

# Fundamentals of Computer Graphics

## Third Edition

**Peter Shirley**

NVIDIA Corporation

**Steve Marschner**

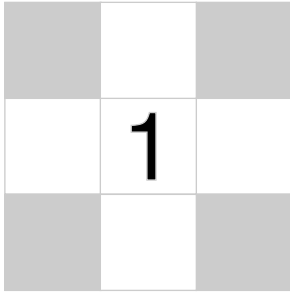
Cornell University

*with*

Michael Ashikhmin  
Michael Gleicher  
Naty Hoffman  
Garrett Johnson  
Tamara Munzner  
Erik Reinhard  
Kelvin Sung  
William B. Thompson  
Peter Willemsen  
Brian Wyvill



A K Peters  
Natick, Massachusetts



# Introduction

The term *computer graphics* describes any use of computers to create and manipulate images. This book introduces the algorithmic and mathematical tools that can be used to create all kinds of images—realistic visual effects, informative technical illustrations, or beautiful computer animations. Graphics can be two- or three-dimensional; images can be completely synthetic or can be produced by manipulating photographs. This book is about the fundamental algorithms and mathematics, especially those used to produce synthetic images of three-dimensional objects and scenes.

Actually doing computer graphics inevitably requires knowing about specific hardware, file formats, and usually a graphics API (see Section 1.3) or two. Computer graphics is a rapidly evolving field, so the specifics of that knowledge are a moving target. Therefore, in this book we do our best to avoid depending on any specific hardware or API. Readers are encouraged to supplement the text with relevant documentation for their software and 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.

API: application program interface.

## 1.1 Graphics Areas

Imposing categories on any field is dangerous, but most graphics practitioners would agree on the following major areas of computer graphics:



- **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. Animation uses modeling and rendering but adds the key issue of movement over time, which is 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, 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 into complex information 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.
- **Computational photography** is the use of computer graphics, computer vision, and image processing methods to enable new ways of photographically capturing objects, scenes, and environments.



## 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.
- **Visual 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 also use 3D modeling and animation to create synthetic environments, objects, and even characters that most viewers will never suspect are not real.
- **Animated films** use many of the same techniques that are used for visual effects, but without necessarily aiming for images that look real.
- **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 process. For example, many mechanical parts are designed in a 3D computer modeling package and 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 computed tomography (CT) dataset is composed of a large 3D rectangular array of density values. Computer graphics is used to create shaded images that help doctors extract 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 see the patterns in such data.

## 1.3 Graphics APIs

A key part of using graphics libraries is dealing with a *graphics API*. An *application program interface* (API) is a standard collection of functions to perform a set of related operations, and a graphics API is a set of functions that perform basic operations such as drawing images and 3D surfaces into windows on the screen.

Every graphics program needs to be able to use two related APIs: a graphics API for visual output and a user-interface API to get input from the user. There are currently two dominant paradigms for graphics and user-interface APIs. The first is the integrated approach, exemplified by 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 to encapsulate the system specific user-interface code.

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

## 1.4 Graphics Pipeline

Every desktop computer today has a powerful 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 tra-



ditional geometric coordinates and a fourth *homogeneous* coordinate that helps with perspective viewing. These 4D coordinates are manipulated using  $4 \times 4$  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 at which images can be generated depends strongly on the number of triangles being drawn. Because interactivity is more important in many 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).

## 1.5 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 different and incompatible ways. 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:

1. **infinity** ( $\infty$ ). This is a valid number that is larger than all other valid numbers.
2. **minus infinity** ( $-\infty$ ). This is a valid number that is smaller than all other valid numbers.
3. **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:

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

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

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

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

IEEE floating-point has two representations for zero, one that is treated as positive and one that is treated as negative. The distinction between  $-0$  and  $+0$  only occasionally matters, but it is worth keeping in mind for those occasions when it does.

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:

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

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

Some care must be taken if negative zero ( $-0$ ) might arise.



There are many numeric computations that become much simpler if the programmer takes advantage of the IEEE rules. 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* is 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  $\infty$  or NaN, but the *if* condition is still well-defined: it is false for  $a = \text{NaN}$  or  $a = -\infty$  and true for  $a = +\infty$ . With care in deciding which values are returned, often the *if* can make the right choice, with no special checks needed. This makes programs smaller, more robust, and more efficient.

## 1.6 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 two decades 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 intermediate results as needed on the fly rather than storing them.
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 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.

## 1.7 Designing and Coding Graphics Programs

Certain common strategies are often useful in graphics programming. In this section we provide some advice that you may find helpful as you implement the methods you learn about in this book.

### 1.7.1 Class Design

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. A universal design question 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; DeRose, 1989). There is little agreement on this question, which can spur hours of heated debate among graphics practitioners, but for the sake of example let’s assume we will not make the distinction.

This implies that some basic classes to 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.

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. —P.S.

I like keeping points and vectors separate because it makes code more readable and can let the compiler catch some bugs. —S.M.



- **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  $4 \times 4$  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 a special class for unit-length vectors, although I have found them more pain than they are worth. —P.S.

### 1.7.2 Float vs. 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. For data that occupies a lot of memory, such as triangle meshes, I suggest storing float data, but converting to double when data is accessed through member functions. —P.S.

