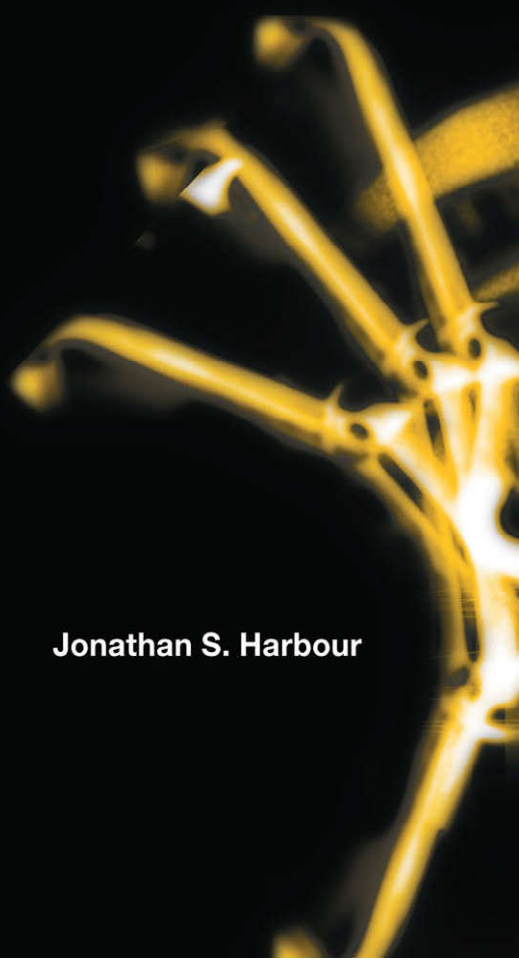
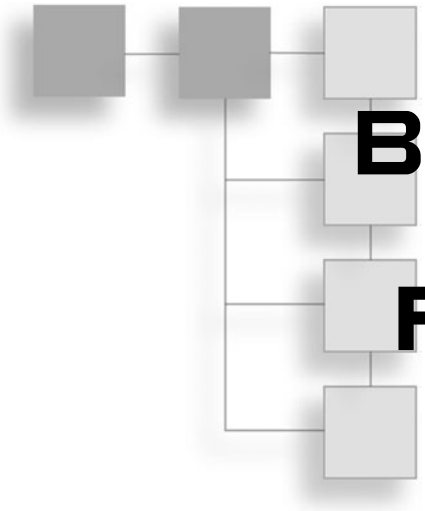
CENGAGE Learning
Professional • Technical • Reference

Beginning Game Programming

Fourth Edition



Jonathan S. Harbour



Beginning Game Programming, Fourth Edition

Jonathan S. Harbour

Cengage Learning PTR

CENGAGE Learning
Professional • Technical • Reference

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

CENGAGE Learning
Professional • Technical • Reference

Beginning Game Programming, Fourth Edition
Jonathan S. Harbour

Publisher and General Manager, Cengage Learning PTR: Stacy L. Hiquet

Associate Director of Marketing: Sarah Panella

Manager of Editorial Services: Heather Talbot

Senior Marketing Manager: Mark Hughes

Senior Product Manager: Emi Smith

Project Editor/Copy Editor: Cathleen D. Small

Technical Reviewer: David Calkins

Interior Layout Tech: MPS Limited

Cover Designer: Mike Tanamachi

Indexer: Kelly Talbot Editing Services

Proofreader: Kelly Talbot Editing Services

Library of Congress Control Number: 2014932088

ISBN-13: 978-1-305-25895-2

ISBN-10: 1-305-25895-9

eISBN-10: 1-305-25910-6

Cengage Learning PTR

20 Channel Center Street

Boston, MA 02210

USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: **international.cengage.com/region**.

