Fundamentals of Computer Graphics

FOURTH EDITION

Steve Marschner
Peter Shirley
with
Michael Ashikhmin
Michael Gleicher
Naty Hoffman
Garrett Johnson
Tamara Munzner
Erik Reinhard
William B. Thompson
Peter Willemsen
Brian Wyvill

CRC Press
Taylor & Francis Group
AN A K PETERS BOOK

Fundamentals *of* Computer Graphics

FOURTH EDITION

Fundamentals *of* Computer Graphics

FOURTH EDITION

Steve Marschner
Cornell University

Peter Shirley
Purity, LLC

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

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20151012

International Standard Book Number-13: 978-1-4822-2941-7 (eBook - PDF)

Contents

Preface

This edition of *Fundamentals of Computer Graphics* includes substantial rewrites of the chapters on textures and graphics hardware, as well as many corrections throughout. The figures are now in color throughout the book.

The organization of the book remains substantially similar to the third edition. In our thinking, Chapters 2 through 8 constitute the "core core," taking the straight and narrow path through what is absolutely required for understanding how images get onto the screen using the complementary approaches of ray tracing and rasterization. Ray tracing is covered first, since it is the simplest way to generate images of 3D scenes, followed by the mathematical machinery required for the graphics pipeline, then the pipeline itself. After that, the "outer core" covers other topics that would commonly be included in an introductory class, such as sampling theory, texture mapping, spatial data structures, and splines. Starting with Chapter 15 is a number of contributed chapters, authored by contributors we have chosen both for their expertise and for their clear way of expressing ideas.

As we have revised this book over the years, we have endeavored to retain the informal, intuitive style of presentation that characterizes the earlier editions, while at the same time improving consistency, precision, and completeness. We hope the reader will find the result is an appealing platform for a variety of courses in computer graphics.

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 a workshop at Cornell University 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 website for this book is http://www.cs.cornell.edu/∼srm/fcg4/. We will continue to maintain a list of errata and links to courses that use the book, as well as teaching materials that match the book's style. Most of the figures in this book are in Adobe Illustrator format, and we would be happy to convert specific figures into portable formats on request. Please feel free to contact us at shirley@cs.utah.edu or srm@cs.cornell.edu.

Acknowledgments

The following people have provided helpful information, comments, or feedback about the various editions of this book: Ahmet Oğuz Akyüz, Josh Andersen, Zeferino Andrade, Kavita Bala, 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, Razen Al Harbi, Dave Hart, John Hart, John "Spike" Hughes, Helen Hu, Vicki Interrante, Wenzel Jakob, Doug James, Henrik Wann Jensen, Shi Jin, Mark Johnson, Ray Jones, Revant Kapoor, Kristin Kerr, Erum Arif Khan, Mark Kilgard, Dylan Lacewell, Mathias Lang, Philippe Laval, Marc Levoy, Howard Lo, Joann Luu, Ron Metoyer, Keith Morley, Eric Mortensen, 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, Hannah Story, Tony Tahbaz, Jan-Phillip Tiesel, Bruce Walter, Alex Williams, Amy Williams, Chris Wyman, and Kate Zebrose.

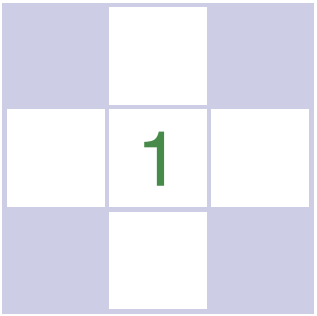
Ching-Kuang Shene and David Solomon allowed us to borrow their examples. Henrik Wann Jensen, Eric Levin, Matt Pharr, and Jason Waltman generously provided images. Brandon Mansfield helped improve the discussion of hierarchical bounding volumes for ray tracing. Philip Greenspun (philip.greenspun.com)

kindly allowed us to use his photographs. John "Spike" Hughes helped improve the discussion of sampling theory. Wenzel Jakob's *Mitsuba* renderer was invaluable in creating many figures. We are extremely thankful to J. W. Baker for helping create the cover Pete envisioned. In addition to being a talented artist, he was a great pleasure to work with personally.

Many works that were helpful in preparing this book are cited in the chapter notes. However, a few key texts that influenced the content and presentation deserve special recognition here. These include the two classic computer graphics texts from which we both learned the basics: *Computer Graphics: Principles & Practice* (Foley, Van Dam, Feiner, & Hughes, 1990) and *Computer Graphics* (Hearn & Baker, 1986). Other texts include both of Alan Watt's influential books (Watt, 1993, 1991), Hill's *Computer Graphics Using OpenGL* (Francis S. Hill, 2000), Angel's *Interactive Computer Graphics: A Top-Down Approach Using OpenGL* (Angel, 2002), Hugues Hoppe's University of Washington dissertation (Hoppe, 1994), and Rogers' two excellent graphics texts (D. F. Rogers, 1985, 1989).

We would like to especially thank Alice and Klaus Peters for encouraging Pete to write the first edition of this book and for their great skill in bringing a book to fruition. Their patience with the authors and their dedication to making their books the best they can be has been instrumental in making this book what it is. This book certainly would not exist without their extraordinary efforts.

Salt Lake City, Utah
Ithaca, New York
February 2015



1 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.

API: application program interface.

This chapter defines some basic terminology and provides some historical background, as well as information sources related to computer graphics.

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 allow 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 traditional geometric coordinates and a fourth *homogeneous* coordinate that helps with perspective viewing. These 4D coordinates are manipulated using 4×4 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 (∞). 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:

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.

$$
\begin{aligned}
+a/(+\infty) &= +0,\\
-a/(+\infty) &= -0,\\
+a/(-\infty) &= -0,\\
-a/(-\infty) &= +0.
\end{aligned}
$$

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

$$
\begin{aligned}
\infty + \infty &= +\infty,\\
\infty - \infty &= \text{NaN},\\
\infty \times \infty &= \infty,\\
\infty/\infty &= \text{NaN},\\
\infty/a &= \infty,\\
\infty/0 &= \infty,\\
0/0 &= \text{NaN}.
\end{aligned}
$$

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 ∞ or NaN, but the *if* condition is still well-defined: it is false for a = 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.

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.

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.
- 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 4×4 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:

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

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.

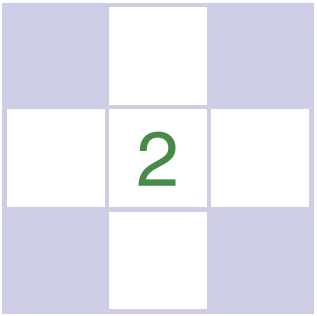
I like to format debugging print statements so that the output happens to be a Matlab or Gnuplot script that makes a helpful plot.
—S.M.

Notes

The discussion of software engineering is influenced by the *Effective C++* series (Meyers, 1995, 1997), the *Extreme Programming* movement (Beck & An-

dres, 2004), and *The Practice of Programming* (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.



2 Miscellaneous Math

Much of graphics is just translating math directly into code. The cleaner the math, the cleaner the resulting code; so much of this book concentrates on using just the right math for the job. This chapter reviews various tools from high school and college mathematics and is designed to be used more as a reference than as a tutorial. It may appear to be a hodge-podge of topics and indeed it is; each topic is chosen because it is a bit unusual in "standard" math curricula, because it is of central importance in graphics, or because it is not typically treated from a geometric standpoint. In addition to establishing a review of the notation used in the book, the chapter also emphasizes a few points that are sometimes skipped in the standard undergraduate curricula, such as barycentric coordinates on triangles. This chapter is not intended to be a rigorous treatment of the material; instead intuition and geometric interpretation are emphasized. A discussion of linear algebra is deferred until Chapter 5 just before transformation matrices are discussed. Readers are encouraged to skim this chapter to familiarize themselves with the topics covered and to refer back to it as needed. The exercises at the end of the chapter may be useful in determining which topics need a refresher.

2.1 Sets and Mappings

Mappings, also called *functions*, are basic to mathematics and programming. Like a function in a program, a mapping in math takes an argument of one *type* and

maps it to (returns) an object of a particular type. In a program we say "type"; in math we would identify the set. When we have an object that is a member of a set, we use the $\in$ symbol. For example,

$$a \in \mathbf{S},$$

can be read "a is a member of set $\mathbf{S}$." Given any two sets $\mathbf{A}$ and $\mathbf{B}$, we can create a third set by taking the *Cartesian product* of the two sets, denoted $\mathbf{A} \times \mathbf{B}$. This set $\mathbf{A} \times \mathbf{B}$ is composed of all possible ordered pairs (a, b) where $a \in \mathbf{A}$ and $b \in \mathbf{B}$. As a shorthand, we use the notation $\mathbf{A}^2$ to denote $\mathbf{A} \times \mathbf{A}$. We can extend the Cartesian product to create a set of all possible ordered triples from three sets and so on for arbitrarily long ordered tuples from arbitrarily many sets.

