This is a draft of a second edition of the book, intended for use by students in Math 155AB at UCSD during Winter-Spring 2019. This draft is available for use for your personal study or projects. Reports of errata or other corrections will be greatly appreciated.

All rights reserved. Copyright, Sam Buss, 2019.
To my family
Teresa, Stephanie, and Ian
# Contents

## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>xi</td>
</tr>
<tr>
<td><strong>I Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>I.1 Points, lines and triangles</td>
<td>2</td>
</tr>
<tr>
<td>I.1.1 Rectangular arrays of pixels</td>
<td>2</td>
</tr>
<tr>
<td>I.1.2 Vector graphics</td>
<td>4</td>
</tr>
<tr>
<td>I.1.3 Triangles</td>
<td>5</td>
</tr>
<tr>
<td>I.2 Coordinate systems</td>
<td>5</td>
</tr>
<tr>
<td>I.3 Points, lines and triangles in OpenGL</td>
<td>7</td>
</tr>
<tr>
<td>I.3.1 Loading vertices into a VAO and VBO</td>
<td>7</td>
</tr>
<tr>
<td>I.3.2 Drawing points and lines</td>
<td>11</td>
</tr>
<tr>
<td>I.3.3 Drawing triangles</td>
<td>13</td>
</tr>
<tr>
<td>I.3.4 Rendering with element arrays</td>
<td>16</td>
</tr>
<tr>
<td>I.3.5 Different colors per vertex</td>
<td>17</td>
</tr>
<tr>
<td>I.3.6 Face orientation and culling</td>
<td>19</td>
</tr>
<tr>
<td>I.3.7 Wireframe triangles</td>
<td>21</td>
</tr>
<tr>
<td>I.4 Vertex shaders and fragment shaders</td>
<td>21</td>
</tr>
<tr>
<td>I.4.1 Very simple vertex and fragment shader example</td>
<td>22</td>
</tr>
<tr>
<td>I.4.2 A fragment shader modifying the color</td>
<td>24</td>
</tr>
<tr>
<td>I.4.3 Flat shading</td>
<td>24</td>
</tr>
<tr>
<td>I.4.4 Hidden surfaces</td>
<td>25</td>
</tr>
<tr>
<td>I.5 Animation with double buffering</td>
<td>27</td>
</tr>
<tr>
<td>I.6 Antialiasing with supersampling</td>
<td>29</td>
</tr>
<tr>
<td>I.7 Additional exercises</td>
<td>29</td>
</tr>
<tr>
<td><strong>II Transformations and Viewing</strong></td>
<td>31</td>
</tr>
<tr>
<td>II.1 Transformations in 2-space</td>
<td>34</td>
</tr>
<tr>
<td>II.1.1 Basic definitions</td>
<td>35</td>
</tr>
<tr>
<td>II.1.2 Matrix representation of linear transformations</td>
<td>38</td>
</tr>
<tr>
<td>II.1.3 Rigid transformations and rotations</td>
<td>42</td>
</tr>
<tr>
<td>II.1.4 Homogeneous coordinates</td>
<td>46</td>
</tr>
<tr>
<td>II.1.5 Matrix representation of an affine transformation</td>
<td>47</td>
</tr>
<tr>
<td>VIII.9</td>
<td>Degree elevation</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>VIII.10</td>
<td>Bézier surface patches</td>
</tr>
<tr>
<td>VIII.10.1</td>
<td>Basic properties of Bézier patches</td>
</tr>
<tr>
<td>VIII.10.2</td>
<td>Joining Bézier patches</td>
</tr>
<tr>
<td>VIII.11</td>
<td>Bézier curves and surfaces in OpenGL</td>
</tr>
<tr>
<td>VIII.11.1</td>
<td>Bézier curves</td>
</tr>
<tr>
<td>VIII.11.2</td>
<td>Bézier patches</td>
</tr>
<tr>
<td>VIII.12</td>
<td>Rational Bézier curves</td>
</tr>
<tr>
<td>VIII.13</td>
<td>Conic sections with rational Bézier curves</td>
</tr>
<tr>
<td>VIII.14</td>
<td>Surface of revolution example</td>
</tr>
<tr>
<td>VIII.15</td>
<td>Interpolating with Bézier curves</td>
</tr>
<tr>
<td>VIII.15.1</td>
<td>Catmull-Rom splines</td>
</tr>
<tr>
<td>VIII.15.2</td>
<td>Bessel-Overhauser splines</td>
</tr>
<tr>
<td>VIII.15.3</td>
<td>Tension-continuity-bias splines</td>
</tr>
<tr>
<td>VIII.16</td>
<td>Interpolating with Bézier surfaces</td>
</tr>
</tbody>
</table>

IX  B-Splines  
IX.1 Uniform B-splines of degree three  
IX.2 Non-uniform B-splines  
IX.3 Examples of non-uniform B-splines  
IX.4 Properties of non-uniform B-splines  
IX.5 The de Boor algorithm  
IX.6 Blossoms  
IX.7 Derivatives and smoothness of B-spline curves  
IX.8 Knot insertion  
IX.9 Bézier and B-spline curves  
IX.10 Degree elevation  
IX.11 Rational B-splines and NURBS  
IX.12 B-splines and NURBS surfaces in OpenGL  
IX.13 Interpolating with B-splines  

X  Ray Tracing  
X.1 Basic ray tracing  
X.1.1 Local lighting and reflection rays  
X.1.2 Transmission rays  
X.1.3 Putting it all together  
X.2 Advanced ray tracing techniques  
X.2.1 Distributed ray tracing  
X.2.2 Backwards ray tracing  
X.3 Special effects without ray tracing  

XI  Intersection Testing  
XI.1 Fast intersections with rays  
XI.1.1 Ray versus sphere intersections  
XI.1.2 Ray versus plane intersections  
XI.1.3 Ray versus triangle intersections  

viii
A.3.1 Matrices and vector products in $\mathbb{R}^3$ .......................... 444
A.3.2 Determinants, inverses, and adjoints ............................... 445
A.3.3 Linear subspaces ............................................. 446
A.4 Multivariable calculus ........................................... 447
A.4.1 Multivariable functions ........................................ 447
A.4.2 Vector-valued functions ........................................ 449
A.4.3 Multivariable vector-valued functions ......................... 450
B RayTrace Software Package ........................................ 451
B.1 Introduction to the ray tracing package ............................. 451
B.2 The high level ray tracing routines ................................ 453
B.3 The RayTrace API .................................................. 456
  B.3.1 Specifying lights ............................................ 456
  B.3.2 Defining the viewport and camera positions ................ 458
  B.3.3 Working with the pixel array ................................ 460
  B.3.4 Defining materials .......................................... 460
  B.3.5 Defining viewable objects ................................... 461
  B.3.6 Viewable spheres ............................................ 462
  B.3.7 Viewable triangles and parallelograms ..................... 463
  B.3.8 Viewable ellipsoids ......................................... 464
  B.3.9 Viewable cylinders ......................................... 465
  B.3.10 Viewable cones ............................................ 467
  B.3.11 Viewable parallelepipeds .................................. 468
  B.3.12 Viewable tori ............................................. 469
  B.3.13 Viewable Bézier patches ................................. 470
  B.3.14 Texture maps ............................................. 473
Bibliography ..................................................................... 477
Index .................................................................................. 487
Preface

Preface to the Second Edition

TO BE WRITTEN

Preface to the First Edition

Computer graphics has undergone phenomenal growth in the past couple decades, progressing from simple 2D graphics to complex, high-quality, three dimensional environments. In entertainment, computer graphics is used extensively in movies and computer games. More and more animated movies are being made entirely with computers. Even non-animated movies depend heavily on computer graphics to develop special effects: witness, for instance, the success of the Star Wars movies beginning already in the mid-1970’s. The development of computer graphics in personal computers and home game consoles has also been phenomenal in recent years, with low-cost systems now able to display millions of polygons per second.

There are also significant uses of computer graphics in non-entertainment applications. For example, virtual reality systems are often used for training applications. Computer graphics is an indispensable tool for scientific visualization and for computer aided design. We need good methods for displaying large data sets in a comprehensible manner, and for showing the results of large-scale scientific simulations.

The art and science of computer graphics has been developing since the advent of computers and started in earnest in the early 1960’s. Since then, it has developed into a rich, deep, and coherent field. The aim of this book is to present the mathematical foundations of computer graphics, along with a practical introduction to programming computer graphics using OpenGL. We believe that understanding the mathematical basis is important for any advanced use of computer graphics. For this reason, this book attempts to thoroughly cover the mathematics underlying computer graphics. The principle guiding the selection of topics for this book has been to choose topics that are of practical significance for computer graphics practitioners, in particular for software developers. Our hope is that this book will serve as a comprehensive introduction to the standard tools used in computer graphics and especially to the mathematical theory behind these tools.
About this book  The plan for this book has been shaped by my personal experiences as an academic mathematician and by my having participated in various applied computer projects, including projects in computer games and virtual reality. This book was first started while I was teaching a mathematics class at UCSD on Computer Graphics and Geometry. That course was structured as an introduction to programming 3D graphics in OpenGL, and to the mathematical foundations of computer graphics. While teaching that course, I became convinced of the need for a book that brings together the mathematical theory underlying computer graphics in an introductory and unified setting.

The other motivation for writing this book has been my involvement in several virtual reality and computer game projects. Many of the topics included in this book are present mainly because I have found them useful in computer game applications. Modern day computer games and virtual reality applications are demanding software projects: these applications require software capable of displaying convincing three dimensional environments. Generally, the software must keep track of the motion of multiple objects; maintain information about the lighting, colors, and textures of many objects; and display them on the screen at 30 or 60 frames per second. In addition, considerable artistic and creative skills are needed to make a worthwhile three dimensional environment. Not surprisingly, this requires sophisticated software development by large teams of programmers, artists, and designers.

Perhaps it is a little more surprising that 3D computer graphics requires a good deal of mathematics. This is however the case. Furthermore, the mathematics tends to be elegant and interdisciplinary. The mathematics needed in computer graphics brings together constructions and methods from several areas of mathematics, including geometry, calculus, linear algebra, numerical analysis, abstract algebra, data structures, and algorithms. In fact, computer graphics is arguably the best example of a practical area where so much mathematics combines so elegantly.

This book includes a blend of applied and theoretical topics. On the more applied side, we recommend the use of OpenGL, a readily available, free, cross-platform programming environment for three dimensional graphics. We have included C and C++ code for OpenGL programs which can be freely downloaded from the internet, and we discuss how OpenGL implements many of the mathematical concepts discussed in this book. We also describe a ray tracer software package: this software can also be downloaded from the internet. On the theoretical side, this book stresses the mathematical foundations of computer graphics, more so than any other text we are aware of. We strongly believe that knowing the mathematical foundations of computer graphics is important for being able to properly use tools such as OpenGL or Direct3D, or, to a lesser extent, CAD programs.

The mathematical topics in this book are chosen because of their importance and relevance to graphics. However, we have not hesitated to introduce more abstract concepts when they are crucial to computer graphics, for instance the projective geometry interpretation of homogeneous coordinates. In our opinion,
a good knowledge of mathematics is invaluable if you want to properly use the techniques of computer graphics software, and even more important if you want to develop new or innovative uses of computer graphics.
Using this book and OpenGL

This book is intended for use as a textbook, as a source for self-study, or as a reference. It is strongly recommended that you try running the programs supplied with the book, and that you write some OpenGL programs of your own.

OpenGL is a platform independent API (application programming interface) for rendering 3D graphics. A big advantage of using OpenGL is that it is a widely supported industry standard. Other 3D environments, notably Microsoft’s Direct3D and Apple’s Metal, have similar capabilities; however, these are specific to the Windows and Macintosh operating systems.

This book is intended to be supplemented with other resources to help you learn OpenGL. The book contains code snippets, of C++ programs using OpenGL features. It also discusses the architecture of OpenGL programs, including how vertex and fragment shaders work within an OpenGL program. We also provide some substantial shader programs, including a shader that implements Phong lighting.

The textbook’s web page at http://math.ucsd.edu/~sbuss/MathCG2/

contains a number of sample OpenGL programs, starting with SimpleDrawModern which is essentially the simplest possible non-trivial modern-style OpenGL program, and working up to programs with sophisticated shaders for Phong light, and manage multiple shaders. It also has a complete Ray Tracing package available. These programs all use the “modern” OpenGL programming style, not the older “immediate mode” OpenGL. All the OpenGL programs include full source code and are accompanied by web pages that explain the code features line-by-line. OpenGL programming can be complex, but it is hoped that these will give you straightforward and accessible introduction to OpenGL programming.

The textbook’s software may be used without any restriction except that its use in commercial products or any kind of substantial project must be acknowledged.

There are many other sources available to learn OpenGL. One very nice source is the https://learnopengl.com web pages created by Joey De Vries. This web site also has the option to download a complete copy of the learnopengl.com tutorials as a PDF e-book. (As of 2019, this is free,
but accepts donations.) The *OpenGL SuperBible* is a book-length tutorial introduction to OpenGL. For more official sources, there are the *OpenGL Programming Guide* and the *OpenGL Shading Language Book* written by some of the OpenGL developers. These last two books can be difficult to read for the beginner however. Whatever source you use, it should cover OpenGL version 3.3, or later.

Finally, it is scarcely necessary to mention it, but the internet is a wonderful resource: there is plenty of great information online about OpenGL! In fact, it is recommended that as you read the book, you do an internet search for every OpenGL command encountered.

**Outline of the book**

The chapters are arranged more-or-less in the order the material might be covered in a course. However, it is not necessary to cover the material in order. In particular, the later chapters can be read largely independently, with the exception that Chapter IX depends on Chapter VIII.


*Chapter III. A new chapter.* There will be a new chapter — DESCRIPTION COMING SOON. —

*Chapter IV. Lighting, illumination, and shading.* The Phong lighting model. Ambient, diffuse, and specular lighting. Lights and material properties in OpenGL. The Cook-Torrance model.

*Chapter V. Averaging and interpolation.* Linear interpolation. Barycentric coordinates. Bilinear interpolation. Convexity. Hyperbolic interpolation. Spherical linear interpolation. This is a more mathematical chapter with a lot of tools that are used elsewhere in the book: you may wish to skip much of this on the first reading, and come back to it as needed.


*Chapter VII. Color.* Color perception. Additive and subtractive colors. RGB and HSL.


Chapter IX. B-splines. Uniform and non-uniform B-splines and their properties. B-splines in OpenGL. The de Boor algorithm. Blossoms. Smoothness properties. NURBS and conic sections. Knot insertion. Relationship with Bézier curves. Interpolation with spline curves. This chapter has a mix of introductory topics and more specialized topics. We include all proofs, but recommend that many of the proofs be skipped on the first reading.


Chapter XI. Intersection testing. Testing rays for intersections with spheres, planes, triangles, polytopes, and other surfaces. Bounding volumes and hierarchical pruning.


Appendix A. Mathematics background. A review of topics from vectors, matrices, linear algebra, and calculus.

Appendix B. RayTrace software package. Description of a ray tracing software package. The software is freely downloadable.

There are exercises scattered throughout the book, especially in the more introductory chapters. These are often supplied with hints and they should not be terribly difficult. It is highly recommended that you do the exercises to master the material. A few sections in the book, as well as some of the theorems, proofs, and exercises, are labeled with a star symbol (⋆). This indicates that the material is optional and/or less important, and can be safely skipped without affecting your understanding of the rest of the book.

Theorems, lemmas, figures, and exercises are numbered separately for each chapter. For instance, the third figure of Chapter II will be referred to as just “Figure II.3”.

For the instructor

This book is intended for use with advanced junior or senior level undergraduate courses or part of introductory graduate level courses. It is based in large part on my teaching computer graphics courses mostly at the upper division level. In a two quarter, undergraduate course, I cover most of the material in the book, more-or-less in the order presented in the book. Some of the more advanced topics would be skipped however; most notably Cook-Torrance
lighting and some of the material on Bézier and B-spline curves and patches are best omitted from an undergraduate course. I also omit the more difficult proofs from an undergraduate course, especially the a number of the proofs for B-splines.

It is certainly recommended that students using this book get programming assignments using OpenGL. Although this book covers a lot of OpenGL material in outline form, students will need to have an additional source for learning the details of programming in OpenGL. Programming prerequisites include some experience in a C-like language such as C, C++ or Java. The first quarters of my own courses have included programming assignments first on two dimensional graphing, second on 3D transformations based on the solar system exercise on page 69, third on polygonal modeling (often asking students to do something creative based on their initials, plus asking them to dynamically render surfaces of rotations or another circularly symmetric object such as the torus in Figure I.11 on page 20), fourth on adding materials and lighting to a scene, fifth on textures, and then ending with an open ended assignment where students choose a project on their own. The second quarter of the course has included assignments on modeling objects with Bézier patches (Blinn’s article [14] on how to construct the Utah teapot is sometimes used to help with this), on writing a program that draws Catmull-Rom and Overhauser spline curves which interpolate points picked with the mouse, on writing vertex or fragment shaders, on using the ray tracing software supplied with this book, on implementing some aspect of distributed ray tracing, and then ending with another final project of their choosing. Past course materials can be found on the web from my course web pages at http://math.ucsd.edu/~sbuss/CourseWeb.

Acknowledgements

Very little of the material in this book is original. The aspects that are original mostly concern matters of organization and presentation: in a number of places, I have tried to present new, simpler proofs than what was known before. In many cases, material is presented without attribution or credit, but in most cases this material is due to others. I have included references for items that I learned by consulting the original literature, and for topics for which it was easy to ascertain what the original source was. However, the book does not try to be comprehensive in assigning credit.

I learned computer graphics from several sources. First, I worked on a computer graphics project with several people at SAIC including Tom Yonkman and my wife, Teresa Buss. Subsequently, I have worked over a period of many years on computer games applications at Angel Studios. At Angel Studios, I benefited greatly, and learned an immense amount, from working with Steve Rotenberg, Brad Hunt, Dave Etherton, Santi Bacerra, Nathan Brown, Ted Carson, Jeff Roorda, Daniel Blumenthal, and others. I am particularly indebted to Steve Rotenberg who has been my “guru” for advanced topics and current research in computer graphics.

I have taught computer graphics courses several times at UCSD, using at
various times the textbooks by Watt and Watt [118], Watt [117], and Hill [65].

This book is written partly from notes developed while teaching these classes.

I am greatly indebted to Frank Chang and Malachi Pust for a thorough proofreading of an early draft of this book. In addition, I’d like to thank Michael Bailey, Stephanie Buss (my daughter), Chris Calabro, Joseph Chow, Daniel Curtis, Tamsen Dunn, Rosalie Iemhoff, Cyrus Jam, Jin-Su Kim, Vivek Manpuria, Jason McAuliffe, Jong-Won Oh, Horng Bin Ou, Chris Pollett, John Rapp, Don Quach, Daryl Sterling, Aubin Whitley, and anonymous referees for corrections to preliminary drafts of this book and Tak Chu, Craig Donner, Jason Eng, Igor Kaplounenko, Alex Kuhungowski, Allan Lam, Peter Olcott, Nevin Shenoy, Mara Silva, Abbie Whynot, and George Yue for corrections incorporated into the second printing of the first edition.

Students from my Math 155AB courses in the winter and spring quarters of 2019 contributed corrections and suggestions for preliminary drafts of the second edition of the book. They included Dylan Greicius, Christopher Rocha, Mihai Ruber, Arshia Sehgal, Ronan Sullivan, Yujie Wang, Wenjie Zhu, and more names to come I hope.

Finally, I am indebted to the teaching assistants for my courses, including Jonathan Conder, Srivastava Shambhukha Kuchibhotla, Janine LoBue, Jefferson Ng, Nicholas Sieger, Xiudi Tang, and Jingwen Wang.

The figures in this book were prepared with several software systems. The majority of the figures were created using T. van Zandt’s \texttt{pstricks} macro package for \LaTeX{}. Some of the figures were created with a modified version of C. Geuzaine’s program \texttt{GL2PS} for converting OpenGL images into postscript files. A few figures were created from screen dump bitmaps and converted to postscript images with Adobe Photoshop.
Chapter I

Introduction

This is a preliminary draft of a second edition of the book 3D Computer Graphics: A Mathematical Introduction with OpenGL. So please read it cautiously and critically! Corrections are appreciated. Draft A.2.d

Author: Sam Buss, sbuss@ucsd.edu

This introductory chapter will explain some of the basic concepts of 3D graphics, with an emphasis on explaining how to get started with simple drawing in OpenGL. We will use the so-called “Modern OpenGL” programming interface. Modern OpenGL is designed to take advantage of the power of graphics processing units (GPU’s). A GPU is a kind of mini-supercomputer; it typically contains hundreds or even thousands of cores. Each core in a GPU is a small computer, able to run small programs and handle small computations. This allows a GPU to act like a powerful parallel computer, able to perform multiple computations at once. GPUs were originally developed for handling graphics computations, for example to run a separate program for each pixel on a computer display. But they have evolved to become much more powerful and capable of handling many applications in diverse fields such as scientific computation, bioinformatics, data science, and machine learning.

A graphics program using OpenGL, is designed to run on the central processing unit (CPU) of a computer; the CPU is used to generate high-level graphics data and small GPU programs called “shaders”. The graphics data and the shaders are uploaded from the CPU into the GPU. The GPU runs these programs on its cores to generate the image which appears on the computer screen. In a typical application rendering a 3D scene, the CPU’s will specify a set of triangles in terms of their vertices. The information about the vertices and triangles is uploaded into the GPU’s memory. The GPU first runs programs called “vertex shaders” that operate independently on each vertex, then OpenGL pieces the vertices into triangles, and finally the GPU runs small programs called “fragment shaders” that operate independently on each pixel in the screen. All this happens every time the screen is rendered, say 30 or 60 times per second!
This introductory chapter will first discuss the conceptual display modes of points, lines and triangles and then how they are described in 3-space using \(x, y, z\) coordinates. The crucial idea is that an arbitrary three dimensional surface can be approximated by a set of triangles. The second part of this chapter describes some simple C++ programs using OpenGL vertex shaders and fragment shaders. It also describes setting colors, orienting triangles, using hidden surface removal, and making animation work with double buffering. The website https://math.ucsd.edu/~sbuss/MathCG2/OpenGLsoft has C++ programs that illustrate how to use OpenGL. The first two, SimpleDrawModern and SimpleAnimModern, give some of the simplest possible examples of complete C++ programs using shaders and OpenGL.

Later chapters will discuss how to use transformations, how to set the viewpoint, how to add lighting and shading, how to add textures, and other topics.

### I.1 Points, lines and triangles

We start by describing three models for graphics display modes: (1) drawing points, (2) drawing lines, and (3) drawing triangles. These three models correspond to different hardware architectures for graphics display. Drawing points corresponds roughly to the model of a graphics image as a rectangular array of pixels. Drawing lines corresponds to vector graphics displays. Drawing triangles corresponds to the methods used by modern graphics systems to display three dimensional images.

#### I.1.1 Rectangular arrays of pixels

The most common low-level model is to treat a graphics image as a rectangular array of pixels, where each pixel can be independently set to a different color and brightness. This is the display model used for LCD displays, CRT’s, projectors, etc.; namely for computer screens, televisions, cell phones, and many movies. If the pixels are small enough, they cannot be individually seen by the human eye, and the image, although composed of points, appears as a single smooth image. This principle is used in art as well, notably in mosaics and even more so in pointillism, where pictures are composed of small patches of solid color, yet appear to form a continuous image when viewed from a sufficient distance.

The model of graphics images as a rectangular array of pixels is only a convenient abstraction, and is not entirely accurate. For instance, on most graphics displays, each pixel actually consists of three separate points: each point generates one of three primary colors, red, blue, and green, and can be independently set to a brightness value. Thus, each pixel is actually formed from three colored dots. With a sufficiently good magnifying glass, you can see the pixel as separate colors. (It is easiest to try this with a low resolution device such as an old-style CRT television: depending on the physical design of the screen, the separate colors may appear in individual dots or in stripes.) The
Figure I.1: A pixel is formed from subregions or subpixels, each of which displays one of three colors. See color plate C.1.

three primary colors of red, green and blue can be blended to generate a wide palette of colors; however, they cannot reproduce all possible visible colors.

A second way in which the rectangular array model is not accurate is that sometimes sub-pixel addressing of images is used. For instance, laser printers and ink jet printers reduce aliasing problems, such as jagged edges on lines and symbols, by micro-positioning toner or ink dots. Some computers use subpixel rendering to display text at a higher resolution than would otherwise be possible by treating each pixel as three independently addressable subpixels. In this way, the device is able to position text at the subpixel level and achieve a higher level of detail and better formed characters.

In this book however, we rarely have occasion to examine issues of subpixels; instead, we will model a pixel as being a single rectangular point that can be set to a desired color and brightness. But there are many occasions when the fact that a computer graphics image is composed of pixels will be important. Section III.1 discusses the Bresenham algorithm for approximating a straight sloping line with pixels. Also, when using texture maps and ray tracing, it is important to avoid the aliasing problems that can arise with sampling a continuous or high-resolution image into a set of pixels.

In principle, any picture can be rendered by directly setting the brightness levels for each pixel in the image. But in practice, this would be difficult and time-consuming. It is far easier to not consider the pixels at all, and to work instead at the higher level of triangle-based modeling. In high-level graphics programming applications, we generally ignore the fact that the graphics image is rendered using a rectangular array of pixels. Most OpenGL programs work by drawing triangles, and let the graphics hardware handles most of the work of translating the results into pixel brightness levels. This builds on top of a variety of sophisticated techniques for drawing triangles on a computer screen as an array of pixels, including methods for shading and smoothing and for applying texture maps.
I.1.2 Vector graphics

Traditional vector graphics renders images as a set of lines. This does not allow drawing surfaces or solid objects; instead it draws two dimensional shapes, graphs of functions, or wireframe images of three dimensional objects. The prototypical example of vector graphics systems are pen plotters: this includes also the “turtle geometry” systems. Pen plotters have a drawing pen which moves over a flat sheet of paper. The commands available include (a) pen up, which lifts the pen up from the surface of the paper, (b) pen down, which lowers the tip of the pen onto the paper, and (c) move-to \((x,y)\), which moves the pen in a straight line from its current position to the point with coordinates \((x,y)\). When the pen is up, it moves without drawing; when the pen is down, it draws as it moves. In addition, there may be commands for switching to a different color pen and commands for moving along curved paths such as circular or elliptical arcs and Bézier curves.

Another example of vector graphics devices are vector graphics display terminals, which traditionally are monochrome monitors that can draw arbitrary lines. On these vector graphics display terminals, the screen is a large expanse of phosphor, and does not have pixels. A old-style traditional oscilloscope is another example of a vector graphics display device. A laser light show is also a kind of vector graphics image.

Vector graphics display terminals have not been commonly for decades, so vector graphics images are usually rendered on pixel-based displays. In pixel-based systems, the screen image is stored as a bitmap, namely, as a table containing all the pixel colors. A vector graphics system instead stores the image as a list of commands, for instance as a list of pen up/down and move commands. Such a list of commands is called a display list.

Since pixel-based graphics hardware is so very prevalent, modern vector graphics images are typically displayed on hardware that is pixel-based. This has the disadvantage that the pixel-based hardware cannot directly draw arbitrary lines or curves and must instead approximate lines and curves with pixels. On
I.2 Coordinate systems

When rendering a geometric object, its position and shape are specified in terms of the positions of its vertices. For instance, a triangle is specified in terms of
the positions of its three vertices. Graphics programming languages, including OpenGL, allow you to set up your own coordinate systems for specifying positions of points; this is done by using a matrix to define a mapping from your coordinate system into the screen coordinates. Chapter II describes how these matrices define linear or affine transformations; they allow you to position points in either 2-space (\( \mathbb{R}^2 \)) or 3-space (\( \mathbb{R}^3 \)) and have OpenGL automatically map the points into the proper location in the graphics image.

