Physics is a hot topic in computer games. No self-respecting action game can get by without a good physics engine, and the trend has recently spread through other genres, including strategy games and puzzles. This growth has been largely fueled by middleware companies offering high-powered physics simulation. Most high-profile games now feature commercial physics engines.

But commercial packages come at a high price, and for a huge range of developers building a custom physics solution can be better, as it can be cheaper, provide more control, and be more flexible. Unfortunately, physics is a topic shrouded in mystery, mathematics, and horror stories.

When I came to build a general physics engine in 2000, I found that there was almost no good information available, almost no code to work from, and lots of contradictory advice. I struggled through and built a commercial engine, and learned a huge amount in the process. Over the last 10 years I’ve applied my own engine and other commercial physics systems to a range of real games. Almost a decade of effort and experience is contained in this book.

There are other books, websites, and articles on game physics, much of it quite excellent. But there is still almost no reliable information on building a physics engine from scratch—a complete simulation technology that can be used in game after game. This book aims to step you through the creation of a physics engine. It goes through a sample physics engine (provided on the CD), as well as giving you insight into the design decisions that were made in its construction. You can use the engine as is, use it as a base for further experimentation, or make various design decisions and create your own system under the guidance that this book provides.
Chapter 1 \textit{Introduction}

1.1 \textbf{What Is Game Physics?}

Physics is a huge discipline, and academic physics has hundreds of subfields. Each describes some aspect of the physical world, from the way light works to the nuclear reactions inside a star.

Lots of these areas of physics might be useful in games. We could use optics, for example, to simulate the way light travels and bounces, and use to make great-looking graphics. This is the way ray tracing works, and (although it is still very slow compared to other approaches) it has been used in several game titles. Although these areas are part of academic physics, they are not part of what we mean by game physics and I won’t consider them in this book.

Other bits of physics have a more tenuous connection with games. I cannot think of a use for nuclear physics simulation in a game, unless the nuclear reactions were the whole point of the game play.

When we talk about physics in a game, we really mean classical mechanics, that is, the laws that govern how large objects move under the influence of gravity and other forces. In academic physics these laws have largely been superceded by relativity and quantum mechanics. Almost all of the physics described in this book has long since stopped being an active area of research; all the results we’ll be relying on were settled before the turn of the twentieth century.

In games, classical mechanics is used to give game objects the feel of being solid things, with mass, inertia, bounce, and buoyancy.

Game physics has been around almost since the first games were written. It was first seen in the way particles move: the ballistics of bullets, sparks, fireworks, smoke, and explosions. Physics simulation has also been used to create flight simulators for nearly three decades. Next came automotive physics, with ever-increasing sophistication of tire, suspension, and engine models.

As processing power became available, we saw crates that could be moved around or stacked, and walls that could be destroyed and crumble into their constituent blocks. This is rigid-body physics, which rapidly expanded to include softer objects: clothes, flags, and rope. Then came the rise of the ragdoll: a physical simulation of the human skeleton that allows more realistic trips, falls, and death throes. And recently we’ve seen a lot of effort focused on simulating fluid flow: water, fire, and smoke.

In this book we’ll cover a representative sample of physics tasks. With a gradually more comprehensive technology suite, our physics engine will support particle effects, flight simulation, car physics, crates, destructible objects, cloth, and ragdolls, along with many other effects.

1.2 \textbf{What Is a Physics Engine?}

Although physics in games is more than 30 years old, there has been a distinct change in recent years in the way that physics is implemented. Originally, each effect was programmed for its own sake, creating a game with only the physics needed for that
1.2 What Is a Physics Engine?

If a game needed arrows to follow trajectories, then the equation of the trajectory could be programmed into the game. It would be useless for simulating anything but the trajectory of arrows, but it would be perfect for that.

This is fine for simple simulations, where the amount of code is small and the scope of the physics is quite limited. As we’ll see, a basic particle system can be programmed in only a hundred lines or so of code. But directly implementing physical behavior becomes a difficult task as the complexity increases.

