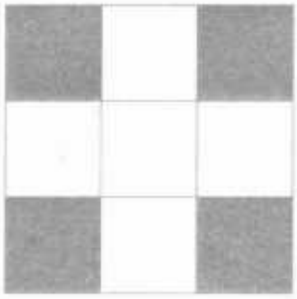


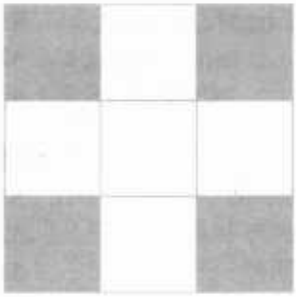
Fundamentals of Computer Graphics

Peter Shirley
School of Computing
University of Utah

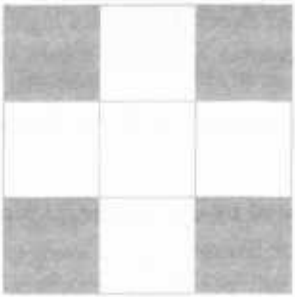
with

Michael Ashikhmin
Michael Gleicher
Stephen R. Marschner
Erik Reinhard
Kelvin Sung
William B. Thompson
Peter Willemsen

A K Peters
Wellesley, Massachusetts



Contents



Preface

This book is a product of several graphics courses I have taught at Indiana University and the University of Utah. All graphics books must choose between teaching the low-level details "under the hood" of graphics programs or teaching how to use modern graphics APIs, such as OpenGL, Direct3D, and Java3D. This book chooses the former approach. I do not have a good justification for this choice other than that I have taught both styles of courses, and the students in the "low-level" courses seemed to understand the material better than the other students and even seemed to use the APIs more effectively. There are many reasons this might be true, and the effect may not transfer to other teachers or schools. However, I believe that teaching the fundamentals is usually the right approach, whether in graphics, another academic discipline, or a sport.

How to Use this Book

The book begins with nine chapters that roughly correspond to a one-semester course which takes students through the graphics pipeline and basic ray tracing. It has students implement everything—i.e., it is not a "learn OpenGL" style text. However, the pipeline presented is consistent with the one implemented in graphics hardware, and students who have used the book should find OpenGL or other common APIs familiar in many ways.

The second part of the book is a series of advanced topics that are not highly ordered. This allows a variety of second-semester courses and a few weeks of advanced topics in a first semester course.

For the first semester, I would suggest the following as a possible outline of initial assignments:

1. Math homework at the end of Chapter 2 followed by at least one in-class exam.
2. Line rasterization.
3. Triangle rasterization with barycentric color interpolation.
4. Orthographic wireframe drawing.
5. Perspective wireframe drawing.
6. BSP-tree with flat-shaded triangles and wireframe edges with only trivial z-clipping and with mouse-driven viewpoint selection.
7. Finite-precision z-buffer implementation with only trivial z-clipping.

Following these assignments, the instructor could do assignments on ray tracing or could have the students add shadow-maps, Phong lighting, clipping, and textures to their z-buffers, or they could move the students into programming with a 3D API.

About the Cover

The cover image is from *Tiger in the Water* by J. W. Baker (brushed and airbrushed acrylic on canvas, 16" by 20", www.jwbart.com).

The subject of a tiger is a reference to a wonderful talk given by Alain Fournier (1943–2000) at the Cornell Workshop in 1998. His talk was an evocative verbal description of the movements of a tiger. He summarized his point:

> Even though modelling and rendering in computer graphics have been improved tremendously in the past 35 years, we are still not at the point where we can model automatically a tiger swimming in the river in all its glorious details. By automatically I mean in a way that does not need careful manual tweaking by an artist/expert.
>
> The bad news is that we have still a long way to go.
>
> The good news is that we have still a long way to go.

Online Resources

The web site for this book is http://www.cs.utah.edu/~shirley/fcg2/. I will maintain an errata list there as well as links to people using the book in classes. Although I do not provide slides for the course, Rich Riesenfeld has graciously agreed to make his excellent slides available, and a pointer to those slides will be available at the book's web site. Most of the figures in this book are in Abobe Illustrator format. I would be happy to convert specific figures into portable formats on request. Please feel free to contact me at shirley@cs.utah.edu.