In the two dimensional \( xy \)-plane, also called \( \mathbb{R}^2 \), a vertex’s position is set by specifying its \( x \)- and \( y \)-coordinates. The usual convention, see Figure I.3, is that the \( x \)-axis is horizontal and pointing to the right, and the \( y \)-axis is vertical and pointing upwards.

In three dimensional \( xyz \)-space, \( \mathbb{R}^3 \), positions are specified by triples \( \langle a, b, c \rangle \) giving the \( x \)-, \( y \)- and \( z \)-coordinates of the vertex. However, the convention for how the three coordinate axes are positioned is different for computer graphics than is usual in mathematics. In computer graphics, the \( x \)-axis is pointing to the right, the \( y \)-axis is pointing upwards, and the \( z \)-axis is pointing towards the viewer. This is different from what you may be used to: for example, in calculus, the \( x \)-, \( y \)- and \( z \)-axes usually point forwards, rightwards, and upwards (respectively). The computer graphics convention was adopted presumably because it keeps the \( x \)- and \( y \)-axes in the same position as for the \( xy \)-plane. But of course it has the disadvantage of taking some getting used to. Figure I.4 shows the orientation of the coordinate axes as used in computer graphics.

It is important to note that the coordinates axes used in computer graphics do form a righthanded coordinate system. This means that if you position your right hand with your thumb and index finger extended to make an “L” shape, and place your hand so that your right thumb points along the positive \( x \)-axis and your index finger points along the positive \( y \)-axis, then your palm will be facing towards the positive \( z \)-axis. In particular, this means that the usual righthand rule applies to cross products of vectors in \( \mathbb{R}^3 \).
I.3.1 Loading vertices into a VAO and VBO (Draft A.2.d)

Figure I.4: The coordinate axes in $\mathbb{R}^3$, and the point $(a, b, c)$. The $z$-axis is pointing towards the viewer.

I.3 Points, lines and triangles in OpenGL

This section gives an overview of how OpenGL specifies the geometries of points, lines and triangles, focusing on the simplest and most common methods. Sample code and explanations of these, and other OpenGL features, can be found in the programs SimpleDrawModern, SimpleAnimModern and BasicDrawModes available from the book’s website. These programs also illustrate how the C++ code, OpenGL function calls and shader programs fit together in a fully functioning program.

For now, we discuss only drawing vertices at fixed positions in the $xy$-plane or in $xyz$-space. These are specified by giving an array of vertex positions, plus the information about how they are joined to form lines or triangles.

Later on in this chapter, we will show some simple shader programs and discuss shading triangles with color, culling front- or back-faces, using the depth buffer for hidden surfaces, and double buffering for animation. Chapter II will explain how to move vertices and geometric shapes around with rotations, translations, and other transformations.

I.3.1 Loading vertices into a VAO and VBO

Points, lines and triangles are always specified in terms of vertex positions. Vertex positions are generally given in terms of their $x, y$-coordinates in $\mathbb{R}^2$ or their $x, y, z$-coordinates in $\mathbb{R}^3$. Let’s start with a simple example of six vertices in $\mathbb{R}^2$. In a C or C++ program, they can be given explicitly in an array as:

\[ \begin{pmatrix} a \\ b \\ c \end{pmatrix} \]

However, later on, when working with homogeneous coordinates, we see that positions might be given in terms $x, y, z, w$-coordinates instead.
Figure I.5: The six points in the \texttt{verts} array.

```cpp
// Array of x,y coordinates for six vertices
float verts[] = {
    0.5, 1.0, // Vertex \( v_0 \)
    2.0, 2.0, // Vertex \( v_1 \)
    1.8, 2.6, // Vertex \( v_2 \)
    0.7, 2.2, // Vertex \( v_3 \)
    1.6, 1.2, // Vertex \( v_4 \)
    1.0, 0.5, // Vertex \( v_5 \)
};
```

This declaration of the array \texttt{verts} allocates space for 12 \texttt{float}'s (floating point numbers) giving the \( x,y \) coordinates of vertices \( v_0 \) through \( v_5 \). For example, \texttt{verts[4]} and \texttt{verts[5]} are equal to 1.8 and 2.6, and give the \( x \)-and \( y \)-coordinates of the point \( v_2 = (1.8, 2.6) \). Figure I.5 shows the positions of these vertices.

The array \texttt{verts} is allocated in the memory space of the C++ program, perhaps even in temporary storage in the C++ execution stack. To render objects using these vertices, it is necessary to load the data into OpenGL's memory space. This is unfortunately a bit complex, but it can be accomplished with the following (minimalistic) code:

```cpp
unsigned int theVAO; // Name of a Vertex Array Object (VAO)
glGenVertexArrays(1, &theVAO);

unsigned int theVBO; // Name of a Vertex Buffer Object (VBO)
glGenBuffers(1, &theVBO);

glBindBuffer(GL_ARRAY_BUFFER, theVBO); // Select the active VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts, GL_STATIC_DRAW);

unsigned int vPos_loc = 0; // Shader's 'location' for positions
glBindVertexArray(theVAO); // Select the active VAO
glVertexAttribPointer(vPos_loc, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);
glEnableVertexAttribArray(vPos_loc);
```
I.3.1. Loading vertices into a VAO and VBO (Draft A.2.d)

There is a lot going on in this code. It starts off by asking OpenGL to setup a Vertex Array Object (VAO) and a Vertex Buffer Object (VBO). A VBO holds vertex data; in our example, it holds the $x, y$ coordinates of vertices. In other applications, as is discussed later, the VBO might hold texture coordinates, surface normals, color or material properties, etc. A VAO holds information about the way the vertex data is stored in the VBO: the primary purpose of a VAO is to let shader programs know how to access the data stored in the VBO. The code copies the vertex data into the VBO by calling `glBufferData`. The call to `glVertexAttribPointer` tells the VAO that the vertex data consists of pairs of floating point numbers and that the data starts at the beginning of the VBO.

**Vertex Buffer Object.** The VBO holds per-vertex data; namely, in our example, each vertex has its own $x, y$ coordinates. These vertex data are also called **vertex attributes**. The two lines

```c
unsigned int theVBO;
glGenBuffers(1, &theVBO);
```

ask OpenGL to setup a new VBO. The first parameter to `glGenBuffers` is the number of VBO’s to be setup. Each VBO is referenced (or, named) by an unsigned integer; the second parameter to `glGenBuffers` is the address where the names of the new VBO’s are returned.

The two lines

```c
glBindBuffer(GL_ARRAY_BUFFER, theVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts, GL_STATIC_DRAW);
```

copy the per-vertex data into the VBO. OpenGL uses several kinds of buffers; one kind is the `GL_ARRAY_BUFFER` and the call to `glBindBuffer` makes `theVBO` the current `GL_ARRAY_BUFFER`. `glBufferData` is used to copy the per-vertex data to the VBO. Its function prototype is:

```c
void glBufferData(GLenum target, int size, void* data, GLenum usage);  
```

The `target` specifies which OpenGL buffer to copy data to. The fourth parameter provides a hint as how the data will be used. `GL_STATIC_DRAW` specifies that the data is used frequently but not changed often; this encourages OpenGL to store the data in the GPU memory for better rendering speed.\(^2\) The second parameter, for us `sizeof(verts)`, is the number of bytes of data to be uploaded. The VBO will be resized as needed to accommodate the data. The

---

\(^2\)This function prototype is not quite correct. We will simplify function prototypes for the sake of simplicity of understanding. For the same reason, we often gloss over many of the command features. You are encouraged to learn more about these OpenGL functions by searching online; when you do so, be careful to find the OpenGL documentation (not the OpenGL ES documentation).

\(^3\)An alternative to `GL_STATIC_DRAW`, if the data is changed every time the scene is rendered is, `GL_DYNAMIC_DRAW`. 
third parameter is a pointer to the beginning of data to be copied into the VBO.

**Vertex Array Object.** The VAO holds the information about the vertex attributes. In our case, this information is that the per-vertex data consists of pairs of floating point numbers and that they are stored in adjacent locations starting at the beginning of the VBO. This is accomplished with following lines of code:

```c
glBindBuffer(GL_ARRAY_BUFFER, theVBO);
···
unsigned int vPos_loc = 0; // Shader's 'location' for positions
glBindVertexArray(theVAO);
glVertexAttribPointer(vPos_loc, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);
```

We first call `glBindBuffer` and `glBindVertexArray` to select the current VAO and VBO. The function prototype for `glVertexAttribPointer` is:

```c
glVertexAttribPointer( int index, int size, GLenum type, bool normalize, int stride, void* bufferOffset );
```

The first parameter, `index`, is the shader program’s “location” for the data: later on, our vertex shader will use “location=0” to access this data. The second and third parameter specifies the number of data values per vertex and their type: in our example, each vertex has two (2) floating point numbers, so we used “2, GL_FLOAT”. The final parameter gives the offset in bytes into the VBO where the data starts. For us, this was “(void*)0”, as the vertex data starts at the beginning of the VBO. The fifth parameter, called the `stride`, specifies the spacing between the data for successive vertices. The “stride” is the number of bytes from the start of the data for one vertex to the start of the data to the next vertex. We used “0” for the stride, which tells OpenGL to calculate the stride based on the assumption the data is tightly packed. We could have also used the command

```c
glVertexAttribPointer( vPos_loc, 2, GL_FLOAT, GL_FALSE, 2*sizeof(float), (void*)0 );
```

with “0” replaced by “2*sizeof(float)” to give the stride explicitly.

The fourth parameter, `normalize`, controls how integer values are converted to floating point when uploaded to the VBO. We are uploading floating point values, so this parameter is ignored in our example.

The final step is to tell the VAO to “enable” the per-vertex data in the VBO. Our code did this with the command

```c
glEnableVertexAttribArray(vPos_loc);
```

This tells the VAO that the vertex attribute with index `vPos_loc` is specified on a per-vertex basis from the VBO. (As we discuss in Chapter III, an alternative is to use a which uses the same value for multiple vertices.)
I.3.2 Drawing points and lines

Now that we have defined an array of vertex positions, and loaded the needed data into the VBO and VAO, we are ready to give the code that actually renders the points. To render the vertices as six isolated points, we use the C++ code

```cpp
glBindVertexArray(theVAO);
int vColor_loc = 1;
glVertexAttrib3f(vColor_loc, 0.0, 0.0, 0.0); // Black color (0,0,0)
glDrawArrays(GL_POINTS, 0, 6);
```

The result of these commands is to draw the six points as shown in Figure I.6a (compare to Figure I.5). The call to `glBindVertexArray` selects the VAO. Since the VAO knows which VBO holds the per-vertex data, there is no need to explicitly bind the VBO. The `glVertexAttrib3f` function call sets a color value, in this case, the color is black. The color is given by three floating point numbers, in the range 0 to 1.0, giving the red, green and blue (RGB) components of the color. In our code, the color is a *generic* attribute; this means that all the vertices have the same color. Consequently, the color value must be set just before rendering the points, instead of being stored in the VBO on a per-vertex basis. The `vColor_loc` value is 1, as our vertex shader program will use `location=1` to access the color.

The call to `glDrawArrays` is the command that actually causes the vertices to be rendered. The first parameter, `GL_POINTS`, tells OpenGL to draw the vertices as isolated points. The second parameter, 0, says to start with vertex number 0, i.e., the first vertex. The third parameter, 6, is the number of vertices to be rendered.

You can also render the points joined by lines. The first mode, `GL_LINES`, uses the code

```cpp
glBindVertexArray(theVAO);
int vColor_loc = 1;
glVertexAttrib3f(vColor_loc, 0.0, 0.0, 0.0); // Black color (0,0,0)
glDrawArrays(GL_LINES, 0, 6);
```

This is identical to the code used to render the vertices as points, but uses `GL_LINES` in place of `GL_POINTS`. It results in drawing the lines shown in Figure I.6b. The effect is to use each successive pair of vertices as endpoints of a line segment. Namely, if there are `n` vertices \( v_0, \ldots, v_{n-1} \) it draws the

---

4Before giving this code, one also has to state which shader program to use. Shaders will be discussed soon.

5In fact, it is possible that multiple VBO's are used to hold different vertex attributes.

6The advantage of this is that if the vertices all have the same color, there is no need to waste memory in the VBO specifying the same color repeatedly. This disadvantage is, at least in the most widely used versions of OpenGL, that one has to call `glVertexAttrib3f` each time the color changes during rendering.
line segments with endpoints \( c_{2i} \) and \( v_{2i+1} \) for \( 0 \leq i < n/2 \). This is shown in Figure I.6b.

Another option for lines is GL\_LINE\_STRIP which draws a connected sequence of line segments, starting with the first vertex, joining a line segment to each successive vertex, ending at the last vertex. In other words, when there are \( n \) vertices, it draws the line segments joining \( v_i \) and \( v_{i+1} \) for \( 0 \leq i < n-1 \). This is pictured in Figure I.6c.

The option GL\_LINE\_LOOP draws these line segments, plus a line from the final vertex back to the first vertex, thus rendering a closed loop of line segments. This draws in addition, the line segment joining \( v_{n-1} \) and \( v_0 \). For this, see Figure I.6d.

There is a sample OpenGL program, SimpleDrawModern, available at the book's website, which contains a complete C++ program with the above code for drawing the six vertices as line segments, as a line strip, and as a line loop. (It also draws a couple other images.) If OpenGL is new to you, it is recommended that you examine the source code and try compiling and running the program.

If you run OpenGL with default settings, you will probably discover that the points are drawn as very small, single pixel points — perhaps so small as to be almost invisible. Similarly, the lines may look much too thin, and may be visibly jagged because of the lines being drawn only one pixel wide. By default, OpenGL draws thin lines, one pixel wide, and does not do any “anti-aliasing” to smooth out the lines. On most OpenGL systems, you can call the following functions to make points display as large, round dots to make lines be drawn wider and smoother.
I.3.3 Drawing triangles (Draft A.2.d)

```
glPointSize(n); // Points are n pixels in diameter
glEnable(GL_POINT_SMOOTH);
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
gLineWidth(m); // Lines are m pixels wide
glEnable(GL_LINE_SMOOTH);
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST); // Antialias lines
glEnable(GL_BLEND);
gBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

In the first line, a number such as 6 for \( n \) may give good results so that points are six pixels in diameter. In the fourth, using \( m = 3 \) so that lines are three pixels wide may work well. The SimpleDrawModern program already includes the above function calls. If you are lucky, executing these lines in the program before the drawing code will cause the program to draw nice round dots for points. However, the effect of these commands varies with different implementations of OpenGL, so you may see square dots instead of round dots, or even no change at all. How well, and whether, the point sizing and blending works and the line width specification and the anti-aliasing work will depend on your implementation of OpenGL.

**Exercise I.1** The OpenGL program SimpleDrawModern includes code to draw the images shown in Figure I.6, and a colorized version of Figure I.12. Run this program, examine its source code, and read the online explanation of how the code works. (At this point, you should be able to understand everything except for drawing triangles and using the vertex and fragment shaders.) Learn how to compile the program. Then try disabling the code for making bigger points, and wider, smoother lines. What changes does this cause?

**Exercise I.2** Write an OpenGL program to generate the two star-shaped images of Figure I.7 as line drawings. You will probably want modify the source code of SimpleDrawModern for this. (Exercises I.3 and I.4 use the same figure.)

I.3.3 Drawing triangles

The fundamental rendering operation for 3D graphics is to draw a triangle. Ordinarily, triangles are drawn as solid, filled-in shapes. That is to say, we usually draw triangles, not edges of triangles or vertices of triangles. The most basic way to draw triangles is to use the `glDrawArrays` command with one of the options `GL_TRIANGLES`, `GL_TRIANGLE_STRIP` or `GL_TRIANGLE_FAN`.

Figure I.8a illustrates how triangles are rendered with `GL_TRIANGLES` and `glDrawArrays`. Here it is assumed that the six vertices \( u_0, \ldots, u_5 \) are stored in the VBO: these vertices are grouped into triples, and each triple defines the three vertices of a triangle. More generally, if there \( n \) vertices \( u_0, \ldots, u_{n-1} \)
then, for each \( i < n/3 \), the vertices \( u_{3i}, u_{3i+1}, u_{3i+2} \) are used to form a triangle. Each triangle has a \textit{front side} and a \textit{back side}. On the front side, the viewer sees the vertices \( u_{3i}, u_{3i+1}, u_{3i+2} \) ordered in counter-clockwise (CCW) order around the triangle. From the back side, a viewer sees the vertices going around the triangle in clockwise (CW) order.

Figure I.8b illustrates how triangles are rendered with \texttt{GL_TRIANGLE_FAN} in a fan-like pattern, sharing a common vertex \( u_0 \). Again, there are six vertices \( u_0, \ldots, u_5 \) (but different ones than in Figure I.8a). If there \( n \) vertices \( u_0, \ldots, u_{n-1} \) with \texttt{GL_TRIANGLE_FAN}, then for each \( 1 \leq i < n-1 \), the vertices \( u_0, u_i, u_{i+1} \) are used to form a triangle. These triangles again have a front side and back side. The front sides are viewed with the vertices \( u_0, u_i, u_{i+1} \) in counter-clockwise order. The triangles in Figure I.8b are all front facing.

Figure I.8c illustrates how triangles are rendered with \texttt{GL_TRIANGLE_STRIP}. Once again, there are six vertices, \( u_0, \ldots, u_5 \), but ordered differently. This renders the triangle with vertices \( u_0, u_1, u_2 \), then the triangle with vertices \( u_1, u_3, u_2 \), then the triangle with vertices \( u_2, u_3, u_4 \), and then the triangle with vertices \( u_3, u_5, u_4 \). The orders of the vertices are chosen so that the front faces of the triangles are the faces where the vertices are seen in counter-clockwise order. Figure I.8c shows the triangles with front faces visible.

XXX INSERT HERE: USING TRIANGLES TO RENDER A SPHERE. ALSO REFERENCE THE TORUS WHICH IS DRAWN LATER ON.

To render the triangles shown in Figure I.8a, we can use the code shown below. First, an array \texttt{verts2} is loaded with seven vertices, given by their \( x, y, z \) coordinates (the \( z \)-coordinates are all zero):
I.3.3. Drawing triangles (Draft A.2.d)

Figure I.8: The three triangle drawing modes. These are shown with the default front face visible to the viewer. The vertices \( u_i \) are numbered in the order needed for the drawing modes. For this, it is important to note the difference in the placement and numberings of vertices in each figure, especially of vertices \( u_4 \) and \( u_5 \) in the first and last figures.

Figure I.9: The points used for rendering the triangles with \texttt{glDrawElements}.

```c
// Array of x,y,z coordinates for seven vertices
float verts2[] = {
    0.25, 0.5, 0.0,  // Vertex w0
    1.25, 1.0, 0.0,  // Vertex w1
    0.75, 1.5, 0.0,  // Vertex w2
    1.75, 1.8, 0.0,  // Vertex w3
    2.0,  3.0, 0.0,  // Vertex w4
    1.05, 2.5, 0.0,  // Vertex w5
    0.4,  2.4, 0.0,  // Vertex w6
};
```

Figure I.9 shows all the points of \texttt{verts2}. The \texttt{verts2} array is loaded into a VAO and VBO with the following code (we assume the VAO and VBO have already been setup and that \texttt{vPos.loc} has been defined):
```c
glBindBuffer(GL_ARRAY_BUFFER, theVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(verts2), verts2, GL_STATIC_DRAW);
glBindVertexArray(theVAO);
glVertexAttribPointer(vPos_loc, 3, GL_FLOAT, GL_FALSE, 3*sizeof(float), (void*)0);
glEnableVertexAttribArray(vPos_loc);
```

The triangles shown in Figure I.8a can be rendered with the code (assuming vColor_loc is defined as before):

```c
glBindVertexArray(theVAO);
glVertexAttrib3f(vColor_loc, 0.7, 0.7, 0.7); // Light gray
glDrawArrays(GL_TRIANGLES, 0, 6);
```

The call to `glDrawArrays` specifies to draw triangles using a total of six vertices, starting with vertex number 0. There is a seventh vertex in `verts2` which has also been loaded into the VBO, but it is ignored for the moment. The code as written will render them as light gray triangles without any borders.

The triangle fan and triangle strip shown in Figure I.8 each use six of the vertices given in `verts2`; but the `verts2` entries are not in the correct order to render them with `glDrawArrays`. We next discuss how this instead can be done using element arrays.

### I.3.4 Rendering with element arrays

An element array allows using indices of vertices for drawing. This can be useful when the vertices are reused for several drawing commands. For example, a single vertex might appear in multiple triangles drawn in `GL_TRIANGLES` mode, or in multiple triangle strips and/or triangle fans. Rather than make multiple copies of the vertices with all their attributes, we can make one copy of the vertex in a VBO and then reference it multiple times via element arrays. (For this, it is necessary that the vertex attributes are the same each time the vertex is referenced.)

An element array is sometimes called an “element array buffer”, “element buffer object” (EBO), or “index buffer object” (IBO). For an example of how to use element arrays, the following code can draw the triangles shown in Figure I.8. First, we allocate and define the element array by:

```c
unsigned int elements[] = {
  0, 1, 2, 3, 4, 5,   // For GL_TRIANGLES
  2, 0, 1, 3, 5, 6,   // For GL_TRIANGLE_FAN
  0, 1, 2, 3, 5, 4    // For GL_TRIANGLE_STRIP
};
```

The indices in the `elements` array indicate which vertices from the `verts` array are to be used with `glDrawArrays`. For example, the triangle fan will be drawn
I.3.5 Different colors per vertex

The earlier code examples assigned the same color to all the vertices, using `glVertexAttrib3f` to set the color as a generic attribute. It is also possible to set the vertex colors on a per-vertex basis, storing them in the VBO similarly to the way the vertex positions are set. We illustrate next how to do this by interleaving vertex positions and colors into a single array. Similar methods can be used for other vertex attributes, such as texture coordinates, normals, material properties, etc.\footnote{It is also possible to store different vertex attributes in different arrays or even in different VBO's, instead of interleaving them in a single array.}

The following code allocates an element buffer object (EBO) and loads the data from the elements array into the EBO:

```cpp
unsigned int theEBO;
glGenBuffers( 1, &theEBO );
glBindVertexArray(theVAO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, theEBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements), elements, GL_STATIC_DRAW);
```

These commands work just like the earlier code that loaded data into the VBO. The triangles of Figure I.8a can then be drawn with

```cpp
glBindVertexArray(theVAO);
glVertexAttrib3f(vColor_loc, 0.7, 0.7, 0.7); // Light gray
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Note that we are now using `glDrawElements` instead of `glDrawArrays`. The second and fourth parameters to `glDrawElements` are 6 and 0: they specify that the drawing should use the vertices referenced by the six indices starting at position 0 in the EBO (the position is measured bytes). The third parameter tells OpenGL that the indices are stored as unsigned integers. To draw the triangle fan of Figure I.8b, the `glDrawElements` call is replaced with

```cpp
glBindVertexArray(theVAO);
glVertexAttrib3f(vColor_loc, 0.7, 0.7, 0.7); // Light gray
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

This again uses six vertices, but now the indices start at byte position `6*sizeof(unsigned int)` in the EBO, i.e., at the seventh index in the array.

Finally, to draw the triangle strip of Figure I.8c, use

```cpp
glBindVertexArray(theVAO);
glVertexAttrib3f(vColor_loc, 0.7, 0.7, 0.7); // Light gray
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

The program SimpleDrawModes at the book's website shows a complete program using `glDrawElements` along with a description of how it works.
// Array of x,y coordinates and r,g,b colors for three vertices
float verts3[] = {
    // x, y position; R, G, B color
    0.0, 0.0, 1.0, 0.0, 0.0, // 1st vertex, Red (1,0,0)
    2.0, 0.0, 0.0, 1.0, 0.0, // 2nd vertex, Green (0,1,0)
    1.0, 1.5, 0.0, 0.0, 1.0, // 3rd vertex, Blue (0,0,1)
};

The verts3 array gives the three vertices different colors by giving the red, green and blue component of the color using a number in the interval [0,1]. Other colors can be specified by blending different amounts of red, green, and blue. For example, white is specified by the color (1,1,1), yellow is specified by (1,1,0), magenta by (1,0,1), and by (0,1,1). You can find many other color codes by searching online of “RGB colors”; usually they are specified with color components given by integers between 0 and 255: they can be converted to the range [0,1] by dividing by 255.\footnote{It is also possible to use integers in the range 0 to 255 with OpenGL: for example, you can use \texttt{GL_UNSIGNED_BYTE} instead of \texttt{GL_FLOAT} when calling \texttt{glVertexAttribPointer}.}

The vert3 data can be loaded into the VAO and VBO by:

```c
glBindBuffer(GL_ARRAY_BUFFER, theVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(verts3), verts3, GL_STATIC_DRAW);
glBindVertexArray(theVAO);
glVertexAttribPointer(vPos_loc, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float), (void*)0);
glVertexAttribPointer(vColor_loc, 3, GL_FLOAT, GL_FALSE, 5*sizeof(float), (void*)(2*sizeof(float)));
glEnableVertexAttribArray(vPos_loc);
glEnableVertexAttribArray(vColor_loc);
```

The first call to \texttt{glVertexAttribPointer} specifies that the vertex position data consists of two float's per vertex, that the stride from one vertex to the next is 5*sizeof(float) bytes, and that the first vertex's position data starts at the beginning of the VBO. The second call to \texttt{glVertexAttribPointer} specifies that the vertex color data consists of three float's per vertex, that the stride is again 5*sizeof(float) bytes, and that the first vertex's color data starts at position 2*sizeof(float) in the VBO (measured in bytes). The two calls to \texttt{glEnableVertexAttribArray} specify that the position and color data are stored on a per-vertex basis in the VBO (that is, they are not generic attributes).

Rendering the three colored vertices as a triangle is done with the usual command, now using only three vertices to render a single triangle,

```c
glDrawArrays(GL_TRIANGLES, 0, 3);
```
I.3.6 Face orientation and culling

OpenGL keeps track of whether triangles are facing towards the viewer or away from the viewer, that is to say, OpenGL assigns each triangle a front face and a back face. Sometimes, it is desirable for only the front faces of triangles to be viewable, and at other times you may want both the front and back faces of a triangle to be visible. If we set the back faces to be invisible by “culling” them, then any triangle whose back face would ordinarily be seen is not drawn at all. In effect, a culled face becomes transparent. By default, no faces are culled so both front- and back-faces are visible.

As we already discussed, OpenGL determines which face of a triangle is the front face by the default convention that vertices on a triangle are specified in counter-clockwise order (with some exceptions for triangle strips). The triangles shown in Figures I.8 and I.10a are all shown with their front faces visible.

You can change the convention for which face is the front face by using the `glFrontFace` command. This command has the format

```
glFrontFace( { GL_CW GL_CCW } );
```

where “CW” and “CCW” stand for clockwise and counter-clockwise. `GL_CCW` is the default. Using `GL_CW` causes the opposite convention for front and back faces to be used on subsequent triangles.

To cull front and/or back faces, use the commands.
Figure I.11: Two wireframe tori. The upper torus does not use culling; the lower torus has back faces culled. NEED TO UPDATE FIGURE

\[
\text{glCullFace}\left( \begin{array}{l} 
\text{GL\_FRONT} \\
\text{GL\_BACK} \\
\text{GL\_FRONT\_AND\_BACK} 
\end{array} \right); \\
\text{glEnable}(\text{GL\_CULL\_FACE}); 
\]

You must explicitly turn on the face culling with the call to \texttt{glEnable}. Face culling can be turned back off with the corresponding \texttt{glDisable} command. When culling is enabled, the default setting for \texttt{glCullFace} is \texttt{GL\_BACK}.