In the original Half-Life game, for example, you can push crates around, but the physics code isn’t quite right, and the way crates move looks odd. The difficulty of getting physics to look good, combined with the need for almost the same effects in game after game encouraged developers to look for general solutions that could be reused.

Reusable technology needs to be quite general: a ballistics simulator that will only deal with arrows can have the behavior of arrows hard coded into it. If the same code needs to cope with bullets too, then the software needs to abstract away from particular projectiles and simulate the general physics that they all have in common. This is what we call a physics engine: a common piece of code that knows about physics in general, but isn’t programmed with the specifics of each game.

This leaves us with a gap. If we have special code for simulating an arrow, then we need nothing else in order to simulate an arrow. If we have a general physics engine for simulating any projectile, and we want to simulate an arrow, we also need to tell the engine the characteristics of the thing we are simulating. We need the physical properties of arrows, or bullets, or crates, and so on.

This is an important distinction. The physics engine is basically a big calculator: it does the mathematics needed to simulate physics. But it doesn’t know what needs to be simulated. In addition to the engine we also need game-specific data that represents the objects in our level.

Although we’ll look at the kind of data we need throughout this book, I won’t focus on how the data gets into the game. In a commercial game, there will likely be some kind of level-editing tool that allows level designers to place crates, flags, ragdolls, or aeroplanes to set their weight, the way they move through the air, their buoyancy, and so on. For a physics engine driving a flight simulator, the data may have to be acquired from real aircraft capabilities. For simpler games, it may be hardcoded somewhere in the source code.

The physics engine we’ll be developing throughout this book needs gradually more and more data to drive it. I’ll cover in depth what kind of data this is, and reasonable values it can take, but for our purposes we will assume this data can be provided to the engine. It is beyond the scope of the book to consider the tool chain that you will use to author these properties for the specific objects in your game.

1.2.1 Advantages of a Physics Engine

There are two compelling advantages for using a physics engine in your games. First, there is the time savings. If you intend to use physics effects in more than one game
(and you’ll probably be using them in most of your games from now on), then putting the effort into creating a physics engine now pays off when you can simply import it into each new project. A lightweight, general-purpose physics system, of the kind we develop in this book, doesn’t have to be difficult to program either. A couple of thousand lines of code will set you up for most of the game effects you need.

The second reason is quality. You will most likely be including more and more physical effects in your game as time goes on. You could implement each of these as you need it, such as building a cloth simulator for capes and flags, and a water simulator for floating boxes, and a separate particle engine. Each might work perfectly, but you would have a very hard time combining their effects. When the character with a flowing cloak comes to stand in the water, how will her clothes behave? If they keep blowing in the wind even when underwater, then the illusion is spoiled.

A physics engine provides you with the ability to have effects interact in believable ways. Remember the moveable crates in Half-Life 1? They formed the basis of only one or two puzzles in the game. When it came to Half-Life 2, crate physics was replaced by a full physics engine. This opens up all kinds of new opportunities. The pieces of a shattered crate float on water, objects can be stacked and used as moveable shields, and so on.

It’s not easy to create a physics engine to cope with water, wind, and clothes, but it’s much easier than trying to take three separate ad-hoc chunks of code and make them look good together in all situations.

### 1.2.2 Weaknesses of a Physics Engine

This isn’t to say that a physics engine is a panacea. There are reasons that you might not want to use a full physics engine in your game.

The most common reason is speed. A general-purpose physics engine is quite processor-intensive. Because it has to be general, it can make no assumptions about the kinds of objects it is simulating. When you are working with a very simple game environment, this generality can mean wasted processing power. This isn’t an issue on modern consoles or the PC, but on handheld devices such as phones and PDAs, it can be significant. You could create a pool game using a full physics engine on a PC, but the same game on a mobile phone would run faster with some specialized pool physics.

The need to provide the engine with data can also be a serious issue. In a game that I worked on we needed no physics other than flags waving in the wind. We could have used a commercial physics engine (one was available to the developer), but the developer would need to have calculated the properties of each flag, its mass, springiness, and so on. This data would then need to be fed into the physics engine to get it to simulate the flags.

