PART 1

Theory, Design, and Preparation

This book is divided into two distinct parts: the first focusing primarily (though not exclusively) on the theoretical, design, and preparatory issues surrounding engine development, and the second focusing mainly on the implementation of a game engine in the C++ language, along with a selection of other tools and libraries. That is not to say that the first part does not feature implementation work, nor that the second does not feature design work, but simply that each part has a key focus to which other subject matter in that section is subordinate.

The first part of this book lays the foundations for the implementation work that is to follow in Part 2. Specifically, it centers on theory, design, and preparation work. The theory comes in Chapter 1 where questions such as the following are asked and addressed: "What is a game engine?" "How does an engine differ from a video game as a whole?" "Do all games need an engine?" "What is the essence of a game engine, and what are its essential building blocks?" These are all essential questions in that they must be answered in order for engine development to be possible. After all, how can a developer build a solid and reliable engine when they are not entirely sure what an engine is? The first chapter also touches on issues of design, planning, and time management. It enumerates the core components of an engine one by one and plans for their development. Some of these issues might at first seem tedious or boring in comparison to the "real" coding and implementation work, but as I hope these pages (as well as experience) will show, the importance of solid planning cannot be underestimated when developing engines.

The second chapter moves away from the issues of theory and design and into the world of preparation—that is, preparation for engine development. Specifically, that chapter highlights a series of C++ IDEs that can be used for coding engines, and also a series of core and cross-platform libraries common in the world of game development. It examines some important and useful classes from the STL library and explains the purpose and usefulness of the game loop messaging structure for games, game engines, and real-time applications. In short, Chapters 1 and 2 are intended to contain enough of the foundations necessary to get started with game engine development. The design and theory does not end with those chapters, but it is only after those chapters that implementation can begin. Thus, it is now time to turn to the first chapter and to explore the theory behind game engines.

1 Game Engines—Details and Design

"Engine [noun] A machine with moving parts that converts power into motion"—*Oxford English Dictionary*

"Engine [noun] A software system, not a complete program, responsible for a technical task" —Wiktionary

Overview

After completing this chapter, you should:

- Understand what is meant by the term "game engine"
- Appreciate the distinction between game engine, game content, and development tools
- Recognize the benefits of dividing engines into manager components
- Appreciate the most common managers
- Understand render managers, audio managers, and resource managers
- Understand the benefits of engines to game development

1.1 Game Engines

The contemporary video game market is filled with an almost countless number of both free and commercial games, and almost all of these games are powered by an engine: a game engine. To say that a game is "powered" by an engine is not to speak literally, like saying that a computer is powered by electricity or a car by fuel. It is to speak metaphorically; it is to say that a game (any game) *depends* on its engine and that without it the game could neither be developed nor executed. However, it does not mean that a game is equal to its engine, that each are one and the same thing. For this reason, by creating an engine and only an engine a developer does not thereby create a complete game. This is because an engine is only a part of a game, just as the heart is only a part of a body. But like the heart, the engine is an essential part. Thus, a game is greater than its engine, but the engine must be in place before the game can

be executed and played on the system of an end user, whether that system be a PC, Mac, Wii™, hand-held device, or some other platform. In addition, the engine must be in place before much of the game development work can occur, since the engine often acts as a pivot around which the team workflow revolves. The engine, for example, often influences the structure and arrangement of the graphics files produced by artist, and the file formats and timing of audio files produced by sound artists and musicians. It follows then that the engine is the heart (or *core*, or *kernel*) of both the game and its development, and thus the building of an engine by the programmers of the team is one of the first steps taken in the development of a game.

It is not enough, however, for a complete understanding of a game engine to say that it is the heart of a game, because this definition is insubstantial and vague insofar as it says nothing specific about the qualities of an engine. Such a definition mentions only how an engine *relates* to a game, and says nothing particular about what an engine *does* or *is*, or *is not*. At this point, however, a problem arises for both the author and the reader. The problem is that there exists no uncontroversial or unchallenged industry standard meaning for the term "game engine." It does not have a precise meaning like "graphics" or "sound," or a mathematical definition. Rather, the term "game engine" is deployed loosely in many contexts by many developers to convey a general sense or idea rather than a precise meaning about which there can be no negotiation. Having said this though, it is not an entirely subjective term either, like the term "beautiful," whose meaning depends almost entirely on individuals with regard to their specific tastes and preferences. "Game engine" then is not simply a catch-all buzzword used to mean whatever any individual chooses at any one moment. True, as a term its meaning is not precise, but still it conveys a sense and idea that is held in common between game developers. This sense and idea is the primary subject of this chapter, along with some general but important guidelines for the design of game engines. To help convey the idea and importance of the game engine, this chapter turns now to explain historically how the concept of an engine developed in order to serve the specific needs of game developers looking to increase their productivity in a competitive market.

1.2 Game Engine as an Idea

Let us imagine that during the mid-1980s a small independent video game studio—let's call it Studio X—opened its doors with much celebration as it publicly announced its latest platformer game to be released by the end of that same year. (Platformer games typically are those in which the gamer navigates a character through a level filled with enemies by running and jumping across both static and moving platforms and ledges. Examples of such games include Sonic the Hedgehog™, Super Mario Bros.®, and LittleBigPlanet.™ See Figure 1.1). Once the end of that year arrived, the developer released their game with much success and to the delight of happy gamers worldwide.

FIGURE 1.1 Screenshot from a platformer game. Image from Darwin the Monkey, courtesy of Rock Solid Games.

Their game outsold a rival platformer game developed by competitive Studio Y. In fact, their game became so successful that a sequel was demanded and subsequently planned by Studio X. Studio X, however, began development of the sequel with a time management problem. It was their aim from the outset to capitalize on the success of the first game by releasing the sequel soon after and ahead of another competitive release planned by their rival. But during the development of the first game, Studio X had not considered the possibility that their work back then might help them in the future with the work for their sequels. Instead, they had considered each game a self-contained entity, and their development as separate and distinct processes. For them, each game was to be built from the ground upward, and no work from one game could possibly be of use for the next since each of them were to be released and sold as separate games. Consequently, the developers of Studio X began work for their sequel as they had for their first game. The result was that the second was in development for no less time than the first, and thus it was released into stores later than expected, accompanied by much disappointment for fans of the series and by much loss of sales for Studio X, which had on this occasion been upstaged by their faster rival Studio Y.