### 1.7.3 Debugging Graphics Programs

If you ask around, you may find that as programmers become more experienced, they use traditional debuggers less and less. One reason for this is that using such debuggers is more awkward for complex programs than for simple programs. Another reason is that the most difficult errors are conceptual ones where the wrong thing is being implemented, and it is easy to waste large amounts of time stepping through variable values without detecting such cases. We have found several debugging strategies to be particularly useful in graphics.

I advocate doing all computations with floats until you find evidence that double precision is needed in a particular part of the code. —S.M.

#### The Scientific Method

In graphics programs there is an alternative to traditional debugging that is often very useful. The downside to it is that it is very similar to what computer programmers are taught not to do early in their careers, so you may feel “naughty” if you do it: we create an image and observe what is wrong with it. Then, we



develop a hypothesis about what is causing the problem and test it. For example, in a ray-tracing program we might have many somewhat random looking dark pixels. This is the classic “shadow acne” problem that most people run into when they write a ray tracer. Traditional debugging is not helpful here; instead, we must realize that the shadow rays are hitting the surface being shaded. We might notice that the color of the dark spots is the ambient color, so the direct lighting is what is missing. Direct lighting can be turned off in shadow, so you might hypothesize that these points are incorrectly being tagged as in shadow when they are not. To test this hypothesis, we could turn off the shadowing check and recompile. This would indicate that these are false shadow tests, and we could continue our detective work. The key reason that this method can sometimes be good practice is that we never had to spot a false value or really determine our conceptual error. Instead, we just narrowed in on our conceptual error experimentally. Typically only a few trials are needed to track things down, and this type of debugging is enjoyable.

#### Images as Coded Debugging Output

In many cases, the easiest channel by which to get debugging information out of a graphics program is the output image itself. If you want to know the value of some variable for part of a computation that runs for every pixel, you can just modify your program temporarily to copy that value directly to the output image and skip the rest of the calculations that would normally be done. For instance, if you suspect a problem with surface normals is causing a problem with shading, you can copy the normal vectors directly to the image ( $x$  goes to red,  $y$  goes to green,  $z$  goes to blue), resulting in a color-coded illustration of the vectors actually being used in your computation. Or, if you suspect a particular value is sometimes out of its valid range, make your program write bright red pixels where that happens. Other common tricks include drawing the back sides of surfaces with an obvious color (when they are not supposed to be visible), coloring the image by the ID numbers of the objects, or coloring pixels by the amount of work they took to compute.

#### Using a Debugger

There are still cases, particularly when the scientific method seems to have led to a contradiction, when there’s no substitute for observing exactly what is going on. The trouble is that graphics programs often involve many, many executions of the same code (once per pixel, for instance, or once per triangle), making it completely impractical to step through in the debugger from the start. And the most difficult bugs usually only occur for complicated inputs.



A useful approach is to “set a trap” for the bug. First, make sure your program is deterministic—run it in a single thread and make sure that all random numbers are computed from fixed seeds. Then, find out which pixel or triangle is exhibiting the bug and add a statement before the code you suspect is incorrect that will be executed only for the suspect case. For instance, if you find that pixel (126, 247) exhibits the bug, then add:

```
if  $x = 126$  and  $y = 247$  then
    print “blarg!”
```

If you set a breakpoint on the print statement, you can drop into the debugger just before the pixel you’re interested in is computed. Some debuggers have a “conditional breakpoint” feature that can achieve the same thing without modifying the code.

In the cases where the program crashes, a traditional debugger is useful for pinpointing the site of the crash. You should then start backtracking in the program, using asserts and recompiles, to find where the program went wrong. These asserts should be left in the program for potential future bugs you will add. This again means the traditional step-through process is avoided, because that would not be adding the valuable asserts to your program.

### Data Visualization for Debugging

Often it is hard to understand what your program is doing, because it computes a lot of intermediate results before it finally goes wrong. The situation is similar to a scientific experiment that measures a lot of data, and one solution is the same: make good plots and illustrations for yourself to understand what the data means. For instance, in a ray tracer you might write code to visualize ray trees so you can see what paths contributed to a pixel, or in an image resampling routine you might make plots that show all the points where samples are being taken from the input. Time spent writing code to visualize your program’s internal state is also repaid in a better understanding of its behavior when it comes time to optimize it.

A special debugging mode that uses fixed random-number seeds is useful.

I like to format debugging print statements so that the output happens to be a Matlab or Gnuplot script that makes a helpful plot.  
—S.M.

## Notes

The discussion of software engineering is influenced by the *Effective C++* series (Meyers, 1995, 1997), the *Extreme Programming* movement (Beck & Andres, 2004), and (Kernighan & Pike, 1999). The discussion of experimental debugging is based on discussions with Steve Parker.

There are a number of annual conferences related to computer graphics, including ACM SIGGRAPH and SIGGRAPH Asia, Graphics Interface, the Game



Developers Conference (GDC), Eurographics, Pacific Graphics, High Performance Graphics, the Eurographics Symposium on Rendering, and IEEE VisWeek. These can be readily found by web searches on their names.