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

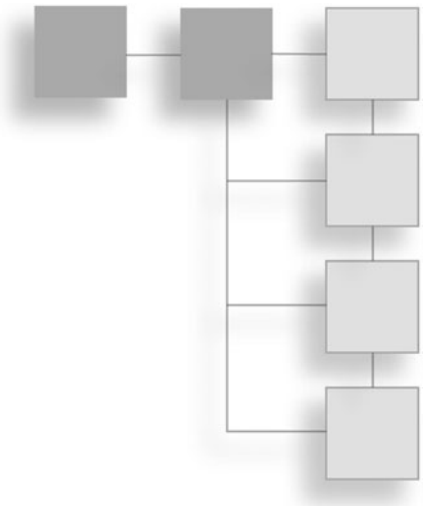
For your lifelong learning solutions, visit **cengageptr.com**.

Visit our corporate website at **cengage.com**.

Printed in the United States of America
1 2 3 4 5 6 7 16 15 14

For my mother, Vicki Myrlene Harbour

Foreword



"I want to be a game designer; how do I get a job?" This is a question I field very often when I do interviews or talk to students. I've even been accosted by the parents of an apparently gifted teenager as I left the stage with my band. My usual answer is, "So, what have you designed?" The vast majority of the time, I am given a long explanation about how the person has lots of great ideas but is in need of a team to make them a reality. My response to this is to try to explain how everyone I work with has great ideas, but only a small percentage of them are designers.

I don't mean to be harsh, but the reality is that there are no successful companies out there that will give someone off the street a development team for 18+ months and a multimillion-dollar budget without some sort of proof of concept. What sets someone like Sid Meier (legendary game designer with whom I'm honored to work at Firaxis Games) apart is his ability to take an idea and make something fun out of it.

Of course, Sid now gets large teams to do his projects, but he always starts the same way—a team of one cranking out prototypes cobbled together with whatever art and sound he can either dig up or create himself. It's these rough proofs of concept that allow people uninvolved with the creation process to immediately see the fun in a given idea, and that's what gets you a budget and a team. Every budding designer should take note and ask, "What would Sid do?"

That's when a book like this is invaluable. I became acquainted with Jonathan a few years ago when I picked up the first edition of this book at the bookstore at the Game Developer's Conference. A programmer buddy of mine helped me pick it out from among numerous similar books. He thought it was very well written and thought the emphasis on DirectX would be very applicable to what we do at Firaxis.

Another buddy mentioned that he had read Jonathan's work on programming the Game Boy Advance and was very impressed. In my opinion, they gave me great advice, and I enjoyed myself immensely while working through the book. While reading, I noticed that Jonathan was a big fan of our *Sid Meier's Civilization* series. I contacted him because I have worked on numerous *Civ* titles, and we have kept in contact ever since.

The beauty of a book like this is that it takes away all of the excuses. It provides an excellent introduction into game programming. It takes you by the hand and walks you through the seemingly complex process of writing C++ code and using DirectX. Before you know it, you'll have a fully usable framework for bringing your ideas to life. In other words, you will have all the tools you need to start making prototypes and prove that you are much more than just someone with great ideas. Believe me, taking this crucial next step will put you at the top of the heap of people looking for jobs in the industry. You will have the ability to stand out, and that's vital when so many people are clamoring for work in game development.

So, what would Sid do? Well, when he was prototyping *Sid Meier's Railroads!*, he wrote the entire prototype in C. He didn't have an artist (they were all busy on another title at the time), so he grabbed a 3D art program, made his own art, and threw it in the game—often using text labels to make sure players knew what things were in the game. He used audio files from previous Firaxis games and the Internet, and sprinkled them around to enhance the player's experience. He created something—in a fairly short amount of time—that showed our publisher and others just how much fun the game was going to be. And he did it on his own...just like the "old days" when he worked from his garage.

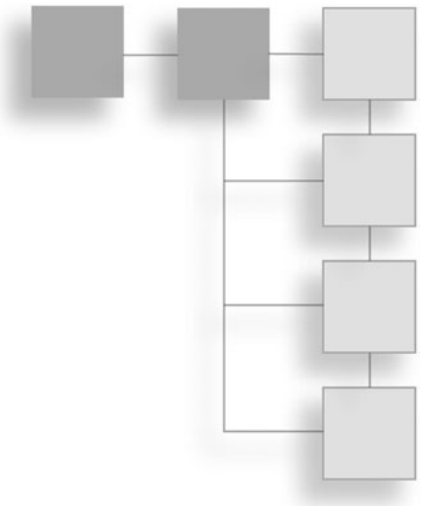
So what should you do? Well, if you want to get a job in the industry as a game designer, or even if you just want to make a cool game to teach math to your daughter, you should buy this book. Jump in and work through the exercises and develop the beginnings of your own game library—Sid has some code he's used since the Commodore 64 days. Let your imagination run wild and then find ways to translate your ideas into something people can actually play. Whatever you do, just do *something*. It's the one true way to