The wireframe torus of Figure I.11 is shown without any face culling. Figure I.11 shows the same torus with back face culling.

The spheres in Figure ?? are rendered with back faces culled. Since the sphere is convex, culling back faces is good enough to correctly remove hidden surfaces.

XXX REPLACE THE WIRE TORUS FIGURES AND DISCUSS
I.3.7  Wireframe triangles

By default, OpenGL draws triangles as filled in. It is possible to change this by using the `glPolygonMode` function to specify whether to draw solid triangles, wireframe triangles, or just the vertices of triangles. This makes it easy for a program to switch between wireframe and non-wireframe mode. The syntax for the `glPolygonMode` command is

```c
glPolygonMode(
{ GL_FRONT, GL_BACK, GL_FRONT_AND_BACK },
{ GL_FILL, GL_LINE, GL_POINT });
```

The first parameter to `glPolygonMode` specifies whether the mode applies to front and/or back faces. The second parameter sets whether triangles are drawn filled in, as lines, or as just vertices.

I.4  Vertex shaders and fragment shaders

Modern graphics systems provide a huge amount of flexibility in how rendering occurs. The simplest functionality is obtained by writing a vertex shader and fragment shader. A vertex shader is a small program that is run once for each vertex. A fragment shader is a small program that is run once for each pixel rendered. (Fragment shaders are sometimes called “pixel shaders”, and in most cases, you can think of “fragment” as being a synonym for “pixel”.) In this section, we’ll discuss some simple uses of vertex and fragment shaders. Later on, we’ll describe more sophisticated shaders.

The terminology “shader” is a bit of a historical anomaly. The term “shading” means the process of letting the color or brightness vary smoothly across a surface. Later on, we shall see that shading is an important tool for creating realistic images, particularly when combined with lighting models that compute colors from material properties and light properties, rather than using colors that are explicitly set by the programmer. Shader programs are so-named because they were originally intended mostly for controlling shading of triangles; however, they can be used for many other purposes as well.

The main inputs to a vertex shader are the attributes of a single vertex; usually some of these attributes come from a VBO, while others are specified globally as generic attributes or “uniform variables”. In a typical application, the vertex shader may receive the $x,y,z$-coordinates of the vertex, and perhaps a color or texture coordinates. The vertex shader can then do rather sophisticated computations, including changing the per-vertex data, possibly changing the color or texture coordinates or even the position of the vertex.

The main output of the vertex shader is a value `gl_Position` which gives the $x,y$-coordinates of the vertex in screen coordinates. The `gl_Position` also gives a “depth” value which will be used for hidden surface computation as discussed later in Section I.4.4 and Chapter II. This depth value is often called the “$z$-value”. In actuality, `gl_Position` has a $w$-component so that values....
are represented in homogeneous coordinates, but this last part will be discussed in Chapter II. For now, we just think of the \texttt{glPosition} as giving the $x$- and $y$-coordinate of a pixel on the screen, with $x$ and $y$ both ranging from $-1$ to 1. As $x$ varies from $-1$ to 1, the pixel position varies from left to right. As $y$ varies from $-1$ to 1, the pixel position varies from the bottom of the image to the top.

A vertex shader will output other, shader-specific values in addition to the \texttt{glPosition}. For example, the shader-specific output values might be the red, green and blue components of the color of the pixel.

The fragment shader is invoked when pixels are actually rendered in a framebuffer. The framebuffer is a two dimensional array of values, usually holding a color value and a depth value for each pixel in the image. (See Section I.4.4 for more on depth values). Consider a triangle with three vertices. The vertex shader is run on each vertex, and the three output values \texttt{glPosition} give three pixels on the screen. These three pixels specify a triangle of pixels. Usually, the triangle will be filled in ("shaded"), with each pixel of the triangle being rendered as some color. Figures III.1 and III.2 on pages 99 and 101 give an idea of how this works. Note that the edges of the filled in triangle will in general be a bit "jaggy" to accommodate the fact that pixels are rectangularly arranged.

When a triangle is rendered, OpenGL determines which framebuffer pixels it covers. OpenGL then calls the fragment shader once per pixel. The job of the fragment shader is to calculate the color that should be placed into the corresponding pixel.

The inputs to the fragment shader correspond to the shader-specific values output by the vertex shader. By default, each such input to the fragment shader is calculated as an average of the corresponding shader-specific values output by the vertex shader. That is, when rendering a triangle, an input to the fragment shader is an average of the outputs from from the vertex shaders for the three triangle vertices. This averaging works in the same way that colors were averaged in the triangle in Figure I.10a in the last code example above. The fragment shader’s main output is a vector giving the red, green, and blue components of the pixel (plus an alpha or transparency value). This output from the fragment shader is the color to displayed in that pixel — unless some other triangle overwrites it.

I.4.1 Very simple vertex and fragment shader example

Vertex and fragment shaders can be quite complex, but for now, we illustrate the idea with some very simple shaders. The following program is a vertex shader that can work with the earlier code examples:
#version 330 core
layout (location = 0) in vec3 vertPos; // Position, at 'location' 0
layout (location = 1) in vec3 vertColor; // Color, at 'location' 1
out vec3 theColor; // Output a color

void main() {
    gl_Position = vec4(vertPos.x-1.0, 0.666*vertPos.y-1.0, vertPos.z, 1.0);
    theColor = vertColor;
}

The shader is written in the OpenGL Shader Language (GLSL). The first line indicates the version of the language being used. The next two lines give the “locations” of the two vertex attributes (position and color) and specify that they are 3-vectors (vec3’s). Note how x, y, z-components of the vec3’s are referenced by using the suffixes .x, .y, and .z. The earlier code samples with the vert array specified only the x- and y-coordinates of vertices in the VBO; however, the vertPos variable is still allowed to be a vec3 as the z-component defaults to 0. The fourth line specifies that there is a shader-specific output value called theColor. The vertex shader’s main program sets the pixel position in gl_Position and sets the shader-specific output theColor equal to the vertex attribute vertColor.

Note that the x- and y-coordinates of the position are transformed by the equations

\[
\text{vertPos.x-1.0 \quad and \quad 0.666*vertPos.y-1.0} \quad \text{(I.1)}
\]

The input x-coordinates vertPos.x from all our examples range between 0 and 2, and thus vertPos.x-1.0 ranges between -1 and 1. Similarly, the input y-coordinates vertPos.y range between 0 and 3, and 0.666*vertPos.x-1.0 ranges between -1 and 1. Therefore the transformations (I.1) ensure that the output x- and y-coordinates in gl_Position lie in the range [-1, 1]. If output coordinates were outside [-1, 1], the pixel would lie outside the visible part of the screen: the vertex would not be rendered (and triangles containing that vertex would be either clipped or culled).

The transformations of (I.1) are ad-hoc and very specific to our earlier examples. Chapter II will discuss more general matrix methods for transforming points to lie in the visible part of the screen.

The simplest possible fragment shader is:

#version 330 core
in vec3 theColor; // Input color (averaged)
out vec4 FragColor; // Output color

void main() {
    FragColor = vec4(theColor, 1.0f); // Alpha value is 1.0
}

The only thing this fragment shader does is copy the input color to the output
color. Note that the “in” declaration of theColor in the fragment shader exactly matches the “out” declaration in the vertex shader. The output FragColor has four components: red, green, blue and alpha (RGBA).

I.4.2 A fragment shader modifying the color

For a small, very simple, example of the power of shaders, we give a second fragment shader. This shader can be used with the same vertex shader as above; it takes the same color (theColor) as input, but increases to maximum brightness while maintaining its hue. It does this by multiplying the color by a scalar to increase its maximum component to 1. Here is the code:

```
#version 330 core
in vec3 theColor; // Input color (averaged)
out vec4 FragColor; // Output color
void main()
{
    float mx = max(theColor.r,max(theColor.g,theColor.b));
    if ( mx!=0 ) {
        FragColor = vec4(theColor/mx, 1.0f); // Scale maximum component to 1.0
    } else {
        FragColor = vec4(1.0, 1.0, 1.0, 1.0); // Replace black with white
    }
}
```

This code uses the suffixes .r, .g and .b to access the three components of the 3-vector theColor. (These have exactly the same meaning as .x, .y and .z, and are used here just as reminder that represents an RGB color.) Thus mx is equal to the maximum of the RGB components of the input color. Figure I.10b shows the triangle as rendered by this fragment shader. For the pixel midway between the red and green vertices, the fragment shader computes mx to equal $\frac{1}{2}$, and outputs the color as $\langle 1,0,1 \rangle$, namely a bright yellow instead a dark yellow. For the pixel at the middle of the triangle, mx equals $\frac{1}{3}$, and its FragColor color is $\langle 1,1,1 \rangle$, namely white.

I.4.3 Flat shading

In flat shading, the shader-specific values output by the vertex shader are not averaged across the triangle. Instead, one of the vertices has its values used for the entire triangle. To use flat shading, the vertex and fragment shaders have to be modified to declare the out/in variable as being flat. For example, a simple vertex shader with flat shading could be
I.4.4 Hidden surfaces

When we render three dimensional scenes, objects that are closer to the viewpoint may occlude, or hide, objects which are farther from the viewer. OpenGL uses a depth buffer that holds a distance or depth value for each pixel. The depth buffer lets OpenGL do hidden surface computations by the

---

9 The `glProvokingIndex` command allows you to make flat mode use the first vertex of the triangle instead of the last vertex.

10 The depth value of a vertex is the z-coordinate of the `glPosition` value output by the vertex shader (after perspective division).
simple expedient of drawing into a pixel only if the new distance will be less than the old distance. The typical use of the depth buffer is as follows: When an object, such as a triangle, is rendered, OpenGL determines which pixels need to be drawn and computes a measure of the distance from the viewer to each pixel image. That distance is compared to the distance associated with the former contents of the pixel. The lesser of these two distances determines which pixel value is saved, since the closer object is presumed to occlude the farther object.

To better appreciate the elegance and simplicity of the depth buffer approach to hidden surfaces, we consider some alternative hidden surface methods. One alternative method, called the painter’s algorithm, sorts the triangles from most distant to closest and renders them in back-to-front order, letting subsequent triangles overwrite earlier ones. The painter’s algorithm is conceptually easy, but not completely reliable. In fact, it is not always possible to consistently sort triangles according to their distance from the viewer (cf. Figure I.12). In addition, the painter’s algorithm cannot handle interpenetrating triangles.

Another hidden surface method is to work out geometrically all the information about how the triangles occlude each other, and render only the visible portions of each triangle. This, however, can be difficult to design and implement robustly.

The depth buffer method, in contrast, is very simple and requires only an extra depth, or distance, value to be stored per pixel. Another big advantage of the depth buffer method is that it allows triangles to be rendered independently and in any order. The painter’s algorithm needs to collect a complete list of the triangles before starting the rendering, so that they can be sorted before rendering. Similarly, calculating occlusions requires processing all the triangles before any rendering occurs. The depth buffer method can render each triangle by storing colors and depths into the pixels covered by the triangle, and immediately discard all information about the triangle; it does not need to collect all the triangles before rendering into pixels.

The depth buffer is not activated by default. To enable it, you must enable GL_DEPTH_TEST:

\[
\text{glEnable(GL\_DEPTH\_TEST);} \quad // \text{Enable depth buffering}
\text{glDepthFunc(GL\_LEQUAL);}\]

The call to glDepthFunc says that a pixel should be overwritten if its new value has depth less than or equal to the current contents of the pixel. The default depth test function is GL\_LESS only keeps a new value if its depth is less than the old depth; in most applications this will work equally well.\(^{11}\)

It is also important to clear the depth buffer each time you render an image. This is typically done with a command such as

\[
\text{glClear( GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT );}\]

\(^{11}\text{Handling depth testing for exactly coincident object can be tricky. The glPolygonOffset function, discussed in Section II.3.5, can help handle this.}\)
I.5. Animation with double buffering

The term “animation” refers to drawing moving objects or scenes. The movement is only a visual illusion however; in practice, animation is achieved by drawing a succession of still scenes, called frames, each showing a static snapshot at an instance in time. The illusion of motion is obtained by rapidly displaying successive frames. Movies and videos typically have a frame rate of 24 or 48 frames per second; higher frame rates give much better results for higher speed motion. The frame rates in computer graphics can vary with the power of the computer and the complexity of the graphics rendering, but typically it is desirable to have 30 frames per second, and more ideally to get 60 frames per second. These frame rates are quite adequate to give smooth motion on a screen. For head mounted displays, where the view changes with the position of the viewer’s head, even higher frame rates are needed to get good effects.

Double buffering can be used to generate successive frames cleanly. While

\[\text{glClearBuffer}()\] or \[\text{glClearColor}()\].

The \texttt{SimpleDrawModern} program illustrates the use of the depth buffering for hidden surfaces. It shows three triangles, each of which partially hides another, as in Figure I.12. This example shows why ordering polygons from back-to-front is not a reliable means of performing hidden surface computation.

Figure I.12: Three triangles. The triangles are turned obliquely to the viewer so that the top portion of each triangle is in front of the base portion of another. This means that the painter’s algorithm is unable to sort the triangles in an order that correctly handles hidden surfaces.

which both clears the color (i.e., initializes the entire image to the default color) and clears the depth values (by setting them to the maximum possible depth). The default color and depth for \texttt{glClear} can be changed by calling \texttt{glClearColor()} or \texttt{glClearDepth}.

\[\text{glClearBuffer}()\] commands instead of \texttt{glClearColor} and \texttt{glClear}.

See the \texttt{SimpleAnimModern} program for an example.

\[\text{glClearBuffer}()\]
one image is displayed on the screen, the next frame is being created in another part of memory. When the next frame is ready to be displayed, the new frame replaces the old frame on the screen instantaneously (or rather: the next time the screen is redrawn, the new image is used). A region of memory where an image is being created or stored is called a buffer. The image being displayed is stored in the front buffer, and the back buffer holds the next frame as it is being created. When the buffers are swapped, the new image replaces the old on the screen. Note that swapping buffers does not generally require copying from one buffer to the other; instead, one can just update pointers to switch the identities of the front and back buffers.

A simple example of animation using double buffering in OpenGL is shown in the program SimpleAnimModern that accompanies this book. The main loop in that program uses the GLFW interface for OpenGL as follows:

```c
while (!glfwWindowShouldClose(window)) {
    myRenderScene(); // Render new scene
    glfwSwapBuffers(window); // Display new scene
    glfwWaitEventsTimeout(1.0/60.0); // Approximately 60 frames/sec
}
```

This call to myRenderScene renders into the current frame buffer, the so-called back buffer. While rendering, the display is showing the contents of the front buffer so that the rendering does not show up instantaneously on the display. Then glfwSwapBuffers interchanges the front and back buffers. This causes the just-rendered scene to be shown on the display. The previous screen image is now ready to be overwritten in the next rendering pass.

The call to glfwWaitEvents tells OpenGL to wait until some “event” has occurred or until 1/60 of second has elapsed. An “event” can be a mouse click or a mouse movement or a keystroke, or a window resizing, etc. When there are no events, the scene is redrawn approximately 60 times per second. This timing is not particularly accurate: for more reliable results, your program can either check the actual elapsed time. If it is supported, vertical synchronization can also be used.

If the scene is fixed and not animating, you can use

```c
glfwWaitEvents(); // Use this if no animation.
```

instead of glfwWaitEventsTimeout. This will cause the image to be redrawn only when an event has occurred. For instance, if the window is resized, the scene will need to be rerendered.

In some cases, you may wish animate as fast as possible, but still check for events. For this, you can use

```c
glfwPollEvents(); // Just check for events
```

The OpenGL programs SimpleDrawModern and SimpleDrawAnim on the book’s website provide examples of capturing keystrokes and resizing the graphics window. In addition, ConnectDotsModern shows how to capture mouse clicks.
I.6 Antialiasing with supersampling

Aliasing problems arise when converting between analog and digital representations, or when converting between different resolution digital representations. A simple example is the jagged lines that result when drawing a straight line on a rectangular array of pixels. These aliasing problems can be particularly distracting for animated scenes.

A simple way to dramatically reduce aliasing problems in an OpenGL program is to include the following two lines to turn on multisampling antialiasing (MSAA):

```cpp
glfwWindowHint(GLFW_SAMPLES, 4); // Invoke MSAA
glEnable(GL_MULTISAMPLE);
```

The parameter “4” instructs OpenGL to use “multisample aliasing” with each screen pixel holding four color and depth values — in a 2x2 arrangement of subpixels. Each screen pixel has its color calculated once per triangle that overlaps the screen pixel: the color and depth values are saved into those subpixels which are actually covered by the triangle. This can greatly improve many common aliasing problems with edges of triangles.

Multisample antialiasing (MSAA) is fairly expensive in terms of memory, since it requires four color and depth values per pixel: this can sometimes cause a noticeable slowdown in graphics performance. However, the fact that it is handled directly by the GPU hardware makes it relatively efficient. Other antialiasing methods will be discussed in Sections VI.1.3, VI.1.4 and X.2.1.

I.7 Additional exercises

**Exercise I.5** Write an OpenGL program that renders a cube with six faces of different colors. Form the cube from eight vertices and twelve triangles, making sure that the front faces are facing outwards. You can do this with either twelve separate triangles using GL_TRIANGLES, or with two or more triangle strips.

Experiment with changing between smooth shading and flat shading. If you already know how to perform rotations, let your program include the ability to spin the cube around.

**Exercise I.6** Repeat the previous exercise, but render the cube using two triangle fans.

**Exercise I.7** Modify the SimpleDrawProgram to draw the three triangles with white borders. This will require toggling the polygon mode. The program will first render the triangles first shaded with colors, and then redraw them with the polygon mode set to GL_LINE. Does it matter if you change the polygon modes are used in the opposite order?
(Depending on your OpenGL implementation, it is possible, but not likely, that there will be z-fighting between the lines and the polygons. If so, this can be fixed with `glPolygonOffset`.)

**Exercise I.8** An octahedron centered at the origin is formed from the eight vertices $\langle \pm 1, \pm 1, \pm 1 \rangle$. Describe way to order the vertices to render the octahedron as two triangle fans, with all faces facing outward.

**Exercise I.9** A tetrahedron-like shape is formed from the four vertices $\langle -1, 0, -1 \rangle$, $\langle 1, 0, -1 \rangle$, $\langle 0, 1, 0 \rangle$ and $\langle 0, 1, 0 \rangle$. Describe how to render the four faces of this shape with a single triangle fan, with all faces facing outward.

**Exercise I.10** A $2 \times 2 \times 2$ cube has the eight vertices $\langle \pm 1, \pm 1, \pm 1 \rangle$. Show how to render the six faces of the cube with two triangle fans, by explicitly listing the vertices used for the triangle fans. Make sure that the usual CCW front faces are facing outward. There are many possible vertex orderings, choose one that uses $\langle 1, 1, 1 \rangle$ as the first (central) vertex for one of the triangle fans.

**Exercise I.11** Consider rendering one triangle strip of a sphere as shown in Figure ?? . Suppose the sphere is formed from $k$ slices and $\ell$ stacks. The $k$-slices cut the sphere like an orange with $k$ slices, each slice takes up $2\pi/k$ radians around the central axis. The $\ell$ stacks are formed from $\ell - 1$ horizontal cuts (perpendicular to the central axis): the north pole is in the top stack and the south pole is in the bottom stack. How many triangles are in the triangle strip shown in Figure ?? running from the south pole to the north pole? How many triangles are in the triangle strip shown in Figure ?? running around the equator?

**Exercise I.12** Gamma correction (which will be discussed Chapter VII) alters an RGB color $\langle r, g, b \rangle$ to the color $\langle r^\gamma, g^\gamma, b^\gamma \rangle$. The value $\gamma$ is a fixed constant, usually between $\frac{1}{3}$ and $3$. The values $r, g, b$ should be in the range $[0, 1]$. Write a fragment shader that implements gamma correction. This will be similar to the fragment shader on page 24, but with a different algorithm for modifying the color. The GLSL function `pow` will be useful for this. The suggestion is to modify the program `Chapter1Figs` available on the book’s webpage and change the way the triangle of Figure I.10b is rendered by that program’s fragment shader. Try this with several values of $\gamma$ between 0.5 and 3.
Chapter II

Transformations and Viewing

This is a preliminary draft of a second edition of the book 3D Computer Graphics: A Mathematical Introduction with OpenGL. So please read it cautiously and critically! Corrections are appreciated. Draft A.2.d

Author: Sam Buss, sbuss@ucsd.edu

This chapter discusses the mathematics of linear, affine, and perspective transformations, as well as their uses in OpenGL. The basic purpose of these transformations is to provide methods of changing the shape and position of objects, but their use is pervasive throughout computer graphics. In fact, affine and perspective transformations are arguably the most fundamental mathematical tool for computer graphics. They are are mathematically very elegant, and even more importantly, are fairly easy for an artist or programmer to use. In addition they have efficient software and hardware implementations.

An obvious use of transformations is to organize and simplify the task of geometric modeling. As an example, suppose an artist is designing a computerized geometric model of a Ferris wheel. A Ferris wheel has a lot of symmetry and includes many repeated elements, such as multiple cars and struts. The artist could design a single model of the car, and then place multiple instances of the car around the Ferris wheel, attached at the proper points. Similarly, the artist could build the main structure of the Ferris wheel by designing one radial “slice” of the wheel and using multiple, rotated copies of this slice to form the entire structure. Affine transformations are used to describe how the parts are placed and oriented.

A second important use of transformations is to describe animation. If the Ferris wheel is animated, then the positions and orientations of its individual geometric components are constantly changing. Thus, for animation, it is necessary to compute time-varying affine transformations to simulate the motion of the Ferris wheel.
A third, more hidden, use of transformations in computer graphics is for rendering. After a 3D geometric model has been created, it is necessary to render it on a two dimensional surface called the *viewport*. Some common examples of viewports are a window on a computer monitor or a smartphone screen, a frame of a movie, and a hardcopy image. There are special transformations, called perspective transformations, that are used to map points from a 3D model to points on a 2D viewport.

To properly appreciate the uses of transformations, it is useful to understand how they are usually used in the *rendering pipeline*, that is, how transformations are used to render and model a 3D scene. Figure II.1 shows the transformations that are most commonly used in 3D graphics rendering. As we discuss later, these transformations are generally represented by $4 \times 4$ matrices acting on homogeneous coordinates. Figure II.1 shows the following elements that go into modelling and transforming a vertex:

**Vertex position:** We start with a vertex position $x$, usually as a point in 3-space, denoting a position for the vertex in *local coordinates*. For example, in the Ferris wheel example, to model a single chair of the Ferris wheel it might be the most convenient to model it as being centered at the origin. Vertices are specified relative to a “local coordinate system” for the chair. In this way, the chair can be modeled once without having to consider where the chairs will be placed in the final scene.

As we see later, it is common to use homogeneous coordinates, so that a 4-vector $x$ represents a position in 3-space. For OpenGL, these vertex positions $x$ would be stored in the VBO as vertex attributes.

**The Model matrix:** The Model matrix $M$ transforms vertex positions $x$ from local coordinates into *world coordinates* (also called *global coordinates*). The world coordinates can be viewed as the “real” coordinates. In the Ferris wheel example, multiple chairs are rendered at different positions and orientations on the Ferris wheel, since the chairs go around with the wheel plus they may rock back-and-forth. Each chair will have its own Model matrix $M$. The Model matrix transforms vertex positions $x$ from local coordinates into global coordinates. Usually, the Model matrix is a $4 \times 4$ matrix $M$, and the matrix-vector product $Mx$ gives the world coordinates for the point with local coordinates $x$.

Model matrix transformations usually represent “affine transformations”; these provide a flexible set of tools for positioning vertices, including methods for rotating, scaling, and re-shaping objects. The next sections will discuss affine transformations in some detail.

**The View matrix:** The View matrix $V$ transforms vertex positions from world coordinates into *view coordinates* (also called *camera coordinates*). The 3D scene is rendered from the viewpoint of some viewer or camera. The viewer is positioned at some viewpoint, and also has an orientation. This information defines the view coordinates. A typical convention is that the viewer is positioned at the origin of the view coordinate system, and looking
II. Transformations in 2-space

II.1. Transformations in 2-space

III.2. Transformations in 2-space

(Draft A.2.d)

33

down the negative z-axis, with the viewer’s y-axis pointing in the viewer’s “up” direction, and the viewer’s x-axis pointing in the viewer’s “rightward” direction.

Like the Model matrix $M$, the View matrix is a $4 \times 4$ matrix and typically represents an affine transformation. If $y = Mx$ describes a position in world coordinates, then $Vy$ is the position in view coordinates. This can be equivalently expressed as $(VM)x$. The matrix product $VM$ is also a $4 \times 4$ matrix, called the Modelview matrix. It is fairly common to work with just the Modelview matrix, and instead of using separate Model and View matrices.

The Projection matrix: The Projection matrix $P$ transforms vertex positions from view coordinates into screen coordinates. The screen coordinates give the $x,y$-coordinates of the position in the final image; OpenGL requires these to be in the range $[-1,1]$. The $z$-component of the screen coordinates are based on the distance of the point from the viewer. We call the $z$-component the “pseudo-distance”, as it is a non-linear function of actual distance from the viewer. The pseudo-distance is used for hidden surface computations (see Section I.4.4).

The Projection matrix is also a $4 \times 4$ matrix, and applies either an “orthographic projection” or a “perspective transformation.”

Perspective Division: The Model, View and Projection matrices are all $4 \times 4$ matrices operating on homogeneous representations of points in 3-space. The homogeneous coordinates are a 4-vector $(x,y,z,w)$ with an extra fourth component $w$. Division by $w$ gives the ordinary screen components as the 3-vector $(x/w, y/w, z/w)$. The values $x/w$ and $y/w$ are the $x$- and $y$-coordinates of the screen position and $z/w$ gives the pseudo-distance. All three of these values $x/w$, $y/w$ and $z/w$ need to be in the range $[-1,1]$: otherwise the point is not visible, either by virtue of being outside of the field of view or being with too close or too far from the viewer.

The mathematics behind all this is described later in this chapter. The use of homogeneous coordinates is particularly useful because it handles both translations and perspective at the same time.

The Model matrix, View matrix, and Perspective matrix are used by the vertex shader to transform vertex positions from local coordinates to screen coordinates. The Perspective Division occurs after the vertex shader has processed the vertices. The fragment shader is not able to alter the screen position, and in most applications does not alter the $z$-value (depth) value. In some applications, the fragment shader needs to use the world coordinates or the view coordinates of a point, for instance when calculating global lighting. (For this, see Chapter IV.) If so, they come from the vertex shader as shader-specific output values.
II.1 Transformations in 2-space

We start by discussing linear and affine transformations on a fairly abstract level. After that, we will give some examples of using transformations in OpenGL. We begin by considering affine transformations in 2-space since they are much simpler than transformations in 3-space. As we shall see, most of the important properties of affine transformations already apply in 2-space.

