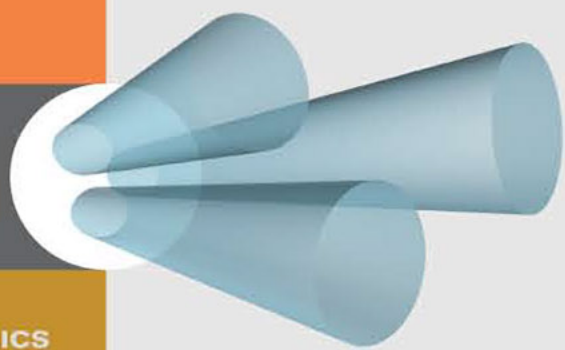
DAVID H. EBERLY

SERIES IN INTERACTIVE 3D TECHNOLOGY

3D GAME ENGINE DESIGN



A PRACTICAL APPROACH TO
REAL-TIME COMPUTER GRAPHICS
SECOND EDITION

MORGAN KAUFMANN

3D Game Engine Design

A Practical Approach to Real-Time Computer Graphics

The Morgan Kaufmann Series in Interactive 3D Technology

Series Editor: David H. Eberly, Geometric Tools, Inc.

The game industry is a powerful and driving force in the evolution of computer technology. As the capabilities of personal computers, peripheral hardware, and game consoles have grown, so has the demand for quality information about the algorithms, tools, and descriptions needed to take advantage of this new technology. To satisfy this demand and establish a new level of professional reference for the game developer, we created the *Morgan Kaufmann Series in Interactive 3D Technology.* Books in the series are written for developers by leading industry professionals and academic researchers, and cover the state of the art in real-time 3D. The series emphasizes practical, working solutions and solid software-engineering principles. The goal is for the developer to be able to implement real systems from the fundamental ideas, whether it be for games or for other applications.

3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics, 2nd Edition
David H. Eberly

Game Physics Engine Development
Ian Millington

X3D: Extensible 3D Graphics for Web Authors
Don Brutzman and Leonard Daly

Artificial Intelligence for Games
Ian Millington

Better Game Characters by Design: A Psychological Approach
Katherine Isbister

Visualizing Quaternions
Andrew J. Hanson

3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic
David H. Eberly

Real-Time Collision Detection
Christer Ericson

Physically Based Rendering: From Theory to Implementation
Matt Pharr and Greg Humphreys

Essential Mathematics for Games and Interactive Applications: A Programmer's Guide
James M. Van Verth and Lars M. Bishop

Game Physics
David H. Eberly

Collision Detection in Interactive 3D Environments
Gino van den Bergen

Forthcoming

In Silico: Cell Biology Science and Animation with Maya
Jason Sharpe, Charles Lumsden, Nicholas Woolridge

Real-Time Cameras
Mark Haigh-Hutchinson

3D Game Engine Design

A Practical Approach to Real-Time Computer Graphics

Second Edition

David H. Eberly
Geometric Tools, Inc.

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Supplementary Resources Disclaimer

Additional resources were previously made available for this title on CD. However, as CD has become a less accessible format, all resources have been moved to a more convenient online download option.

You can find these resources available here: www.routledge.com/9780122290633

Please note: Where this title mentions the associated disc, please use the downloadable resources instead.

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

Version Date: 20150223

International Standard Book Number-13: 978-1-4822-6730-3 (eBook - PDF)

TRADEMARKS

The following trademarks, mentioned in this book and the accompanying CD-ROM, are the property of the following organizations:

- AltiVec is a trademark of Freescale Semiconductor.
- DirectX, Direct3D, Visual C++, Windows, Xbox, and Xbox 360 are trademarks of Microsoft Corporation.
- GameCube is a trademark of Nintendo.
- GeForce, Riva TNT, and the Cg Language are trademarks of NVIDIA Corporation.
- Java 3D is a trademark of Sun Microsystems.
- Macintosh is a trademark of Apple Corporation.
- Morrowind and The Elder Scrolls are trademarks of Bethesda Softworks, LLC.
- NetImmerse and Gamebryo are trademarks of Emergent Game Technologies.
- OpenGL is a trademark of Silicon Graphics, Inc.
- Pentium and Streaming SIMD Extensions (SSE) are trademarks of Intel Corporation.
- PhysX is a trademark of Ageia Technologies, Inc.
- Playstation 2 and Playstation 3 are trademarks of Sony Corporation.
- PowerPC is a trademark of IBM.
- Prince of Persia 3D is a trademark of Brøderbund Software, Inc.
- 3DNow! is a trademark of Advanced Micro Devices.
- 3D Studio Max is a trademark of Autodesk, Inc.

About the Author

Dave Eberly is the president of Geometric Tools, Inc. (*www.geometrictools.com*), a company that specializes in software development for computer graphics, image analysis, and numerical methods. Previously, he was the director of engineering at Numerical Design Ltd. (NDL), the company responsible for the real-time 3D game engine, NetImmerse. He also worked for NDL on Gamebryo, which was the next-generation engine after NetImmerse. His background includes a BA degree in mathematics from Bloomsburg University, MS and PhD degrees in mathematics from the University of Colorado at Boulder, and MS and PhD degrees in computer science from the University of North Carolina at Chapel Hill. He is the author of *Game Physics* (2004) and *3D Game Engine Architecture* (2005) and coauthor with Philip Schneider of *Geometric Tools for Computer Graphics* (2003), all published by Morgan Kaufmann. As a mathematician, Dave did research in the mathematics of combustion, signal and image processing, and length-biased distributions in statistics. He was an associate professor at the University of Texas at San Antonio with an adjunct appointment in radiology at the U.T. Health Science Center at San Antonio. In 1991, he gave up his tenured position to retrain in computer science at the University of North Carolina. After graduating in 1994, he remained for one year as a research associate professor in computer science with a joint appointment in the Department of Neurosurgery, working in medical image analysis. His next stop was the SAS Institute, working for a year on SAS/Insight, a statistical graphics package. Finally, deciding that computer graphics and geometry were his real calling, Dave went to work for NDL (which is now Emergent Game Technologies), then to Magic Software, Inc., which later became Geometric Tools, Inc. Dave's participation in the newsgroup *comp.graphics.algorithms* and his desire to make 3D graphics technology available to all are what has led to the creation of his company's Web site and his books.

Contents

CHAPTER 16

NUMERICAL METHODS 719

Preface

The first edition of *3D Game Engine Design* appeared in print over six years ago (September 2000). At that time, shader programming did not exist on consumer graphics hardware. All rendering was performed using the fixed-function pipeline, which consisted of setting render states in order to control how the geometric data was affected by the drawing pass.

The first edition contained a CDROM with the source code for Wild Magic Version 0.1, which included 1,015 source files and 17 sample applications, for a total of 101,293 lines of code. The distribution contained support only for computers running the Microsoft Windows operating system; the renderer was built on top of OpenGL; and project files were provided for Micrsoft Visual C++ 6. Over the years, the source code evolved to Wild Magic Version 3.9, which contained additional support for Linux and Macintosh platforms, had OpenGL and Direct3D renderers, and included some support for shader programming. However, the design of the engine was still based on a fixed-function pipeline. The distribution also included support for multiple versions of Microsoft's compilers, support for other compilers on the various platforms, and contained some tools such as importers and exporters for processing of art assets.

This is the second edition of *3D Game Engine Design*. It is much enhanced, describing the foundations for shader programming and how an engine can support it. The second edition is about twice the size of the first. The majority of the increase is due to a more detailed description of all aspects of the graphics system, particularly about how shaders fit into the geometric pipeline. The material on scene graphs and their management is also greatly expanded. The second edition has more figures and less emphasis on the mathematical aspects of an engine.

The second edition contains a CDROM with the source code for Wild Magic Version 4.0, which includes 1,587 source files and 105 sample applications, for a total of 249,860 lines of code. The Windows, Linux, and Macintosh platforms are still supported, using OpenGL renderers. The Windows platform also has a Direct3D renderer whose performance is comparable to that of the OpenGL renderer. Multiple versions of Microsoft's C++ compilers are supported—versions 6, 7.0, 7.1, and 8.0 (Professional and Express Editions). The MINGW compiler and MSYS environment are also supported on the Windows platform. The Linux platform uses the g++ compiler, and the Macintosh platform uses Apple's Xcode tools.

The graphics system of Wild Magic Version 4.0 is fully based on shader programming and relies on NVIDIA's Cg programming language. The precompiled shader programs were created using the `arbvp1` and `arbfp1` profiles for OpenGL and using the `vs_2_0` and `ps_2_0` profiles for Direct3D, so your graphics hardware must

support these in order to run the sample applications. If your graphics hardware supports only lesser profiles such as `vs_1_1` and `ps_1_1`, you must recompile the shader programs with these profiles and use the outputs instead of what is shipped on the CDROM. The distribution also contains a fully featured, shader-based software renderer to illustrate all aspects of the geometric pipeline, not just the vertex and pixel shader components.

The replacement of the fixed-function approach by a shader-based approach has made Wild Magic Version 4 a much more powerful graphics engine for use in all graphics applications, not just in games. Much effort went into making the engine easier to use and to extend, and into improving the performance of the renderers. I hope you enjoy this new manifestation of Wild Magic!

A book is never just the product of the author alone. Many people were involved in making this book as good as it possibly can be. Thanks to the reviewers for providing valuable and insightful feedback about the first edition regarding how to improve it for a second edition. A special thanks goes to Marc Olano (University of Maryland, Baltimore County) for taking the time to provide me with detailed comments based on his experience using the first edition as a textbook. Thanks to Elisabeth Beller, the production editor and project manager for all of my Morgan Kaufmann Publisher books, for assembling yet another fine group of people who have the superb ability to take my unattractive book drafts and make them look really good. And, as always, thanks to my editor Tim Cox for his patience and help in producing yet another book for Morgan Kaufmann Publishers.

INTRODUCTION

I have no fault to find with those who teach geometry. That science is the only one which has not produced sects; it is founded on analysis and on synthesis and on the calculus; it does not occupy itself with probable truth; moreover it has the same method in every country.

— Frederick the Great

1.1 THE EVOLUTION OF GRAPHICS HARDWARE AND GAMES

The first edition of *3D Game Engine Design* was written in the late 1990s when 3dfx Voodoo cards were in style and the NVIDIA Riva TNT cards had just arrived. The book was written based on the state of graphics at that time. Six years have passed between that edition and this, the second edition. Graphics hardware has changed dramatically. So have games. The hardware has extremely powerful graphics processing units (GPUs), lots of video memory, and the option of programming it via shader programs. (These did not exist on consumer cards when I wrote the first edition.) Games have evolved also, having much richer (and much more) content and using more than graphics. We now have physics engines and more recently physics processors (PhysX from Ageia).

The Sony Playstation 2 was not quite released when I started writing the first edition. We've also seen Microsoft's Xbox arrive on the scene, as well as the Nintendo GameCube. These days we have Microsoft's Xbox 360 with multiple processors, and the Sony Playstation 3 is soon to follow with the Cell architecture. Smaller game-playing devices are available. Mobile phones with video screens are also quite popular.

With all this evolution, the first edition of the book has shown its age regarding the discussion of real-time graphics. The time is right for the second edition, so here it is.

1.2 The Evolution of This Book and Its Software

In the late 1990s when I conceived the idea of writing a book on real-time graphics as used in games, I was employed by Numerical Design, Ltd. (now Emergent Game Technologies) designing and developing NetImmerse (now Gamebryo). At that time the term *game engine* really did refer to the graphics portion of the system. Companies considering using NetImmerse wanted the real-time graphics in order to free up the computer processing unit (CPU) for use by other systems they themselves were used to building: the game logic, the game artificial intelligence (AI), rudimentary collision and physics, networking, and other components. The first edition of *3D Game Engine Design* is effectively a detailed summary of what went into building NetImmerse.

Over the years I have received some criticism for using "game engine" in the title when the book is mainly about graphics. Since that time, the term *game engine* has come to mean a collection of engines—for graphics, physics, AI, networking, scripting, and you name it. It is not feasible to write a book in a reasonable amount of time with sufficient depth to cover all these topics, nor do I intend to write such a massive tome. To address the criticism about the book title, I could have changed the title itself. However, I have chosen to keep the original title—the book is known now by its name, for better or for worse. The second edition includes some discussion about physics and some discussion about an application layer and how the engines must interact, but probably this is not enough to discourage the criticism about the title. So be it.

The first edition appeared in print in September 2000. It is now six years later and the book remains popular in many circles. The algorithmic aspects are still relevant, the scene graph management still applies, but the material on rendering is clearly out of date. That material was essentially about the *fixed-function pipeline* view of a graphics system. The evolution of graphics hardware to support a *shader-based pipeline* has been rapid, now allowing us to concentrate on the special effects themselves (via shader programming) rather than trying to figure out how to call the correct set of state-enabling functions in the fixed-function pipeline to obtain a desired effect.

The second edition of the book now focuses on the design of the scene graph managment system and its associated rendering layer. Most of the algorithmic concepts have not changed regarding specialized scene graph classes such as the controller classes, the sorting classes, or level-of-detail classes. Core classes such as the spatial, geometry, and node classes have changed to meet the needs of a shader-based system. The current scene graph management system is much more powerful, flexible, and efficient than its predecessors. The shader effect system is integrated with the scene graph management so that you may do single-pass drawing, multipass drawing with a single effect, or even drawing with multiple effects. I have paid much attention to hiding as many details as possible from the application developer, relying on well-designed and automated subsystems to do the work that earlier versions of my scene graph management system forced the developer to do.

One characteristic of my books that I believe sets them apart from other technical books is the inclusion of large source code libraries, a lot of sample applications that actually compile and run, and support for multiple platforms (PC, Mac, Linux/Unix, and various compilers on each platform). What you have purchased is a book *and* a software product to illustrate what is described in the book. The sample source code that ships with many books is not carefully planned, lacks quality control, is not multiplatform, and usually is not maintained by the book authors. I am interested in carefully designed and planned code. I believe in quality source code. I maintain the source code on a regular basis, so I encourage people to send email about problems they encounter, both with the source code and in the book material. The Geometric Tools website lists all the updates to the software, including bug fixes as well as new features, and the site has pages for book corrections.

The first edition of this book shipped with Wild Magic version 0.1. The book had two additional printings in its first edition, one shipping with Wild Magic version 0.2 and one shipping with Wild Magic version 0.4. The second edition ships with Wild Magic version 4.0, which when compared to version 0.1 looks very little like it. I believe the quality of the Wild Magic source code is a significant feature that has attracted many users. The latest version represents a significant rewrite to the rendering layer that has led to easier use of the engine and better performance by the renderers. The rewrite represents three months of dedicated time, something most authors would never consider investing time in, and it includes implementing a shader-based software renderer just to illustrate the book concepts in detail. I hope you enjoy using Wild Magic for your leisure projects!

1.3 A Summary of the Chapters

The book is partitioned into six parts.

Graphics. Chapter 2 discusses the details of a rendering system, including transformations, camera models, culling and clipping, rasterizing, and issues regarding software versus hardware rendering and about specific graphics application programmer interfaces (graphics APIs) in use these days. Chapter 3 is about rendering from the perspective of actually writing all the subsystems for a software renderer. The chapter includes what I consider a reasonable abstraction of the interface for a shader-based rendering system. This interface essentially shows that the renderer spends most of its time doing resource management. Chapter 3 also includes details about shader programs—not about writing them but about dealing with data management issues. Here I address such things as matching geometric vertex data to vertex program inputs, matching vertex program outputs to pixel program inputs, and ensuring that the infrastructure is set so that all resources are in the right place at the right time, hooked up, and ready to use for real-time rendering.

Scene Graph Management. Chapter 4 is about the essentials of organizing your data as a scene graph. This system is designed to be high level to allow ease of use

by application programmers, to be an efficient system to feed a renderer, and to be naturally extensible. The chapter also includes a section that talks about scene graph compiling—converting scene graphs to more optimized forms for target platforms. Chapters 5, 6, and 7 are about specially designed nodes and subsystems of the scene graph management system. These include subsystems to support animation, spatial sorting, and level of detail.

Collision Detection and Physics. Some general concepts you see in attempting to have physical realism in a three-dimensional (3D) application are discussed in Chapters 8 and 9. A generic approach to collision detection is presented, one that I have successfully implemented in a real environment. I also discuss picking operations and briefly talk about automatic pathfinding as a means for collision avoidance. The chapter on physics is a brief discussion of some concepts you will see often when working with physical simulations, but it does not include a discussion about the black-box-style physics you see in a commercial physics engine, such as Havok. That type of physics requires a lot more discussion than space allows in this book. Such physics is heavily mathematical and requires attention to issues of numerical round-off errors when using floating-point arithmetic.

Mathematical Topics. Chapters 10 through 17 include a lot of the mathematical detail for much of the source code you will find in Wild Magic. These chapters include a discussion of standard objects encountered in geometric manipulation and queries, including curves and surfaces covered in Chapters 11 and 12. You will also find material on queries regarding distance, containment, and intersection. Chapter 16 presents some common numerical methods that are useful in graphics and physics applications. The final chapter in this partition is about the topic of rotation, including basic properties of rotation matrices and how quaternions are related to matrices.

Software Engineering. Chapter 18 is a brief summary of basic principles of object-oriented design and programming. Various base-level support for large libraries is important and includes topics such as run-time type information, shared objects and reference counting, streaming of data (to/from disk, across a network), and initialization and termination for disciplined object creation and destruction in an application. Chapter 19 is about memory management. This is of particular importance when you want to write your own memory managers in order to build a system that is handed a fixed-side memory block and told it may only use memory from that block. In particular, this approach is used on game consoles where each engine (graphics, physics, sound, and so on) is given its memory "budget." This is important for having predictable behavior of the engines. The last thing you want to happen in your game is one system consuming so much memory from a global heap that another system fails because it cannot successfully allocate what it needs.

Special Effects Using Shaders. Chapter 20 shows a handful of sample shaders and the applications that use them. This is just to give you an idea of what you can do with shaders and how Wild Magic handles them. The appendix describes how you can add new shader effects to Wild Magic. The process is not difficult (by design).

I believe the organization here is an improvement over that of the first edition of the book. A number of valid criticisms of the first edition were about the amount

of mathematics interleaved in the discussions. Sorry, but I remain a firm believer that you need a lot of mathematics when building sophisticated graphics and physics engines. That said, I have made an attempt to discuss first the general concepts for graphics, factoring the finer detail into the chapters in the mathematics section that occurs late in the book. For example, it is possible to talk about culling of bounding volumes against frustum planes without immediately providing the details of the algorithm for specific bounding volumes. When discussing distance-based collision detection, it is possible to motivate the concepts without the specific distance algorithms for pairs of objects. The mathematics is still here, but factored to the end of the book rather than interleaved through the entire book.

Another criticism of the first edition of the book was its lack of figures. I believe I have remedied this, adding quite a few more figures to the second edition. That said, there may be places in the book where someone might feel the need for a figure where I thought the concept did not require one. My apologies if this happens. Send me feedback by email if you believe certain parts of the book can be improved by the addition of figures.

Finally, I have included some exercises in the book. Creating a large set of well-crafted exercises is a full-time job. In fact, I recall meeting a person who worked for Addison-Wesley (back in the early 1980s). His full-time job was managing the exercises for the calculus textbook by George Thomas and Ross Finney (seventh edition at that time). As much as I would like to have included more exercises here, my time budget for writing the book, writing the Wild Magic source code to go with the book, and making a living doing contract programming already exceeded 24 hours per day. I hope the exercises I have included will support the use of the book as a textbook in a graphics course. Most of them are programming exercises, requests to modify the source code to do something different or to do something in addition to what it does. I can imagine some of these taking quite some time to do. But I also believe they will make students think—the point of exercises!

This page intentionally left blank

THE GRAPHICS SYSTEM

This chapter provides some basic concepts that occur in a computer graphics system. Some of these concepts are mathematical in nature. I am assuming that you are familiar with trigonometry, vector and matrix algebra, and dot products and cross products. A warning to those who have a significant mathematical background: I intentionally discuss the mathematical concepts in a somewhat informal manner. My goal is to present the relevant ideas without getting tied down in the minutiae of stating rigorous definitions for the concepts. The first edition of this book was criticized for overemphasizing the mathematical details—and rightly so. Learn computer graphics first, and then later explore the beauty of formal mathematical exposition!

The foundations of coordinate systems (Section 2.1) and transformations (Section 2.2) are pervasive throughout a game engine. They are found not only in the graphics engines but in the physics engines and sound engines. Getting a model out of a modeling package and into the game world, setting up a camera for viewing, and displaying the model vertices and triangles is a process for which you must absolutely understand the coordinate systems and transformations. Scene graph management (Chapter 4) also requires a thorough understanding of these topics.

Sections 2.3 through 2.6 are the foundation for drawing 3D objects on a 2D screen. In a programming environment using graphics APIs such as OpenGL or Direct3D to access the graphics hardware, your participation in the process is typically restricted to selecting the parameters of the camera, providing the triangle primitives whose vertices have been assigned various attributes, and identifying objects that are not within the viewing region so that you do not have to draw them. The low-level processing of vertices and triangles is the responsibility of the graphics drivers. A discussion of the low-level processing is provided in this book, and a software renderer is part of the source code so you can see an actual implementation of the ideas.

Section 2.7 is a discussion about issues that are relevant when designing and implementing a graphics engine. The first edition of this book had a similar section, but I have added new material, further delineating how you must think when designing an engine or building a component to live on top of an existing graphics API. Section 2.7.2 is about the trade-offs you must consider if you want your code to be portable (or not). As you will see, the most important trade-off is which computational unit has the responsibility for certain operations.

Section 2.8 is about the vector and matrix conventions used by OpenGL, Direct3D, and Wild Magic. In your own code you must also choose conventions. These include how to store vectors and matrices, how they multiply together, how rotations apply, and so on. The section also mentions a few other conventions that make the APIs different enough that you need to pay attention to them when creating a cross-platform graphics engine.

2.1 The Foundation

We are all familiar with the notation of *tuples*. The standard 3-tuple is written as (x, y, z). The components of the 3-tuple specify the location of a point in space relative to an origin. The components are referred to as the *Cartesian coordinates* of the point. You may have seen a diagram like the one in Figure 2.1 that illustrates the *standard coordinate system*.

Welcome to the book's first rendering of a 3D scene to a 2D screen, except this one was hand drawn! The standard coordinate system is simple enough, is it not? Coordinate systems other than the standard one may be imposed. Given that many people new to the field of computer graphics have some confusion about coordinate systems, perhaps it is not that simple after all. The confusion stems from having to work with multiple coordinate systems and knowing how they interact with each other. I will introduce these coordinate systems throughout the chapter and discuss their meaning. The important coordinate systems, called *spaces*, are *Cartesian space* (everything else is built on top of this), *model space* (or *object space*), *world space*, *view space* (or *camera space* or *eye space*), *clip space* (or *projection space* or *homogeneous space*), and *window space*.

Figure 2.1 is quite deceptive, which also leads to some confusion. I have drawn the figure as if the z-axis were in the upward direction. This is an unintentional consequence of having to draw *something* to illustrate a coordinate system. As humans who rely heavily on our vision, we have the notion of a *view direction*, an *up direction*, and a complementary *right direction* (sorry about that, left-handers). Just when you have become accustomed to thinking of the positive z-direction as the up direction, a modeling package comes along and insists that the positive y-direction is the up direction. The choice of view directions can be equally inconsistent, especially the defaults for various graphics engines and APIs. It is important to understand the co-

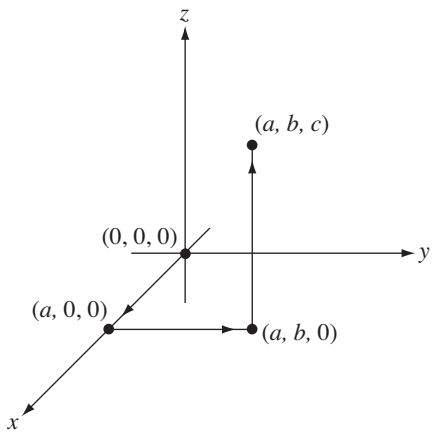


Figure 2.1 The standard coordinate system in three dimensions. The point at (a, b, c) is reached by starting at the origin $(0, 0, 0)$, moving a units in the direction $(1, 0, 0)$ to the point $(a, 0, 0)$, then moving b units in the direction $(0, 1, 0)$ to the point $(a, b, 0)$, and then moving c units in the direction $(0, 0, 1)$ to the point (a, b, c).

ordinate system conventions for all the packages you use in your game development. Throughout the book, my discussions about coordinate systems will refer to view, up, and right directions with coordinate names d, u, and r, respectively, rather than to axis names such as x, y, and z.