Changes in this Edition

There are many small changes in the material from the first edition of this book, but the two large changes are the addition of a bibliography and the addition of new chapters written by colleagues. These colleagues are people I think are clear thinkers and communicators, and I invited them each to write a chapter with arm-twisting designed to get certain key topics covered. Most of them have used the book in a class and were thus familiar with its approach. The bibliography is not meant to be extensive, but is instead a place for readers to get started. I am sure there are omissions there, and I would like to hear about any crucial references we have missed. The new chapters are:

Signal Processing by Stephen Marschner, Cornell University (Chapter 4).

Curves by Michael Gleicher, University of Wisconsin (Chapter 15).

Computer Animation by Michael Ashikhmin, SUNY at Stony Brook (Chapter 16).

Using Graphics Hardware by Peter Willemsen, University of Minnesota Duluth (Chapter 17).

Building Interactive Graphics Applications by Kelvin Sing, University of Washington Bothell (Chapter 18)

Visual Perception by William B. Thompson, University of Utah (Chapter 21).

Tone Reproduction by Erik Reinhard, University of Central Florida (Chapter 22).

Acknowledgements

The following people have provided helpful comments about the book: Josh Andersen, Zeferino Andrade, Michael Ashikhman, Adam Berger, Adeel Bhutta, Solomon Boulos, Stephen Chenney, Michael Coblenz, Greg Coombe, Frederic Cremer, Brian Curtin, Dave Edwards, Jonathon Evans, Karen Feinauer, Amy Gooch, Eungyoung Han, Chuck Hansen, Andy Hanson, Dave Hart, John Hart, Helen Hu, Vicki Interrante, Henrik Wann Jensen, Shi Jin, Mark Johnson, Ray Jones, Kristin Kerr, Dylan Lacewell, Mathias Lang, Philippe Laval, Marc Levoy, Howard Lo, Ron Metoyer, Keith Morley, Eric Mortensen, Tamara Munzner, Koji Nakamaru, Micah Neilson, Blake Nelson, Michael Nikelsky, James O'Brien, Steve Parker, Sumanta Pattanaik, Matt Pharr, Peter Poulos, Shaun Ramsey, Rich Riesenfeld, Nate Robins, Nan Schaller, Chris Schryvers, Tom Sederberg, Richard Sharp, Sarah Shirley, Peter-Pike Sloan, Tony Tahbaz, Jan-Phillip Tiesel, Bruce Walter, Alex Williams, Amy Williams, Chris Wyman, and Kate Zebrose.

Ching-Kuang Shene and David Solomon allowed me to borrow examples from their works. Henrik Jensen, Eric Levin, Matt Pharr, and Jason Waltman generously provided images. Brandon Mansfield was very helpful in improving the content of the discussion of hierarchical bounding volumes for ray tracing. Carrie Ashust, Jean Buckley, Molly Lind, Pat Moulis, and Bob Shirley, provided valuable logistical support. Miranda Shirley provided valuable distractions.

I am extremely thankful to J. W. Baker helping me to get the cover I envisioned. In addition to being a talented artist, he was a great pleasure to work with personally.

Many works were helpful in preparing this book, and most of them appear in the notes for the appropriate chapters. However, a few pieces that influenced the content and presentation do not, and I list them here. I thank the authors for their help. These include the two classic computer graphics texts I first learned the basics from as a student: *Computer Graphics: Principles & Practice* (Foley, Van Dam, Feiner, & Hughes, 1990), and *Computer Graphics* (Hearn & Baker, 1986). Other texts include both of Alan Watt's classic books (Watt, 1993, 1991), Hill's *Computer Graphics Using OpenGL* (Francis S. Hill, 2000), Angel's *Interactive Computer Graphics: A Top-Down Approach With OpenGL* (Angel, 2002), Hughes Hoppe's University of Washington dissertation (Hoppe, 1994), and Rogers' two classic graphics texts (D. F. Rogers, 1985, 1989).

This book was written using the LaTeX document preparation software on an Apple Powerbook. The figures were made by the author using the *Adobe Illustrator* package. I would like to thank the creators of those wonderful programs.

I'd like to thank the University of Utah for allowing me to work on this book during sabbatical.

