Advanced 3-D Game Programming using DirectX 7.0

By Adrian Perez, with Dan Royer

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Perez, Adrian.
Advanced 3-D game programming using DirectX 7.0 / by Adrian Perez, with Dan Royer.
p. cm.
ISBN 1-55622-721-3 (pbk.)
1. Computer games—Programming. 2. DirectX. I. Royer, Dan. II. Title.

QA76.76.C672 P47 2000
794.8'15265--dc21 00-032086
CIP

2320 Los Rios Boulevard
Plano, Texas 75074

Printed in the United States of America

ISBN 1-55622-721-3
10 9 8 7 6 5 4 3 2 1
0005

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

Dedications

To my parents, Manny and Maria.

Adrian

The woman who wouldn't love me back,
The head bully,
My parents:

Well look who's laughing now!

Dan

Contents

Acknowledgments

This book couldn't have been completed without the help and guidance of a whole lot of people. I'll try to remember them all here. First, thanks go to Wes Beckwith and Jim Hill at Wordware Publishing. They were extremely forgiving of my hectic schedule, and they helped guide me to finishing this book. I also must thank Alex Dunne for letting me write an article in 1998 for *Game Developer* magazine. If I hadn't written that article, I never would have written this book.

Everything I know about the topics in this book I learned from other people. Some of these people were mentors, others were bosses, still others were professors and teachers. Some were just cool people who took the time to sit and talk with me. I can't thank them enough. Paul Heckbert, Tom Funkhouser, Eric Petajan, Charles Boyd, Mike Toelle, Kent Griffin, David Baraff, Randy Pausch, Howie Choset, Michael Abrash, Hugues Hoppe, and Mark Stehlik: You guys rock, thank you.

Thanks to Microsoft, ATI, nVidia, id Software, and Lydia Choy for helping me with some of the images used in the text.

A lot of people helped assure the technical correctness and general sanity of this text. Ian Parberry and his class at University of North Texas were immensely helpful: Thanks, guys. Michael Krause was an indispensable help in assuring the correctness of the DirectX chapters. Bob Gaines, Mikey Wetzel, and Jason Sandlin from the DirectX team at Microsoft helped make sure Chapters 2, 6, and 8 were ship-shape: Mad props to them. David Black was kind enough to look over Chapter 9 and help remove some errors and clarify a few points.

Finally I need to thank all of the people who helped me get this thing done. I know I won't be able to remember all of them, but here's a short list: Manual and Maria Perez, Katherin Peperzak, Lydia Choy (again), Mike Schuresko, Mike Breen (and the rest of the Originals), Vick Mukherjee, Patrick Nelson, Brian Sharp, and Marcin Krieger.

Introduction

A wise man somewhere, somehow, at some point in history, may have said the best way to start a book is with an anecdote. Never to question the words of a wise man who may or may not have existed, here we go.

When I was a freshman in high school back in 1993, I took the required biology class most kids of my age end up having to take. It involved experiments, lab reports, dissecting of various animals, and the like. One of my lab partners was a fellow named Chris V. We were both interested in computers, and quickly became friends, to the point where talking about biology in class was second to techno-babble.

One night, in the middle of December, Chris called me up. The lab report that was due the next day required results from the experiment we had done together in class, and he had lost his copy of our experiment results. He wanted to know if I could copy mine and bring them over to his place so he could finish writing up the lab. Of course, this was in those heinous pre-car days, so driving to his house required talking my parents into it, finding his address, and various other hardships. While I was willing to do him the favor, I wasn't willing to do it for free. So I asked him what he could do to reciprocate my kind gesture.

"Well," he said, "I guess I can give you a copy of this game I just got."

"Really? What's it called?" said I.

"*Doom*. By the Wolf 3-D guys."

"It's called *Doom*? What kind of name is that??"

