

(excerpt For ECE660, Fall 1999) - F.S.Hill,Jr.

CHAPTER 1. Introduction to Computer Graphics

*"Begin at the beginning," the King said gravely,
"and go on till you come to the end; then stop."
Lewis Carroll, Alice in Wonderland*

*The machine does not isolate
the man from the great
problems of nature
but plunges him more deeply
into them
Antoine de Saint-Exupéry*

*"Any sufficiently advanced
technology is indistinguishable from magic."
Arthur C. Clarke*

1.1 What is computer graphics?

Good question. People use the term “computer graphics” to mean different things in different contexts. Most simply, computer graphics are pictures that are generated by a computer. Everywhere you look today there are examples to be found, especially in magazines and on television. This book was typeset using a computer: every character (even this one: G) was “drawn” from a library of character shapes stored in computer memory. Books and magazines abound with pictures created on a computer. Some look so natural you can’t distinguish them from photographs of a “real” scene. Others have an artificial or surreal feeling, intentionally fashioned to achieve some visual effect. And movies today often show scenes that never existed, but were carefully crafted by computer, mixing the real and the imagined.

“Computer graphics” also refers to the tools used to make such pictures. The purpose of this book is to show what the tools are and how to apply them. There are both hardware and software tools. Hardware tools include video monitors and printers that display graphics, as well as input devices like a mouse or trackball that let a user point to items and draw figures. The computer itself, of course, is a hardware tool, along with its special circuitry to facilitate graphical display or image capture.

As for software tools, you are already familiar with the usual ones: the operating system, editor, compiler, and debugger, that are found in any programming environment. For graphics there must also be a collection of “graphics routines” that produce the pictures themselves. For example, all graphics libraries have functions to draw a simple line or circle (or characters such as G). Some go well beyond this, containing functions to draw and manage windows with pull-down menus and dialog boxes, or to set up a “camera” in a three-dimensional coordinate system and to make “snapshots” of objects stored in some data base.

In this book we show how to write programs that utilize graphics libraries, and how to add functionality to them. Not too long ago, programmers were compelled to use highly “device dependent” libraries, designed for use on one specific computer system with one specific display device type. This made it very difficult to “port” a program to another system, or to use it with another device: usually the programmer had to make substantial changes to the program to get it to work, and the process was time-consuming and highly error-prone. Happily the situation is far better today. Device independent graphics libraries are now available that allow the programmer to use a common set of functions within an application, and to run the same application on a variety of systems and displays. OpenGL is such a library, and serves as the main tool we use in this book. The OpenGL way of creating graphics is used widely in both universities and industry. We begin a detailed discussion of it in Chapter 2.

Finally, “computer graphics” often means the whole **field of study** that involves these tools and the pictures they produce. (So it’s also used in the singular form: “computer graphics is...”). The field is often acknowledged to have started in the early 1960’s with Ivan Sutherland’s pioneering doctoral thesis at MIT on ‘Sketchpad’ [ref]. Interest in graphics grew quickly, both in academia and industry, and there were rapid advances in display technology and in the algorithms used to manage pictorial information. The special interest group in graphics, SIGGRAPH¹, was formed in 1969, and is very active today around the world. (The must-not-miss annual SIGGRAPH meeting now attracts 30,000 participants a year.) More can be found at <http://www.siggraph.org>. Today there are hundreds of companies around the world having some aspect of computer graphics as their main source of revenue, and the subject of computer graphics is taught in most computer science or electrical engineering departments.

Computer graphics is a very appealing field of study. You learn to write programs that create pictures, rather than streams of text or numbers. Humans respond readily to pictorial information, and are able to absorb much more information from pictures than from a collection of numbers. Our eye-brain systems are highly attuned to recognizing visual patterns. Reading text is of course one form of pattern recognition: we instantly recognize character shapes, form them into words, and interpret their meaning. But we are even more acute when glancing at a picture. What might be an inscrutable blather of numbers when presented as text becomes an instantly recognizable shape or pattern when presented graphically. The amount of information in a picture can be enormous. We not only recognize what’s “in it”, but also glean a world of information from its subtle details and texture

People study computer graphics for many reasons. Some just want a better set of tools for plotting curves and presenting the data they encounter in their other studies or work. Some want to write computer-animated games, while others are looking for a new medium for artistic expression. Everyone wants to be more productive, and to communicate ideas better, and computer graphics can be a great help.

There is also the “input” side. A program generates output — pictures or otherwise — from a combination of the algorithms executed in the program and the data the user inputs to the program. Some programs accept input crudely through characters and numbers typed at the keyboard. Graphics program, on the other hand, emphasize more familiar types of input: the movement of a mouse on a desktop, the strokes of a pen on a drawing tablet, or the motion of the user’s head and hands in a virtual reality setting. We examine many techniques of “interactive computer graphics” in this book; that is, we combine the techniques of natural user input with those that produce pictorial output.

(Section 1.2 on uses of Computer Graphics deleted.)

1.3. Elements of Pictures Created in Computer Graphics.

What makes up a computer drawn picture? The basic objects out of which such pictures are composed are called **output primitives**. One useful categorization of these is:

- **polylines**
- **text**
- **filled regions**
- **raster images**

We will see that these types overlap somewhat, but this terminology provides a good starting point. We describe each type of primitive in turn, and hint at typical software routines that are used to draw it. More detail on these tools is given in later chapters, of course. We also discuss the various attributes of each output primitive. The **attributes** of a graphic primitive are the characteristics that affect how it appears, such as color and thickness.

¹ SIGGRAPH is a Special Interest Group in the ACM: the Association for Computing Machinery.

1.3.1. Polylines.

A polyline is a connected sequence of straight lines. Each of the examples in Figure 1.8 contain several polylines: a). one polyline extends from the nose of the dinosaur to its tail; the plot of the mathematical function is a single polyline, and the “wireframe” picture of a chess pawn contains many polylines that outline its shape .

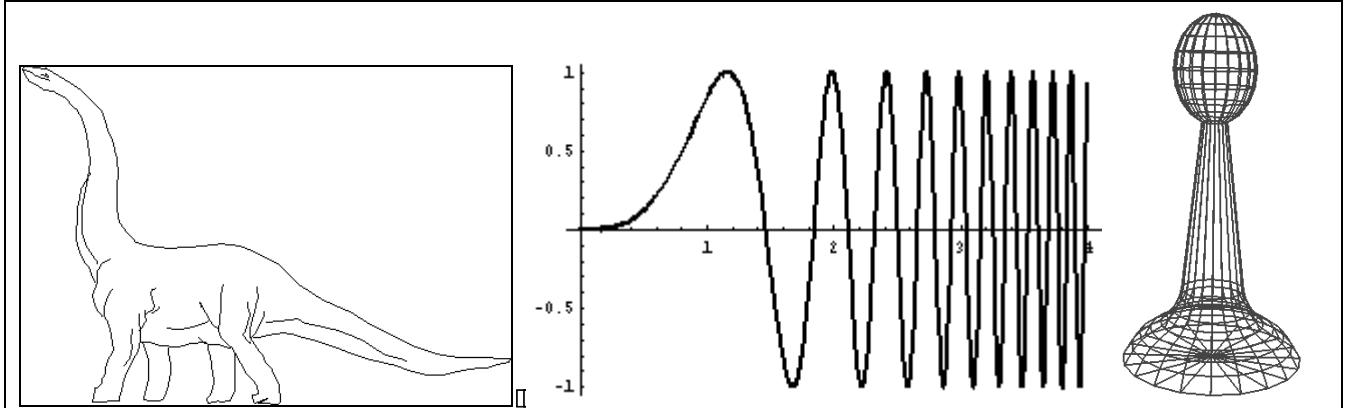


Figure 1.8. a). a polyline drawing of a dinosaur (courtesy of Susan Verbeck),
b). a plot of a mathematical function,
c). a wireframe rendering of a 3D object.

Note that a polyline can appear as a smooth curve. Figure 1.9 shows a blow-up of a curve revealing its underlying short line segments. The eye blends them into an apparently smooth curve.

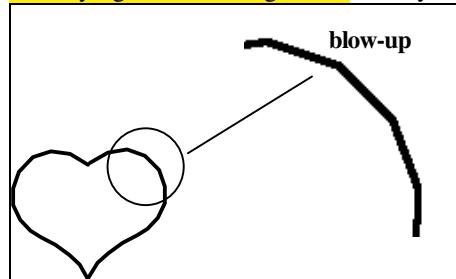


Figure 1.9. A curved line made up of straight line segments.

Pictures made up of polylines are sometimes called **line drawings**. Some devices, like a pen plotter, are specifically designed to produce line drawings.

