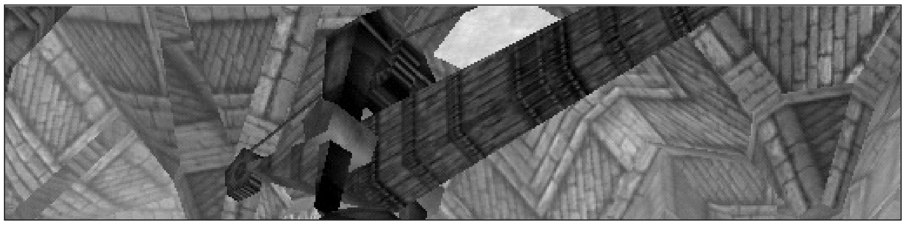
CHAPTER 1



INTRODUCTION

I have no fault to find with those who teach geometry. That science is the only one which has not produced sects; it is founded on analysis and on synthesis and on the calculus; it does not occupy itself with probable truth; moreover it has the same method in every country.

—Frederick the Great

1.1 A Brief Motivation

Computer graphics has been a popular area of computer science for the last few decades. Much of the research has been focused on obtaining physical realism in rendered images, but generating realistic images comes at a price. The algorithms tend to be computationally expensive and must be implemented on high-end, special-purpose graphics hardware affordable only by universities through research funding or by companies whose focus is computer graphics. Although computer games have also been popular for decades, for most of that time the personal computers available to the general public have not been powerful enough to produce realistic images. The game designers and programmers have had to be creative to produce immersive environments that draw the attention of the player to the details of game play and yet do not detract from the game by the low-quality graphics required for running on a low-end machine.

Times are changing. As computer technology has improved, the demand for more realistic computer games that support real-time interaction has increased. Moreover, the group of computer gamers itself has evolved from a small number of, shall we say, computer geeks to a very large segment of the population. One of the most popular, successful, and best-selling games was *Myst*, created and produced by Cyan Productions and published through Broderbund. This game and others like it showed that an entirely new market was possible—a market that included the general consumer, not just computer-savvy people. The increased demand for games and the potential size of the market has created an impetus for increased improvement in the computer technology—a not-so-vicious circle.

One result of the increased demand has been the advent of hardware-accelerated graphics cards that off-load a lot of the work a CPU normally does for software rendering. The initial cards were add-ons that handled only the 3D acceleration and ran only in full-screen mode. The 2D graphics cards were still used for the standard graphics display interface (GDI) calls. Later-generation accelerators have been designed to handle both 2D GDI and 3D acceleration within a window that is not full screen. Since triangle rasterization has been the major bottleneck in software rendering, the hardware-accelerated cards have acted as fast triangle rasterizers. As of the time of this writing, the next-generation hardware cards are being designed to off-load even more work. In particular, the cards will perform point transformations and lighting calculations in hardware.

Another result of the increased demand for games has been the evolution of the CPUs themselves to include support for operations that typically arise in game applications: fast division, fast inverse square roots (for normalizing vectors), and parallelism to help with transforming points and computing dot products. The possibilities for the evolutionary paths are endless. Many companies are now exploring new ways to use the 3D technology in applications other than games, for example, in Web commerce and in plug-ins for business applications.

And yet one more result of the increased demand is that a lot of people now want to write computer games. The Internet newsgroups related to computer graphics, computer games, and rendering application programmer interfaces (APIs) are filled with questions from eager people wanting to know how to program for games. At its highest level, developing a computer game consists of a number of factors. First and foremost (at least in my opinion) is having a good story line and good game play—without this, everything else is irrelevant. Creation of the story line and deciding what the game play should be can be categorized as *game design.* Once mapped out, artists must build the *game content,* typically through modeling packages. Interaction with the content during run time is controlled through *game artificial intelligence*, more commonly called *game AI*. Finally, programmers must create the application to load content when needed, integrate the AI to support the story line and game play, and build the *game engine* that manages the data in the world and renders it on the computer screen. The last topic is what this book is about—building a sophisticated real-time game engine. Although games certainly benefit from real-time computer graphics, the ideas in this book are equally applicable to any other area with three-dimensional data, such as scientific visualization, computer-aided design, and medical image analysis.

1.2 A Summary of the Chapters

The classical view of what a computer graphics engine does is the *rendering* of triangles (or polygons). Certainly this is a necessary component, but it is only half the story. Viewed as a black box, a renderer is a consumer-producer. It consumes triangles and produces output on a graphics raster display. As a consumer it can be fed too much data, too quickly, or it can be starved and sit idly while waiting for something to do. A front-end system is required to control the input data to the renderer; this process is called *scene graph management.* The main function of the scene graph management is to provide triangles to the renderer, but how those triangles are obtained in the first place is a key aspect of the front end. The more realistic the objects in the scene, the more complex the process of deciding which triangles are sent to the renderer. Scene graph management consists of various modules, each designed to handle a particular type of object in the world or to handle a particular type of process. The common theme in most of the modules is *geometry*.

Chapter 2 covers basic background material on geometrical methods, including matrix transformations, coordinate systems, quaternions, Euler angles, the standard three-dimensional objects that occur most frequently when dealing with bounding volumes, and a collection of distance calculation methods.