2.1.1 Coordinate Systems

Rather than constantly writing tuples, it is convenient to have a shorter notation to represent points and vectors. I will use boldface to do so. For example, the tuple (x, y, z) can refer to a point named **P**. Although mathematicians distinguish between a *point* and the *coordinates of a point*, I will be loose with the notation here and simply say $\mathbf{P} = (x, y, z)$. The typical names I use for the direction vectors are **D** for the view direction, **U** for the up direction, and **R** for the right direction. Using the standard convention, a direction vector must have unit length. For example, $(1, 0, 0)$ is a direction vector, but $(1, 1, 1)$ is not since its length is $\sqrt{3}$. In our agreed-upon notation, we can now define a coordinate system and how points are represented within a coordinate system.

A *coordinate system* consists of an *origin* point **E** and three independent direction vectors, **D** (view direction), **U** (up direction), and **R** (right direction). You may think

of the origin as the location of an observer who wishes to make measurements from his own perspective. The coordinate system is written succinctly as

$$\{\mathbf{E}; \mathbf{D}, \mathbf{U}, \mathbf{R}\} \tag{2.1}$$

Any point $\mathbf{X}$ may be represented in the coordinate system as

$$\mathbf{X} = \mathbf{E} + d\mathbf{D} + u\mathbf{U} + r\mathbf{R} \tag{2.2}$$

where d, u, and r are scalars that measure how far along the related direction you must move to get to the point $\mathbf{X}$. The tuple (d, u, r) lists the coordinates of $\mathbf{X}$ relative to the coordinate system in Equation (2.1).

The direction vectors are nearly always chosen to be mutually perpendicular. This is not necessary for coordinate systems. All that matters is that the directions are independent (*linearly independent* to those of you with some training in linear algebra). In this book, I will assume that the directions are indeed mutually perpendicular. If for any reason I need a coordinate system that does not have this property, I will make it very clear to you in that discussion. The assumption that the direction vectors in the coordinate system of Equation (2.1) are unit length and mutually perpendicular allows us to easily solve for the coordinates

$$d = \mathbf{D} \cdot (\mathbf{X} - \mathbf{E}), \qquad u = \mathbf{U} \cdot (\mathbf{X} - \mathbf{E}), \qquad r = \mathbf{R} \cdot (\mathbf{X} - \mathbf{E}) \tag{2.3}$$

where the bullet symbol $(\cdot)$ denotes the dot product of vectors. The construction of the coefficients relies on the directions having unit length ($\mathbf{D} \cdot \mathbf{D} = \mathbf{U} \cdot \mathbf{U} = \mathbf{R} \cdot \mathbf{R} = 1$) and being mutually perpendicular ($\mathbf{D} \cdot \mathbf{U} = \mathbf{D} \cdot \mathbf{R} = \mathbf{U} \cdot \mathbf{R} = 0$). In addition to being mutually perpendicular, the direction vectors are assumed to form a *right-handed system*. Specifically, I require that $\mathbf{R} = \mathbf{D} \times \mathbf{U}$, where the times symbol $(\times)$ denotes the cross product operator. This condition quantifies the usual *right-hand rule* for computing a cross product. A coordinate system may also be constructed to be a *left-handed system* using the *left-hand rule* for computing a cross product.

But wait. What does it really mean to be right-handed or left-handed? And what really is a cross product? The concepts have both algebraic and geometric interpretations, and I have seen many times where the two interpretations are misunderstood. This is the topic I want to analyze next.

2.1.2 Handedness and Cross Products

To muddy the waters of coordinate system terminology, the Direct3D documentation [Cor] has a section

```
DirectX Graphics | Direct3D 9 | Programming Guide | Getting Started |
Coordinate Systems and Geometry | Coordinate Systems
```

This section is a one-page description of coordinate systems, but unfortunately it is sparse on details and is not precise in its language. The documentation describes right-handed versus left-handed coordinate systems, but assumes that the positive y-axis is the up direction, the positive x-axis is the right direction, and the positive z-axis points into the plane of the page for left-handed coordinates but out of the plane of the page for right-handed coordinates. Later in the documentation you will find this quote:

> Although left-handed and right-handed coordinates are the most common systems, there is a variety of other coordinate systems used in 3D software.

Coordinate systems are either left-handed or right-handed: there are no other choices. The remainder of the quote is

> For example, it is not unusual for 3D modeling applications to use a coordinate system in which the y-axis points toward or away from the viewer, and the z-axis points up. In this case, right-handedness is defined as any positive axis (x, y, or z) pointing toward the viewer. Left-handedness is defined as any positive axis (x, y, or z) pointing away from the viewer.

The terminology here is imprecise. First, coordinate systems may be chosen for which none of the axis direction vectors are the x-, y-, or z-axes. Second, handedness has to do with the order in which you list your vectors and components. The way you draw your coordinate system in a figure is intended to illustrate the handedness, not to define the handedness.

Let us attempt to make the notions precise. The underlying structure for everything we do in three dimensions is the *tuple*. Essentially, we all assume the existence of Cartesian space, as described previously and illustrated in Figure 2.1. Cartesian space is the one on which *all of us* base our coordinate systems. The various spaces such as model space, world space, and view space are all built on top of Cartesian space by specifying a coordinate system. More importantly, *Cartesian space has no preferential directions for view, up, or right*. Those directions are what you specify *when you impose a coordinate system on Cartesian space for an observer to use*.

I have already presented an intuitive way to specify a coordinate system in Equation (2.1). I have imposed the requirement that the coordinate system is right-handed. But what does this really mean? The *geometric interpretation* is shown in Figure 2.2 (a).

In Figure 2.2 (a), the placement of **R** relative to **D** and **U** follows the right-hand rule. The *algebraic interpretation* is the equation $\mathbf{R} = \mathbf{D} \times \mathbf{U}$, which uses the following definition. Given two Cartesian tuples, (x_0, y_0, z_0) and (x_1, y_1, z_1), the cross product is defined by

$$(x_0, y_0, z_0) \times (x_1, y_1, z_1) = (y_0z_1 - z_0y_1, z_0x_1 - x_0z_1, x_0y_1 - y_0x_1) \qquad (2.4)$$

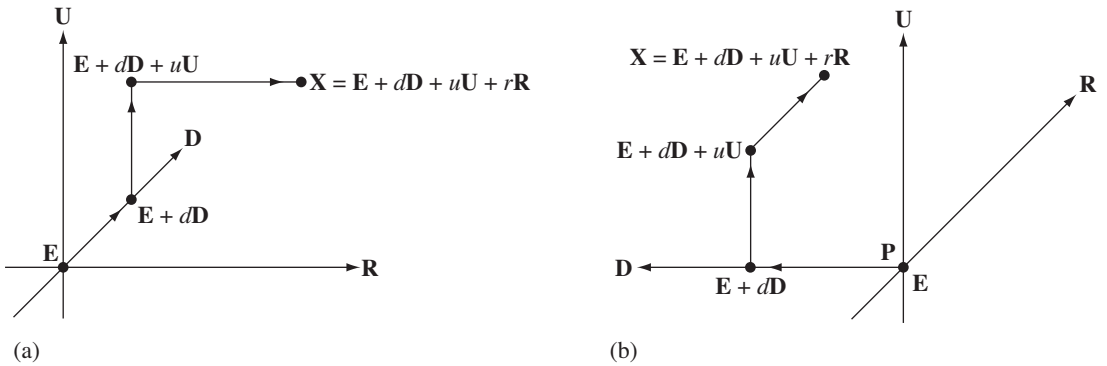


Figure 2.2 (a) A geometric illustration of the right-handed coordinate system in Equation (2.1). (b) A geometric illustration of the left-handed coordinate system in Equation (2.5).

Both the algebraic interpretation and the geometric interpretation are founded on the standard Cartesian coordinate system.

A point **X** is represented in the coordinate system of Equation (2.1) via Equation (2.2), where the components of the coordinate tuple (d, u, r) are computed by Equation (2.3). Instead, I could have used the coordinate system

$$\{\mathbf{E}; \mathbf{R}, \mathbf{U}, \mathbf{D}\} \tag{2.5}$$

with the representation of **X** given by

$$\mathbf{X} = \mathbf{E} + r\mathbf{R} + u\mathbf{U} + d\mathbf{D} \tag{2.6}$$

This coordinate system is left-handed and the coordinate tuple for **X** is (r, u, d). Algebraically, Equations (2.2) and (2.6) produce the same point **X**. All that is different is the bookkeeping, so to speak, which manifests itself as a geometric property as illustrated by Figure 2.2. In the right-handed system shown in Figure 2.2 (a), **D** points into the plane of the page and the last coordinate direction **R** points to the right. It is the case that $\mathbf{D} \times \mathbf{U} = \mathbf{R}$; the last vector is the cross product of the first two vectors. In the left-handed system shown in Figure 2.2 (b), **R** points into the plane of the page and the last coordinate direction **D** points to the left. It is the case that $\mathbf{R} \times \mathbf{U} = -\mathbf{D}$; the last vector is the negative of the cross product of the first two vectors.

Whereas Figure 2.2 illustrates the geometric difference between left-handed and right-handed coordinate systems, the algebraic way to classify whether a coordinate system is left-handed or right-handed is via the cross product operation. Generally, if you have a coordinate system

$$\{\mathbf{P}; \mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2\}$$

where $\mathbf{P}$ is the origin and where the $\mathbf{U}_i$ are unit length and mutually perpendicular, then the coordinate system is *right-handed* when

$$\mathbf{U}_0 \times \mathbf{U}_1 = \mathbf{U}_2 \quad \text{(right-handed)}$$

and is *left-handed* when

$$\mathbf{U}_0 \times \mathbf{U}_1 = -\mathbf{U}_2 \quad \text{(left-handed)}$$

Equivalently, if you write the coordinate direction vectors as the columns of a matrix, say, $Q = [\mathbf{U}_0\ \mathbf{U}_1\ \mathbf{U}_2]$, then Q is an orthogonal matrix. Right-handed systems occur when $\det(Q) = 1$ and left-handed systems occur when $\det(Q) = -1$.

I have more to say about this discussion. Sometimes I read news posts where people say that Direct3D has a "left-handed cross product." This is imprecise terminology, and I will explain why, using an example. Let $\mathbf{A}$ and $\mathbf{B}$ be vectors (not points). The representations of the vectors relative to the coordinate system of Equation (2.1) are

$$\mathbf{A} = d_a\mathbf{D} + u_a\mathbf{U} + r_a\mathbf{R}, \qquad \mathbf{B} = d_b\mathbf{D} + u_b\mathbf{U} + r_b\mathbf{R}$$

and their cross product is

$$\mathbf{A} \times \mathbf{B} = (u_a r_b - r_a u_b)\mathbf{D} + (r_a d_b - d_a r_b)\mathbf{U} + (d_a u_b - u_a d_b)\mathbf{R} \tag{2.7}$$

The representations of the vectors relative to the coordinate system of Equation (2.5) are

$$\mathbf{A} = r_a\mathbf{R} + u_a\mathbf{U} + d_a\mathbf{D}, \qquad \mathbf{B} = r_b\mathbf{R} + u_b\mathbf{U} + d_b\mathbf{D}$$

and their cross product is

$$\mathbf{A} \times \mathbf{B} = (d_a u_b - u_a d_b)\mathbf{R} + (r_a d_b - d_a r_b)\mathbf{U} + (u_a r_b - r_a u_b)\mathbf{D} \tag{2.8}$$

Regardless of which coordinate system was used, Equations (2.7) and (2.8) produce the same vector in Cartesian space.

Now let's work with the coordinates themselves, but with specific instances just to simplify the discussion. Let $\mathbf{A} = \mathbf{D}$ and $\mathbf{B} = \mathbf{U}$. In the coordinate system of Equation (2.1), the coordinate tuple of $\mathbf{A}$ is (1, 0, 0). This says that you have 1 of $\mathbf{D}$ and none of the other two vectors. The coordinate tuple of $\mathbf{B}$ is (0, 1, 0). This says you have 1 of $\mathbf{U}$ and none of the other two vectors. According to Equation (2.4), the cross product of these tuples is

$$(1, 0, 0) \times (0, 1, 0) = (0, 0, 1) \tag{2.9}$$

In the coordinate system of Equation (2.5), the coordinate tuple of $\mathbf{A}$ is (0, 0, 1) since $\mathbf{D}$ is the last vector in the list of coordinate axis directions. The coordinate tuple of $\mathbf{B}$

is (0, 1, 0). According to Equation (2.4), the cross product of these tuples is

$$(0, 0, 1) \times (0, 1, 0) = (-1, 0, 0) \tag{2.10}$$

Whereas Equations (2.7) and (2.8) produce the same Cartesian tuple, Equations (2.9) and (2.10) produce different tuples. Is this a contradiction? No. The tuples (0, 0, 1) and (−1, 0, 0) are not for the Cartesian space; they are coordinate tuples relative to the coordinate systems imposed on Cartesian space. The tuple (0, 0, 1) is relative to the right-handed coordinate system of Equation (2.1), so the actual Cartesian tuple is $0\mathbf{D} + 0\mathbf{U} + 1\mathbf{R} = \mathbf{R}$; that is, $\mathbf{A} \times \mathbf{B} = \mathbf{D} \times \mathbf{U} = \mathbf{R}$. In Figure 2.2 (a), you obtain **R** from **D** and **U** by using the right-hand rule. Similarly, the tuple (−1, 0, 0) is relative to the left-handed coordinate system of Equation (2.5), so the actual Cartesian tuple is $-1\mathbf{R} + 0\mathbf{U} + 0\mathbf{D} = -\mathbf{R}$; that is, $\mathbf{A} \times \mathbf{B} = \mathbf{D} \times \mathbf{U} = -\mathbf{R}$. In Figure 2.2 (b), you obtain $-\mathbf{R}$ from **D** and **U** by using the left-hand rule. Table 2.1 summarizes our findings when $\mathbf{A} = \mathbf{D}$ and $\mathbf{B} = \mathbf{U}$.

If you compute cross products using coordinate tuples, as shown in Equation (2.10) for a left-handed camera coordinate system, you have a left-handed cross product, so to speak. But if you compute cross products using right-handed Cartesian tuples, as shown in Equation (2.8) also for a left-handed camera coordinate system, you wind up with a right-handed cross product, so to speak. This means you have to be very careful when computing cross products, making certain you know *which* coordinate system you are working with. The Direct3D function for computing the cross product does not care about the coordinate system:

```
D3DXVECTOR A = <a tuple (x0,y0,z0)>;
D3DXVECTOR B = <a tuple (x1,y1,z1)>;
D3DXVECTOR C;
D3DXVec3Cross(&C,&A,&B); // C = (y0 z1 - z0 y1, z0 x1 - x0 z1, x0 y1 - y0 x1)
```

The function simply implements Equation (2.4) without regard to which coordinate system the "tuples" `A` and `B` come from. The function knows algebra. You are the one who imposes geometry.

Table 2.1 Summary of handedness and cross product calculations.

Equation	*Result*	*Coordinate System Handedness*	*Cross Product Applied To*
(2.7)	$\mathbf{D} \times \mathbf{U} = \mathbf{R}$	Right-handed	Cartesian tuples
(2.8)	$\mathbf{D} \times \mathbf{U} = \mathbf{R}$	Left-handed	Cartesian tuples
(2.9)	$\mathbf{D} \times \mathbf{U} = \mathbf{R}$	Right-handed	Coordinate tuples
(2.10)	$\mathbf{D} \times \mathbf{U} = -\mathbf{R}$	Left-handed	Coordinate tuples

2.1.3 Points and Vectors

You might have noticed that I have referred to points and to vectors. These two concepts are considered distinct and are the topics of *affine algebra*. A point is *not* a vector and a vector is *not* a point. To distinguish between points and vectors within text, some authors use different fonts. For the sake of argument, let us do so and consider three points $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$ and a vector $\mathbf{V}$. The following operations are axioms associated with affine algebra, but once again to be loose with the notation, I will just say the following:

1. The difference of the two points is a vector, $\mathbf{V} = \mathcal{P} - \mathcal{Q}$.
2. The sum of a point and a vector is a point, $\mathcal{Q} = \mathcal{P} + \mathbf{V}$.
3. $(\mathcal{P} - \mathcal{Q}) + (\mathcal{Q} - \mathcal{R}) = (\mathcal{P} - \mathcal{R})$. The intuition for this is to draw a triangle whose vertices are the three points. This equation says that the sum of the directed edges of the triangle is the zero vector.

In the third axiom, rather than appealing to the geometry of a triangle, you might be tempted to remove the parentheses. In fact, the removal of the parentheses is a *consequence* of this axiom, but you have to be careful in doing such things that, at first glance, seem intuitive. The axioms do not allow you to *add points* in any manner you like; for example, the expression $\mathcal{P} + \mathcal{Q}$ is not valid. However, additional axioms may be postulated that allow a special form of addition in expressions called *affine combinations*. Specifically, the following axioms support this:

4. $\mathcal{P} = \sum_{i=1}^{n} c_i \mathcal{P}_i$ is a point, where $\sum_{i=1}^{n} c_i = 1$.
5. $\mathbf{V} = \sum_{i=1}^{n} d_i \mathcal{P}_i$ is a vector, where $\sum_{i=1}^{n} d_i = 0$.

With the additional axioms, an expression such as

$$(1/3)\mathcal{P} + (1/3)\mathcal{Q} + (1/3)\mathcal{R}$$

is valid since the coefficients sum to one. This example produces the average of the points. The expression

$$(\mathcal{P} - \mathcal{Q}) + (\mathcal{Q} - \mathcal{R}) = \mathcal{P} - \mathcal{Q} + \mathcal{Q} - \mathcal{R}$$

is valid. The implied coefficients of the points on the right-hand side are 1, -1, 1, and -1 (in that order), and their sum is zero. Thus, the expression on the right-hand side is a vector.

At the beginning of this chapter, I promised not to delve into the finer mathematical details of the topics. I have done so here, but for practical reasons. Some graphics programmers choose to enforce the distinction between points and vectors, say, in an object-oriented language such as C++. To hint at the topic of Section 2.2.5, a fourth component is provided for both points and vectors. Points are represented as

4-tuples of the form $(x, y, z, 1)$ and vectors are represented as 4-tuples of the form $(x, y, z, 0)$. The five axioms for affine algebra are satisfied by these representations.

Axiom 1 is satisfied,

$$(x_0, y_0, z_0, 1) - (x_1, y_1, z_1, 1) = (x_0 - x_1, y_0 - y_1, z_0 - z_1, 0) \tag{2.11}$$

Axiom 2 is satisfied,

$$(x_0, y_0, z_0, 1) + (x_1, y_1, z_1, 0) = (x_0 + x_1, y_0 + y_1, z_0 + z_1, 1) \tag{2.12}$$

Axiom 3 is satisfied,

$$\begin{aligned}
&((x_0, y_0, z_0, 1) - (x_1, y_1, z_1, 1)) + ((x_1, y_1, z_1, 1) - (x_2, y_2, z_2, 1)) \\
&\quad = (x_0 - x_1, y_0 - y_1, z_0 - z_1, 0) + (x_1 - x_2, y_1 - y_2, z_1 - z_2, 0) \\
&\quad = (x_0 - x_2, y_0 - y_2, z_0 - z_2, 0) \\
&\quad = ((x_0, y_0, z_0, 1) - (x_2, y_2, z_2, 1))
\end{aligned}$$

Axiom 4 is satisfied,

$$\begin{aligned}
\sum_{i=1}^{n} c_i(x_i, y_i, z_i, 1) &= \left(\sum_{i=1}^{n} c_i x_i, \sum_{i=1}^{n} c_i y_i, \sum_{i=1}^{n} c_i z_i, \sum_{i=1}^{n} c_i \right) \\
&= \left(\sum_{i=1}^{n} c_i x_i, \sum_{i=1}^{n} c_i y_i, \sum_{i=1}^{n} c_i z_i, 1 \right)
\end{aligned} \tag{2.13}$$

where I used the fact that the c_i sum to one. Axiom 5 is satisfied,

$$\begin{aligned}
\sum_{i=1}^{n} d_i(x_i, y_i, z_i, 1) &= \left(\sum_{i=1}^{n} d_i x_i, \sum_{i=1}^{n} d_i y_i, \sum_{i=1}^{n} d_i z_i, \sum_{i=1}^{n} d_i \right) \\
&= \left(\sum_{i=1}^{n} d_i x_i, \sum_{i=1}^{n} d_i y_i, \sum_{i=1}^{n} d_i z_i, 0 \right)
\end{aligned} \tag{2.14}$$

where I used the fact that the d_i sum to zero.

Two classes are implemented, one called `Point` and one called `Vector`. The interface for the `Vector` class has minimally the following structure:

```
class Vector
{
public:
    // 'this' is vector U
    Vector operator+ (Vector V) const;                  // U + V
    Vector operator- (Vector V) const;                  // U - V
```

```
    Vector operator* (float c) const;                   // V*c
    friend Vector operator* (float c, Vector V) const;  // c*V

private:
    float tuple[4];  // tuple[3] = 0 always
};
```

The interface for the `Point` class has minimally the following structure:

```
class Point
{
public:
   // 'this' is point P
   Point operator+ (Vector V);  // P + V, Equation (2.12)
   Point operator- (Vector V);  // P - V, Equation (2.12)
   Vector operator- (Point Q);  // P - Q, Equation (2.11)
   static Point AffineCSum (int N, float c[], Point Q[]); // Equation (2.13)
   static Vector AffineDSum (int N, float d[], Point Q[]); // Equation (2.14)

private:
   float tuple[4];  // tuple[3] = 1 always
};
```

I have shown the points and vectors stored as 4-tuples. The fourth component should be private so that applications cannot access them and inadvertently change them. Thus, if you were to support an `operator[]` member function, you would need to trap attempts to access the fourth component (via assertions, exceptions, or some other mechanism). It is possible to use 3-tuples, relying on the fact that the class names themselves imply the correct fourth component. Given current CPUs and game consoles, it is better, though, to have 16-byte (4-float) alignment of data because the hardware expects it in order to perform well.

Two issues come to mind. First, having classes `Point` and `Vector` means maintaining more code than having just a single class to represent points and vectors. The amount of additional source code might not be of concern to you. Second, and perhaps the more important issue, is that the clipping process occurs using 4-tuples of the form (x, y, z, w), where the fourth component w is not necessarily 0 or 1. Clipping involves arithmetic operations on 4-tuples, but the `Point` class does not support this when it insists on $w = 1$. If you were to allow the fourth component to be public and not force it to be 1, you would be violating the purist attempt to distinguish between points and vectors. You could implement yet another class, say, `HPoint`, and represent the general 4-tuples, but this means maintaining even more code. My opinion is to keep it simple and just support a vector class, keeping the distinction between points and vectors in your own mind, being consistent about it when coding, and not maintaining additional code.

2.2 Transformations

We want to construct functions that map points in 3D space to other points in 3D space. The motivation was provided in the last section: taking an object built in model space and placing it in world space. The transformations considered in this section are the simplest ones you encounter in computer graphics.

2.2.1 Linear Transformations

The topic of *linear transformations* is usually covered in a course on linear algebra. Such transformations are applied to vectors rather than points. I will not give a detailed overview here. You can read about the topic in any standard textbook on linear algebra.

The basic idea is to construct functions of the form $\mathbf{Y} = \mathbf{L}(\mathbf{X})$. The input to the function is the vector $\mathbf{X}$ and the output is the vector $\mathbf{Y}$. The function name itself is also typeset as a vector, namely, $\mathbf{L}$, to indicate that its output is a vector. A *linear function* or *linear transformation* is defined to have the following property:

$$\mathbf{L}(c\mathbf{U} + \mathbf{V}) = c\mathbf{L}(\mathbf{U}) + \mathbf{L}(\mathbf{V}) \tag{2.15}$$

where c is a scalar. The expression $c\mathbf{U} + \mathbf{V}$ is called a *linear combination* of the two vectors. In words, Equation (2.15) says that the function value of a linear combination is the linear combination of the function values.