The simplest polyline is a single straight line segment. A line segment is specified by its two endpoints, say (x_1, y_1) and (x_2, y_2) . A drawing routine for a line might look like `drawLine(x1, y1, x2, y2);`

It draws a line between the two endpoints. We develop such a tool later, and show many examples of its use. At that point we get specific about how coordinates like x_1 are represented (by integers or by real numbers), and how colors can be represented in a program.

A special case arises when a line segment shrinks to a single point, and is drawn as a “dot”. Even the lowly dot has important uses in computer graphics, as we see later. A dot might be programmed using the routine `drawDot(x1, y1);`

When there are several lines in a polyline each one is called an **edge**, and two adjacent lines meet at a **vertex**. The edges of a polyline can cross one another, as seen in the figures. Polylines are specified as a list of vertices, each given by a coordinate pair:

$$(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n) \quad (1.1)$$

For instance, the polyline shown in Figure 1.10 is given by the sequence (2, 4), (2, 11), (6, 14), (12, 11), (12, 4), (what are the remaining vertices in this polyline?).

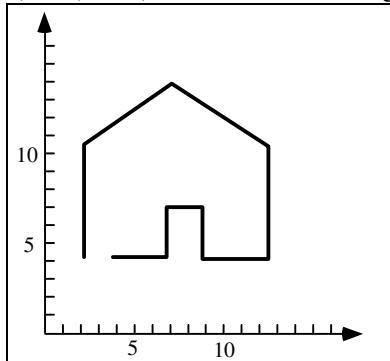


Figure 1.10. An example polyline.

To draw polylines we will need a tool such as: `drawPolyline(poly);`

where the variable `poly` is a list containing all the endpoints (x_i, y_i) in some fashion. There are various ways to capture a list in a program, each having its advantages and disadvantages.

A polyline need not form a closed figure, but if the first and last points are connected by an edge the polyline is a **polygon**. If in addition no two edges cross, the polygon is called **simple**. Figure 1.11 shows some interesting polygons; only A and D are simple. Polygons are fundamental in computer graphics, partly because they are so easy to define, and many drawing (rendering) algorithms have been finely tuned to operate optimally with polygons. Polygons are described in depth in Chapter 3.

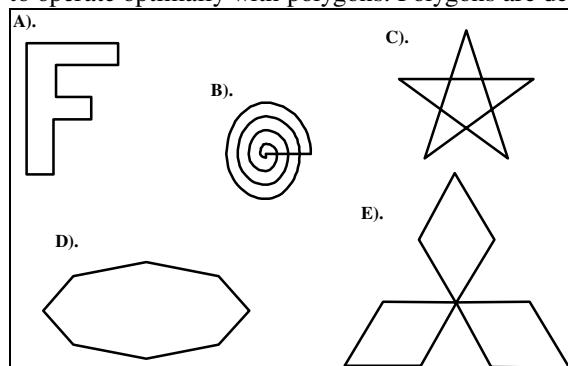


Figure 1.11. Examples of polygons..

Attributes of Lines and Polylines.

Important attributes of a polyline are the color and thickness of its edges, the manner in which the edges are dashed, and the manner in which thick edges blend together at their endpoints. Typically all of the edges of a polyline are given the same attributes.

The first two polylines in Figure 1.12 are distinguished by the line thickness attribute. The third polyline is drawn using dashed segments.

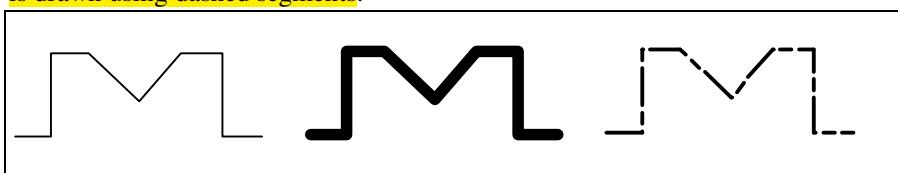


Figure 1.12. Polylines with different attributes.

When a line is thick its ends have shapes, and a user must decide how two adjacent edges “join”. Figure 1.13 shows various possibilities. Case a) shows “butt-end” lines that leave an unseemly “crack” at the joint. Case b) shows rounded ends on the lines so they join smoothly, part c) shows a mitered joint, and part d) shows a trimmed mitered joint. Software tools are available in some packages to allow the user to choose the type of joining. Some methods are quite expensive computationally.

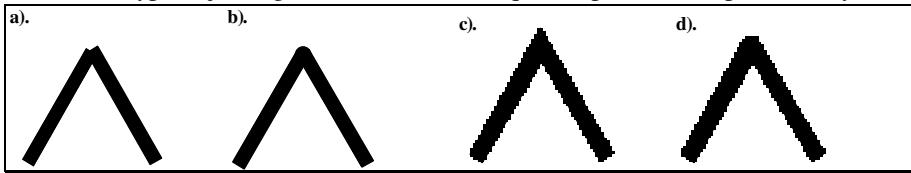


Figure 1.13. Some ways of joining two thick lines in a polyline.

The attributes of a polyline are sometimes set by calling routines such as `setDash(dash7)` or `setLineThickness(thickness)`.

1.3.2. Text.

Some graphics devices have two distinct display modes, a **text mode** and a **graphics mode**. The text mode is used for simple input/output of characters to control the operating system or edit the code in a program. Text displayed in this mode uses a built-in character generator. The character generator is capable of drawing alphabetic, numeric, and punctuation characters, and some selection of special symbols such as **♥**, **đ**, and **⊕**. Usually these characters can't be placed arbitrarily on the display but only in some row and column of a built-in grid.

A graphics mode offers a richer set of character shapes, and characters can be placed arbitrarily. Figure 1.14 shows some examples of text drawn graphically.

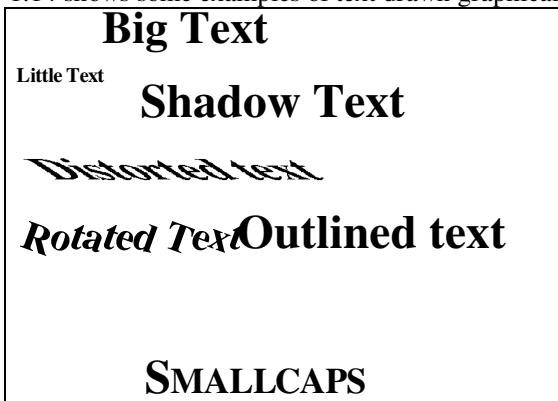


Figure 1.14. Some text drawn graphically.

A tool to draw a character string might look like: `drawString(x, y, string);` It places the starting point of the string at position (x, y) , and draws the sequence of characters stored in the variable `string`.

Text Attributes.

There are many text attributes, the most important of which are typeface, color, size, spacing, and orientation.

Font. A font is a specific set of character shapes (a **typeface**) in a particular style and size. Figure 1.15 shows various character styles.

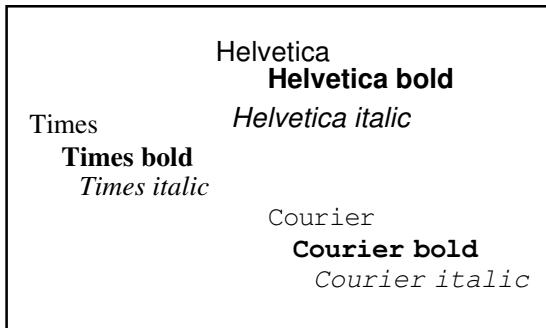


Figure 1.15. Some examples of fonts.

The shape of each character can be defined by a polyline (or more complicated curves such as Bezier curves – see Chapter 11), as shown in Figure 1.16a, or by an arrangement of dots, as shown in part b. Graphics packages come with a set of predefined fonts, and additional fonts can be purchased from companies that specialize in designing them.

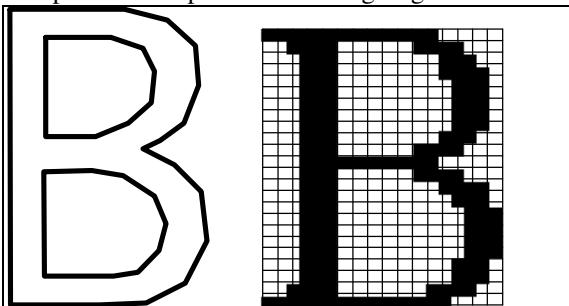


Figure 1.16. A character shape defined by a polyline and by a pattern of dots.

Orientation of characters and strings: Characters may also be drawn tilted along some direction. Tilted strings are often used to annotate parts of a graph. The graphic presentation of high-quality text is a complex subject. Barely perceptible differences in detail can change pleasing text into ugly text. Indeed, we see so much printed material in our daily lives that we subliminally expect characters to be displayed with certain shapes, spacings, and subtle balances.