The $xy$-plane, denoted $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$, is the usual cartesian plane consisting of points $\langle x, y \rangle$. To avoid writing too many coordinates, we often use the vector notation $x$ for a point in $\mathbb{R}^2$, with the usual convention being that $x = \langle x_1, x_2 \rangle$, where $x_1, x_2 \in \mathbb{R}$. This notation is convenient, but potentially confusing, since we will use the same notation for vectors as for points.\footnote{Points and vectors in 2-space both consist of a pair of real numbers. The difference is that a point specifies a particular location, whereas a vector specifies a particular displacement, or change in location. That is to say, a vector is the difference of two points. Rather than adopting a confusing and non-standard notation that clearly distinguishes between points and vectors, we will instead follow the more common, but ambiguous, convention of using the same notation for points as for vectors.}

We write 0 for the origin, or zero vector. So, $0 = \langle 0, 0 \rangle$. We write $x + y$ and $x - y$ for the component-wise sum and difference of $x$ and $y$. A real number $\alpha \in \mathbb{R}$ is called a scalar, and the product of a scalar and a vector is defined by $\alpha x = \langle \alpha x_1, \alpha x_2 \rangle$.\footnote{In view of the distinction between points and vectors, it can be useful to form the sums and differences of two vectors, or of a point and a vector, or the difference of two points, but it is not generally useful to form the sum of two points. The sum or difference of two vectors is a vector. The sum or difference of a point and a vector is a point. The difference of two points is a vector. Likewise, a vector may be multiplied by a scalar, but it is less frequently appropriate to multiply a scalar and point. However, we gloss over these issues, and define the sums and products on all combinations of points and vectors. In any event, we frequently blur the distinction between points and vectors.} The magnitude or norm of a vector $x = \langle x_1, x_2 \rangle$ is equal to $||x|| = \sqrt{x_1^2 + x_2^2}$. A unit vector is a vector with magnitude equal to 1. See Appendix 1 for more basic facts about vectors.
II.1.1 Basic definitions

A transformation on \( \mathbb{R}^2 \) is any mapping \( A : \mathbb{R}^2 \to \mathbb{R}^2 \). That is to say, each point \( x \in \mathbb{R}^2 \) is mapped to a unique point, \( A(x) \), also in \( \mathbb{R}^2 \).

**Definition** Let \( A \) be a transformation. \( A \) is a linear transformation provided the following two conditions hold:

1. For all \( \alpha \in \mathbb{R} \) and all \( x \in \mathbb{R}^2 \), \( A(\alpha x) = \alpha A(x) \).
2. For all \( x, y \in \mathbb{R}^2 \), \( A(x + y) = A(x) + A(y) \).

Note that \( A(0) = 0 \) for any linear transformation \( A \). This follows from condition 1 with \( \alpha = 0 \).

We defined transformations as acting on a single point at a time, but of course a transformation also acts on arbitrary geometric objects by mapping the points in the object. Figure II.3 shows six examples of linear transformations and how they act on the “F” shape of Figure II.2. These include

- (a) \( \langle x, y \rangle \mapsto \langle x, y \rangle \). The identity transformation \( I \).
- (b) \( \langle x, y \rangle \mapsto \langle \frac{1}{2}x, \frac{1}{2}y \rangle \). The uniform scaling \( S_{\frac{1}{2}} \).
- (c) \( \langle x, y \rangle \mapsto \langle \frac{3}{2}x, \frac{1}{2} \rangle \). The nonuniform scaling \( S_{\langle \frac{3}{2}, \frac{1}{2} \rangle} \).
- (d) \( \langle x, y \rangle \mapsto \langle -y, x \rangle \). Rotation 90° counterclockwise, \( R_{90^\circ} \).
- (e) \( \langle x, y \rangle \mapsto \langle x + y, y \rangle \). A shearing transformation.
- (f) \( \langle x, y \rangle \mapsto \langle -y, -x \rangle \). Reflection across the line \( y = -x \).

**Exercise II.1** Verify that the five transformations \( A_1 - A_5 \) below are linear. Draw pictures showing how they transform the “F” shape.

- \( A_1 : \langle x, y \rangle \mapsto \langle -y, x \rangle \).
- \( A_2 : \langle x, y \rangle \mapsto \langle x, 2y \rangle \).
- \( A_3 : \langle x, y \rangle \mapsto \langle x - y, y \rangle \).
- \( A_4 : \langle x, y \rangle \mapsto \langle x, -y \rangle \).
- \( A_5 : \langle x, y \rangle \mapsto \langle -x, -y \rangle \).
Figure II.3 shows the uniform scaling $S_{\frac{1}{2}}$. For $\alpha$ any scalar, the uniform scaling $S_\alpha : (x, y) \mapsto (\alpha x, \alpha y)$ is the linear transformation which changes the sizes of object (centered at the origin) by a factor $\alpha$. Figure II.3c shows an example, of nonuniform scaling. In general, the nonuniform scaling $S_{(\alpha, \beta)} : (x, y) \mapsto (\alpha x, \beta y)$ scales $x$-coordinates by the factor $\alpha$ and $y$-coordinates by $\beta$. Figure II.3d shows a rotation by 90$^\circ$ in the counterclockwise direction. More generally, for $\theta$ a scalar, $R_\theta$ performs a rotation around the origin by $\theta$ radians in the counterclockwise direction. (A matrix formula for $R_\theta$ is given later in Equation II.2.)
One simple, but important, kind of non-linear transformation is a “translation,” which changes the position of objects by a fixed amount but does not change the orientation or shape of geometric objects.

**Definition** A transformation $A$ is a translation provided that there is a fixed $u \in \mathbb{R}^2$ such that $A(x) = x + u$ for all $x \in \mathbb{R}^2$.

The notation $T_u$ is used to denote this translation, thus $T_u(x) = x + u$.

The composition of two transformations $A$ and $B$ is the transformation which is computed by first applying $B$ and then applying $A$. This transformation is denoted $A \circ B$, or just $AB$, and satisfies $(A \circ B)(x) = A(B(x))$.

The identity transformation maps every point to itself. The inverse of a transformation $A$ is the transformation $A^{-1}$ such that $A \circ A^{-1}$ and $A^{-1} \circ A$ are both the identity transformation. Not every transformation has an inverse, but when $A$ is one-to-one and onto, the inverse transformation $A^{-1}$ always exists.

Note that the inverse of $T_u$ is $T_{-u}$.

**Definition** A transformation $A$ is affine provided it can be written as the composition of a translation and a linear transformation. That is to say, provided it can be written in the form $A = T_u B$ for some $u \in \mathbb{R}^2$ and some linear transformation $B$.

In other words, a transformation $A$ is affine if it equals

$$A(x) = B(x) + u,$$

with $B$ a linear transformation and $u$ a point.

Since it is permitted that $u = 0$, every linear transformation is affine. However, not every affine transformation is linear. In particular, if $u \neq 0$, then the transformation (II.1) is not linear, since it does not map $0$ to $0$.

**Proposition II.1** Let $A$ be an affine transformation. The translation vector $u$ and the linear transformation $B$ are uniquely determined by $A$.

**Proof** First, let’s see how to determine $u$ from $A$. We claim that in fact $u = A(0)$. This is proved by the following equalities:

$$A(0) = T_u(B(0)) = T_u(0) = 0 + u = u.$$  

Then $B = T_u^{-1}A = T_{-u}A$, so $B$ is also uniquely determined. $\square$

**Corollary II.2** An affine transformation $A$ is linear iff $A(0) = 0$.

Another useful fact is that linear transformations and affine transformations are preserved under composition:
Theorem II.3 Let $A$ and $B$ be transformations of $\mathbb{R}^2$.

a. If $A$ and $B$ are linear, then $A \circ B$ is a linear transformation.

b. If $A$ and $B$ are affine transformations, then $A \circ B$ is an affine transformation.

Proof First suppose $A$ and $B$ are linear. Then

$$(A \circ B)(\alpha x) = A(B(\alpha x)) = A(\alpha B(x)) = \alpha A(B(x)) = \alpha (A \circ B)(x),$$

and

$$(A \circ B)(x+y) = A(B(x+y)) = A(B(x)) + A(B(y)) = (A \circ B)(x) + (A \circ B)(y).$$

That shows $A \circ B$ is linear.

Now suppose $A$ and $B$ are affine, with $A = T_u \circ C$ and $B = T_v \circ D$, where $C$ and $D$ are linear transformations. Then

$$(A \circ B)(x) = A(D(x) + v) = A(D(x)) + A(v) = C(D(x)) + u + A(v).$$

Therefore $(A \circ B)(x) = E(x) + w$ where $E = C \circ D$ is linear and $w = u + A(v)$. Thus $A \circ B$ is affine.

II.1.2 Matrix representation of linear transformations

The above mathematical definition of linear transformations is stated rather abstractly. However, there is a very concrete way to represent a linear transformation $A$, namely as a $2 \times 2$ matrix.

Define $i = (1, 0)$ and $j = (0, 1)$. The two vectors $i$ and $j$ are the unit vectors which are aligned with the $x$-axis and $y$-axis, respectively. Any vector $x = \langle x_1, x_2 \rangle$ can be uniquely expressed as a linear combination of $i$ and $j$, namely, as $x = x_1i + x_2j$.

Let $A$ be a linear transformation. Let $u = \langle u_1, u_2 \rangle = A(i)$ and $v = \langle v_1, v_2 \rangle = A(j)$. Then, by linearity, for any $x \in \mathbb{R}^2$,

$$A(x) = A(x_1i + x_2j) = x_1A(i) + x_2A(j) = x_1u + x_2v$$

$$= \langle u_1x_1 + v_1x_2, u_2x_1 + v_2x_2 \rangle.$$

Let $M$ be the matrix $\begin{pmatrix} u_1 & v_1 \\ u_2 & v_2 \end{pmatrix}$. Then,

$$M \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} u_1 & v_1 \\ u_2 & v_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} u_1x_1 + v_1x_2 \\ u_2x_1 + v_2x_2 \end{pmatrix},$$

so the matrix $M$ computes the same thing as the transformation $A$. We call $M$ the matrix representation of $A$.

We have just shown that every linear transformation $A$ is represented by some matrix. Conversely, it is easy to check that every matrix represents
a linear transformation. Thus, it is reasonable to henceforth think of linear transformations on $\mathbb{R}^2$ as being the same as $2 \times 2$ matrices.

One notational complication is that a linear transformation $A$ operates on points $x = (x_1, x_2)$, whereas a matrix $M$ acts on column vectors. It would be convenient, however, to use both of the notations $A(x)$ and $Mx$. To make both notations be correct, we adopt the following rather special conventions about the meaning of angle brackets and the representation of points as column vectors:

Notation The point or vector $(x_1, x_2)$ is identical to the column vector $egin{pmatrix} x_1 \\ x_2 \end{pmatrix}$. So ‘point,’ ‘vector,’ and ‘column vector’ all mean the same thing. A column vector is the same as a single column matrix. A row vector is a vector of the form $(x_1, x_2)$; i.e., a matrix with a single row.

A superscript ‘$T$’ denotes the matrix transpose operator. In particular, the transpose of a row vector is a column vector, and vice-versa. Thus, $x^T$ equals the row vector $(x_1, x_2)$.

It is a simple, but important, fact that the columns of a matrix $M$ are the images of $i$ and $j$ under $M$. That is to say, the first column of $M$ is equal to $Mi$ and the second column of $M$ is equal to $Mj$. This gives an intuitive method of constructing a matrix for a linear transformation. For example, consider the reflection across the line $y = x$ shown in Figure II.3f. Just by the shape of the transformed “F”, we see that it maps $i$ to $(0, -1)$ and $j$ to $(-1, 0)$. Taking these vectors as the first and second columns of the matrix, we get immediately that this reflection is represented by the matrix $\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$.

Exercise II.2 Determine the $2 \times 2$ matrices representing the other five transformations shown in Figure II.3.

A more complicated use of this method for determining matrices is is shown in the next example.

Example: Let $M = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix}$. The action of $M$ on the “F” is shown in Figure II.4. We’ll use a purely geometric method to find the matrix representation of its inverse $M^{-1}$. For this, it is enough to determine $M^{-1}i$ and $M^{-1}j$ since they will be the columns of $M^{-1}$. Looking at the righthand graph in Figure II.4, there are two vectors drawn with dotted lines: one of these is the vector $i = (1, 0)$ and the other is $2j = (0, 2)$. The preimages of these two vectors are drawn as dotted vectors on the lefthand graph are equal to $(1, -1/2)$ and $(0, 1)$. From this, we get immediately that

$$M^{-1}i = M^{-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ -1/2 \end{pmatrix} \quad \text{and} \quad M^{-1}j = \frac{1}{2} M^{-1} \begin{pmatrix} 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1/2 \end{pmatrix}.$$

Therefore, $M^{-1}$ is equal to $\begin{pmatrix} 1 & 0 \\ -1/2 & 1/2 \end{pmatrix}$.
Figure II.4: An “F” shape transformed by a linear transformation from the example. The dotted vectors in the righthand graph are the vectors $\mathbf{i}$ and $2\mathbf{j}$. The horizontal dotted vector going from $\langle 0, 1 \rangle$ to $\langle 1, 1 \rangle$ is equal to $\mathbf{i}$ since $\langle 1, 1 \rangle - \langle 0, 1 \rangle = \langle 1, 0 \rangle$. Similarly, the vertical dotted vector going from $\langle 1, 1 \rangle$ to $\langle 1, 3 \rangle$ is equal to $2\mathbf{j}$. The preimages of these two vectors are shown as dotted vectors in the lefthand graph.

The example illustrates a rather intuitive way to find the inverse of a matrix, but it depends on being able to visually find preimages of $\mathbf{i}$ and $\mathbf{j}$. It is worth explaining in a little more detail how to visualize the preimages above. Let’s consider the preimage of $\mathbf{j}$. For this, we noted that the two vertices in the image $\langle 1, 3 \rangle$ and $\langle 1, 1 \rangle$ have difference equal to $\langle 0, 2 \rangle = 2\mathbf{j}$, which is a scalar multiple of $\mathbf{j}$. This is shown as the vertical dotted vector in the righthand side of Figure II.4. And, it is clear from the picture that $A(\langle 1, 1 \rangle) = \langle 1, 3 \rangle$ and $A(\langle 1, 0 \rangle) = \langle 1, 1 \rangle$. The latter vector is shown as a dotted vector in the lefthand side of Figure II.4. By linearity, we have

$$A(\langle 1, 1 \rangle - \langle 1, 0 \rangle) = A(\langle 1, 1 \rangle) - A(\langle 1, 0 \rangle) = \langle 1, 3 \rangle - \langle 1, 1 \rangle = \langle 0, 2 \rangle = 2\mathbf{j}.$$ 

From this, $A(\langle 0, 1 \rangle) = 2\mathbf{j}$. A similar method is used to visualize $A^{-1}(\mathbf{i})$.

One can instead compute the inverse of a $2 \times 2$ matrix by the well-known formula

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{\det(M)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix},$$

where $\det(M) = ad - bc$ is the determinant of $M$.

Exercise II.3  Figure II.5 shows an affine transformation acting on an “F”.  (a) Is this a linear transformation? Why or why not? (b) Express this affine transformation in the form $\mathbf{x} \mapsto M\mathbf{x} + \mathbf{u}$, by explicitly giving $M$ and $\mathbf{u}$.

A rotation is a transformation which rotates the points in $\mathbb{R}^2$ by a fixed angle around the origin. Figure II.6 shows the effect of a rotation of $\theta$ degrees
II.1.3. Rigid transformations and rotations

\[ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \]

\[ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \]

\[ \Rightarrow \]

Figure II.5: An affine transformation acting on an “F.”

Figure II.6: Effect of a rotation through angle \( \theta \). The origin 0 is held fixed by the rotation.

In the counter-clockwise (CCW) direction. As shown in Figure II.6, the images of \( i \) and \( j \) under a rotation of \( \theta \) degrees are \( \langle \cos \theta, \sin \theta \rangle \) and \( \langle -\sin \theta, \cos \theta \rangle \). Therefore, a counter-clockwise rotation through an angle \( \theta \) is represented by the matrix

\[
R_\theta = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}.
\]

(II.2)

**Conventions on row and column vectors, and transposes.** The conventions adopted in this book are that points in space are represented by column vectors, and linear transformations with matrix representation \( M \) are computed as \( Mx \). Thus, our matrices multiply on the left. Unfortunately, this convention is not universally followed and it is also common in computer graphics applications to use row vectors for points and vectors, and to use matrix representations which act on the right. That is to say, many workers in computer graphics use a row vector to represent a point: instead of using \( x \), they use the row vector \( x^T \). Then, instead of multiplying on the left with \( M \), they multiply on the right with its transpose \( M^T \). Since \( x^T M^T \) equals \( (Mx)^T \) this has the same meaning. Similarly, when multiplying matrices to compose transformations, the fact that \( (MN)^T = N^T M^T \) means that when working with transposed matrices, one has to reverse the order of the multiplications.
II.1.3 Rigid transformations and rotations

A rigid transformation is a transformation which only repositions objects, leaving their shape and size unchanged. If the rigid transformation also preserves the notions of “clockwise” versus “counter-clockwise,” then it is called orientation preserving.

**Definition** A transformation is called rigid if and only if it preserves both:

1. Distances between points.
2. Angles between lines.\(^3\)

A transformation is said to be orientation preserving if it preserves the direction of angles; i.e., if a counter-clockwise direction of movement stays counter-clockwise after being transformed by \(A\).

Rigid, orientation preserving transformations are widely used. One application of these transformations is in animation: the position and orientation of a moving rigid body can be described by a time-varying transformation \(A(t)\). This transformation \(A(t)\) will be rigid and orientation preserving, provided the body does not deform or change size or shape.

The two most common examples of rigid, orientation preserving transformations are rotations and translations. Another example of a rigid, orientation preserving transformation is a “generalized rotation” which performs a rotation around an arbitrary center point. We prove below that every rigid, orientation preserving transformation over \(\mathbb{R}^2\) is either a translation or a generalized rotation.

**Exercise II.4** Which of the five linear transformations in exercise II.1 on page 35 are rigid? Which ones are both rigid and orientation preserving?

For linear transformations, an equivalent definition of rigid transformation is that a linear transformation \(A\) is rigid if and only if it preserves dot products. That is to say, if and only if, for all \(x, y \in \mathbb{R}^2\), \(x \cdot y = A(x) \cdot A(y)\). To see that this preserves distances, recall that \(|x|^2 = x \cdot x\) is the square of the magnitude of \(x\), or the square of \(x\)’s distance from the origin. Thus, \(|x|^2 = x \cdot x = A(x) \cdot A(x) = |A(x)|^2\). From the definition of the dot product as \(x \cdot y = |x| \cdot |y| \cos \theta\), where \(\theta\) is the angle between \(x\) and \(y\), the transformation \(A\) must also preserve angles between lines.

**Exercise II.5** Prove more fully the assertions in the previous paragraph by showing that if \(A(0) = 0\) and \(A\) preserves distances between points, then \(A\) preserves dot products. Conversely, prove that if \(A\) preserves dot products, then \(A(0) = 0\) and \(A\) preserves distances between points.

\(^3\)Strictly speaking, the second condition could be omitted from the definition of “rigid”. This is because the SSS Theorem (Side-Side-Side Theorem) of geometry implies that if distances are preserved, then also angles are preserved.
II.1.3. Rigid transformations and rotations (Draft A.2.d)

Exercise II.6 Let \( M = (u, v) \), i.e., \( M = \begin{pmatrix} u_1 & v_1 \\ u_2 & v_2 \end{pmatrix} \). Show that the linear transformation represented by the matrix \( M \) is rigid if and only if \( ||u|| = ||v|| = 1 \), and \( u \cdot v = 0 \). Prove that if \( M \) represents a rigid transformation, then \( \det(M) = \pm 1 \).

A matrix \( M \) of the type in the previous exercise is called an orthonormal matrix.

Exercise II.7 Prove that the linear transformation represented by the matrix \( M \) is rigid if and only if \( M^T = M^{-1} \).

Exercise II.8 Show that the linear transformation represented by the matrix \( M \) is orientation preserving if and only if \( \det(M) > 0 \). [Hint: Let \( M = (u, v) \). Let \( u' \) be \( u \) rotated counter-clockwise 90°. Then \( M \) is orientation preserving if and only if \( u' \cdot v > 0 \).]

Theorem II.4 Every rigid, orientation preserving, linear transformation is a rotation.

The converse to Theorem II.4 holds too: every rotation is obviously a rigid, orientation preserving, linear transformation.

Proof Let \( A \) be a rigid, orientation preserving, linear transformation. Let \( \langle a, b \rangle = A(i) \). By rigidity, \( A(i) \cdot A(i) = a^2 + b^2 = 1 \). Also, \( A(j) \) must be the vector which is obtained by rotating \( A(i) \) counter-clockwise 90°; thus, \( A(j) = (-b, a) \), as shown in Figure II.7.

Therefore, the matrix \( M \) which represents \( A \) is equal to \( \begin{pmatrix} a & -b \\ b & a \end{pmatrix} \). Since \( a^2 + b^2 = 1 \), there must be an angle \( \theta \) such that \( \cos \theta = a \) and \( \sin \theta = b \), namely, either \( \theta = \cos^{-1} a \) or \( \theta = -\cos^{-1} a \). From Equation (II.2), we see that \( A \) is a rotation through the angle \( \theta \).

Many programming languages, including C and C++, have a two parameter version of the arctangent function which lets you compute the rotation angle as \( \theta = \text{atan2}(b, a) \).

This is often more robust, and more useful, than using C++’s arctangent \( \text{atan} \), arcsine (\( \text{asin} \)) or arccosine (\( \text{acos} \)). In GLSL for shader programming, the arctangent function is named just \( \text{atan} \) instead of \( \text{atan2} \), and can take either one or two inputs.

We can improve on Theorem II.4 by weakening the assumption that \( A \) is linear to assume only that \( A(0) = 0 \).

Theorem II.5 Suppose \( A \) is a rigid, orientation preserving transformation with \( A(0) = 0 \). Then \( A \) is a rotation.
Figure II.7: A rigid, orientation preserving, linear transformation acting on the unit vectors \(i\) and \(j\).

**Proof** As in the proof of Theorem II.4, there are scalars \(a\) and \(b\) such that \(a^2 + b^2 = 1\) and such that \(A(i) = (a, b)\) and \(A(j) = (-b, a)\). (That part of the proof used only the rigid and orientation properties.) Let \(\theta\) satisfy \(\cos(\theta) = a\) and \(\sin(\theta) = b\). Define \(B = R_{-\theta} \circ A\). Then \(B(0) = 0\), and \(B(i) = i\) and \(B(j) = j\). Furthermore, as a composition of two rigid and orientation preserving transformations, \(B\) is also rigid and orientation preserving.

It is not hard to verify that every point \(x \in \mathbb{R}^2\) is uniquely characterized by its distances from the three points \(0, i\) and \(j\). Therefore, since \(B\) is rigid and maps those three points to themselves, \(B\) must map every point \(x\) in \(\mathbb{R}^2\) to itself. In other words, \(B\) is the identity. Therefore, \(A = (R_{-\theta})^{-1} = R_{\theta}\), and so is a rotation.

Theorem II.5, and the definition of affine transformations, gives the following characterization.

**Corollary II.6** Every rigid, orientation preserving transformation \(A\) can be (uniquely) expressed as the composition of a translation and a rotation. Hence \(A\) is an affine transformation.

Corollary II.6 is proved by applying Theorem II.5 to the rigid, orientation preserving transformation \(T_{-vecu} \circ A\) where \(u = A(0)\).

**Definition** A *generalized rotation* is a transformation which holds a center point \(u\) fixed and rotates all other points around \(u\) through a fixed angle \(\theta\). This transformation is denoted \(R_{\theta}^u\).

An example of a generalized rotation is given in Figure II.8. Clearly, a generalized rotation is a rigid, orientation preserving, affine transformation.

One way to perform a generalized rotation is to first apply a translation to move the point \(u\) to the origin, then rotate around the origin, and then translate the origin back to \(u\). Thus, the generalized rotation \(R_{\theta}^u\) can be expressed as

\[
R_{\theta}^u = T_u R_\theta T_{-u}.
\]  

(II.3)

You should convince yourself that formula (II.3) is correct.
II.1.3. Rigid transformations and rotations

\[ \begin{pmatrix} \langle 0, 1 \rangle & \langle 1, 1 \rangle & \langle 1, 0 \rangle \\ \langle 0, 0 \rangle & \langle 0, -1 \rangle & \langle 0, 3 \rangle \end{pmatrix} \]

\[ \theta \]

\( x \)

\( y \)

\[ \langle 0, 3 \rangle \]

\[ \langle 0, 0 \rangle \]

\[ \langle 0, 1 \rangle \]

\[ \langle 1, 1 \rangle \]

\[ \langle 1, 0 \rangle \]

\[ \langle 0, -1 \rangle \]

\begin{proof}

Let \( A \) be a rigid, orientation preserving transformation. By Corollary II.6, \( A \) is affine. Let \( \mathbf{u} = A(\mathbf{0}) \). If \( \mathbf{u} = \mathbf{0} \), \( A \) is actually a linear transformation, and Theorem II.4 implies that \( A \) is a rotation. So suppose \( \mathbf{u} \neq \mathbf{0} \). It will suffice to prove that either \( A \) is a translation or there is some point \( \mathbf{v} \in \mathbb{R}^2 \) which is a fixed point of \( A \), i.e., such that \( A(\mathbf{v}) = \mathbf{v} \). This is sufficient since, if there is a fixed point \( \mathbf{v} \), then the reasoning of the proofs of Theorems II.4 and II.5 shows that \( A \) is a generalized rotation around \( \mathbf{v} \).

Let \( L \) be the line that contains the two points \( \mathbf{0} \) and \( \mathbf{u} \). We consider two cases. First, suppose that \( A \) maps \( L \) to itself. By rigidity, and by choice of \( \mathbf{u} \), \( A(\mathbf{u}) \) is distance \( ||\mathbf{u}|| \) from \( \mathbf{u} \), so we must have either \( A(\mathbf{u}) = \mathbf{u} + \mathbf{u} \) or \( A(\mathbf{u}) = \mathbf{0} \). If \( A(\mathbf{u}) = \mathbf{u} + \mathbf{u} \), then \( A \) must be the translation \( T_\mathbf{u} \). This follows since, again by the rigidity of \( A \), every point \( \mathbf{x} \in L \) must map to \( \mathbf{x} + \mathbf{u} \) and, by the rigidity and orientation preserving properties, the same holds for every point not on \( L \). On the other hand, if \( A(\mathbf{u}) = \mathbf{0} \), then rigidity implies that \( \mathbf{v} = \frac{1}{2} \mathbf{u} \) is a fixed point of \( A \), and thus \( A \) is a generalized rotation around \( \mathbf{v} \).

Second, suppose that the line \( L \) is mapped to a different line \( L' \). Let \( L' \) make an angle of \( \theta \) with \( L \), as shown in Figure II.9. Since \( L' \neq L \), \( \theta \) is nonzero and is not a multiple of 180°. Let \( L_2 \) be the line perpendicular to \( L \) at the point \( \mathbf{0} \), and let \( L'_2 \) be the line perpendicular to \( L \) at the point \( \mathbf{u} \). Note that \( L_2 \) and \( L'_2 \) are parallel. Now let \( L_3 \) be the line obtained by rotating \( L_2 \)
Figure II.9: Finding the center of rotation. The point $v$ is fixed by the rotation.

around the origin through a clockwise angle of $\theta/2$, and let $L'_3$ be the line obtained by rotating $L'_2$ around the point $u$ through a counter-clockwise angle of $\theta/2$. Since $A$ is rigid and orientation preserving, and the angle between $L$ and $L_3$ equals the angle between $L'$ and $L'_3$, the line $L_3$ is mapped to $L'_3$ by $A$. The two lines $L_3$ and $L'_3$ are not parallel and intersect in a point $v$. By the symmetry of the constructions, $v$ is equidistant from $0$ and $u$. Therefore, again by rigidity, $A(v) = v$. It follows that $A$ is the generalized rotation $R_v^\theta$, which performs a rotation through an angle $\theta$ around the center $v$. $\blacksquare$

II.1.4 Homogeneous coordinates

Homogeneous coordinates provide a method of using a triple of numbers $\langle x, y, w \rangle$ to represent a point in $\mathbb{R}^2$.

**Definition** If $x, y, w \in \mathbb{R}$ and $w \neq 0$, then $\langle x, y, w \rangle$ is a homogeneous coordinate representation of the point $\langle x/w, y/w \rangle \in \mathbb{R}^2$.