Studio Y produced their sequel in half the time it took to make the first. They did not achieve this by cutting corners; each member of the team worked just as hard as they always did. Nor did they achieve this by reducing the length or the content of the sequel; their sequel was bigger and better than the previous game. They achieved this by realizing when developing their first game that many properties and features common to all games and to all platformer games can be extracted from their particular contexts and given an abstract form. An abstract form is one that has lost all reference to concrete circumstances and applies not only to one game but to all. The developers realized, for example, that *all* platformer games featured a player character subject to the laws of physics, who after jumping in the air must eventually fall to the ground because of gravity. This rule would apply just as much to the sequel as to the first game. Similarly, both games would need to play sound and music in connection with events in the game, and though the sounds themselves may vary between games, the mechanisms by which the sounds are played to the speakers need not change since they do not depend on the content of the sounds; a single audio framework plays many sounds. Thinking in this way, Studio Y recognized not only some but almost all the generalizable components of a game and integrated them into a *single system* that could be *reused to produce many different games*. They knew that game content (graphics and sound) would probably need to be created on a game-by-game basis, since games look and sound different, but they realized also that there existed a framework (or infrastructure) supporting this content, and that it was this that was mostly generalizable. This framework or system is known as a game engine.

1.3 Game Engines and Game Development

The idea that the game engine is the heart or core containing almost all the generalizable components that can be found in a game implies that there are other parts of a game and of game development that do not belong to the engine component on account of their specific, nongeneralizable nature. These parts include at least game content and game development tools.

1.3.1 Game Content

Game content refers to all the particulars featured in any one game. Graphics and sound are the most notable members of this category. Any two platformer games, for example, might share many features in common such as enemies that run, jump, and shoot at the player character. Yet these enemies differ between games in both appearance and sound according to the work produced by the art and sound departments of a team, respectively. The graphics and sound, therefore, are specific to a game and are thus part of what makes a game unique. The engine is distinguished from content because it is concerned less

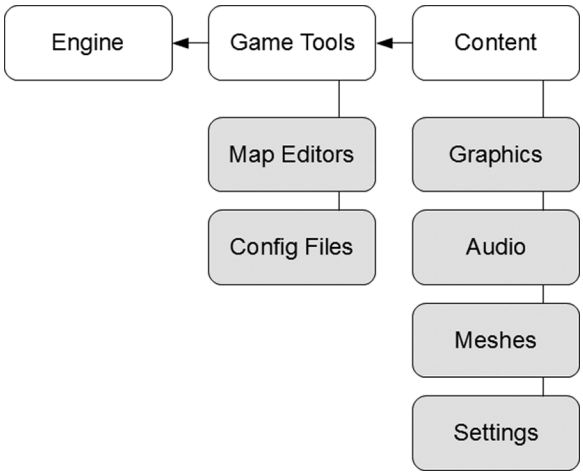

FIGURE 1.2 The game engine and its relationship to other parts of game development.

with the features unique to any one game than with the features common to all games. For this reason, game content and content creation tools are not the main concern of this book, though they are important for game development generally (see Figure 1.2).

1.3.2 Game Development Tools

The distinction between a game engine on the one hand and the game content on the other raises a problem for developers regarding the relationship between the two. The game engine represents everything that is abstract and applicable to all or most games, and the content that is specific to either a single game or a few games. On its own, the engine is only an active collection of rules, forms, and algorithms, and the content is only an inactive collection of images and sounds. Since each of these two parts focuses on a different aspect of the game, a bridge or protocol must be formed between them to allow the engine access to the content so as to form a complete game. It is not enough for a game developer to create the engine and content in isolation from each other, because then the engine has nothing to act upon and the content nothing to bring it to life. For example, an artist might use graphics software to produce some images of weapons and objects for a platformer game, but still the engine is required to direct and move those objects in accordance with gravity or inertia or any other laws appropriate for the game and circumstances. Images and sounds and other content do not act of their own accord; an engine is required to act on them. To build the bridge between game engine and game content during the development process, game development tools are required. These tools take the form of level editors, map generators, mesh exporters, path planners, and others. Their primary purpose is to connect the inactive game content to the active

powers of the engine. A level editor, for example, is usually a custom-made GUI tool used by the developer to produce a text or binary file. This file is fed into the engine and defines a specific arrangement of much of the game graphics (trees, walls, doors, and floors, etc.) into an integrated layout of levels and environments with which players will contend as they play the game. Game tools occupy an important place in the relationship between engine and content, and are considered in further detail later in this book. For the purposes of this chapter, it is enough to state that game tools—whatever their form—represent the link between engine and content.

1.4 Game Engines in Detail

Thus far the game engine has been conceived as a single and enclosed entity representing everything that is generalizable about games and as having a relationship to two other entities—game content and game development tools. However, at present no specific mention has been made of the architecture of an engine, nor of those features of all games that are sufficiently generalizable to warrant their inclusion in an engine. There has been the occasional allusion to formulas, laws, and physics to convey a sense of some of the things that are likely to belong, but no detailed list of features or explanation of their significance. This section considers some of the features of games most likely to be found in a contemporary game engine and how those features are built into the engine so as to work in unison according to an optimal design. However, this examination by no means represents an exhaustive catalog of *all* features found in *all* engines. This is partly because there are too many features and engines to examine in one book and partly because there is no one consensus among all game developers regarding what is and is not an appropriate feature for an engine. One developer might propose an architecture and a set of features almost wholly at odds with those proposed by another, yet neither of them can be said to be entirely wrong in their choices. It is for this reason that any one engine cannot be said to be entirely better or worse than another, but only more or less appropriate for a specific purpose. To some degree, the range and kind of features a developer chooses to put into an engine reflects their professional experience, design preferences, and business intentions. Unsurprisingly, there is great variation among the many engines in circulation today.