Example 2.1 Let $\mathbf{X} = (x_0, x_1, x_2)$ and $\mathbf{Y} = (y_0, y_1, y_2)$. The function $\mathbf{Y} = \mathbf{L}(\mathbf{X})$, where $\mathbf{L}(x_0, x_1, x_2) = (x_0 + x_1, 2x_0 - x_2, 3x_0 + x_1 + 2x_2)$, is a linear transformation. To verify this, apply the function to the linear combination $c\mathbf{U} + \mathbf{V}$, where $\mathbf{U} = (u_0, u_1, u_2)$, $\mathbf{V} = (v_0, v_1, v_2)$, and c is a scalar:

$$\begin{aligned}
\mathbf{L}(c\mathbf{U} + \mathbf{V}) &= \mathbf{L}(c(u_0, u_1, u_2) + (v_0, v_1, v_2)) \\
&= \mathbf{L}(cu_0 + v_0, cu_1 + v_1, cu_2 + v_2) \\
&= ((cu_0 + v_0) + (cu_1 + v_1), 2(cu_0 + v_0) - (cu_2 + v_2), \\
&\quad 3(cu_0 + v_0) + (cu_1 + v_1) + 2(cu_2 + v_2)) \\
&= (c(u_0 + u_1) + (v_0 + v_1), c(2u_0 - u_2) + (2v_0 - v_2), \\
&\quad c(3u_0 + u_1 + 2u_2) + (3v_0 + v_1 + 2v_2)) \\
&= c(u_0 + u_1, 2u_0 - u_2, 3u_0 + u_1 + 2u_2) \\
&\quad + (v_0 + v_1, 2v_0 - v_2, 3v_0 + v_1 + 2v_2) \\
&= c\mathbf{L}(u_0, u_1, u_2) + \mathbf{L}(v_0, v_1, v_2) \\
&= c\mathbf{L}(\mathbf{U}) + \mathbf{L}(\mathbf{V})
\end{aligned}$$

These algebraic steps verify that $\mathbf{L}(c\mathbf{U} + \mathbf{V}) = c\mathbf{L}(\mathbf{U}) + \mathbf{V}$, so the function is linear. ■

EXAMPLE 2.2 The function $\mathbf{L}(x_0, x_1, x_2) = x_0^2$ is not a linear transformation. To verify this, it is enough to show that the function applied to *one* linear combination is not the linear combination of the function results. For example, let $c = 2$, $\mathbf{U} = (1, 0, 0)$, and $\mathbf{V} = (0, 0, 0)$; then $\mathbf{L}(\mathbf{U}) = \mathbf{L}(1, 0, 0) = 1$, $\mathbf{L}(\mathbf{V}) = \mathbf{L}(0, 0, 0) = 0$, and $\mathbf{L}(c\mathbf{U} + \mathbf{V}) = \mathbf{L}(2, 0, 0) = 4$. Also, $2\mathbf{L}(\mathbf{U}) + \mathbf{L}(\mathbf{V}) = 2\mathbf{L}(1, 0, 0) + \mathbf{L}(0, 0, 0) = 2$. This specific case does not satisfy the constraint $\mathbf{L}(c\mathbf{U} + \mathbf{V}) = c\mathbf{L}(\mathbf{U}) + \mathbf{L}(\mathbf{V})$, so the function is not linear. ■

EXAMPLE 2.3 Translation of a vector is not linear. The translation function is $\mathbf{L}(x_0, x_1, x_2) = (x_0, x_1, x_2) + (b_0, b_1, b_2)$, where $(b_0, b_1, b_2) \neq (0, 0, 0)$ is the vector used to translate any point in space. If you choose $c = 1$, $\mathbf{U} = (1, 1, 1)$, and $\mathbf{V} = (0, 0, 0)$, then

$$\mathbf{L}(\mathbf{U} + \mathbf{V}) = \mathbf{L}(1, 1, 1) = (1 + b_0, 1 + b_1, 1 + b_2)$$

which is different from

$$\begin{aligned}\mathbf{L}(\mathbf{U}) + \mathbf{L}(\mathbf{V}) &= \mathbf{L}(1, 1, 1) + \mathbf{L}(0, 0, 0) = (1 + b_0, 1 + b_1, 1 + b_2) + (b_0, b_1, b_2) \\ &= (1 + 2b_0, 1 + 2b_1, 1 + 2b_2)\end{aligned}$$

Translation is, however, an example of an *affine transformation*, which I discuss later in this section. ■

Linear transformations are convenient to use because they have a representation that makes them easy to implement and compute in a program. Let us write the transformation input $\mathbf{X}$ and output $\mathbf{Y}$ as column vectors (3×1 vectors). A linear transformation is necessarily of the form

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} m_{00}x_0 + m_{01}x_1 + m_{02}x_2 \\ m_{10}x_0 + m_{11}x_1 + m_{12}x_2 \\ m_{20}x_0 + m_{21}x_1 + m_{22}x_2 \end{bmatrix} \tag{2.16}$$

where the coefficients m_{ij} of the 3×3 matrix are constants. The more compact notation is

$$\mathbf{Y} = M\mathbf{X} \tag{2.17}$$

where M is a 3×3 matrix. This form is suggestive of the one-dimensional case $y = mx$, which is the equation of a straight line that passes through the origin. The geometry of linear functions in higher dimensions is slightly more complicated. Once again, I refer you to any standard textbook on linear algebra to see the details.

By using the representation $\mathbf{Y} = M\mathbf{X}$, I have already chosen a convention regarding the manipulation of vectors and matrices. Vectors are columns for me, and the application of a matrix to a vector puts the matrix on the left and the vector on the

right. This is the mathematician in me speaking—I chose what I was raised with. OpenGL and Direct3D choose the opposite convention, which is to represent vectors as rows and to apply a matrix to a vector by putting the vector on the left and the matrix on the right. This is the usual convention chosen by computer graphics people. There is no right or wrong for choosing your conventions. For any choice you make, you will find users who disagree with that choice because they have made another choice. The fact is, *you* make your decisions. *You* live by the consequences. What is important is to ensure that you have documented your engine and code well, making it clear to clients exactly what your choices are. There are quite a few other conventions you must decide on when designing a computer graphics system. A comparison of the conventions for transformations is provided in Section 2.8 regarding Wild Magic, OpenGL, and Direct3D.

EXERCISE 2.1 Verify that the function defined by Equation (2.16) is a linear transformation. ■

Computer graphics has a collection of linear transformations that arise frequently in practice. These are presented here.

Rotation

The motivation for 3D rotation comes from two dimensions, where a rotation matrix is

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = I + (\sin\theta)S + (1-\cos\theta)S^2 \tag{2.18}$$

where I is the identity matrix and S is the skew-symmetric matrix, as shown:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad S = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

I have chosen to factor the matrix R in terms of I, S, and θ because it is suggestive of how to build the matrix for rotation about an arbitrary axis. For a positive angle θ, $R\mathbf{V}$ rotates the 2×1 vector $\mathbf{V}$ *counterclockwise* about the origin, as shown in Figure 2.3. Whether a positive angle represents a counterclockwise or a clockwise rotation is yet another convention you must choose and make clear to your users.

In three dimensions, the matrix representing a rotation in the xy-plane is

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = I + (\sin\theta)S + (1-\cos\theta)S^2 \tag{2.19}$$

where

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad S = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Assuming a choice of coordinate axes as is shown in Figure 2.1, the direction of rotation is counterclockwise about the z-axis when viewed by an observer who is on the positive z-side of the xy-plane and looking at the plane with view direction in the negative z-direction, $(0, 0, -1)$. Think of Figure 2.3 as what such an observer sees. In that figure, the positive z-direction is out of the page; the negative z-direction is into the page. A 3D view is shown in Figure 2.4 (a).

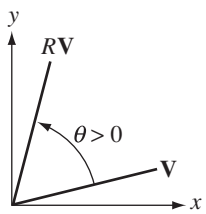


Figure 2.3 A positive angle corresponds to a counterclockwise rotation.

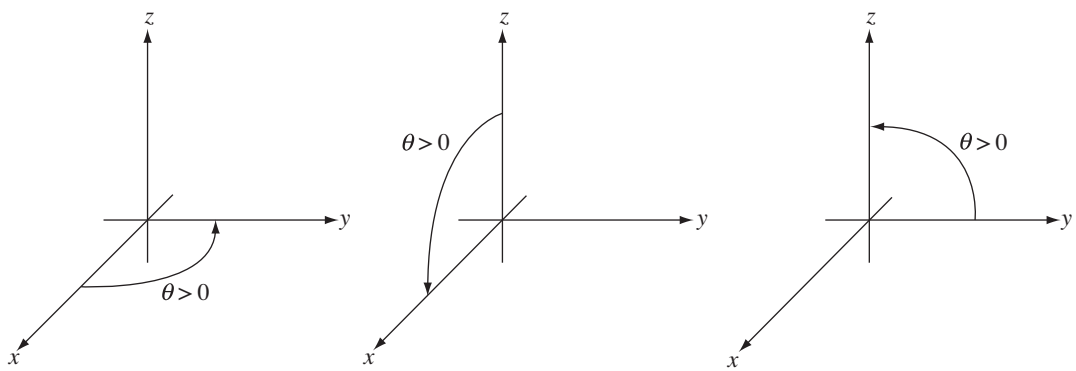


Figure 2.4 Rotations about the coordinate axes. (a) A positive-angle rotation about the z-axis. (b) A positive-angle rotation about the y-axis. (c) A positive-angle rotation about the x-axis.

Similar rotation matrices may be constructed for rotations about the other coordinate axes. The matrix representing a rotation in the xz-plane is

$$R = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} = I + (\sin\theta)S + (1-\cos\theta)S^2 \qquad (2.20)$$

where

$$S = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

Figure 2.4 (b) is a 3D view of a positive-angle rotation about the y-axis, which is a counterclockwise rotation in the xz-plane as shown. The matrix representing a rotation in the yz-plane is

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} = I + (\sin\theta)S + (1-\cos\theta)S^2 \qquad (2.21)$$

where

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

Figure 2.4 (c) is a 3D view of a positive-angle rotation about the x-axis, which is a counterclockwise rotation in the yz-plane as shown.

In general, the matrix representing a rotation about the axis with direction vector (u_0, u_1, u_2) is as shown. Define the skew-symmetric matrix

$$S = \begin{bmatrix} 0 & -u_2 & u_1 \\ u_2 & 0 & -u_0 \\ -u_1 & u_0 & 0 \end{bmatrix}$$

then the rotation matrix is

$$\begin{aligned} R &= I + (\sin\theta)S + (1-\cos\theta)S^2 \\ &= \begin{bmatrix} \gamma + (1-\gamma)u_0^2 & -u_2\sigma + (1-\gamma)u_0u_1 & +u_1\sigma + (1-\gamma)u_0u_2 \\ +u_2\sigma + (1-\gamma)u_0u_1 & \gamma + (1-\gamma)u_1^2 & -u_0\sigma + (1-\gamma)u_1u_2 \\ -u_1\sigma + (1-\gamma)u_0u_2 & +u_0\sigma + (1-\gamma)u_1u_2 & \gamma + (1-\gamma)u_2^2 \end{bmatrix} \end{aligned} \qquad (2.22)$$

where $\sigma = \sin\theta$ and $\gamma = \cos\theta$.

EXERCISE 2.2 Verify that Equation (2.22) produces the coordinate plane rotations mentioned in Equations (2.19), (2.20), and (2.21). Also verify that the S-matrix in Equation (2.22) produces the S-matrices for the coordinate plane rotations. ■

EXERCISE 2.3 Using algebraic methods, construct the formula of Equation (2.22). ■

Reflection

A plane passing through the origin is represented algebraically by the equation $\mathbf{N} \cdot \mathbf{X} = 0$, where $\mathbf{N}$ is a unit-length vector perpendicular to the plane and $\mathbf{X}$ is any point on the plane. Figure 2.5 (a) provides a 3D view of the plane, a vector $\mathbf{V}$, and the reflection $\mathbf{W}$ of $\mathbf{V}$ through the plane. Figure 2.5 (b) shows a 2D side view.

The vector $\mathbf{N}^\perp$ is a point on the plane and is the midpoint of the line segment connecting $\mathbf{V}$ and its reflection $\mathbf{U}$. The superscript symbol $\perp$ denotes perpendicularity. In this case, $\mathbf{N}^\perp$ is a vector perpendicular to $\mathbf{N}$.

The side view gives you a good idea of how the vectors are related algebraically. The input vector is the linear combination

$$\mathbf{V} = c\mathbf{N} + \mathbf{N}^\perp$$

for the scalar c. The perpendicular component $\mathbf{N}^\perp$ is naturally dependent on your choice of $\mathbf{V}$. In fact, the scalar is easily determined to be $c = \mathbf{N} \cdot \mathbf{V}$, which uses the conditions that $\mathbf{N}$ is unit length and that $\mathbf{N}$ and $\mathbf{N}^\perp$ are perpendicular. The reflection vector is the linear combination

$$\mathbf{U} = -c\mathbf{N} + \mathbf{N}^\perp$$

The difference is that the normal component of $\mathbf{V}$ is negated to form the normal component for $\mathbf{U}$; this is the reflection. Subtracting the two equations and using the

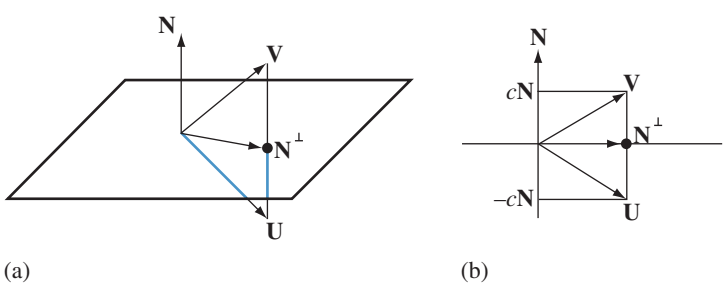


Figure 2.5 The reflection of a vector through a plane. (a) A 3D view. (b) A 2D side view.

formula for c leads to

$$\mathbf{U} = \mathbf{V} - 2(\mathbf{N} \cdot \mathbf{V})\mathbf{N} = \left(I - 2\mathbf{N}\mathbf{N}^{\mathrm{T}}\right)\mathbf{V}$$

where I is the 3×3 identity matrix and the superscript T denotes the transpose operation. Using my conventions, $\mathbf{N}$ is a 3×1 (column) vector, which makes $\mathbf{N}^{\mathrm{T}}$ a 1×3 (row) vector. The product $\mathbf{N}\mathbf{N}^{\mathrm{T}}$ is a 3×3 matrix. Notice that $\mathbf{N}^{\mathrm{T}}\mathbf{N}$ is the product in the other order but is necessarily a scalar (a 1×1 matrix as it were).

If $\mathbf{N} = (n_0, n_1, n_2)$, the reflection matrix is

$$R = I - \mathbf{N}\mathbf{N}^{\mathrm{T}} = \begin{bmatrix} 1 - n_0^2 & -n_0 n_1 & -n_0 n_2 \\ -n_0 n_1 & 1 - n_1^2 & -n_1 n_2 \\ -n_0 n_2 & -n_1 n_2 & 1 - n_2^2 \end{bmatrix} \tag{2.23}$$

Rotation and reflection matrices are said to be *orthogonal matrices*. An orthogonal matrix M has the property that $MM^{\mathrm{T}} = M^{\mathrm{T}}M = I$, which says that the inverse operation of M is its transpose. Apply M to a vector $\mathbf{V}$ to obtain $\mathbf{U} = M\mathbf{V}$. Now apply M^{T} to obtain $M^{\mathrm{T}}\mathbf{U} = M^{\mathrm{T}}M\mathbf{V} = I\mathbf{V} = \mathbf{V}$. Geometrically, if R is a rotation matrix that rotates a vector about an axis by θ radians, R^{T} is a rotation matrix about the same axis that rotates a vector by $-\theta$ radians. If R is a reflection matrix through a plane $\mathbf{N} \cdot \mathbf{X} = 0$, then $R^{\mathrm{T}} = R$, which says that if you reflect twice, you end up where you started.

One distinguishing algebraic characteristic between rotations and reflections is the value of their determinants. Recall that the determinant of a 3×3 matrix M is

$$\begin{aligned} \det(M) &= \det \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \\ &= m_{00}m_{11}m_{22} + m_{01}m_{12}m_{20} + m_{02}m_{10}m_{21} \\ &\quad - m_{02}m_{11}m_{20} - m_{01}m_{10}m_{22} - m_{00}m_{12}m_{21} \end{aligned} \tag{2.24}$$

The determinant of a rotation matrix is 1. The determinant of a reflection matrix is -1.

EXERCISE 2.4 Verify that the determinant of the matrix in Equation (2.22) is 1. Verify that the determinant of the matrix in Equation (2.23) is -1. ■

Scaling

Scaling is a simple transformation. You scale each component of a vector by a desired amount: $(y_0, y_1, y_2) = (s_0x_0, s_1x_1, s_2x_2)$, where s_0, s_1, and s_2 are the scaling factors. The matrix that represents scaling is

$$S = \begin{bmatrix} s_0 & 0 & 0 \\ 0 & s_1 & 0 \\ 0 & 0 & s_2 \end{bmatrix} \tag{2.25}$$

This is a diagonal matrix. Usually, we assume that the scaling factors are positive numbers, but 3D modeling packages tend to allow you to set them to negative numbers, which can cause all sorts of problems for exporters and engines that rely on positive scales. If the scales are all the same value, $s_0 = s_1 = s_2$, the transformation is said to be a *uniform scaling*; otherwise, it is a *nonuniform scaling*. Nonuniform scales can also lead to problems in the design of a graphics engine. I will get into the details of this later in the design of a scene graph hierarchy for which each node stores rotation and scaling matrices and translation vectors but also stores a composite matrix for the entire transformation. Given a composite matrix, a frequently asked question is how to extract the rotational component and the scaling factors. We will see that this is an ill-posed problem; see Section 17.5.

The scaling matrix of Equation (2.25) represents scaling in the directions of the coordinate axes. It is possible to scale in different directions. For example, if you want to scale the vectors by s in the direction $\mathbf{D}$, you need to decompose the input point $\mathbf{X}$ into a $\mathbf{D}$ component and a remainder:

$$\mathbf{X} = d\mathbf{D} + \mathbf{R}$$

where $\mathbf{R}$ is perpendicular to $\mathbf{D}$. The decomposition is similar to what was used to construct reflections. The component of $\mathbf{X}$ in the $\mathbf{D}$ direction is $d\mathbf{X}$, where $d = \mathbf{D} \cdot \mathbf{X}$. The vector scaled in the $\mathbf{D}$ direction is

$$\mathbf{Y} = sd\mathbf{D} + \mathbf{R}$$

and is illustrated in Figure 2.6.

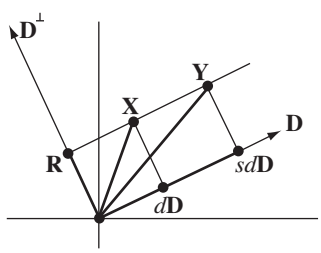


Figure 2.6 Scaling of a vector $\mathbf{X}$ by a scaling factor s in the direction $\mathbf{D}$ to produce a vector $\mathbf{Y}$.

Some algebra will show that

$$\mathbf{Y} = \left(s\mathbf{D}\mathbf{D}^{\mathrm{T}} + \mathbf{D}^{\perp}\left(\mathbf{D}^{\perp}\right)^{\mathrm{T}} \right) \mathbf{X}$$

where $\mathbf{D}^{\perp}$ is a unit-length vector perpendicular to the unit-length vector $\mathbf{D}$.

Generally, in three dimensions you can apply scaling in three noncoordinate axis directions $\mathbf{D}$, $\mathbf{U}$, and $\mathbf{R}$ by representing the input point in this new coordinate system as

$$\mathbf{X} = d\mathbf{D} + u\mathbf{U} + r\mathbf{R}$$

and then scaling each of the components:

$$\mathbf{Y} = s_0 d\mathbf{D} + s_1 u\mathbf{U} + s_2 r\mathbf{R}$$

In matrix form, this becomes

$$\mathbf{Y} = \left(s_0\mathbf{D}\mathbf{D}^{\mathrm{T}} + s_1\mathbf{U}\mathbf{U}^{\mathrm{T}} + s_2\mathbf{R}\mathbf{R}^{\mathrm{T}} \right) \mathbf{X} \tag{2.26}$$

EXERCISE 2.5 Construct Equation (2.26). *Hint:* Use the fact that $d = \mathbf{D} \cdot \mathbf{X}$, $u = \mathbf{U} \cdot \mathbf{X}$, and $r = \mathbf{R} \cdot \mathbf{X}$. Use the construction that led to Equation (2.23) to help you decide how to rearrange the various vector terms appropriately. ■

A more intuitive equation for general scaling than Equation (2.26) is obtained by using a matrix $M = [\mathbf{D}\ \mathbf{U}\ \mathbf{R}]$. The notation means that the first column of M is the 3×1 vector $\mathbf{D}$, the second column is the 3×1 vector $\mathbf{U}$, and the third column is the 3×1 vector $\mathbf{R}$. The transpose of M is also written in a concise form,

$$M^{\mathrm{T}} = \begin{bmatrix} \mathbf{D}^{\mathrm{T}} \\ \mathbf{U}^{\mathrm{T}} \\ \mathbf{R}^{\mathrm{T}} \end{bmatrix}$$

The notation means that the first row of M is the 1×3 vector $\mathbf{D}^{\mathrm{T}}$, the second row of M is the 1×3 vector $\mathbf{U}^{\mathrm{T}}$, and the third row of M is the 1×3 vector $\mathbf{R}^{\mathrm{T}}$. Let S be the diagonal matrix of Equation (2.25). The scaling matrix becomes

$$s_0\mathbf{D}\mathbf{D}^{\mathrm{T}} + s_1\mathbf{U}\mathbf{U}^{\mathrm{T}} + s_2\mathbf{R}\mathbf{R}^{\mathrm{T}} = [\mathbf{D}\ \mathbf{U}\ \mathbf{R}] \begin{bmatrix} s_0 & 0 & 0 \\ 0 & s_1 & 0 \\ 0 & 0 & s_2 \end{bmatrix} \begin{bmatrix} \mathbf{D}^{\mathrm{T}} \\ \mathbf{U}^{\mathrm{T}} \\ \mathbf{R}^{\mathrm{T}} \end{bmatrix} = MSM^{\mathrm{T}} \tag{2.27}$$

Both M and M^{T} are rotation matrices since we are assuming that our coordinate systems have direction vectors that are mutually perpendicular and form a right-handed system. In words, M^{T} rotates $\mathbf{X}$ to the new coordinate system, S scales the rotated vector, and M rotates the result back to the old coordinate system.

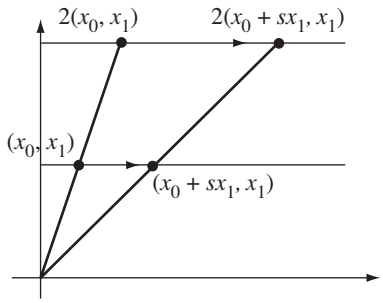


Figure 2.7 Shearing of points (x_0, x_1) in the x_0 direction. As the x_1 value increases for points, the amount of shearing in the x_0 direction increases.

Shearing

Shearing operations are applied less often than rotations, reflections, and scalings, but I include them here anyway since they tend to be grouped into those transformations of interest in computer graphics. To motivate the idea, consider a shearing in two dimensions, as illustrated in Figure 2.7.

In two dimensions, the shearing in the x_0 direction has the matrix representation

$$S = \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix}$$

This maps the point (x_0, x_1) to $(y_0, y_1) = (x_0 + sx_1, x_1)$, as shown in Figure 2.7. If you want to shear in the x_1 direction, the matrix is similar,

$$S = \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix}$$

This maps the point (x_0, x_1) to $(y_0, y_1) = (x_0, x_1 + sx_0)$.

In three dimensions, shearing is applied within planes. For example, if you want to shear within the planes parallel to the x_0x_1 plane, you would use the matrix

$$S_{01} = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.28}$$

to shear within each plane $x_2 = c$ (constant) in the direction of x_0. Notice that (x_0, x_1, x_2) is mapped to $(x_0 + sx_1, x_1, x_2)$, so only the x_0 value is changed by the

shearing. If you want to shear within these planes, but in the x_1 direction, you would use the matrix

$$S_{10} = \begin{bmatrix} 1 & 0 & 0 \\ s & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.29}$$

In the two equations, the subscripts on the matrix names refer to the indices of the matrix entries that contain the shearing factors.

Similarly, you can shear in the planes $x_1 = c$ (constant) in the x_0 direction by using the matrix

$$S_{02} = \begin{bmatrix} 1 & 0 & s \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.30}$$

or you can shear in the x_2 direction by using the matrix

$$S_{20} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ s & 0 & 1 \end{bmatrix} \tag{2.31}$$

You can shear in the planes $x_0 = c$ (constant) in the x_1 direction by using the matrix

$$S_{12} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & s \\ 0 & 0 & 1 \end{bmatrix} \tag{2.32}$$

or you can shear in the x_2 direction by using the matrix

$$S_{21} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & s & 1 \end{bmatrix} \tag{2.33}$$

Shearing in an arbitrary plane containing the origin is possible. The construction of the matrix is similar to what was done for scaling; see Equation (2.27) and the discussion leading to it. You rotate the input point $\mathbf{X}$ to the coordinate system of interest, apply one of the coordinate scaling transformations, and then rotate the result back to the original coordinate system. The general transformation is

$$[\,\mathbf{D} \quad \mathbf{U} \quad \mathbf{R}\,]\, S_{ij} \begin{bmatrix} \mathbf{D}^{\mathrm{T}} \\ \mathbf{U}^{\mathrm{T}} \\ \mathbf{R}^{\mathrm{T}} \end{bmatrix} \tag{2.34}$$

where S_{ij} is one of the shearing matrices from Equations (2.28) through (2.33).