learn and develop as a designer, and it is your ticket to finding game-designer fulfillment and maybe even a job. And if Sid wasn't Sid and didn't already have all of those tools at his disposal, it just might be what he would do, too.

Barry E. Caudill

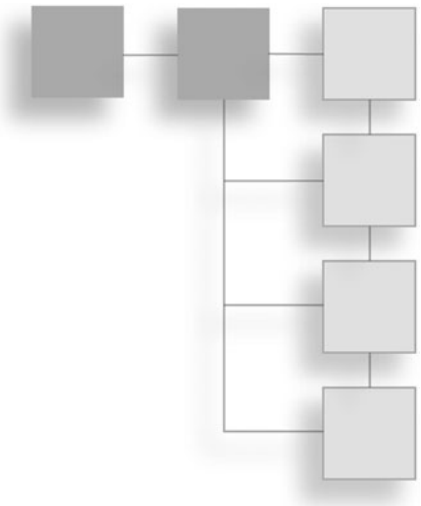
Executive Producer
Firaxis Games
2K Games
Take 2 Interactive

Acknowledgments



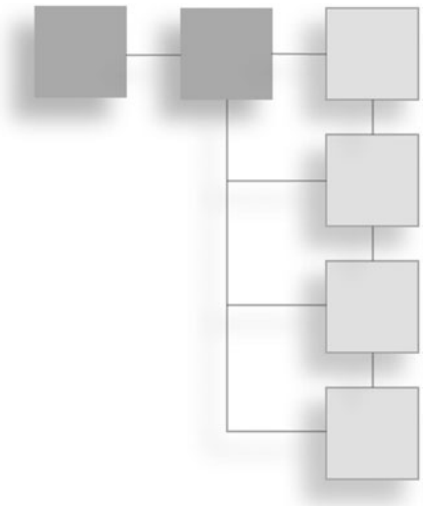
I am thankful to my family for their understanding and support while preparing this new edition. Many thanks to the editors: Cathleen Small, David Calkins, and Emi Smith. And to the readers who have offered suggestions, comments, and errata over these past 10 years—thank you!

About the Author

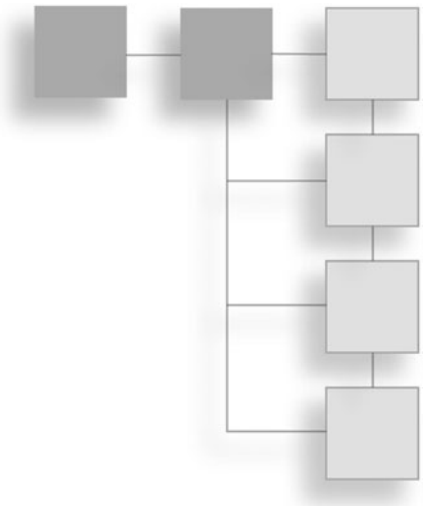


Jonathan Harbour has written 19 books, mainly covering game development for PCs, with a few covering consoles and phones. After 15 years in the sweltering heat of Phoenix, he fled to Ohio with his wife and four kids, only to face Antarctic-like subzero winters and steaming, humid summers. But at least it's not a dry heat! He can be reached at jharbour.com.

Contents



Introduction



Welcome to the adventure of game programming! I have enjoyed playing and programming games my whole life, and probably share the same enthusiasm for this subject that you do. Games were once found within the realm of Geek Land, where hardy adventurers would explore vast imaginary worlds and then struggle to create similar worlds on their own; meanwhile, out in the real world, people were living normal lives: hanging out with friends, going to the movies, cruising downtown, and playing MMOGs.