The graphics pipeline, the subject of Chapter 3, is discussed in textbooks on computer graphics to varying degrees. Some people would argue against the inclusion of some parts of this chapter, most notably the sections on rasterization, contending that hardware-accelerated graphics cards handle the rasterization for you, so why bother expounding on the topic. My argument for including these sections is twofold. First, the computer games industry has been evolving in a way that makes it difficult for the "garage shop" companies to succeed. Companies that used to focus on creating games in-house are now becoming publishers and distributors for other companies. If you have enough programmers and resources, there is a chance you can convince a publisher to support your effort. However, publishers tend to think about reaching the largest possible market and often insist that games produced by their clients run on low-end machines without accelerated graphics cards. And so the clients, interested in purchasing a third-party game engine, request that software renderers and rasterizers be included in the package. I hope this trend goes the other way, but the commercial reality is that it will not, at least in the near future. Second, hardware-accelerated cards do perform rasterization, but hardware requires drivers that implement the high-level graphics algorithms on the hardware. The cards are evolving rapidly, and the quality of the drivers is devolving at the same rate—no one wants to fix bugs in the drivers for a card that will soon be obsolete. But another reason for poor driver quality is that programming 3D hardware is a much more difficult task than programming 2D hardware. The driver writers need to understand the hardware and the graphics pipeline. This chapter may be quite useful to that group of programmers.

Chapter 4 introduces scene graph management and provides the foundation for a hierarchical organization designed to feed the renderer efficiently, whether a software or hardware renderer. The basic concepts of local and world transforms, bounding volumes for culling, render state management, and animation support are covered.

Chapters 5 and 6 discuss aspects of the intersection of objects in the world. Picking is the process of computing the intersection of a line, ray, or line segment with objects. Collision detection refers to computing intersections between planar or volumetric objects. Some people include picking as part of the definition of collision detection, but the complexity of collision systems for nonlinear objects greatly exceeds that for picking, so I have chosen to separate the two systems.

Chapters 7 through 12 cover various systems that are supported by the scene graph management system. Chapters 7 and 8, on curves and surfaces, are somewhat general, but the emphasis is on tessellation. The next-generation game consoles have powerful processors but are limited in memory and bandwidth between processors. The dynamic tessellation of surfaces is desirable since the surfaces can be modeled with a small number of control points (reducing memory usage and bandwidth requirements) and tessellated to as fine a level as the processors have cycles to spare. The emphasis will start to shift from building polygonal models to building curved surface models to support the trend in new hardware on game consoles. Chapter 9 discusses the animation of geometric data, and in particular, key frame animation, inverse kinematics, and skin-and-bones systems. Level of detail is the subject of Chapter 10, with a special focus on continuous level of detail, which supports dynamic change in the number of triangles to render based on view frustum parameters.

Chapter 11 presents an algorithm for handling terrain. Although there are other algorithms that are equally viable, I chose to focus on one in detail rather than briefly talk about many algorithms. The key ideas in implementing this terrain algorithm are applicable to implementing other algorithms. High-level sorting algorithms, including portals and binary space partitioning trees, are the topic of Chapter 12.

Chapter 13 provides a brief survey of special effects that can be used in a game engine. The list is not exhaustive, but it does give an idea of what effects are possible with not much effort.

Building a commercial game engine certainly requires understanding a lot about computer graphics, geometry, mathematics, and data structures. Just as important is properly architecting the modules so that they all integrate in an efficient manner. A game engine is a large library to which the principles of object-oriented design apply. Appendix A provides a brief review of those principles and includes a discussion on an object-oriented infrastructure that makes maintenance of the library easier down the road. These aspects of building an engine are often ignored because it is faster and easier to try to get the basic engine up and running right away. However, shortterm satisfaction will inevitably come at the price of long-term pain in maintenance. Appendix B is a summary of various numerical methods that, in my experience, are necessary to implement the modules described in Chapters 7 through 12.

1.3 Text Is Not Enough

This book is not like the academic textbooks you would find in the school bookstore or the popular computer game programming books that you see at your favorite bookseller. Academic texts on computer graphics tend to be tomes covering a large number of general topics and are designed for learning the basic concepts, not for implementing a full-blown system. Algorithmic details are modest in some books and lacking in others. The popular programming books present the basic mathematics and concepts, but in no way indicate how complex a process it is to build a good engine. The technical level in those books is simply insufficient.

A good collection of books that address more of the algorithmic issues for computer graphics is the *Graphics Gems* series (Glassner 1990; Aarvo 1991; Kirk 1992; Heckbert 1994; Paeth 1995). Although providing a decent set of algorithms, the collection consists of contributions from various people with no guidance as to how to incorporate these into a larger integrated package such as a game engine. The first real attempt at providing a comprehensive coverage of the topics required for real-time rendering is Möller and Haines (1999), which provides much more in-depth coverage about the computer graphics topics relevant to a real-time graphics engine. The excellent references provided in that book are a way to investigate the roots of many of the concepts that current-generation game engines incorporate.

But there is one last gap to fill. Textual descriptions of graphics algorithms, no matter how detailed, are difficult to translate into real working code, even for experienced programmers. Just try to implement some of the algorithms described in the ACM SIGGRAPH proceedings! Many of those articles were written after the authors had already worked out the details of the algorithms and implemented them. That process is not linear. Ideas are formulated, algorithms are designed, then implemented. When the results of the coding point out a problem with the algorithmic formulation, the ideas and algorithms are reformulated. This natural process iterates until the final results are acceptable. Written and published descriptions of the algorithms are the final summary of the final algorithm. However, taken out of context of the idea-to-code environment, they sometimes are just not enough. Because having an actual implementation to look at while attempting to learn the ideas can only accelerate the learning process, a CD-ROM containing an implementation of a game engine accompanies this book. While neither as feature complete nor as optimized as a commercial engine, the code should help in understanding the ideas and how they are implemented. Pointers to the relevant source code files that implement the ideas are given in the text.