Common sets of interest include

- $\mathbb{R}$—the real numbers;
- $\mathbb{R}^+$—the nonnegative real numbers (includes zero);
- $\mathbb{R}^2$—the ordered pairs in the real 2D plane;
- $\mathbb{R}^n$—the points in n-dimensional Cartesian space;
- $\mathbb{Z}$—the integers;
- S^2—the set of 3D points (points in $\mathbb{R}^3$) on the unit sphere.

Note that although S^2 is composed of points embedded in three-dimensional space, they are on a surface that can be parameterized with two variables, so it can be thought of as a 2D set. Notation for mappings uses the arrow and a colon, for example:

$$f : \mathbb{R} \mapsto \mathbb{Z},$$

which you can read as "There is a function called f that takes a real number as input and maps it to an integer." Here, the set that comes before the arrow is called the *domain* of the function, and the set on the right-hand side is called the *target*. Computer programmers might be more comfortable with the following equivalent language: "There is a function called f which has one real argument and returns an integer." In other words, the set notation above is equivalent to the common programming notation:

$$\text{integer } f(\text{real}) \quad \leftarrow \text{equivalent} \rightarrow \quad f : \mathbb{R} \mapsto \mathbb{Z}.$$

So the colon-arrow notation can be thought of as a programming syntax. It's that simple.

The point $f(a)$ is called the *image* of a, and the image of a set A (a subset of the domain) is the subset of the target that contains the images of all points in A. The image of the whole domain is called the *range* of the function.

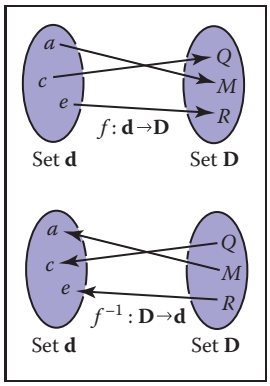


Figure 2.1. A bijection f and the inverse function f^{-1}. Note that f^{-1} is also a bijection.

2.1.1 Inverse Mappings

If we have a function $f : \mathbf{A} \mapsto \mathbf{B}$, there may exist an *inverse function* $f^{-1} : \mathbf{B} \mapsto \mathbf{A}$, which is defined by the rule $f^{-1}(b) = a$ where $b = f(a)$. This definition only works if every $b \in \mathbf{B}$ is an image of some point under f (that is, the range equals the target) and if there is only one such point (that is, there is only one a for which $f(a) = b$). Such mappings or functions are called *bijections*. A bijection maps every $a \in \mathbf{A}$ to a unique $b \in \mathbf{B}$, and for every $b \in \mathbf{B}$, there is exactly one $a \in \mathbf{A}$ such that $f(a) = b$ (Figure 2.1). A bijection between a group of riders and horses indicates that everybody rides a single horse, and every horse is ridden. The two functions would be *rider(horse)* and *horse(rider)*. These are inverse functions of each other. Functions that are not bijections have no inverse (Figure 2.2).

An example of a bijection is $f : \mathbb{R} \mapsto \mathbb{R}$, with $f(x) = x^3$. The inverse function is $f^{-1}(x) = \sqrt[3]{x}$. This example shows that the standard notation can be somewhat awkward because x is used as a dummy variable in both f and f^{-1}. It is sometimes more intuitive to use different dummy variables, with $y = f(x)$ and $x = f^{-1}(y)$. This yields the more intuitive $y = x^3$ and $x = \sqrt[3]{y}$. An example of a function that does not have an inverse is $sqr : \mathbb{R} \mapsto \mathbb{R}$, where $sqr(x) = x^2$. This is true for two reasons: first $x^2 = (-x)^2$, and second no members of the domain map to the negative portions of the target. Note that we can define an inverse if we restrict the domain and range to $\mathbb{R}^+$. Then $\sqrt{x}$ is a valid inverse.

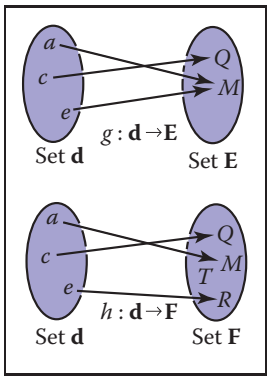


Figure 2.2. The function g does not have an inverse because two elements of **d** map to the same element of **E**. The function h has no inverse because element T of **F** has no element of **d** mapped to it.

2.1.2 Intervals

Often we would like to specify that a function deals with real numbers that are restricted in value. One such constraint is to specify an *interval*. An example of an interval is the real numbers between zero and one, not including zero or one. We denote this $(0, 1)$. Because it does not include its endpoints, this is referred to as an *open interval*. The corresponding *closed interval*, which does contain its endpoints, is denoted with square brackets: $[0, 1]$. This notation can be mixed, i.e., $[0, 1)$ includes zero but not one. When writing an interval $[a, b]$, we assume that $a \leq b$. The three common ways to represent an interval are shown in Figure 2.3. The Cartesian products of intervals are often used. For example, to indicate that a point $\mathbf{x}$ is in the unit cube in 3D, we say $\mathbf{x} \in [0, 1]^3$.

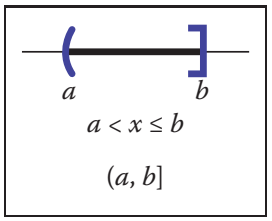


Figure 2.3. Three equivalent ways to denote the interval from a to b that includes b but not a.

Intervals are particularly useful in conjunction with set operations: *intersection*, *union*, and *difference*. For example, the intersection of two intervals is the set of points they have in common. The symbol $\cap$ is used for intersection. For example, $[3, 5) \cap [4, 6] = [4, 5)$. For unions, the symbol $\cup$ is used to denote points in either interval. For example, $[3, 5) \cup [4, 6] = [3, 6]$. Unlike the first two operators, the difference operator produces different results depending on argument order.

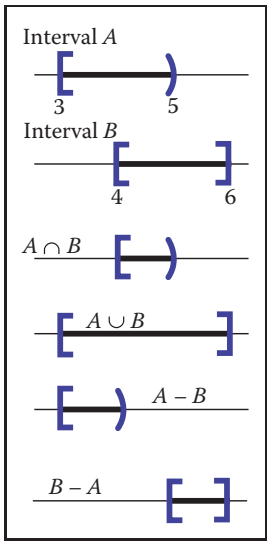


Figure 2.4. Interval operations on [3,5) and [4,6].

The minus sign is used for the difference operator, which returns the points in the left interval that are not also in the right. For example, $[3,5) - [4,6] = [3,4)$ and $[4,6] - [3,5) = [5,6]$. These operations are particularly easy to visualize using interval diagrams (Figure 2.4).

2.1.3 Logarithms

Although not as prevalent today as they were before calculators, *logarithms* are often useful in problems where equations with exponential terms arise. By definition, every logarithm has a *base* a. The "log base a" of x is written $\log_a x$ and is defined as "the exponent to which a must be raised to get x," i.e.,

$$y = \log_a x \quad \Leftrightarrow \quad a^y = x.$$

Note that the logarithm base a and the function that raises a to a power are inverses of each other. This basic definition has several consequences:

$$\begin{aligned} a^{\log_a(x)} &= x; \\ \log_a(a^x) &= x; \\ \log_a(xy) &= \log_a x + \log_a y; \\ \log_a(x/y) &= \log_a x - \log_a y; \\ \log_a x &= \log_a b \ \log_b x. \end{aligned}$$

When we apply calculus to logarithms, the special number $e = 2.718\ldots$ often turns up. The logarithm with base e is called the *natural logarithm*. We adopt the common shorthand $\ln$ to denote it:

$$\ln x \equiv \log_e x.$$

Note that the "$\equiv$" symbol can be read "is equivalent by definition." Like π, the special number e arises in a remarkable number of contexts. Many fields use a particular base in addition to e for manipulations and omit the base in their notation, i.e., $\log x$. For example, astronomers often use base 10 and theoretical computer scientists often use base 2. Because computer graphics borrows technology from many fields we will avoid this shorthand.

The derivatives of logarithms and exponents illuminate why the natural logarithm is "natural":

$$\begin{aligned} \frac{d}{dx} \log_a x &= \frac{1}{x \ln a}; \\ \frac{d}{dx} a^x &= a^x \ln a. \end{aligned}$$

The constant multipliers above are unity only for $a = e$.

2.2 Solving Quadratic Equations

A *quadratic equation* has the form

$$Ax^2 + Bx + C = 0,$$

where x is a real unknown, and A, B, and C are known constants. If you think of a 2D xy plot with $y = Ax^2 + Bx + C$, the solution is just whatever x values are "zero crossings" in y. Because $y = Ax^2 + Bx + C$ is a parabola, there will be zero, one, or two real solutions depending on whether the parabola misses, grazes, or hits the x-axis (Figure 2.5).

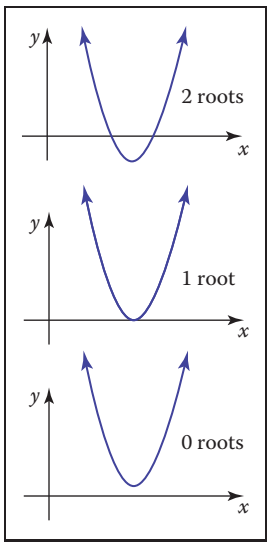


Figure 2.5. The geometric interpretation of the roots of a quadratic equation is the intersection points of a parabola with the x-axis.