1.4.1 The Architecture and Features of a Typical Engine

Given what this section has already said about variation among engines in the industry, it might seem something of a contradiction to talk of a typical game engine. Indeed, no engine can strictly be said to be typical where there are no formal or even informal standards governing engine design. However, the term "typical" is used here to refer to a basic engine design that offers enough general components and features to allow the

creation of a wide range of games, including platformers, first-person shooters, sports sims, and RPGs (role playing games). These features and components are listed below and explained briefly, and later chapters of this book will tackle the implementation of these items. That is, they will explain how to create these features and components using a range of development tools common in the contemporary games industry.

Architecturally speaking, a typical game engine is divided by a programmer into many components called *managers* or *subsystems*, and each manager is responsible for a range of distinct features related to one another by kind. The physics manager, for example, is responsible for making sure that the laws of physics are applied to the objects of the game, ensuring that gravity pulls airborne objects downward to a ground surface and that collisions between objects are detected to prevent any solid objects from moving through walls or other solids when they are not supposed to. Likewise, an error manager is trusted with the specific duty of catching run-time errors and exceptions when they occur, and with subsequently responding to them appropriately, either by logging their occurrence in a local text file or by cleanly terminating the application. This division of engine work by kind into corresponding manager components, as opposed to sharing the work in one manager, is useful for a programmer because it allows them to segment and sort their source code across many files according to the number of managers supported. By arranging the physics code into one group of files, the error handling code into another, and all other sets of features according to their kind, the programmer can separate and manage vast sections of code according to their functional status. If a run-time error occurs in the game that is related to physics, then the programmer can know without searching the engine source code that debugging is to begin in the physics manager and nowhere else. The remainder of this section details some manager components commonly found in game engines. Before continuing, please see Exercise 1.1, then consider Figure 1.3.

> **NOTE**. In terms of object-oriented programming, a manager is often implemented as a singleton class, and it exposes its features through the class methods and properties. A singleton class is a class that allows one and only one instance of itself to be instantiated at any one time. Thus, by implementing each manager of an engine as a singleton class, the programmer can be certain that there can be no more than one instance of each manager in memory at run time.

EXERCISE 1.1

List at least three other managers that you think would be suitable for almost any game engine, and why. State at least two reasons for each answer. Answers to this question are to be found throughout subsequent sections of this chapter.

Resource Manager

The term "resources" (or "assets") is used by developers in the games industry to refer collectively to all the digital materials used by a game. Many nondigital resources may indeed be expended by developers in the production of a game—from blood and sweat to money and tears—but these are not what is meant by a game developer when they speak of resources in a technical sense. Here, "resources" refer more narrowly to two kinds of digital materials essential for games: media data and behavioral data. Media data refers to *all* the graphics, sound, animations, and other digital media that is to be featured in a game. Their purpose ultimately is to define how a game looks and sounds at run time. In contrast, behavioral data defines how a game is to be *behave at run time*, such as whether it should run in full-screen or windowed mode, or run at one resolution rather than another, or use rather than avoid subtitles. Typically, all resources—both media data and behavioral data—are encoded into files that are loaded by the engine at run time as appropriate. Media resources such as graphics are encoded into image files in formats such as PNG, JPG, and TGA, audio into files such as OGG, WAV, and MP3, and animation data into custom-made formats or video formats such as AVI and MPG. Behavioral data usually is in the form of text, sometimes in standard text format, but more often in the format of XML or CFG, and sometimes in the format of a scripting language.

To emphasize the importance of resources generally for video games, it is helpful to consider an imaginary game and the extent of its dependency on resource files. For example, a first-person shooter game, such as Half-Life® or Unreal® Tournament, is one in which the player controls the actions of a character and views the game environment as seen from their eyes. Throughout the game the player is called upon for various reasons to shoot enemies and objects and to avoid dangers. Such a game as this might depend on a wide range of resources for its existence, including all of those mentioned above. For graphics, it will require at least the images necessary to display the environment and characters, from the leaf images used for the trees and shrubbery that are scattered around the game arena to leather images used to texture the clothing of all the characters found there. For sound, it will likely require upbeat background music tracks to be played to enhance the intensity of the atmosphere, and the sound of laser beams or gunfire to be played when weapons are fired, and many others. Finally, for config files it will require a settings file to define the resolution and subtitle modes to be used by the game on execution and a level data file to define the layout of both the game environment and the objects in that environment such as the positions of walls and doors, ceilings and floors, and power-ups and bonuses. In sum, the typical video game and the game engine find themselves dependent for their existence on potentially many megabytes or gigabytes worth of resources of varying kinds. Any particular game will inevitably be dependent on a particular set of resources whose content varies according to the game, but any game is dependent on some set of resources regardless of

their content. Thus, as resources apply to games generally they are therefore pertinent to game engines, and the sheer number and variety of resources, compounded with the extent of the dependency of the game on those resources, introduces for the developer a management problem. That is, it signals a *need* for a manager component in the form of a resource manager to govern the relationship between the engine and the resources. It is the duty of the resource manager therefore to: (1) identify and distinguish between all the resources available to the game, (2) both load *and* unload the resources to and from their files and in and out of memory, and (3) ensure that no more than one instance of each resource exists in memory at any one time. Resource managers and their implementation are considered later in this book.

Render Manager

It is generally accepted that almost all games feature graphics, either 3D, 2D, or a mixture of both. It has been stated already that it is the role of the resource manager to load graphics from files on a storage device and into memory ready for use as appropriate, but this process alone is not enough for the graphics resources themselves to be displayed to the gamer on-screen. For this to occur, a second process must access the loaded graphics resource in memory and then draw it to the screen via the graphics hardware. For example, having loaded a rectangle of pixels from an image file via the resource manager, the engine must then decide how those pixels are plotted to the screen where they will be seen. Should the pixels be drawn as they appear in the file? Or should they be animated? And where on the screen should they appear—at the top left corner, at the bottom right, or elsewhere? Should the image be reversed, tiled, or repeated across the screen? These and many other questions relate to the drawing of pixels to the screen. This process is achieved by way of a graphics rendering library such as OpenGL® or DirectX or SDL, and any one of these in combination with a series of other functions and tools a developer may create from the rendering infrastructure of the game engine, or the render manager. It is the job of the render manager to (1) efficiently communicate between the engine and the graphics hardware installed on the end user system, (2) render image resources from memory to the screen, and (3) set game resolution. Render managers and their implementation are considered later in this book.