There was no suitable level-design tool that could be easily extended to provide this data, so instead we created a special bit of code just for flag simulation, the characteristics of flags were hardcoded in the software, and the designer needed to do
nothing special to support it. We avoided using a physics engine because special-case code was more convenient.

A final reason to avoid physics engines is scope. If you are a one-person hobbyist working on your game in the evenings, then developing a complete physics solution might take your time away from improving other aspects of your game, such as the graphics or game play. Or worse, it might distract you from finishing, releasing, and promoting your game. On the other hand, even amateur games need to compete with commercial titles for attention, and top-quality physics is a must for a top-quality title of any kind.

1.3 Approaches to Physics Engines

There are several different approaches to building a physics engine. From the very simple (and wrong) to the cutting-edge physics engines of top middleware companies. Creating a usable engine means balancing the complexity of the programming task with the sophistication of the effects you need to simulate.

There are a few broad distinctions we can make to categorize different approaches.

1.3.1 Types of Objects

The first distinction is between engines that simulate full rigid bodies or so-called “mass aggregate” engines. Rigid-body engines treat objects as a whole, and work out the way they move and rotate. A crate is a single object, and can be simulated as a whole. Mass aggregate engines treat objects as if they were made up of lots of little masses. A box might be simulated as if it were made up of eight masses, one at each corner, connected by rods.

Mass aggregate engines are easier to program because they don’t need to understand rotations. A large amount of effort is needed to support rotations, and it forms a sizable chunk of this book. Mass aggregate engines treat each mass as if it were located at a single point, and the equations of motion can be expressed purely in terms of linear motion. The whole object rotates naturally as a result of the connections between masses.

Because it is very difficult to make things truly rigid in a physics engine, it is difficult to make really firm objects in a mass aggregate system. Our eight-mass crate will have a certain degree of flex in it. To avoid this being visible to the player, extra code is needed to reconstruct the rigid box from the slightly springy set of masses. While the basic mass aggregate system is very simple to program, these extra checks and corrections are more hit and miss, and very quickly the engine becomes a mess of fixes and ugly code.

Fortunately, we can extend a mass aggregate engine into a full rigid-body system, simply by adding rotations. In this book, we will develop a mass aggregate physics engine on the way to a full rigid-body physics engine. Because we are heading for a
more robust engine, I won’t spend the time creating the correction code for springy aggregates.

1.3.2 Contact Resolution

The second distinction involves the way in which touching objects are processed. As we’ll see in this book, a lot of the difficulty in writing a rigid-body physics engine is simulating contacts—locations where two objects touch or are connected. This includes objects resting on the floor, objects connected together, and, to some extent, collisions.

One approach is to handle these contacts one by one, making sure each works well on its own. This is called the “iterative” approach and it has the advantage of speed. Each contact is fast to resolve, and with only a few tens of contacts, the whole set can be resolved quickly. It has the downside that one contact can affect another, and sometimes these interactions can be significant. This is the easiest approach to implement, and can form the basics of more complex methods. It is the technique we will use in the engine in this book.

A more physically realistic way is to calculate the exact interaction between different contacts and calculate an overall set of effects to apply to all objects at the same time. This is called a “Jacobian-based” approach, but it is very time consuming. The mathematics needed to process the Jacobian is very complex, and solving the equations can involve millions of calculations. In some cases there is simply no valid answer and the developer needs to add special code to fall back on when the equations can’t be solved. Most physics middleware packages and several open-source physics engines use this approach, and each has its own techniques for solving the equations and dealing with inconsistencies.

A third option is to calculate a set of equations based on the contacts and constraints between objects. Rather than use Newton’s laws of motion, we can create our own set of laws for the specific configuration of objects we are dealing with. These equations will change from frame to frame, and most of the effort for the physics engine goes into creating them (even though solving them is no picnic either). This is called a “reduced coordinate” approach. Some physics systems have been created with this approach, and it is the most common one used in engineering software to achieve really accurate simulation. Unfortunately, it is very slow, and isn’t very useful in games, where speed and believability are more important than accuracy.

We’ll return to the Jacobian and reduced coordinate approaches in Chapter 20, after we’ve looked at the physics involved in the first approach.

