
Preface



Not all that long ago, at least for some of us, all aspects of computer graphics were programmable. In fact, “programmable” is probably not a good term, because that implies that there was an option when creating an image. There was not. If you wanted anything to happen, you had no choice but to program it.

Computer graphics APIs changed that for most graphics practitioners. With a good API, you could write very good graphics programs much more easily because you could let the API’s functionality take over large portions of the graphics process. However, you paid for this in giving up anything that the API didn’t know how to handle. A good example is in surface shading, where most of the 1990s APIs could not support anything beyond simple smooth lighted surfaces.

Fortunately, neither the computer graphics research community nor advanced graphics practitioners were satisfied with this. First in software

and then in hardware, as graphics processors were developed, specific functionality was developed to support the programming of features that fixed-function graphics APIs had fenced off. This functionality has now developed its own standards, including the GLSL shader language that is part of the OpenGL 2.1 standard. Programmable graphics shaders, programs that can be downloaded to a graphics processor to carry out operations outside the fixed-function pipeline of earlier standards, have become a key feature of computer graphics.

This process is now being paralleled in the teaching and learning of computer graphics. Just as students usually first learned computer graphics through a graphics standard, most often OpenGL, students now need to understand the role of programmable shaders and to have experience in writing and using them. One of the remarkable things about shader-level programming is that it brings us all back to the same kind of graphics questions that were being examined in the 1970s. We can now manipulate vertices and even individual pixels while still having full OpenGL API support whenever we want to use it. This gives students and practitioners a wonderful range of capabilities that can be used in games, in scientific visualization, and in general graphical communication.

This book is designed to open computer graphics shader programming to the student, whether in a traditional class or on their own. It is intended to complement texts based on fixed-function graphics APIs, specifically OpenGL. It introduces shader programming in general, and specifically the GLSL shader language. It also introduces a flexible, easy-to-use tool, *glman*, that helps you develop and tune shaders outside an application that would use them.

This book is intended as a text for a second course in computer graphics at either the undergraduate or graduate level. It is not a textbook for a first course in computer graphics, because it assumes knowledge of not only OpenGL, but of general graphics concepts. Knowledge of another graphics API, such as Direct3D, will work, but we focus on GLSL and will use OpenGL terminology consistently. Because shader programming lets you work in areas that APIs might hide from you, sometimes you will need to work at fundamental levels of geometry, lighting, shading, and similar concepts. You will benefit from a prior understanding of these. You will also find that shader programming exposes some areas of API operation that you may not have fully understood, so you may need to review some of these details.

Our choice of GLSL as the vehicle for teaching shaders is based on its integration into the widely-used OpenGL multiplatform API and its solid performance. The concepts presented here will also help anyone who works with other shader APIs, such as Cg or HLSL, because the basic ideas of shaders

are all similar. The book is designed to take the student from a review of the fixed-function graphics pipeline, through an understanding of the basic role and functions of shader programming, to solid experience in writing vertex, fragment, and geometry shaders for both *glman* and actual applications.

While it might seem logical to treat shaders in the order in which they are applied in the expanded graphics pipeline, with vertex shaders first, followed by geometry shaders and then fragment shaders, we have chosen to lay out their order a little differently. Again, it might seem logical to treat shaders in the order of frequency of use, with fragment shaders first, followed by vertex shaders and then geometry shaders, but that also does not quite seem to work. Because many of the operations of a fragment shader depend on things that come out of a vertex shader, we treat vertex shaders first, followed by fragment shaders, and then geometry shaders.

The overall outline of the text is straightforward. In the first four chapters, which make up the background for the rest of the book, we begin by covering the fixed-function graphics pipeline of OpenGL in Chapter 1. We then present the basic principles of vertex, fragment, and geometry shaders in Chapter 2, including several examples, using the GLSL shader language. Chapter 3 introduces the *glman* tool with a kind of mini-manual on its use. Finally, Chapter 4 presents the GLSL shader language and discusses its similarities and differences from the C programming language.