After getting the results to his house and the game to mine, I fired the program up on my creaky old 386 DX-20 clone, burning rubber with a whopping 4 MB of RAM. As my space marine took his first tenuous steps down the corridors infested with hellspawn, my life changed. I had done some programming before in school (Logo and Basic), but after I finished playing the first time, I had a clear picture in my head of what I wanted to do with my life: I wanted to write games, write something like *Doom*. I popped onto a few local BBoards and asked two questions: what language was the game written in, and what compiler was used.

Within a day or so I purchased Watcom C 10.0 and got my first book on C programming. My first C program was "Hello, World." My second was a slow, crash-happy, non-robust, wireframe spinning cube.

I tip my hat to John Carmack, John Romero, and the rest of the team behind Doom: My love for creating games was fully realized via their masterpiece. It's because of them that I learned everything that I have about this exceedingly interesting and dynamic area of computer acquired programming. The knowledge I have is what I hope to fill these pages with, so that other people can get into graphics and game programming.

I've found that the best way to get a lot of useful information down in a short amount of space is to use the tried-and-true FAQ (frequently asked questions) format. I figured if people needed answers to some questions about this book as they stood in their local bookstore trying to decide whether or not to buy it, these would be them.

Who are you? What are you doing here?

I'll be the first to say it: I am not a professional game programmer. I hope to be one eventually, but for right now I am but a simple college student. A few years ago I wrote an article for *Game Developer* that led to the initial discussion with Wordware to write this book. Besides that, I've spent summers working at Lucent (in their Graphics department) and at Microsoft (on the Direct3D, Immediate Mode team). I'm not claiming to be a "game programming guru," but I've read all I can on a lot of topics and have a lot of practical experience in graphics programming.

It wasn't too long ago that I was learning the ropes, so I'm hoping that explaining some of the pitfalls that I've gone through can help other people aspiring to write games.

Another thing that I am not is a professional author, nor do I ever plan to be. So I beg that you be quick to forgive any technical and grammatical errors; I assure you that I took every possible means to eliminate both. It seems to be almost a cliché for game programming books to be written by non-game programmers, but I hope I can buck the status quo and create a book that teaches you a lot.

Why was this book written?

I've learned from a lot of amazingly brilliant people, covered a lot of difficult ground, and asked a lot of dumb questions. One thing that I've found is that the game development industry is all about sharing. If everyone shares, everyone knows more neat stuff and the net knowledge of the industry increases. This is a good thing, because then we all get to play better games. No one person could discover all the principles behind computer graphics and game programming themselves; no one can learn in a vacuum. People took the time to share what

they learned with me, and now I'm taking the time to share what I've learned with you.

Who should read this book?

This book was intended specifically for people who know how to program already, but have taken only rudimentary stabs at graphics/game programming, or have never taken any stab at all. You may be a programmer in another field or a college student looking to embark on some side projects.

Who should not read this book?

This book was not designed for beginners. I'm not trying to sound aloof or anything; I'm sure if a beginner picks up this book they'll be able to trudge through it if they feel up to it. However, since I'm so constrained for space, oftentimes I need to breeze past certain concepts (such as inheritance in C++). If you've never programmed before, you'll have an exceedingly difficult time with this book.

On the other hand, this book isn't really designed for professionals either. I'm sure that most people who have pushed games out the door will only find one or two chapters in this book have any material they haven't seen before.

What are the requirements for using the code?

The code was written in C++, using Microsoft Visual C++ 6.0. The .DSPs and .DSWs are provided on the CD; the .DSPs will work with versions previous to 6.0, and the .DSWs will work with 6.0 and up. If you choose to use a different compiler, getting the source code to work should be a fairly trivial task. I specifically wrote this code to use as little non-standard C++ as possible (as far as I know, the only non-standard C++ I use is nameless structures within unions).

Why use Windows? Why not use Linux?