1.3.3 Impulses and Forces

The third distinction is in how the engine actually resolves contacts. This takes a little explaining, so bear with me.

1. The “Jacobian” itself is a way of mathematically representing the effects of one contact on another.
When a book rests on a table, the table is pushing the book upwards with a force equal to the gravity pulling it down. If there were no force from the table to the book, then the book would sink into the table. This force is constantly pushing up on the book as long as the book is there. The speed of the book doesn’t change.

Contrast this with the way a ball bounces on the ground. The ball collides with the ground, and the ground pushes back on the ball, accelerating the ball upward until it bounces back off the floor with an upward velocity. This change in velocity is caused by a force, but the force acts for such a small fraction of a second that it is easier to think of it as simply a change in velocity. This is called an impulse.

Some physics engines use forces for resting contacts and impulses for collisions. This is relatively complex, because it involves treating forces and impulses differently. More commonly physics engines treat everything as a force: impulses are simply forces acting over a very small space of time. This is a “force-based” physics engine and it works in the way the real world does. Unfortunately, the mathematics of forces are more difficult than the mathematics of impulses. Engines that are force-based tend to employ a Jacobian or reduced coordinate approach.

Other engines use impulses for everything: the book on the table is kept there by lots of miniature collisions, rather than a constant force. This is, not surprisingly, called an “impulse-based” physics engine. Each frame of the game, the book receives a little collision that keeps it on the surface of the table until the next frame. If the frame rate slows down dramatically, things lying on surfaces can appear to vibrate. Under most circumstances, however, it is indistinguishable from a forced-based approach. This is the approach we will use in this book, as it is easy to implement, and has the advantage of being very flexible and adaptable. It has been used in several middleware packages, in a large number of the in-house physics systems I have seen, and has been proven in many commercial titles.

### 1.3.4 What We’re Building

In this book I will cover in depth the creation of a rigid-body, iterative, impulse-based physics engine that I call Cyclone. The engine has been written specifically for this book, although it is broadly based on a commercial physics engine I was involved with writing a few years ago.

I am confident that the impulsive-based approach is best for developing a simple, robust, and understandable engine for a wide range of game styles, and for using as a basis for adding more complex and exotic features. It can be used as a foundation for experimenting with other approaches: I’ve used the skeleton structure to implement a Jacobian force-based engine, for example.

As we move through the book, I will give pointers for various approaches, and Chapter 20 will provide some background to techniques for extending the engine to take advantage of more complex simulation algorithms. While we won’t cover other approaches in the same depth, the engine is an excellent starting point for any kind of game physics. You will need to understand the content of this book to be able to create a more exotic system.
Chapter 1  Introduction

1.4  The Mathematics of Physics Engines

Creating a physics engine involves a lot of mathematics. If you're the kind of person who feels nervous working with math, then you may find some bits hard going. I've tried throughout the book to step through the mathematical background slowly, but unfortunately there's no way to avoid the mathematics entirely.

If you have difficulty following the mathematics, don't worry: you can still use the accompanying source code for the corresponding section. While it is better to understand all of the engine in case you need to tweak or modify it, you can still implement and use it quite successfully without such understanding.

As a quick reference, the mathematical equations and formulas in the book are brought together in Appendix C, for easy location when programming.

If you are an experienced game developer, then chances are you will know a fair amount of 3D mathematics, including vectors, matrices, and linear algebra. If you are relatively new to games, then these topics may be beyond your comfort zone.

In this book I will assume you know some mathematics, and I will cover the rest. If I assume something that you aren't comfortable with, then it would be worthwhile to find a reference book, or look for a web tutorial before proceeding, so that you can stay with the flow of the text.

1.4.1  The Math You Need to Know

I'm going to assume that every potential physics developer knows some mathematics.

The most important thing to be comfortable with is algebraic notation. I will introduce new concepts directly in notation, and if you flick through this book you will see many formulas written into the text.

I'll assume you are happy to read an expression such as:

\[ x = \frac{4}{t} \sin \theta^2 \]

and are able to understand that \(x\), \(t\), and \(\theta\) are variables, and how to combine them to get a result.