Input Manager

Games in the contemporary market differ almost as much in their support for input devices as they do in their use of resources. The term "input device" is used by developers to signify any unit of hardware the player may use to communicate their intentions to the engine for the purposes of the game. Most PC games expect to receive input via keyboard and mouse devices, console games via remote controllers such as game pads, joysticks, and dance mats, and portable games via the device keypad. The range

of input methods accepted by any game and the extent to which each is supported differs between games and platforms. However, across almost all games and platforms there is a *general* need for a mechanism both to receive and to respond to user input, regardless of the specificity of the device or of the input provided and regardless of the particular response appropriate. Thus, in the context of game engines a developmental need arises for an input manager component. The purpose of this component is to: (1) read user input at run time from the entire range of devices accepted by the game and (2) encode that input into a single, device-independent form. That is, to generate from the input a raw interpretation that does not depend for its being understood by the developer or the engine on knowledge of any specifics regarding input hardware. Input managers and their implementation are considered later in this book.

Audio Manager

What is true for the render manager regarding graphics resources and their rendering to their screen is generally true for the audio manager regarding audio resources and their playback on the system speakers. Though the content of audio differs according to particular games, almost all games demonstrate a generalizable need to play audio. Audio here means both music and sound. The former refers to audio usually longer than 1 minute in duration and intended to be played on a loop, and as the background for any other audio that may play. The latter refers to short sounds (sound effects or SFX) such as door knocks, gunshots, and footsteps that are likely to be played in response to particular events in the game, and are less than 1 minute in duration. It is the role of the resource manager to load audio from files to memory, but it is the role of the audio manager to accept those resources and to play them to the speakers as appropriate for the game. The audio manager is also responsible for: (1) communicating between the game and audio hardware, (2) setting the overall game volume levels, and (3) applying effects to sound such as fade effects, pan effects, and echo effects. Audio managers and their implementation are considered later in this book.

Error Manager

The esteemed computer scientist Edsger W. Dijkstra once suggested that "Testing shows the presence, not the absence of bugs." This quote often serves to remind developers of the limits of their debugging tools and of their ability to claim certainty regarding the absence of bugs in their software. It serves to emphasize that even when the debugging and testing of an application has uncovered no bugs a developer is still not in a position to claim that their software is bug free. This is because saying, "There are no bugs," is not the same as saying, "No bugs were found." Debugging and testing might uncover no bugs in an application, but that application might not have undergone all possible tests in all possible scenarios, and thus the developer is not in a position to know for certain that all bugs in their software were found and eliminated

based on their tests. On this basis, the most a developer is entitled to claim is that no bugs were found in their software, and this limitation applies as much to the engine developer as to any other developer. For this reason, the developer of game engines should accept the possibility that bugs might exist in their engine and in their game no matter how thorough they consider their testing methods to be. If it were possible for a developer to find and eliminate bugs in their software and to know at the same time with complete certainty that no other bugs existed, then perhaps there would be good reason to build game engines that were error free, that is, an engine in which the occurrence of an error was an impossibility. Such an error-free engine would make the developer's life easier insofar as they would have no need to code an error manager to detect and report errors, because errors could not happen. But since the author of this book knows of no such means of achieving certainty regarding the absence of bugs in software, the possibility of error holds for each and every engine developed. Hence, a game engine needs an error manager. The purpose of the error manager is to: (1) detect or "catch" run-time exceptions in a game, (2) handle those exceptions gracefully if and when it encounters them to avoid sudden and shocking crashes to the desktop by displaying a message box before exiting to alert the user to the occurrence of an error, (3) log the occurrence of an error in a human readable text file report that can be sent by the user to the developer to aid the latter in repairing the problem, and (4) identify and mark each error with a unique identification number that can be printed in an error message dialog or in the error log report. This allows the developer to isolate each error and its possible causes. Error managers and their implementation are considered later in this book.

Scene Manager

In theater or film, a "scene" refers to the stage or place where actors and props come together in action. Much of this meaning applies to the concept of a scene in games. The general outline of a game engine as presented so far in this chapter has conceived of a resource manager for loading resources from disk and into memory, a render manager for drawing graphics resources to the screen via graphics hardware, an audio manager for playing audio resources to the speakers via sound hardware, and an input manager for reading user input from input devices. These components are essential for almost all games to the extent that a game could not exist in a playable form without them, but even together they are not enough to create a complete game. It is not enough, for example, to load a graphics resource from a file only to display it anywhere and anyhow on the screen, without reason; nor is it enough to load an audio resource to likewise play it anyhow and meaninglessly on the speakers. Between the loading of resources from files and their presentation in the game there exists an intervening layer of management, of composition. This layer decides *how* resources are associated to one another and used so as to give them meaning for the player in relation to the actions and events of the game with the

passing of time. Consider the famous puzzle game of blocks called Tetris®. In Tetris, the player controls the movement of a series of falling blocks arranged in sequences of four that drop one after another from the top of the screen to the bottom across a game board. The aim is to manipulate each arrangement as it falls by moving it left or right or by rotating it 90 degrees so that as the arrangement reaches the bottom it forms a complete horizontal line of blocks in combination with the other arrangements that fell previously. The line that is formed should contain no gaps if it is to be valid, and valid lines disappear, leaving vacant space for new blocks. As time progresses in the game, the tempo increases, meaning the player encounters every new arrangement of blocks faster than the one before. Consequently, they are forced to think faster as to how to arrange the blocks that fall. The game is lost if a new block falls and finds no vacant space on the board in which it can be held. This game has a precise set of rules governing how the resources of the game relate both to each other and to the input provided by the player. The game does not play sounds arbitrarily or display graphics anywhere on-screen without reason; rather, the graphics resources are used to represent the falling blocks of the board and are positioned and deployed on-screen in combination with the sound according to the rules of the game. In this way, the resources of the game are managed into a meaningful scene, just as a puppet-master deploys his puppets on stage not randomly and meaninglessly according to his fancies, but coherently and sensibly according to the logic of the story to be performed for the audience. The purpose of the scene manager then is to synthesize (or coordinate) the resources of the engine according to the logic of the game, to tie together the work of many managers. Its duty is in part to put on the show for the player. To do this, the scene manager must perform many tasks, some of which include: (1) communicating between many managers such as the render manager, audio manager, resource manager, and others, (2) keeping track of time for the purposes of firing events and coordinating motion on the objects of the game, and (3) enumerating the objects of the scene to allow the developer to iterate and access each of them independently. Scene managers and their implementation are considered later in this book.