I chose Win32 as the API environment to use because 90% of computer users currently work on Windows. Win32 is not an easy API to understand, especially after using DOS coding conventions. It isn't terribly elegant either, but I suppose it could be worse. We could choose other platforms to work on, but doing so reduces our target audience by a factor of 9 or more.

If you've never heard of Linux, Linux is an open source operating system. This means anyone can download the source to see how the system works, and

anyone can join teams that work on future versions and make contributions to the operating system. The Linux community has a lot of pride for what it does, and as a result Linux is an incredibly small, fast, and stable operating system. There are a variety of window managers available for download, some that emulate other WMs like Windows or MacOS, some that take new directions (like the ultra-simplistic Blackbox and the uber-complicated Enlightenment). Check out www.linux.org.

Why use Direct3D? Why not use OpenGL?

For those of you who have never used it, OpenGL is another graphics API. Silicon Graphics designed it in the early '90s for use on their high-end graphics workstations. It has been ported to countless platforms and operating systems. Outside of the games industry, in areas like simulation and academic research, OpenGL is the de facto standard for doing computer graphics. It is a simple, elegant, and fast API. Check out www.opengl.org for more information.

But it isn't perfect. First of all, OpenGL has a large amount of functionality in it. Making the interface so simple requires that the implementation take care of a lot of ugly details to make sure everything works correctly. Because of the way drivers are implemented, each company that makes a 3-D card has to support the entire OpenGL feature set in order to have a fully compliant OpenGL driver. These drivers are extremely difficult to implement correctly, and the performance on equal hardware can vary wildly based on driver quality. In addition, DirectX has the added advantage of being able to move quicker to accommodate new hardware features. DirectX is controlled by Microsoft (which can be a good or bad thing, depending on your view of it) while OpenGL extensions need to be deliberated by committees.

My initial hope was to have two versions of the source code, one for Windows and Direct3D, the other for Linux and OpenGL. This ended up not being possible, so I had to choose one or the other; I chose Direct3D.

Why use C++? Why not {C, ASM, Java, *}

I had a few other language choices I was kicking around when planning this book. Although there are acolytes out there for Delphi, VB, and even ML, the only languages I seriously considered were C++, Java, and C. Java is designed by Sun Microsystems and is an inherently object-oriented language, with some high-level language features like garbage collection. C is about as low level as programming gets without dipping into assembly. It has very few if any high-level constructs and doesn't abstract anything away from the programmer.

C++ is an interesting language because it essentially sits directly between the functionality of the other two languages. C++ supports COM better than C does (this will be more thoroughly discussed in Chapter 1). Also, class systems and operator overloading generally make code easier to read (although of course any good thing can and will be abused). Java, although very cool, is an interpreted language. Every year this seems to be less important: JIT compilation gets faster and more grunt work is handed off to the APIs. However, I felt C++ would be a better fit for the book. Java is still a very young language and is still going through a lot of change.

Do I need a 3-D accelerator?

That depends. Technically no, you can get by without any accelerator at all, just using Direct3D's software rasterizer. However, it's extremely slow, far from real time for anything but trivially simple scenes. It's almost impossible to buy a computer these days without some sort of 3-D acceleration, and an accelerator capable of handling all the code in this book can be purchased for under $100.

How hardcore is the C++ in this book?

Some people see C++ as a divine blade to smite the wicked. They take control of template classes the likes of which you have never seen. They overload the iostream operators for all of their classes. They see multiple inheritance as a hellspawn of satan himself. I see C++ as a tool. The more esoteric features of the language (such as the iostream library), I don't use at all. Less esoteric features (like multiple inheritance), I use when it makes sense. To a large degree I'm still a learner. My coding style still leaves a great deal to be desired, but I did everything in my power to make the code clean and easy to read. Having a coding style you stick to is invaluable. The code for this book was written over an eleven-month period, but I can pick up the code I wrote at the beginning and still grok it, because I commented and used some good conventions. If I can understand it, hopefully you can too.

What are the coding conventions used in the source?