Why did we choose to miss out on all that fun? Because we thought it was more fun to stare at pixels on the screen? Precisely! But one man's pixel is another man's fantasy world or outer-space adventure. And the earliest games in "gaming" were little more than globs of pixels being shuffled around on the screen. Our imaginations filled in more details than we often realized when we played the primitive games of the past.

So, what's your passion? Or rather, what's your favorite type of game? Is it a classic arcade shoot-'em-up, a fantasy adventure, a real-time strategy game, a role-playing game, a sports-related game? I'd like to challenge you to design a game *in your mind* while reading this book, and imagine how you might go about creating that game as you delve into each chapter.

This book was not written to reminisce about light subjects like game design, with a few patchy code listings and directions on where to go next. I really take the subject quite seriously and prefer to give you a sense of completion upon finishing the last chapter. This is a self-contained book to a certain degree, in that what you will learn is applicable toward your own early game projects. What you will learn here will allow you to write a complete

game with enough quality that you may feel confident to share it with others (assuming your artwork is decent).

This book will teach you how to write DirectX code in the C++ language using Visual Studio 2013. Game programming is a challenging subject that is not just difficult to master; it is difficult just to get started. This book takes away the mystery of game programming using the tools of the trade. You will learn how to harness the power of DirectX to render 2D and 3D graphics.

You will learn how to write a simple Windows program. From there, you will learn about the key DirectX components: rendering, audio, input, fonts, and sprites. You will learn how to make use of the DirectX components while studying code that is easy to understand at a pace that will not leave you behind. Along the way, you will put all of the new information gleaned from each chapter into a game library that you can reuse for future game projects. After you have learned all that you need to know to write a simple game, you will see how to create a side-scrolling shoot-'em-up game!

Tenth Anniversary!

This new fourth edition marks the tenth anniversary since *Beginning Game Programming* was first released, way back in 2004! Keeping a book on this subject viable for so long has been hard work! This edition is leaner and meaner than preceding editions, with more emphasis on game-play topics at the beginner level.

The first edition came out in 2004 during a transition period for DirectX, which was quickly evolving from 9.0b to 9.0c—which has remained the mainstay since. This edition covered 3D modeling to a limited degree, showing via tutorial how to create a 3D car using free 3D modeling software, and then how to load and render it as a model file. But the emphasis remained primarily on sprite programming using Visual Studio 2003.

The second edition came out in 2006 to address quite a few changes to DirectX that made it difficult to compile the original sources, such as changes to the DirectSound files. Featuring support for the new Visual Studio 2005, this edition became an academic favorite for several years.

The third edition came out in 2009 and was a massive rewrite with an update to Visual Studio 2008. Every chapter was affected. Many were combined and reorganized to be leaner and more focused. Support was added for Xbox 360 controllers via XInput. This edition would continue to sell for five years!

Academic Adoption

The chapter structure and content remains largely the same as that found in the third edition, in order to maintain existing academic support. New details have been added to support Visual Studio 2013, along with a new configuration tutorial in Appendix A, "Configuring Visual Studio 2013." All figures throughout the book have been reshot for this new edition using Visual Studio 2013. Since most of the core source code remains unchanged, existing exams and lectures based on the book will remain usable. The final game in the last chapter has been updated but not dramatically changed.

What Will You Learn?

My philosophy for game development is neither limited nor out of reach for the average programmer. I want to really get down to business early on and not have to explain every function call in the standard C++ library. Game programming is not something that you just pick up after reading a single book. Although this book has everything you need to write simple 2D and 3D games, no single book can claim to cover everything, because game development is a complex subject.

I am confident that you will manage to follow along and grasp the concepts in this book just fine without a C++ primer, but a primer will give you a very good advantage before getting into Windows and DirectX. This book does not teach the C++ language; it jumps right into DirectX quickly, followed by a new subject in each chapter, so you will want to have a working knowledge of C++.

This book was written in a progressive style that is meant to challenge you at every step, and relies on repetition rather than memorization. I don't cover a difficult subject just once and expect you to know it from that point on. Instead, I present similar code sections in each program so you'll get the hang of it over time.