2.2.2 Affine Transformations

As noted previously, translation is not a linear transformation. However, it is what distinguishes an *affine transformation* from a linear one. The definition of a linear transformation says that a transformation of a linear combination is a linear combination of the transformations. In mathematical terms, if c_i are scalars and $\mathbf{V}_i$ are vectors for $1 \le i \le n$, and if $\mathbf{L}$ is a linear transformation, then

$$\mathbf{L}\left(\sum_{i=1}^{n} c_i \mathbf{V}_i\right) = \sum_{i=1}^{n} c_i \mathbf{L}(\mathbf{V}_i)$$

An affine transformation must deal with the distinction between points and vectors. For the moment, I will switch back to the typesetting conventions for points and vectors. Let $\mathcal{P}$ and $\mathcal{Q}$ be points. Let $\mathcal{Q} = \mathcal{A}(\mathcal{P})$ be a transformation that maps points to points. The transformation is an *affine transformation* when it satisfies the following conditions:

1. Consider points $\mathcal{P}_i$ and $\mathcal{Q}_i = \mathcal{A}(\mathcal{P}_i)$ for $1 \le i \le 4$. If $\mathcal{P}_2 - \mathcal{P}_1 = \mathcal{P}_4 - \mathcal{P}_3$, then it must be that $\mathcal{Q}_2 - \mathcal{Q}_1 = \mathcal{Q}_4 - \mathcal{Q}_3$.
2. If $\mathbf{X} = \mathcal{P}_2 - \mathcal{P}_1$ and $\mathbf{Y} = \mathcal{Q}_2 - \mathcal{Q}_1$, then the transformation $\mathbf{Y} = \mathbf{L}(\mathbf{X})$ must be a linear transformation.

This is a fancy mathematical way of saying that an affine transformation is composed of two parts, one part that maps a point to a point and one part that maps a vector to a vector. Suppose that the linear transformation is written in matrix form, $\mathbf{Y} = M\mathbf{X}$, for some matrix M. The second condition in the definition implies

$$\mathcal{P}_2 = \mathcal{P}_1 + \mathbf{X} \tag{2.35}$$

and

$$\mathcal{Q}_2 = \mathcal{Q}_1 + \mathbf{Y} = \mathcal{Q}_1 + M\mathbf{X}$$

Since $\mathcal{Q}_1$ is a point, we can write it as an offset from $\mathcal{P}_1$; namely,

$$\mathcal{Q}_1 = \mathcal{P}_1 + \mathbf{B}$$

for some vector $\mathbf{B}$. Combining the last two displayed equations, we have

$$\mathcal{Q}_2 = \mathcal{P}_1 + (M\mathbf{X} + \mathbf{B}) \tag{2.36}$$

The affine transformation between points is

$$\mathcal{Q}_2 = \mathcal{A}(\mathcal{P}_2)$$

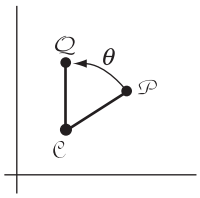


Figure 2.8 Rotation of a point $\mathcal{P}$ about a central point $\mathcal{C}$ to obtain a point $\mathcal{Q}$.

but Equation (2.36) says that as long as we use the same origin for the coordinate space, we may compute the transformation in vector terms:

$$\mathbf{Y} = M\mathbf{X} + \mathbf{B} \tag{2.37}$$

You should recognize this as the standard form in which you have manipulated vectors when translations are allowed. It is important to note that the right-hand side of Equation (2.37) consists only of vector and matrix operations. There are no "point" operations. Perhaps this is yet another argument why your graphics engine need not enforce a distinction between points and vectors.

Although translation is the obvious candidate to illustrate affine transformations, another one of interest is rotation about a point that is not at the origin, as Figure 2.8 illustrates.

The origin of the coordinate system is $\mathcal{O} = (0, 0, 0)$. The vector from the origin to the center of rotation is defined by $\mathbf{C} = \mathcal{C} - \mathcal{O}$. The rotation matrix is R. The points are $\mathcal{P} = \mathcal{O} + \mathbf{X}$ and

$$\begin{aligned}
\mathcal{Q} &= \mathcal{O} + \mathbf{Y} \\
&= \mathcal{C} + (\mathcal{O} - \mathcal{C}) + \mathbf{Y} \\
&= \mathcal{C} + (\mathbf{Y} - \mathbf{C}) \\
&= \mathcal{C} + R(\mathbf{X} - \mathbf{C}) \\
&= \mathcal{O} + (\mathcal{C} - \mathcal{O}) + R(\mathbf{X} - \mathbf{C}) \\
&= \mathcal{O} + \mathbf{C} + R(\mathbf{X} - \mathbf{C})
\end{aligned}$$

Removing the point notation, the vector calculations are

$$\mathbf{Y} = R\mathbf{X} + (I - R)\mathbf{C} \tag{2.38}$$

where I is the identity matrix. The translational component of this representation for the affine transformation is $(I - R)\mathbf{C}$.

Note: For the remainder of the book, I will use the same typesetting convention for points and vectors. When necessary, I will state explicitly whether something is a point or a vector.

2.2.3 Projective Transformations

Yet another class of transformations involve *projections*. There are different types of projections that will interest us: orthogonal, oblique, and perspective.

Orthogonal Projection onto a Line

Orthogonal projection is the simplest type of projection to analyze. Consider the goal of projecting a point onto a line. Figure 2.9 illustrates this.

The point $\mathbf{X}$ is to be projected onto a line containing a point $\mathbf{P}$ and having unit-length direction $\mathbf{D}$. In the figure, the point $\mathbf{Y}$ is the projection and has the property that the vector $\mathbf{X} - \mathbf{Y}$ is perpendicular to $\mathbf{D}$; that is,

$$0 = \mathbf{D} \cdot (\mathbf{X} - \mathbf{Y})$$

Because $\mathbf{Y}$ is on the line, it is of the form

$$\mathbf{Y} = \mathbf{P} + d\mathbf{D}$$

for some scalar d. Substituting this in the previous displayed equation, we have

$$0 = \mathbf{D} \cdot (\mathbf{X} - \mathbf{P} - d\mathbf{D})$$

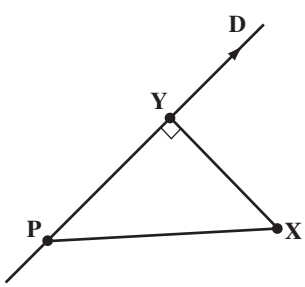


Figure 2.9 The orthogonal projection of a point onto a line.

which can be solved to obtain

$$d = \mathbf{D} \cdot (\mathbf{X} - \mathbf{P})$$

The point of projection is therefore defined by

$$\mathbf{Y} - \mathbf{P} = (\mathbf{D} \cdot (\mathbf{X} - \mathbf{P}))\mathbf{D} = \mathbf{D}\mathbf{D}^{\mathrm{T}}(\mathbf{X} - \mathbf{P})$$

Equivalently, the projection is

$$\mathbf{Y} = \mathbf{D}\mathbf{D}^{\mathrm{T}}\mathbf{X} + \left(I - \mathbf{D}\mathbf{D}^{\mathrm{T}}\right)\mathbf{P} \tag{2.39}$$

which is of the form $\mathbf{Y} = M\mathbf{X} + \mathbf{B}$, therefore orthogonal projection onto a line is an affine transformation. Unlike our previous examples, this transformation is not invertible. Each point on a line has infinitely many points that project to it (an entire plane's worth), so you cannot unproject a point from the line unless you have more information. Algebraically, the noninvertibility shows up in that $M = \mathbf{D}\mathbf{D}^{\mathrm{T}}$ is not an invertible matrix.

Orthogonal Projection onto a Plane

Consider projecting a point $\mathbf{X}$ onto a plane defined by $\mathbf{N} \cdot (\mathbf{Y} - \mathbf{P}) = 0$, where $\mathbf{N}$ is a unit-length normal vector, $\mathbf{P}$ is a specified point, and $\mathbf{Y}$ is any point on the plane. Figure 2.10 illustrates this.

The projection point is $\mathbf{Y}$. The vector $\mathbf{X} - \mathbf{P}$ has a component in the plane, $\mathbf{Y} - \mathbf{P}$, and a component on the normal line to the plane, $n\mathbf{N}$, for some scalar n. Thus,

$$\mathbf{X} - \mathbf{P} = (\mathbf{Y} - \mathbf{P}) + n\mathbf{N}$$

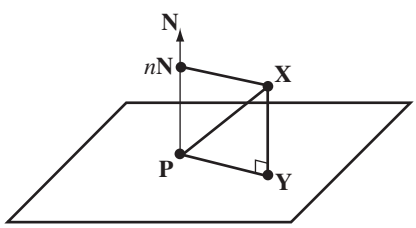


Figure 2.10 The orthogonal projection of a point onto a plane.

Dotting with $\mathbf{N}$, we have

$$\mathbf{N} \cdot (\mathbf{X} - \mathbf{P}) = n$$

which uses the facts that $\mathbf{N}$ is unit length and $\mathbf{Y} - \mathbf{P}$ is perpendicular to $\mathbf{N}$. We may solve to obtain

$$\mathbf{Y} - \mathbf{P} = (\mathbf{X} - \mathbf{P}) - (\mathbf{N} \cdot (\mathbf{X} - \mathbf{P}))\mathbf{N} = \left(I - \mathbf{N}\mathbf{N}^{\mathrm{T}}\right)(\mathbf{X} - \mathbf{P})$$

Equivalently, the projection is

$$\mathbf{Y} = \left(I - \mathbf{N}\mathbf{N}^{\mathrm{T}}\right)\mathbf{X} + \mathbf{N}\mathbf{N}^{\mathrm{T}}\mathbf{P} \tag{2.40}$$

You will notice that this is of the form $\mathbf{Y} = M\mathbf{X} + \mathbf{B}$, therefore orthogonal projection onto a plane is also an affine transformation. Moreover, it is not invertible since infinitely many points in space project to the same point on the plane (an entire line's worth), so you cannot unproject a point from the plane unless you have more information. Algebraically, the noninvertibility shows up in that $M = I - \mathbf{N}\mathbf{N}^{\mathrm{T}}$ is not an invertible matrix.

Oblique Projection onto a Plane

As before, let the plane contain a point $\mathbf{P}$ and have a unit-length normal vector $\mathbf{N}$. The projection of a point onto a plane does not have to be in the normal direction to the plane. This type of projection is said to be *oblique* to the plane. Let $\mathbf{D}$ be the unit-length direction in which to project the points. This direction should not be perpendicular to the plane; that is, $\mathbf{N} \cdot \mathbf{D} \neq 0$. Figure 2.11 illustrates this.

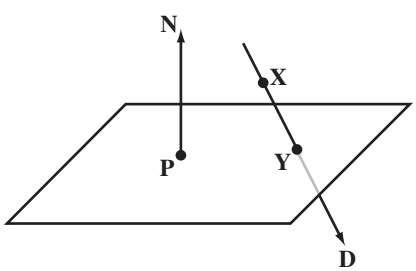


Figure 2.11 The oblique projection of a point onto a plane.

The projection point is $\mathbf{Y} = \mathbf{X} + d\mathbf{D}$ for some scalar d. Subtracting the known plane point, we have

$$\mathbf{Y} - \mathbf{P} = (\mathbf{X} - \mathbf{P}) + d\mathbf{D}$$

Dotting with $\mathbf{N}$, we have

$$0 = \mathbf{N} \cdot (\mathbf{X} - \mathbf{P}) + d\mathbf{N} \cdot \mathbf{D}$$

which uses the fact that $\mathbf{Y} - \mathbf{P}$ is perpendicular to $\mathbf{N}$. We may solve for d,

$$d = -\frac{\mathbf{N} \cdot (\mathbf{X} - \mathbf{P})}{\mathbf{N} \cdot \mathbf{D}}$$

The projection is defined, therefore, by

$$\mathbf{Y} - \mathbf{P} = (\mathbf{X} - \mathbf{P}) - \frac{\mathbf{N} \cdot (\mathbf{X} - \mathbf{P})}{\mathbf{N} \cdot \mathbf{D}} \mathbf{D} = \left(I - \frac{\mathbf{D}\mathbf{N}^{\mathrm{T}}}{\mathbf{D}^{\mathrm{T}}\mathbf{N}} \right) (\mathbf{X} - \mathbf{P})$$

Equivalently, the projection is

$$\mathbf{Y} = \left(I - \frac{\mathbf{D}\mathbf{N}^{\mathrm{T}}}{\mathbf{D}^{\mathrm{T}}\mathbf{N}} \right) \mathbf{X} + \frac{\mathbf{D}\mathbf{N}^{\mathrm{T}}}{\mathbf{D}^{\mathrm{T}}\mathbf{N}} \mathbf{P} \tag{2.41}$$

Once again, this is of the form $M\mathbf{X} + \mathbf{B}$, therefore oblique projection onto a plane is an affine transformation. And as with the other projections, it is not invertible.

Perspective Projection onto a Plane

We now encounter a projection that is not an affine transformation, and one that is central to rendering—the perspective projection. Points are now projected onto a plane, but along rays with a common origin $\mathbf{E}$, called the *eye point*. Figure 2.12 illustrates this.

The point $\mathbf{X}$ is projected to the point $\mathbf{Y}$. The ray to use has origin $\mathbf{E}$, but the direction is determined by the vector $\mathbf{X} - \mathbf{E}$. At the moment there is no need to use a unit-length vector for the direction. As a ray point, we have

$$\mathbf{Y} = \mathbf{E} + t(\mathbf{X} - \mathbf{E})$$

for some scalar $t > 0$. Subtract the known plane point $\mathbf{P}$ to obtain

$$\mathbf{Y} - \mathbf{P} = (\mathbf{E} - \mathbf{P}) + t(\mathbf{X} - \mathbf{E})$$

and dot with $\mathbf{N}$ to obtain

$$0 = \mathbf{N} \cdot (\mathbf{E} - \mathbf{P}) + t\mathbf{N} \cdot (\mathbf{X} - \mathbf{E})$$

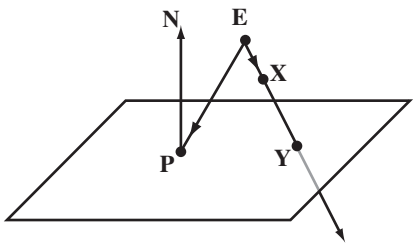


Figure 2.12 The perspective projection of a point onto a plane using an eye point **E**.

This uses the fact that $\mathbf{Y} - \mathbf{P}$ is perpendicular to the plane normal **N**. Solving for t, we have

$$t = -\frac{\mathbf{N} \cdot (\mathbf{E} - \mathbf{P})}{\mathbf{N} \cdot (\mathbf{X} - \mathbf{E})}$$

To reaffirm the constraint that $t \geq 0$, Figure 2.12 shows the vectors $\mathbf{P} - \mathbf{E}$ and $\mathbf{X} - \mathbf{E}$. Both vectors form an obtuse angle with the normal vector, so $\mathbf{N} \cdot (\mathbf{P} - \mathbf{E}) < 0$ and $\mathbf{N} \cdot (\mathbf{X} - \mathbf{E}) < 0$, which imply $t > 0$.

The projection point is, therefore, defined by

$$\mathbf{Y} = \mathbf{E} - \frac{\mathbf{N} \cdot (\mathbf{E} - \mathbf{P})}{\mathbf{N} \cdot (\mathbf{X} - \mathbf{E})} (\mathbf{X} - \mathbf{E}) = \frac{\left(\mathbf{E}\mathbf{N}^{\mathrm{T}} - \mathbf{N} \cdot (\mathbf{E} - \mathbf{P})I\right) (\mathbf{X} - \mathbf{E})}{\mathbf{N} \cdot (\mathbf{X} - \mathbf{E})} \tag{2.42}$$

where I is the identity matrix. It is not possible to write the expression in the form $\mathbf{Y} = M\mathbf{X} + \mathbf{B}$, where M and **B** are independent of **X**. What prevents this is the division by $\mathbf{N} \cdot (\mathbf{X} - \mathbf{E})$. You will hear reference to this division as the *perspective divide*. However, Section 2.2.5 will show a unifying format for representing linear, affine, and perspective transformations.

Exercise 2.6 The point **X** is "behind the eye" when the distance from **X** to the plane is larger than or equal to the distance from **E** to the plane. In this case, prove that the ray from **E** to **X** cannot intersect the plane. ■

2.2.4 Properties of Perspective Projection

Consider the coordinate system whose origin is the eye point, whose view direction is **D**, whose up direction is **U**, and whose right direction is $\mathbf{R} = \mathbf{D} \times \mathbf{U}$. The point to be projected has a representation in this coordinate system as

$$\mathbf{X} = \mathbf{E} + d\mathbf{D} + u\mathbf{U} + r\mathbf{R}$$

where $d = \mathbf{D} \cdot (\mathbf{X} - \mathbf{E})$, $u = \mathbf{D} \cdot (\mathbf{X} - \mathbf{E})$, and $r = \mathbf{D} \cdot (\mathbf{X} - \mathbf{E})$. The plane normal is in the opposite direction of the view; namely, $\mathbf{N} = -\mathbf{D}$. The point on the plane closest to the eye point is $\mathbf{P} = \mathbf{E} + d_{\min}\mathbf{D}$ for some distance $d_{\min} > 0$. Substituting all this into Equation (2.42), we have the projected point

$$\mathbf{Y} = \mathbf{E} + \frac{d_{\min}(d\mathbf{D} + u\mathbf{U} + r\mathbf{R})}{d} = \mathbf{E} + \frac{d\mathbf{D} + u\mathbf{U} + r\mathbf{R}}{d/d_{\min}}$$

Thus, in the new coordinate system, (d, u, r) is projected to $(d, u, r)/(d/d_{\min})$. Notice that the first component is actually $d_{\min}$, which is to be expected since the projection point is on the plane $d = d_{\min}$ (in the new coordinate system).

Within the new coordinate system, it is relatively easy to demonstrate some properties of perspective projection. In all of the cases mentioned here, the objects are assumed to be in front of the eye point; that is, all points on the objects are closer to the projection plane than the eye point. Line segments must project to line segments, or to a single point when the line segment is fully contained by a single ray emanating from **E**. Triangles must project to triangles, or to a line segment when the triangle and **E** are coplanar. Finally, conic sections project to conic sections, with the degenerate case of a conic section projecting to linear components when the conic section is coplanar with **E**. A consequence of verifying these properties is that we will have an idea of how uniformly spaced points on a line segment are projected to nonuniformly spaced points. This relationship is important in perspective projection for rendering, in particular when depth must be computed (which is nearly always) and in interpolating vertex attributes.

Lines Project to Lines

Consider a line segment with endpoints $\mathbf{Q}_i = (d_i, u_i, r_i)$ for $i = 0, 1$. Using only the two components relevant to the projection plane, let the corresponding projected points be $\mathbf{P}_i = (u_i/w_i, r_i/w_i)$, with $w_i = d_i/d_{\min}$ for $i = 0, 1$. The 3D line segment is $\mathbf{Q}(s) = \mathbf{Q}_0 + s(\mathbf{Q}_1 - \mathbf{Q}_0)$ for $s \in [0, 1]$. For each s, let $\mathbf{P}(s)$ be the projection of $\mathbf{Q}(s)$. Thus,

$$\mathbf{Q}(s) = (d_0 + s(d_1 - d_0), u_0 + s(u_1 - u_0), r_0 + s(r_1 - r_0))$$

and

$$\begin{aligned}\mathbf{P}(s) &= \left(\frac{u_0 + s(u_1 - u_0)}{w_0 + s(w_1 - w_0)}, \frac{r_0 + s(r_1 - r_0)}{w_0 + s(w_1 - w_0)}\right) \\ &= \left(\frac{u_0}{w_0} + \frac{w_1 s}{w_0 + (w_1 - w_0)s}\left(\frac{u_1}{w_1} - \frac{u_0}{w_0}\right),\right. \\ &\qquad \left.\frac{r_0}{w_0} + \frac{w_1 s}{w_0 + (w_1 - w_0)s}\left(\frac{r_1}{w_1} - \frac{r_0}{w_0}\right)\right)\end{aligned}$$

$$= \mathbf{P}_0 + \frac{w_1 s}{w_0 + (w_1 - w_0)s}(\mathbf{P}_1 - \mathbf{P}_0)$$
$$= \mathbf{P}_0 + \bar{s}(\mathbf{P}_1 - \mathbf{P}_0)$$

where the last equality defines

$$\bar{s} = \frac{w_1 s}{w_0 + (w_1 - w_0)s} \tag{2.43}$$

a quantity that is also in the interval [0, 1]. We have obtained a parametric equation for a 2D line segment with endpoints $\mathbf{P}_0$ and $\mathbf{P}_1$, so in fact line segments are projected to line segments, or if $\mathbf{P}_0 = \mathbf{P}_1$, the projected segment is a single point. The inverse mapping from $\bar{s}$ to s is actually important for perspectively correct rasterization, as we will see later:

$$s = \frac{w_0 \bar{s}}{w_1 + (w_0 - w_1)\bar{s}} \tag{2.44}$$

EXERCISE 2.7 Construct Equation (2.44) from Equation (2.43). ■

Equation (2.43) has more to say about perspective projection. Assuming $w_1 > w_0$, a uniform change in s does not result in a uniform change in $\bar{s}$. The graph of $\bar{s} = F(s)$ is shown in Figure 2.13.

The first derivative is $F'(s) = w_0 w_1/[w_0 + s(w_1 - w_0)]^2 > 0$, and the second derivative is $F''(s) = -2w_0 w_1/[w_0 + s(w_1 - w_0)]^3 < 0$. The slopes of the graph at the endpoints are $F'(0) = w_1/w_0 > 1$ and $F'(1) = w_0/w_1 < 1$. Since the second derivative is always negative, the graph is concave. An intuitive interpretation is to select a set of uniformly spaced points on the 3D line segment. The projections of these

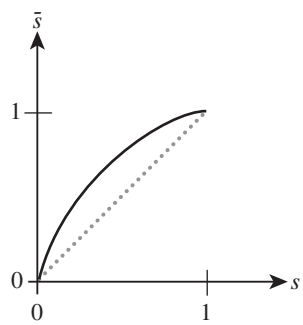


Figure 2.13 The relationship between s and $\bar{s}$.

points are not uniformly spaced. More specifically, the spacing between the projected points decreases as $\bar{s}$ increases from 0 to 1. The relationship between s and $\bar{s}$ and limited floating-point precision are what contribute to depth buffering artifacts, to be discussed later.

Triangles Project to Triangles

Because line segments project to line segments, we can immediately assert that triangles project to triangles, although possibly degenerating to a line segment. However, let's derive the parametric relationships that are analogous to those of Equations (2.43) and (2.44) anyway.