Physics Manager

It was mentioned earlier in this chapter that the physics manager is a component of the engine dedicated to applying the laws of physics to the objects of the game, or more accurately to applying forces and torques to objects of the scene. This physics manager is responsible for, among other things, applying the effects of gravity and inertia to game objects. The former ensures that specified airborne objects—such as chairs, apples, and people but not airplanes, fairies, and dragons—are pulled downward to a ground surface. The latter ensures that moving objects such as cars and runaway mine carts do not come unrealistically to complete and immediate stops whenever they cease to accelerate, but gradually reduce their speed over time toward a stopped state to imitate the real-world resistance of mass to changes in its state of motion. At this

point, however, it is important to note a distinction between the physics manager and the scene manager. Given what has been said so far of the duties of both with regard to their role as controllers of the behavior of game objects, it might at first seem that the two managers should be merged into one or that one makes the other redundant. If the purpose of the physics manager is to control at run time the behavior of objects and their relationships in the scene according to the laws of physics, then what place can there be for the scene manager as a controller of behavior? This question draws attention to the distinction between game logic (or game rules) on the one hand and physics on the other, the former being the responsibility of the scene manager. The scene manager, for example, does not apply gravity or inertia to objects, nor does it govern their behavior according to any physical laws; that duty falls to the physics manager. Rather, it governs scene objects according to the rules of the game as they are determined by the developers thinking outside the laws of physics. The rules of the game define the unwritten contract between the gamer and the game, and they are outside the laws of physics in the sense that the laws of physics could alter while the rules of the game remained unchanged. They refer to the terms of play and stipulate the conditions and circumstances that constitute a win for the player and those that constitute a loss, and further stipulate the range and kinds of actions a player may pursue to achieve a win. For example, there might be a set of rules governing a platformer game that specify that a loss is incurred by the player whenever their health drops below zero as the result of attack from an enemy or of damage from environmental hazards, and that a win is incurred whenever they reach the end of a level and defeat in combat the end-of-level boss. Thus, the objects and relationships of a scene in a game are governed by at least two sets of independent laws, those of physics and those of the game. The enforcement of the latter is one of the duties of the scene manager, and the enforcement of the former is the sole duty of the physics manager. Physics managers and physics handling is considered later in this book via a look at the Bullet physics library.

Scripting Manager

A game engine is typically built by programmers into a standalone executable file or a DLL (dynamic link library) using a compiled language such as C++ or C#. Using a compiled language to build an engine into machine code form often improves its run-time performance but poses a number of challenges to developers seeking to customize it according to their needs after it is compiled. It has been stated already that one means by which a general engine is customized for a specific game is game tools. These tools are used by developers to produce a variety of text and configuration files detailing the content and behavior specific to a game, and are fed into the engine to customize it accordingly. By accepting many of its instructions through external and independent game files, the compiled engine remains open to the developer in the

sense that its behavior can be changed *without having to be recompiled*. This is useful if only because an engine that can be customized without having to be recompiled does not depend on its original creators and their knowledge of its internal mechanics for its use. That is, an open engine can be adapted and customized by any who learn the protocol between the engine and its files. For example, an artist can change the graphics used by the engine simply by substituting a new set of graphics files for the previous set, and a programmer or gamer can change the run-time resolution of the engine by editing the settings of a resolution configuration file before the engine is executed. Thus, the primary reason for an engine's dependency on external files is to produce in it an openness not only to the creators of the engine but also to other developers and even to some gamers; to produce a flexible capacity to be easily customized or *scripted* for specific games without having to be recompiled. Thus, an engine is said to be *scriptable* if it has the general ability to adjust its behavior according to a set of instructions in a file. Those instructions may take many forms, and engines differ in the extent of their support for any specific instruction set. Some engines support instructions written in plain English or in XML, and others support those written in established scripting languages and frameworks such as Lua and Python™ to allow for greater control. The choice of language to be supported by any particular engine often reflects the popularity and currency of the language as well as the professional preferences and intentions of the developers, but whichever language is chosen there is a general need to *manage* the support of that language and its interaction with the engine. This need therefore creates a demand for a scripting manager, and the job of this manager is to: (1) expose a subset of classes and functions of the engine and allow access to them through the scripting language and (2) read and process instructions in a file and respond accordingly. Scripting managers and details surrounding scripting generally are discussed later in this book when considering game tools.

1.5 Game Engines—Necessary Conditions

Games are said by many developers to be powered by engines, and by this they mean that an engine is a necessary condition for a game, that a game cannot run without an engine. True, any specific game is more than an engine because it contains unique content or dressing such as graphics and sound, but the engine represents the integrated framework on which this dressing hangs. Conceiving of the engine as a framework or a necessary condition for a game is useful for identifying whether or not any library or software is or is not a game engine. Presented with any library or software, such as DirectX or OpenGL, the developer can proceed to ask, "Can I make a game with this and *only* this?"; if the answer to that question is in the affirmative, then it either *is* or *can be used as* a game engine. It is not enough to show that only in combination with other tools could it make games. The software as it is must be capable of producing

games on its own merits to be considered a game engine. Thus, OpenGL is a *dedicated* graphics API but not a game engine. Its purpose is to render graphics to the screen via the hardware. No doubt many engines have made use of this library for that specific purpose, but despite this, OpenGL as a library is not responsible for other features of the engine that are equally essential for creating a game—from the playing of game audio to the handling of game logic. In diagram form, the engine exists on a level above dedicated libraries; the engine represents the integrated core of a game that delegates essential duties via manager components to dedicated libraries. Consider Figure 1.3.