I would like to especially thank Alice and Klaus Peters for encouraging me to write the first edition of this book, for their great skill in bringing a book to fruition and for their dedication to making their books the best they can be. In addition to finding many errors in formulas and language in the second edition, they put in many weeks of extremely long hours in the home stretch of the process, and I have no doubt this book would not have been finished without their extraordinary efforts.

Salt Lake City
April 2005

Peter Shirley

1 Introduction

The term *Computer Graphics* describes any use of computers to create or manipulate images. This book takes a slightly more specific view and deals mainly with algorithms for image generation. Doing computer graphics inevitably requires some knowledge of specific hardware, file formats, and usually an API[1] or two. The specifics of that knowledge are a moving target due to the rapid evolution of the field, and therefore such details will be avoided in this text. Readers are encouraged to supplement the text with relevant documentation for their software/hardware environment. Fortunately the culture of computer graphics has enough standard terminology and concepts that the discussion in this book should map nicely to most environments. This chapter defines some basic terminology, and provides some historical background as well as information sources related to computer graphics.

1.1 Graphics Areas

It is always dangerous to try to categorize endeavors in any field, but most graphics practitioners would agree on the following major areas and that they are part of the field of computer graphics:

[1] An *application program interface* (API) is a software interface for basic operations such as line drawing. Current popular APIs include OpenGL, Direct3D, and Java3D.

- **Modeling** deals with the mathematical specification of shape and appearance properties in a way that can be stored on the computer. For example, a coffee mug might be described as a set of ordered 3D points along with some interpolation rule to connect the points and a reflection model that describes how light interacts with the mug.
- **Rendering** is a term inherited from art and deals with the creation of shaded images from 3D computer models.
- **Animation** is a technique to create an illusion of motion through sequences of images. Here, modeling and rendering are used, with the handling of time as a key issue not usually dealt with in basic modeling and rendering.

There are many other areas that involve computer graphics, and whether they are core graphics areas is a matter of opinion. These will all be at least touched on in the text. Such related areas include the following:

- **User interaction** deals with the interface between input devices such as mice and tablets, the application, and feedback to the user in imagery and other sensory feedback. Historically, this area is associated with graphics largely because graphics researchers had some of the earliest access to the input/output devices that are now ubiquitous.
- **Virtual reality** attempts to *immerse* the user into a 3D virtual world. This typically requires at least stereo graphics and response to head motion. For true virtual reality, sound and force feedback should be provided as well. Because this area requires advanced 3D graphics and advanced display technology, it is often closely associated with graphics.
- **Visualization** attempts to give users insight via visual display. Often there are graphic issues to be addressed in a visualization problem.
- **Image processing** deals with the manipulation of 2D images and is used in both the fields of graphics and vision.
- **3D scanning** uses range-finding technology to create measured 3D models. Such models are useful for creating rich visual imagery, and the processing of such models often requires graphics algorithms.

1.2 Major Applications

Almost any endeavor can make some use of computer graphics, but the major consumers of computer graphics technology include the following industries:

- **Video games** increasingly use sophisticated 3D models and rendering algorithms.
- **Cartoons** are often rendered directly from 3D models. Many traditional 2D cartoons use backgrounds rendered from 3D models which allows a continuously moving viewpoint without huge amounts of artist time.
- **Film special effects** use almost all types of computer graphics technology. Almost every modern film uses digital compositing to superimpose backgrounds with separately filmed foregrounds. Many films use computer-generated foregrounds with 3D models.
- **CAD/CAM** stands for *computer-aided design* and *computer-aided manufacturing*. These fields use computer technology to design parts and products on the computer and then, using these virtual designs, to guide the manufacturing procedure. For example, many mechanical parts are now designed in a 3D computer modeling package, and are then automatically produced on a computer-controlled milling device.
- **Simulation** can be thought of as accurate video gaming. For example, a flight simulator uses sophisticated 3D graphics to simulate the experience of flying an airplane. Such simulations can be extremely useful for initial training in safety-critical domains such as driving, and for scenario training for experienced users such as specific fire-fighting situations that are too costly or dangerous to create physically.
- **Medical imaging** creates meaningful images of scanned patient data. For example, a magnetic resonance imaging (MRI) dataset is composed of a 3D rectangular array of density values. Computer graphics is used to create shaded images that help doctors digest the most salient information from such data.
- **Information visualization** creates images of data that do not necessarily have a "natural" visual depiction. For example, the temporal trend of the price of ten different stocks does not have an obvious visual depiction, but clever graphing techniques can help humans find patterns in such data.