Let $\mathbf{Q}_i = (d_i, u_i, r_i)$ for $i = 0, 1, 2$ be the vertices of a triangle. The triangle is specified parametrically as $\mathbf{Q}(s, t) = \mathbf{Q}_0 + s(\mathbf{Q}_1 - \mathbf{Q}_0) + t(\mathbf{Q}_2 - \mathbf{Q}_0)$ for $0 \le s \le 1$, $0 \le t \le 1$, and $s + t \le 1$. Let the projected points for the $\mathbf{Q}_i$ be $\mathbf{P}_i = (u_i/w_i, r_i/w_i)$ for $i = 0, 1, 2$, where $w_i = d_i/d_{\min}$. For each s and t, let $\mathbf{P}(s, t)$ be the projection of $\mathbf{Q}(s, t)$. Some algebra will show the following, where $\Delta = w_0 + (w_1 - w_0)s + (w_2 - w_0)t$,

$$\begin{aligned}
\mathbf{P}(s,t) &= \left(\frac{u_0 + s(u_1 - u_0) + t(u_2 - u_0)}{\Delta}, \frac{r_0 + s(r_1 - r_0) + t(r_2 - r_0)}{\Delta} \right) \\
&= \left(\frac{u_0}{w_0} + \frac{w_1 s}{\Delta}\left(\frac{u_1}{w_1} - \frac{u_0}{w_0}\right) + \frac{w_2 t}{\Delta}\left(\frac{u_2}{w_2} - \frac{u_0}{w_0}\right), \frac{y_0}{w_0} + \frac{w_1 s}{\Delta}\left(\frac{r_1}{w_1} - \frac{r_0}{w_0}\right) \right. \\
&\quad \left. + \frac{w_2 t}{\Delta}\left(\frac{r_2}{w_2} - \frac{r_0}{w_0}\right) \right) \\
&= \mathbf{P}_0 + \frac{w_1 s}{\Delta}(\mathbf{P}_1 - \mathbf{P}_0) + \frac{w_2 t}{\Delta}(\mathbf{P}_2 - \mathbf{P}_0)
\end{aligned}$$

Define

$$(\bar{s}, \bar{t}) = \frac{(w_1 s, w_2 t)}{w_0 + (w_1 - w_0)s + (w_2 - w_0)t} \tag{2.45}$$

in which case the projected triangle is

$$\mathbf{P}(s, t) = \mathbf{P}_0 + \bar{s}(\mathbf{P}_1 - \mathbf{P}_0) + \bar{t}(\mathbf{P}_2 - \mathbf{P}_0)$$

The inverse mapping can be used by the rasterizers for perspectively correct triangle rasterization. The inverse is

$$(s, t) = \frac{(w_0 w_2 \bar{s}, w_0 w_1 \bar{t})}{w_1 w_2 + w_2(w_0 - w_1)\bar{s} + w_1(w_0 - w_2)\bar{t}} \tag{2.46}$$

EXERCISE 2.8 Construct Equation (2.46) from Equation (2.45). ■

Conics Project to Conics

Showing that the projection of a conic section is itself a conic section requires a bit more algebra. Let $\mathbf{Q}_i = (x_i, y_i, z_i)$ for $i = 0, 1, 2$ be points such that $\mathbf{Q}_1 - \mathbf{Q}_0$ and $\mathbf{Q}_2 - \mathbf{Q}_0$ are unit length and orthogonal. The points in the plane containing the $\mathbf{Q}_i$ are represented by $\mathbf{Q}(s, t) = \mathbf{Q}_0 + s(\mathbf{Q}_1 - \mathbf{Q}_0) + t(\mathbf{Q}_2 - \mathbf{Q}_0)$ for any real numbers s and t. Within that plane, a conic section is defined by

$$As^2 + Bst + Ct^2 + Ds + Et + F = 0 \tag{2.47}$$

To show that the projection is also a conic, substitute the formulas in Equation (2.46) into Equation (2.47) to obtain

$$\bar{A}\bar{s}^2 + \bar{B}\bar{s}\bar{t} + \bar{C}\bar{t}^2 + \bar{D}\bar{s} + \bar{E}\bar{t} + \bar{F} = 0 \tag{2.48}$$

where

$$\bar{A} = w_2^2\left(w_0^2 A + w_0(w_0 - w_1)D + (w_0 - w_1)^2 F\right)$$

$$\bar{B} = w_1 w_2\left(w_0^2 B + w_0(w_0 - w_2)D + w_0(w_0 - w_1)E + 2(w_0 - w_1)(w_0 - w_2)F\right)$$

$$\bar{C} = w_1^2\left(w_0^2 C + w_0(w_0 - w_2)E + (w_0 - w_2)^2 F\right)$$

$$\bar{D} = w_1 w_2^2\left(w_0 D + 2(w_0 - w_1)F\right)$$

$$\bar{E} = w_1^2 w_2\left(w_0 E + 2(w_0 - w_2)F\right)$$

$$\bar{F} = w_1^2 w_2^2 F.$$

A special case is $D = E = F = 0$, in which case the conic is centered at $\mathbf{Q}_0$ and has axes $\mathbf{Q}_1 - \mathbf{Q}_0$ and $\mathbf{Q}_2 - \mathbf{Q}_0$. Consequently, $\bar{A} = w_2^2 w_0^2 A$, $\bar{B} = w_1 w_2 w_0^2 B$, $\bar{C} = w_1^2 w_0^2 C$, and $\bar{B}^2 - 4\bar{A}\bar{C} = B^2 - 4AC$. The sign of $B^2 - 4AC$ is preserved, so ellipses are mapped to ellipses, hyperbolas are mapped to hyperbolas, and parabolas are mapped to parabolas.

EXERCISE 2.9 At the beginning of the discussion about projecting line segments, triangles, and conic sections, the assumption was made that all points on these objects are in front of the eye point. What can you say about the projections of the objects when some points are behind the eye point? ■

2.2.5 Homogeneous Points and Matrices

We have seen that linear transformations are of the form

$$\mathbf{Y} = M\mathbf{X}$$

where $\mathbf{X}$ is the 3×1 input vector, $\mathbf{Y}$ is the 3×1 output vector, and M is a 3×3 matrix of constants. Affine transformations extend this to

$$\mathbf{Y} = M\mathbf{X} + \mathbf{B}$$

where $\mathbf{B}$ is a 3×1 vector of constants. The perspective transformation did not fit within this framework. Equation (2.42) is of the form

$$\mathbf{Y} = \frac{M(\mathbf{X} - \mathbf{E})}{\mathbf{N} \cdot (\mathbf{X} - \mathbf{E})}$$

We can unify these into a single matrix representation by introducing the concept of *homogeneous points*, which are represented as 4-tuples but written as 4×1 vectors when used in matrix-vector operations, and *homogeneous matrices*, which are represented as 4×4 matrices.

Using the standard naming conventions that graphics practitioners have used for homogeneous points, the 4-tuples are of the form (x, y, z, w). I had already hinted at using 4-tuples to distinguish between points and vectors; see Section 2.1.3. The 4-tuple represents a point when $w = 1$, so $(x, y, z, 1)$ is a point. The 4-tuple represents a vector when $w = 0$, so $(x, y, z, 0)$ is a vector. In computer graphics, though, there is more to homogeneous points than specifying w to be zero or one.

Defining points as 4-tuples already allows us to unify linear and affine transformations into a single matrix representation. Specifically, the linear transformation is currently of the form

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \mathbf{Y} = M\mathbf{X} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

Appending a fourth component of 1 to the vectors and increasing the size of the matrix, adding certain entries as needed, this equation becomes

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{bmatrix}$$

A convenient *block-matrix form* is

$$\left[\begin{array}{c} \mathbf{Y} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} M & \mathbf{0} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{X} \\ \hline 1 \end{array}\right]$$

The output vector has an upper block that is the 3×1 vector $\mathbf{Y}$. The lower block is the (1×1) scalar 1. The input vector is structured similarly. The matrix of coefficients has the following structure. The upper-left block is the 3×3 matrix M; the upper-right block is the 3×1 zero vector; the lower-left block is the 1×3 zero vector; and the lower-right block is the (1×1) scalar 1.

Similarly, the affine transformation currently is

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \mathbf{Y} = M\mathbf{X} + \mathbf{B} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

but may be extended to use 4×1 points and a 4×4 matrix,

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & b_0 \\ m_{10} & m_{11} & m_{12} & b_1 \\ m_{20} & m_{21} & m_{22} & b_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{bmatrix}$$

It also has a block-matrix form,

$$\left[\begin{array}{c} \mathbf{Y} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} M & \mathbf{B} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{X} \\ \hline 1 \end{array}\right]$$

The perspective transformation can *almost* be made to fit into this framework, with two notable exceptions. First, the choice of $w = 1$ for the input is acceptable, but the output value for w is generally nonzero and not 1. Second, we still cannot handle the perspective divide. Despite this, consider the following block-matrix expression that gets us closer to our goal:

$$\left[\begin{array}{c} \mathbf{Y} \\ \hline 1 \end{array}\right] \sim \left[\begin{array}{c} \mathbf{Y}' \\ \hline w \end{array}\right] = \left[\begin{array}{c} M(\mathbf{X} - \mathbf{E}) \\ \hline \mathbf{N}^{\mathrm{T}}(\mathbf{X} - \mathbf{E}) \end{array}\right] = \left[\begin{array}{c|c} M & -M\mathbf{E} \\ \hline \mathbf{N}^{\mathrm{T}} & -\mathbf{N}^{\mathrm{T}}\mathbf{E} \end{array}\right] \left[\begin{array}{c} \mathbf{X} \\ \hline 1 \end{array}\right] \tag{2.49}$$

where $M = \mathbf{E}\mathbf{N}^{\mathrm{T}} - \mathbf{N} \cdot (\mathbf{E} - \mathbf{P})I$. The leftmost expression is what we want to construct, where $\mathbf{Y}$ is the output vector defined by Equation (2.42). The remaining portions of the expression are what we can construct using matrix operations. The output of these operations has a 3×1 vector $\mathbf{Y}'$, which is not the actual output we want. The output also has a w component, which is not necessarily 1. However, if we were to perform the perspective divide, we obtain

$$\mathbf{Y} = \mathbf{Y}'/w$$

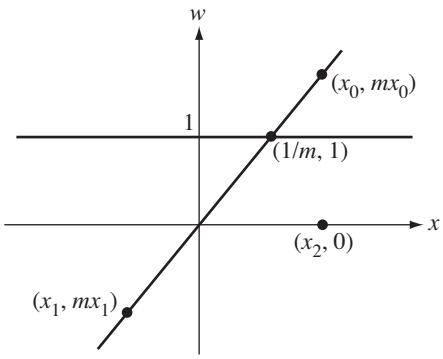


Figure 2.14 All homogeneous points along a line of slope m, excluding the origin, are equivalent to the homogeneous point $(1/m, 1)$. The points with a w-component of zero are vectors and are said to be equivalent to the *point at infinity*.

The division is not a matrix operation, because it involves a quantity dependent on the input $\mathbf{X}$. The use of the symbol $\sim$ in

$$\begin{bmatrix} \mathbf{Y} \\ \hline 1 \end{bmatrix} \sim \begin{bmatrix} \mathbf{Y}' \\ \hline w \end{bmatrix}$$

indicates that the 4-tuples are not equal but *equivalent* in the sense that the division by w does produce two equal 4-tuples. This equivalence is the basis for *projective geometry*.

To understand the equivalence in two dimensions, look at Figure 2.14. The figure shows a couple of homogeneous points, (x_0, mx_0) and (x_1, mx_1), on the line of slope m. All homogeneous points on this line, excluding the origin, are equivalent to $(1/m, 1)$. The homogenenous point $(x_2, 0)$ corresponds to a vector since its w-component is zero. The division by zero cannot be performed, but the point is said to be equivalent to the *point at infinity*.

Now that we have the concept of equivalence of homogeneous points, notice that the matrix operations in the linear and affine transformations produce outputs whose w-component is 1. If we were to divide by w anyway, we would obtain the correct results for the transformations. This allows us finally to have a unifying representation for linear, affine, and perspective transformations—as homogeneous matrix operations. The most general form allows for inputs to have w-components that are not 1:

$$\begin{bmatrix} \mathbf{Y}' \\ \hline w_1 \end{bmatrix} = \begin{bmatrix} M\mathbf{X}' + w_0\mathbf{B} \\ \hline \mathbf{C}^{\mathrm{T}}\mathbf{X}' + dw_0 \end{bmatrix} = \left[\begin{array}{c|c} M & \mathbf{B} \\ \hline \mathbf{C}^{\mathrm{T}} & d \end{array}\right] \begin{bmatrix} \mathbf{X}' \\ \hline w_0 \end{bmatrix} \tag{2.50}$$

When necessary, the equivalent point is computed by doing the perspective division. Linear transformations are characterized by $\mathbf{B} = \mathbf{0}$, $\mathbf{C} = \mathbf{0}$, $d = 1$, $w_0 = 1$, and $w_1 = 0$. Affine transformations are characterized by $\mathbf{C} = \mathbf{0}$, $d = 1$, $w_0 = 1$, and $w_1 = 0$. Perspective transformations occur when $\mathbf{C} \neq \mathbf{0}$ and, as long as $w_1 \neq 0$, you obtain the actual 3D projection point by doing the perspective divide.

Homogeneous transformations, specifically projective ones, are a major part of culling and clipping of triangles against the planes defining a view frustum; see Sections 2.3.5 and 2.4.3. They also occur in special effects such as planar projected shadows (Section 20.11), planar reflections (Section 20.10), projected textures (Section 20.12), and shadow maps (Section 20.13).

2.3 Cameras

Only a portion of the world is displayed at any one time. This region is called the *view volume*. Objects outside the view volume are not visible and therefore not drawn. The process of determining which objects are not visible is called *culling*. Objects that intersect the boundaries of the view volume are only partially visible. The visible portion of an object is determined by intersecting it with the view volume, a process called *clipping*.

The display of visible data is accomplished by projecting it onto a *view plane*. In this book I consider only perspective projection, as discussed in Sections 2.2.3 through 2.2.5. Orthogonal projection may also be used for viewing. In a graphics API, this amounts to choosing the parameters for a projection matrix.

2.3.1 The Perspective Camera Model

Our assumption is that the view volume is a bounded region in space, so the projected data lies in a bounded region in the view plane. A rectangular region in the view plane that contains the projected data is called a *viewport*. The viewport is what is drawn on the rectangular computer screen. The standard view volume used is called the *view frustum*. It is constructed by selecting an eye point and forming an infinite pyramid with four planar sides. Each plane contains the eye point and an edge of the viewport. The infinite pyramid is truncated by two additional planes called the *near plane* and the *far plane*. Figure 2.15 shows a view frustum.

The perspective projection is computed by intersecting a ray with the view plane. The ray has origin $\mathbf{E}$, the eye point, and passes through the world point $\mathbf{X}$. The intersection point is $\mathbf{Y}$. Equation (2.42) tells you how to construct $\mathbf{Y}$ from $\mathbf{X}$ as long as you know the eye point and the equation of the view plane, which I mention in the next paragraph. The combination of an eye point, a set of coordinate axes assigned to the eye point, a view plane, a viewport, and a view frustum is called a *camera model*.

The camera has a coordinate system associated with it. The *camera origin* is the eye point $\mathbf{E}$. The *camera view direction* is a unit-length vector $\mathbf{D}$ that is perpendicular

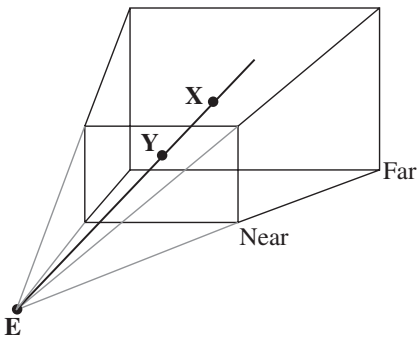


Figure 2.15 An eye point **E** and a view frustum. The point **X** in the view frustum is projected to the point **Y** in the viewport.

to the view plane. This direction vector is chosen to point away from the observer, so the eye point is considered to be on the negative side of the plane. If the view plane is at a distance $d_{\min}$ from the eye point, measured in the **D** direction, then the view plane normal to use in Equation (2.42) is $\mathbf{N} = -\mathbf{D}$ and the view plane point to use is $\mathbf{P} = \mathbf{E} + d_{\min}\mathbf{D}$. The *camera up vector* is the unit-length **U** vector chosen to be parallel to two opposing edges of the viewport. The *camera right vector* is the unit-length vector **R** chosen to be perpendicular to the camera direction and camera up vector with $\mathbf{R} = \mathbf{D} \times \mathbf{U}$. The coordinate system $\{\mathbf{E}; \mathbf{D}, \mathbf{U}, \mathbf{R}\}$ is a right-handed system.

Figure 2.16 shows the camera model, including the camera coordinate system and the view frustum. The six frustum planes are labeled with their names: near, far, left, right, bottom, top. The camera location **E** and the camera axis directions **D**, **U**, and **R** are shown. The view frustum has eight vertices. The near-plane vertices are $\mathbf{V}_{t\ell}$, $\mathbf{V}_{b\ell}$, $\mathbf{V}_{tr}$, and $\mathbf{V}_{br}$. Each subscript consists of two letters, the first letters of the frustum planes that share that vertex. The far-plane vertices have the name **W** and use the same subscript convention. The equations for the vertices are

$$\begin{aligned}
\mathbf{V}_{b\ell} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\min}\mathbf{R} \\
\mathbf{V}_{t\ell} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\min}\mathbf{R} \\
\mathbf{V}_{br} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\max}\mathbf{R} \\
\mathbf{V}_{tr} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\max}\mathbf{R}
\end{aligned} \tag{2.51}$$

$$\mathbf{W}_{b\ell} = \mathbf{E} + \frac{d_{\max}}{d_{\min}}\left(d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\min}\mathbf{R}\right)$$

$$\mathbf{W}_{t\ell} = \mathbf{E} + \frac{d_{\max}}{d_{\min}}\left(d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\min}\mathbf{R}\right)$$

$$\mathbf{W}_{br} = \mathbf{E} + \frac{d_{\max}}{d_{\min}}\left(d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\max}\mathbf{R}\right)$$

$$\mathbf{W}_{tr} = \mathbf{E} + \frac{d_{\max}}{d_{\min}}\left(d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\max}\mathbf{R}\right)$$

The near plane is at a distance $d_{\min}$ from the camera location and the far plane is at a distance $d_{\max}$. These distances are the extreme values in the $\mathbf{D}$ direction. The extreme values in the $\mathbf{U}$ direction are $u_{\min}$ and $u_{\max}$. The extreme values in the $\mathbf{R}$ direction are $r_{\min}$ and $r_{\max}$.

The equations of the six view frustum planes are provided here in the form that is used for object culling. The near plane has inner-pointing, unit-length normal $\mathbf{D}$. A point on the plane is $\mathbf{E} + d_{\min}\mathbf{D}$. An equation of the near plane is

$$\mathbf{D}\cdot\mathbf{X} = \mathbf{D}\cdot(\mathbf{E} + d_{\min}\mathbf{D}) = \mathbf{D}\cdot\mathbf{E} + d_{\min} \tag{2.52}$$

The far plane has inner-pointing, unit-length normal $-\mathbf{D}$. A point on the plane is $\mathbf{E} + d_{\max}\mathbf{D}$. An equation of the far plane is

$$-\mathbf{D}\cdot\mathbf{X} = -\mathbf{D}\cdot(\mathbf{E} + d_{\max}\mathbf{D}) = -(\mathbf{D}\cdot\mathbf{E} + d_{\max}) \tag{2.53}$$

The left plane contains the three points $\mathbf{E}$, $\mathbf{V}_{t\ell}$, and $\mathbf{V}_{b\ell}$. A normal vector that points inside the frustum is

$$\begin{aligned}(\mathbf{V}_{b\ell} - \mathbf{E}) \times (\mathbf{V}_{t\ell} - \mathbf{E}) &= (d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\min}\mathbf{R}) \times (d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\min}\mathbf{R}) \\ &= (d_{\min}\mathbf{D} + r_{\min}\mathbf{R}) \times (u_{\max}\mathbf{U}) + (u_{\min}\mathbf{U}) \times (d_{\min}\mathbf{D} + r_{\min}\mathbf{R}) \\ &= (d_{\min}\mathbf{D} + r_{\min}\mathbf{R}) \times ((u_{\max} - u_{\min})\mathbf{U}) \\ &= (u_{\max} - u_{\min})(d_{\min}\mathbf{D} \times \mathbf{U} + r_{\min}\mathbf{R} \times \mathbf{U}) \\ &= (u_{\max} - u_{\min})(d_{\min}\mathbf{R} - r_{\min}\mathbf{D})\end{aligned}$$

An inner-pointing, unit-length normal and the left plane are

$$\mathbf{N}_{\ell} = \frac{d_{\min}\mathbf{R} - r_{\min}\mathbf{D}}{\sqrt{d_{\min}^2 + r_{\min}^2}}, \qquad \mathbf{N}_{\ell}\cdot(\mathbf{X} - \mathbf{E}) = 0 \tag{2.54}$$

An inner-pointing normal to the right plane is $(\mathbf{V}_{tr} - \mathbf{E}) \times (\mathbf{V}_{br} - \mathbf{E})$. A similar set of calculations as before will lead to an inner-pointing, unit-length normal and

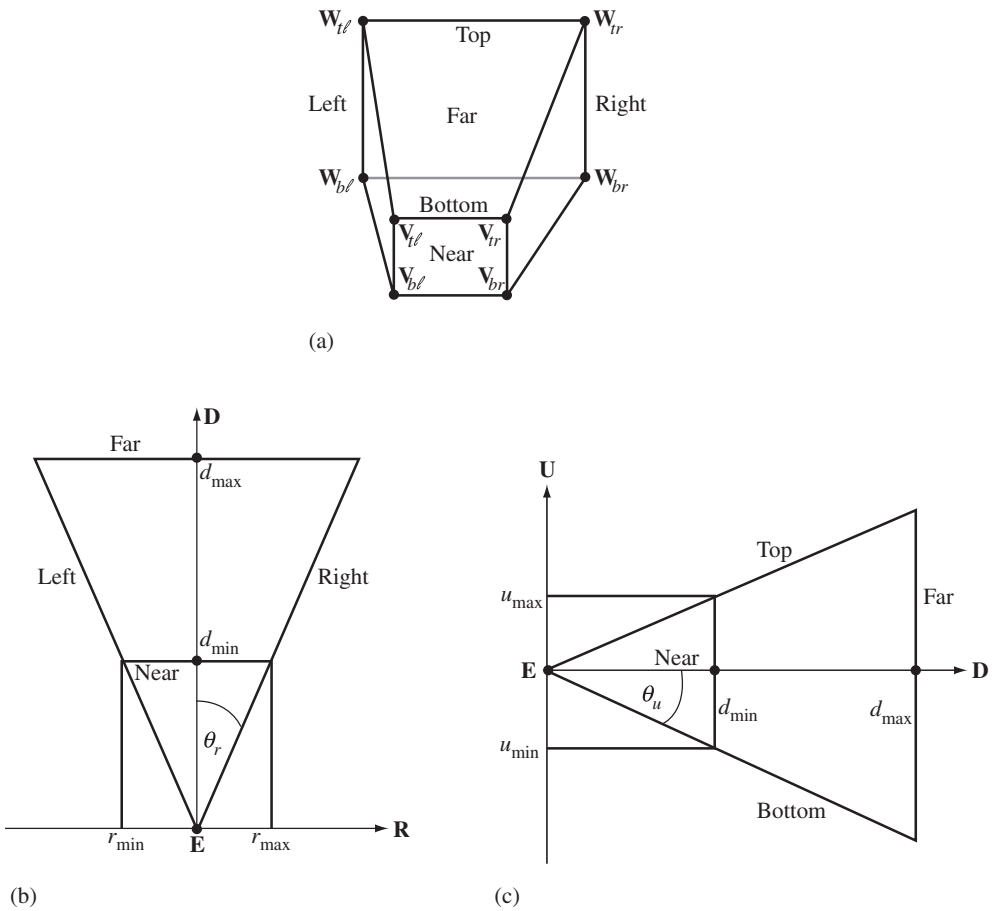


Figure 2.16 (a) A 3D drawing of the view frustum. The left, right, bottom, top, near, and far planes are labeled, as are the eight vertices of the frustum. (b) A 2D drawing of the view frustum as seen from the top side. (c) A 2D drawing of the view frustum as seen from the right side.

the right plane:

$$\mathbf{N}_r = \frac{-d_{\min}\mathbf{R} + r_{\max}\mathbf{D}}{\sqrt{d_{\min}^2 + r_{\max}^2}}, \qquad \mathbf{N}_r \cdot (\mathbf{X} - \mathbf{E}) = 0 \tag{2.55}$$

Similarly, an inner-pointing, unit-length normal and the bottom plane are

$$\mathbf{N}_b = \frac{d_{\min}\mathbf{U} - u_{\min}\mathbf{D}}{\sqrt{d_{\min}^2 + u_{\min}^2}}, \qquad \mathbf{N}_b \cdot (\mathbf{X} - \mathbf{E}) = 0 \tag{2.56}$$

An inner-pointing, unit-length normal and the top plane are

$$\mathbf{N}_t = \frac{-d_{\min}\mathbf{U} + u_{\max}\mathbf{D}}{\sqrt{d_{\min}^2 + u_{\max}^2}}, \qquad \mathbf{N}_t \cdot (\mathbf{X} - \mathbf{E}) = 0 \tag{2.57}$$

It is common when choosing a camera model to have an *orthogonal view frustum*. The frustum is symmetric in that $u_{\min} = -u_{\max}$ and $r_{\min} = -r_{\max}$. The four independent frustum parameters are $d_{\min}$, $d_{\max}$, $u_{\max}$, and $r_{\max}$. An alternate way to specify the frustum is to use the *field of view* in the **U** direction and the *aspect ratio* for the viewport. In Figure 2.16 (c), the field of view is the angle $2\theta_u$. The aspect ratio is the width divided by height, in this case $\rho = r_{\max}/u_{\max}$. The frustum is completely determined by specifying $d_{\min}$, $d_{\max}$, θ_u, and ρ. The values for $u_{\max}$ and $r_{\max}$ are determined from

$$u_{\max} = d_{\min} \tan(\theta_u), \qquad r_{\max} = \rho u_{\max} \tag{2.58}$$

The term *orthogonal* is used in this context to refer to the fact that the central axis of the frustum is orthogonal to the near face of the frustum. It does *not* refer to an orthographic projection.

Although every indication so far is that the projections of the points will be to the entire rectangular viewport of the view frustum, there are circumstances when you want to view a scene only in a subrectangle of the viewport. Using relative measurements, the full viewport is thought of as a unit square, as shown in Figure 2.17.