To solve the quadratic equation analytically, we first divide by A:

$$x^2 + \frac{B}{A}x + \frac{C}{A} = 0.$$

Then we "complete the square" to group terms:

$$\left(x + \frac{B}{2A}\right)^2 - \frac{B^2}{4A^2} + \frac{C}{A} = 0.$$

Moving the constant portion to the right-hand side and taking the square root gives

$$x + \frac{B}{2A} = \pm\sqrt{\frac{B^2}{4A^2} - \frac{C}{A}}.$$

Subtracting $B/(2A)$ from both sides and grouping terms with the denominator $2A$ gives the familiar form:[1]

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}. \tag{2.1}$$

Here the "$\pm$" symbol means there are two solutions, one with a plus sign and one with a minus sign. Thus 3 ± 1 equals "two or four." Note that the term that determines the number of real solutions is

$$D \equiv B^2 - 4AC,$$

which is called the *discriminant* of the quadratic equation. If $D > 0$, there are two real solutions (also called *roots*). If $D = 0$, there is one real solution (a "double" root). If $D < 0$, there are no real solutions.

For example, the roots of $2x^2 + 6x + 4 = 0$ are $x = -1$ and $x = -2$, and the equation $x^2 + x + 1$ has no real solutions. The discriminants of these equations are $D = 4$ and $D = -3$, respectively, so we expect the number of solutions given. In programs, it is usually a good idea to evaluate D first and return "no roots" without taking the square root if D is negative.

[1] A robust implementation will use the equivalent expression $2C/(-B \mp \sqrt{B^2 - 4AC})$ to compute one of the roots, depending on the sign of B (Exercise 7).

2.3 Trigonometry

In graphics we use basic trigonometry in many contexts. Usually, it is nothing too fancy, and it often helps to remember the basic definitions.

2.3.1 Angles

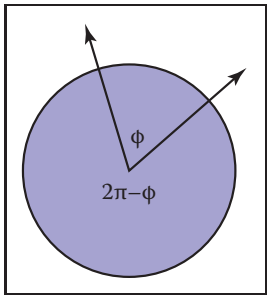


Figure 2.6. Two half-lines cut the unit circle into two arcs. The length of either arc is a valid angle "between" the two half-lines. Either we can use the convention that the smaller length is the angle, or that the two half-lines are specified in a certain order and the arc that determines angle ϕ is the one swept out counterclockwise from the first to the second half-line.

Although we take angles somewhat for granted, we should return to their definition so we can extend the idea of the angle onto the sphere. An angle is formed between two half-lines (infinite rays stemming from an origin) or directions, and some convention must be used to decide between the two possibilities for the angle created between them as shown in Figure 2.6. An *angle* is defined by the length of the arc segment it cuts out on the unit circle. A common convention is that the smaller arc length is used, and the sign of the angle is determined by the order in which the two half-lines are specified. Using that convention, all angles are in the range $[-\pi, \pi]$.

Each of these angles is *the length of the arc of the unit circle that is "cut" by the two directions*. Because the perimeter of the unit circle is 2π, the two possible angles sum to 2π. The unit of these arc lengths is *radians*. Another common unit is degrees, where the perimeter of the circle is 360 degrees. Thus, an angle that is π radians is 180 degrees, usually denoted $180°$. The conversion between degrees and radians is

$$\text{degrees} = \frac{180}{\pi} \text{ radians};$$

$$\text{radians} = \frac{\pi}{180} \text{ degrees}.$$

2.3.2 Trigonometric Functions

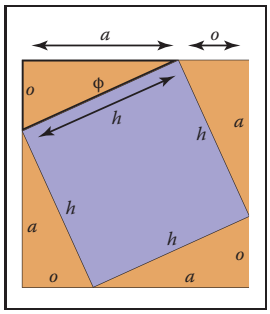


Figure 2.7. A geometric demonstration of the Pythagorean theorem.

Given a right triangle with sides of length a, o, and h, where h is the length of the longest side (which is always opposite the right angle), or *hypotenuse*, an important relation is described by the *Pythagorean theorem*:

$$a^2 + o^2 = h^2.$$

You can see that this is true from Figure 2.7, where the big square has area $(a+o)^2$, the four triangles have the combined area $2ao$, and the center square has area h^2.

Because the triangles and inner square subdivide the larger square evenly, we have $2ao + h^2 = (a + o)^2$, which is easily manipulated to the form above.

We define *sine* and *cosine* of ϕ, as well as the other ratio-based trigonometric expressions:

$$\begin{aligned}\sin\phi &\equiv o/h;\\ \csc\phi &\equiv h/o;\\ \cos\phi &\equiv a/h;\\ \sec\phi &\equiv h/a;\\ \tan\phi &\equiv o/a;\\ \cot\phi &\equiv a/o.\end{aligned}$$

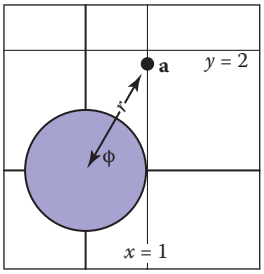


Figure 2.8. Polar coordinates for the point $(x_a, y_a) = (1,\sqrt{3})$ is $(r_a, \phi_a) = (2, \pi/3)$.

These definitions allow us to set up *polar coordinates*, where a point is coded as a distance from the origin and a signed angle relative to the positive x-axis (Figure 2.8). Note the convention that angles are in the range $\phi \in (-\pi, \pi]$, and that the positive angles are counterclockwise from the positive x-axis. This convention that counterclockwise maps to positive numbers is arbitrary, but it is used in many contexts in graphics so it is worth committing to memory.

Trigonometric functions are periodic and can take any angle as an argument. For example, $\sin(A) = \sin(A + 2\pi)$. This means the functions are not invertible when considered with the domain $\mathbb{R}$. This problem is avoided by restricting the range of standard inverse functions, and this is done in a standard way in almost all modern math libraries (e.g., (Plauger, 1991)). The domains and ranges are:

$$\begin{aligned}&\text{asin} : [-1, 1] \mapsto [-\pi/2, \pi/2];\\ &\text{acos} : [-1, 1] \mapsto [0, \pi];\\ &\text{atan} : \mathbb{R} \mapsto [-\pi/2, \pi/2];\\ &\text{atan2} : \mathbb{R}^2 \mapsto [-\pi, \pi].\end{aligned} \tag{2.2}$$

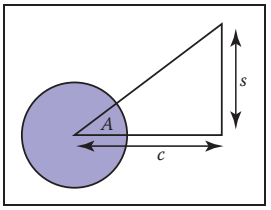


Figure 2.9. The function atan2(*s*,*c*) returns the angle *A* and is often very useful in graphics.

The last function, $\text{atan2}(s, c)$ is often very useful. It takes an s value proportional to $\sin A$ and a c value that scales $\cos A$ by the same factor and returns A. The factor is assumed to be positive. One way to think of this is that it returns the angle of a 2D Cartesian point (s, c) in polar coordinates (Figure 2.9).

2.3.3 Useful Identities

This section lists without derivation a variety of useful trigonometric identities.

Shifting identities:

$$\begin{aligned}\sin(-A) &= -\sin A\\ \cos(-A) &= \cos A\\ \tan(-A) &= -\tan A\\ \sin(\pi/2 - A) &= \cos A\\ \cos(\pi/2 - A) &= \sin A\\ \tan(\pi/2 - A) &= \cot A\end{aligned}$$

Pythagorean identities:

$$\sin^2 A + \cos^2 A = 1$$
$$\sec^2 A - \tan^2 A = 1$$
$$\csc^2 A - \cot^2 A = 1$$

Addition and subtraction identities:

$$\sin(A+B) = \sin A \cos B + \sin B \cos A$$
$$\sin(A-B) = \sin A \cos B - \sin B \cos A$$
$$\sin(2A) = 2 \sin A \cos A$$
$$\cos(A+B) = \cos A \cos B - \sin A \sin B$$
$$\cos(A-B) = \cos A \cos B + \sin A \sin B$$
$$\cos(2A) = \cos^2 A - \sin^2 A$$
$$\tan(A+B) = \frac{\tan A + \tan B}{1 - \tan A \tan B}$$
$$\tan(A-B) = \frac{\tan A - \tan B}{1 + \tan A \tan B}$$
$$\tan(2A) = \frac{2 \tan A}{1 - \tan^2 A}$$

Half-angle identities:

$$\sin^2(A/2) = (1 - \cos A)/2$$
$$\cos^2(A/2) = (1 + \cos A)/2$$

Product identities:

$$\sin A \sin B = -(\cos(A+B) - \cos(A-B))/2$$
$$\sin A \cos B = \quad (\sin(A+B) + \sin(A-B))/2$$
$$\cos A \cos B = \quad (\cos(A+B) + \cos(A-B))/2$$

The following identities are for arbitrary triangles with side lengths a, b, and c, each with an angle opposite it given by A, B, C, respectively (Figure 2.10):

$$\frac{\sin A}{a} = \frac{\sin B}{b} = \frac{\sin C}{c} \qquad \text{(Law of sines)}$$

$$c^2 = a^2 + b^2 - 2ab \cos C \qquad \text{(Law of cosines)}$$

$$\frac{a+b}{a-b} = \frac{\tan\left(\frac{A+B}{2}\right)}{\tan\left(\frac{A-B}{2}\right)} \qquad \text{(Law of tangents)}$$

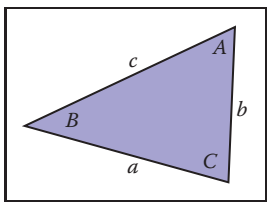


Figure 2.10. Geometry for triangle laws.