1.3.3. Filled Regions

The **filled region** (sometimes called “fill area”) primitive is a shape filled with some color or pattern. The boundary of a filled region is often a polygon (although more complex regions are considered in Chapter 4). Figure 1.17 shows several filled polygons. **Polygon A** is filled with its edges visible, whereas **B** is filled with its border left undrawn. Polygons **C** and **D** are non-simple. **Polygon D** even contains polygonal holes. Such shapes can still be filled, but one must specify exactly what is meant by a polygon’s “interior”, since filling algorithms differ depending on the definition. Algorithms for performing the filling action are discussed in Chapter 10.

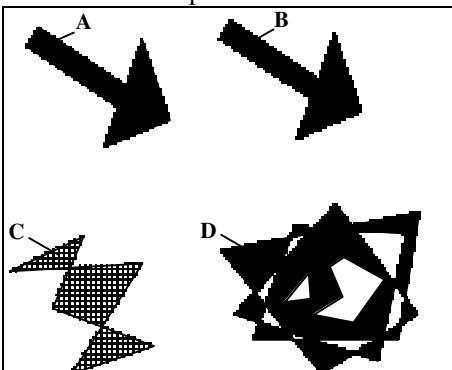


Figure 1.17. Examples of filled Polygons.

To draw a filled polygon one would use a routine like: `fillPolygon(poly, pattern);`

where the variable `poly` holds the data for the polygon - the same kind of list as for a polyline - and the variable `pattern` is some description of the pattern to be used for filling. We discuss details for this in Chapter 4.

Figure 1.18 shows the use of filled regions to shade the different faces of a 3D object. **Each polygonal “face” of the object is filled with a certain shade of gray that corresponds to the amount of light that would reflect off that face.** This makes the object appear to be bathed in light from a certain direction. Shading of 3D objects is discussed in Chapter 8.

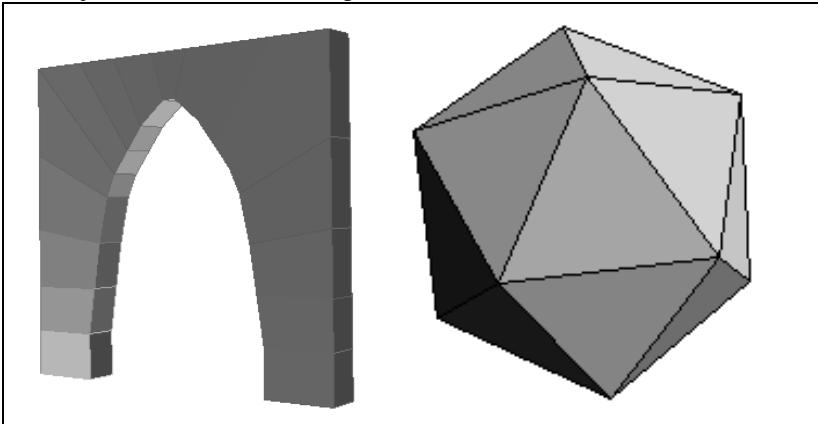


Figure 1.18. Filling polygonal faces of 3D objects to suggest proper shading.

The attributes of a filled region include the attributes of the enclosing border, as well as the pattern and color of the filling.

1.3.4. Raster Image.

Figure 1.19a shows a **raster image** of a chess piece. It is made up of many small “cells”, in different shades of gray, as revealed in the blow-up shown in Figure 1.19b. **The individual cells are often called “pixels”** (short for “picture elements”). Normally your eye can’t see the individual cells; it blends them together and synthesizes an overall picture.

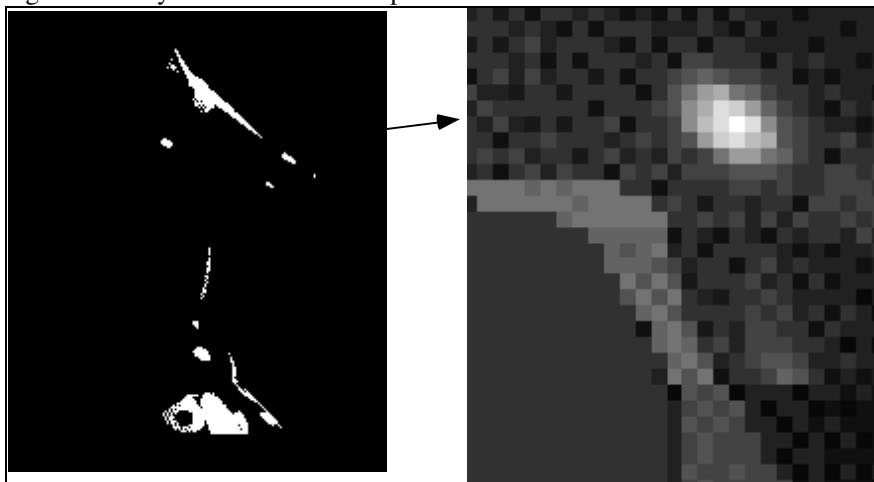


Figure 1.19. a). A raster image of a chess piece. b). A blow-up of the image. (Raytracing courtesy of Andrew Slater)

A raster image is stored in a computer as an array of numerical values. This array is thought of as being rectangular, with a certain number of rows and a certain number of columns. Each numerical value represents the value of the pixel stored there. The array as a whole is often called a “pixel map”. The term “bitmap” is also used, (although some people think this term should be reserved for pixel maps wherein each pixel is represented by a single bit, having the value 0 or 1.)

Figure 1.20 shows a simple example where a figure is represented by a 17 by 19 array (17 rows by 19 columns) of cells in three shades of gray. Suppose the three gray levels are encoded as the values 1, 2, and 7. Figure 1.20b shows the numerical values of the pixel map for the upper left 6 by 8 portion of the image.

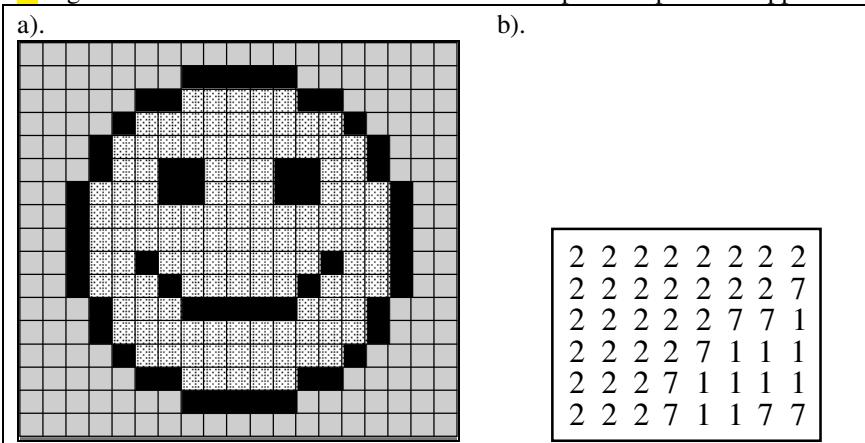


Figure 1.20. A simple figure represented as a bitmap.

How are raster images created? The three principal sources are:

1). Hand designed images.

A designer figures out what values are needed for each cell, and types them into memory. Sometimes a paint program can be used to help automate this: the designer can draw and manipulate various graphical shapes, viewing what has been made so far. When satisfied, the designer stores the result in a file. The icon above was created this way.

2). Computed Images.

An algorithm is used to “render” a scene, which might be modeled abstractly in computer memory. As a simple example, a scene might consist of a single yellow smooth sphere illuminated by a light source that emanates orange light. The model contains descriptions of the size and position of the sphere, the placement of the light source, and a description of the hypothetical “camera” that is to “take the picture”. The raster image plays the role of the film in the camera. In order to create the raster image, an algorithm must calculate the color of light that falls on each pixel of the image in the camera. This is the way in which ray traced images such as the chess piece in Figure 1.20 are created; see Chapter 16.

Raster images also frequently contain images of straight lines. A line is created in an image by setting the proper pixels to the line's color. But it can require quite a bit of computation to determine the sequence of pixels that “best fit” the ideal line between two given end points. Bresenham's algorithm (see Chapter 2) provides a very efficient approach to determining these pixels.

Figure 1.21a shows a raster image featuring several straight lines, a circular arc, and some text characters. Figure 1.21b shows a close-up of the raster image in order to expose the individual pixels that are “on” the lines. For a horizontal or vertical line the black square pixels line up nicely forming a sharp line. But for the other lines and the arc the “best” collection of pixels produces only an approximation to the “true” line desired. In addition, the result shows the dread “jaggies” that have a relentless presence in raster images.



Figure 1.21. a). a collection of lines and text. b). Blow-up of part a, having “jaggies”.

3). Scanned images.

A photograph or television image can be digitized as described above. In effect a grid is placed over the original image, and at each grid point the digitizer reads into memory the “closest” color in its repertoire. The bitmap is then stored in a file for later use. The image of the kitten in Figure 1.22 was formed this way.