Note that any given point in $\mathbb{R}^2$ has many representations in homogeneous coordinates. For example, the point $\langle 2, 1 \rangle$ can be represented by any of the following sets of homogeneous coordinates: $\langle 2, 1, 1 \rangle$, $\langle 4, 2, 2 \rangle$, $\langle 6, 3, 3 \rangle$, $\langle -2, -1, -1 \rangle$, etc. More generally, the triples $\langle x, y, w \rangle$ and $\langle x', y', w' \rangle$ represent the same point in homogeneous coordinates if and only if there is a nonzero scalar $\alpha$ such that $x' = \alpha x$, $y' = \alpha y$ and $w' = \alpha w$.

So far, we have only specified the meaning of the homogeneous coordinates $\langle x, y, w \rangle$ when $w \neq 0$, since the definition of the meaning of $\langle x, y, w \rangle$ required dividing by $w$. However, we shall see in Section II.1.8 below that when $w = 0$, $\langle x, y, w \rangle$ is the homogeneous coordinate representation of a “point at infinity.” (Alternatively, graphics software such as OpenGL will sometimes
II.1.5 Matrix representation of an affine transformation

Recall that any affine transformation \( A \) can be expressed as a linear transformation \( B \) followed by a translation \( T_u \), that is, \( A = T_u \circ B \). Let \( M \) be a \( 2 \times 2 \) matrix representing \( B \) and suppose

\[
M = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \text{and} \quad u = \begin{pmatrix} e \\ f \end{pmatrix}.
\]

Then the mapping \( A \) can be defined by

\[
\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto M \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} ax_1 + bx_2 + e \\ cx_1 + dx_2 + f \end{pmatrix}.
\]

Now define \( N \) to be the \( 3 \times 3 \) matrix

\[
N = \begin{pmatrix} a & b & e \\ c & d & f \\ 0 & 0 & 1 \end{pmatrix}.
\]

Using the homogeneous representation \( \langle x_1, x_2, 1 \rangle \) of \( \langle x_1, x_2 \rangle \), we see that

\[
N \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & e \\ c & d & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} ax_1 + bx_2 + e \\ cx_1 + dx_2 + f \\ 1 \end{pmatrix}.
\]

The effect of \( N \) acting on \( \langle x, y, 1 \rangle \) is identical to the effect of the affine transformation \( A \) acting on \( \langle x, y \rangle \). The only difference is that the third coordinate of “1” is being carried around. More generally, for any other homogeneous representation of the same point, \( \langle \alpha x_1, \alpha x_2, \alpha \rangle \) with \( \alpha \neq 0 \), the effect of multiplying by \( N \) is

\[
N \begin{pmatrix} \alpha x_1 \\ \alpha x_2 \\ \alpha \end{pmatrix} = \begin{pmatrix} \alpha(ax_1 + bx_2 + e) \\ \alpha(cx_1 + dx_2 + f) \\ \alpha \end{pmatrix}.
\]
which is another representation of the point $A(x)$ in homogeneous coordinates. Thus, the $3 \times 3$ matrix $N$ provides a faithful representation of the affine map $A$ in that, when working with homogeneous coordinates, multiplying by the matrix $N$ provides exactly the same results as applying the transformation $A$. Further, $N$ acts consistently on different homogeneous representations of the same point.

The method used to obtain $N$ from $A$ was completely general, and therefore any affine transformation can be represented as a $3 \times 3$ matrix which acts on homogeneous coordinates. So far, we have used only matrices that have the bottom row $(0 \ 0 \ 1)$; these matrices are sufficient for representing any affine transformation. In fact, an affine transformation may henceforth be viewed as being identical to a $3 \times 3$ matrix that has bottom row $(0 \ 0 \ 1)$.

When we discuss perspective transformations, which are more general than affine transformations, it will be necessary to have other values in the bottom row of the matrix.

**Exercise II.10** Figure II.10 shows an affine transformation acting on an “F.”

(a) Is this a linear transformation? Why or why not? (b) Give a $3 \times 3$ matrix which represents the affine transformation.

*Hint: In this case, the easiest way to find the matrix is to split the transformation into a linear part and a translation. Then consider what the linear part does to the vectors $\mathbf{i}$ and $\mathbf{j}.*

For the next exercise, it is not necessary to invert a $3 \times 3$ matrix. Instead, note that if a transformation is defined by $\mathbf{y} = A\mathbf{x} + \mathbf{u}$, then its inverse is $\mathbf{x} = A^{-1}\mathbf{y} - A^{-1}\mathbf{u}$.

**Exercise II.11** Give the $3 \times 3$ matrix which represents the inverse of the transformation in the previous exercise.

**Exercise II.12** Give an example of how two different $3 \times 3$ homogeneous matrices can represent the same affine transformation. *[Hint: the bottom row can contain 0 0 $\alpha$.]*
II.1.6 Two dimensional transformations for computer graphics

This section and the next take a break from the mathematical theory of affine transformations and discuss how they are used in computer graphics when forming the Model matrix $M$. Recall that the Model matrix is used for positioning objects in space. The Model matrix will operate on homogeneous coordinates, and thus can be used to apply translations, rotations, uniform and non-uniform scalings, reflections, and in fact any affine transformation. This section will only discuss the intuitions, not the details, of how this is done. In particular, we illustrate the ideas with a simple example in 2-space, whereas OpenGL programs generally work in 3-space. This section does not give any actual OpenGL code; instead it gives only high-level pseudo-code. Sections II.2.3 through refsec:solarsystemCode describe how this kind of pseudo-code can be translated into OpenGL code.

The purpose of the Model matrix $M$ is to hold a homogeneous matrix representing an affine transformation. We shall therefore think of $M$ as being a $3 \times 3$ matrix acting on homogeneous representations of points in 2-space. (In actual practice, $M$ is a $4 \times 4$ matrix operating on points in 3-space.) We illustrate using Model matrices to render the “F” shapes shown in Figures II.11 and II.12.

Figure II.11 shows an example of how rotation and translation transformation are not commutative; in other words, that different orders of operations can yield different results. Define two transformations $A_1$ and $A_2$ by

$$A_1 = T(\ell, 0) \circ R_\theta \quad \text{and} \quad A_2 = R_\theta \circ T(\ell, 0). \quad (II.4)$$

It is can also apply perspective transformations, but this is usually not so useful for the Model matrix.
The two transformed “F” shapes $A_1(F)$ and $A_2(F)$ are shown in Figure II.11. Namely, the “F” shape is modeled in standard position as shown in Figure II.2, by drawing lines joining the five points (0, −1), (0, 0), (0, 1), (1, 0) and (1, 1). Transforming the points in this F in standard position by $A_1$ or by $A_2$ gives the “F” shapes $A_1(F)$ and $A_2(F)$. The lower “F” shape (the one on the $x$-axis) is $A_1(F)$; the upper one is $A_2(F)$.

To understand this, first observe that $R_θ(F)$ is an “F” shape still positioned at the origin, but rotated counter-clockwise by the angle $θ$. Then $A_1(F) = T_{(ℓ,0)}(R_θ(F))$ is this rotated “F” shape translated distance $ℓ$ along the positive $x$-axis; so $A_1(F)$ is the rotated “F” shape positioned at $(ℓ, 0)$. On the other hand, $T_{(ℓ,0)}(F)$ is an upright “F” shape positioned at $(ℓ, 0)$; and $A_2(F)$ is this “F” shape rotated around the origin, so as to be positioned at $(ℓ \cos θ, ℓ \sin θ)$.

The intuition behind forming $A_1(F)$ and $A_2(F)$ is that transformations are applied in right-to-left order. For $A_1(F) = (T_{(ℓ,0)} \circ R_θ)(F)$, this means we first apply the rotation $R_θ$ to the $F$, and then apply the translation $T_{(ℓ,0)}$. Similarly, for $A_2(F)$, the translation is applied first and the rotation is applied to the result. The pseudo-code for this is as follows. We use the notation “$M_1$” and “$M_2$” instead of “$A_1$” and “$A_2$” to indicate matrices representing the transformations. Likewise, $R_θ$ and $T_{(ℓ,0)}$ denote the matrices for these transformations acting on homogeneous coordinates.

```
Set $M_1 = Identity; // Identity matrix
Set $M_1 = M_1 \cdot T_{(ℓ,0)}; // Multiply on the right
Set $M_1 = M_1 \cdot R_θ; // Multiply on the right
Render $M_1(F); // Render the lower F
Set $M_2 = Identity; // Identity matrix
Set $M_2 = M_2 \cdot R_θ; // Multiply on the right
Set $M_2 = M_2 \cdot T_{(ℓ,0)}; // Multiply on the right
Render $M_2(F); // Render the upper F
```

This pseudo-code generates the “F” shapes of Figure II.11. The matrix multiplication commands update the Model matrix by multiplying with a rotation or translation on the right. This means that if you read the program forward, the “F” shape is transformed by these operations in reverse order. At times this can seem counterintuitive, but it can help with hierarchically modeling a scene.

Now let’s look at the two “F” shapes shown in Figure II.12. We’ll define two transformations represented by matrices $M_3$ and $M_4$, so that $M_3(F)$ generates the upper “F” shape in Figure II.12 and $M_4(F)$ generates the lower one. We claim that for the upper “F”, the matrix $M_3$ should represent the transformation

$$R_θ \circ T_{(ℓ,0)} \circ T_{(0, r+1)}.$$  \hfill (II.5)

This is illustrated in Figure II.13, where three F’s are shown. The first one, placed above the origin is the shape $T_{(0, r+1)}(F)$; namely, is the “F” shape translated up the positive $y$-axis a distance $r + 1$. The second one is
II.1.6. Two dimensional transformations for computer graphics (Draft A.2.d) 51

Figure II.12: The results of drawing the “F” shape with two different Model matrices. The dotted lines are not rendered by the code and are present only to indicate the placement of the shapes. This figure is used for the pseudo-code on page 52.

\[ T_{(\ell,0)} \circ T_{(0,r+1)}(F); \] namely it is further translated distance \( \ell \) in the direction of the positive \( x \)-axis. The third one is \( R_\theta \circ T_{(\ell,0)} \circ T_{(0,r+1)}(F); \) namely the second “F” is rotated around the origin.

The same kind of reasoning shows that the lower “F” shape is equal to \( M_4(F) \) where matrix \( M_4 \) represents the transformation

\[ R_\theta \circ T_{(\ell,0)} \circ R_\pi \circ T_{(0,r+1)}. \] (II.6)

You should try working this out to verify that this transformation is correct. Note that \( R_\pi \) performs a 180° rotation around the origin, so that \( R_\pi \circ T_{(0,r+1)}(F) \) is an upside down \( F \) with its center vertex positioned at \( (0, -(r+1)). \)

Here is some possible pseudo-code for rendering the two “F” shapes of Figure II.12:
Figure II.13: The intermediate stages of placing the upper “F” shape of Figure II.12. The two upright “F” shapes are $T_{(0,r+1)}(F)$ and $T_{(ℓ,0)} \circ T_{(0,r+1)}(F)$. The “F” is never actually placed in these two intermediate positions; rather this just helps us figure out what the Model matrix should be.

\[
\begin{align*}
&\text{Set } M_0 = \text{Identity;} \\
&\text{Set } M_0 = M_0 \cdot R_θ; \\
&\text{Set } M_0 = M_0 \cdot T_{(ℓ,0)}; \\
&\text{Set } M_3 = M_0 \cdot T_{(0,r+1)}; & \quad \text{// Calculate } M_3 \text{ from } M_0 \\
&\text{Render } M_3(F); & \quad \text{// Render the lower F} \\
&\text{Set } M_4 = M_0 \cdot R_π; & \quad \text{// Calculate } M_4 \text{ from } M_0 \\
&\text{Set } M_4 = M_4 \cdot T_{(0,r+1)}; \\
&\text{Render } M_4(F); & \quad \text{// Render the upper F}
\end{align*}
\]

As before, the matrix multiplication commands update the Model matrix by multiplying with a rotation or translation on the right, the “F” shapes are transformed by these operations in the reverse of the order given the code. This example shows how it can help with hierarchically rendering scenes. For example, changing just the second line (the multiplication by $R_θ$) causes both “F” shapes are rotated together as a group around the origin. Similarly the translation by $T_{(ℓ,0)}$ moves both “F” shapes together.

**Exercise II.13** Consider the transformation shown in Figure II.14.

a. Give pseudo-code that will draw the “F” as shown on the right-hand side of Figure II.14.
II.1.7 Another outlook on composing transformations

So far we have discussed the actions of transformations (rotations and translations) as acting on the objects being drawn, and viewed them as being applied in reverse order from the order given in the (pseudo)code. However, it is also possible to view transformations as acting not on objects, but instead on coordinate systems. In this alternative viewpoint, one thinks of the transformations acting on local coordinate systems (and within the local coordinate system), and now the transformations are applied in the same order as given in the code.

To illustrate the alternate view of transformations, consider the “F” shape that is shown uppermost in Figures II.12 and II.13. Recall this is transformed by

\[ R_\theta \circ T_{(\ell,0)} \circ T_{(0,r+1)}. \]

Figure II.16 shows four local coordinate systems placed a different positions and orientations on the xy-plane. These axiom systems are denoted \(x^0, y^0\) and

Exercise II.14 Repeat the previous exercise for the affine transformation shown in Figure II.15. You may use \(S_{(a,b)}\) with one of \(a\) or \(b\) negative to perform a reflection.

b. Give the 3×3 homogeneous matrix which represents the affine transformation shown in the figure.
The successive transformations are carried out relative to the previous local coordinate system. $x^1, y^1$ and $x^2, y^2$ and $x^3, y^3$. The first set of local coordinates $x^0$ and $y^0$ is identical to the original $xy$-coordinate axes.

The second set of local coordinates is $x^1$ and $y^1$. These are obtained by applying the rotation $R_\theta$ to the $x^0, y^0$ coordinate system, namely the coordinate axes are rotated counter-clockwise by the angle $\theta$.

The third set of local coordinates is $x^2$ and $y^2$. This is obtained by translating the $x^1, y^1$ coordinate system by $(0, \ell)$, doing the translation relative to the $x^1, y^1$ coordinate system. The fourth set of local coordinates is and is obtained by an additional translation by $(0, r+1)$, this relative to the local coordinate system of $x^2$ and $y^2$. The upper "F" shape shown in Figures II.12 and II.13 is obtained by placing an "F" shape in standard position relative to the $x^3, y^3$ coordinate system.

It should be stressed again that in this framework where transformations are viewed as acting on local coordinate systems, the meanings of the transformations are to be interpreted within the framework of the local coordinate system. In some applications, it can be useful to use this framework. For instance, when setting up a View matrix for defining a camera position and direction, it might be intuitive to think of the camera moving through a scene with the movements being specified relative to a local coordinate system attached to the camera. In this case, the intuition is that the scene is being transformed by the inverse of the camera movements.
Exercise II.15 Review the transformations used to draw the lower "F" shape shown in Figure II.12. Understand how this works from the viewpoint that transformations act on local coordinate systems. Draw a figure similar to Figure II.16 showing all the intermediate local coordinate systems that are implicitly defined by transformation (II.6).

II.1.8 Two dimensional projective geometry

Projective geometry provides an elegant mathematical interpretation of the homogeneous coordinates for points in the xy-plane. In this interpretation, the triples \( \langle x, y, w \rangle \) do not represent points just in the usual flat Euclidean plane, but in a larger geometric space known as the projective plane. The projective plane is an example of a projective geometry. A projective geometry is a system of points and lines which satisfy the following two axioms:

P1. Any two distinct points lie on exactly one line.

P2. Any two distinct lines contain exactly one common point (i.e., intersect in exactly one point).

Of course, the usual Euclidean plane, \( \mathbb{R}^2 \), does not satisfy the second axiom since parallel lines do not intersect in \( \mathbb{R}^2 \). However, by adding appropriate "points at infinity" and a "line at infinity," the Euclidean plane \( \mathbb{R}^2 \) can be enlarged so as to become a projective geometry. In addition, homogeneous coordinates are a suitable way of representing the points in the projective plane.

The intuitive idea of the construction of the projective plane is as follows: for each family of parallel lines in \( \mathbb{R}^2 \), we create a new point, called a point at infinity. This new point is added to each of these parallel lines. In addition, we add one new line: the line at infinity, which contains exactly all the new points at infinity. It is not hard to verify that the axioms P1 and P2 hold.

Consider a line \( L \) in Euclidean space \( \mathbb{R}^2 \): it can be specified by a point \( u \) on \( L \) and by a nonzero vector \( v \) in the direction of \( L \). In this case, \( L \) consists of the set of points

\[
\{ u + \alpha v : \alpha \in \mathbb{R} \} = \{ (u_1 + \alpha v_1, u_2 + \alpha v_2) : \alpha \in \mathbb{R} \}.
\]

For each value of \( \alpha \), the corresponding point on the line \( L \) has homogeneous coordinates \( (u_1/\alpha + v_1, u_2/\alpha + v_2, 1/\alpha) \). As \( \alpha \to \infty \), this triple approaches the limit \( (v_1, v_2, 0) \). This limit is a point at infinity and is added to the line \( L \) when we extend the Euclidean plane to the projective plane. If one takes the limit as \( \alpha \to -\infty \), then the triple \( (-v_1, -v_2, 0) \) is approached in the limit. This is viewed as being the same point as \( (v_1, v_2, 0) \), since multiplication by the nonzero scalar \(-1\) does not change the meaning of homogeneous coordinates. Thus, the intuition is that the same point at infinity on the line is found at both ends of the line.

\footnote{This is not a complete list of the axioms for projective geometry. For instance, it is required that every line has at least three points, etc.}
Note that the point at infinity, \((v_1, v_2, 0)\), on the line \(L\) did not depend on \(u\). If the point \(u\) is replaced by some point not on \(L\), then a different line is obtained; this line will be parallel to \(L\) in the Euclidean plane, and any line parallel to \(L\) can be obtained by appropriately choosing \(u\). Thus, any line parallel to \(L\) has the same point infinity as the line \(L\).

More formally, the projective plane is defined as follows. Two triples, \(\langle x, y, w \rangle\) and \(\langle x', y', w' \rangle\), are equivalent if there is a nonzero \(\alpha \in \mathbb{R}\) such that \(x = \alpha x', y = \alpha y'\) and \(w = \alpha w'\). We write \(\langle x, y, w \rangle^P\) to denote the equivalence class which contains the triples which are equivalent to \(\langle x, y, w \rangle\). The projective points are the equivalence classes \(\langle x, y, w \rangle^P\) such that at least one of \(x, y, w\) is nonzero. A projective point is called a point at infinity if \(w = 0\).

A projective line is either a usual line in \(\mathbb{R}^2\) plus a point at infinity, or the line at infinity. Formally, for any triple \(a, b, c\) of real numbers, with at least one of \(a, b, c\) nonzero, there is a projective line \(L\) defined by

\[
L = \{ \langle x, y, w \rangle^P : ax + by + cw = 0, x, y, w \text{ not all zero} \}. \quad (II.7)
\]

Suppose at least one of \(a, b\) is nonzero. Considering the \(w = 1\) case, the projective line \(L\) contains a point \(\langle x, y, 1 \rangle\) provided \(ax + by + c = 0\): this is the equation of a general line in the Euclidean space \(\mathbb{R}^2\). Thus \(L\) contains all triples which are representations of points on the Euclidean line \(ax + by + c = 0\). In addition, the line \(L\) contains the point at infinity \(\langle -b, a, 0 \rangle^P\). Note that \(\langle -b, a \rangle\) is a Euclidean vector parallel to the line defined by \(ax + by + c = 0\).

The projective line defined by (II.7) with \(a = b = 0\) and \(c \neq 0\) is the line at infinity; it contains those points \(\langle x, y, 0 \rangle^P\) such that \(x\) and \(y\) are not both zero.

**Exercise II.16** Another geometric model for the two dimensional projective plane is provided by the 2-sphere, with antipodal points identified. The 2-sphere is the sphere in \(\mathbb{R}^3\) which is centered at the origin and has radius 1. Points on the 2-sphere are represented by normalized triples \(\langle x, y, w \rangle\), which have \(x^2 + y^2 + w^2 = 1\). In addition, the antipodal points \(\langle x, y, w \rangle\) and \(\langle -x, -y, -w \rangle\) are treated as equivalent. Prove that lines in projective space correspond to great circles on the sphere, where a great circle is defined as the intersection of the sphere with a plane containing the origin. For example, the line at infinity corresponds to the intersection of the 2-sphere with the \(xy\)-plane. [Hint: Equation (II.7) can be viewed as defining \(L\) in terms of a dot product with \(\langle a, b, c \rangle\).]

Yet another way of mathematically understanding the two dimensional projective space is to view it as the space of linear subspaces of three dimensional Euclidean space. To understand this, let \(x = \langle x_1, x_2, x_3 \rangle\) be a representation of a point in the projective plane. This point is equivalent to the points \(\alpha x\) for all nonzero \(\alpha \in \mathbb{R}\); these points plus the origin form a line through the origin in \(\mathbb{R}^3\). A line through the origin is of course a one dimensional subspace, and we identify this one dimensional subspace of \(\mathbb{R}^3\) with the point \(x\).
Now consider a line $L$ in the projective plane. If $L$ is not the line at infinity, then it corresponds to a line in $\mathbb{R}^2$. One way to specify the line $L$ is choose $u = \langle u_1, u_2 \rangle$ on $L$ and a vector $v = \langle v_1, v_2 \rangle$ in the direction of $L$. The line $L$ then is the set of points $\{u + \alpha v : \alpha \in \mathbb{R}\}$. It is easy to verify that, after adding the point at infinity, the line $L$ contains exactly the following set of homogeneous points:

$$\{\beta\langle u_1, u_2, 1 \rangle + \gamma\langle v_1, v_2, 0 \rangle : \beta, \gamma \in \mathbb{R} \text{ s.t. } \beta \neq 0 \text{ or } \gamma \neq 0\}.$$

This set of triples is, of course, a plane in $\mathbb{R}^3$ with a hole at the origin. Thus, we can identify this two dimensional subspace of $\mathbb{R}^3$ (that is, the plane) with the line in the projective plane. If, on the other hand, $L$ is the line at infinity, then it corresponds in the same way to the two dimensional subspace $\{(x_1, x_2, 0) : x_1, x_2 \in \mathbb{R}\}$.

These considerations give rise to another way of understanding the two dimensional projective plane. The “points” of the projective plane are one dimensional subspaces of $\mathbb{R}^3$. The “lines” of the projective plane are two dimensional subspaces of $\mathbb{R}^3$. A “point” lies on a “line” if and only if the corresponding one dimensional subspace is a subset of the two dimensional subspace.

The historical development of projective geometry arose from the development of the theory of perspective by Brunelleschi in the early 15th century. The basic tenet of the theory of perspective for drawings and paintings is that families of parallel lines point towards a common “vanishing point,” which is essentially a point at infinity. The modern mathematical development of projective geometry based on homogeneous coordinates came much later of course, being developed by Feuerbach and Möbius in 1827 and Klein in 1871. Homogeneous coordinates have long been recognized as being useful for many computer graphics applications; see, for example, the early textbook by Newman and Sproull [81]. An accessible mathematical introduction to abstract projective geometry is the textbook by Coxeter [30].

**II.2 Transformations in 3-space**

We turn next to transformations in 3-space. This turns out to be very similar in many regards to transformations in 2-space. There are however some new features, most notably, rotations are more complicated in 3-space than in 2-space. First, we discuss how to extend, to 3-space, the concepts of linear and affine transformations, matrix representations for transformations, and homogeneous coordinates. We then explain the basic modeling commands in OpenGL for manipulating matrices. After that, we give a mathematical derivation of the rotation matrices needed in 3-space and give a proof of Euler’s theorem.
II.2.1 Moving from 2-space to 3-space

In 3-space, points, or vectors, are triples \( \langle x_1, x_2, x_3 \rangle \) of real numbers. We denote 3-space by \( \mathbb{R}^3 \) and use the notation \( \mathbf{x} \) for a point, with it being understood that \( \mathbf{x} = \langle x_1, x_2, x_3 \rangle \). The origin, or zero vector, now is \( \mathbf{0} = \langle 0, 0, 0 \rangle \). As before, we will identify \( \langle x_1, x_2, x_3 \rangle \) with the column vector with the same entries. By convention, we always use a “righthanded” coordinate system, as shown in Figure I.4 on page 7. This means that if you position your right hand so that your thumb points along the \( x \)-axis and your index finger is extended straight and points along the \( y \)-axis, then your palm will be facing in the positive \( z \)-axis direction. It also means that vector crossproducts are defined with the righthand rule. As discussed in Section I.2, it is common in computer graphics applications to visualize the \( x \)-axis as pointing to the right, the \( y \)-axis as pointing upwards, and the \( z \)-axis as pointing towards you.

Homogeneous coordinates for points in \( \mathbb{R}^3 \) are vectors of four numbers. The homogeneous coordinates \( \langle x, y, z, w \rangle \) represents the point \( \langle x/w, y/w, z/w \rangle \) in \( \mathbb{R}^3 \). The 2-dimensional projective geometry described in Section II.1.8 can be straightforwardly extended to a 3-dimensional geometry, by adding a “plane at infinity”: each line has a single point at infinity, and each plane has a line of points at infinity. (See Section II.2.7 for more on projective geometry.)

A transformation on \( \mathbb{R}^3 \) is any mapping from \( \mathbb{R}^3 \) to \( \mathbb{R}^3 \). The definition of a linear transformation on \( \mathbb{R}^3 \) is identical to the definition used for \( \mathbb{R}^2 \), except that now the vectors \( \mathbf{x} \) and \( \mathbf{y} \) range over \( \mathbb{R}^3 \). Similarly, the definitions of translation and of affine transformation are word-for-word identical to the definitions given for \( \mathbb{R}^2 \), except that now the translation vector \( \mathbf{u} \) is in \( \mathbb{R}^3 \). In particular, an affine transformation is still defined as the composition of a translation and a linear transformation.

Every linear transformation \( A \) in \( \mathbb{R}^3 \) can be represented by a \( 3 \times 3 \) matrix \( M \) as follows. Let \( \mathbf{i} = \langle 1, 0, 0 \rangle \), \( \mathbf{j} = \langle 0, 1, 0 \rangle \), and \( \mathbf{k} = \langle 0, 0, 1 \rangle \), and let \( \mathbf{u} = A(\mathbf{i}) \), \( \mathbf{v} = A(\mathbf{j}) \), and \( \mathbf{w} = A(\mathbf{k}) \). Set \( M \) equal to the matrix \( (\mathbf{u}, \mathbf{v}, \mathbf{w}) \), i.e., the matrix whose columns are \( \mathbf{u}, \mathbf{v}, \) and \( \mathbf{w} \), so

\[
M = \begin{pmatrix}
u_1 & v_1 & w_1 \\
u_2 & v_2 & w_2 \\
u_3 & v_3 & w_3
\end{pmatrix}.
\] (II.8)

Then \( M\mathbf{x} = A(\mathbf{x}) \) for all \( \mathbf{x} \in \mathbb{R}^3 \), that is to say, \( M \) represents \( A \). In this way, any linear transformation of \( \mathbb{R}^3 \) can be viewed as being a \( 3 \times 3 \) matrix. (Compare this to the analogous construction for \( \mathbb{R}^2 \) explained at the beginning of Section II.1.2.)

A rigid transformation is one that preserves the size and shape of an object, and changes only its position and orientation. Formally, a transformation \( A \) is defined to be rigid provided it preserves distances between points and preserves angles between lines. Recall that the length of a vector \( \mathbf{x} \) is equal to \(|\mathbf{x}| = \sqrt{\mathbf{x} \cdot \mathbf{x}} = \sqrt{x_1^2 + x_2^2 + x_3^2} \). An equivalent definition of rigidity is that a transformation \( A \) is rigid if it preserves dot products, that is to say, if \( A(\mathbf{x}) \cdot A(\mathbf{y}) = \mathbf{x} \cdot \mathbf{y} \) for all \( \mathbf{x}, \mathbf{y} \in \mathbb{R}^3 \). It is not hard to prove that \( M = (\mathbf{u}, \mathbf{v}, \mathbf{w}) \).
represents a rigid transformation if and only if $||u|| = ||v|| = ||w|| = 1$ and $u \cdot v = v \cdot w = u \cdot w = 0$. From this, it is straightforward to show that $M$ represents a rigid transformation if and only if $M^{-1} = M^T$ (c.f. exercises II.6 and II.7 on page 43).