The area of a triangle can also be computed in terms of these side lengths:

$$\text{triangle area} = \frac{1}{4}\sqrt{(a+b+c)(-a+b+c)(a-b+c)(a+b-c)}.$$

2.4 Vectors

A *vector* describes a length and a direction. It can be usefully represented by an arrow. Two vectors are equal if they have the same length and direction even if we think of them as being located in different places (Figure 2.11). As much as possible, you should think of a vector as an arrow and not as coordinates or numbers. At some point we will have to represent vectors as numbers in our programs, but even in code they should be manipulated as objects and only the low-level vector operations should know about their numeric representation (DeRose, 1989). Vectors will be represented as bold characters, e.g., $\mathbf{a}$. A vector's length is denoted $\|\mathbf{a}\|$. A *unit vector* is any vector whose length is one. The *zero vector* is the vector of zero length. The direction of the zero vector is undefined.

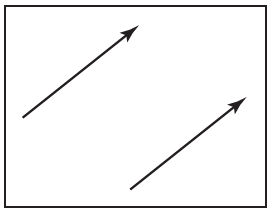


Figure 2.11. These two vectors are the same because they have the same length and direction.

Vectors can be used to represent many different things. For example, they can be used to store an *offset*, also called a *displacement*. If we know "the treasure is buried two paces east and three paces north of the secret meeting place," then we know the offset, but we don't know where to start. Vectors can also be used to store a *location*, another word for *position* or *point*. Locations can be represented as a displacement from another location. Usually there is some understood *origin* location from which all other locations are stored as offsets. Note that locations are not vectors. As we shall discuss, you can add two vectors. However, it usually does not make sense to add two locations unless it is an intermediate operation when computing weighted averages of a location (Goldman, 1985). Adding two offsets does make sense, so that is one reason why offsets are vectors. But this emphasizes that a location is not an offset; it is an offset from a specific origin location. The offset by itself is not the location.

2.4.1 Vector Operations

Vectors have most of the usual arithmetic operations that we associate with real numbers. Two vectors are equal if and only if they have the same length and direction. Two vectors are added according to the *parallelogram rule*. This rule states that the sum of two vectors is found by placing the tail of either vector against the head of the other (Figure 2.12). The sum vector is the vector that "completes the triangle" started by the two vectors. The parallelogram is formed by taking the sum in either order. This emphasizes that vector addition is commutative:

$$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}.$$

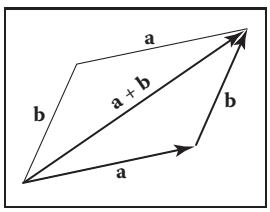


Figure 2.12. Two vectors are added by arranging them head to tail. This can be done in either order.

Note that the parallelogram rule just formalizes our intuition about displacements. Think of walking along one vector, tail to head, and then walking along the other.

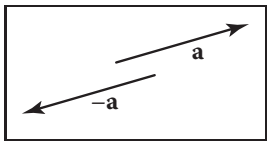


Figure 2.13. The vector –**a** has the same length but opposite direction of the vector **a**.

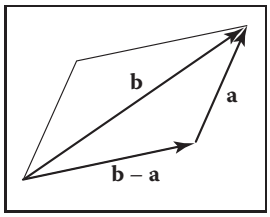


Figure 2.14. Vector subtraction is just vector addition with a reversal of the second argument.

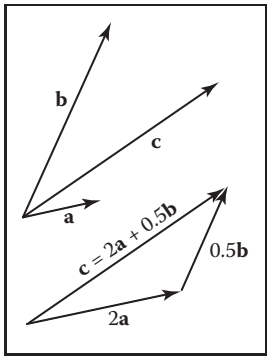


Figure 2.15. Any 2D vector **c** is a weighted sum of any two nonparallel 2D vectors **a** and **b**.

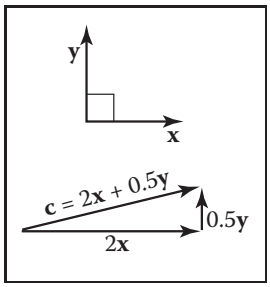


Figure 2.16. A 2D Cartesian basis for vectors.

The net displacement is just the parallelogram diagonal. You can also create a *unary minus* for a vector: $-\mathbf{a}$ (Figure 2.13) is a vector with the same length as $\mathbf{a}$ but opposite direction. This allows us to also define subtraction:

$$\mathbf{b} - \mathbf{a} \equiv -\mathbf{a} + \mathbf{b}.$$

You can visualize vector subtraction with a parallelogram (Figure 2.14). We can write

$$\mathbf{a} + (\mathbf{b} - \mathbf{a}) = \mathbf{b}.$$

Vectors can also be multiplied. In fact, there are several kinds of products involving vectors. First, we can *scale* the vector by multiplying it by a real number k. This just multiplies the vector's length without changing its direction. For example, $3.5\mathbf{a}$ is a vector in the same direction as $\mathbf{a}$ but it is 3.5 times as long as $\mathbf{a}$. We discuss two products involving two vectors, the dot product and the cross product, later in this section, and a product involving three vectors, the determinant, in Chapter 5.

2.4.2 Cartesian Coordinates of a Vector

A 2D vector can be written as a combination of any two nonzero vectors which are not parallel. This property of the two vectors is called *linear independence*. Two linearly independent vectors form a 2D *basis*, and the vectors are thus referred to as *basis vectors*. For example, a vector $\mathbf{c}$ may be expressed as a combination of two basis vectors $\mathbf{a}$ and $\mathbf{b}$ (Figure 2.15):

$$\mathbf{c} = a_c\mathbf{a} + b_c\mathbf{b}. \tag{2.3}$$

Note that the weights a_c and b_c are unique. Bases are especially useful if the two vectors are *orthogonal*, i.e., they are at right angles to each other. It is even more useful if they are also unit vectors in which case they are *orthonormal*. If we assume two such "special" vectors $\mathbf{x}$ and $\mathbf{y}$ are known to us, then we can use them to represent all other vectors in a *Cartesian* coordinate system, where each vector is represented as two real numbers. For example, a vector $\mathbf{a}$ might be represented as

$$\mathbf{a} = x_a\mathbf{x} + y_a\mathbf{y},$$

where x_a and y_a are the real Cartesian coordinates of the 2D vector $\mathbf{a}$ (Figure 2.16). Note that this is not really any different conceptually from Equation (2.3), where the basis vectors were not orthonormal. But there are several advantages to a Cartesian coordinate system. For instance, by the Pythagorean theorem, the length of $\mathbf{a}$ is

$$\|\mathbf{a}\| = \sqrt{x_a^2 + y_a^2}.$$

It is also simple to compute dot products, cross products, and coordinates of vectors in Cartesian systems, as we'll see in the following sections.

By convention we write the coordinates of $\mathbf{a}$ either as an ordered pair (x_a, y_a) or a column matrix:

$$\mathbf{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix}.$$

The form we use will depend on typographic convenience. We will also occasionally write the vector as a row matrix, which we will indicate as $\mathbf{a}^{\mathrm{T}}$:

$$\mathbf{a}^{\mathrm{T}} = \begin{bmatrix} x_a & y_a \end{bmatrix}.$$

We can also represent 3D, 4D, etc., vectors in Cartesian coordinates. For the 3D case, we use a basis vector $\mathbf{z}$ that is orthogonal to both $\mathbf{x}$ and $\mathbf{y}$.

2.4.3 Dot Product

The simplest way to multiply two vectors is the *dot* product. The dot product of $\mathbf{a}$ and $\mathbf{b}$ is denoted $\mathbf{a} \cdot \mathbf{b}$ and is often called the *scalar product* because it returns a scalar. The dot product returns a value related to its arguments' lengths and the angle ϕ between them (Figure 2.17):

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \, \cos\phi, \tag{2.4}$$

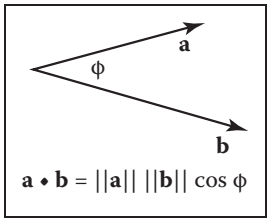


Figure 2.17. The dot product is related to length and angle and is one of the most important formulas in graphics.

The most common use of the dot product in graphics programs is to compute the cosine of the angle between two vectors.

The dot product can also be used to find the *projection* of one vector onto another. This is the length $\mathbf{a}{\rightarrow}\mathbf{b}$ of a vector $\mathbf{a}$ that is projected at right angles onto a vector $\mathbf{b}$ (Figure 2.18):

$$\mathbf{a}{\rightarrow}\mathbf{b} = \|\mathbf{a}\| \, \cos\phi = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|}. \tag{2.5}$$

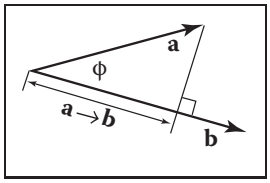


Figure 2.18. The projection of $\mathbf{a}$ onto $\mathbf{b}$ is a length found by Equation (2.5).