1.3 Graphics APIs

A key part of using graphics libraries is dealing with an *application program interface* (API). An API is a software interface that provides a model for how an

application program can access system functionality, such as drawing an image into a window. Typically, the two key issues in designing graphics programs are dealing with graphics calls such as "draw triangle" and handling user interaction such as a button press.

Most APIs have a user-interface toolkit of some kind that uses *callbacks*. Callbacks refer to the process of using function pointers or virtual functions to pass a reference to a function. For example, to associate an action with a button press, an underlying function is dynamically associated with the button press. In this way, the user-interface toolkit can process the event of the button press, and any action can be associated with it by the programmer.

There are currently two dominant paradigms for APIs. The first is the integrated approach of Java where the graphics and user-interface toolkits are integrated and portable *packages* that are fully standardized and supported as part of the language. The second is represented by Direct3D and OpenGL, where the drawing commands are part of a software library tied to a language such as C++, and the user-interface software is an independent entity that might vary from system to system. In this latter approach, it is problematic to write portable code, although for simple programs it may be possible to use a portable library layer on top of the system specific event-handling.

Whatever your choice of API, the basic graphics calls will be largely the same, and the concepts of this book will apply.

1.4 3D Geometric Models

A key part of graphics programs is using 3D geometric models. These models describe 3D objects using mathematical primitives such as spheres, cubes, cones, and polygons. The most ubiquitous type of model is composed of 3D triangles with shared vertices, which is often called a *triangle mesh*. These meshes are sometimes generated by artists using an interactive modeling program and sometimes by range scanning devices. In either case, these models usually contain many triangles, most of them small, so your programs should be optimized for such datasets.

1.5 Graphics Pipeline

Almost all modern computers now have an efficient 3D *graphics pipeline*. This is a special software/hardware subsystem that efficiently draws 3D primitives in

perspective. Usually these systems are optimized for processing 3D triangles with shared vertices. The basic operations in the pipeline map the 3D vertex locations to 2D screen positions, and shade the triangles so that they both look realistic and appear in proper back-to-front order.

Although drawing the triangles in valid back-to-front order was once the most important research issue in computer graphics, it is now almost always solved using the *z-buffer*, which uses a special memory-buffer to solve the problem in a brute-force manner.

It turns out that the geometric manipulation used in the graphics pipeline can be accomplished almost entirely in a 4D coordinate space composed of three traditional geometric coordinates and a fourth *homogeneous* coordinate that helps us handle perspective viewing. These 4D coordinates are manipulated using 4 by 4 matrices and 4-vectors. The graphics pipeline, therefore, contains much machinery for efficiently processing and composing such matrices and vectors. This 4D coordinate system is one of the most subtle and beautiful constructs used in computer science, and it is certainly the biggest intellectual hurdle to jump when learning computer graphics. A big chunk of the first part of every graphics book deals with these coordinates.

The speed of most modern graphics pipelines is roughly proportional to the number of triangles being drawn. Because interactivity is typically more important to applications than visual quality, it is worthwhile to minimize the number of triangles used to represent a model. In addition, if the model is viewed in the distance, fewer triangles are needed than when the model is viewed from a closer distance. This suggests that it is useful to represent a model with a varying *level-of-detail* (LOD).

1.6 Numerical Issues

Many graphics programs are really just 3D numerical codes. Numerical issues are often crucial in such programs. In the "old days," it was very difficult to handle such issues in a robust and portable manner because machines had different internal representations for numbers, and even worse, handled exceptions in many incompatible fashions. Fortunately, almost all modern computers conform to the *IEEE floating point* standard (IEEE Standards Association, 1985). This allows the programmer to make many convenient assumptions about how certain numeric conditions will be handled.

Although IEEE floating point has many features that are valuable when coding numeric algorithms, there are only a few that are crucial to know for most

situations encountered in graphics. First, and most important, is to understand that there are three "special" values for real numbers in IEEE floating point:

infinity (∞) This is a valid number that is larger than all other valid numbers.

minus infinity ($-\infty$) This is a valid number that is smaller than all other valid numbers.

not a number (NaN) This is an invalid number that arises from an operation with undefined consequences, such as zero divided by zero.