We define an orientation preserving transformation to be one which preserves “righthandedness.” Formally, we say that $A$ is orientation preserving provided that $(A(u) \times A(v)) \cdot A(u \times v) > 0$ for all non-collinear $u, v \in \mathbb{R}^3$. By recalling the righthand rule used to determine the direction of a cross product, you should be able to convince yourself that this definition makes sense.

**Exercise II.17** Let $M = (u, v, w)$ be a $3 \times 3$ matrix. Prove that $\det(M)$ is equal to $(u \times v) \cdot w$. Conclude that $M$ represents an orientation preserving transformation if and only if $\det(M) > 0$. Also, prove that if $u$ and $v$ are unit vectors which are orthogonal to each other, then setting $w = u \times v$ makes $M = (u, v, w)$ a rigid, orientation preserving transformation.

Any affine transformation is the composition of a linear transformation and a translation. Since a linear transformation can be represented by a $3 \times 3$ matrix, any affine transformation can be represented by a $3 \times 3$ matrix and a vector in $\mathbb{R}^3$ representing a translation amount. That is, any affine transformation $A$ can be written as:

$$A : \begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} u \\ v \\ w \end{pmatrix},$$

so that $A(x) = B(x) + u$ where $B$ is the linear transformation with matrix representation

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix},$$

and $u$ is the vector $(u, v, w)$. We can rewrite this using a single $4 \times 4$ homogeneous matrix which acts on homogeneous coordinates, as follows:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} a & b & c & u \\ d & e & f & v \\ g & h & i & w \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$
Simple examples of transformations in 3-space include translations \( T_u \), uniform scalings \( S_\alpha \) and nonuniform scalings \( S_u \) where \( \alpha \) is a scalar, and \( u = \langle u_1, u_2, u_3 \rangle \). These have \( 4 \times 4 \) matrix representations

\[
T_u = \begin{pmatrix}
1 & 0 & 0 & u_1 \\
0 & 1 & 0 & u_2 \\
0 & 0 & 1 & u_3 \\
0 & 0 & 0 & 1
\end{pmatrix},
\ S_\alpha = \begin{pmatrix}
\alpha & 0 & 0 & 0 \\
0 & \alpha & 0 & 0 \\
0 & 0 & \alpha & 0 \\
0 & 0 & 0 & 1
\end{pmatrix},
\ S_u = \begin{pmatrix}
u_1 & 0 & 0 & 0 \\
0 & u_2 & 0 & 0 \\
0 & 0 & u_3 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}.
\]

Rotations in 3-space are considerably more complicated than in 2-space. The reason for this is that a rotation can be performed about any axis whatsoever. This includes not just rotations around the \( x \)-, \( y \)- and \( z \)-axes, but also rotations around an axis pointing in an arbitrary direction. A rotation which fixes the origin can be specified by giving a rotation axis \( u \) and a rotation angle \( \theta \), where the axis \( u \) can be any nonzero vector. We think of the base of the vector being placed at the origin, and the axis of rotation is the line through the origin parallel to the vector \( u \). The angle \( \theta \) gives the amount of rotation. The direction of the rotation is determined by the right-hand rule; namely, if one mentally grasps the vector \( u \) with one’s right hand so that the thumb, when extended, is pointing in the direction of the vector \( u \), then one’s fingers will curl around \( u \) pointing in the direction of the rotation. In other words, if one views the vector \( u \) head-on, namely, down the axis of rotation in the opposite direction that \( u \) is pointing, then the rotation direction is counter-clockwise (for positive values of \( \theta \)). A rotation of this type is denoted \( R_{\theta,u} \). By convention, the axis of rotation always passes through the origin, and thus the rotation fixes the origin. Figure II.19 on page 72 illustrates the action of \( R_{\theta,u} \) on a point \( v \).

Clearly, \( R_{\theta,u} \) is a linear transformation and is rigid and orientation preserving. The general formula for the matrix representation of the rotation \( R_{\theta,u} \) quite complicated:

\[
R_{\theta,u} = \begin{pmatrix}
(1 - c)u_1^2 + c & (1 - c)u_1u_2 - su_3 & (1 - c)u_1u_3 + su_2 & 0 \\
(1 - c)u_1u_2 + su_3 & (1 - c)u_2^2 + c & (1 - c)u_2u_3 - su_1 & 0 \\
(1 - c)u_1u_3 - su_2 & (1 - c)u_2u_3 + su_1 & (1 - c)u_3^2 + c & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}.
\] (II.9)

The formula (II.9) for \( R_{\theta,u} \) will be derived later in Section II.2.5. This complicated formula is not needed in common simple cases. In particular, it is easy to compute by hand the matrix representation of a rotation around the three \( xyz \) coordinate axes. For instance, \( R_{\theta,j} \) is represented by

\[
R_{\theta,j} = \begin{pmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta
\end{pmatrix}.
\]

To calculate the matrix, you just need to notice that \( R_{\theta,j}(i) = \langle \cos \theta, 0, -\sin \theta \rangle \), \( R_{\theta,j}(j) = j \), and \( R_{\theta,j}(k) = \langle \sin \theta, 0, \cos \theta \rangle \). These three vectors for the column of the upper left \( 3 \times s \) submatrix in the matrix representation of \( R_{\theta,j} \).
II.2.2 A solar system example (Draft A.2.d) 

Exercise II.18 What are the $4 \times 4$ matrix representations for $R_{z}^{\pi} , R_{-z}^{\frac{\pi}{3} \cdot k}$, and $R_{z}^{\pi} \cdot j$?

Section II.2.6 below shows that every rigid, orientation preserving, linear transformation in 3-space is a rotation. As a corollary, every rigid, orientation preserving, affine transformation can be (uniquely) expressed as the composition of a translation and a rotation about a line through the origin.

It is of course possible to have rotations about axes that do not pass through the origin. These are called “glide rotations” or “screw transformations”, and are discussed in Section II.2.6.

II.2.2 A solar system example

This section illustrates how to use transformations to position objects in a solar system containing a sun, an earth and a moon. It uses a View matrix $V$ to set the viewer’s position, and multiple Model matrices to position the sun, earth and moon. We’ll describe this in terms of matrices first; the next section will give C++ code based on the SolarModern program available at the book’s website.

We model the sun as placed at the origin, and the earth and moon lying in the $xz$-plane, as shown in Figure II.17. The earth has a circular orbit around the sun of radius $r_{E}$. The moon has a circular orbit around the earth of radius $r_{M}$. At some instant in time, the earth has orbited by $\theta_{E}$ radians as measured by the angle from the $z$-axis. And, the moon has orbited $\theta_{M}$ radians around the earth as measured by the angle from the line containing the centers of the sun and the earth. Finally, the earth has rotated $\varphi$ radians on its axis.

This is all pictured in Figure II.17 showing a top view of the solar system: in this view, the $y$-axis is pointing up out of the image towards the viewer, the $z$-axis is pointing downwards, and the $x$-axis is pointing rightwards.

We first give pseudo-code for placing the sun, earth and moon as shown in Figure II.17. After that, we describe how to modify the code to use the View matrix to position the solar system in front of the viewer. We will render the sun as a sphere of radius 1, the earth as a sphere of radius 0.7, and the moon as a sphere of radius 0.4. We let $S$ denote a sphere of radius 1 centered at the origin; then $M(S)$ denotes $S$ as transformed by a Model matrix $M$.

Here is the pseudo-code for rendering the sun and the earth; it uses three Model view matrices, $M_{S}$, $M_{E_{0}}$ and $M_{E_{1}}$. $M_{S}$ is for the sun; $M_{E_{0}}$ is for the earth-moon system; and $M_{E_{1}}$ is for the earth.
Figure II.17: A solar system with a sun, an earth, and a moon. This is a top view, from a viewpoint looking down the y-axis, with the z-axis pointing downward, and the x-axis rightward.

Set $M_S = \text{Identity}$. 
Render $M_S(S)$. // Render the sun
Set $M_{E_0} = M_S$. 
Set $M_{E_0} = M_{E_0} \cdot R_{\theta_E, \hat{j}}$. // Revolve around the sun.
Set $M_{E_0} = M_{E_0} \cdot T_{(0,0,r_E)}$. // Translate to earth’s orbit.
Set $M_{E_1} = M_{E_0}$. 
Set $M_{E_1} = M_{E_1} \cdot R_{\phi, \hat{j}}$. // Rotate earth on its axis.
Set $M_{E_1} = M_{E_1} \cdot S_{0.7}$. // Scale the earth smaller.
Render $M_{E_1}(S)$. // Render the earth

The second line of pseudo-code renders the sun as transformed by only the identity matrix, hence as a unit sphere at the origin. The final line renders the earth as a unit sphere transformed by the matrix $R_{\theta_E, \hat{j}} \cdot T_{(0,0,r_E)} \cdot R_{\phi, \hat{j}} \cdot S_{0.7}$. The effect of the final two matrices, $R_{\phi, \hat{j}} \cdot S_{0.7}$, is to rotate the earth on its axis and scale it by the factor 0.7. These two operations commute, so they could be have equivalently been applied in the reverse order. The first matrices, $R_{\theta_E, \hat{j}} \cdot T_{(0,0,r_E)}$, have the effect of positioning the earth onto its orbit, and then revolving it around the sun. The order of the matrices is important of course, since rotations and translation in general do not commute. You should verify that these operations place the earth as illustrated in Figure II.17: the constructions are very similar to the methods used in Section II.1.6 to position the “F” shapes.

The next block of pseudo-code positions the moon. It uses $M_m$ as a Model view matrix for the moon, and bases it off the Model matrix $M_{E_0}$ for the
II.2.2. A solar system example (Draft A.2.d)

```
earth-system. 

Set \( M_m = M_{E_0} \). // Copy earth-system matrix.
Set \( M_m = M_m \cdot R_{\theta_m,j} \). // Revolve around the earth.
Set \( M_m = M_m \cdot T_{(0,0,r_M)} \). // Translate to the moon's orbit.
Set \( M_m = M_m \cdot S_{0.4} \). // Scale the moon smaller.
Render \( M_m(S) \). // Render the moon

This has the effect of rendering the moon as a unit sphere at the origin, but transformed by the matrix

\[
R_{\theta,j} \cdot T_{(0,0,r_E)} \cdot R_{\theta_m,j} \cdot T_{(0,0,r_M)} \cdot S_{0.4}.
\]

The point of basing the Model matrix \( M_m \) for the moon off of \( M_{E_0} \) instead of the Model matrix \( M_{E_1} \) for the earth is that this means that the moon is not affected by the matrices that rotated and scaled the earth.

So far we have discussed only the Model view matrix; it is also necessary to compute also a View matrix.\(^6\) The purpose of the View matrix is to position the scene in front of the viewer so that we render the right part of the scene. The usual convention in OpenGL is that the viewer is positioned at the origin looking in the direction of the negative \( z \)-axis. The View matrix is used to place the scene in front of the viewer; for this, the center of the viewed scene should be placed on the negative \( z \)-axis.

We give some sample code that makes the sun the center of the view. The viewer will be placed slightly above the \( xz \)-plane so that the viewer is looking down slightly with the sun in the center of the view. The View matrix \( V \) with the following code:

```
Set \( V = \text{Identity} \). 
Set \( V = V \cdot T_{(0,0,-30)} \). // Translate 30 units down the \(-z\) axis.
Set \( V = V \cdot R_{\pi/20,i} \). // Tilt the solar system down about \(16^\circ\).
```

The rotation \( R_{\pi/20,i} \) rotates the solar system down approximately \(16^\circ\): the rotation holds the \( x \)-axis fixed, and moves the front of the solar system downward and the back of the solar system upward. The translation \( T_{(0,0,-25)} \) pushes the solar system away, so that its center (the sun) is 25 units in front of the viewer.

The View matrix \( V \) is sometimes incorporated in the Model view matrix \( M \), and sometimes calculated as a separate matrix. In the former case, the vertex shader uses the Model matrix in the calculation of the position \text{gl\_Position} of the vertex. (It also uses the Projection matrix.) In the latter case, the vertex shader receives separate \( V \) and \( M \) matrices, and applies them both to calculate \text{gl\_Position}.

In the former case, to incorporate the View matrix \( V \) into all the Model matrices, the first line of pseudo-code would change from \text{Set } \text{M} = \text{Identity}\(^6\) There is almost always also a Projection matrix, which controls the field of view and depth of field of the viewer. This will be discussed later in this chapter.
to Set \( M = V \). This is the method used in the SolarModern, whose code is discussed in the next section.

### II.2.3 Transformations and OpenGL

This section describes how the Model and View matrices can be used with a C++/OpenGL program. The older “legacy” or “immediate mode” OpenGL system had built-in methods (called the “fixed function pipeline”) for working with Modelview and Projection matrices, including using a stack for hierarchial use of Modelview matrices, and automatic application of Modelview and Projection matrices. Much of this fixed function pipeline has been removed in the “modern” OpenGL system. In modern OpenGL, a programmer must maintain all matrices the C++ code, plus must program the GPU shader programs to correctly apply the matrices. This removes a lot of the built-in functionality that was supported in legacy OpenGL, but has the advantage of providing the programmer much more flexibility and customizability.

Nonetheless, a lot of the features of legacy of OpenGL are still used in modern OpenGL systems. For instance, although it is not required to use the Modelview and Projection matrices in the same way as they were used in OpenGL, it still common to use the same general approach, albeit now explicitly coded into the shader programs. We will illustrate this with some code fragments from the SolarModern program.

First though, we describe the GLLinearMath software, available from the book’s website that we use for handling matrices and vectors. We then describe how matrices are given a shader program as a uniform variable. At the same time, for completeness, we show how colors are given to a vertex shader as a generic attribute. After that, we present some code fragments from SolarModern.

**The GLLinearMath package.** The GLLinearMath package contains full-featured support for C++ classes encapsulating vectors and matrices over \( \mathbb{R}^2 \), \( \mathbb{R}^3 \) and \( \mathbb{R}^4 \). We describe only a small set of its features here. The main C++ class we use is the \texttt{LinearMapR4} class, which defines \( 4 \times 4 \) matrices. It includes special functions for reproducing the functionality of the functions \texttt{glTranslatef}, \texttt{glRotatef}, \texttt{glScalef}, \texttt{glLoadMatrixf}, \texttt{glFrustum}, \texttt{glOrtho}, \texttt{gluPerspective} and \texttt{gluLookAt}, which are in legacy OpenGL but are no longer available in modern OpenGL.

The basic classes are \texttt{VectorR3}, \texttt{VectorR4} and \texttt{LinearMapR4}. Members of these classes are declared by

```cpp
VectorR3 vec3; // A 3-vector
VectorR3 vec4; // A 4-vector
LinearMapR4 mat; // A 4 \times 4 matrix
```

To initialize the matrix \texttt{mat} to the identity, use
II.2.3. Transformations and OpenGL (Draft A.2.d)

mat.SetIdentity();

To set mat equal to the $4 \times 4$ translation matrix $T_u$, where $u = \langle u_1, u_2, u_3 \rangle$, use the command

mat.Set_glTranslate($u_1$, $u_2$, $u_3$);

To multiply mat on the right by $T_u$, use instead the command

mat.Mult_glTranslate($u_1$, $u_2$, $u_3$);

To either set the matrix mat equal to the uniform scaling matrix $S_\alpha$, or to multiply it on the right with $S_\alpha$, use

mat.Set_glScale($\alpha$);
or
mat.Mult_glScale($\alpha$);

The corresponding commands for the non-uniform scaling $S_{(\alpha, \beta, \gamma)}$ are

mat.Set_glScale($\alpha$, $\beta$, $\gamma$);
mat.Mult_glScale($\alpha$, $\beta$, $\gamma$);

To set the matrix mat equal to the rotation matrix $R_{\theta, u}$, or to multiply it on the right by $R_{\theta, u}$, use

mat.Set_glRotate($\theta$, $u_1$, $u_2$, $u_3$);
or
mat.Mult_glRotate($\theta$, $u_1$, $u_2$, $u_3$);

The vector $u = \langle u_1, u_2, u_3 \rangle$ should be non-zero, but does not need to be a unit vector. Set_glRotate and Mult_glRotate normalize their input $u$ before using the rotation matrix formula (II.9).

All of the “Mult” methods multiply on the right by the transformation matrix. There are no built-in methods for multiplying on the left; however, this can be done with explicit multiplication. For example, you multiply on left by $R_{\theta, u}$ by using

```c++
LinearMapR4 rotMat;
rotMat.Set_glRotate($\theta$, $u_1$, $u_2$, $u_3$);
mat = rotMat * mat;
```

The glTranslate and glRotate methods can also take a VectorR3 argument. For example, if we first initialize a vector with

```c++
VectorR3 uVec($u_1$, $u_2$, $u_3$);
```

then the Mult_glTranslate and Mult_glRotate commands above can be equivalently expressed as
mat.Mult.glTranslate(uVec);
and
mat.Mult.glRotate(θ, uVec);

Neither legacy OpenGL nor GLLinearMath have special functions for reflections or shearing transformations. A reflection transformation is a transformation which transforms points to their “mirror image” across some plane, as illustrated in Figures II.3f and II.21. A reflection across the coordinate planes can easily be done with a nonuniform scaling. For example, $S_{(-1,1,1)}$ performs a reflection across the $yz$-plane by negating the $x$-coordinate of a point. Reflections across other planes can in principle be done by combining $S_{(-1,1,1)}$ with rotations, but this would generally be unnecessarily complicated.

A *shearing* transformation is an even more complicated kind of transformation; a two dimensional example is shown in Figure II.3e. In principle, one can use nonuniform scalings in combination with rotations to perform arbitrary shearing transformations. In practice, this is usually difficult and much more trouble than it is worth. It is often much more efficient and convenient to just explicitly give the components of a $4 \times 4$ matrix which perform any desired affine transformation. For example, the formulas from exercises II.22 and II.23 below can be used to get the entries of a $4 \times 4$ matrix that carries out a reflection.

GLLinearMath has a number of methods to set the contents of a matrix explicitly. (These replace the legacy OpenGL *LoadMatrix* method.) The primary methods are the *Set* methods which set the 16 entries of the $4 \times 4$ matrix of a *LinearMapR4* in column order. These are used as

```c
mat.Set( m11, m21, m31, m41, m12, ..., m44 );
```
and

```c
mat.Set( uVec1, uVec2, uVec3, uVec4 );
```

where uVec1, uVec2, uVec3, uVec4 are *VectorR4* objects. Again, these methods set the entries of *mat* by column order. There are similar *SetByRows* to set the entries in column order.

GLLinearMath also implements three legacy OpenGL methods for perspective transformations. These will be discussed later, in Section ???. For the record, these functions are:

```c
Set.glOrtho( left, right, bottom, top, near, far );
Set.glFrustum( left, right, bottom, top, near, far );
Set.gluPerspective( fieldofview_y, aspectratio, near, far, );
```

GLLinearMath also implements a version of the *gluLookAt* function of legacy OpenGL, which makes it easy to position the viewpoint at an arbitrary location, looking in an arbitrary direction with an arbitrary orientation. The GLLinearMath function is invoked by
II.2.3. Transformations and OpenGL

Set `gluLookAt( eyePos, lookAtPos, upDir);`

The variables `eyePos`, `lookAtPos` and `upDir` are `VectorR3` objects. The vector `eyePos` specifies a location in 3-space for the viewpoint. The `lookAtPos` vector specifies the point in the center of the field of view. This of course should be different than the view position `eyePos`. The vector `upDir` specifies an upward direction for the y-axis of the viewer. It is not necessary for `upDir` vector to be orthogonal to the vector `eyePos` to `lookAtPos`, but it must not be parallel to it.

The `gluLookAt` command should be used to set the View matrix. This is to maintain the convention that the viewer should always be placed at the origin.

Uniform variables and generic vertex attributes.

After the C++ program has computed the matrices needed for rendering, they need to be sent to the shader program so that they can be used to transform vertices within the shader program. This is typically done by making the matrices a “uniform variable” in the shader program.

Chapter I described using vertex attributes which were specified on a per-vertex basis and stored in a VBO. Transformation matrices, however, are typically fixed for a large number of vertices. For example, when rendering a sphere, the VBO might hold the vertices for a unit sphere centered at the origin, and the transformation matrices would be used to position the sphere vertices for rendering. Since the same matrices are used for all the vertices of the sphere, it would make no sense to store the matrices on a per-vertex basis.

OpenGL provides two ways to provide data a shader program that is the same for a range of vertices: generic vertex attributes and uniform variables. Generic vertex attributes are available as inputs only to the vertex shader, but not to the fragment shader or other shaders. On the other hand, uniform variables are available as inputs to the all the shaders as thus to both the vertex shader and the fragment shader.

Generic vertex attributes are specified with the `glVertexAttrib` family of commands. An example of this will be shown in the C++ solar system code below to set the color of the spheres. Uniform variables are set with the `glUniform` family of commands. This will be illustrated by using the `glUniformMatrix4f` command to load a $4 \times 4$ Modelview matrix into the shader.

Let’s start with the vertex shader code used by the SolarModern program.
The vertex shader has two vertex attributes as input, \texttt{vertPos} and \texttt{vertColor}. The \texttt{vertPos} values are given on a per-vertex basis, obtained from the VBO. The \texttt{vertColor} is a generic vertex attribute and will be specified by a \texttt{glVertexAttrib3f} command; it is the same for many vertices. Note that the vertex shader does not distinguish between generic vertex attributes and per-vertex vertex attributes. Indeed, it is possible that the same variable is a generic vertex attribute for some VAO's and a per-vertex attribute for other VAO's.

There are also two uniform inputs, the \texttt{projectionMatrix} and the \texttt{modelviewMatrix}. These matrices are also the same for many vertices at a time. As uniform variables, they are available as inputs to both the vertex shader and the fragment shader; however, the SolarModern program uses them only in the vertex shader. Nonetheless it is traditional to specify matrices as uniform inputs instead of generic vertex attributes, since some applications such as Phong interpolation need to access the matrices in the fragment shader.\footnote{It would possible to for the matrices to be generic inputs in the SolarModern program, but OpenGL makes this awkward as it would require the matrix to be loaded column-by-column with four calls to \texttt{glVertexAttrib4f}.}

An example of how to set a generic vertex attribute can be found in the SolarModern program:

\begin{verbatim}
unsigned int vertColor_loc = 1;
glVertexAttrib3f(vertColor_loc, 1.0f, 1.0f, 0.0f);
\end{verbatim}

The effect of this command is that a vertex shader program accessing the variable at location \texttt{vertColor\_loc} will use the value \(\langle 1,1,0 \rangle\) (representing bright yellow).\footnote{This code example hard-codes the location of \texttt{vertColor} as 1. If this cannot be hard-coded for some reason, it is also possible to use \texttt{glGetAttribLocation} to get the location for the vertex attribute \texttt{vertColor}.} This happens unless \texttt{glEnableVertexAttribArray} has been used to tell the VAO to take the vertex attribute from the VBO.

A uniform matrix variable is set by the commands below. The input to \texttt{glUniformMatrix4fv} is an array of single precision floating point numbers (\texttt{float}'s). The \texttt{GLLin\texttt{linearMath}} package works with double precision numbers so
II.2.4 Solar system in OpenGL

**Draft A.2.d**

It is necessary to dump out the matrix entries into an array of 16 float’s first. The matrix mat is the model view matrix.