It has been established that a game depends on an engine for its existence, and that the essence of an engine (that which makes it what it is) consists of its ability to make games. This ability and its scope are determined in large part by the manager components from which the engine is made, some of which were discussed in the previous section. The render manager determines the feature set for graphics rendering, the audio manager for audio playback, and each other manager for its corresponding duties. Thus, the manager components represent the chief executors of the engine and are the primary means by which engine work is performed. For this reason, the managers are to the engine as the engine is to the game; that is, they are the core. A game depends on its engine and the engine on its managers. Having determined this,

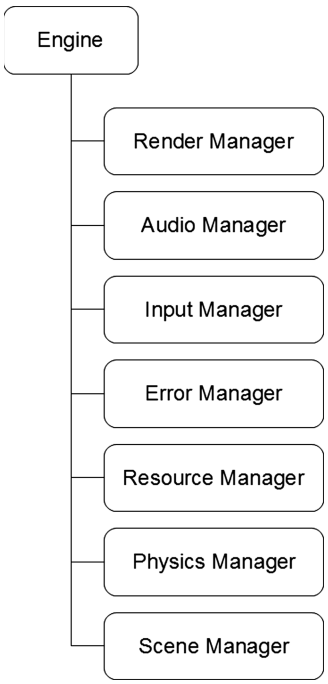

FIGURE 1.3 Common set of managers for a game engine.

the question then arises as to where the line is drawn between that which is an engine and that which is not in terms of the number and type of managers supported. Is it possible, for example, to conceive of a minimum collection of managers that together warrant the title of "engine" and from which no manager could be removed without compromising that title? In other words, what is the simplest engine possible? Such an engine might be suitable for creating only the simplest of games, but even so can it be defined? If so, which managers would belong to that group of essential managers?

It is helpful for a developer to think about this question of essential managers because the answer (if there is a definitive one) points to only those managers that are essential for all games and which mark the starting point for the development of almost all engines. Some might be tempted to argue that *all* engine managers listed in this chapter thus far are equally essential for all games. They might argue that many games from platformers to sports sims require a physics manager, render manager, audio manager, and many other managers, and that for this reason these managers are essential in any engine that is intended to create those games. But this response misses the point of the exercise. It is no doubt the case that all managers are important for specific games, but it does not follow from this that all managers are equally important for all games. A physics manager is unquestionably important to a game dependent on physical reactions, such as a flight simulation or a car racing game, but not so important for a card matching game or a game of Tetris or any other game that does not depend on physical laws. Thus, many games exist that do not require an engine with a physics manager. However, it seems unlikely that even the simplest game could exist without an input manager to receive and process user input from a game controller. The task of this section is one of reduction: to create a list of only those managers essential to all games under all circumstances, or essential to the simplest possible game. Having created such a list and confirmed those managers as the foundation of almost all engines, the developer can then recognize the implementation of those managers as the starting point or the first steps on a long road of engine development. In short, with the exception of design, engine development begins with the implementation of its essential managers. So what are the essential managers? Unfortunately, there is no definitive answer to this question, or at least none of which the author is aware. But for the purposes of this book, concerned as it is with the creation of a

EXERCISE 1.2

List three managers you think should be included in the essential managers group, and state two reasons for each manager to justify your choice. Answers can be found in the following section. Your answers might differ from mine, but it is important to have reasons for your choices and to appreciate the reasons for mine.

general engine suitable for many game genres, it is nonetheless possible to build a rough argument as to which managers ought to be included.

1.5.1 Game Engines—Essential Managers

Any manager component may belong to either one of two groups: essential and nonessential. This status relates to its significance for the simplest of games. If the simplest of games cannot exist without a given manager, then that manager is an essential manager. The simplest of engines contains no more than the essential managers, and thus the implementation of these managers represents the starting point of engine development after the stage of design. The following table lists each manager stated earlier in this chapter that is essential, and for each states two reasons why it is an essential manager.

Essential Manager	Reasons
Render Manager Status: Essential	1. Game must *show on-screen* any essential error messages and information. 2. Almost all games feature graphics.
Resource Manager Status: Essential	1. The resource manager processes resources, and the render manager depends on resources. The render manager is essential, and therefore the resource manager is essential. 2. The game must keep a list of all resources for the purposes of memory management and tidy resource unloading.
Scene Manager Status: Essential	1. The scene manager is required to organize the contents of a scene. Without this manager, resources cannot be placed in meaningful arrangements. 2. Scene managers are essential for monitoring game logic.
Input Manager Status: Essential	1. A game depends in part on the input received from the user, and an engine processes user input through the input manager. Therefore, the input manager is an essential manager. 2. User input is necessary to terminate the execution of a game once the gamer has finished playing.
Error Manager Status: Essential	1. Developers cannot be sure their engine is free of all errors through standard debugging techniques. Therefore, an error manager is required to handle errors as they occur. 2. The error manager might need to notify users of errors.

TABLE 1.1 Essential Managers

1.6 Principles of Engine Design—RAMS

The essential managers are the render manager, input manager, scene manager, error manager, and resource manager. Having identified these as the essential managers of an engine, it might appear that there is little else for an engine developer to do but to

get started with the implementation of those managers. However, there are a variety of approaches a programmer might take toward coding their implementation. Some might choose to implement each manager as a unique class with its own methods and properties; since the managers must interact with one another, the developer might choose to expose the interface of each manager to every other and to have much *direct* interaction between them. This solution offers the benefit of performance insofar as each manager has a direct channel of communication to every other, but introduces the disadvantage that each manager must have at least some knowledge concerning the implementation of other managers. Another solution might be to choose an indirect route: to hide each manager class from every other and to code an additional messenger class to pass data between the managers as each goes about its work. These two scenarios represent only two possible responses to the one problem of communication between managers, and this design problem is only one among many. The solution to that problem, and those to many others, will influence the form the managers take when implemented. Again, there is no clear-cut right or wrong solution to many, if any, of the design problems an engine developer encounters, but this is not the same as saying that all solutions are equally suitable or that all approaches are just as good. A person might offer 10 different answers to a sum and all of them might be wrong, but some are closer to being right than others. There is simply not enough space in this book or perhaps in any book to address all the possible design problems that might arise and their corresponding solutions, even if all such problems and solutions could be listed. So instead of attempting to list specific design problems and to provide specific solutions, this book details a number of general key design principles or ideas that can be found to underpin the design and implementation of many successful engines used in the contemporary games industry. These ideas detail some sound design advice concerning game engines. Having understood these principles or guidelines, the reader can then proceed to apply them to their specific case and to tailor their solutions accordingly. They can do this by considering all the solutions before them and establishing which of them most coincides with the design principles. The four design principles listed here can be remembered using the acronym RAMS. Many successful engines are designed according to the principles of recyclability, abstractness, modularity, and simplicity. Each of these is now considered in more detail.