The designers of IEEE floating point made some decisions that are extremely convenient for programmers. Many of these relate to the three special values above in handling exceptions such as division by zero. In these cases an exception is logged, but in many cases the programmer can ignore that. Specifically, for any positive real number a, the following rules involving division by infinite values hold:

$$+a/(+\infty) = +0$$
$$-a/(+\infty) = -0$$
$$+a/(-\infty) = -0$$
$$-a/(-\infty) = +0$$

Note that IEEE floating point distinguishes between -0 and $+0$. In most graphics programs this distinction does not matter, but it is worth keeping in mind for more classical numeric algorithms.

Other operations involving infinite values behave the way one would expect. Again for positive a, the behavior is:

$$\infty + \infty = +\infty$$
$$\infty - \infty = \text{NaN}$$
$$\infty \times \infty = \infty$$
$$\infty/\infty = \text{NaN}$$
$$\infty/a = \infty$$
$$\infty/0 = \infty$$
$$0/0 = \text{NaN}$$

The rules in a Boolean expression involving infinite values are as expected:

1. All finite valid numbers are less than $+\infty$.
2. All finite valid numbers are greater than $-\infty$.
3. $-\infty$ is less than $+\infty$.

The rules involving expressions that have NaN values are simple:

1. Any arithmetic expression that includes NaN results in NaN.
2. Any Boolean expression involving NaN is false.

Perhaps the most useful aspect of IEEE floating point is how divide-by-zero is handled; for any positive real number a, the following rules involving division by zero values hold:

$$+a/+0 = +\infty$$
$$-a/-0 = -\infty$$

Note that some care must be taken if negative zero (-0) might arise in a code, but there are many numeric codes that become much simpler if the programmer takes advantage of IEEE floating point. For example, consider the expression:

$$a - \frac{1}{\frac{1}{b} + \frac{1}{c}}.$$

Such expressions arise with resistors and lenses. If divide-by-zero resulted in a program crash (as was true in many systems before IEEE floating point), then two *if* statements would be required to check for small or zero values of b or c. Instead, with IEEE floating point, if b or c are zero, we will get a zero value for a as desired. Another common technique to avoid special checks is to take advantage of the Boolean properties of NaN. Consider the following code segment:

$a = f(x)$
if $(a > 0)$ **then**
  do something

Here, the function f may return "ugly" values such as ∞ or NaN. Because the *if* statement is false for $a =$ NaN or $a = -\infty$ and true for $a = +\infty$, no special checks are needed. This makes programs smaller, more robust, and more efficient.

1.7 Efficiency

There are no magic rules for making code more efficient. Efficiency is achieved through careful tradeoffs, and these tradeoffs are different for different architectures. However, for the foreseeable future, a good heuristic is that programmers should pay more attention to memory access patterns than to operation counts. This is the opposite of the best heuristic of a decade ago. This switch has occurred because the speed of memory has not kept pace with the speed of processors. Since that trend continues, the importance of limited and coherent memory access for optimization should only increase.

A reasonable approach to making code fast is to proceed in the following order, taking only those steps which are needed:

1. Write the code in the most straightforward way possible. Compute data as needed on the fly without storing it.
2. Compile in optimized mode.
3. Use whatever profiling tools exist to find critical bottlenecks.
4. Examine data structures to look for ways to improve locality. If possible, make data unit sizes match the cache/page size on the target architecture.
5. If profiling reveals bottlenecks in numeric computations, examine the assembly code generated by the compiler for missed efficiencies. Rewrite source code to solve any problems you find.

The most important of these steps is the first one. Most "optimizations" make the code harder to read without speeding things up. In addition, time spent upfront optimizing code is usually better spent correcting bugs or adding features. Also, beware of suggestions from old texts; some classic tricks such as using integers instead of reals may no longer yield speed because some modern CPUs can usually perform floating point operations just as fast as they perform integer operations. In all situations, profiling is needed to be sure of the merit of any optimization for a specific machine and compiler.