The dot product obeys the familiar associative and distributive properties we have in real arithmetic:

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= \mathbf{b} \cdot \mathbf{a}, \\ \mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) &= \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}, \\ (k\mathbf{a}) \cdot \mathbf{b} &= \mathbf{a} \cdot (k\mathbf{b}) = k\mathbf{a} \cdot \mathbf{b}. \end{aligned} \tag{2.6}$$

If 2D vectors $\mathbf{a}$ and $\mathbf{b}$ are expressed in Cartesian coordinates, we can take advantage of $\mathbf{x} \cdot \mathbf{x} = \mathbf{y} \cdot \mathbf{y} = 1$ and $\mathbf{x} \cdot \mathbf{y} = 0$ to derive that their dot product

is

$$\begin{aligned}\mathbf{a}\cdot\mathbf{b} &= (x_a\mathbf{x}+y_a\mathbf{y})\cdot(x_b\mathbf{x}+y_b\mathbf{y})\\ &= x_ax_b(\mathbf{x}\cdot\mathbf{x})+x_ay_b(\mathbf{x}\cdot\mathbf{y})+x_by_a(\mathbf{y}\cdot\mathbf{x})+y_ay_b(\mathbf{y}\cdot\mathbf{y})\\ &= x_ax_b+y_ay_b.\end{aligned}$$

Similarly in 3D we can find

$$\mathbf{a}\cdot\mathbf{b} = x_ax_b+y_ay_b+z_az_b.$$

2.4.4 Cross Product

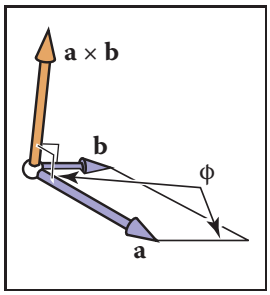


Figure 2.19. The cross product $\mathbf{a}\times\mathbf{b}$ is a 3D vector perpendicular to both 3D vectors $\mathbf{a}$ and $\mathbf{b}$, and its length is equal to the area of the parallelogram shown.

The cross product $\mathbf{a}\times\mathbf{b}$ is usually used only for three-dimensional vectors; generalized cross products are discussed in references given in the chapter notes. The cross product returns a 3D vector that is perpendicular to the two arguments of the cross product. The length of the resulting vector is related to $\sin\phi$:

$$\|\mathbf{a}\times\mathbf{b}\| = \|\mathbf{a}\|\,\|\mathbf{b}\|\sin\phi.$$

The magnitude $\|\mathbf{a}\times\mathbf{b}\|$ is equal to the area of the parallelogram formed by vectors $\mathbf{a}$ and $\mathbf{b}$. In addition, $\mathbf{a}\times\mathbf{b}$ is perpendicular to both $\mathbf{a}$ and $\mathbf{b}$ (Figure 2.19). Note that there are only two possible directions for such a vector. By definition, the vectors in the direction of the x-, y- and z-axes are given by

$$\begin{aligned}\mathbf{x} &= (1,0,0),\\ \mathbf{y} &= (0,1,0),\\ \mathbf{z} &= (0,0,1),\end{aligned}$$

and we set as a convention that $\mathbf{x}\times\mathbf{y}$ must be in the plus or minus $\mathbf{z}$ direction. The choice is somewhat arbitrary, but it is standard to assume that

$$\mathbf{z} = \mathbf{x}\times\mathbf{y}.$$

All possible permutations of the three Cartesian unit vectors are

$$\begin{aligned}\mathbf{x}\times\mathbf{y} &= +\mathbf{z},\\ \mathbf{y}\times\mathbf{x} &= -\mathbf{z},\\ \mathbf{y}\times\mathbf{z} &= +\mathbf{x},\\ \mathbf{z}\times\mathbf{y} &= -\mathbf{x},\\ \mathbf{z}\times\mathbf{x} &= +\mathbf{y},\\ \mathbf{x}\times\mathbf{z} &= -\mathbf{y}.\end{aligned}$$

Because of the $\sin\phi$ property, we also know that a vector cross itself is the zero-vector, so $\mathbf{x} \times \mathbf{x} = \mathbf{0}$ and so on. Note that the cross product is *not* commutative, i.e., $\mathbf{x} \times \mathbf{y} \neq \mathbf{y} \times \mathbf{x}$. The careful observer will note that the above discussion does not allow us to draw an unambiguous picture of how the Cartesian axes relate. More specifically, if we put $\mathbf{x}$ and $\mathbf{y}$ on a sidewalk, with $\mathbf{x}$ pointing east and $\mathbf{y}$ pointing north, then does $\mathbf{z}$ point up to the sky or into the ground? The usual convention is to have $\mathbf{z}$ point to the sky. This is known as a *right-handed* coordinate system. This name comes from the memory scheme of "grabbing" $\mathbf{x}$ with your *right* palm and fingers and rotating it toward $\mathbf{y}$. The vector $\mathbf{z}$ should align with your thumb. This is illustrated in Figure 2.20.

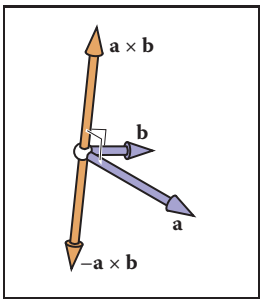


Figure 2.20. The "right-hand rule" for cross products. Imagine placing the base of your right palm where **a** and **b** join at their tails, and pushing the arrow of **a** toward **b**. Your extended right thumb should point toward **a** × **b**.

The cross product has the nice property that

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c},$$

and

$$\mathbf{a} \times (k\mathbf{b}) = k(\mathbf{a} \times \mathbf{b}).$$

However, a consequence of the right-hand rule is

$$\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a}).$$

In Cartesian coordinates, we can use an explicit expansion to compute the cross product:

$$\begin{aligned}
\mathbf{a} \times \mathbf{b} &= (x_a\mathbf{x} + y_a\mathbf{y} + z_a\mathbf{z}) \times (x_b\mathbf{x} + y_b\mathbf{y} + z_b\mathbf{z}) \\
&= x_a x_b \mathbf{x} \times \mathbf{x} + x_a y_b \mathbf{x} \times \mathbf{y} + x_a z_b \mathbf{x} \times \mathbf{z} \\
&\quad + y_a x_b \mathbf{y} \times \mathbf{x} + y_a y_b \mathbf{y} \times \mathbf{y} + y_a z_b \mathbf{y} \times \mathbf{z} \\
&\quad + z_a x_b \mathbf{z} \times \mathbf{x} + z_a y_b \mathbf{z} \times \mathbf{y} + z_a z_b \mathbf{z} \times \mathbf{z} \\
&= (y_a z_b - z_a y_b)\mathbf{x} + (z_a x_b - x_a z_b)\mathbf{y} + (x_a y_b - y_a x_b)\mathbf{z}.
\end{aligned} \tag{2.7}$$

So, in coordinate form,

$$\mathbf{a} \times \mathbf{b} = (y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b). \tag{2.8}$$

2.4.5 Orthonormal Bases and Coordinate Frames

Managing coordinate systems is one of the core tasks of almost any graphics program; the key to this is managing *orthonormal bases*. Any set of two 2D vectors $\mathbf{u}$ and $\mathbf{v}$ form an orthonormal basis provided that they are orthogonal (at right angles) and are each of unit length. Thus,

$$\|\mathbf{u}\| = \|\mathbf{v}\| = 1,$$

and

$$\mathbf{u} \cdot \mathbf{v} = 0.$$

In 3D, three vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ form an orthonormal basis if

$$\|\mathbf{u}\| = \|\mathbf{v}\| = \|\mathbf{w}\| = 1,$$

and

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{u} = 0.$$

This orthonormal basis is *right-handed* provided

$$\mathbf{w} = \mathbf{u} \times \mathbf{v},$$

and otherwise it is left-handed.

Note that the Cartesian canonical orthonormal basis is just one of infinitely many possible orthonormal bases. What makes it special is that it and its implicit origin location are used for low-level representation within a program. Thus, the vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ are never explicitly stored and neither is the canonical origin location $\mathbf{o}$. The global model is typically stored in this canonical coordinate system, and it is thus often called the *global coordinate system*. However, if we want to use another coordinate system with origin $\mathbf{p}$ and orthonormal basis vectors

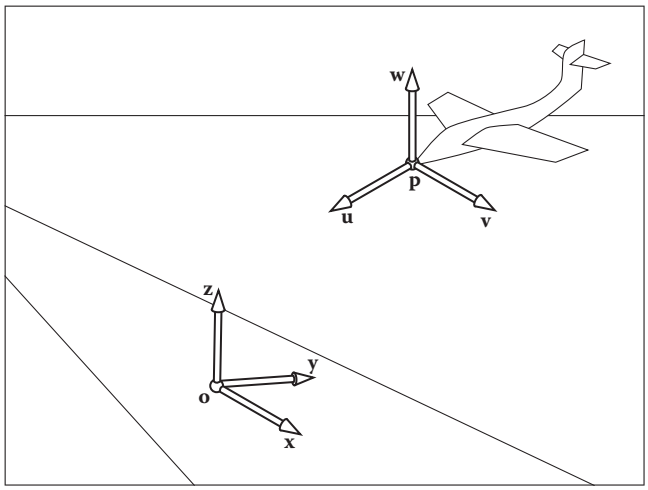


Figure 2.21. There is always a master or "canonical" coordinate system with origin **o** and orthonormal basis **x, y**, and **z**. This coordinate system is usually defined to be aligned to the global model and is thus often called the "global" or "world" coordinate system. This origin and basis vectors are never stored explicitly. All other vectors and locations are stored with coordinates that relate them to the global frame. The coordinate system associated with the plane are explicitly stored in terms of global coordinates.