One of the greatest books I've ever read on programming was *Code Complete* (Microsoft Press). It's a handbook on how to program well, not just how to program. Nuances like the length of variable names, design of subroutines, and length of files are covered in detail in this book; I strongly encourage anyone who wants to become a great programmer to pick it up. You may notice some of the conventions I use in this book are similar to the conventions described in

Code Complete: some of them are borrowed from the great game programmers like John Carmack, and some of them are borrowed from source in DirectX, MFC, and Win32.

I've tried really hard to make the code in this book accessible to everyone. I comment anything I think is unclear, I strive for good choice in variable names, and I try to make my code look clean while still trying to be fast. Of course, I won't please everyone. Assuredly, there are some C++ coding standards I'm probably not following correctly. There are some pieces of code that would get much faster with a little obfuscation.

If you've never used C++ before, or are new to programming, this book is going to be extremely hard to digest. A good discussion on programming essentials and the C++ language is *C++ Primer* (Lippman et al; Addison-Wesley Publishing).

Class/structure names

MFC names its classes with a prefixed C. As an example, a class that represents the functionality of a button is called CButton. I like this fine, but due to namespace clashing, I instead prefix my own classes with a lowercase c for classes, a lowercase s for structs, a lowercase i for interfaces, and a lowercase e for enumerations (`cButton` or `sButton`).

There is one notable exception. While most classes are intended to hide functionality away and act as components, there are a few classes/structures that are intended to be instantiated as basic primitives. So for basic mathematic primitives like points and matrices, I have no prefix, and I postfix with the dimension of the primitive (2-D points are `point2`, 3-D points are `point3`, etc.). This is to allow them to have the same look and feel as their closest conceptual neighbor, float. For the same reason, all of the mathematic primitives have a lot of overloaded operators to simplify math-laden code.

Variable names

Semi-long variable names are a good thing. They make your code self-comment itself. One needs to be careful though: make them too long, and they distract from both the code itself and the process of writing it.

I use short variables very sporadically; `int i`, `j`, `k` pop up a lot in my code for loops and whatnot, but besides that I strive to give meaningful names to the variables I use. Usually, this means that they have more than one word in them. The system I use specifies lowercase for the first word and initial cap for each word after that, with no underscores (an example would be `int numObjects`). If the last letter of a word is a capital letter, an underscore is placed to separate it from the next word (example, class `cD3D_App`).

A popular nomenclature for variables is Hungarian notation, which we'll touch on in Chapter 1. I'm not hardcore about it, but generally my floats are

prefixed with f, my ints with i, and my pointers with p (examples: `float fTimer`; `int iStringSize`; `char* pBuffer`). Note that the prefix counts as the first word, making all words after it caps. (I find `pBuffer` much more readable than `pbuffer`.)

I also use prefixes to define special qualities of variables. Global variables are preceded with a "g_" (an example would be `int g_hInstance`); static variables are preceeded with an "s_" (`static float s_fTimer`); member variables of classes are preceeded with an "m_" (`int m_iNumElements`).

Ellis' distinctive Wordware title, Learn 3D Game Programming with DirectX 7.0, focuses on the most innovative and updated feature of DirectX 7.0, Direct3D, which is the most frequently used, but also the most complex feature of this popular game development platform. Get A Copy. Kindle Store. Amazon. Stores â–¾. Audible Barnes & Noble Walmart eBooks Apple Books Google Play Abebooks Book Depository Indigo Alibris Better World Books IndieBound. Libraries. Read advanced chapters on interacting with game players using. DirectInput API, adding sound effects and music using DirectSound, and allowing users to battle live. opponents over a modem or network using DirectPlay.Â Development Series) Advanced 3-D Game Programming With Directx 7.0 (Wordware Game. Developer's Library) Strategy Game Programming with DirectX 9 (Wordware Game and Graphics. Library) Microsoft Win32 Developer's Reference Library - GDI (Microsoft Developers Library Win 32.