The next set of chapters set up vertex and fragment shader concepts. Chapter 5 covers lighting from the point of view of shaders, and introduces the ADS (ambient, diffuse, specular) lighting function that we will use several times in later chapters. This is fundamental in both vertex and fragment shaders, since vertex shaders often need to compute lighting for each vertex, and fragment shaders may want to compute lighting for each pixel. In Chapter 6 we cover vertex shaders, emphasizing their inputs and outputs, as well as the ways they can be used to modify vertex geometry. Finally, in Chapter 7 we cover fragment shaders, again emphasizing their inputs and outputs, and showing how they can be used to replace the usual fixed-function fragment operations.

The next three chapters discuss particular capabilities of fragment shaders. In Chapter 8 we describe the way fragment shaders handle texture mapping, including bump mapping, cube mapping, and rendering a scene to a texture. Chapter 9 discusses noise functions and their role in writing textures and shaders, and introduces a tool, *noisegraph*, that lets you experiment with the properties of 1D and 2D noise functions. Finally, Chapter 10 examines some ways you can manipulate 2D images, treated as textures, with the tools that fragment shaders make available.

Chapter 11 stands somewhat alone. It presents geometry shaders, including how they are related to vertex and fragment shaders, as well as their own capabilities. Several examples highlight the way geometry shaders can expand the geometric capability of your models or show the capability of geometry shaders to handle simple level-of-detail operations.

The final set of chapters focus on computer graphics shaders in applications. Chapter 12 describes the GLSL API that lets you compile, link, and use shaders in an application. It also discusses passing data and graphics state information to shader programs and presents a simple C++ class that encapsulates the process of incorporating shader programs in an application. In Chapter 13 we focus on how shaders can be used in scientific visualization applications, and show examples of a number of specific visualization operations. Finally, in Chapter 14 we explore some fun things you can do with computer graphics shaders, under the guise of getting real work done.

While most of the topics in this text are straightforward, others are tricky or deserve special attention. We have followed the lead of the Nicholas Bourbaki mathematics texts of the early 20th century and have highlighted these with a “dangerous curves ahead” sign, as shown to the left. We hope this will help you notice these points.



We are confident that the tools and capabilities we describe in this book will both make you a better graphics programmer and make graphics programming a much more interesting experience for you. And as OpenGL evolves toward the future and shaders become the only way that geometry and rendering are handled, we believe that you will find this text to be an invaluable guide.

Thanks

The authors of this book owe thanks to a number of people, primarily on Mike Bailey’s side.

To faculty colleagues at Oregon State University for their support and camaraderie: Bella Bose, Terri Fiez, Karti Mayaram, Ron Metoyer, Eric Mortensen, Cherri Pancake, Sinisa Todorovic, and Eugene Zhang.

To the superbly talented UCSD and OSU graphics students who have shared this shader expedition: Tim Bauer, William Brendel, Guoning Chen, Matt Clothier, John Datuin, Will Dillon, Jonathan Dodge, Chuck Evans, Nick Gebbie, Kyle Hatcher, Nick Hogle, Chris Janik, Ankit Khare, Vasu Lakshmanan, Adam Leibel, Jessica McGregor, Daniel Moffitt, Chris Moore, Patrick Neill,

Jonathan Palacios, Nadia Payet, Randy Rauwendaal, Dwayne Robinson, Avneet Sandhu, Ian South-Dickinson, Madhu Srinivasan, Ben Tribelhorn, and Ben Weiss.

To professional colleagues: Ryan Bailey, Mike Gannis, Jenny Orr, and Todd Shechter.

To the folks at NVIDIA for their support, especially Dave Luebke, David Kirk, and Jen-Hsun Huang.

To the folks at AMD/ATI for their support, especially Bill Licea-Kane.

To Randi Rost, for his support from positions at both 3D Labs and Intel, and for writing his “Orange Book,” from which so much of what went into this book was learned.

To Paramount Pictures for their permission to reprint the image in Figure 2.2, and to Pixar, Inc. for providing the original images.

We also thank Alice Peters and Kevin Jackson-Mead for their advice and assistance in developing this project, and the reviewers for helping us refine some key points in the text.

Mike Bailey
Corvallis, Oregon

Steve Cunningham
Coralville, Iowa