1.6.1 Recyclability

Materials and processes that can be reused are considered recyclable, and the more frequently and widely they can be reused the more recyclable they are. For game engines, recyclability often relates to efficiency. An engine that consumes half the resources of another engine to perform the same tasks with equal or greater reliability and efficacy is generally considered to be the more efficient of the two. Efficiency refers to the relationship between the resources available and the performance and

reliability that result from the processing of those resources. A process is efficient to the extent that it achieves success and reliability from a given set of resources, and efficiency is improved when success and reliability are increased from the same resources or from fewer resources. For this reason, efficiency is improved through recycling whenever a single set of resources can be used and reused for two or more processes successfully and reliably. This is because performance is increased without a corresponding aggregate increase in resources. To illustrate, consider recycling in the context of a resource manager for an engine that is being used to power an imaginary platformer game. The purpose of the resource manager is to load game resources from files on disk or over the Internet and into memory, either system memory or other hardware memory. Resources include graphics, audio, and many other file types. During the playing of the game, the player enters a level and encounters five enemy alien clones, all of which look alike. At this point, the resource manager is called upon to load the graphics resources to be used as a representation of the alien creatures. The resource manager can achieve this using one of at least two solutions: It can either load five copies of the same image, one for each alien, or it can load one copy of the same image and share this among all like aliens. The second solution is the more efficient because it acknowledges the principle of recyclability by using a single resource for many instances. In short, the principle of recyclability advises the developer to consider each and every resource as precious and scarce, and to design the engine so as to extract the greatest use from the fewest resources possible.

1.6.2 Abstractness

Abstraction is about producing generalizations from particular instances, and is a matter of degree. A complete abstraction can be identified by its entirely general nature, for it makes no reference to particular cases. The idea of a chair, for example, is an abstraction because it refers to chairs in general and not to any particular chair. Earlier in this chapter, a number of engine managers were identified; these managers represent abstractions also because a general demand for them was identified after having considered particular details. From those particular details, a general and nonparticular idea of a manager was constructed, and from the idea of a manager, various *kinds of managers* were further created. The manager components and even the concept of an engine itself are all abstractions because they are ideas that do not depend on any particular instances but apply to all games as such. The principle of abstraction is useful for engine developers because by designing the components of an engine to be abstract rather than particular, the developer increases their field of use or their versatility. Their versatility is proportional to their degree of abstraction. A render manager, for example, can be used by any and all games, because rendering is an essential process for all games. In contrast, a text render manager is less abstract than a render manager and is consequently more limited in its scope, assuming its purpose is

to render text and only text. In short, the principle of abstraction encourages the engine developer to think in the abstract. It encourages them to start by identifying the many particular and contingent needs of one case and then to proceed from this by building abstractions that serve not only the needs of that one particular case but all the like needs of all cases. Thus, the building of abstractions serves to increase the versatility of the engine, and the greater the degree of abstraction, the greater the versatility.

1.6.3 Modularity

The idea of a general manager component is the result of abstraction. Having identified this idea, it can be developed further through the principle of modularity. The idea that a manager should be an independent and functional unit responsible for a range of related tasks is often what results when the engine is thought of in terms of modularity. The principle of modularity begins by considering an entity as a working whole (such as a game engine) and then proceeds to subdivide that entity into constituent pieces or modules, each according to their purpose or function as it relates to the whole. In object-oriented programming, each module is likely to correspond to a class. By the principle of modularity, each module is seen as both independent and exchangeable: independent in the sense that it is distinguished from other modules by its function, for no two modules belonging to the same whole should share the same function, and exchangeable in the sense that it could be replaced without damage to the whole only by another module that serves the same function. For example, the principle of modularity when applied to the idea of a game engine will likely subdivide the engine into manager components, each manager satisfying a unique function. The purpose of the render manager is to render, and the audio manager to play audio, and the error manager to log errors. None of these modules can be removed without reducing the whole, and no module can be replaced safely except by another that performs the same function. Two modules that perform the same function need not be entirely identical in every respect; their function and output might be identical, but their working methods might differ for there may be many roads to the same destination. The purpose of the render manager is to draw graphics via the hardware to the screen, and it may achieve this single end via OpenGL, SDL, DirectX, or another graphics library. A render manager that uses the DirectX library, for example, *is not* substitutable for an input manager or a resource manager since their functions differ by kind from that of a render manager, but *is* substitutable for another render manager regardless of which graphics library that manager chooses to use to fulfill its purpose. In this way, modularity allows an engine to be divided into modules, and further allows each module to hide the details and specifics of its implementation while maintain its relationship to the whole. The module does this by focusing on achieving a single purpose. In short, the principle of modularity recommends that an entity is subdivided into smaller functional units. Doing this offers the developer several design and debugging benefits: (1) It allows

them to translate complex entities into a collection of simpler ones according to the contribution each makes to the whole, and (2) it allows them to make changes to the implementation of specific modules without affecting the implementation of other modules, or the working of the whole.