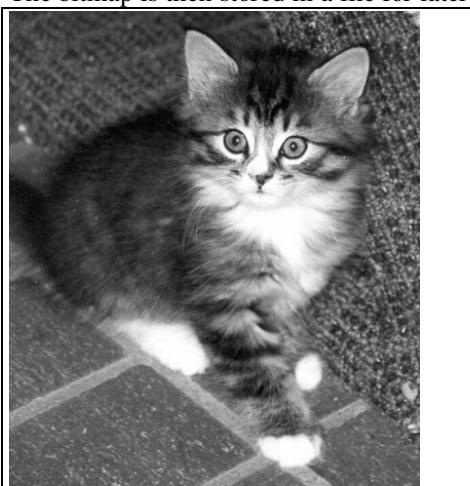


Figure 1.22. A scanned image.

Because raster images are simply arrays of numbers, they can be subsequently processed to good effect by a computer. For instance, Figure 1.23 shows three successive enlargements of the kitten image above.

These are formed by “pixel replication” (discussed in detail in Chapter 10). Each pixel has been replicated three times in each direction in part a; by six times in part b, and by 12 times in part c.

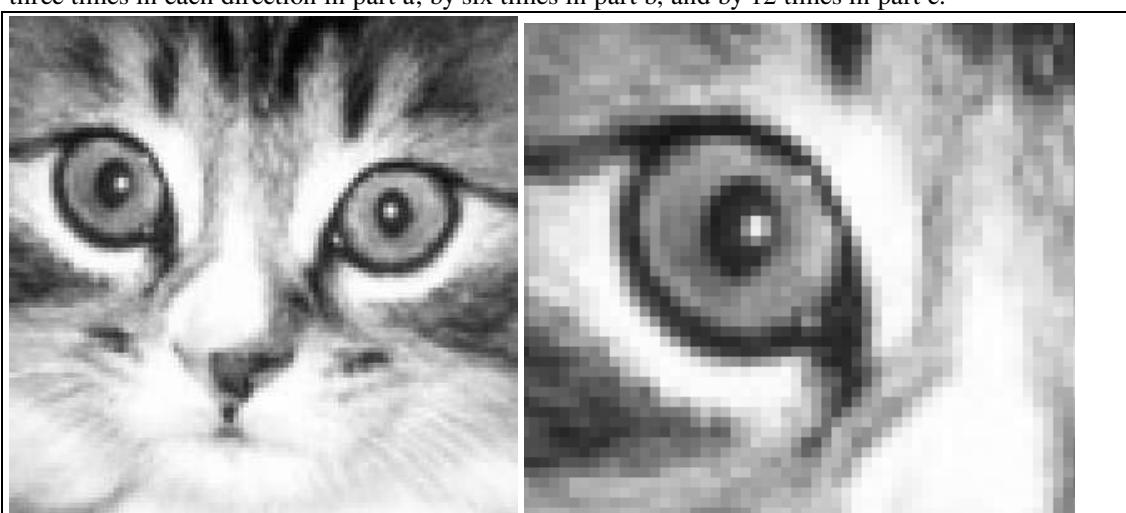


Figure 1.23. Three successive blow-ups of the kitten image. a). three times enlargement, b). six times enlargement.

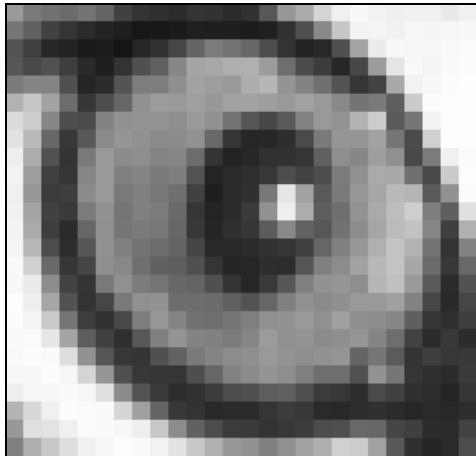


Figure 1.23c. Twelve times enlargement.

As another example, one often needs to “clean up” a scanned image, for instance to remove specks of noise or to reveal important details. Figure 1.24a shows the kitten image with gray levels altered to increase the contrast and make details more evident, and Figure 1.24b shows the effect of “edge enhancement”, achieved by a form of filtering the image.

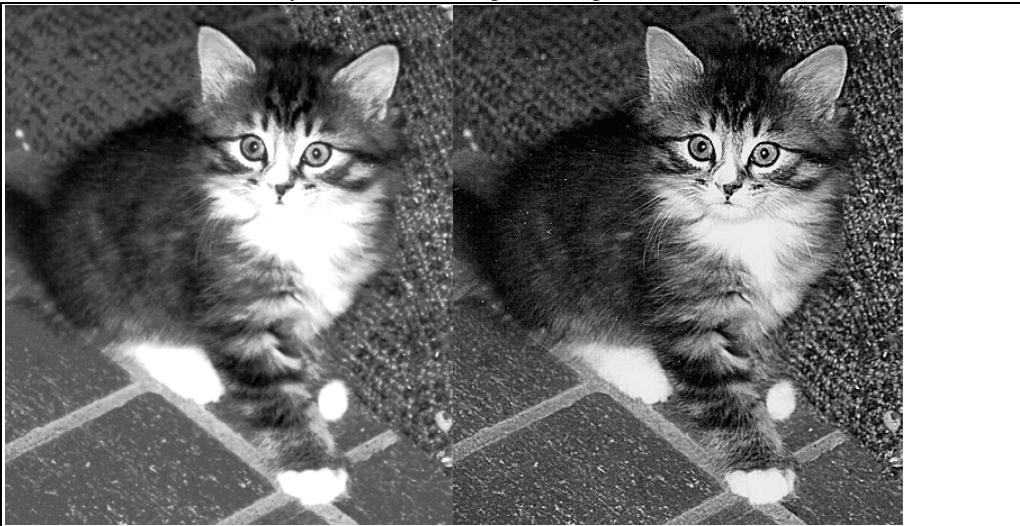


Figure 1.24. Examples of image enhancement

Figure 1.25 shows two examples of editing an image to accomplish some visual effect. Part a shows the kitten image “embossed”, and part b shows it distorted geometrically.



Figure 1.25. Examples of altering an image for visual effect.

1.3.5. Representation of gray shades and color for Raster Images

An important aspect of a raster image is the manner in which the various colors or shades of gray are represented in the bitmap. We briefly survey the most common methods here.

1.3.5.1. Gray-scale Raster Images.

If there are only two pixel values in a **raster image it is called bi-level**. Figure 1.26a shows a simple bi-level image, representing a familiar arrow-shaped cursor frequently seen on a computer screen. Its raster consists of 16 rows of 8 pixels each. Figure 1.26b shows the bitmap of this image as an array of 1's and 0's. The image shown at the left associates black with a 1 and white with a 0, but this association might just as easily be reversed. Since one bit of information is sufficient to distinguish two values, a **bilevel image is often referred to as a “1 bit per pixel” image**.

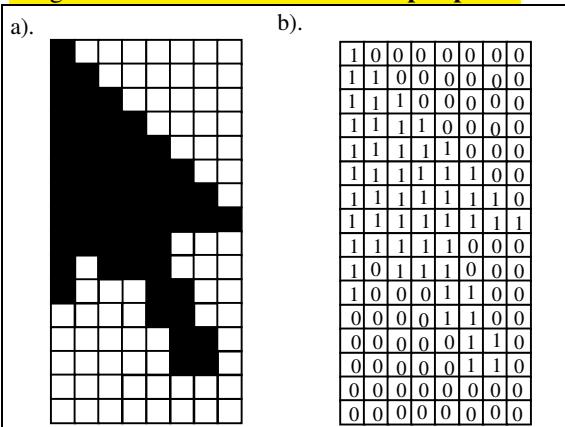


Figure 1.26. A bilevel image of a cursor, and its bitmap.

When the pixels in a gray-scale image take on more than two values, each pixel requires more than a single bit to represent it in memory. Gray-scale images are often classified in terms of their **pixel depth**, the number of bits needed to represent their gray levels. Since an n -bit quantity has 2^n possible values, there can be 2^n gray levels in an image with pixel depth n . The most common values are:

- 2 bits/pixel produce 4 gray levels
- 4 bits/pixel produce 16 gray levels
- 8 bits/pixel produce 256 gray levels

Figure 1.27 shows 16 gray levels ranging from black to white. Each of the sixteen possible pixel values is associated with a binary **4-tuple** such as 0110 or 1110. Here 0000 represents black, 1111 denotes white, and the other 14 values represent gray levels in between.