$\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$, then we *do* store those vectors explicitly. Such a system is called a *frame of reference* or *coordinate frame*. For example, in a flight simulator, we might want to maintain a coordinate system with the origin at the nose of the plane, and the orthonormal basis aligned with the airplane. Simultaneously, we would have the master canonical coordinate system (Figure 2.21). The coordinate system associated with a particular object, such as the plane, is usually called a *local coordinate system*.

At a low level, the local frame is stored in canonical coordinates. For example, if $\mathbf{u}$ has coordinates (x_u, y_u, z_u),

$$\mathbf{u} = x_u\mathbf{x} + y_u\mathbf{y} + z_u\mathbf{z}.$$

A location implicitly includes an offset from the canonical origin:

$$\mathbf{p} = \mathbf{o} + x_p\mathbf{x} + y_p\mathbf{y} + z_p\mathbf{z},$$

where (x_p, y_p, z_p) are the coordinates of $\mathbf{p}$.

Note that if we store a vector $\mathbf{a}$ with respect to the $\mathbf{u}$-$\mathbf{v}$-$\mathbf{w}$ frame, we store a triple (u_a, v_a, w_a) which we can interpret geometrically as

$$\mathbf{a} = u_a\mathbf{u} + v_a\mathbf{v} + w_a\mathbf{w}.$$

To get the canonical coordinates of a vector $\mathbf{a}$ stored in the $\mathbf{u}$-$\mathbf{v}$-$\mathbf{w}$ coordinate system, simply recall that $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are themselves stored in terms of Cartesian coordinates, so the expression $u_a\mathbf{u} + v_a\mathbf{v} + w_a\mathbf{w}$ is already in Cartesian coordinates if evaluated explicitly. To get the $\mathbf{u}$-$\mathbf{v}$-$\mathbf{w}$ coordinates of a vector $\mathbf{b}$ stored in the canonical coordinate system, we can use dot products:

$$u_b = \mathbf{u} \cdot \mathbf{b}; \quad v_b = \mathbf{v} \cdot \mathbf{b}; \quad w_b = \mathbf{w} \cdot \mathbf{b}.$$

This works because we know that for *some* u_b, v_b, and w_b,

$$u_b\mathbf{u} + v_b\mathbf{v} + w_b\mathbf{w} = \mathbf{b},$$

and the dot product isolates the u_b coordinate:

$$\begin{aligned}\mathbf{u} \cdot \mathbf{b} &= u_b(\mathbf{u} \cdot \mathbf{u}) + v_b(\mathbf{u} \cdot \mathbf{v}) + w_b(\mathbf{u} \cdot \mathbf{w}) \\ &= u_b.\end{aligned}$$

This works because $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are orthonormal.

Using matrices to manage changes of coordinate systems is discussed in Sections 6.2.1 and 6.5.

2.4.6 Constructing a Basis from a Single Vector

Often we need an orthonormal basis that is aligned with a given vector. That is, given a vector $\mathbf{a}$, we want an orthonormal $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ such that $\mathbf{w}$ points in the same direction as $\mathbf{a}$ (Hughes & Möller, 1999), but we don't particularly care what $\mathbf{u}$ and $\mathbf{v}$ are. One vector isn't enough to uniquely determine the answer; we just need a robust procedure that will find any one of the possible bases.

This can be done using cross products as follows. First make $\mathbf{w}$ a unit vector in the direction of $\mathbf{a}$:

$$\mathbf{w} = \frac{\mathbf{a}}{\|\mathbf{a}\|}.$$

This same procedure can, of course, be used to construct the three vectors in any order; just pay attention to the order of the cross products to ensure the basis is right-handed.

Then choose any vector $\mathbf{t}$ not collinear with $\mathbf{w}$, and use the cross product to build a unit vector $\mathbf{u}$ perpendicular to $\mathbf{w}$:

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|}.$$

If $\mathbf{t}$ is collinear with $\mathbf{w}$ the denominator will vanish, and if they are nearly collinear the results will have low precision. A simple procedure to find a vector sufficiently different from $\mathbf{w}$ is to start with $\mathbf{t}$ equal to $\mathbf{w}$ and change the smallest magnitude component of $\mathbf{t}$ to 1. For example, if $\mathbf{w} = (1/\sqrt{2}, -1/\sqrt{2}, 0)$ then $\mathbf{t} = (1/\sqrt{2}, -1/\sqrt{2}, 1)$. Once $\mathbf{w}$ and $\mathbf{u}$ are in hand, completing the basis is simple:

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

An example of a situation where this construction is used is surface shading, where a basis aligned to the surface normal is needed but the rotation around the normal is often unimportant.

2.4.7 Constructing a Basis from Two Vectors

The procedure in the previous section can also be used in situations where the rotation of the basis around the given vector is important. A common example is building a basis for a camera: it's important to have one vector aligned in the direction the camera is looking, but the orientation of the camera around that vector is *not* arbitrary, and it needs to be specified somehow. Once the orientation is pinned down, the basis is completely determined.

$\mathbf{u} = \mathbf{a} \times \mathbf{b}$ also produces an orthonormal basis, but it is left-handed.

A common way to fully specify a frame is by providing two vectors $\mathbf{a}$ (which specifies $\mathbf{w}$) and $\mathbf{b}$ (which specifies $\mathbf{v}$). If the two vectors are known to be perpendicular it is a simple matter to construct the third vector by $\mathbf{u} = \mathbf{b} \times \mathbf{a}$.

To be sure that the resulting basis really is orthonormal, even if the input vectors weren't quite, a procedure much like the single-vector procedure is advisable:

$$\mathbf{w} = \frac{\mathbf{a}}{\|\mathbf{a}\|},$$
$$\mathbf{u} = \frac{\mathbf{b} \times \mathbf{w}}{\|\mathbf{b} \times \mathbf{w}\|},$$
$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

In fact, this procedure works just fine when $\mathbf{a}$ and $\mathbf{b}$ are not perpendicular. In this case, $\mathbf{w}$ will be constructed exactly in the direction of $\mathbf{a}$, and $\mathbf{v}$ is chosen to be the closest vector to $\mathbf{b}$ among all vectors perpendicular to $\mathbf{w}$.

If you want me to set $\mathbf{w}$ and $\mathbf{v}$ to two nonperpendicular directions, something has to give—with this scheme I'll set everything the way you want, except I'll make the smallest change to $\mathbf{v}$ so that it is in fact perpendicular to $\mathbf{w}$.

This procedure *won't* work if $\mathbf{a}$ and $\mathbf{b}$ are collinear. In this case $\mathbf{b}$ is of no help in choosing which of the directions perpendicular to $\mathbf{a}$ we should use: it is perpendicular to all of them.

In the example of specifying camera positions (Section 4.3), we want to construct a frame that has $\mathbf{w}$ parallel to the direction the camera is looking, and $\mathbf{v}$ should point out the top of the camera. To orient the camera upright, we build the basis around the view direction, using the straight-up direction as the reference vector to establish the camera's orientation around the view direction. Setting $\mathbf{v}$ as close as possible to straight up exactly matches the intuitive notion of "holding the camera straight."

What will go wrong with the computation if $\mathbf{a}$ and $\mathbf{b}$ are parallel?

2.4.8 Squaring Up a Basis

Occasionally you may find problems caused in your computations by a basis that is supposed to be orthonormal but where error has crept in—due to rounding error in computation, or to the basis having been stored in a file with low precision, for instance.

The procedure of the previous section can be used; simply constructing the basis anew using the existing $\mathbf{w}$ and $\mathbf{v}$ vectors will produce a new basis that is orthonormal and is close to the old one.

This approach is good for many applications, but it is not the best available. It does produce accurately orthogonal vectors, and for nearly orthogonal starting bases the result will not stray far from the starting point. However, it is asymmetric: it "favors" $\mathbf{w}$ over $\mathbf{v}$ and $\mathbf{v}$ over $\mathbf{u}$ (whose starting value is thrown away). It chooses a basis close to the starting basis but has no guarantee of choosing *the* closest orthonormal basis. When this is not good enough, the SVD (Section 5.4.1) can be used to compute an orthonormal basis that *is* guaranteed to be closest to the original basis.