You will learn to use the DirectX SDK to make a game in the final chapter. You will dive into Direct3D headfirst and learn about surfaces, textures, models, fonts, and sprites (with animation). Since this book is dedicated to teaching the basics of game programming, it will cover a lot of subjects very quickly, so you'll need to be on your toes! Each chapter builds on the one before, but each chapter covers a new subject, so if there is any one subject that you are interested in at the start, you should be able to skip ahead without feeling lost. However, the game framework built in this book does refer back to prior chapters.

Visual Studio 2013

The programs in this book were written with Microsoft Visual Studio 2013. The complete source code projects can be downloaded from the Cengage website (www.cengageptr.com/downloads) or from the author's website. The projects are in Visual Studio 2013 format, since that is the latest version at this time. You can download the free Express Edition of Visual Studio 2013 from Microsoft at http://www.visualstudio.com/en-US/products/visual-studio-express-vs. (Or, since web pages change frequently, you can perform a web search for "Visual Studio Express.")

Conventions Used in This Book

The following style is used in this book to highlight portions of text that are important. You will see such boxes here and there throughout the book.

Advice

This is what an advice pop-out looks like. Advice pop-outs provide additional information related to the text.

Book Summary

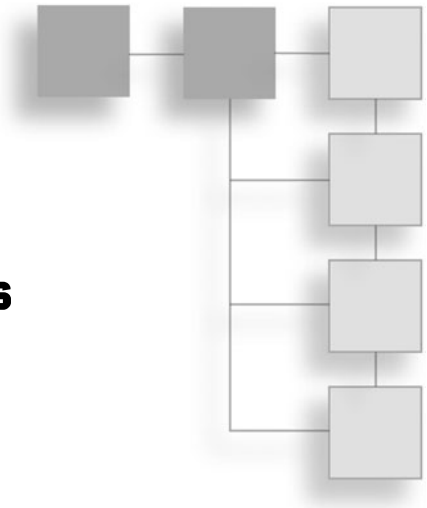
This book is divided into three parts:

- **Part I: Introduction to Windows and DirectX.** This first part provides the information you will need to get started writing Windows code and initializing Direct3D.
- **Part II: Game Programming Toolbox.** This large part covers all of the relevant components of DirectX, including images, sprites, input devices, audio, rendering, shaders, collision detection, and basic game-play mechanics.
- **Part III: Appendixes.** This part includes the two appendixes.

Companion Website Downloads

You may download the companion website files from www.cengageptr.com/downloads.

Part I



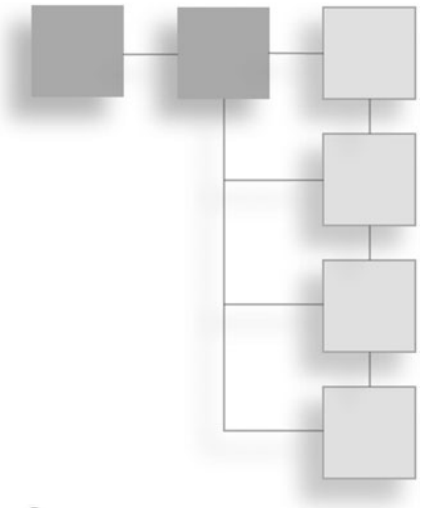
Introduction to Windows and DirectX

This first part provides an introduction to the Windows Application Programming Interface (API), which is an important foundation you'll need before learning DirectX. The first two chapters will give you an overview of how Windows works by explaining how to write a simple Windows program, how the Windows messaging system works, and how to create a message loop (which allows a program to "see" events). The third chapter gives a brief introduction to DirectX, wherein you will learn how to create a Direct3D rendering device and set up the rendering system.

- Chapter 1, "Getting Started with Windows"
- Chapter 2, "Listening to Windows Messages"
- Chapter 3, "Initializing Direct3D"

This page intentionally left blank

Chapter 1



Getting Started with Windows

© Clipart.com.