I will also assume you know some basic algebra. You should be able to understand that, if the formula above is correct, then

\[ t = \frac{4}{x} \sin \theta^2 \]

These kinds of algebraic manipulations will pop up throughout the book without explanation.

Finally, I'll assume you are familiar with trigonometry and coordinate geometry: sines, cosines, tangents, their relationship to the right-angled triangles, and to two-dimensional geometry in general.
In particular, you should know that if we have the triangle shown in Figure 1.1, then these formulas hold:

\[ b = a \sin \theta \]
\[ c = a \cos \theta \]
\[ b = c \tan \theta \]

Especially when \( a \) is of length 1, we will use these results tens of times in the book without further discussion.

1.4.2 The Math We’ll Review

Because the experience of developers varies so much, I will not assume you are familiar with three-dimensional mathematics to the same extent. This isn’t taught in high schools and is often quite specialized to computer graphics. If you are a long-standing game developer, then you will be able to skip through these reviews as they arise.

We will cover the way that vectors work in the next chapter, including the way a three-dimensional coordinate system relates to the 2D mathematics of high school geometry. I will review the way that vectors can be combined, including the scalar and vector product, and their relationship to positions and directions in three dimensions.

We will also review matrices. Matrices are used both to transform vectors, representing movement in space, or to change other matrices from one set of coordinates into another. We will also see matrices called tensors at a couple of points, which have different uses but the same structure. We will review the mathematics of matrices, including matrix multiplication, the transformation of vectors, and matrix inversion.

These topics are fundamental to any kind of 3D programming, and are used extensively in graphics development, and in many AI algorithms too. I assume that most readers will be at least a little familiar with them, and there are comprehensive books available that cover them in great depth.
1.4.3 The Math I’ll Introduce

Finally, there is a good deal of mathematics that you may not have discovered unless you have done some physics programming in the past. This is the content I’ll try not to assume you know, and cover in more depth.

At the most well-known end of the spectrum this includes quaternions, a vector-like structure that represents the orientation of an object in 3D space. We will take some time to understand why such a strange structure is needed, and how it can be manipulated, converted into a matrix, combined with other quaternions, and affected by rotations.

We will also need to cover vector calculus, or the way vectors change with time and through space. Most of the book requires only simple calculus—numerical integration and first-order differentiation. The more complex physics approaches of Chapter 20 get considerably more exotic, including both partial differentials and differential operators. Fortunately, we will have completely built the physics engine by this point, so the content is purely optional.

Finally, we will cover a few more advanced topics in matrix manipulation. In particular, resolving contacts in the engine development involves changing the coordinates of existing transform matrices. This kind of manipulation is rarely needed in graphics development, so it will be covered in some depth in the relevant section.

1.5 The Source Code in the Book

Throughout the book the source code from the Cyclone physics engine is given in the text. The complete engine is available on the accompanying website, but repeating the code in the text has allowed me to comment more fully on how it works.

The latest Cyclone source, including errata and new features, is available at its own site, http://www.procyclone.com. It is also hosted on Google’s open-source code website at: http://code.google.com/p/game-libraries/. Check the site from time to time for the latest release of the package.

In each section of the book, we will cover the mathematics or concepts needed, and then view them in practice in code. I’d encourage you to try to follow the equations or algorithms in the code, and find how it has been implemented.
I have used an object-oriented design for the source code, and always tried to err on the side of clarity. The code is contained within a *cyclone* namespace, and its layout is designed to make naming clashes unlikely.

I have used C++ throughout the code. This is still the most common programming language used for game development worldwide. I’m aware, however, that over the last few years, C++ has become less exclusive. With the advent of a wide range of gaming platforms and coding environments, it is no longer C++ or nothing. I know of readers of the first edition who implemented the engine in languages ranging from Microsoft’s C# and Apple’s Objective-C, through Adobe’s Actionscript for Flash, to high-level dynamic languages such as Javascript and Python. I have therefore revised and extended Chapter 19 in this edition, which discusses implementation for a range of languages.