2.5 Curves and Surfaces

The geometry of curves, and especially surfaces, plays a central role in graphics, and here we review the basics of curves and surfaces in 2D and 3D space.

2.5.1 2D Implicit Curves

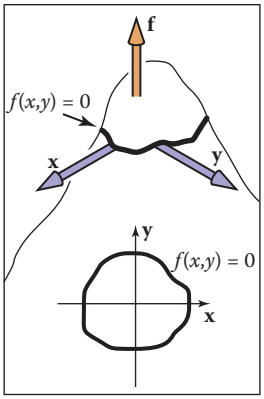


Figure 2.22. An implicit function $f(x,y) = 0$ can be thought of as a height field where f is the height (top). A path where the height is zero is the implicit curve (bottom).

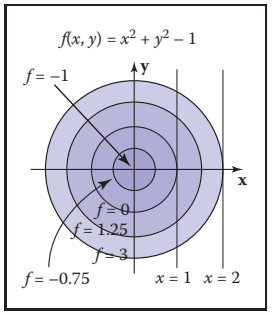


Figure 2.23. An implicit function $f(x,y) = 0$ can be thought of as a height field where f is the height (top). A path where the height is zero is the implicit curve (bottom).

Intuitively, a *curve* is a set of points that can be drawn on a piece of paper without lifting the pen. A common way to describe a curve is using an *implicit equation*. An implicit equation in two dimensions has the form

$$f(x, y) = 0.$$

The function $f(x, y)$ returns a real value. Points (x, y) where this value is zero are on the curve, and points where the value is nonzero are not on the curve. For example, let's say that $f(x, y)$ is

$$f(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2, \tag{2.9}$$

where (x_c, y_c) is a 2D point and r is a nonzero real number. If we take $f(x, y) = 0$, the points where this equality holds are on the circle with center (x_c, y_c) and radius r. The reason that this is called an "implicit" equation is that the points (x, y) on the curve cannot be immediately calculated from the equation and instead must be determined by solving the equation. Thus, the points on the curve are not generated by the equation *explicitly*, but they are buried somewhere *implicitly* in the equation.

It is interesting to note that f does have values for all (x, y). We can think of f as a terrain, with sea level at $f = 0$ (Figure 2.22). The shore is the implicit curve. The value of f is the altitude. Another thing to note is that the curve partitions space into regions where $f > 0$, $f < 0$, and $f = 0$. So you evaluate f to decide whether a point is "inside" a curve. Note that $f(x, y) = c$ is a curve for any constant c, and $c = 0$ is just used as a convention. For example, if $f(x, y) = x^2 + y^2 - 1$, varying c just gives a variety of circles centered at the origin (Figure 2.23).

We can compress our notation using vectors. If we have $\mathbf{c} = (x_c, y_c)$ and $\mathbf{p} = (x, y)$, then our circle with center $\mathbf{c}$ and radius r is defined by those position vectors that satisfy

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - r^2 = 0.$$

This equation, if expanded algebraically, will yield Equation (2.9), but it is easier to see that this is an equation for a circle by "reading" the equation geometrically. It reads, "points $\mathbf{p}$ on the circle have the following property: the vector from $\mathbf{c}$ to

$\mathbf{p}$ when dotted with itself has value r^2." Because a vector dotted with itself is just its own length squared, we could also read the equation as, "points $\mathbf{p}$ on the circle have the following property: the vector from $\mathbf{c}$ to $\mathbf{p}$ has squared length r^2."

Even better, is to observe that the squared length is just the squared distance from $\mathbf{c}$ to $\mathbf{p}$, which suggests the equivalent form

$$\|\mathbf{p} - \mathbf{c}\|^2 - r^2 = 0,$$

and, of course, this suggests

$$\|\mathbf{p} - \mathbf{c}\| - r = 0.$$

The above could be read "the points $\mathbf{p}$ on the circle are those a distance r from the center point $\mathbf{c}$," which is as good a definition of circle as any. This illustrates that the vector form of an equation often suggests more geometry and intuition than the equivalent full-blown Cartesian form with x and y. For this reason, it is usually advisable to use vector forms when possible. In addition, you can support a vector class in your code; the code is cleaner when vector forms are used. The vector-oriented equations are also less error prone in implementation: once you implement and debug vector types in your code, the cut-and-paste errors involving x, y, and z will go away. It takes a little while to get used to vectors in these equations, but once you get the hang of it, the payoff is large.

2.5.2 The 2D Gradient

If we think of the function $f(x, y)$ as a height field with height $= f(x, y)$, the *gradient* vector points in the direction of maximum upslope, i.e., straight uphill. The gradient vector $\nabla f(x, y)$ is given by

$$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right).$$

The gradient vector evaluated at a point on the implicit curve $f(x, y) = 0$ is perpendicular to the *tangent* vector of the curve at that point. This perpendicular vector is usually called the *normal vector* to the curve. In addition, since the gradient points uphill, it indicates the direction of the $f(x, y) > 0$ region.

In the context of height fields, the geometric meaning of partial derivatives and gradients is more visible than usual. Suppose that near the point (a, b), $f(x, y)$ is a plane (Figure 2.24). There is a specific uphill and downhill direction. At right angles to this direction is a direction that is level with respect to the plane. Any intersection between the plane and the $f(x, y) = 0$ plane will be in the direction that is level. Thus the uphill/downhill directions will be perpendicular to the line of intersection $f(x, y) = 0$. To see why the partial derivative has something to do

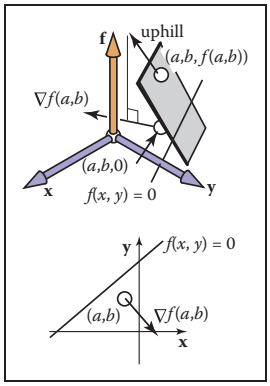


Figure 2.24. A surface height = $f(x,y)$ is locally planar near $(x,y) = (a,b)$. The gradient is a projection of the uphill direction onto the height = 0 plane.

with this, we need to visualize its geometric meaning. Recall that the conventional derivative of a 1D function $y = g(x)$ is

$$\frac{dy}{dx} \equiv \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{g(x + \Delta x) - g(x)}{\Delta x}. \tag{2.10}$$

This measures the *slope* of the *tangent* line to g (Figure 2.25).

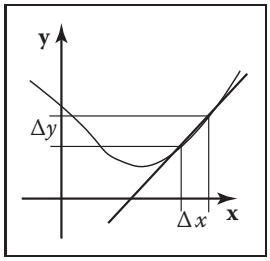


Figure 2.25. The derivative of a 1D function measures the slope of the line tangent to the curve.

The partial derivative is a generalization of the 1D derivative. For a 2D function $f(x, y)$, we can't take the same limit for x as in Equation (2.10), because f can change in many ways for a given change in x. However, if we hold y constant, we can define an analog of the derivative, called the *partial derivative* (Figure 2.26):

$$\frac{\partial f}{\partial x} \equiv \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x}.$$

Why is it that the partial derivatives with respect to x and y are the components of the gradient vector? Again, there is more obvious insight in the geometry than in the algebra. In Figure 2.27, we see the vector $\mathbf{a}$ travels along a path where $\mathbf{f}$ does not change. Note that this is again at a small enough scale that the surface height $(x, y) = f(x, y)$ can be considered locally planar. From the figure, we see that the vector $\mathbf{a} = (\Delta x, \Delta y)$.

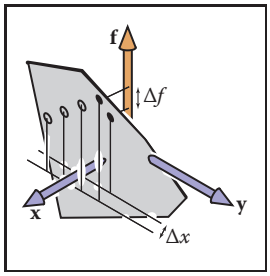


Figure 2.26. The partial derivative of a function f with respect to x must hold y constant to have a unique value, as shown by the dark point. The hollow points show other values of f that do not hold y constant.

Because the uphill direction is perpendicular to $\mathbf{a}$, we know the dot product is equal to zero:

$$(\nabla f) \cdot \mathbf{a} \equiv (x_\nabla, y_\nabla) \cdot (x_a, y_a) = x_\nabla \Delta x + y_\nabla \Delta y = 0. \tag{2.11}$$

We also know that the change in f in the direction (x_a, y_a) equals zero:

$$\Delta f = \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y \equiv \frac{\partial f}{\partial x}x_a + \frac{\partial f}{\partial y}y_a = 0.$$

Given any vectors (x, y) and (x', y') that are perpendicular, we know that the angle between them is 90 degrees, and thus their dot product equals zero (recall that the dot product is proportional to the cosine of the angle between the two vectors). Thus, we have $xx' + yy' = 0$. Given (x, y), it is easy to construct valid vectors whose dot product with (x, y) equals zero, the two most obvious being $(y, -x)$ and $(-y, x)$; you can verify that these vectors give the desired zero dot product with (x, y). A generalization of this observation is that (x, y) is perpendicular to $k(y, -x)$ where k is any nonzero constant. This implies that

$$(x_a, y_a) = k\left(\frac{\partial f}{\partial y}, -\frac{\partial f}{\partial x}\right). \tag{2.12}$$

Combining Equations (2.11) and (2.12) gives

$$(x_\nabla, y_\nabla) = k'\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right),$$

where k' is any nonzero constant. By definition, "uphill" implies a positive change in f, so we would like $k' > 0$, and $k' = 1$ is a perfectly good convention.