The full viewport has relative coordinates between 0 and 1. A smaller viewport is specified by choosing p_ℓ, p_r, p_b, and p_t so that $0 \le p_\ell < p_r \le 1$ and $0 \le p_b < p_t \le 1$. These relative coordinates will come into play when computing the actual pixel locations to draw in a window. The range of d values is $[d_{\min}, d_{\max}]$. A relative *depth range* is $[0, 1]$. The value 0 corresponds to $d_{\min}$ and the value 1 corresponds to $d_{\max}$. Some applications might want the depth range to be a subset $[p_n, p_f] \subseteq [0, 1]$.

In summary, you specify a perspective camera model by choosing an eye point **E**; a right-handed orthonormal set of coordinate axis directions **D** (view direction), **U** (up direction), and **R** (right direction); the view frustum values $d_{\min}$ (near-plane distance from the eye point), $d_{\max}$ (far-plane distance from the eye point), $r_{\min}$ (minimum in right direction), $r_{\max}$ (maximum in right direction), $u_{\min}$ (minimum in up direction), and $u_{\max}$ (maximum in up direction); the viewport values p_ℓ (left), p_r (right), p_b (bottom), and $p_t(top)$; and the depth range p_n (near) and p_f (far).

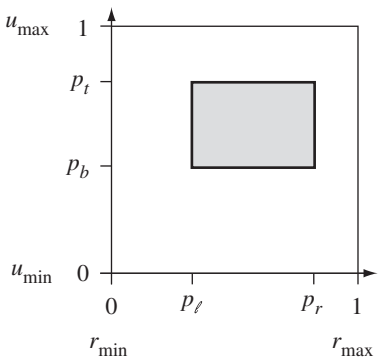


Figure 2.17 The full viewport of the view frustum is the full rectangle on the view plane. A smaller viewport is shown.

2.3.2 Model or Object Space

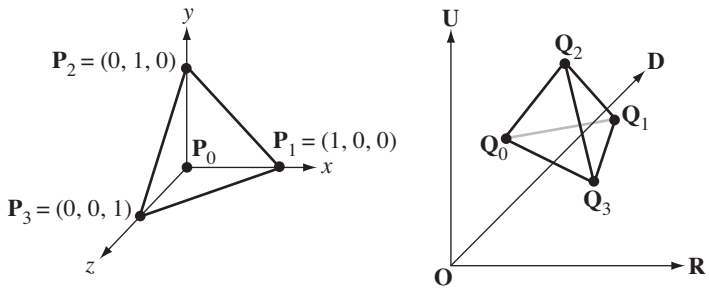
Three-dimensional modeling packages have their own specified coordinate systems for building polygonal models. I call the space in which the models are built *model space*. Others sometimes call this *object space*. I suppose if you are used to the art content being called models, you use model space, and if you are used to the content being called objects, you use object space.

2.3.3 World Space

The coordinate system that is most prominent in a game is the *world coordinate system*, or *world space*. The choice is not important from a theoretical standpoint. From a practical standpoint, the choice might be related to constraints you place on the artists regarding the coordinate systems they use in their modeling packages. For example, if a modeling package has the convention that the positive y-axis is in the upward direction, then you might very well choose the world coordinates to use the positive y-axis for the upward direction. Most likely if you chose a world coordinate system for your previous project, you will choose the same one for the next project.

The main problem in dealing with both a world space and a model space is positioning, orienting, and possibly scaling the models so that they are correctly placed in the world. For example, Figure 2.18 (a) shows a tetrahedron built in a coordinate system provided by a modeling package.

The tetrahedron vertices in model space are $\mathbf{P}_0 = (0, 0, 0)$, $\mathbf{P}_1 = (1, 0, 0)$, $\mathbf{P}_2 = (0, 1, 0)$, and $\mathbf{P}_3 = (0, 0, 1)$. We want each tetrahedron vertex $\mathbf{P}_i$ to be located in world



(a) Object in model space (b) Object in world space

Figure 2.18 (a) A tetrahedron built in the model coordinate system. The origin is (0, 0, 0) and the up direction is (0, 1, 0). (b) The tetrahedron placed in the world coordinate system whose origin is **O**, whose view direction is **D**, whose up direction is **U** = (0, 0, 1), and whose right direction is **R**.

space at the point $\mathbf{Q}_i = \mathbf{O} + d_i\mathbf{D} + u_i\mathbf{U} + r_i\mathbf{R}$, $0 \le i \le 3$. This is accomplished by constructing an affine transformation that maps the point $\mathbf{P}_0$ to the point $\mathbf{Q}_0$ and that maps the vectors $\mathbf{P}_i - \mathbf{P}_0$ to the vectors $\mathbf{Q}_i - \mathbf{Q}_0$ for $1 \le i \le 3$. The transformation is

$$\mathbf{Q} = \mathbf{Q}_0 + M(\mathbf{P} - \mathbf{P}_0)$$

where $M(\mathbf{P}_i - \mathbf{P}_0) = \mathbf{Q}_i - \mathbf{Q}_0$. In algebraic terms, we need

$$M\,[\,\mathbf{P}_1 - \mathbf{P}_0 \quad \mathbf{P}_2 - \mathbf{P}_0 \quad \mathbf{P}_3 - \mathbf{P}_0\,] = [\,\mathbf{Q}_1 - \mathbf{Q}_0 \quad \mathbf{Q}_2 - \mathbf{Q}_0 \quad \mathbf{Q}_3 - \mathbf{Q}_0\,]$$

where the two block matrices have columns using the vectors as indicated. The matrix M is therefore

$$M = [\,\mathbf{Q}_1 - \mathbf{Q}_0 \quad \mathbf{Q}_2 - \mathbf{Q}_0 \quad \mathbf{Q}_3 - \mathbf{Q}_0\,]\,[\,\mathbf{P}_1 - \mathbf{P}_0 \quad \mathbf{P}_2 - \mathbf{P}_0 \quad \mathbf{P}_3 - \mathbf{P}_0\,]^{-1}$$

The matrix M is said to be the *model-to-world transformation* for the tetrahedron, sometimes called the *model transformation* and sometimes called the *world transformation*. Other transformations involved in converting model-space points to points in other spaces have names that indicate the range of the transformation—the set of outputs from the transformation. To be consistent with this terminology, I will use the term *world transformation*.

Given a 3×3 matrix M, which represents scaling, rotation, reflection, shearing, and other linear operations, given a 3×1 translation vector **B**, and given a 3×1 model-space point $\mathbf{X}_{\text{model}}$, the corresponding 3×1 world-space point $\mathbf{X}_{\text{world}}$ is

generated by the homogeneous equation

$$\left[\begin{array}{c|c} \mathbf{X}_{\text{world}} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} M & \mathbf{B} \\ \hline \mathbf{0}^{\text{T}} & 1 \end{array}\right] \left[\begin{array}{c|c} \mathbf{X}_{\text{model}} \\ \hline 1 \end{array}\right] = H_{\text{world}} \left[\begin{array}{c|c} \mathbf{X}_{\text{model}} \\ \hline 1 \end{array}\right] \tag{2.59}$$

The matrix H_{world} is the *world matrix* in homogeneous form. Naturally, as long as M is invertible, we can map world-space points to model-space points by

$$\left[\begin{array}{c|c} \mathbf{X}_{\text{model}} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} M^{-1} & -M^{-1}\mathbf{B} \\ \hline \mathbf{0}^{\text{T}} & 1 \end{array}\right] \left[\begin{array}{c|c} \mathbf{X}_{\text{world}} \\ \hline 1 \end{array}\right] = H_{\text{world}}^{-1} \left[\begin{array}{c|c} \mathbf{X}_{\text{world}} \\ \hline 1 \end{array}\right] \tag{2.60}$$

where H_{world}^{-1} is the *inverse world matrix* in homogeneous form.

In the sample applications that ship with Wild Magic, the choice of the world space varies. Any objects that are loaded from disk are repositioned or reoriented as needed so that they are placed correctly in the world.

2.3.4 View, Camera, or Eye Space

So far we know about model space, the space where objects are created by the artists, and we know about world space, the space for the game environment itself. The objects are loaded into the game, but it is necessary to associate with them their world transformations. Model-space points are mapped to world-space points as needed.

A world-space point may also be located within the camera coordinate system. Once it is, the point is said to be in *view space* or *camera space* or *eye space* (all used by various people in the industry). The point must be represented as

$$\mathbf{X}_{\text{world}} = \mathbf{E} + d\mathbf{D} + u\mathbf{U} + r\mathbf{R}$$

where $\{\mathbf{E}; \mathbf{D}, \mathbf{U}, \mathbf{R}\}$ is the coordinate system for the camera. The coefficients are

$$d = \mathbf{D} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}), \qquad u = \mathbf{U} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}), \qquad r = \mathbf{R} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E})$$

The eye point and camera directions are chosen to be consistent with your world coordinate system. In the beginning, there was nothing—except for Cartesian space. Your intent is to fill Cartesian space with your beautiful creations, and then place an observer in the world to admire them. Of course, this requires you to impose a world coordinate system. In many cases, you will have an idea of which direction you want to be the up direction. Two directions perpendicular to the up direction are chosen to complete your coordinate axes. The choice of origin is made. The world coordinates are of your choosing. How you position and orient the observer is a separate matter. Nothing prevents you from placing the observer on the ground standing on his head! However, the typical placement will be to have the observer's up direction align with the world's up direction. What the observer sees is determined by your camera model.

The coefficients of $\mathbf{X}$ in the camera coordinate system are stored in a 3×1 vector and referred to as the *view coordinates* for the world point,

$$\mathbf{X}_{\text{view}} = \begin{bmatrix} r \\ u \\ d \end{bmatrix} = \begin{bmatrix} \mathbf{R} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}) \\ \mathbf{U} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}) \\ \mathbf{D} \cdot (\mathbf{X}_{\text{world}} - \mathbf{E}) \end{bmatrix} = \begin{bmatrix} \mathbf{R}^{\mathrm{T}} \\ \mathbf{U}^{\mathrm{T}} \\ \mathbf{D}^{\mathrm{T}} \end{bmatrix} (\mathbf{X}_{\text{world}} - \mathbf{E})$$
$$= [\, \mathbf{R} \quad \mathbf{U} \quad \mathbf{D} \,]^{\mathrm{T}} (\mathbf{X}_{\text{world}} - \mathbf{E})$$

where the first equality defines $\mathbf{X}_{\text{view}}$. Please observe that I am listing the components in the order (r, u, d), not in the natural order (d, u, r) that is associated with the coordinate system $\{\mathbf{E}; \mathbf{D}, \mathbf{U}, \mathbf{R}\}$! Effectively, (r, u, d) are the coordinates for the permuted coordinate system $\{\mathbf{E}; \mathbf{R}, \mathbf{U}, \mathbf{D}\}$, which happens to be left-handed. The Wild Magic software renderer implements the camera model in this way so that the last component of $\mathbf{X}_{\text{view}}$ is the view direction component. This choice was made to be consistent with the camera model of Direct3D, which is left-handed. OpenGL's camera coordinate system is internally stored as $\{\mathbf{E}; \mathbf{R}, \mathbf{U}, -\mathbf{D}\}$, which is right-handed. My initial attempt at dealing with this choice was to apply a sign change to the internal representation to produce $\mathbf{D}$. Having a consistent ordering is particularly important in vertex shader programs that transform and manipulate points and vectors in view space. My goal is to allow for the vertex shader programs to work with Wild Magic's software renderer, with the Direct3D renderer, and with the OpenGL renderer. The sample application for spherical environment mapping is a prototypical example where you manipulate view-space data.

Struggling with all the graphics APIs to make them consistent amounted to making programmatic adjustments to information obtained by API calls. For example, the camera coordinate system may be specified through Direct3D's utility functions `D3DXDXMATRIXLookAt*`. In OpenGL, the camera coordinate system may be specified through the utility function `gluLookAt`. For projections, Direct3D has utility functions `D3DXMatrixPerspective*` and `D3DXMatrixOrtho*`, whereas OpenGL has functions `glFrustum` and `glOrtho`. In the end, I tired of struggling and simply set the matrices directly—according to the coordinate system conventions I wanted, not the ones the graphics APIs want. The renderers were greatly simplified and a lot of code was factored into the base class for the renderers, a pleasant consequence. More details about this issue are found in Sections 2.8.3 and 2.8.4.

In homogeneous matrix form,

$$\left[\begin{array}{c} \mathbf{X}_{\text{view}} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} Q^{\mathrm{T}} & -Q^{\mathrm{T}}\mathbf{E} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{X}_{\text{world}} \\ \hline 1 \end{array}\right] = H_{\text{view}} \left[\begin{array}{c} \mathbf{X}_{\text{world}} \\ \hline 1 \end{array}\right] \tag{2.61}$$

where $Q = [\mathbf{R}\ \mathbf{U}\ \mathbf{D}]$ is the orthogonal matrix whose columns are the specified vectors. The homogenous matrix H_{view} in Equation (2.61) is referred to as the *view matrix*

and maps points from world space to view space. Points may be mapped from view space to world space using the inverse,

$$\left[\begin{array}{c} \mathbf{X}_{\text{world}} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c|c} Q & \mathbf{E} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{X}_{\text{view}} \\ \hline 1 \end{array}\right] = H_{\text{view}}^{-1} \left[\begin{array}{c} \mathbf{X}_{\text{view}} \\ \hline 1 \end{array}\right] \tag{2.62}$$

Just a reminder: Section 2.8.3 goes into great detail on the view matrix representation for Wild Magic, OpenGL, and Direct3D. These details were essential in making a single vertex shader program work for *all* the graphics APIs. You should definitely read the details if you plan on using more than one graphics API.

2.3.5 Clip, Projection, or Homogeneous Space

We are now ready to take our points in view coordinates and project them to obtain 2D coordinates for the screen. The process is factored into a few steps. The first step is to look more closely at the projection of Equation (2.42). We already looked at the projection in terms of camera coordinates in Section 2.2.4. The presentation here is in terms of homogeneous matrices so that you become comfortable with this approach rather than always relying on manipulating one component of a vector at a time.

The eye point $\mathbf{E}$ was chosen to be on the side of the projection plane (view plane) to which the normal vector $\mathbf{N}$ points. For our camera model, this direction is opposite to the view direction; namely, $\mathbf{N} = -\mathbf{D}$. A point on the view plane is $\mathbf{P} = \mathbf{E} + d_{\min}\mathbf{D}$. Using these choices and dividing the numerator and the denominator by -1, Equation (2.42) becomes

$$\mathbf{Y} = \frac{\left(\mathbf{E}\mathbf{D}^{\mathrm{T}} + d_{\min} I\right)(\mathbf{X} - \mathbf{E})}{\mathbf{D}^{\mathrm{T}}(\mathbf{X} - \mathbf{E})}$$

The homogeneous form of this equation, which by convention does not include the perspective divide, and whose general form is Equation (2.49), is shown in the following equation:

$$\left[\begin{array}{c} \mathbf{Y}'_{\text{world}} \\ \hline w_{\text{world}} \end{array}\right] = \left[\begin{array}{c|c} \mathbf{E}\mathbf{D}^{\mathrm{T}} + d_{\min} I & -(\mathbf{E}\mathbf{D}^{\mathrm{T}} + d_{\min} I)\mathbf{E} \\ \hline \mathbf{D}^{\mathrm{T}} & -\mathbf{D}^{\mathrm{T}}\mathbf{E} \end{array}\right] \left[\begin{array}{c} \mathbf{X}_{\text{world}} \\ \hline 1 \end{array}\right]$$

Notice that I have subscripted the various terms to make it very clear that they are quantities in world coordinates. Since we already know how to map points from model space to world space, and then from world space to view space, it will be convenient to formulate the homogeneous equation so that its inputs are points in view space and its outputs are points in homogeneous view space, so to speak. We can convert the output from world space to view space using the view matrix of Equation

(2.61), replace the world-space input with the inverse view matrix of Equation (2.62) times the view-space input, and use $M = \mathbf{E}\mathbf{D}^T + d_{\min} I$ to obtain

$$\begin{aligned}
\left[\begin{array}{c} \mathbf{Y}'_{\text{view}} \\ \hline w_{\text{view}} \end{array}\right] &= \left[\begin{array}{c|c} Q^{\mathrm{T}} & -Q^{\mathrm{T}}\mathbf{E} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{Y}'_{\text{world}} \\ \hline w_{\text{world}} \end{array}\right] \\
&= \left[\begin{array}{c|c} Q^{\mathrm{T}} & -Q^{\mathrm{T}}\mathbf{E} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c|c} M & -M\mathbf{E} \\ \hline \mathbf{D}^{\mathrm{T}} & -\mathbf{D}^{\mathrm{T}}\mathbf{E} \end{array}\right] \left[\begin{array}{c} \mathbf{X}_{\text{world}} \\ \hline 1 \end{array}\right] \\
&= \left[\begin{array}{c|c} Q^{\mathrm{T}} & -Q^{\mathrm{T}}\mathbf{E} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c|c} M & -M\mathbf{E} \\ \hline \mathbf{D}^{\mathrm{T}} & -\mathbf{D}^{\mathrm{T}}\mathbf{E} \end{array}\right] \left[\begin{array}{c|c} Q & \mathbf{E} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{X}_{\text{view}} \\ \hline 1 \end{array}\right] \\
&= \left[\begin{array}{c|c} d_{\min} Q^{\mathrm{T}} & -d_{\min} Q^{T}\mathbf{E} \\ \hline \mathbf{D}^{\mathrm{T}} & -\mathbf{D}^{\mathrm{T}}\mathbf{E} \end{array}\right] \left[\begin{array}{c|c} Q & \mathbf{E} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{X}_{\text{view}} \\ \hline 1 \end{array}\right] \\
&= \left[\begin{array}{c|c} d_{\min} I & \mathbf{0} \\ \hline \mathbf{D}^{\mathrm{T}} Q & 0 \end{array}\right] \left[\begin{array}{c} \mathbf{X}_{\text{view}} \\ \hline 1 \end{array}\right] \\
&= \left[\begin{array}{c} d_{\min}\mathbf{X}_{\text{view}} \\ \hline \mathbf{D}^{\mathrm{T}} Q \mathbf{X}_{\text{view}} \end{array}\right] \\
&= \left[\begin{array}{c} d_{\min} r \\ d_{\min} u \\ \hline d_{\min} d \\ \hline d \end{array}\right]
\end{aligned} \tag{2.63}$$

where you will recall that $\mathbf{X}_{\text{view}} = (r, u, d)$. The perspective divide produces the actual projection,

$$\mathbf{Y}_{\text{proj}} = \frac{\mathbf{Y}'_{\text{view}}}{w_{\text{view}}} = \begin{bmatrix} \frac{d_{\min} r}{d} \\ \frac{d_{\min} u}{d} \\ d_{\min} \end{bmatrix} \tag{2.64}$$

The last component of $\mathbf{Y}_{\text{proj}}$ makes sense because the view plane is $d_{\min}$ units of distance from the eye point and we designed the projection to be onto the view plane.

The axis of the view frustum is the ray that contains both the origin and the center point of the viewport. This ray is parameterized by d in view coordinates as

$$\left(\frac{(r_{\min} + r_{\max})d}{2d_{\min}}, \frac{(u_{\min} + u_{\max})d}{2d_{\min}}, d \right), \quad d_{\min} \le d \le d_{\max}$$

It is convenient to transform the (possibly) skewed view frustum into an orthogonal frustum with viewport $[-1, 1]^2$. We accomplish this by removing the skew, then scaling the result:

$$\begin{aligned} r' &= \frac{2}{r_{\max} - r_{\min}} \left(d_{\min} r - \frac{r_{\min} + r_{\max}}{2} d \right), \\ u' &= \frac{2}{u_{\max} - u_{\min}} \left(d_{\min} u - \frac{u_{\min} + u_{\max}}{2} d \right) \end{aligned} \tag{2.65}$$

To keep consistent with the primed notation r' and u', define

$$w' = d \tag{2.66}$$

The view frustum is now delimited by $|r'| \le w'$, $|u'| \le w'$, and $d_{\min} \le w' \le d_{\max}$. The projection is $(r'/w', u'/w')$, so $|r'/w'| \le 1$ and $|u'/w'| \le 1$.

It is also convenient to transform the d-values in $[d_{\min}, d_{\max}]$ so that the new range is $[0, 1]$. This is somewhat tricky because the transformation should be consistent with the perspective projection. The affine transformation $d' = (d - d_{\min})/(d_{\max} - d_{\min})$ is not the correct one to use. Equation (2.43) saves the day. The d-values in $[d_{\min}, d_{\max}]$ can be written as

$$d = (1 - s) d_{\min} + s d_{\max}$$

for $s \in [0, 1]$. We can solve this for $s = (d - d_{\min})/(d_{\max} - d_{\min})$ and use Equation (2.43) with $w_0 = d_{\min}$, the minimum w'-value, and $w_1 = d_{\max}$, the maximum w'-value, to obtain

$$\bar{s} = \frac{w_1 s}{w_0 + (w_1 - w_0) s} = \frac{d_{\max}}{d_{\max} - d_{\min}} \left(1 - \frac{d_{\min}}{d} \right)$$

Observe that $\bar{s} \in [0, 1]$. This value plays the role of a *normalized depth* in rendering. The equation for $\bar{s}$ already has the perspective division. Before division, we can define

$$d' = \frac{d_{\max}}{d_{\max} - d_{\min}} \left(d - d_{\min} \right) \tag{2.67}$$

so that $\bar{s} = d'/w'$.

Equations (2.65) through (2.67) may be combined into a homogeneous matrix transformation that maps $(r, u, d, 1)$ to (r', u', d', w'):

$$\begin{aligned}
\mathbf{X}_{\text{clip}} &= \begin{bmatrix} r' \\ u' \\ d' \\ \hline w' \end{bmatrix} \\
&= \left[\begin{array}{cccc} \frac{2d_{\min}}{r_{\max}-r_{\min}} & 0 & \frac{-(r_{\max}+r_{\min})}{r_{\max}-r_{\min}} & 0 \\ 0 & \frac{2d_{\min}}{u_{\max}-u_{\min}} & \frac{-(u_{\max}+u_{\min})}{u_{\max}-u_{\min}} & 0 \\ 0 & 0 & \frac{d_{\max}}{d_{\max}-d_{\min}} & \frac{-d_{\max}d_{\min}}{d_{\max}-d_{\min}} \\ \hline 0 & 0 & 1 & 0 \end{array}\right] \begin{bmatrix} r \\ u \\ d \\ \hline 1 \end{bmatrix} \\
&= H_{\text{proj}} \begin{bmatrix} \mathbf{X}_{\text{view}} \\ \hline 1 \end{bmatrix}
\end{aligned} \tag{2.68}$$

This equation defines two quantities, the homogeneous *projection matrix* H_{proj} and the homogeneous point $\mathbf{X}_{\text{clip}}$, which is a point said to be in *clip space* and its components are referred to as *clip coordinates*.

Clip coordinates are used both for culling back-facing triangles and for clipping triangles against the view frustum. Although you could do these calculations in world space or in view space, the number of calculations is fewer in clip space. Moreover, the access to the graphics pipeline provided via vertex shaders essentially requires you to compute points in clip coordinates, which are then returned to the graphics driver for rasterization.

All that said, you might have looked at Equation (2.68) and concluded that it looks neither like OpenGL's projection matrix nor like Direct3D's projection matrix. I will explicitly compare these in Sections 2.8.3 and 2.8.4. Suffice it to say that my OpenGL and Direct3D renderers have been implemented to use the exact same view and projection matrices, thereby ignoring the defaults that occur when you go through utility functions provided by the graphics APIs.

Just as I have provided the inverses for the world matrix and the view matrix, the inverse of the projection matrix is

$$H_{\text{proj}}^{-1} = \begin{bmatrix} \frac{r_{\max}-r_{\min}}{2d_{\min}} & 0 & 0 & \frac{r_{\max}+r_{\min}}{2d_{\min}} \\ 0 & \frac{u_{\max}-u_{\min}}{2d_{\min}} & 0 & \frac{u_{\max}+u_{\min}}{2d_{\min}} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{d_{\max}-d_{\min}}{d_{\max}d_{\min}} & \frac{1}{d_{\min}} \end{bmatrix} \tag{2.69}$$

2.3.6 WINDOW SPACE

The clip-space point (r', u', d', w') has the properties that $|r'| \le w'$, $|u'| \le w'$, $0 \le d' \le d_{\max}$, and $d_{\min} \le w' \le d_{\max}$. We finally perform the perspective division to obtain

$$\mathbf{X}_{\text{ndc}} = \begin{bmatrix} r'' \\ u'' \\ d'' \\ 1 \end{bmatrix} = \begin{bmatrix} r'/w' \\ u'/w' \\ d'/w' \\ w'/w' \end{bmatrix} \tag{2.70}$$

where $r'' \in [-1, 1]$, $u'' \in [-1, 1]$, and $d'' \in [0, 1]$. The 3-tuples (r'', u'', d'') are said to be *normalized device coordinates* (NDCs). The term *normalized* was intended to refer to the components of the 3-tuples being somehow in intervals $[0, 1]$ or $[-1, 1]$. The normalization, however, is not normal across APIs. Wild Magic and Direct3D use $d'' \in [0, 1]$. OpenGL has a default projection matrix that leads to a projected value $d'' \in [-1, 1]$. This is yet another API convention you need to be aware of; see Section 2.8.4 for more details. But as I have mentioned repeatedly, my implementations of the renderers all use the same projection matrix, so in fact my OpenGL renderer has $d'' \in [0, 1]$.