There are many parts of the engine that can be optimized, or rewritten, to take advantage of mathematics hardware on consoles, graphics cards, and some PC processors. If you need to eke out every ounce of speed from the engine, you will find that you need to optimize some of the code to make it less clear and more efficient. Chances are, however, it will be perfectly usable as is. It has a strong similarity to code I have used in real game development projects, that has proved to be easily fast enough to cope with reasonably complex physics tasks.

There are a number of demonstration programs in the source code, and I will use them as case studies in the course of the book. The demonstrations were created to show off physics rather than graphics, so I’ve tried to use the simplest graphics output possible. The source code is based on the GLUT toolkit, which wraps OpenGL in a platform-independent way. The graphics tend to be as simple as possible, as in calling GLUT’s built-in commands for drawing cubes, spheres, and other primitives. This selection doesn’t betray any bias on my part and you should be able to transfer the physics so that it works with whatever rendering platform you are using.

The license for your use of the source code is the MIT license. It is designed to allow it to be used in your own projects, but it is not copyright-free. Please read through the software license accompanying the source code for more details.

It is my hope that although the source code will provide a foundation, you’ll implement your own physics system as we go. I make decisions throughout this book about my implementation, and chances are that you’ll make different decisions at least some of the time. My aim is to give you enough information to understand the decision, and to go a different route if you want to.

### 1.6 How the Book Is Structured

We will build our physics engine in stages, starting with the simplest engine that is useful and adding new functionality until we have a system capable of running the physics in your game.
The book is split into six sections:

- In Particle Physics, we look at building our initial physics engine, including the basic vector mathematics and the laws of motion for particles.
- The Mass Aggregate Physics section turns the particle physics engine into one capable of simulating any kind of object by connecting masses together with springs and rods.
- In Rigid-Body Physics, rotation and the added complexity of rotational forces are introduced. Overall, the physics engine we end up with is less powerful than the mass aggregate system we started with, but is useful in its own right and as a basis for the final stage.
- The Collision Detection section takes a detour from building engines to look at how the collisions and contacts are generated. A basic collision detection system is built, allowing us to look at general techniques.
- The Contact Physics section is the final stage of our engine, adding collisions and resting contacts to the engine and allowing us to apply the result to almost any game.
- Finally, in Horizons we look beyond the engine we have built. In Chapter 20 we examine means of extending the engine to take advantage of other approaches, without providing the detailed step-by-step source code to do so.

As we begin each part, the content will be quite theoretical, and it can be sometimes difficult to immediately see the kinds of physical effects that the technology supports. At the end of each part, there is a chapter with the payoff, showing ways in which our new functionality may be used in a game. As we go through the book we start with engines controlling fireworks and bullets, and end up with ragdolls and cars.

1.6.1 Exercises and Projects

At the end of most chapters, particularly those that introduce new technical content, there is a set of exercises. These are designed to solidify the new concepts introduced in that chapter and to allow you to think about other implications of what you’ve learned. The chapter exercises are typically quite narrow and focused.

At the end of each part of the book, I’ve included some additional exercises and project suggestions. These are designed to be broader and bring together the content of that part into something practical. I’ve split this content into further exercises, mini-projects, and game projects.

The mini-projects are typically implementation challenges. They are suitable as an exercise over a week or two, or as a homework assignment in a course on game physics. I’ve tried to indicate the difficulty of these projects using a three-star system. One star is a project that should be relatively simple and accessible for all. I’ve reserved one star for projects that tweak the code in fairly predictable or minor ways, or that merely apply it to a new scenario. Three stars indicates a problem that requires novel thinking,
or modifications to the core algorithms or mathematics beyond what is introduced in the chapter. It should be suitable for readers who really want to stretch themselves.

The game projects give suggestions for how to use the physics engine developed so far in a complete game, showing off the physics as much as possible. These projects will take longer, and can be used as an end-of-semester project, or as inspiration for a complete game.

For all the projects in this book there is no right or wrong answer: you decide how much or how little you want to develop the physics. I hope they will provide a framework for applying the content of the book. One of the challenges of learning a whole new area, like game physics, is seeing how to apply it as you go, without having to learn everything there is to know before you start.