As an example of the gradient, consider the implicit circle $x^2 + y^2 - 1 = 0$ with gradient vector $(2x, 2y)$, indicating that the outside of the circle is the positive region for the function $f(x, y) = x^2 + y^2 - 1$. Note that the length of the gradient vector can be different depending on the multiplier in the implicit equation. For example, the unit circle can be described by $Ax^2 + Ay^2 - A = 0$ for any nonzero A. The gradient for this curve is $(2Ax, 2Ay)$. This will be normal (perpendicular) to the circle, but will have a length determined by A. For $A > 0$, the normal will point outward from the circle, and for $A < 0$, it will point inward. This switch from outward to inward is as it should be, since the positive region switches inside the circle. In terms of the height-field view, $h = Ax^2 + Ay^2 - A$, and the circle is at zero altitude. For $A > 0$, the circle encloses a depression, and for $A < 0$, the circle encloses a bump. As A becomes more negative, the bump increases in height, but the $h = 0$ circle doesn't change. The direction of maximum uphill doesn't change, but the slope increases. The length of the gradient reflects this change in degree of the slope. So intuitively, you can think of the gradient's direction as pointing uphill and its magnitude as measuring how uphill the slope is.

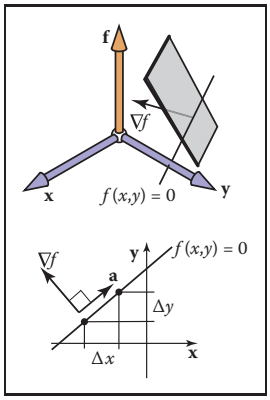


Figure 2.27. The vector **a** points in a direction where f has no change and is thus perpendicular to the gradient vector ∇f.

Implicit 2D Lines

The familiar "slope-intercept" form of the line is

$$y = mx + b. \tag{2.13}$$

This can be converted easily to implicit form (Figure 2.28):

$$y - mx - b = 0. \tag{2.14}$$

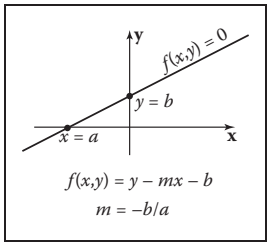


Figure 2.28. A 2D line can be described by the equation $y - mx - b = 0$.

Here m is the "slope" (ratio of rise to run) and b is the y value where the line crosses the y-axis, usually called the *y-intercept*. The line also partitions the 2D plane, but here "inside" and "outside" might be more intuitively called "over" and "under."

Because we can multiply an implicit equation by any constant without changing the points where it is zero, $kf(x, y) = 0$ is the same curve for any nonzero k. This allows several implicit forms for the same line, for example,

$$2y - 2mx - 2b = 0.$$

One reason the slope-intercept form is sometimes awkward is that it can't represent some lines such as $x = 0$ because m would have to be infinite. For this

reason, a more general form is often useful:

$$Ax + By + C = 0, \tag{2.15}$$

for real numbers A, B, C.

Suppose we know two points on the line, (x_0, y_0) and (x_1, y_1). What A, B, and C describe the line through these two points? Because these points lie on the line, they must both satisfy Equation (2.15):

$$Ax_0 + By_0 + C = 0,$$
$$Ax_1 + By_1 + C = 0.$$

Unfortunately we have two equations and *three* unknowns: A, B, and C. This problem arises because of the arbitrary multiplier we can have with an implicit equation. We could set $C = 1$ for convenience:

$$Ax + By + 1 = 0,$$

but we have a similar problem to the infinite slope case in slope-intercept form: lines through the origin would need to have $A(0) + B(0) + 1 = 0$, which is a contradiction. For example, the equation for a 45-degree line through the origin can be written $x - y = 0$, or equally well $y - x = 0$, or even $17y - 17x = 0$, but it cannot be written in the form $Ax + By + 1 = 0$.

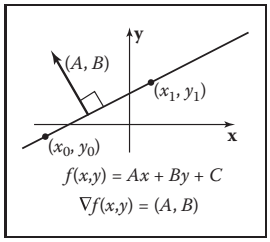


Figure 2.29. The gradient vector (A, B) is perpendicular to the implicit line $Ax + By + C = 0$.

Whenever we have such pesky algebraic problems, we try to solve the problems using geometric intuition as a guide. One tool we have, as discussed in Section 2.5.2, is the gradient. For the line $Ax + By + C = 0$, the gradient vector is (A, B). This vector is perpendicular to the line (Figure 2.29), and points to the side of the line where $Ax + By + C$ is positive. Given two points on the line (x_0, y_0) and (x_1, y_1), we know that the vector between them points in the same direction as the line. This vector is just $(x_1 - x_0, y_1 - y_0)$, and because it is parallel to the line, it must also be perpendicular to the gradient vector (A, B). Recall that there are an infinite number of (A, B, C) that describe the line because of the arbitrary scaling property of implicits. We want any one of the valid (A, B, C).

We can start with any (A, B) perpendicular to $(x_1 - x_0, y_1 - y_0)$. Such a vector is just $(A, B) = (y_0 - y_1, x_1 - x_0)$ by the same reasoning as in Section 2.5.2. This means that the equation of the line through (x_0, y_0) and (x_1, y_1) is

$$(y_0 - y_1)x + (x_1 - x_0)y + C = 0. \tag{2.16}$$

Now we just need to find C. Because (x_0, y_0) and (x_1, y_1) are on the line, they must satisfy Equation (2.16). We can plug either value in and solve for C. Doing this for (x_0, y_0) yields $C = x_0y_1 - x_1y_0$, and thus the full equation for the line is

$$(y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0. \tag{2.17}$$

Again, this is one of infinitely many valid implicit equations for the line through two points, but this form has no division operation and thus no numerically degenerate cases for points with finite Cartesian coordinates. A nice thing about Equation (2.17) is that we can always convert to the slope-intercept form (when it exists) by moving the non-y terms to the right-hand side of the equation and dividing by the multiplier of the y term:

$$y = \frac{y_1 - y_0}{x_1 - x_0}x + \frac{x_1 y_0 - x_0 y_1}{x_1 - x_0}.$$

An interesting property of the implicit line equation is that it can be used to find the signed distance from a point to the line. The value of $Ax + By + C$ is proportional to the distance from the line (Figure 2.30). As shown in Figure 2.31, the distance from a point to the line is the length of the vector $k(A, B)$, which is

$$\text{distance} = k\sqrt{A^2 + B^2}. \tag{2.18}$$

For the point $(x, y) + k(A, B)$, the value of $f(x, y) = Ax + By + C$ is

$$\begin{aligned} f(x + kA, y + kB) &= Ax + kA^2 + By + kB^2 + C \\ &= k(A^2 + B^2). \end{aligned} \tag{2.19}$$

The simplification in that equation is a result of the fact that we know (x, y) is on the line, so $Ax + By + C = 0$. From Equations (2.18) and (2.19), we can see that the signed distance from line $Ax + By + C = 0$ to a point (a, b) is

$$\text{distance} = \frac{f(a, b)}{\sqrt{A^2 + B^2}}.$$

Here "signed distance" means that its magnitude (absolute value) is the geometric distance, but on one side of the line, distances are positive and on the other they are negative. You can choose between the equally valid representations $f(x, y) = 0$ and $-f(x, y) = 0$ if your problem has some reason to prefer a particular side being positive. Note that if (A, B) is a unit vector, then $f(a, b)$ is the signed distance. We can multiply Equation (2.17) by a constant that ensures that (A, B) is a unit vector:

$$\begin{aligned} f(x, y) = \frac{y_0 - y_1}{\sqrt{(x_1 - x_0)^2 + (y_0 - y_1)^2}}x + \frac{x_1 - x_0}{\sqrt{(x_1 - x_0)^2 + (y_0 - y_1)^2}}y \\ + \frac{x_0 y_1 - x_1 y_0}{\sqrt{(x_1 - x_0)^2 + (y_0 - y_1)^2}} = 0. \end{aligned} \tag{2.20}$$

Note that evaluating $f(x, y)$ in Equation (2.20) directly gives the signed distance, but it does require a square root to set up the equation. Implicit lines will turn out to be very useful for triangle rasterization (Section 8.1.2). Other forms for 2D lines are discussed in Chapter 14.

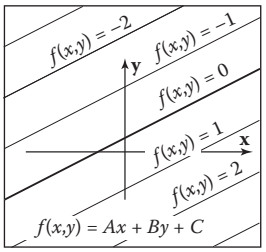


Figure 2.30. The value of the implicit function $f(x,y) = Ax + By + C$ is a constant times the signed distance from $Ax + By + C = 0$.

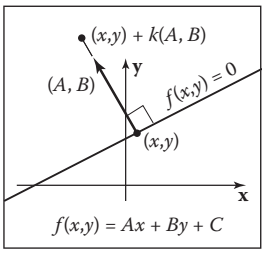


Figure 2.31. The vector $k(A,B)$ connects a point (x,y) on the line closest to a point not on the line. The distance is proportional to k.