Programming a video game is one of the most enjoyable ways to learn a new language, such as C++. A video game is both a work of art and a technical achievement. Many technically impressive games are under-appreciated if they aren't considered *fun*, while less complex games might achieve worldwide fame. Regardless of your goals as a programmer, programming a game may be one of the most enjoyable hobbies you have ever pursued. Just be prepared for an equal amount of frustration and exhilaration—I hope you're ready for the adventure that is about to begin! This chapter provides the crucial information

necessary to get started writing Windows programs, leading into the next chapter, which provides an overview of Windows messaging.

In this chapter, I am going to show you what a simple Windows program looks like. This is valuable information you will need in the following three chapters, which build on this knowledge to take you into the world of DirectX. These introductory topics will come back to haunt you later on if you have not spent a little time with them, as the chapters to follow will rely on your basic understanding of how Windows works. It will be very helpful if you have some experience writing Windows programs already, but I won't assume you do. Instead, I'll just cover the basics of a Windows program—all that is necessary to start writing DirectX code.

Windows programming is actually quite fun once you get the hang of it! While some of the code might look like a foreign language, it will soon be second nature to you. If you feel a bit overwhelmed by the amount of information, don't worry too much about memorizing details, since you will be using this code over and over again. We're going to learn to write a simple Windows program first; you will create a new project in Visual Studio, type in the code, and run it. Here is what you will learn:

- How to put game programming into perspective
- How to choose the best compiler for your needs
- How to create a Win32 Application project
- How to write a simple Windows program

An Overview of Windows Programming

If you're new to Windows programming, then you're in for a treat, because Windows is a fun operating system to use for writing games. (This was not always the case, though!) First of all, there are so many great compilers and languages available for Windows. Second, it's the most popular operating system in the world, so any game you write for Windows has the potential to become quite popular. The third great thing about Windows is that we have the amazing DirectX SDK at our disposal. Not only is DirectX the most widely used game programming library in *existence*, it is also easy to get into. Now, don't misunderstand my meaning—DirectX is easy to learn, but *mastering* it is another matter. I will teach you how to use it—and wield it, so to speak—to create your own games. Mastering it will require a lot more work and knowledge than this single book provides. Studying DirectX is a very worthwhile way to spend your time, especially if you want to get up to date with the latest research in game development (since most articles and books on game development today focus on DirectX).

There are some very exciting services available for PC gamers today that make PC gaming more consistent than it may have been in the past, such as Steam by Valve Software, Inc. If you create your own compelling video game using Windows and DirectX, you have the ability to earn some money by selling it on Steam (although there is a process to follow). For more information about Steam, check out http://store.steampowered.com. Steam is very "indie friendly," meaning they support indie game developers. By adding the Steam library to your game, you can take advantage of features such as achievements. I mention this right away because it is helpful for aspiring game designers and programmers to get their games out in the market and get noticed, and this is the best way I know of today to achieve that goal.

Before you can start writing DirectX code, you will need to learn how Windows handles messages. So let's start at the beginning. What *is* Windows?

Windows is a multi-tasking, multi-threaded operating system. What this means is that Windows can run many programs at the same time, and each of those programs can have several threads running as well. As you might imagine, this operating system architecture works well with multi-core processors.

Advice

The programs featured in this book were tested on a PC with an Intel i5 quad-core CPU, 16GB DDR3 RAM, and an Nvidia GeForce 660 GTX 2GB video card. At the time of this writing, this PC is upper–middle class in performance and will run most games at the highest settings with a decent frame rate.

"Getting" Windows

Few operating systems will scale as well as Windows from one version to the next. The numerous versions of Windows that are in use—primarily Windows 7 and 8 at the time of this writing—are still so similar that programs compiled for one version of Windows will run without changes on other versions as well (such as Windows XP, which is quite old at this date). For instance, a program compiled with Microsoft Visual C++ 6.0 back in 1998 under Windows NT 4.0 or Windows 98 will still run on the latest versions of Windows. You may even have a few games in your game library that came out in the late 1990s that supported an early version of DirectX. It should come as no surprise that those older games usually still run on newer PCs. This is very helpful because we can rely on the platform running our code for years to come. This is an area that console developers have enjoyed in the past (hardware consistency) but that often has been a source of difficulty for Windows game developers. You can rely on a console system (such as the Xbox 360) remaining the same for several years, while PC specifications vary widely and

evolve very rapidly. Technical issues make it very difficult for game publishers to deal with customers using sub-par computer systems to run modern games.