The goal now is to map (r'', u'') to a pixel of the window created by your application. One important detail is that (r'', u'') are right-handed coordinates with respect to the viewport on the view plane. The r''-values increase as you move to the right within the viewport and the u''-values increase as you move up within the viewport. The window pixels have coordinates (x, y) that are left-handed. The x-values increase as you move to the right in the window and the y-values increase as you move down the window. The conversion from (r'', u'') to (x, y) requires a reflection in u'' to switch handedness. If the window has width W pixels and height H pixels, then $0 \le x < W$, $0 \le y < H$, and a mapping is $x = W(1 + r'')/2$ and $y = H(1 - u'')/2$. The computations are real-valued, but in software the values are truncated to the nearest integer and then clamped to be within the valid pixel domain to produce the indices into video memory for the screen. This mapping takes clip-space points to the full viewport on the view plane. As mentioned in Section 2.3.1, you might want the drawing of objects to occur in a subrectangle of the viewport. The camera model includes parameters p_ℓ, p_r, p_b, and p_t with $0 \le p_\ell < p_r \le 1$ and $0 \le p_b < p_t \le 1$. The mapping from $(r'', u'') \in [-1, 1]^2$ to the subrectangle is

$$\begin{aligned} x &= \left(\frac{1 - r''}{2}\right) p_\ell W + \left(\frac{1 + r''}{2}\right) p_r W = \frac{W}{2}\left[(p_r + p_\ell) + (p_r - p_\ell) r''\right] \\ y &= H - \left[\left(\frac{1 - u''}{2}\right) p_b H + \left(\frac{1 + u''}{2}\right) p_t H\right] \\ &= \frac{H}{2}\left[(2 - p_t - p_b) + (p_b - p_t) u''\right] \end{aligned} \tag{2.71}$$

The slightly more complicated conversion for u'' has to do with the switch from right-handed to left-handed coordinates.

The depth values $d'' \in [0, 1]$ can also be mapped to a depth range that is a subset of $[0, 1]$. Section 2.3.1 introduced the depth range interval $[p_n, p_f] \subseteq [0, 1]$. The new depth values for this range are

$$\delta = (p_f - p_n)\, d'' + p_n \tag{2.72}$$

Equations (2.71) and (2.72) may be combined into a single 4-tuple, which I will call the *window coordinates* of the corresponding clip-space point:

$$\begin{bmatrix} \mathbf{X}_{\text{window}} \\ \hline 1 \end{bmatrix} = \left[\begin{array}{ccc|c} \frac{W(p_r-p_\ell)}{2} & 0 & 0 & \frac{W(p_r+p_\ell)}{2} \\ 0 & \frac{H(p_b-p_t)}{2} & 0 & \frac{H(2-p_t-p_b)}{2} \\ 0 & 0 & p_f - p_n & p_n \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} \mathbf{X}_{\text{ndc}} \\ \hline 1 \end{bmatrix}$$

$$= H_{\text{window}} \begin{bmatrix} \mathbf{X}_{\text{ndc}} \\ \hline 1 \end{bmatrix} \tag{2.73}$$

The CD-ROM accompanying this book contains a software renderer that implements all the transformations described in this section. The vertex shader unit takes model-space points and produces clip-space points. The rasterizer clips the points and generates the pixels that are covered by a triangle via interpolation. Each interpolated clip-space point is mapped to a window-space point to produce the pixel location and depth. The pixel shader unit processes each such pixel.

EXERCISE 2.10

The window matrix of Equation (2.73) was developed using the mapping of $r'' \in [-1, 1]$ to $x \in [p_\ell W, p_r W]$ and $u'' \in [-1, 1]$ to $y \in [p_b H, p_t H]$, with a reflection when computing the y-value. This choice was made to be consistent with OpenGL, according to the documentation describing this mapping. The DirectX documentation [Cor] does not mention the precise details of the mapping. When the full viewport is used ($p_\ell = 0$, $p_r = 1$, $p_b = 0$, $p_t - 1$), notice that $r'' = 1$ is mapped to $x = W$ and $u'' = -1$ is mapped to $y = H$, but actual pixel coordinates must satisfy $x \leq W - 1$ and $y \leq H - 1$, so clamping will always occur at these extremes. What differences in visual behavior would you expect if you were to use a different mapping?

One alternative is to map r'' to $x \in [p_\ell W, p_r W - 1]$ and u'' to $y \in [p_b H, p_t H - 1]$, with a reflection. What is the window matrix for this transformation?

Another alternative is to map r'' to $x \in [p_\ell(W - 1), p_r(W - 1)]$ and u'' to $y \in [p_b(H - 1), p_t(H - 1)]$, with a reflection. What is the window matrix for this transformation?

Try experimenting with these in the software renderer contained on this book's companion CD-ROM. For each suggested alternative, also modify the OpenGL and Direct3D rendering code (functions `OnViewportChange`) and see how the visual behavior changes. ■

EXERCISE 2.11 What is the inverse matrix for the window matrix of Equation (2.73)? ■

2.3.7 PUTTING THEM ALL TOGETHER

The application of transformations from model space to window space is referred to as the *geometric pipeline*. The following diagram shows all the steps, including references to the equations that define the transformations.

$$
\begin{array}{lllll}
\mathbf{X}_{\text{model}} & & & & \\
\downarrow & \text{world matrix,} & H_{\text{world}}, & \text{Equation (2.59)} & \\
\mathbf{X}_{\text{world}} & & & & \\
\downarrow & \text{view matrix,} & H_{\text{view}}, & \text{Equation (2.61)} & \\
\mathbf{X}_{\text{view}} & & & & \\
\downarrow & \text{projection matrix,} & H_{\text{proj}}, & \text{Equation (2.68)} & (2.74) \\
\mathbf{X}_{\text{clip}} & & & & \\
\downarrow & \text{perspective divide,} & & \text{Equation (2.70)} & \\
\mathbf{X}_{\text{ndc}} & & & & \\
\downarrow & \text{window matrix,} & H_{\text{window}}, & \text{Equation (2.73)} & \\
\mathbf{X}_{\text{window}} & & & &
\end{array}
$$

A software renderer implements the entire geometric pipeline. The companion CD-ROM has such a renderer to illustrate the concepts discussed in this book. A hardware-accelerated renderer implements the pipeline and allows you, through a graphics API, to specify the matrices in the pipeline, either directly or indirectly.

EXAMPLE 2.4 A triangle is created in a model space with points labeled (x, y, z). The model-space vertices are $\mathbf{V}_0 = (0, 0, 0)$, $\mathbf{V}_1 = (1, 0, 0)$, and $\mathbf{V}_2 = (0, 0, 1)$. Figure 2.19 shows a rendering of the triangle in model space. The world space is chosen with origin $(0, 0, 0)$ and with an up vector in the direction of the positive z-axis. The model triangle is to be rotated and translated so that the world-space vertices are $\mathbf{W}_0 = (1, 1, 1)$, $\mathbf{W}_1 = (1, 2, 1)$, and $\mathbf{W}_2 = (1, 1, 2)$. Figure 2.20 shows a rendering of the triangle in world space. The world matrix is

$$H_{\text{world}} = \left[\begin{array}{ccc|c} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

and transforms $(\mathbf{V}_i, 1)$ to $(\mathbf{W}_i, 1)$ for all i.

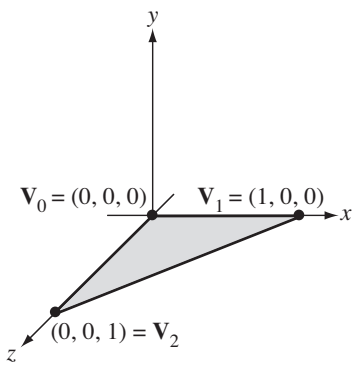


Figure 2.19 A model triangle to be sent through the geometric pipeline.

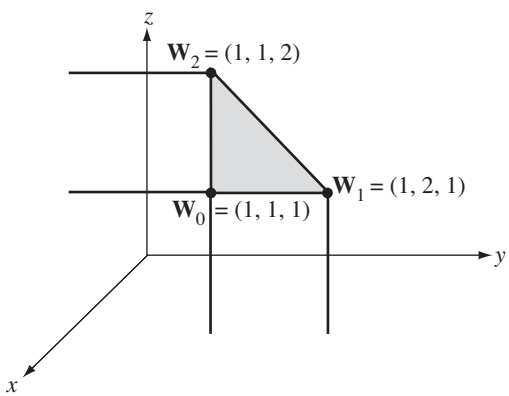


Figure 2.20 The world triangle corresponding to the model triangle of Figure 2.19.

The screen is chosen to have a width of 640 pixels and a height of 480 pixels. The camera is positioned in the world with eye point at $\mathbf{E} = (5/2, 3, 7/2)$, with view direction $\mathbf{D} = (-1, -1, -1)/\sqrt{3}$, and up direction $\mathbf{U} = (-1, -1, 2)/\sqrt{6}$. The right direction is $\mathbf{R} = \mathbf{D} \times \mathbf{U} = (-1, 1, 0)/\sqrt{2}$. The view matrix is

$$H_{\text{view}} = \left[\begin{array}{cccc} \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & \frac{-1}{2\sqrt{2}} \\ \frac{-1}{\sqrt{6}} & \frac{-1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & \frac{-3}{2\sqrt{6}} \\ \frac{-1}{\sqrt{3}} & \frac{-1}{\sqrt{3}} & \frac{-1}{\sqrt{3}} & \frac{9}{\sqrt{3}} \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

An orthogonal frustum will be used to render the triangle. The frustum near and far parameters are chosen to be $d_{\min} = 1$ and $d_{\max} = 10$, respectively. The vertical field of view is chosen to be $2\theta_u = \pi/3$ and the aspect ratio is $\rho = 4/3 = 640/480$. Equation (2.58) is used to compute $u_{\max} = d_{\min} \tan(\theta_u) = 1/\sqrt{3}$ and $r_{\max} = \rho u_{\max} = 4/(3\sqrt{3})$. By symmetry, $u_{\min} = -u_{\max}$ and $r_{\min} = -r_{\max}$. The projection matrix is

$$H_{\text{proj}} = \left[\begin{array}{cccc} \frac{3\sqrt{3}}{4} & 0 & 0 & 0 \\ 0 & \sqrt{3} & 0 & 0 \\ 0 & 0 & \frac{10}{9} & \frac{-10}{9} \\ \hline 0 & 0 & 1 & 0 \end{array}\right]$$

We will use the full viewport, so $p_\ell = p_b = 0$ and $p_r = p_t = 1$. Also, we will use the full depth range, so $p_n = 0$ and $p_f = 1$. The screen matrix is

$$H_{\text{screen}} = \left[\begin{array}{cccc} \frac{639}{2} & 0 & 0 & \frac{639}{2} \\ 0 & \frac{-479}{2} & 0 & \frac{479}{2} \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

All the matrices are ready to use, so let us transform the model-space vertices and see where they land on the screen. To make Table 2.2 typesetting friendly, I will write the 4-tuples in the form $(a, b, c; d)$, using a semicolon to separate the last component from the first three. Computing the screen-space coordinates, the final points (x, y) and normalized depths $\delta \in [0, 1]$ are

$$(x_0, y_0; \delta_0) = (277.139884, 312.831599; 0.790360)$$

$$(x_1, y_1; \delta_1) = (370.332138, 386.163198; 0.726210)$$

$$(x_2, y_2; \delta_2) = (268.667861, 210.167360; 0.726210)$$

Table 2.2 Results of the transformations applied during the geometric pipeline.

Space	*Vertex 0*	*Vertex 1*	*Vertex 2*
Model	(0, 0, 0; 1)	(1, 0, 0; 1)	(0, 0, 1; 1)
World	(1, 1, 1; 1)	(1, 2, 1; 1)	(1, 1, 2; 1)
View	$\left(\frac{-1}{2\sqrt{2}}, \frac{-3}{2\sqrt{6}}, \frac{6}{\sqrt{3}}; 1\right)$	$\left(\frac{1}{2\sqrt{2}}, \frac{-5}{2\sqrt{6}}, \frac{5}{\sqrt{3}}; 1\right)$	$\left(\frac{-1}{2\sqrt{2}}, \frac{1}{2\sqrt{6}}, \frac{5}{\sqrt{3}}; 1\right)$
Clip	$\left(\frac{-3\sqrt{3}}{8\sqrt{2}}, \frac{-3}{2\sqrt{2}}, \frac{20}{3\sqrt{3}} - \frac{10}{9}; \frac{6}{\sqrt{3}}\right)$	$\left(\frac{3\sqrt{3}}{8\sqrt{2}}, \frac{-5}{2\sqrt{2}}, \frac{50}{9\sqrt{3}} - \frac{10}{9}; \frac{5}{\sqrt{3}}\right)$	$\left(\frac{-3\sqrt{3}}{8\sqrt{2}}, \frac{1}{2\sqrt{2}}, \frac{50}{9\sqrt{3}} - \frac{10}{9}; \frac{5}{\sqrt{3}}\right)$
NDC	$\left(\frac{-3}{16\sqrt{2}}, \frac{-\sqrt{3}}{4\sqrt{2}}, \frac{10}{9} - \frac{5\sqrt{3}}{27}; 1\right)$	$\left(\frac{9}{40\sqrt{2}}, \frac{-\sqrt{3}}{2\sqrt{2}}, \frac{10}{9} - \frac{2\sqrt{3}}{9}; 1\right)$	$\left(\frac{-9}{40\sqrt{2}}, \frac{\sqrt{3}}{10\sqrt{2}}, \frac{10}{9} - \frac{2\sqrt{3}}{9}; 1\right)$
Screen	$\left(\frac{639}{2} - \frac{1917}{32\sqrt{2}}, \frac{479}{2} + \frac{479\sqrt{3}}{8\sqrt{2}}, \frac{10}{9} - \frac{5\sqrt{3}}{27}; 1\right)$	$\left(\frac{639}{2} + \frac{5751}{80\sqrt{2}}, \frac{479}{2} + \frac{479\sqrt{3}}{4\sqrt{2}}, \frac{10}{9} - \frac{2\sqrt{3}}{9}; 1\right)$	$\left(\frac{639}{2} - \frac{5751}{80\sqrt{2}}, \frac{479}{2} - \frac{479\sqrt{3}}{20\sqrt{2}}, \frac{10}{9} - \frac{2\sqrt{3}}{9}; 1\right)$

The x and y values are rounded to the nearest integer, so the actual pixel locations for the projected vertices are (277, 313), (370, 386), and (269, 210). Figure 2.21 shows the final image drawn by the Wild Magic software renderer to a 640×480 window. The coordinate axes were drawn as three separate polylines. The 640×480 image was reduced in size, with averaging, to a 320×240 image. The border around the window and the axis labels were added via a paint program.

Naturally, the geometric pipeline is part of the rendering system. The application code that led to Figure 2.21 created the model-space triangle, the world matrix, and a simple scene, and it did some basic setup for rendering. The application header file is

```
#ifndef GEOMETRICPIPELINE_H
#define GEOMETRICPIPELINE_H

#include "Wm4WindowApplication3.h"
using namespace Wm4;

class GeometricPipeline : public WindowApplication3
{
    WM4_DECLARE_INITIALIZE;
```

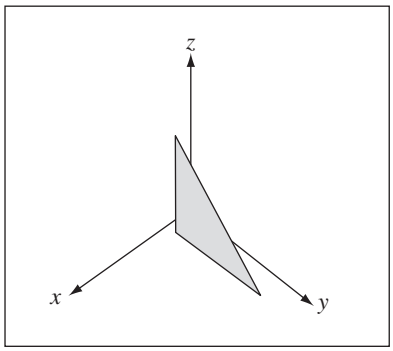


Figure 2.21 A software-rendered image of the triangle.

```
public:
    GeometricPipeline ();

    virtual bool OnInitialize ();
    virtual void OnTerminate ();
    virtual void OnIdle ();

protected:
    void CreateScene ();

    NodePtr m_spkScene;
    TriMeshPtr m_spkTriangle;
    PolylinePtr m_spkAxes;
    Culler m_kCuller;
};

WM4_REGISTER_INITIALIZE(GeometricPipeline);

#endif
```

The application source code is

```
#include "GeometricPipeline.h"

WM4_WINDOW_APPLICATION(GeometricPipeline);
```

```
//----------------------------------------------------------------------------
GeometricPipeline::GeometricPipeline ()
    :
    WindowApplication3("GeometricPipeline",0,0,640,480,ColorRGBA::WHITE)
{
}
//----------------------------------------------------------------------------
bool GeometricPipeline::OnInitialize ()
{
    if (!WindowApplication3::OnInitialize())
    {
        return false;
    }

    // Create the camera model.
    m_spkCamera->SetFrustum(60.0f,4.0f/3.0f,1.0f,10.0f);
    Vector3f kCLoc(2.5f,3.0f,3.5f);
    Vector3f kCDir(-1.0f,-1.0f,-1.0f);
    kCDir.Normalize();
    Vector3f kCUp(-1.0f,-1.0f,2.0f);
    kCUp.Normalize();
    Vector3f kCRight = kCDir.Cross(kCUp);
    m_spkCamera->SetFrame(kCLoc,kCDir,kCUp,kCRight);

    CreateScene();

    // The initial update of objects.
    m_spkScene->UpdateGS();
    m_spkScene->UpdateRS();

    // The initial culling of the scene.
    m_kCuller.SetCamera(m_spkCamera);
    m_kCuller.ComputeVisibleSet(m_spkScene);

    InitializeCameraMotion(0.1f,0.01f);
    InitializeObjectMotion(m_spkScene);
    return true;
}
//----------------------------------------------------------------------------
void GeometricPipeline::OnTerminate ()
{
    m_spkScene = 0;
    m_spkTriangle = 0;
```

```
    m_spkAxes = 0;
    WindowApplication3::OnTerminate();
}
//----------------------------------------------------------------------------
void GeometricPipeline::OnIdle ()
{
    MeasureTime();

    if (MoveCamera())
    {
        m_kCuller.ComputeVisibleSet(m_spkScene);
    }

    if (MoveObject())
    {
        m_spkScene->UpdateGS();
        m_kCuller.ComputeVisibleSet(m_spkScene);
    }

    m_pkRenderer->ClearBuffers();
    if (m_pkRenderer->BeginScene())
    {
        m_pkRenderer->DrawScene(m_kCuller.GetVisibleSet());
        DrawFrameRate(8,GetHeight()-8,ColorRGBA::WHITE);
        m_pkRenderer->EndScene();
    }
    m_pkRenderer->DisplayBackBuffer();

    UpdateFrameCount();
}
//----------------------------------------------------------------------------
void GeometricPipeline::CreateScene ()
{
    // Create the model-space triangle.
    Attributes kAttr;
    kAttr.SetPChannels(3);
    VertexBuffer* pkVBuffer = WM4_NEW VertexBuffer(kAttr,3);
    pkVBuffer->Position3(0) = Vector3f(0.0f,0.0f,0.0f);
    pkVBuffer->Position3(1) = Vector3f(1.0f,0.0f,0.0f);
    pkVBuffer->Position3(2) = Vector3f(0.0f,0.0f,1.0f);

    IndexBuffer* pkIBuffer = WM4_NEW IndexBuffer(3);
    int* aiIndex = pkIBuffer->GetData();
```

```
    aiIndex[0] = 0;
    aiIndex[1] = 1;
    aiIndex[2] = 2;

    m_spkTriangle = WM4_NEW TriMesh(pkVBuffer,pkIBuffer);

    // Set the world matrix.
    m_spkTriangle->Local.SetTranslate(Vector3f(1.0f,1.0f,1.0f));
    m_spkTriangle->Local.SetRotate(Matrix3f(Vector3f::UNIT_Z,Mathf::HALF_PI));

    // Attach a material to the triangle.
    MaterialState* pkMS = WM4_NEW MaterialState;
    pkMS->Diffuse = ColorRGB(0.5f,0.5f,0.5f);
    m_spkTriangle->AttachGlobalState(pkMS);
    m_spkTriangle->AttachEffect(WM4_NEW MaterialEffect);

    // Create the coordinate axes.
    pkVBuffer = WM4_NEW VertexBuffer(kAttr,6);
    pkVBuffer->Position3(0) = Vector3f::ZERO;
    pkVBuffer->Position3(1) = 2.0f*Vector3f::UNIT_X;
    pkVBuffer->Position3(2) = Vector3f::ZERO;
    pkVBuffer->Position3(3) = 2.0f*Vector3f::UNIT_Y;
    pkVBuffer->Position3(4) = Vector3f::ZERO;
    pkVBuffer->Position3(5) = 2.0f*Vector3f::UNIT_Z;

    m_spkAxes = WM4_NEW Polyline(pkVBuffer,false,false);

    // Attach a material to the axes.
    pkMS = WM4_NEW MaterialState;
    pkMS->Diffuse = ColorRGB::BLACK;
    m_spkAxes->AttachGlobalState(pkMS);
    m_spkAxes->AttachEffect(WM4_NEW MaterialEffect);

    m_spkScene = WM4_NEW Node;
    m_spkScene->AttachChild(m_spkTriangle);
    m_spkScene->AttachChild(m_spkAxes);
}
//----------------------------------------------------------------------------
```

In Wild Magic, the application layer is agnostic of renderer type. The code works for the OpenGL renderer, for the Direct3D renderer, and for the Wild Magic software renderer.

EXERCISE 2.12 Repeat the calculations in Example 2.4, but using a camera positioned at $\mathbf{E} = (4, 4, 4)$ and with a far-plane distance of $d_{\max} = 4$. Also repeat the calculations with the original settings, except place the camera at $\mathbf{E} = (1, -1, 3/2)$. How is the rendering of the triangle in this case different from the rendering in Figure 2.21? ■

EXERCISE 2.13 Suppose you want your application to support selecting a window pixel with the left button of the mouse. When the selected pixel is part of a rendered 3D object, compute the world-space coordinates for the 3D object point that was rendered to the selected pixel. Add this code to the `GeometricPipeline` application whose source code was listed previously. Write text to the upper-left corner of the screen that displays the (x, y) value you selected with the mouse and the corresponding world-space coordinates of the object drawn to that pixel. ■

2.4 CULLING AND CLIPPING

Culling and clipping of objects reduces the amount of data sent to the rasterizer for drawing. Culling refers to eliminating portions of an object, possibly the entire object, that are not visible to the eye point. For an object represented by a triangle mesh, the typical culling operations amount to determining which triangles are outside the view frustum and which triangles are facing away from the eye point. Clipping refers to computing the intersection of an object with the view frustum, and with additional planes provided by the application such as in a portal system (see Section 6.3), so that only the visible portion of the object is sent to the rasterizer. For an object represented by a triangle mesh, the typical clipping operations amount to splitting triangles by the various view frustum planes and retaining only those triangles inside the frustum.

2.4.1 OBJECT CULLING

Object culling involves deciding whether or not an object as a whole is contained in the view frustum. If an object is not in the frustum, there is no point in consuming CPU or GPU cycles to process the object for the rasterizer. Typically, the application maintains a bounding volume for each object. The idea is to have an inexpensive test for nonintersection between bounding volume and view frustum that can lead to quick rejection of an object for further processing. If the bounding volume of an object does intersect the view frustum, then the entire object is processed further even if that object does not lie entirely inside the frustum. It is also possible that the bounding volume and view frustum intersect, but the object is completely outside the frustum.

A test to determine if the bounding volume and view frustum intersect can be an expensive operation. Such a test is said to be an *exact culling test*. An *inexact culling test* is designed to be faster, reporting nonintersections in *most* cases, but is conservative in that it might report an intersection when there is none. The idea is that the total

time for culling and drawing is, hopefully, less than the total time if you were to use exact culling. Specifically, what you hope to be the common situation is

$$\text{Cost(inexact\_culling)} < \text{Cost(exact\_culling)}$$

$$\text{Cost(inexact\_drawing)} > \text{Cost(exact\_drawing)}$$

$$\text{Cost(inexact\_culling)} + \text{Cost(inexact\_drawing)} < \text{Cost(exact\_culling)} + \text{Cost(exact\_drawing)}$$

The only way you can test this hypothesis is by experimenting within your own applications and graphics framework. If you find that over the lifetime of your application's execution the total time of culling and drawing is smaller when using exact culling, then you should certainly use exact culling. Some exact culling tests are described in Section 15.7.