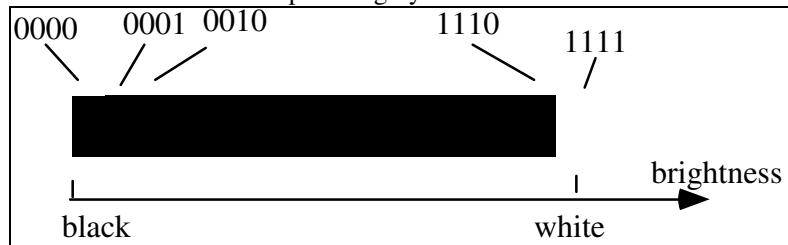


Figure 1.27. Sixteen levels of gray.

Many gray scale images² employ 256 gray levels, since this usually gives a scanned image acceptable quality. Each pixel is represented by some 8-bit values such as 01101110. The pixel value usually represents "brightness", where black is represented by 00000000, white by 11111111, and a medium gray by 10000000. Figure 1.23 seen earlier uses 256 gray levels.

Effect of Pixel depth: Gray-scale Quantization.

Sometimes an image that initially uses 8 bits per pixel is altered so that fewer bits per pixel are used. This might occur if a particular display device is incapable of displaying so many levels, or if the full image takes up too much memory. Figures 1.28 through 1.30 show the effect on the kitten image if pixel values are simply truncated to fewer bits. The loss in fidelity is hardly noticeable for the images in Figure 1.28, which use 6 and 5 bits/pixel (providing 64 and 32 different shades of gray, respectively).

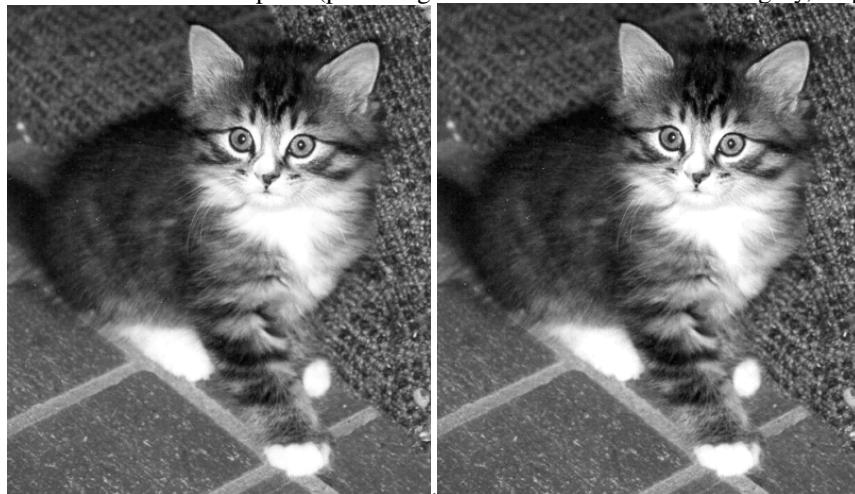


Figure 1.28. The image reduced to 6 bits/pixel and 5 bits/pixel.

But there is a significant loss in quality in the images of Figure 1.29. Part a shows the effect of truncating each pixel value to 4 bits, so there are only 16 possible shades of gray. For example, pixel value 01110100 is replaced with 0111. In part b the eight possible levels of gray are clearly visible. Note that some areas of the figure that show gradations of gray in the original now show a "lake" of uniform gray. This is often called **banding**, since areas that should show a gradual shift in the gray level instead show a sequence of uniform gray "bands".

² Thousands are available on the Internet, frequently as Gif, Jpeg, or Tiff images.

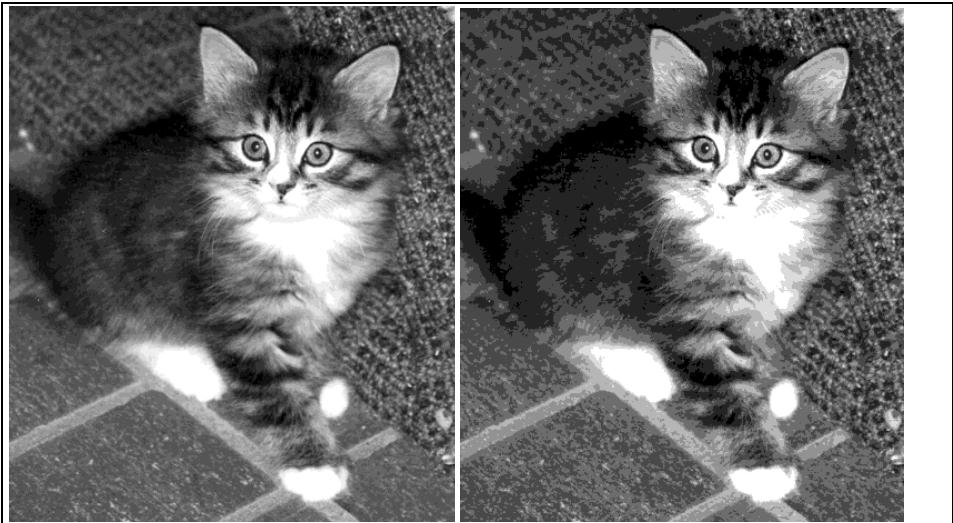


Figure 1.29. The image reduced to 4 bits/pixel and to 3 bits/pixel.

Figure 1.30 shows the cases of 2 and 1 bits/pixel. In part a the four levels are clearly visible and there is a great deal of banding. In part b there is only black and white and much of the original image information has been lost. In Chapter 10 we show techniques such as dithering for improving the quality of an image when too few bits are used for each pixel.



Figure 1.30. The image reduced to 2 bits/pixel and 1 bit/pixel.

1.3.5.2. Color Raster Images.

Color images are desirable because they match our daily experience more closely than do gray-scale images. Color raster images have become more common in recent years as the cost of high quality color displays has come down. The cost of scanners that digitize color photos has also become reasonable.

Each pixel in a color image has a “color value”, a numerical value that somehow represents a color. There are a number of ways to associate numbers and colors (see Chapter 12 for a detailed discussion), but one of the most common is to describe a color as a combination of amounts of red, green, and blue light. **Each pixel value is a 3-tuple**, such as (23, 14, 51), that prescribes the intensities of the red, green, and blue light components in that order.

The number of bits used to represent the color of each pixel is often called its **color depth**. Each value in the (red, green, blue) 3-tuple has a certain number of bits, and the color depth is the sum of these values. A color depth of 3 allows one bit for each component. For instance the pixel value (0, 1, 1) means that the red component is “off”, but both green and blue are “on”. In most displays the contributions from each component are added together (see Chapter 12 for exceptions such as in printing), so (0,1,1) would represent the addition of green and blue light, which is perceived as cyan. Since each component can be on or off there are eight possible colors, as tabulated in Figure 1.31. As expected, equal amounts of red, green, and blue, (1, 1, 1), produce white.

color value	displayed
0,0,0	black
0,0,1	blue
0,1,0	green
0,1,1	cyan
1,0,0	red
1,0,1	magenta
1,1,0	yellow
1,1,1	white

Figure 1.31. A common correspondence between color value and perceived color.

A color depth of 3 rarely offers enough precision for specifying the value of each component, so larger color depths are used. Because a byte is such a natural quantity to manipulate on a computer, many images have a color depth of eight. Each pixel then has one of 256 possible colors. A simple approach allows 3 bits for each of the red and the green components, and 2 bits for the blue component. But more commonly the association of each byte value to a particular color is more complicated, and uses a “color look-up” table, as discussed in the next section.

The highest quality images, known as **true color images**, have a color depth of 24, and so use a byte for each component. This seems to achieve as good color reproduction as the eye can perceive: more bits don’t improve an image. But such images require a great deal of memory: three bytes for every pixel. A high quality image of 1080 by 1024 pixels requires over three million bytes!

Plates 19 through 21 show some color raster images having different color depths. Plate 19 shows a full color image with a color depth of 24 bits. Plate 20 shows the degradation this image suffers when the color depth is reduced to 8 by simply truncating the red and green components to 3 bits each, and the blue component to 2 bits. Plate 21 also has a color depth of 8, so its pixels contain only 256 colors, but the 256 particular colors used have been carefully chosen for best reproduction. Methods to do this are discussed in Chapter 12.

author-supplied

Plate 19. Image with 24 bits/pixel.

author-supplied

Plate 20. Image with 3 bits for red and green pixels, and two bits for blue pixels.

author-supplied

Plate 21. Image with 256 carefully chosen colors.

1.4. Graphics Display Devices

We present an overview of some hardware devices that are used to display computer graphics. The devices include video monitors, plotters, and printers. A rich variety of graphics displays have been developed over the last thirty years, and new ones are appearing all the time. The quest is to display pictures of ever higher quality, that recreate more faithfully what is in the artist’s or engineer’s mind. In this section we look over the types of pictures that are being produced today, how they are being used, and the kinds of devices used to display them. In the process we look at ways to measure the “quality” of an image, and see how different kinds of display devices measure up.