So we have established that Windows programs have great longevity (also known as "shelf life" in the software industry). What can Windows really do?

Advice

Whenever I refer to "Windows" in this book, I'm including every recent version of Windows that is relevant to the topic at hand—that is, PCs and game programming. This should include all previous, current, and future versions of Windows that are compatible. For all practical purposes, this really is limited just to 32-bit programs, since we won't be covering 64-bit programming. You can assume any reference to "Windows" applies to Windows 7, 8, and may also apply to Vista and XP.

Windows programming can be simple or complex, depending on the type of program you are writing. If you have a development background with experience writing applications, then you probably have a good understanding of how complex a graphical user interface (GUI) can be to program. All it takes is a few menus, a few forms, and you will find yourself inundated with dozens (if not hundreds) of controls with which you must contend. Windows is very good as a multi-tasking operating system because it is message-driven. Object-oriented programming proponents would argue that Windows is an object-oriented operating system. In fact, it isn't. The latest version of the Windows SDK today is still similar in architecture to early versions of Windows (such as Windows 3.0). The operating system is similar to the human nervous system, although not nearly as intricate or complicated. But if you simplify the human nervous system in an abstract way, you'll see impulses moving through the neurons in the human body from the senses to the brain, and from the brain to the muscles.

Advice

Although 64-bit computing is the wave of the future, it will not be as big of an issue for programmers as the 16-to-32 bit transition was, because the processors, operating systems, and development tools have been evolving in unison, with the result being that the transition will go largely unnoticed (as it has already). Visual Studio 2013 supports 64-bit code but it's not necessary to delve into that to write a high-performance video game.

Understanding Windows Messaging

Let's talk about a common scenario to help with the analogy of comparing an operating system to the human nervous system. Suppose that some *event* is detected by nerves on

your skin. This event might be a change of temperature, or something may have touched you. If you touch your left arm with a finger of your right hand, what happens? You "feel" the touch. Why? When you touch your arm, it is not your arm that is feeling the touch, but rather, your brain. The sense of "touch" is not felt by your arm, per se; rather, your brain localizes the event so that you recognize the source of the touch. It is almost as if the neurons in your central nervous system are queried as to whether they participated in that "touch event." Your brain "sees" the neurons in the chain that relayed the touch message, so it is able to determine where the touch occurred on your arm. Now touch your arm, and move your finger back and forth on it. What do you sense is happening? It is not a constant "analog" measurement, because there are a discrete number of touch-sensitive neurons in your skin. The sense of motion is, in fact, digitally relayed to your brain. Now, you might refute my claim here by saying that the sense of pressure is analog. We are getting into some abstract ideas at this point, but I would pose that the sense of pressure is relayed to your brain in discrete increments, not as a capacitive analog signal.

How is this subject related to Windows programming? The sense of touch is very similar to the way in which Windows messaging works. An external event, like a mouse click, causes a small electrical signal to pass from the mouse to the USB port into the system bus, which might be thought of as the nervous system of the computer. From there, the signal is picked up by the operating system (Windows), and a message is generated and passed to applications that are running (like your game). Your program, then, is like a conscious mind that reacts to that "sense of touch." The subconscious mind of the computer (the operating system that handles all of the logistics of processing events) "presented" this event to your program's awareness.

Adivce

It seems that over time, our information systems (computer networks) increasingly mimic the natural world; perhaps when we have finally built the ultimate supercomputer, it will resemble a human brain?

There is yet another issue at hand. We humans have two sides to our brain. Remember my comment about technology mimicking biological brains? It is common to find six- and eight-core processors today! Multi-core systems were once exotic, high-performance niche products, but they are the norm today, even in smartphones. When the first edition of this book was published in 2004, multi-core processors were extremely rare; back then, it was common to find a motherboard with two to four processors, each with a single core.

DirectX 9 or 11?