```c
unsigned int vertColor_loc = 1;
int modelviewMatLocation = glGetUniformLocation(shaderProgram1, "modelviewMatrix");
float matEntries[16];
...
glUseProgram(shaderProgram1);
...
mat.DumpByColumns(matEntries);
glUniformMatrix4fv(modelviewMatLocation, 1, false, matEntries);
```

The shader program is the compiled program containing both the vertex shader and the fragment shader. It is set as the current shader program by the command `glUseProgram`. The `glUniformMatrix4fv` command sets the uniform variable in only the current shader program.

Unfortunately, the most commonly used versions of OpenGL do not allow hard-coding the locations of uniform variables. Thus the function `glGetUniformLocation` must be used to get the location of the uniform variable based on its name.

There are a lot of different forms of the `glVertexAttrib` and `glUniform` commands; for these, see the OpenGL documentation.

### II.2.4 Solar system in OpenGL

The SolarModern program creates a simple solar system with a central yellow sun, a blue planet revolving around the sun every 365 days, and a moon revolving around the planet 12 times per year. In addition, the planet rotates on its axis once per day, i.e., once per 24 hours. The planets are drawn as wire spheres, using the `GlGeomSpheres` C++ class.

Figure II.18 shows the code fragments of the SolarModern C++ code that deal with the Modelview matrix and set the uniform variables and generic vertex attributes. This uses the OpenGL features discussed in the last two section, and follows the pseudo-code of Section II.2.2. The complete SolarModern program and its documentation is available at the book’s website.

**Exercise II.19** Review the Solar program and understand how it works. Try making some of the following extensions to create a more complicated solar system.

a. Add one or more planets.

b. Add more moons. Make a geostationary moon, which always stays above the same point on the planet. Make a moon with a retrograde orbit. (A retrograde orbit means the moon revolves opposite to the usual direction, that is, in the clockwise direction instead of counter-clockwise.)
int modelviewMatLocation = glGetUniformLocation(shaderProgram1, "modelviewMatrix");

float matEntries[16];

viewMatrix.Set(glTranslate(0.0, 0.0, -CameraDistance);
viewMatrix.Mult(glRotate(viewAzimuth, 1.0, 0.0, 0.0);)

glUseProgram(shaderProgram1);
LinearMapR4 SunPosMatrix = viewMatrix;
SunPosMatrix.DumpByColumns(matEntries);
glUniformMatrix4fv(modelviewMatLocation, 1, false, matEntries);
glVertexAttrib3f(vertColor_loc, 1.0f, 1.0f, 0.0f); // Yellow
Sun.Render();

LinearMapR4 EarthPosMatrix = SunPosMatrix;
double revolveAngle = (DayOfYear / 365.0)*PI2;
EarthPosMatrix.Mult(glRotate(revolveAngle, 0.0, 1.0, 0.0);
LinearMapR4 EarthMatrix = EarthPosMatrix;
double earthRotationAngle = (HourOfDay / 24.0)*PI2;
EarthMatrix.Mult(glRotate(earthRotationAngle, 0.0, 1.0, 0.0);
EarthMatrix.Mult(glScale(0.5));
EarthMatrix.DumpByColumns(matEntries);
glUniformMatrix4fv(modelviewMatLocation, 1, false, matEntries);
glVertexAttrib3f(vertColor_loc, 0.2f, 0.4f, 1.0f); // Cyan-blue
Earth.Render();

LinearMapR4 MoonMatrix = EarthPosMatrix;
double moonRotationAngle = (DayOfYear*12.0 / 365.0)*PI2;
MoonMatrix.Mult(glRotate(moonRotationAngle, 0.0, 1.0, 0.0);
MoonMatrix.Mult(glTranslate(0.0, 0.0, 1.0));
MoonMatrix.Mult(glScale(0.2));
MoonMatrix.DumpByColumns(matEntries);
glUniformMatrix4fv(modelviewMatLocation, 1, false, matEntries);
glVertexAttrib3f(vertColor_loc, 0.9f, 0.9f, 0.9f); // Bright gray
Moon1.Render();

Figure II.18: Selected SolarModern code, showing the use of the use of the Modelview matrix and setting generic vertex attributes and uniform variables. See Sections II.2.2, II.2.3 and II.2.3 for discussions of how the code works.
II.2.5 Derivation of the rotation matrix

This section contains the mathematical derivation of formula (II.9) for the matrix representing a rotation, $R_{\theta, \mathbf{u}}$, through an angle $\theta$ around axis $\mathbf{u}$. Recall that this formula was

$$
R_{\theta, \mathbf{u}} = \begin{pmatrix}
(1-c)u_1^2 + c & (1-c)u_1u_2 - su_3 & (1-c)u_1u_3 + su_2 & 0 \\
(1-c)u_1u_2 + su_3 & (1-c)u_2^2 + c & (1-c)u_2u_3 - su_1 & 0 \\
(1-c)u_1u_3 - su_2 & (1-c)u_2u_3 + su_1 & (1-c)u_3^2 + c & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}, \quad (\text{II.9})
$$

where $c = \cos \theta$ and $s = \sin \theta$. The vector $\mathbf{u}$ must be a unit vector. There is no loss of generality in assuming that $\mathbf{u}$ is a unit vector, since if not, then it may be normalized by dividing by $||\mathbf{u}||$.

To derive the matrix for $R_{\theta, \mathbf{u}}$, let $\mathbf{v}$ be an arbitrary point and consider what $\mathbf{w} = R_{\theta, \mathbf{u}}\mathbf{v}$ is equal to. For this, we split $\mathbf{v}$ into two components, $\mathbf{v}_1$ and $\mathbf{v}_2$, so that $\mathbf{v} = \mathbf{v}_1 + \mathbf{v}_2$, with $\mathbf{v}_1$ parallel to $\mathbf{u}$ and $\mathbf{v}_2$ orthogonal to $\mathbf{u}$. The vector $\mathbf{v}_1$ is the projection of $\mathbf{v}$ onto the line of $\mathbf{u}$, and is equal to $\mathbf{v}_1 = (\mathbf{u} \cdot \mathbf{v})\mathbf{u}$, since the dot product $\mathbf{u} \cdot \mathbf{v}$ is equal to $||\mathbf{u}|| \cdot ||\mathbf{v}|| \cos(\phi)$ where $\phi$ is the angle between $\mathbf{u}$ and $\mathbf{v}$, and since $||\mathbf{u}|| = 1$. (Refer to Figure II.19.) We rewrite this as

$$
\mathbf{v}_1 = (\mathbf{u} \cdot \mathbf{v})\mathbf{u} = \mathbf{u}(\mathbf{u} \cdot \mathbf{v}) = \mathbf{u}(\mathbf{u}^T\mathbf{v}) = (\mathbf{uu}^T)\mathbf{v}.
$$

The above equation uses the fact that a dot product $\mathbf{u} \cdot \mathbf{v}$ can be rewritten as a matrix product $\mathbf{u}^T\mathbf{v}$ (recall that our vectors are all column vectors) and the fact that matrix multiplication is associative. The product $\mathbf{uu}^T$ is the symmetric $3 \times 3$ matrix

$$
\text{Proj}_\mathbf{u} = \mathbf{uu}^T = \begin{pmatrix}
u_1 \\
u_2 \\
u_3
\end{pmatrix} \begin{pmatrix}
u_1 & u_1u_2 & u_1u_3 \\
u_2 & u_2^2 & u_2u_3 \\
u_3 & u_3u_2 & u_3^2
\end{pmatrix} = \begin{pmatrix}
u_1^2 & u_1u_2 & u_1u_3 \\
u_1u_2 & u_2^2 & u_2u_3 \\
u_1u_3 & u_2u_3 & u_3^2
\end{pmatrix}.
$$
Transformations and Viewing

Figure II.19: The vector $v$ being rotated around $u$. The vector $v_1$ is $v$’s projection onto $u$. The vector $v_2$ is the component of $v$ orthogonal to $u$. The vector $v_3$ is $v_2$ rotated $90^\circ$ around $u$. The dashed line segments in the figure all meet at right angles.

$$v = v_1 + v_2$$

Since $v = v_1 + v_2$, we therefore have

$$v_1 = \text{Proj}_u v \quad \text{and} \quad v_2 = (I - \text{Proj}_u)v,$$

where $I$ is the $3 \times 3$ identity matrix.

We know that $R_{\theta,u}v_1 = v_1$, since $v_1$ is a scalar multiple of $u$ and is not affected by a rotation around $u$. In order to compute $R_{\theta,u}v_2$, we further define $v_3$ to be the vector

$$v_3 = u \times v_2 = u \times v.$$

The second equality holds since $v$ and $v_2$ differ by a multiple of $u$. The vector $v_3$ is orthogonal to both $u$ and $v_2$. Furthermore, since $u$ is a unit vector orthogonal to $v_2$, $v_3$ has the same magnitude as $v_2$. That is to say, $v_3$ is equal to the rotation of $v_2$ around the axis $u$ through an angle of $90^\circ$.

Figure II.20 shows a view of $v_2$ and $v_3$ looking straight down the $u$ axis of
rotation. From the figure, it is obvious that rotating \( v_2 \) through an angle of \( \theta \) around \( u \) results in the vector

\[
(\cos \theta)v_2 + (\sin \theta)v_3.
\] (II.10)

Therefore, \( R_{\theta,u}v \) is equal to

\[
R_{\theta,u}v = R_{\theta,u}v_1 + R_{\theta,u}v_2 \\
= v_1 + (\cos \theta)v_2 + (\sin \theta)v_3 \\
= \text{Proj}_u v + (\cos \theta)(I - \text{Proj}_u) v + (\sin \theta)(u \times v).
\]

To finish deriving the matrix for \( R_{\theta,u} \), we define the matrix

\[
M_{u\times} = \begin{pmatrix}
0 & -u_3 & u_2 \\
-3 & 0 & -u_1 \\
u_2 & -u_1 & 0
\end{pmatrix}
\]

and see, by a simple calculation, that \((M_{u\times})v = u \times v\) holds for all \( v \). From this, it is immediate that

\[
R_{\theta,u}v = \left[(1 - \cos \theta)\text{Proj}_u + (\cos \theta)(I - \text{Proj}_u) + (\sin \theta)(u \times M_{u\times})\right]v.
\]

The quantity inside the square brackets is a 3 \( \times \) 3 matrix, so this completes the derivation of the matrix representation of \( R_{\theta,u} \). An easy calculation shows that this corresponds to the representation given earlier (in homogeneous form) by Equation (II.9).

**Exercise II.20** Carry out the calculation to show that the formula for \( R_{\theta,u} \) above is equivalent to the formula in Equation (II.9).

**Exercise II.21** Let \( u, v \) and \( w \) be orthogonal unit vectors with \( w = u \times v \). Prove that \( R_{\theta,u} \) is represented by the following 3 \( \times \) 3 matrix:

\[
\begin{pmatrix}
0 & -u_3 & u_2 \\
u_2 & -u_1 & 0
\end{pmatrix}
\]

It is also possible to convert a rotation matrix back into a unit rotation vector \( u \) and a rotation angle \( \theta \). For this, refer back to Equation (II.9). Suppose we are given such a 4 \( \times \) 4 rotation matrix \( M = (m_{i,j})_{i,j} \) so that the entry in row \( i \) and column \( j \) is \( m_{i,j} \). The sum of the first three entries on the diagonal of \( M \) (that is to say, the trace of the 3 \( \times \) 3 submatrix representing the rotation) is equal to

\[
m_{1,1} + m_{2,2} + m_{3,3} = (1 - c) + 3c = 1 + 2c
\]

since \( u_1^2 + u_2^2 + u_3^2 = 1 \). Thus, \( \cos \theta = (m_{1,1} + m_{2,2} + m_{3,3} - 1)/2 \), or

\[
\theta = \arccos(\alpha/2),
\]
where $\alpha = m_{1,1} + m_{2,2} + m_{3,3} - 1$. Letting $s = \sin \theta$, we can determine $u$'s components from:

$$
\begin{align*}
    u_1 &= \frac{m_{3,2} - m_{2,3}}{2s} \\
    u_2 &= \frac{m_{1,3} - m_{3,1}}{2s} \\
    u_3 &= \frac{m_{2,1} - m_{1,2}}{2s}.
\end{align*}
$$

(II.11)

The above method of computing $\theta$ and $u$ from $M$ will have problems with stability if $\theta$ is very close to 0 since, in that case, $\sin \theta \approx 0$, and then the determination of the values of $u_i$ requires dividing by values near zero. The problem is that dividing by a near-zero value tends to introduce unstable or inaccurate results, since small roundoff errors can have a large effect on the results of the division.

Of course, if $\theta$, and thus $\sin \theta$, are exactly equal to zero, then the rotation angle is zero and any vector $u$ will work. Absent roundoff errors, this situation occurs only if $M$ is the identity matrix.

To mitigate the problems caused dividing by a near-zero value, one should instead compute

$$
\beta = \sqrt{(m_{3,2} - m_{2,3})^2 + (m_{1,3} - m_{3,1})^2 + (m_{2,1} - m_{1,2})^2}.
$$

Note that $\beta$ will equal $2s = 2\sin \theta$, since dividing by $2s$ in equations (II.11) was what was needed to normalize the vector $u$. If $\beta$ is zero, then the rotation angle $\theta$ is zero and, in this case, $u$ may be an arbitrary unit vector. If $\beta$ is nonzero, then

$$
\begin{align*}
    u_1 &= \frac{(m_{3,2} - m_{2,3})}{\beta} \\
    u_2 &= \frac{(m_{1,3} - m_{3,1})}{\beta} \\
    u_3 &= \frac{(m_{2,1} - m_{1,2})}{\beta}.
\end{align*}
$$

This way of computing $u$ makes it more likely that a (nearly) unit vector will be obtained for $u$ when the rotation angle $\theta$ is near zero. From $\alpha$ and $\beta$, the angle $\theta$ can be computed as

$$
\theta = \arctan(\beta, \alpha).
$$

This is a more robust way to compute $\theta$ than using the arccos function.

For an alternate, and often better, method of representing rotations in terms of 4-vectors, see the parts of Section XIII.3 on quaternions (pages 408-419).

**Exercise II.22**

A plane $P$ containing the origin can be specified by giving a unit vector $u$ which is orthogonal to the plane. That is, let $P = \{ x \in \mathbb{R}^3 : u \cdot x = 0 \}$. A reflection across $P$ is the linear transformation which maps each point $x$ to its 'mirror image’ directly across $P$, as illustrated in Figure II.21.
II.2.6 Euler’s theorem

A fundamental fact about rigid, orientation preserving, linear transformations is that they are always equivalent to a rotation around an axis passing through the origin.

Theorem II.8 If $A$ is a rigid, orientation preserving, linear transformation of $\mathbb{R}^3$, then $A$ is the same as some rotation $R_{\theta}$. 

This theorem was formulated by Euler in 1776. We give a concrete proof based on symmetries; it is also possible to give a proof based on matrix properties.

Proof The idea of the proof is similar to the proof of Theorem II.7, which showed that every rigid, orientation preserving, affine transformation is either a generalized rotation or a translation. However, now we shall consider the action of $A$ on points on the unit sphere instead of on points in the plane.

Since $A$ is rigid, unit vectors are mapped to unit vectors. So, $A$ maps the unit sphere onto itself. In fact, it will suffice to show that $A$ maps some point $v$ on the unit sphere to itself, since if $v$ is a fixed point, then $A$ fixes the
line through the origin containing $v$. The rigidity and orientation preserving properties then imply that $A$ is a rotation around this line, because the action of $A$ on $v$ and on a vector perpendicular to $v$ determines all the values of $A$.

Assume that $A$ is not the identity map. First, note that $A$ cannot map every point $u$ on the unit sphere to its antipodal point $-u$, since otherwise, $A$ would not be orientation preserving. Therefore, there is some unit vector $u_0$ on the sphere such that $A(u_0) \neq -u_0$. Fix such a point, and let $u = A(u_0)$. If $u = u_0$, we are done, so suppose $u \neq u_0$. Let $C$ be the great circle containing both $u_0$ and $u$, and let $L$ be the shorter portion of $C$ connecting $u_0$ to $u$, i.e., $L$ is spanning less than $180^\circ$ around the unit sphere. Let $L'$ be the image of $L$ under $A$ and let $C'$ be the great circle containing $L'$. Suppose that $L = L'$, i.e., that $A$ maps this line to itself. In this case, rigidity implies that $A$ maps $u$ to $u_0$. Then, rigidity further implies that the point $v$ midway between $u_0$ and $u$ is a fixed point of $A$, so $A$ is a rotation around $v$.

Otherwise, suppose $L \neq L'$. Let $L'$ make an angle of $\theta$ with the great circle $C$, as shown in Figure II.22. Since $L \neq L'$, we have $-180^\circ < \theta < 180^\circ$. Let $C_2$, respectively $C_2'$, be the great circle which is perpendicular to $L$ at $u_0$, respectively at $u$. Let $C_3$ be $C_2$ rotated an angle of $-\theta/2$ around the vector $u_0$, and let $C_3'$ be $C_2'$ rotated an angle of $\theta/2$ around $u$. Then $C_3$ and $C_3'$ intersect at a point $v$ which is equidistant from $u_0$ and $u$. Furthermore, by rigidity considerations and the definition of $\theta$, $A$ maps $C_3$ to $C_3'$ and $v$ is a fixed point of $A$. Thus, $A$ is a rotation around the vector $v$.

One can define a generalized rotation in 3-space to be a transformation $R_{\theta,u}^v$ which performs a rotation through angle $\theta$ around the line $L$, where $L$ is the line which contains the point $v$ and is parallel to $u$. However, unlike the situation for 2-space (see Theorem II.7), it is not the case that every rigid, orientation preserving, affine transformation in 3-space is equivalent to either a translation or a generalized rotation of this type. Instead, we need a more general notion of "glide rotation" which incorporates a screw-like motion. For example, consider a transformation which both rotates around the $y$-axis and translates along the $y$-axis.

A glide rotation is a mapping which can be expressed as a translation along an axis $u$ composed with a rotation $R_{\theta,u}^v$ along the line which contains $v$ and is parallel to $u$.

**Exercise II.24** Prove that every rigid, orientation preserving, affine transformation is a glide rotation. (This is the “Mozzi-Chasles theorem”, first formulated by Mozzi in 1763, and later by Chasles in 1830.) (Hint: First consider $A$’s action on planes, and define a linear transformation $B$ as follows: let $r$ be a unit vector perpendicular to a plane $P$, and define $B(r)$ to be the unit vector perpendicular to the plane $A(P)$. $B$ is a rigid, orientation preserving map on the unit sphere. Furthermore, $B(r) = A(r) - A(0)$, so $B$ is a linear transformation. By Euler’s theorem, $B$ is a rotation. Let $w$ be a unit vector fixed by $B$, and $Q$ be the plane through the origin perpendicular to $w$, so $A(Q)$ is parallel to $Q$. Let $C$ be a transformation on $Q$ defined by letting $C(x)$ be
Figure II.22: Finding the axis of rotation. We have \( \mathbf{u} = A(\mathbf{u}_0) \) and \( \mathbf{v} = A(\mathbf{v}) \).
Compare this with Figure II.9.
the value of $A(x)$ projected onto $Q$. Then $C$ is a two dimensional, generalized rotation around a point $v$ in the plane $Q$. (Why?) From this, deduce that $A$ has the desired form.]

II.2.7 Three dimensional projective geometry

Three dimensional projective geometry can be developed analogously to the two dimensional geometry discussed in Section II.1.8, and three dimensional projective space can be viewed either as the usual three dimensional Euclidean space augmented with points at infinity, or as the space of linear subspaces of the four dimensional $\mathbb{R}^4$.

We first consider how to represent three dimensional projective space as $\mathbb{R}^3$ plus points at infinity. The new points at infinity are obtained as follows: let $\mathcal{F}$ be a family of parallel lines, i.e., let $\mathcal{F}$ be the set of lines parallel to a given line $L$, where $L$ is a line in $\mathbb{R}^3$. We have a new point at infinity, $u_\mathcal{F}$, and this point is added to every line in $\mathcal{F}$. The three dimensional projective space consists of $\mathbb{R}^3$ plus these new points at infinity. Each plane $P$ in $\mathbb{R}^3$ gets a new line of points at infinity in the projective space, namely the points at infinity that belong to the lines in the plane $P$. The set of lines of the projective space are (a) the lines of $\mathbb{R}^3$ (including their new point at infinity), and (b) the lines at infinity that lie in a single plane. Finally, the set of all points at infinity forms the plane at infinity.

You should check that, in three dimensional projective space, any two distinct planes intersect in a unique line.

Three dimensional projective space can also be represented by linear subspaces of the four dimensional space $\mathbb{R}^4$. This corresponds to the representation of points in $\mathbb{R}^3$ by homogeneous coordinates. A point in the projective space is equal to a one dimensional subspace of $\mathbb{R}^4$, namely, a set of points of the form $\{au : a \in \mathbb{R}\}$ for $u$ a fixed nonzero point of $\mathbb{R}^4$. The 4-tuple $u$ is just a homogeneous representation of a point; if its fourth component ($w$-component) is zero, then the point is a point at infinity. The lines in projective space are just the two dimensional subspaces of $\mathbb{R}^4$. A line is a line at infinity if and only if all its 4-tuples have zero as fourth component. The planes in projective space are precisely the three dimensional subspaces of $\mathbb{R}^4$.

**Exercise II.25** Work out the correspondence between the two ways of representing three dimensional projective space.

OpenGL and other similar systems use 4-tuples as homogeneous coordinates for points in 3-space extensively. In OpenGL, the function call `glVertex4f(a, b, c, d)` is used to specify a point $(a, b, c, d)$ in homogeneous coordinates. Of course, it is more common for a programmer to specify a point with only three (non-homogeneous) coordinates, but then, whenever a point in 3-space is specified by a call to `glVertex3f(a, b, c)`, OpenGL translates this to the point $(a, b, c, 1)$. 
However, OpenGL does not usually deal explicitly with points at infinity (although there are some exceptions, namely, defining Bézier and B-spline curves). Instead, points at infinity are typically used for indicating directions. As we shall see later, when a light source is given a position, OpenGL interprets a point at infinity as specifying a direction. Strictly speaking, this is not a mathematically correct use of homogeneous coordinates, since taking the negative of the coordinates does not yield the same result, but instead indicates the opposite direction for the light.

II.3 Viewing transformations and perspective

XXX THIS SECTION NEEDS A PARTIAL REWRITE.

So far, we have used affine transformations as a method for placing geometric models of objects in 3-space. This is represented by the first stage of the rendering pipeline shown in Figure ?? on page ???. In this first stage, points are placed in 3-space, controlled by the model view matrix.

We now turn our attention to the second stage of the pipeline. This stage deals with how the geometric model in 3-space is viewed; namely, it places the camera or eye with a given position, view direction, and field of view. The placement of the camera or eye position determines what parts of the 3D model will be visible in the final graphics image. Of course, there is no actual camera, it is only virtual; instead, transformations are used to map the geometric model in 3-space into the $xy$-plane of the final image. Transformations used for this purpose are called viewing transformations. Viewing transformations include not only the affine transformations discussed earlier, but also a new class of “perspective transformations.”

In order to understand properly the purposes and uses of viewing transformations, it is necessary to consider the end result of the rendering pipeline (Figure ??). The final output of the rendering pipeline is usually a rectangular array of pixels. Each pixel has an $xy$-position in the graphics image. In addition, each pixel has a color or grayscale value. Finally, it is common for each pixel to store a “depth value” or “distance value” which measures the distance to the object visible in that pixel.

Storing the depth is important since it is used by the hidden surface algorithm. When rendering a scene, there may be multiple objects that lie behind a given pixel. As the objects are drawn onto the screen, the depth value, or distance, to the relevant part of the object is stored into each pixel location. By comparing depths, it can be determined whether one object is in front of another object, and thereby that the further object, being hidden behind the closer object, is not visible.

The use of the depth values is discussed more in Section III.1, but for now it is enough for us to keep in mind that it is important to keep track of the distance of objects from the camera position.

Stages 2 and 3 of the rendering pipeline are best considered together. These two stages are largely independent of the resolution of the screen or other output.
device. During the second stage, vertices are mapped by a $4 \times 4$ affine matrix into new homogeneous coordinates $\langle x, y, z, w \rangle$. The third stage, *perspective division*, further transforms these points by converting them back to points in $\mathbb{R}^3$ by the usual map

$$\langle x, y, z, w \rangle \mapsto \langle x/w, y/w, z/w \rangle.$$ 

The end result of the second and third stages is that they map the viewable objects into the $2 \times 2 \times 2$ cube centered at the origin, which contains the points with $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, and $-1 \leq z \leq 1$. This cube will be mapped by simple rectangular scaling into the final graphics image during stage four of the rendering pipeline. The points with $x = 1$ (respectively, $x = -1$) are to be at the right (respectively, left) side of the screen or final image, and points with $y = 1$ (respectively, $y = -1$) are at the top (respectively, bottom) of the screen. Points with $z = 1$ are closest to the viewer and points with $z = -1$ are farthest from the viewer.\(^9\)

There are two basic kinds of viewing transformations: orthographic projections and perspective transformations. An orthographic projection acts similarly to placing the viewer at an infinite distance (with a suitable telescope). Thus, orthographic projections map the geometric model by projecting at right angles onto a plane perpendicular to the view direction. Perspective transformations put the viewer at a finite position, and perspective makes closer objects appear larger than distant objects of the same size. The difference between orthographic and perspective transformations is illustrated in Figure II.23.

To simplify the definitions of orthographic and perspective transformations, it is convenient to define them only for a viewer who is placed at the origin and is looking in the direction of the negative $z$-axis. If the viewpoint is to be placed elsewhere or directed elsewhere, ordinary affine transformations can be used to adjust the view accordingly.

### II.3.1 Orthographic viewing transformations

Orthographic viewing transformations carry out a parallel projection of a 3D model onto a plane. Unlike the perspective transformations described later, orthographic viewing projections do not cause closer objects to appear larger and distant objects to appear smaller. For this reason, orthographic viewing projections are generally preferred for applications such as architecture or engineering applications, including computer aided design and manufacturing (CAD/CAM), since the parallel projection is better at preserving relative sizes and angles.

\(^9\)OpenGL uses the reverse convention on $z$, with $z = -1$ for the closest objects and $z = 1$ for the farthest objects. Of course, this is merely a simple change of sign of the $z$ component, but OpenGL’s convention seems less intuitive since the transformation into the $2 \times 2 \times 2$ cube is no longer orientation preserving. Since the OpenGL conventions are hidden from the programmer in most situations anyway, we will instead adopt the more intuitive convention.
Figure II.23: The cube on the left is rendered with an orthographic projection. The one on the right with a perspective transformation. With the orthographic projection, the rendered size of a face of the cube is independent of its distance from the viewer; compare, for example, the front and back faces. Under a perspective transformation, the closer a face is, the larger it is rendered.

For convenience, orthographic projections are defined in terms of an observer who is at the origin and is looking down the $z$-axis in the negative $z$-direction. The view direction is perpendicular to the $xy$-plane, and if two points differ in only their $z$-coordinate, then the one with higher $z$-coordinate is closer to the viewer.

An orthographic projection is generally specified by giving six axis-aligned “clipping planes” which form a rectangular prism. The geometry which lies inside the rectangular prism is scaled to have dimensions $2 \times 2 \times 2$ and translated to be centered at the origin. The rectangular prism is specified by six values $\ell$, $r$, $b$, $t$, $n$ and $f$. These variable names are mnemonics for “left,” “right,” “bottom,” “top,” “near,” and “far.” The rectangular prism then consists of the points $\langle x, y, z \rangle$ such that

\[
\begin{align*}
\ell & \leq x \leq r, \\
b & \leq y \leq t, \\
and & \quad n \leq -z \leq f.
\end{align*}
\]

The $-z$ has a negative sign because of the convention that the viewer is looking down the $z$-axis, facing in the negative $z$ direction. This means that the distance of a point $(x, y, z)$ from the viewer is equal to $-z$. The usual convention is for $n$ and $f$ to be positive values; however, this is not actually required. The plane $z = -n$ is called the near clipping plane and the plane $z = -f$ is called the far clipping plane. Objects which are closer than the near clipping plane or farther than the far clipping plane will be culled and not be rendered.

The orthographic projection must map points from the rectangular prism into the $2 \times 2 \times 2$ cube centered at the origin. This consists of, firstly, scaling along the coordinate axes and, secondly, translating so that the cube is centered.
at the origin. It is not hard to verify that this is accomplished by the following
$4 \times 4$ homogeneous matrix:

$$
\begin{pmatrix}
  \frac{2}{r - \ell} & 0 & 0 & -\frac{r + \ell}{r - \ell} \\
  \frac{2}{t - b} & 0 & -\frac{t + b}{t - b} & 0 \\
  0 & 2 & \frac{f + n}{f - n} & 0 \\
  0 & 0 & \frac{f - n}{f - n} & 1
\end{pmatrix}.
$$

(II.12)

II.3.2 Perspective transformations

Perspective transformations are used to create the view when the camera or eye position is placed at a finite distance from the scene. The use of perspective means that an object will be displayed larger the closer it is. Perspective is useful for giving the viewer the sense of being “in” a scene, since a perspective view shows the scene from a particular viewpoint. Perspective is heavily used in entertainment applications where it is desired to give an immersive experience; it is particularly useful in dynamic situations where the combination of motion and correct perspective gives a strong sense of the three dimensionality of the scene. It is also used in applications as diverse as architectural modeling and crime re-creation, to show the view from a particular viewpoint.

As was mentioned in Section II.1.8, perspective was originally discovered for applications in drawing and painting. An important principle in the classic theory of perspective is the notion of a “vanishing point” shared by a family of parallel lines. An artist who is incorporating perspective in a drawing will choose appropriate vanishing points to aid the composition of the drawing. In computer graphics applications, we are able to avoid all considerations of vanishing points, etc. Instead, we place objects in 3-space, choose a viewpoint (camera position), and mathematically calculate the correct perspective transformation to create the scene as viewed from the viewpoint.

For simplicity, we consider only a viewer who is placed at the origin looking down the negative $z$-axis. We mentally choose as a “viewscreen” the plane $z = -d$, which is parallel to the $xy$-plane at distance $d$ from the viewpoint at the origin. Intuitively, the viewscreen serves as a display screen onto which viewable objects are projected. Let a vertex in the scene have position $(x, y, z)$. We form the line from the vertex position to the origin, and calculate the point $(x', y', z')$ where the line intersects the viewscreen (see Figure II.24). Of course, we have $z' = -d$. Referring to Figure II.24 and arguing using similar triangles, we have

$$
x' = \frac{d \cdot x}{-z} \quad \text{and} \quad y' = \frac{d \cdot y}{-z}.
$$

(II.13)

The values $x'$, $y'$ give the position of the vertex as seen on the viewscreen from the viewpoint at the origin.
II.3.2. Perspective transformations (Draft A.2.d)

83

Figure II.24: Perspective projection onto a viewscreen at distance $d$. The viewer is at the origin, looking in the direction of the negative $z$ axis. The point $(x, y, z)$ is perspective projected onto the plane $z = -d$, which is at distance $d$ in front of the viewer at the origin.

So far, projective transformations have been very straightforward, but now it is necessary to incorporate also the "depth" of the vertex, i.e., its distance from the viewer. The obvious first attempt would be to use the value $-z$ for the depth. Another, albeit less appealing, possibility would be to record the true distance $\sqrt{x^2 + y^2 + z^2}$ as the depth. Both of these ideas, however, fail to work well. The reason they fail to work well is that if perspective mappings are defined with a depth defined in either of these ways, then lines in the three dimensional scene can be mapped to curves in the viewscreen space. That is to say, a line of points with coordinates $x, y, z$, will map to a curve which is not a line in the viewscreen space.

An example of how a line can map to a curve is shown in Figure II.25. For this figure, we use the transformation

$$
\begin{align*}
    x &\mapsto \frac{d \cdot x}{-z} \\
y &\mapsto \frac{d \cdot y}{-z} \\
z &\mapsto z
\end{align*}
$$

(II.14)

so that the $z$-coordinate directly serves a measure of depth. (Since the viewpoint is looking down the negative $z$-axis, greater values of $z$ correspond to closer points.) In Figure II.25, we see points $A$, $B$, and $C$ that are mapped by (II.14) to points $A'$, $B'$, and $C'$. Obviously, $A$ and $C$ are fixed points of the transformation, so $A = A'$ and $C = C'$. However, the point $B$ is mapped to the point $B'$ which is not on the line segment from $A'$ to $C'$. Thus, the image of the line segment is not straight.

One might question at this point why it is undesirable for lines to map to curves. The answer to this question lies in the way the fourth stage of the graphics rendering pipeline works. In the fourth stage, the endpoints of a line
Figure II.25: The undesirable transformation of a line to a curve. The mapping used is \( \langle x, y, z \rangle \mapsto \langle -d \cdot x/z, -d \cdot y/z, z \rangle \). The points \( A \) and \( C \) are fixed by the transformation and \( B \) is mapped to \( B' \). The dotted curve is the image of the line segment \( AC \). (The small unlabeled circles show the images of \( A \) and \( B \) under the mapping of Figure II.24.)

segment are used to place a line in the screen space. This line in screen space typically has not only a position on the screen, but also depth (distance) values stored in a depth buffer.\(^{10}\) When the fourth stage processes a line segment, say as shown in Figure II.25, it is given only the endpoints \( A' \) and \( C' \) as points \( \langle x_A, y_A, z_A \rangle \) and \( \langle x_C, y_C, z_C \rangle \). It then uses linear interpolation to determine the rest of the points on the line segment. This then gives an incorrect depth to intermediate points such as \( B' \). With incorrect depth values, the hidden surface algorithm can fail in dramatically unacceptable ways, since the depth buffer values are used to determine which points are in front of other points.

Thus, we need another way to handle depth information. In fact, it is enough to find a definition of a “fake” distance or a “pseudo-distance” function which has the following two properties:

1. The pseudo-distance preserves relative distances, and
2. It causes lines to map to lines.

As it turns out, a good choice for this pseudo-distance is any function of the form:

\[
\text{pseudo-dist}(z) = A + B/z,
\]

with \( A \) and \( B \) constants such that \( B < 0 \). Since \( B < 0 \), property 1 certainly holds, as \( \text{pseudo-dist}(z_1) < \text{pseudo-dist}(z_2) \) holds whenever \( z_1 < z_2 \).

\(^{10}\)Other information, such as color values, is also stored along with depth, but this does not concern the present discussion.
II.3.2. Perspective transformations

It is a common convention to choose the values for $A$ and $B$ so that points on the near and far clipping planes have pseudo-distances equal to $+1$ and $-1$, respectively. The near and far clipping planes have $z = -n$ and $z = -f$, so we need:

$$ \text{pseudo-dist}(-n) = A - B/n = 1 \quad \text{pseudo-dist}(-f) = A - B/f = -1. $$

Solving these two equations for $A$ and $B$ yields

$$ A = -\frac{(f + n)}{f - n} \quad \text{and} \quad B = -\frac{2fn}{f - n} \quad (\text{II.15}) $$

Before discussing property 2, it is helpful to see how this definition of the pseudo-distance function fits into the framework of homogeneous representation of points. With the use of the pseudo-dist function, the perspective transformation becomes the mapping $$ \langle x, y, z \rangle \mapsto \langle -d \cdot x/z, -d \cdot y/z, A + B/z \rangle. $$

We can rewrite this in homogeneous coordinates as $$ \langle x, y, z, 1 \rangle \mapsto \langle d \cdot x, d \cdot y, -A \cdot z - B, -z \rangle \quad (\text{II.16}) $$

since multiplying through by $(-z)$ does not change the point represented by the homogeneous coordinates. More generally, because the homogeneous representation $\langle x, y, z, w \rangle$ is equivalent to $\langle x/w, y/w, z/w, 1 \rangle$, the mapping (II.16) acting on this point is

$$ \langle x/w, y/w, z/w, 1 \rangle \mapsto \langle d \cdot x/w, d \cdot y/w, -A \cdot (z/w) - B, -z/w \rangle, $$

and, multiplying both sides by $w$, this becomes

$$ \langle x, y, z, w \rangle \mapsto \langle d \cdot x, d \cdot y, -A \cdot z + B \cdot w, -z \rangle. $$

Thus, we have established that the perspective transformation incorporating the pseudo-dist function is represented by the following $4 \times 4$ homogeneous matrix:

$$ \begin{pmatrix}
    d & 0 & 0 & 0 \\
    0 & d & 0 & 0 \\
    0 & 0 & -A & -B \\
    0 & 0 & 1 & 0
\end{pmatrix} \quad (\text{II.17}) $$

The fact that the perspective transformation based on pseudo-distance can be expressed as a $4 \times 4$ matrix has two unexpected benefits. First, homogeneous matrices provide a uniform framework for representing both affine transformations and perspective transformations. Second, in Section II.3.3, we shall prove the following theorem:

\textbf{Theorem II.9} The perspective transformation represented by the $4 \times 4$ matrix (II.17) maps lines to lines.
In choosing a perspective transformation, it is important to select values for $n$ and $f$, the near and far clipping plane distances, so that all the desired objects are included in the field of view. At the same time, it is also important not to choose the near clipping plane to be too near, or the far clipping plane to be too distant. The reason is that the depth buffer values need to have enough resolution so as to allow different (pseudo)distance values to be distinguished. To understand how the use of pseudo-distance affects how much resolution is needed to distinguish between different distances, consider the graph of pseudo-distance versus distance in Figure II.26. Qualitatively, it is clear from the graph that pseudo-distance varies faster for small distance values than for large distance values (since the graph of the pseudo-distance function is sloping more steeply at smaller distances than at larger distances). Therefore, the pseudo-distance function is better at distinguishing differences in distance at small distances than at large distances. In most applications this is good, since, as a general rule, small objects tend to be close to the viewpoint, whereas more distant objects tend to either be larger or, if not larger, then errors in depth comparisons for distant objects make less noticeable errors in the graphics image.

It is common for stage four of the rendering pipeline to convert the pseudo-distance into a value in the range 0 to 1, with 0 used for points at the near clipping plane and with 1 representing points at the far clipping plane. This number, in the range 0 to 1, is then represented in fixed point, binary notation, i.e., as an integer, with 0 representing the value at the near clipping plane and the maximum integer value representing the value at the far clipping plane. In modern graphics hardware systems, it is common to use a 32 bit integer to store the depth information, and this gives sufficient depth resolution to allow the hidden surface calculations to work well in most situations. That is, it will work well provided the near and far clipping distances are chosen wisely. Older systems used 16 bit depth buffers, and this tended to occasionally cause resolution problems. By comparison, the usual single-precision floating point numbers have 24 bits of resolution.
II.3.3 Mapping lines to lines

As was discussed in the previous section, the fact that perspective transformations map lines in 3-space to lines in screen space is important for interpolation of depth values in the screen space. In fact, more than this is true: any transformation which is represented by a $4\times 4$ homogeneous matrix maps lines in 3-space to lines in 3-space. Since the perspective maps are represented by $4\times 4$ matrices, as shown by Equation (II.17), the same is true a fortiori of perspective transformations.

**Theorem II.10** Let $M$ be a $4\times 4$ homogeneous matrix acting on homogeneous coordinates for points in $\mathbb{R}^3$. If $L$ is a line in $\mathbb{R}^3$, then the image of $L$ under the transformation represented by $M$, if defined, is either a line or a point in $\mathbb{R}^3$.

This immediately gives the following corollary.

**Corollary II.11** Perspective transformations map lines to lines.

For proving Theorem II.10, the most convenient way to represent the three dimensional projective space is as the set of linear subspaces of the Euclidean space $\mathbb{R}^4$, as was described in Section II.2.7. The “points” of the three dimensional projective space are the one dimensional subspaces of $\mathbb{R}^4$. The “lines” of the three dimensional projective space are the two dimensional subspaces of $\mathbb{R}^4$. The “planes” of the three dimensional projective geometry are the three dimensional subspaces of $\mathbb{R}^4$.

The proof of Theorem II.10 is now immediate. Since $M$ is represented by a $4\times 4$ matrix, it acts linearly on $\mathbb{R}^4$. Therefore, $M$ must map a two dimensional subspace representing a line onto a subspace of dimension at most two: i.e., onto either a two dimensional subspace representing a line, or a one dimensional subspace representing a point, or a zero dimensional subspace. In the last case, the value of $M$ on points on the line is undefined, since the point $\langle 0, 0, 0, 0 \rangle$ is not a valid set of homogeneous coordinates for a point in $\mathbb{R}^3$.

II.3.4 Another use for projection: shadows

In the next chapter, we shall study local lighting and illumination models. These lighting models, by virtue of their tracking only local features, cannot handle phenomena such as shadows or indirect illumination. There are global methods for calculating lighting that do handle shadows and indirect illumination (see chapters X and XII), but these global methods are often computationally very difficult, and cannot be done with ordinary OpenGL commands in any event. There are also some multi-pass rendering techniques for rendering shadows that can be used in OpenGL (see Section X.3).

An alternative way to cast shadows that works well for casting shadows onto flat, planar surfaces is to render the shadow of an object explicitly. This can be done in OpenGL by setting the current color to black (or whatever shadow color is desired), and then drawing the shadow as a flat object on
Figure II.27: A light is positioned at \( \langle 0, y_0, 0 \rangle \). An object is positioned at \( \langle x, y, z \rangle \). The shadow of the point is projected to the point \( \langle x', 0, z' \rangle \), where \( x' = x/(1 - y/y_0) \) and \( z' = z/(1 - y/y_0) \).

This has several advantages, chief among them being that it requires very little coding effort. One can merely render the object twice: once in its proper location in 3-space, and once with the model view matrix set to project it down flat onto the plane. This handles arbitrarily complex shapes properly, including objects that contain holes.

To determine what the model view matrix should be for shadow projections, suppose that the light is positioned at \( \langle 0, y_0, 0 \rangle \), that is, at height \( y_0 \) up the \( y \)-axis, and that the plane of projection is the \( xz \)-plane where \( y = 0 \). It is not difficult to see using similar triangles that the projection transformation needed to cast shadows should be (see Figure II.27)

\[
\langle x, y, z \rangle \mapsto \langle \frac{x}{1 - y/y_0}, 0, \frac{z}{1 - y/y_0} \rangle.
\]

This transformation is represented by the following homogeneous matrix:

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & -\frac{1}{y_0} & 0 & 1
\end{pmatrix}.
\]

Exercise II.26 Prove the correctness of the above formula for the shadow transformation and the homogeneous matrix representation.
II.3.5 z-fighting

One potential pitfall with drawing shadows on a flat plane is that if the shadow is drawn exactly coincident with the plane, then z-fighting may cause the plane and shadow to show through each other. z-fighting occurs when two objects are drawn at the same depth from the viewer: due to roundoff errors, it can happen that some pixel positions have the first object closer than the other and other pixels have the second closer than the first. The effect is a pattern of pixels where one object shows through the other. One way to combat z-fighting is to slightly lift the shadow up from the plane, but this can cause problems from some viewpoints where the gap between the plane and the shadow can become apparent. To solve this problem, you can use the OpenGL polygon offset feature. The polygon offset mode perturbs the depth values (pseudo-distance values) of points before performing depth testing against the pixel buffer. This allows the depth values to be perturbed for depth comparison purposes without affecting the position of the object on the screen.

To use polygon offset to draw a shadow on a plane, you would first enable polygon offset mode with a positive offset value, draw the plane, and disable polygon offset mode. Finally, you would render the shadow without any polygon offset.

The OpenGL commands for enabling polygon offset mode are