1.4.1. Line Drawing Displays.

Some devices are naturally line-drawers. Because of the technology of the time, most early computer graphics were generated by line-drawing devices. The classic example is the **pen plotter**. A pen plotter moves a pen invisibly over a piece of paper to some spot that is specified by the computer, puts the pen down, and then sweeps the pen across to another spot, leaving a trail of ink of some color. Some plotters have a carousel that holds several pens which the program can exchange automatically in order to draw in different colors. Usually the choice of available colors is very limited: a separate pen is used for each color. The “quality” of a line-drawing is related to the precision with which the pen is positioned, and the sharpness of the lines drawn.

There are various kinds of pen plotters. **Flatbed plotters** move the pen in two dimensions over a stationary sheet of paper. **Drum plotters** move the paper back and forth on a drum to provide one direction of motion, while the pen moves back and forth at the top of the drum to provide the other direction.

There are also video displays called “vector”, “random-scan”, or “calligraphic” displays, that produce line-drawings. They have internal circuitry specially designed to sweep an electronic beam from point to point across the face of a cathode ray tube, leaving a glowing trail.

Figure 1.32 shows an example of a vector display, used by a flight controller to track the positions of many aircraft. Since each line segment to be displayed takes only a little data (two end points and perhaps a color), vector displays can draw a picture very rapidly (hundreds of thousands of vectors per second).

author-supplied

Figure 1.32. Example of a vector display (Courtesy Evans & Sutherland)

Vector displays, however, cannot show smoothly shaded regions or scanned images. Region filling is usually simulated by **cross hatching** with different line patterns, as suggested in Figure 1.33. Today raster displays have largely replaced vector displays except in very specialized applications.

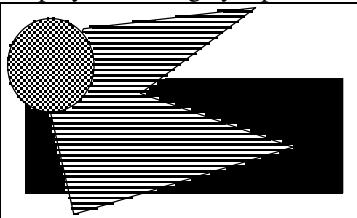


Figure 1.33. Cross-hatching to simulate filling a region.

1.4.2. Raster Displays.

Most displays used today for computer graphics are raster displays. The most familiar raster displays are the **video monitor connected to personal computers and workstations** (see Figure 1.34a), and the **flat panel** display common to portable personal computers (see Figure 1.34b). Other common examples produce **hard copy** of an image: the **laser printer**, **dot matrix printer**, **ink jet plotter**, and **film recorder**. We describe the most important of these below.

author-supplied

Figure 1.34. a). video monitors on PC, b). flat panel display.

Raster devices have a **display surface** on which the image is presented. The display surface has a certain number of pixels that it can show, such as 480 rows, where each row contains 640 pixels. So this display surface can show $480 \times 640 = 307,200$ pixels simultaneously. All such displays have a built-in coordinate system that associates a given pixel in an image with a given physical position on the display surface. Figure 1.35 shows an example. Here the horizontal coordinate *sx* increases from left to right, and the

vertical coordinate sy increases from top to bottom. This “upside-down” coordinate system is typical of raster devices.

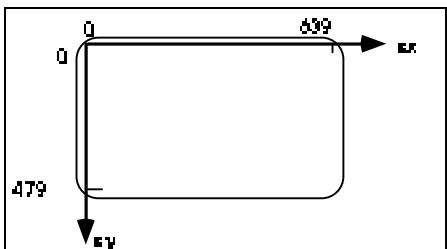


Figure 1.35. The built-in coordinate system for the surface of a raster display.

Raster displays are always connected one way or another to a **frame buffer**, a region of memory sufficiently large to hold all of the pixel values for the display (i.e. to hold the bitmap). The frame buffer may be physical memory on-board the display, or it may reside in the host computer. For example, a graphics card that is installed in a personal computer actually houses the memory required for the frame buffer.

Figure 1.36 suggests how an image is created and displayed. The **graphics program is stored in system memory** and is **executed** instruction by instruction **by the central processing unit (CPU)**. The program **computes** appropriate **values for each pixel** in the desired image and **loads them into the frame buffer**. (This is the part we focus on later when it comes to programming: building tools that write the “correct” pixel values into the frame buffer.) A “**scan controller**” takes care of the actual **display process**. It runs autonomously (rather than under program control), and does the same thing pixel after pixel. It **causes the frame buffer to “send” each pixel** through a converter to the **appropriate physical spot** on the display surface. The converter takes a pixel value such as 01001011 and converts it to the corresponding quantity that produces a spot of color on the display.

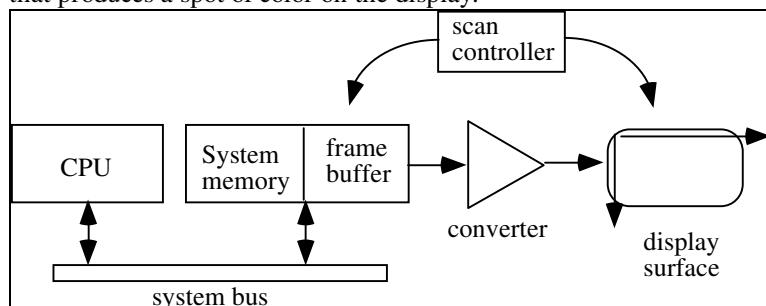


Figure 1.36. Block diagram of a computer with raster display.

The scanning process.

Figure 1.37 provides more detail on the scanning process. The main issue is how each pixel value in the frame buffer is “sent” to the right place on the display surface. Think of each of the pixels in the frame buffer as having a two-dimensional address (x, y). For address (136, 252), for instance, there is a specific memory location that holds the pixel value. Call it `mem[136][252]`.

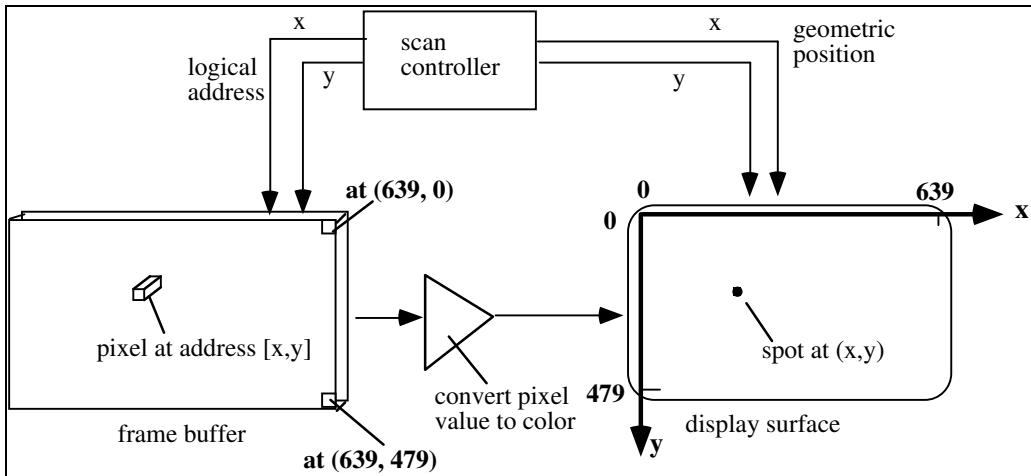


Figure 1.37. Scanning out an image from the frame buffer to the display surface.

The scan controller sends logical address (136, 252) to the frame buffer, which emits the value $\text{mem}[136][252]$. The controller also simultaneously “addresses” a physical (geometric) position (136, 252) on the display surface. Position (136, 252) corresponds to a certain physical distance of 136 units horizontally, and 252 units vertically, from the upper left hand corner of the display surface. Different raster displays use different units.

The value $\text{mem}[136][252]$ is converted to a corresponding intensity or color in the conversion circuit, and the intensity or color is sent to the proper physical position (136, 252) on the display surface.

To scan out the image in the entire frame buffer, every pixel value is visited once, and its corresponding spot on the display surface is “excited” with the proper intensity or color.

In some devices this scanning must be repeated many times per second, in order to "refresh" the picture. The video monitor to be described next is such a device.

With these generalities laid down, we look briefly at some specific raster devices, and see the different forms that arise.

- **Video Monitors.**

Video monitors are based on a **CRT**, or cathode-ray tube, similar to the display in a television set. Figure 1.38 adds some details to the general description above for a system using a video monitor as the display device. In particular, the conversion process from pixel value to “spot of light” is illustrated. The system shown has a color depth of 6 bits; the frame buffer is shown as having six bit “planes”. Each pixel uses one bit from each of the planes.