There are two versions of DirectX still in use today. This might seem strange, but DirectX 9 was so successful that it has lasted for a decade, and a lot of game engines were designed for it. What does this mean for anyone still programming games with DirectX 9? It means that it is still a reasonable starting point and is not at all obsolete. Most professional game engines today still feature a low-end DirectX 9 version to support older computers. That is not likely to continue much longer, since even the low-end computers tend to come with very respectable hardware today.

The reason for the continued widespread support for DirectX 9 is that it supports older versions of Windows, which are still widely used. DirectX 10 and 11 games will not run on Windows XP, while DirectX 9 code will run on any modern version of Windows—which accounts for the continued popularity of DirectX 9. You can *continue* to write DirectX 9 code for Windows 7 and 8, 32-bit and 64-bit.

Multi-Tasking

The discussion of multi-tasking is not as important for programmers today as it was a few years ago, when single-core processors were the norm. It's helpful to know how things work "under the hood," so to speak, even when most users take such things for granted. Windows, like most operating systems today, uses *preemptive* multi-tasking. This means that your PC can run many programs at the same time. Windows accomplishes this by running each program for a very short amount of time, counted in microseconds, or millionths of a second. This jumping from one program to another very quickly is called *time slicing*, and Windows handles time slicing by creating a virtual address space (a small "simulated" computer) for each program in memory. Each time Windows jumps to the next program, the state of the current program is stored so that it can be brought back again when it is that program's turn to receive some processor time. This includes processor register values and any data that might be overwritten by the next process. Then, when the program comes around again in the time-slicing scheme, these values are restored into the processor registers, and program execution continues where it left off. This happens at a very low level, at the processor register level, and is handled by the Windows core.

Advice

If this sounds like a wasteful use of processor cycles, you should be aware that during those few microseconds, the processor is able to run thousands of instructions. Modern processors already run at the gigaflop level, able to easily crunch a billion math calculations in a short "time slice."

The Windows operating system might be thought of as having a central nervous system of its own—based on events. When you press a key, a message is created for that *keypress* event and circulated through the system until a program picks it up and uses it. I should

clarify a point here, as I have brought up "circulation." Windows 3.0 was a *non-preemptive* operating system that was technically just an advanced program running on top of 16-bit MS-DOS. These early versions of Windows were more like MS-DOS shells than true operating systems, and thus were not able to truly "own" the entire computer system. You could write a program for Windows 3.0 and have it completely take over the system, without freeing up any processor cycles for other programs. You could even lock up the entire operating system if you wanted to. Early Windows programs had to release control of the computer's resources in order to be "Windows Logo" certified (which was an important marketing issue at the time). Windows 95 was the first 32-bit version of Windows and was a revolutionary step forward for this operating system family in that it was a *preemptive* operating system (although it, too, still ran on top of a 32-bit MS-DOS).

What this means is that the operating system has a very low-level core that manages the computer system, and no single program can take over the system, which was the case under Windows 3.0. *Preemptive* means that the operating system can preempt the functioning of a program, causing it to pause, and the operating system can then allow the program to start running again later. When you have many programs and processes (each with one or more threads) begging for processor time, this is called a *time-slicing system*, which is how Windows works. As you might imagine, having a multi-processor system is a real advantage when you are using an operating system such as this.

A quad- or hexa-core system is a great choice for a game developer. For one thing, SMP (symmetric multiprocessing) processors usually have more internal cache memory. The more processing power the better! While you may have had to shut down some applications while playing or developing a game in the past, a modern system can easily handle many applications running at the same time while you are working on a game, and you won't notice any drag on the system. Of course, a ton of memory helps too—8GB of RAM is crucial for game development today using 64-bit Windows 7 or 8.

Multi-Threading

Multi-threading is the process of breaking up a program into multiple threads, each of which is like a separate program running. This is not the same as multi-tasking on the system level. Multi-threading is sort of like *multi-multi*-tasking, where each program has running parts of its own, and those small program fragments are oblivious of the time-slicing system performed by the operating system. As far as your main Windows program and all of its threads are concerned, they all have complete control over the system and have no "sense" that the operating system is slicing up the time allotted to each thread or process. Therefore, multi-threading means that each program is capable of delegating