The standard approach to inexact culling against the view frustum is to compare the object's bounding volume against the view frustum planes, one at a time. Figure 2.22 illustrates the various possibilities for culling by testing a plane at a time. The situation shown in Figure 2.22 (a) occurs whether you use exact culling or inexact culling of bounding volumes. The problem is simply that the bounding volume is an approximation of the region that the object occupies; there will always be situations when the bounding volume intersects the frustum but the object does not. The situation shown in Figure 2.22 (c) is what makes the plane-by-plane culling inexact. The bounding volume is not outside any frustum plane, but it is outside the entire view frustum.

2.4.2 Back-Face Culling

Object culling is an attempt to eliminate the entire object from being processed by the renderer. If an object is not culled based on its bounding volume, then the renderer has the opportunity to reduce the amount of data it must draw. The next level of culling is called *back-face culling*. The triangles are oriented so that their normal vectors point outside the object whose surface they comprise. If the triangle is oriented away from the eye point, then that triangle is not visible and need not be drawn by the renderer. For a perspective projection, the test for a back-facing triangle is to determine if the eye point is on the negative side of the plane of the triangle (the triangle is a "back face" of the object to be rendered). If $\mathbf{E}$ is the world eye point and if the plane of the triangle is $\mathbf{N} \cdot \mathbf{X} = d$, then the triangle is back facing if $\mathbf{N} \cdot \mathbf{E} < d$. Figure 2.23 shows the front view of an object. The front-facing triangles are drawn with solid lines. The back-facing triangles are indicated with dotted lines (although they would not be drawn at all by the renderer).

The vertex data that is sent to the graphics driver stores only vertex positions, not triangle normals. This means the renderer must compute the normal vector for each triangle to use in the back-face test. Mathematically, it does not matter in which

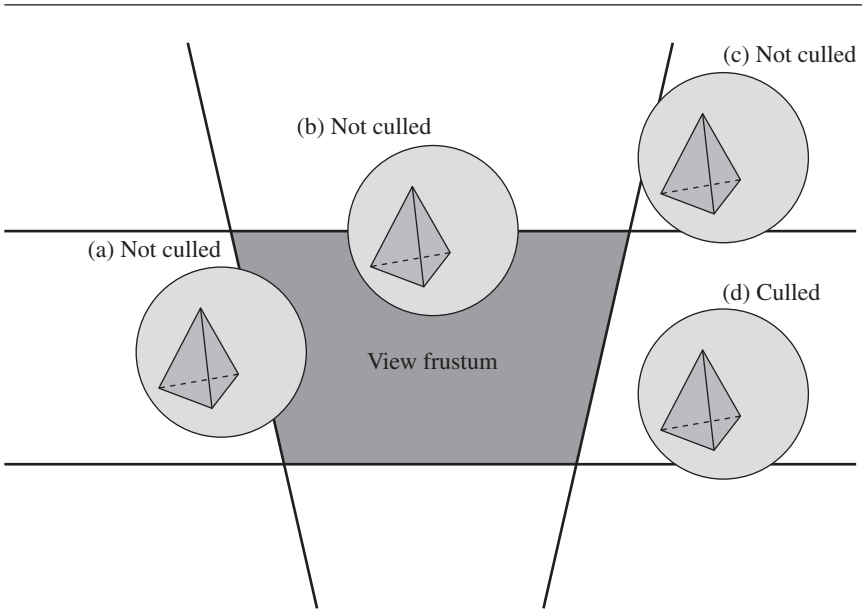


Figure 2.22 Attempts to cull objects, whose bounding volumes are spheres, a frustum plane at a time. In (a), (b), and (c), the bounding volumes are not outside any of the frustum planes, so an attempt will be made to draw those objects. In (a), the bounding volume is not outside any of the frustum planes, so an attempt is made to draw the object. The object is outside the frustum even though its bounding volume is not. The renderer processes the object and determines that no part of it will be drawn on the screen. In (b), part of the object is inside the frustum, so the renderer will draw that portion. In (c), the object and its bounding volume are outside the frustum, but because the bounding volume was not outside at least one of the frustum planes, the object is sent to the renderer and it is determined that no part of it will be drawn on the screen. In (d), the bounding volume is outside the right plane of the frustum, so the object is outside and no attempt is made to draw it.

coordinate system you do the back-face culling. However, vertex shader programs require you to transform the vertex positions from model-space coordinates to clip-space coordinates for the purpose of clipping, so it is natural to do the back-face culling in these same coordinates. The transformation of the triangle vertices from model space to view space produces points $\mathbf{V}_i = (r_i, u_i, d_i, 1)$ for $0 \le i \le 2$. A triangle normal vector is

$$\mathbf{N} = (\mathbf{V}_1 - \mathbf{V}_0) \times (\mathbf{V}_2 - \mathbf{V}_0)$$

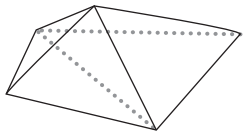


Figure 2.23 Object with front-facing and back-facing triangles indicated.

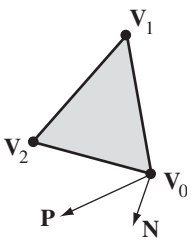


Figure 2.24 A triangle that is front facing to the observer. Because the camera coordinate system is left-handed, the sign test for the dot product of vectors is the opposite of what you are used to.

The eye point in view coordinates is $\mathbf{P} = (0, 0, 0, 1)$. Figure 2.24 shows the situation when the triangle is deemed visible to the observer. The vector $\mathbf{P} - \mathbf{V}_0 = (-r_0, -u_0, -d_0, 0)$ must form an acute angle with the normal vector $\mathbf{N}$. The test for the triangle to be front facing is

$$0 < (\mathbf{P} - \mathbf{V}_0) \cdot \mathbf{N} = \det \begin{bmatrix} -r_0 & r_1 - r_0 & r_2 - r_0 \\ -u_0 & u_1 - u_0 & u_2 - u_0 \\ -d_0 & d_1 - d_0 & d_2 - d_0 \end{bmatrix}$$

Define the homogeneous matrix

$$M = \begin{bmatrix} r_0 & r_1 & r_2 & 0 \\ u_0 & u_1 & u_2 & 0 \\ d_0 & d_1 & d_2 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The first three columns of the matrix are the triangle vertices and the last column of the matrix is the eye point, all listed in view coordinates. The determinant of the matrix is computed as follows, using a cofactor expansion in the last column.

$$\det(M) = -\det\begin{bmatrix} r_0 & r_1 & r_2 \\ u_0 & u_1 & u_2 \\ d_0 & d_1 & d_2 \end{bmatrix}$$ Cofactor expansion is by last column.

$$= -\det\begin{bmatrix} r_0 & r_1 - r_0 & r_2 - r_0 \\ u_0 & u_1 - u_0 & u_2 - u_0 \\ d_0 & d_1 - d_0 & d_2 - d_0 \end{bmatrix}$$ Subtracting columns preserves determinants.

$$= \det\begin{bmatrix} -r_0 & r_1 - r_0 & r_2 - r_0 \\ -u_0 & u_1 - u_0 & u_2 - u_0 \\ -d_0 & d_1 - d_0 & d_2 - d_0 \end{bmatrix}$$ Changing column sign reverses determinant sign.

$$= (\mathbf{P} - \mathbf{V}_0) \cdot \mathbf{N}$$

Thus, the triangle is visible when $\det(M) > 0$.

Multiplying M by the projection matrix of Equation (2.68), we have

$$H_{\text{proj}}M = \begin{bmatrix} r'_0 & r'_1 & r'_2 & 0 \\ u'_0 & u'_1 & u'_2 & 0 \\ d'_0 & d'_1 & d'_2 & -\frac{d_{\max}d_{\min}}{d_{\max}-d_{\min}} \\ w'_0 & w'_1 & w'_2 & 0 \end{bmatrix}$$

Using a cofactor expansion on the last column, we may compute the determinant of this matrix:

$$\det\left(H_{\text{proj}}M\right) = -\frac{d_{\max}d_{\min}}{d_{\max} - d_{\min}}\begin{bmatrix} r'_0 & r'_1 & r'_2 \\ u'_0 & u'_1 & u'_2 \\ w'_0 & w'_1 & w'_2 \end{bmatrix}$$

A front-facing triangle occurs when $\det(M) > 0$, so equivalently it occurs when $\det(H_{\text{proj}}M) = \det(H_{\text{proj}})\det(M) < 0$. That is, the triangle is front facing when

$$\det\begin{bmatrix} r'_0 & r'_1 & r'_2 \\ u'_0 & u'_1 & u'_2 \\ w'_0 & w'_1 & w'_2 \end{bmatrix} > 0$$

This expression is what the Wild Magic software renderer implements, and is found in the file `Wm4SoftDrawElements.cpp`, function `SoftRenderer::DrawTriMesh`.

2.4.3 Clipping to the View Frustum

Clipping is the process by which the front-facing triangles of an object in the world are intersected with the view frustum planes. A triangle either is completely inside the frustum (no clipping necessary), is completely outside the frustum (triangle is

culled), or intersects at least one frustum plane (needs clipping). In the last case the portion of the triangle that lies on the frustum side of the clipping plane must be calculated. That portion is either a triangle itself or a quadrilateral.

Plane-at-a-Time Clipping

One possibility for a simple clipping algorithm is to clip the triangle against a frustum plane. If the portion inside the frustum is a triangle, process that triangle against the next frustum plane. If the portion inside the frustum is a quadrilateral, split it into two triangles and process both against the next frustum plane. After all clipping planes are processed, the renderer has a list of triangles that are completely inside the view frustum. The pseudocode for this process is shown next.

```
set<Triangle> input, output;
input.Insert(initialTriangle);
for each frustum plane do
{
    for each triangle in input do
    {
        set<Triangle> inside = Split(triangle,plane);
        if (inside.Quantity() == 2)
        {
            output.Insert(inside.Element[0]);
            output.Insert(inside.Element[1]);
        }
        else if (inside.Quantity() == 1)
        {
            output.Insert(inside.Element[0]);
        }
        else
        {
            // Inside is empty, triangle is culled.
        }
        input.Remove(triangle);
    }
    input = output;
}

for each triangle in output do
{
    // Draw the triangle.
}
```

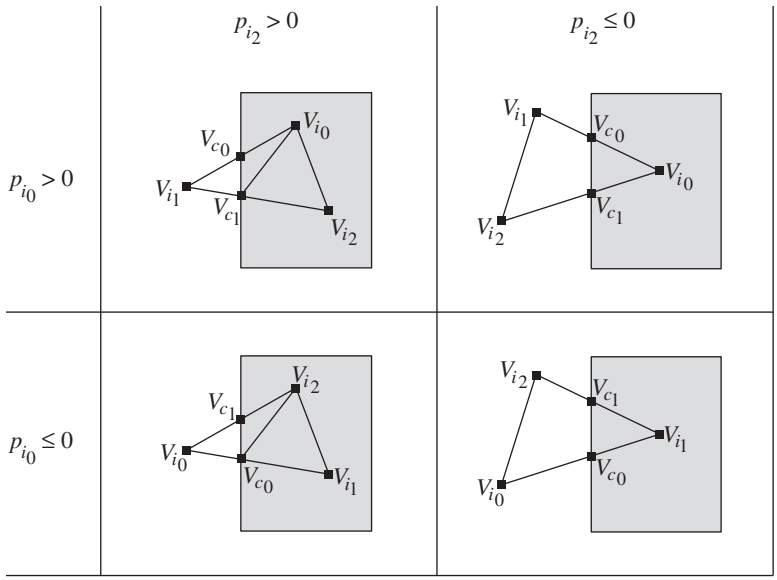


Figure 2.25 Four configurations for triangle splitting. Only the triangles in the shaded region are important, so the quadrilaterals outside are not split. The subscript c indicates clip vertices.

The splitting of a triangle by a frustum plane is accomplished by computing the intersection of the triangle edges with the plane. The three vertices of the triangle are tested for inclusion in the frustum. If the frustum plane is $\mathbf{N} \cdot \mathbf{X} = d$ and if the vertices of the triangle are $\mathbf{V}_i$ for $0 \le i \le 2$, then the edge with endpoints $\mathbf{V}_{i_0}$ and $\mathbf{V}_{i_1}$ intersects the plane if $p_{i_0} p_{i_1} < 0$, where $p_i = \mathbf{N} \cdot \mathbf{V}_i - d$ for $0 \le i \le 2$. This simply states that one vertex is on the positive side of the plane and one vertex is on the negative side of the plane. The point of intersection, called a *clip vertex*, is

$$\mathbf{V}_{\text{clip}} = \mathbf{V}_{i_0} + \frac{p_{i_0}}{p_{i_0} - p_{i_1}} \left(\mathbf{V}_{i_1} - \mathbf{V}_{i_0} \right) \tag{2.75}$$

Figure 2.25 illustrates the possible configurations for clipping of a triangle against a plane. The vertices $\mathbf{V}_{i_0}$, $\mathbf{V}_{i_1}$, and $\mathbf{V}_{i_2}$ are assumed to be in counterclockwise order.

The pseudocode for clipping a single triangle against a plane is given next. After splitting, the new triangles have vertices that are in counterclockwise order.

```
void ClipConfiguration (pi0,pi1,pi2,Vi0,Vi1,Vi2)
{
    // assert: pi0*pi1 < 0
```

```
    Vc0 = Vi0+(pi0/(pi0-pi1))*(Vi1-Vi0);
    if (pi0 > 0)
    {
        if (pi2 > 0)  // Figure 2.25, top left
        {
            Vc1 = Vi1+(pi1/(pi1-pi2))*(Vi2-Vi1);
            add triangle <Vc0,Vc1,Vi0> to triangle list;
            add triangle <Vc1,Vi2,Vi0> to triangle list;
        }
        else          // Figure 2.25, top right
        {
            Vc1 = Vi0+(pi0/(pi0-pi2))*(Vi2-Vi0);
            add triangle <Vc0,Vc1,Vi0> to triangle list;
        }
    }
    else
    {
        if (pi2 > 0)  // Figure 2.25, bottom left
        {
            Vc1 = Vi0+(pi0/(pi0-pi2))*(Vi2-Vi0);
            add triangle <Vc0,Vi1,Vi2> to triangle list;
            add triangle <Vc0,Vi2,Vc1> to triangle list;
        }
        else          // Figure 2.25, bottom right
        {
            Vc1 = Vi1+(pi1/(pi1-pi2))*(Vi2-Vi1);
            add triangle <Vc0,Vi1,Vc1> to triangle list;
        }
    }
}

void ClipTriangle ()
{
    remove triangle <V0,V1,V2> from triangle list;

    p0 = Dot(N,V0)-d;
    p1 = Dot(N,V1)-d;
    p2 = Dot(N,V2)-d;

    if (p0*p1 < 0)
    {
        // Triangle needs splitting along edge <V0,V1>.
        ClipConfiguration(p0,p1,p2,V0,V1,V2);
    }
    else if (p0*p2 < 0)
```

```
    {
        // Triangle needs splitting along edge <V0,V2>.
        ClipConfiguration(p2,p0,p1,V2,V0,V1);
    }
    else if (p1*p2 < 0)
    {
        // Triangle needs splitting along edge <V1,V2>.
        ClipConfiguration(p1,p2,p0,V1,V2,V0);
    }
    else if (p0 > 0 || p1 > 0 || p2 > 0)
    {
        // Triangle is completely inside frustum.
        add triangle <V0,V1,V2> to triangle list;
    }
}
```

To avoid copying vertices, the triangle representation can store pointers to vertices in a vertex pool, adding clip vertices as needed.

Polygon-of-Intersection Clipping

The plane-at-a-time clipping algorithm keeps track of a set of triangles that must be clipped against frustum planes. Processing only triangles leads to simple data structures and algorithms. The drawback is that the number of triangles can be larger than is really necessary.

An alternate method for clipping computes the convex polygon of intersection of the triangle with the frustum. After clipping, a triangle fan is generated for the polygon and these triangles are drawn. The number of triangles in this approach is smaller than or equal to the number produced by the plane-at-a-time clipping algorithm. An illustration of this is provided by the sequence of images shown in Figures 2.26 through 2.30. For the sake of simplicity, the example is shown in two dimensions with the frustum drawn as a rectangle. Figure 2.26 shows a triangle intersecting a frustum. The convex polygon of intersection has seven vertices. The triangle fan is drawn, indicating that the renderer will draw five triangles.

Let us clip the triangle against the four frustum planes one at a time. Figure 2.27 shows the triangle clipped against the bottom frustum plane. Two clip vertices are generated. The portion of the triangle on the frustum side of the bottom plane is a quadrilateral, so it is split into two triangles T_1 and T_2.

Figure 2.28 shows the triangles clipped against the top frustum plane. The triangle T_1 is clipped, generates two clip vertices, and is split into two triangles, T_3 and T_4. The triangle T_2 is clipped, generates two clip vertices, and is split into two triangles, T_5 and T_6.

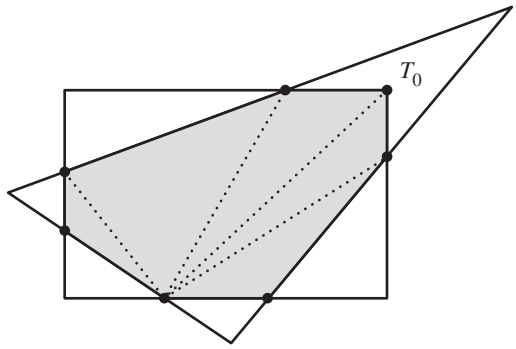


Figure 2.26 A triangle T_0 intersecting a frustum in multiple faces. The convex polygon of intersection has seven vertices and is represented by a triangle fan with five triangles.

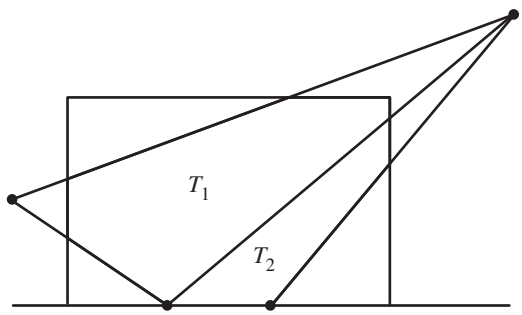


Figure 2.27 The triangle is clipped against the bottom frustum plane.

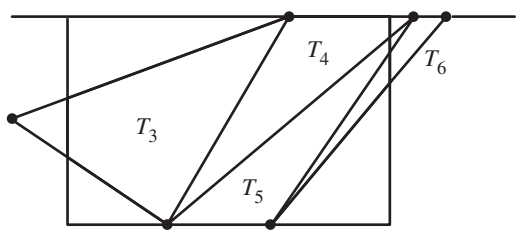


Figure 2.28 The triangles are clipped against the top frustum plane.

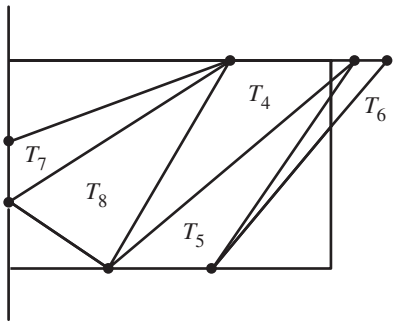


Figure 2.29 The triangles are clipped against the left frustum plane.

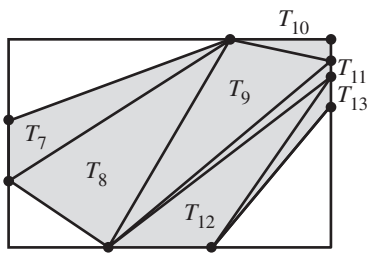


Figure 2.30 The triangles are clipped against the right frustum plane.

Figure 2.29 shows the triangles clipped against the left frustum plane. In this case, only triangle T_3 intersects the left frustum plane. It generates two clip vertices and the quadrilateral inside the frustum is split into two triangles, T_7 and T_8.

Finally, Figure 2.30 shows the triangles clipped against the right frustum plane. Triangle T_4 is clipped and split into triangles T_9 and T_{10}. Triangle T_5 is clipped and split into triangles T_{11} and T_{12}. Triangle T_6 is clipped, producing a single triangle T_{13}. The end result is a collection of nine vertices and seven triangles in contrast to the polygon-of-intersection clipping algorithm, which produced seven vertices and five triangles.

At first glance, the polygon-of-intersection clipping algorithm is attractive because it tends to generate fewer triangles than the plane-at-a-time clipping algorithm. However, the example here is slightly misleading because the triangle is very large compared to the frustum size. In a realistic application, the observer is positioned so that triangles are generally small compared to the frustum size, so you would expect a

triangle to be clipped by one frustum plane (triangle intersects a face of the frustum), by two frustum planes (triangle intersects near an edge of the frustum), or three frustum planes (triangle intersects near a corner of the frustum). In these cases, either clipping method should perform equally well.

EXERCISE 2.14 The Wild Magic software renderer implements the polygon-of-intersection clipping algorithm. Modify the renderer to use the plane-at-a-time clipping algorithm. Devise an experiment to test the performance of the two clipping algorithms and compare the results. ■

2.5 RASTERIZING

Rasterization is the process of taking a geometric entity in window space and selecting those pixels to be drawn that correspond to the entity. The standard objects that most engines rasterize are line segments and triangles, but rasterization of circles and ellipses is also discussed here. You might have a situation where you want to rasterize such objects to a texture and then use the texture for one of your 3D objects. The constructions contained in this section all assume integer arithmetic since the main goal is to rasterize as fast as possible. Floating-point arithmetic tends to be more expensive than integer arithmetic.

EXERCISE 2.15 This is a large project. The Wild Magic software renderer uses floating-point arithmetic for its rasterization; that is, the renderer is not optimized for speed (it was designed to illustrate concepts). If you feel adventuresome, reimplement the rasterizing code to use integer arithmetic. This code is found in files `Wm4SoftDrawElements.cpp` and `Wm4SoftEdgeBuffers.cpp`. ■

2.5.1 LINE SEGMENTS

Given two screen points (x_0, y_0) and (x_1, y_1), a line segment must be drawn that connects them. Since the pixels form a discrete set, decisions must be made about which pixels to draw in order to obtain the "best" line segment, which Figure 2.31 illustrates. If $x_1 = x_0$ (vertical segment) or $y_1 = y_0$ (horizontal segment), it is clear which pixels to draw. And if $|x_1 - x_0| = |y_1 - y_0|$, the segment is diagonal and it is clear which pixels to draw. But for the other cases, it is not immediately apparent which pixels to draw.

The algorithm should depend on the magnitude of the slope. If the magnitude is larger than 1, each row that the segment intersects should have a pixel drawn. If the magnitude is smaller than 1, each column that the segment intersects should have a pixel drawn. Figure 2.32 illustrates the cases. The two blocks of pixels in (a) illustrate the possibilities for drawing pixels for a line with a slope whose magnitude is larger than 1. The case in (a) draws one pixel per column. The case in (b) draws one pixel

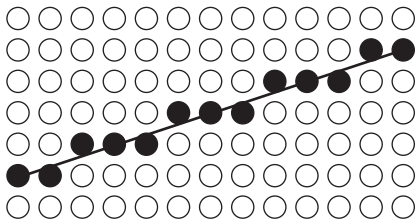


Figure 2.31 Pixels that form the best line segment between two points.

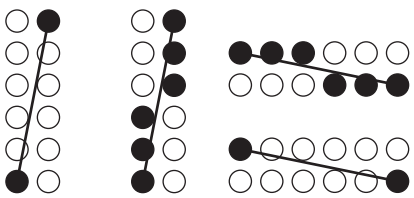


Figure 2.32 Pixel selection based on slope.

per row, the correct decision. The two blocks of pixels in (b) illustrate the possibilities for drawing pixels for a line with a slope whose magnitude is less than 1. The bottom case draws one pixel per row. The top case draws one pixel per column, the correct decision.

The process of pixel selection, called Bresenham's algorithm [Bre65], uses an integer decision variable that is updated for each increment in the appropriate input variable. The sign of the decision variable is used to select the correct pixel to draw at each step. Define $dx = x_1 - x_0$ and $dy = y_1 - y_0$. For the sake of argument, assume that $dx > 0$ and $dy \neq 0$. The decision variable is d_i, and its value is determined by the pixel (x_i, y_i) that was drawn at the previous step. Figure 2.33 shows two values s_i and t_i, the fractional lengths of the line segment connecting two vertical pixels. The value of s_i is determined by $s_i = (y_0 - y_i) + (dy/dx)(x_i + 1 - x_0)$ and $s_i + t_i = 1$. The decision variable is $d_i = dx(s_i - t_i)$. From the figure it can be seen that

- If $d_i \geq 0$, then the line is closer to the pixel at $(x_i + 1, y_i + 1)$, so draw that pixel.
- If $d_i < 0$, then the line is closer to the pixel at $(x_i + 1, y_i)$, so draw that pixel.