities of the objects after colliding and instantly changes their velocities accordingly. To perform the required calculations, the collision response system requires the objects colliding, of course, the collision points, and the velocities of those points. Also, each object must have some associated mass and coefficient of restitution, which are likewise used to compute the resulting velocities of the objects after the collision.

In practice, the collision response system works hand in hand with the collision detection system, particularly when dealing with objects that may be penetrating each other. As we mentioned earlier, in many cases an object penetrating another, such as an object penetrating a wall or floor, can simply be moved so that it is just touching the wall or floor. Other cases may be more complicated, and iterative algorithms are used to resolve the penetration. For example, if penetration is detected, then the simulation may back up to the previous time step and take a short time step to see if penetration still occurs. If it does not, the simulation proceeds; otherwise, the simulation takes an even smaller time step. This process repeats until penetration does not occur. This works fine in many cases; however, sometimes the penetration is never resolved and the simulation could get stuck taking smaller and smaller time steps. This failure to resolve could be attributed to objects that are just outside the collision distance tolerance at one instant, and due to numerical errors, exceedingly small time steps are required to stay outside of the distance tolerance. Some programmers simply put an iteration limit in the code to prevent the simulation from getting stuck, but the consequences in every situation may be unpredictable. The continuous collision detection approach we mentioned earlier avoids this sort of problem by predicting the future collision or penetration and dealing with it ahead of time. Whatever the approach, there will be some back and forth and data exchange between your collision detection and response systems to avoid excessive penetration situations.

Additionally, there are situations when an object may come to rest in contact with another—for example, a box resting on the floor. There are many ways to deal with such contact situations, one of which is to just allow the impulse-momentum approach to deal with it. This works just fine in many cases; however, sometimes the objects in resting contact will jitter with the impulse-momentum approach. One resolution to this jittering problem is to put those objects to sleep—that is, if they are found to be colliding, but their relative velocities are smaller than some tolerance, they are put to sleep. A related but somewhat more complicated approach is to compute the contact normal between the object and the floor and set that velocity to 0. This serves as a constraint, preventing the object from penetrating the floor while still allowing it to slide along the floor.

Force Effectors

Force effectors apply direct or indirect force on the objects in your simulations. Your physics engine may include several. For example, if your engine allows users to move

objects around with the mouse, then you'll need some virtualization of the force applied by the user via the mouse or a finger on a touch screen. This is an example of a direct force. Another direct force effector could be a virtual jet engine. If you associate that virtual engine, which produces some thrust force, with some object, then the associated object will behave as though it were pushed around by the jet.

Some examples of indirect force effectors include gravity and wind. Gravity applies force on objects by virtue of their mass, but it is typically modeled as body acceleration and not an explicit force. Wind can be viewed as exerting a pressure force on an object, and that force will be a function of the object's size and drag coefficients.

You can imagine all sorts of force effectors, from ones similar to those just described to perhaps some otherworldly ones. Whatever you imagine, you must remember that a force has magnitude, direction, and some central point of application. If you put a jet engine on the side of a box, the box will not only translate but will spin as well. Wind creates a force that has a center of pressure, which is the point through which you can assume the total wind force acts. The direction of the force and point of application are important for capturing both translation and rotation. As an example, consider the hovercraft we modeled in Chapter 9 that included two bow thrusters for steering and a propeller for forward motion. Each of these direct force effectors—the bow thrusters and the propeller—is applied at specific locations on the hovercraft. The bow thrusters are located toward the bow and point sideways in order to create spin, thus allowing some steering. The propeller is located on the center line of the hovercraft, which passes through the hovercraft's center of gravity so that it does not create spin and instead simply pushes the craft forward. There's another force effector in that model—aerodynamic drag, which is an indirect force effector. The drag force is applied at a point aft of the center of gravity so that it creates some torque, or moment, which in this model helps keep the hovercraft pointed straight; it provides some directional stability.

Whatever force effectors you contrive, they all must be aggregated for each object and dealt with in your numerical integrator. Thus, your integrator must have some means of accessing all the force effector information required to accurately simulate their effect on each associated object.

Numerical Integrator

The integrator is responsible for solving the equations of motion for each object. We showed you how to do this earlier, in Chapter 7 through Chapter 13. In your generic physics engine you'll iterate through all the objects in the simulation to compute their new velocities, positions, and orientations at every time step. To make these calculations, your integrator must have access to the force effectors associated with each object. The forces will be used to compute accelerations, which will then be integrated to compute velocities, and those in turn will be used to compute positions and orientations.

You can handle aggregating the forces in a few ways. You could aggregate all the forces on all objects prior to looping through the objects integrating the equations of motion, but this would require looping through all the objects twice. Alternatively, since you're looping through the object list in order to integrate each object's equations of motion, you can simply aggregate each object's forces during the integration step. A complication arises when object pairs apply forces to each other. A linear spring and damper, for example, connected between two objects, apply equal and opposite forces to each object. The force is a function of the relative distance between the objects (the spring component) and their relative velocity (the damping component). So, if during a given time step you aggregate the forces on one of the objects in the pair, the resulting force will be a function of the current relative position and velocities of the objects. Integrating that object will give it a new position and velocity. Then, when you get to the other object in the pair, if you recomputed the spring force, it will be a function of the new relative position and velocity between the objects that includes the new displacement and velocity of the previously updated object but the old displacement and velocity of the current object. This is inconsistent with Newton's law of equal and opposite forces. You can resolve this problem by storing the force computed for the first object and applying it to the second one without recomputing the spring force for the second object.