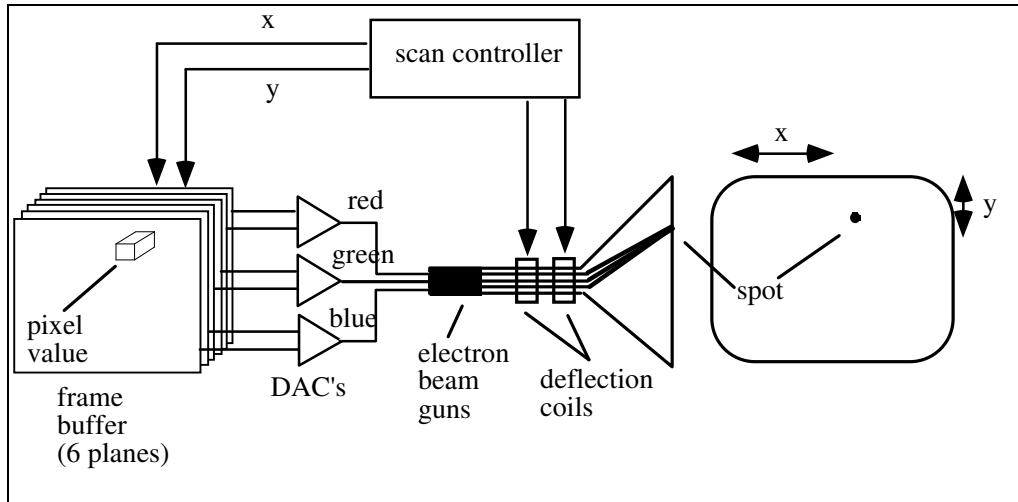


Figure 1.38. Operation of a color video monitor display system.

The red, green, and blue components of a pixel each use a pair of bits. These pairs are fed to three **digital-to-analog converters** (DAC's), which convert logical values like 01 into actual voltages. The correspondence between digital input values and output voltages is shown in Figure 1.39, where *Max* is the largest voltage level the DAC can produce.

input	voltage/brightness
00	0 * Max
01	0.333 * Max
10	0.666 * Max
11	1 * Max

Figure 1.39. Input-output characteristic of a two-bit DAC.

The three voltage levels drive three “guns” inside the CRT, which in turn excite three electron beams with intensities proportional to the voltages. The deflection coils divert the three beams so they stimulate three tiny phosphor dots at the proper place (*x*, *y*) on the inside of the cathode ray tube. Because of the phosphor materials used, one dot glows red when stimulated, one glows green, and one glows blue. The dots are so close together your eye sees one composite dot, and perceives a color that is the sum of the three component colors. Thus the composite dot can be made to glow in a total of $4 \times 4 \times 4 = 64$ different colors.

As described earlier the scan controller addresses one pixel value `mem[x][y]` in the frame buffer at the same time it “addresses” one position (*x*, *y*) on the face of the CRT by sending the proper signal to the deflection coils. Because the glow of a phosphor dot quickly fades when the stimulation is removed, a CRT image must be **refreshed** rapidly (typically 60 times a second) to prevent disturbing **flicker**. During each “refresh interval” the scan controller scans quickly through the entire frame buffer memory, sending each pixel value to its proper spot on the screen’s surface.

Scanning proceeds row by row through the frame buffer, each row providing pixel values for one **scanline** across the face of the CRT. The order of scanning is usually left to right along a **scanline** and from top to bottom by scanline. (Historians say this convention has given rise to terms like scanline, as well as the habit of numbering scanlines downward with 0 at the top, resulting in upside down coordinate systems.)

Some more expensive systems have a frame buffer that supports 24 planes of memory. Each of the DAC's has eight input bits, so there are 256 levels of red, 256 of green, and 256 of blue, for a total of $2^{24} = 16$ million colors.

At the other extreme, there are **monochrome** video displays, which display a single color in different intensities. A single DAC converts pixel values in the frame buffer to voltage levels, which drive a single electron beam gun. The CRT has only one type of phosphor so it can produce various intensities of only one color. Note that 6 planes of memory in the frame buffer gives $2^6 = 64$ levels of gray.

The color display of Figure 1.39 has a *fixed* association with a displayed color. For instance, the pixel value 001101 sends 00 to the “red DAC”, 11 to the “green DAC”, and 01 to the “blue DAC”, producing a mix of bright green and dark blue — a bluish-green. Similarly, 110011 is displayed as a bright magenta, and 000010 as a medium bright blue.

1.4.3. Indexed Color and the LUT.

Some systems are built using an alternative method of associating pixel values with colors. They use a **color lookup table** (or **LUT**), which offers a *programmable* association between pixel value and final color. Figure 1.40 shows a simple example. The color depth is again six, but the six bits stored in each pixel go through an intermediate step before they drive the CRT. They are used as an *index* into a table of 64 values, say $\text{LUT}[0] \dots \text{LUT}[63]$. (Why are there exactly 64 entries in this LUT?) For instance, if a pixel value is 39, the values stored in $\text{LUT}[39]$ are used to drive the DAC’s, as opposed to having the bits in the value 39 itself drive them. As shown $\text{LUT}[39]$ contains the 15 bit value 01010 11001 10010. Five of these bits (01010) are routed to drive the “red DAC”, five others drive the “green DAC”, and the last five drive the “blue DAC”.

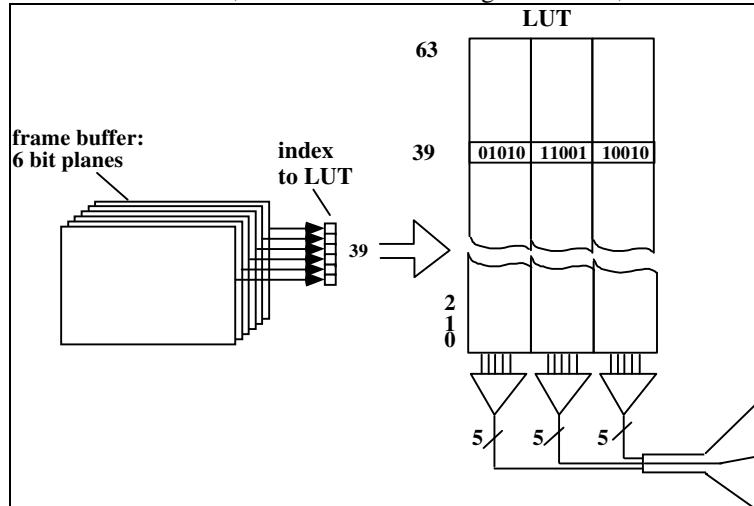


Figure 1.40. A color display system that incorporates a LUT.

Each of the $\text{LUT}[\]$ entries can be set under program control, using some system routine such as `setPalette()`. For example, the instruction: `setPalette(39, 17, 25, 4);`

would set the value in $\text{LUT}[39]$ to the fifteen bit quantity 10001 11001 00100 (since 17 is 10001 in binary, 25 is 11001, and 4 is 00100).

To make a particular pixel glow in this color, say the pixel at location $(x, y) = (479, 532)$, the value 39 is stored in the frame buffer, using `drawDot()` defined earlier:

```
drawDot(479, 532, 39); // set pixel at (479, 532) to value 39
```

Each time the frame buffer is “scanned out” to the display, this pixel is read as value 39, which causes the value stored in $\text{LUT}[39]$ to be sent to the DAC’s.

This programmability offers a great deal of flexibility in choosing colors, but of course it comes at a price: the program (or programmer) has to figure out which colors to use! We consider this further in Chapter 10.

What is the potential of this system for displaying colors? In the system of Figure 1.41 each entry of the LUT consists of 15 bits, so each color can be set to one of $2^{15} = 32K = 32,768$ possible colors. The set of 2^{15} possible colors displayable by the system is called its **palette**, so we say this display “has a palette of 32K colors”.

The problem is that each pixel value lies in the range 0..63, and only 64 different colors can be stored in the LUT at one time. Therefore this system can display a maximum of 64 different colors *at one time*. “At one time” here means during one scan-out of the entire frame buffer — something like 1/60-th of a second. The contents of the LUT are not changed in the middle of a scan-out of the image, so one whole scan-out uses a fixed set of 64 palette colors. Usually the LUT contents remain fixed for many scan-outs, although a program can change the contents of a small LUT during the brief dormant period between two successive scan-outs.

In more general terms, suppose that a raster display system has a color depth of b bits (so there are b bit planes in its frame buffer), and that each LUT entry is w bits wide. Then we have that:

The system can display 2^w colors, any 2^b at one time.

Examples.

- (1). A system with $b = 8$ bit planes and a LUT width $w = 12$ can display 4096 colors, any 256 of them at a time.
- (2). A system with $b = 8$ bitplanes and a LUT width $w = 24$ can display $2^{24} = 16,777,216$ colors, any 256 at a time.
- (3). If $b = 12$ and $w = 18$, the system can display $256k = 262,144$ colors, 4096 at a time.