1.6.4 Simplicity

The 14th-century Franciscan Friar William of Occam stated that "entities should not be multiplied unnecessarily." This principle is now known as Occam's Razor; applied to game engines it might be translated into the mantra "Keep things simple." In this case, keeping things simple refers to a process of reductionism, which refers to the process of reducing the complex into the simple. This works side by side with modularity. The principle of modularity recommends that an engine be subdivided into a range of modules each according to their unique function, and the principle of simplicity is there to remind us that the process of modularity must be performed with simplicity in mind, with the aim of reductionism. Modularity combined with simplicity suggests that an engine should not only be subdivided into modules, but should be subdivided into the *fewest* number of modules possible. This process might proceed as follows: A developer identifies each and every entity or part belonging to an engine, and then for each entity he pauses to ascertain its function. If any two or more parts share the same function or very similar functions, then these parts are candidates for amalgamation. That is, these parts should be merged together into a larger unit. Having performed this process once for each unit, the developer should repeat the procedure for the newly formed larger units, and then repeat it continually on each level upward until the process yields no more amalgamations. At this point, the developer can know that the simplest arrangement of modules has been found. In short, the principle of simplicity, when applied appropriately, ensures that an engine features no entities sharing the same purpose or very similar purposes. This is helpful for a developer because working with the simplest arrangements ensures they not duplicate work across modules and that they work only with the minimum number of modules necessary for the purpose.

EXERCISE 1.3

Answer the following questions and then compare your answers to those given.

Q1. How does the principle of simplicity differ from the principle of modularity?

A1. Modularity suggests that an entity should be subdivided into smaller functional components. Simplicity suggests that entities should not be multiplied unnecessarily. When the principle of modularity is combined with that

of simplicity, the advice is that an entity should be subdivided into the fewest number of functional components possible.

Q2. Does creating a DirectX render manager and an OpenGL render manager for the same engine violate the principle of simplicity?

A2. No. The two modules share the purpose but are implemented differently, and thus are exchangeable modules. The existence of two render managers where one uses OpenGL and another DirectX is useful for creating cross-platform engines. The Windows version might use the DirectX render manager, and the Linux® or Mac version might use the OpenGL render manager.

1.7 Engines in Practice

Engines designed according to the four principles of modularity, simplicity, recyclability, and abstractness often benefit the game developer and prove cost effective in many ways, including the following: (1) By their versatility, since they can be used for many games and for many games across many genres, (2) by their increasing the reliability of games, since a bug corrected in the engine potentially represents a fix for all games powered by that engine, and (3) by their saleability, since an engine can be licensed by its creator to one or more other developers for the purpose of powering their games. Having mentioned some of the business and developmental benefits an engine offers to a developer, this section briefly explores some of the commercial and free and open source engines in use in the contemporary games industry.

EXERCISE 1.4

Select at least two engines from the list below and visit their home pages to read their features and technical information. On the basis of this information, list for each engine chosen all the manager components you can identify.

Torque 3D and Torque 2D
Developer: GarageGames®
License: Commercial
Website: http://www.garagegames.com
Supported Platforms: PC Windows, Mac, iPhone®, Wii, Xbox® 360

Torque 3D and Torque 2D are two separate commercial game engines created and licensed by GarageGames for the creation of 3D and 2D games, respectively. The engines are not genre specific, meaning that they are intended for the creation

of games of various genres and run on several platforms. This engine has been involved in the development of at least the following games: Marble Blast, Wildlife Tycoon, ThinkTanks, and The Destiny of Zorro.

Unity Engine
Developer: Unity Technologies
License: Commercial
Website: http://unity3d.com/
Supported Platforms: PC Windows, Mac, iPhone, Wii, web deployment

The Unity Engine is a proprietary game engine designed and licensed by Unity Technologies for the creation of 3D games on a variety of platforms, including Windows and Mac. It can also be used to create web browser games. The Unity Engine has been used for the development of several games, including Open Fire and Tiki Magic Mini Golf.

Crystal Space
Developer: Crystal Space Team
License: Open source, free
Website: http://www.crystalspace3d.org
Supported Platforms: PC Windows, Linux, FreeBSD®

Crystal Space is a free, open source, and cross-platform engine developed by the Crystal Space Team that allows developers to produce 3D games on a variety of platforms. This engine powers many games, both commercial and free, including The Icelands and the free RPG PlaneShift (http://www.planeshift.it).

Game Blender
Developer: Blender Foundation
License: Open source, free
Website: http://www.gameblender.org
Supported Platforms: PC Windows, Linux, and Mac

Game Blender is a comparative newcomer among the free, open source, and cross-platform game engines designed for producing 3D games. It is associated with the Blender 3D rendering software.

ClanLib
Developer: ClanLib Team
License: Open source, free
Website: http://www.clanlib.org/
Supported Platforms: PC Windows, Linux, and Mac

ClanLib is a free, open source, and cross-platform game engine developed by the ClanLib team for creating 2D games for Windows, Linux, and Mac.

Novashell
Developer: Robinson Technologies
License: Open source, free
Website: http://www.rtsoft.com/novashell/
Supported Platforms: PC Windows, Linux, and Mac

Novashell, a free, open source, and cross-platform game engine developed by Robinson Technologies, ships with an integrated level editor for importing game art and defining game maps.

Multimedia Fusion
Developer: Clickteam
License: Commercial
Website: http://www.clickteam.com
Supported Platforms: PC Windows

Multimedia Fusion is a commercial, proprietary game engine designed and licensed by Clickteam for the creation of 2D games for Windows.

Leadwerks
Developer: Leadwerks Software
License: Commercial
Website: http://www.leadwerks.com
Supported Platforms: PC Windows

Leadwerks is a proprietary game engine designed and licensed by Leadwerks Software for the creation of 3D games on the Windows platform. It can work with a variety of programming languages including C++, C#, and BlitzMax.

1.8 Chapter Summary

This chapter sought to provide some answers to several key questions. These were: (1) what is a game engine? (2) what does a game engine do? (3) how does a game engine differ from other parts of a game and the game development process? and (4) what are the core components of a game engine? Before proceeding to the next chapter, try to provide a one-paragraph answer to each of those questions. Then check your answers by reading the appropriate chapter sections. The next chapter considers the first stage of engine implementation: the configuration of an IDE and a project in preparation for coding.

In short, this chapter has detailed the following:

- A game engine is an integrated framework of managers and components designed for powering games.
- Game engines are often designed according to the four principles of: abstractness, modularity, simplicity, and recyclability.
- Each manager of an engine is an exchangeable unit serving a unique purpose. Managers include but are not limited to render managers, audio managers, input managers, and scene managers.
- There are many third-party managers available in the contemporary games industry, some are commercial and others are free and open source. These can be used to produce games.