1.8 Software Engineering

A key part of any graphics program is to have good classes or routines for geometric entities such as vectors and matrices, as well as graphics entities such as RGB colors and images. These routines should be made as clean and efficient as

possible. Most graphics programmers use C++, so some discussion of that language is in order. A critical issue is whether locations and displacements should be separate classes because they have different operations, e.g., a location multiplied by one-half makes no geometric sense while one-half of a displacement does (Goldman, 1985). This is a personal decision, but I believe strongly in the KISS ("keep it simple, stupid") principle, and in that light the argument for two classes is not compelling enough to justify the added complexity (for a counter argument see (DeRose, 1989)). This implies that some basic classes that should be written include:

- vector2: A 2D vector class that stores an x and y component. It should store these components in a length-2 array so that an indexing operator can be well supported. You should also include operations for vector addition, vector subtraction, dot product, cross product, scalar multiplication, and scalar division.
- vector3: A 3D vector class analogous to vector2.
- hvector: A homogeneous vector with four components (see Chapter 7).
- rgb: An RGB color that stores three components. You should also include operations for RGB addition, RGB subtraction, RGB multiplication, scalar multiplication, and scalar division.
- transform: A four-by-four matrix for transformations. You should include a matrix multiply and member functions to apply to locations, directions, and surface normal vectors. As shown in Chapter 6, these are all different.
- image: A 2D array of RGB pixels with an output operation.

In addition, you might or might not want to add classes for intervals, orthonormal bases, and coordinate frames. You might also consider unit-length vectors, although I have found them more pain than they are worth. There are several basic decisions to be made which are outlined in the following sections.

1.8.1 Float versus Double

Modern architecture suggests that keeping memory use down and maintaining coherent memory access are the keys to efficiency. This suggests using single-precision data. However, avoiding numerical problems suggests using double-precision arithmetic. The tradeoffs depend on the program, but it is nice to have

a default in your class definitions. I suggest using doubles for geometric computation and floats for color computation. Where memory usage is high, as it is for triangle meshes, I suggest storing float data, but converting to double when data is accessed through member functions.

1.8.2 Inlining

Inlining is a key to efficiency for utility classes such as RGB. Almost all RGB and vector functions should be inlined. Be sure to profile your code to make sure that things are actually being inlined. Non-utility code and other large functions should not be inlined unless the profiler shows them to be hogging runtime. Even then be sure making them inline does not slow the code further. Note, that on most systems, the inline function definitions must be in the header files. For member functions, these can be linked to the declarations, for example:

```
class vector3 {
      .
      .
      .
    double lengthSquared ( return x()*x()+y()*y()
        +z()*z(); }
};
```

1.8.3 Member Functions versus Non-Member Operators

For operators such as the addition of two vectors, we can make them either a member of a vector class or an operator that exists outside of the class. I suggest that such operators always exist outside of a class. This is because it is the only solution for something like the multiplication operator for a double and a vector (as opposed to vector times double). Since we have to make it a non-member in such cases, we may as well be consistent and always make it a non-member. We should make such operators as compact as possible, for example:

```
inline vector3 operator+(vector3 a, vector3 b) {
    return vector3( a.x() + b.x(), a.y()
        + b.y(), a.z() + b.z() );
}
```

Note that for non-inlined operators and for some compilers, using a `const` reference for argument passing avoids some data copying:

```
inline vector3 operator+(const vector3& a, const vector3& b) {
    return vector3( a.x() + b.x(), a.y()
        + b.y(), a.z() + b.z() );
}
```

1.8.4 Include Guards

All classes should have include guards surrounding the class declarations. The names of these guards should follow some simple naming convention. For example:

```
#ifndef VECTOR3H
#define VECTOR3H

class vector3 {
      .
      .
      .
};

#endif
```

This prevents problems when a header file is included more than once which is almost unavoidable in practice. Note that when VECTOR3H is already defined, the header file is still opened and one line is read. For large libraries, this file opening can dominate compilation time (Lakos, 1996). In such cases, an ugly, but effective, solution is to add a check when the include is made:

```
#ifndef VECTOR3H
#include <vector3.h>
#endif
```

1.8.5 Debugging Compiles

You should generously sprinkle asserts throughout your code. An assert is a macro that stops the program if the Boolean statement it contains is false. For example:

```
#include <assert.h>

assert( fabs( v.length() - 1 ) < 0.00001 );
```