```glsl
glPolygonOffset(1.0, 1.0);
glEnable(GL_POLYGON_OFFSET_FILL, GL_POLYGON_OFFSET_LINE, GL_POLYGON_OFFSET_POINT);
```

Similar options for `glDisable` will disable polygon offset. The amount of offset is controlled by the `glPolygonOffset()` command; setting both parameters to 1.0 is a good choice in most cases where you wish to increase the distance from the viewer. You can also use negative values -1.0 to use offset to pull objects closer to the viewer. For details on what these parameters mean, see the OpenGL documentation.

II.3.6 The OpenGL perspective transformations

XXX THIS SECTION NEEDS A REWRITE TO USE MODERN OPENGL.

OpenGL provides special functions for setting up viewing transformations as either orthographic projections or perspective transformations. The direction and location of the camera can be controlled with the same affine transformations used for modeling transformations, and in addition, there is a function, `gluLookAt`, that provides a convenient method to set the camera location and view direction.

The basic OpenGL command for creating an orthographic projection is:

```glsl
glOrtho(float ℓ, float r, float b, float t, float n, float f);
```
As discussed in Section II.3.1, the intent of the glOrtho command is to set up the camera or eye position so as to be looking down the negative $z$-axis, at the rectangular prism of points with $\ell \leq x \leq r$ and $b \leq y \leq t$ and $n \leq -z \leq f$. Any part of the scene which lies outside this prism is clipped and not displayed. In particular, objects which are closer than the near clipping plane, defined by $(-z) = n$, are not visible, and do not even obstruct the view of more distant objects. In addition, objects further than the far clipping plane, defined by $(-z) = f$ are likewise not visible. Of course, objects, or parts of objects, outside the left, right, bottom, and top planes are not visible.

Internally, the effect of the glOrtho command is to multiply the current matrix, which is usually the projection matrix $P$, by the matrix

$$
S = \begin{pmatrix}
\frac{2}{r - \ell} & 0 & 0 & \frac{r + \ell}{r - \ell} \\
0 & \frac{2}{t - b} & 0 & \frac{t + b}{t - b} \\
0 & 0 & \frac{-2}{f - n} & \frac{f + n}{f - n} \\
0 & 0 & 0 & 1
\end{pmatrix}.
$$

This is the same as the matrix shown in Equation (II.12) on page 82, except the signs of the third row are reversed. This is because OpenGL’s convention for the meaning of points in the $2 \times 2 \times 2$ cube is that $z = -1$ is used for the closest objects and $z = 1$ is used for the farthest objects, so the $z$ values need to be negated. As usual, the multiplication is on the right, i.e., it has the effect of performing the assignment $P = P \cdot S$, where $P$ is the current matrix (presumably the projection matrix).

A special case of orthographic projections in OpenGL is provided by the following function:

```c
gluOrtho2D( float \ell, float r, float b, float t );
```

gluOrtho2D is exactly like glOrtho, but with $n = -1$ and $f = 1$. That is to say, gluOrtho2D views points which have $z$-value between $-1$ and $1$. Usually gluOrtho2D is used when drawing two dimensional figures that lie in the $xy$-plane, with $z = 0$. It is a convenience function, along with glVertex2*, intended for drawing two dimensional objects.

OpenGL has two commands that implement perspective transformations, glFrustum and gluPerspective. Both these commands make the usual assumption that the viewpoint is at the origin and the view direction is towards the negative $z$-axis. The most basic command is the glFrustum command, which has the following syntax:

```c
glFrustum( float \ell, float r, float b, float t, float n, float f );
```
II.3.6. The OpenGL perspective transformations

The near clipping plane is \( z = -n \). The far clipping plane is \( z = -f \). The frustum is the set of points satisfying (II.18) and (II.19).

A frustum is a six-sided geometric shape formed from a rectangular pyramid by removing a top portion. In this case, the frustum consists of the points \( \langle x, y, z \rangle \) satisfying the conditions (II.18) and (II.19). (Refer to Figure II.28):

a. The points lie between the near and far clipping planes:

\[
n \leq -z \leq f. \tag{II.18}
\]

b. The perspective mapping, which performs a perspective projection onto the near clipping plane, maps \( \langle x, y, z \rangle \) to a point \( \langle x', y', z' \rangle \) with \( \ell \leq x' \leq r \) and \( b \leq y' \leq t \). In view of Equation (II.13), this is the same as

\[
\ell \leq \frac{n \cdot x}{-z} \leq r \quad \text{and} \quad b \leq \frac{n \cdot y}{-z} \leq t. \tag{II.19}
\]

The effect of the \texttt{glFrustum} command is to form the matrix

\[
S = \begin{pmatrix}
\frac{2n}{r - \ell} & 0 & \frac{r + \ell}{r - \ell} & 0 \\
0 & \frac{2n}{t - b} & \frac{t + b}{t - b} & 0 \\
0 & 0 & -\frac{(f + n)}{f - n} & -2fn \\
0 & 0 & -1 & 0
\end{pmatrix} \tag{II.20}
\]
and then multiply the current matrix (usually the projection matrix) on the right by $S$. This matrix $S$ is chosen so that the frustum is mapped onto the $2 \times 2 \times 2$ cube centered at the origin. The formula for the matrix $S$ is obtained in a manner similar to the derivation of the Equation (II.17) for the perspective transformation in Section II.3.2. There are three differences between equations (II.20) and (II.17). First, the OpenGL matrix causes the final $x$ and $y$ values to lie in the range $-1$ to $1$ by performing appropriate scaling and translation: the scaling is caused by the first two diagonal entries, and the translation is effected by the top two values in the third column. The second difference is that the values in the third row are negated since OpenGL negates the $z$ values from our own convention. The third difference is that (II.17) was derived under the assumption that the view frustum was centered on the $z$-axis. For glFrustum, this happens if $\ell = -r$ and $b = -t$. But, glFrustum also allows more general view frustums that are not centered on the $z$-axis.

**Exercise II.27** Derive formula (II.20) for the glFrustum matrix.

OpenGL provides a function gluPerspective, which can be used as an alternative to glFrustum. gluPerspective limits you to perspective transformations for which the $z$-axis is in the center of the field of view; but this is usually what is wanted anyway. gluPerspective works by making a single call to glFrustum. The usage of gluPerspective is

\[
\text{gluPerspective}(\text{float } \theta, \text{float aspectRatio, float } n, \text{float } f); \]

where $\theta$ is an angle (measured in degrees) specifying the vertical field of view. That is to say, $\theta$ is the solid angle between the top bounding plane and the bottom bounding plane of the frustum in Figure II.28. The *aspect ratio* of an image is the ratio of its width to its height, so the parameter aspectRatio specifies the ratio of the width of the frustum to the height of the frustum. It follows that a call to gluPerspective is equivalent to calling glFrustum with

\[
\begin{align*}
    t &= n \cdot \tan(\theta/2) \\
    b &= -n \cdot \tan(\theta/2) \\
    r &= (\text{aspectRatio}) \cdot t \\
    \ell &= (\text{aspectRatio}) \cdot b
\end{align*}
\]

As an example of the use of gluPerspective, consider the following code fragment from the Solar.c program:

```c
// Called when the window is resized
// Sets up the projection view matrix (somewhat poorly, however)
void ResizeWindow(int w, int h)
{
    glViewport( 0, 0, w, h ); // Viewport uses whole window
    // ...
}
```
II.3.6. The OpenGL perspective transformations (Draft A.2.d)

```c
float aspectRatio;
h = (h == 0) ? 1 : h; // Avoid divide by zero
aspectRatio = (float)w/(float)h;

// Set up the projection view matrix
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, aspectRatio, 1.0, 30.0);
```

The routine `ResizeWindow` is called whenever the program window is resized, and is given the new width and height of the window in pixels. `ResizeWindow` first specifies that the **viewport** is to be the entire window, giving its lower left-hand corner as the pixel with coordinates 0, 0, and its upper right-hand corner as the pixel with coordinates \( w - 1, h - 1 \). The viewport is the area of the window in which the OpenGL graphics are displayed. The routine then makes the projection matrix the active matrix, restores it to the identity, and calls `gluPerspective`. This call picks a vertical field of view angle of 60 degrees and makes the aspect ratio of the viewed scene equal to the aspect ratio of the viewport.

It is illuminating to consider potential problems with the way `gluPerspective` is used in the sample code. First, the vertical field of view being 60 degrees is probably higher than optimal. By making the field of view too large, the effects of perspective are exaggerated, making the image appear as if it were viewed through a wide-angle or “fish-eye” lens. On the other hand, if the field of view is too small, then the image does not have enough perspective and looks too close to an orthographic projection. Ideally, the field of view should be chosen to be equal to the angle that the final screen image takes up in the field of view of the person looking at the image. Of course, to set the field of view precisely in this way, one would need to know the dimensions of the viewport (in inches, say) and the distance of the person from the screen. In practice, one can usually only guess at these values.

The second problem with the above sample code is that the field of view angle is controlled by only the up-down, \( y \)-axis, direction. To see why this is a problem, try running the `Solar` program and resizing the window first to be wide and short, and then to be narrow and tall. In the second case, only a small part of the solar system will be visible.

**Exercise II.28** Rewrite the `ResizeWindow` function in `Solar.c` so that the entire solar system is visible, no matter what the aspect ratio of the window is.

---

11 This is set up by the earlier call to `glutReshapeFunc` in the main program of `Solar.c`.
12 Pixel positions are numbered by values from 0 to \( h - 1 \) from the bottom row of pixels to the top row, and are numbered from 0 to \( w - 1 \) from the left column of pixels to the right column.
gluLookAt, that provides a convenient method to set the camera location and view direction. OpenGL provides another function gluLookAt to make it easy to position a viewpoint at an arbitrary location in 3-space, looking in an arbitrary direction with an arbitrary orientation. This function is called with nine parameters:

\[
\text{gluLookAt}(\text{eye}_x, \text{eye}_y, \text{eye}_z, \text{center}_x, \text{center}_y, \text{center}_z, \\
\text{up}_x, \text{up}_y, \text{up}_z);
\]

The three “eye” values specify a location in 3-space for the viewpoint. The three “center” values must specify a different location so that the view direction is towards the center location. The three “up” values specify an upward direction for the y-axis of the viewer. It is not necessary for the “up” vector to be orthogonal to the vector from the eye to the center, but it must not be parallel to it. The gluLookAt command should be used when the current matrix is the model view matrix, not the projection matrix. This is because the viewer should always be placed at the origin, in order for OpenGL’s lighting to work properly.

**Exercise II.29** Change the SolarModern program on the book’s webpage to use gluLookAt to set the View matrix instead of the translation and rotation.

### II.4 Additional exercises

**Exercise II.30** Define functions

- \( f_1 : \langle x_1, x_2 \rangle \mapsto \langle x_2, -x_1 \rangle, \) and
- \( f_2 : \langle x_1, x_2 \rangle \mapsto \langle 2x_1, -\frac{1}{2}x_2 \rangle, \) and
- \( f_3 : \langle x_1, x_2 \rangle \mapsto \langle x_1 - \frac{1}{2}x_2, x_2 \rangle, \) and
- \( f_4 : \langle x_1, x_2 \rangle \mapsto \langle x_2, x_1 - \frac{1}{2}x_2 \rangle. \)

Verify (but do not show your work) that each \( f_i \) is linear. Draw figures showing how these four functions transform the “F”-shape. Your four figures should be similar to the images in Figures II.3, II.4 and II.5. Label enough of the points to make it clear your answer is correct.

**Exercise II.31** Consider the functions \( f_1, \ldots, f_4 \) from the previous Exercise II.30. Which of the four \( f_i \)'s are affine? Which of the four \( f_i \)'s are rigid? Which of the \( f_i \)'s are orientation-preserving?

**Exercise II.32** Prove the angle sum formulas for \( \sin \) and \( \cos \):

\[
\sin(\theta + \varphi) = \sin \theta \cos \varphi + \cos \theta \sin \varphi
\]
\[
\cos(\theta + \varphi) = \cos \theta \cos \varphi - \sin \theta \sin \varphi,
\]
by considering what the rotation \( R_\theta \) does to the point \( x = \langle \cos \varphi, \sin \varphi \rangle \) and using the matrix representation of \( R_\theta \) given by Equation II.2.

**Exercise II.33** Let \( A \) be the transformation of \( \mathbb{R}^2 \) shown in Figure II.29. Is \( A \) linear? Is \( A \) rigid? Is \( A \) orientation preserving? Give a matrix representing \( A \): either a \( 2 \times 2 \) matrix if \( A \) is linear, or a \( 3 \times 3 \) matrix if \( A \) is only affine.

**Exercise II.34** Let \( A \) be the transformation of \( \mathbb{R}^2 \) shown in Figure II.30. Is \( A \) linear? Is \( A \) rigid? Is \( A \) orientation preserving? Give a matrix representing \( A \): either a \( 2 \times 2 \) matrix if \( A \) is linear, or a \( 3 \times 3 \) matrix if \( A \) is only affine.

**Exercise II.35** Let an affine transformation \( C \) of \( \mathbb{R}^2 \) act on the “F” shape as shown in Figure II.31. The “F” shape has been scaled uniformly by a factor of \( \frac{1}{2} \) and moved to a new position and orientation. Express \( A \) as a composition of transformations of the forms \( T_\mathbf{u} \), \( R_\theta \), and \( S_{\frac{1}{2}} \) (for appropriate vectors \( \mathbf{u} \) and angles \( \theta \)).

**Exercise II.36** Express the affine transformation in Figure II.5 as a \( 3 \times 3 \) matrix acting on homogeneous equations. [Hint: The first part will be easy if you have already worked Exercise II.3.]

**Exercise II.37** Let \( f \) be the affine transformation in Figure II.5. Express \( f(\mathbf{x}) \) in the form \( N\mathbf{x} + \mathbf{v} \) where \( N \) is a \( 2 \times 2 \) matrix. Also give the \( 3 \times 3 \) matrix representing \( f^{-1} \) acting on homogeneous coordinates.
Exercise II.38 Let $A$ be the transformation $S_{(1/2,1/2,3/2)} \circ T_{(4,2,0)}$. What is $A((0,0,0))$? Give the $4 \times 4$ matrix representation for $A$.

Exercise II.39 Let $B$ be the transformation $R_{\pi,0} \circ T_{(1,2,0)}$. What is $B((0,0,0))$? Give the $4 \times 4$ matrix representation for $B$.

Exercise II.40 Give the $4 \times 4$ matrix representations of $T_{i-j} \circ R_{\pi/2,k}$ and of $R_{\pi/2,k} \circ T_{i-j}$. What are the images of $0, i, j$ and $k$ under these two compositions? (It is recommended to do this by visualizing the action of the composed transformations. The problem can also be worked by multiplying matrices, but in this case it is highly recommended to try visualizing afterwards.)

Exercise II.41 Give the $4 \times 4$ matrix representations of $T_{i-j} \circ S_2$ and of $S_2 \circ T_{i-j}$. Recall $S_2$ is a uniform scaling transformation. What are the images of $0, i, j$ and $k$ under these two compositions. As in the previous exercise, this problem can be solved by either visualization or matrix multiplication.

Exercise II.42 Let $A$ be the linear transformation such that $A((x,y,z)) = (z,x,y)$. That is, $A$ cyclically permutes the three coordinate axes. What $4 \times 4$ matrix represents $A$ over homogeneous coordinates? Express $A$ as a rotation $A = R_{\theta,u}$ by giving values for $\theta$ and $u$. [Hint: You can visualize this by considering the symmetries in the way $A$ acts.]

Exercise II.43 Let $B$ be the transformation $(x,y,z) \mapsto (\frac{1+x}{1-y}, 1, 0, \frac{z}{1-y})$. Give a $4 \times 4$ matrix which represents this transformation over homogeneous coordinates. (When $0 < y < 1$, $B$ gives the transformation for a shadow cast from a light at $(−1,1,0)$ onto the plane $y = 0$. You do not need to use this fact to work the problem!)