There is no enforced relationship between the number of bit planes, b , and the width of the LUT, w . Normally w is a multiple of 3, so the same number of bits ($w/3$) drives each of the three DAC's. Also, b never exceeds w , so the palette is at least as large as the number of colors that can be displayed at one time. (Why would you never design a system with $w < b$?)

Note that the LUT itself requires very little memory, only 2^b words of w bits each. For example, if $b = 12$ and $w = 18$ there are only 9,216 bytes of storage in the LUT.

So what is the motivation for having a LUT in a raster display system? It is usually a need to reduce the cost of memory. Increasing b increases significantly the amount of memory needed for the frame buffer, mainly because there are so many pixels. The tremendous amount of memory needed can add significantly to the cost of the overall system.

To compare the costs of two systems, one with a LUT and one without, Figure 1.41 shows an example of two 1024 by 1280 pixel displays, (so each of them supports about 1.3 million pixels). Both systems allow colors to be defined with a precision of 24 bits, often called “true color”.

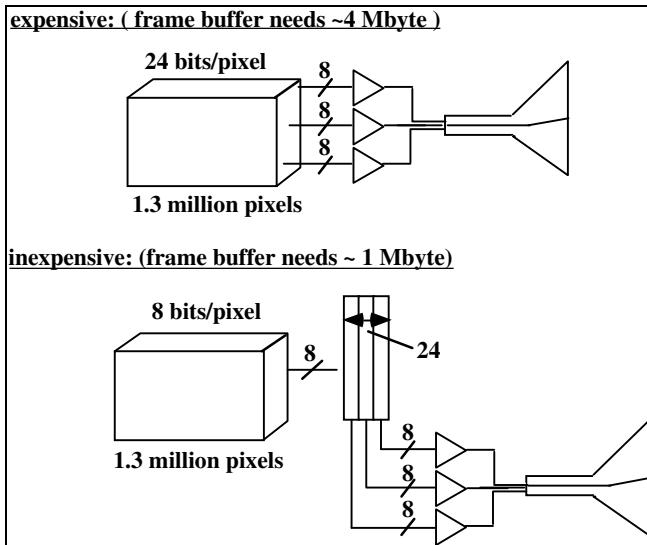


Figure 1.41. Comparison of two raster display systems.

System #1 (expensive): The first has a 24 bit/pixel frame buffer and no LUT, so each of its 1.3 million pixels can be set to any one of 2^{24} colors. Eight bits drive each of the DAC's. (The number '8' and the slash through the line into each DAC indicate the presence of eight bit lines feeding into the DAC.) The amount of memory required for the frame buffer here is $1024 \times 1280 \times 24$ bits which is almost 4 megabytes.

System #2 (inexpensive): The second has an 8 bit/pixel frame buffer along with a LUT, and the LUT is 24 bits wide. The system can display 2^{24} different colors, but only 256 at a time. The amount of memory required for the frame buffer here is $1024 \times 1280 \times 8$ which is about 1 megabyte. (The LUT requires a trivial 768 bytes of memory.) If memory costs a significant amount per megabyte, this system is much less expensive than system #1.

Putting a LUT in an inexpensive system attempts to compensate for the small number of different pixel values possible. The LUT allows the programmer to create a full set of colors, even though a given image can contain only a restricted set of them.

Displays with LUT's are still quite common today because memory costs are still high. The situation is changing rapidly, however, as memory prices plummet. Many reasonably priced personal computers today have 24 bit frame buffers.

Practice Exercises.

1.4.1. Why not always have a LUT? Since a LUT is inexpensive, and offers the advantage of flexibility, why not have a LUT even in a system with 24 bits per pixel?

1.4.2. Configure your own system. For each of the systems described below:

- draw the circuit diagram, similar to Figure 1.41;
- Label the number of bits associated with the frame buffer, the DAC's, and the LUT (if present);
- Calculate (in bytes) the amount of storage required for the frame buffer and the LUT (if present):
 - $b = 15$, no LUT;
 - $b = 15$, $w = 24$;
 - $b = 8$, $w = 18$;
 - $b = 12$, no LUT.

1.5. Graphics Input Primitives and Devices.

Many input devices are available that let the user control a computer. Whereas typing a command might be awkward, it is natural to “point” to a particular object displayed on the screen to make a choice for the next action.

You can look at an input device in two ways: what it *is*, and what it *does*. Each device is physically some piece of machinery like a mouse, keyboard, or trackball. It fits in the hand in a certain way and is natural for the user to manipulate. It measures these manipulations and sends corresponding numerical information back to the graphics program.

We first look at what input devices do, by examining the kinds of data each sends to the program. We then look at a number of input devices in common use today.

1.5.1. Types of Input Graphics Primitives.

Each device transmits a particular kind of data (e.g. a number, a string of characters, or a position) to the program. The different types of data are called **input primitives**. Two different physical devices may transmit the same type of data, so logically they generate the same graphics primitives.

The important input primitives are:

String. The **string** “device” is the most familiar, producing a **string of characters** and thus modeling the action of a keyboard. When an application requests a string, the program pauses while the user types it in followed by a termination character. The program then resumes with the string stored in memory.

Choice. A **choice** device reports a **selection** from a fixed number of choices. The programmer's model is a bank of buttons, or a set of buttons on a mouse.

Valuator. A **valuator** produces a real value between 0.0 and 1.0, which can be used to fix the length of a line, the speed of an action, or perhaps the size of a picture. The model in the programmer's mind is a knob that can be turned from 0 to 1 in smooth gradations.

Locator. A basic requirement in interactive graphics is to allow the user to point to a position on the display. The **locator** input device performs this function, because it produces a **coordinate pair** (x, y). The user manipulates an input device (usually a mouse) in order to position a visible cursor to some spot and then triggers the choice. This returns to the application the values of x and y , along with the trigger value.

Pick. The **pick** input device is used to identify a portion of a picture for further processing. Some graphics packages allow a picture to be defined in terms of **segments**, which are groups of related graphics. The package provides tools to define segments and to give them identifying names. When using `pick()`, the user “points” to a part of a picture with some physical input device, and the package figures out which segment is being pointed to. `pick()` returns the name of the segment to the application, enabling the user to erase, move, or otherwise manipulate the segment.

The graphics workstation is initialized when an application starts running: among other things each logical input function is associated with one of the installed physical devices.

1.5.2. Types of Physical Input Devices.

We look at the other side of input devices: the physical machine that is connected to the personal computer or workstation.

Keyboard. All workstations are equipped with a keyboard, which sends strings of characters to the application upon request. Hence a keyboard is usually used to obtain a **string** device. Some keyboards have cursor keys or function keys, which are often used to produce **choice** input primitives.

Buttons. Sometimes a separate bank of buttons is installed on a workstation. The user presses one of the buttons to perform a choice input function.

Mouse. The **mouse** is perhaps the most familiar input device of all, as it is easy and comfortable to operate. As the user slides the mouse over the desktop, the mouse sends the changes in its position to the workstation. Software within the workstation keeps track of the mouse's position and moves a **graphics cursor** — a small dot or cross — on the screen accordingly. The mouse is most often used to perform a `locate` or a `pick` function. There are usually some buttons on the mouse that the user can press to trigger the action.

Tablet. Like a mouse, a tablet is used to generate `locate` or `pick` input primitives. A **tablet** provides an area on which the user can slide a stylus. The tip of the stylus contains a microswitch. By pressing down on the stylus the user can trigger the logical function.

The tablet is particularly handy for digitizing drawings: the user can tape a picture onto the tablet surface and then move the stylus over it, pressing down to send each new point to the workstation. A menu area is sometimes printed on the tablet surface, and the user *Pick*'s a menu item by pressing down the stylus inside one of the menu item boxes. Suitable software associates each menu item box with the desired function for the application that is running.

Space Ball and Data Glove. The Space Ball and Data Glove are relatively new input devices. Both are designed to give a user explicit control over several variables at once, by performing hand and finger motions. Sensors inside each device pick up subtle hand motions and translate them into *Valuator* values that get passed back to the application. They are particularly suited to situations where the hand movements themselves make sense in the context of the program, such as when the user is controlling a virtual robot hand, and watching the effects of such motions simulated on the screen.

(Section 1.6 Summary - deleted.)

1.7. For Further Reading.

A number of books provide a good introduction to the field of computer graphics. Hearn and Baker [hearn94] gives a leisurely and interesting overview of the field with lots of examples. Foley and Van Dam [foley93] and David Rogers[rogers98] give additional technical detail on the many kinds of graphics input and output devices. An excellent series of five books known as “Graphics Gems” [gems], first published in 1990, brought together many new ideas and “gems” from graphics researchers and practitioners around the world.

There are also a number of journals and magazines that give good insight into new techniques in computer graphics. The most accessible is the IEEE Computer Graphics and Applications, which often features survey articles on new areas of effort with graphics. The classic repositories of new results in graphics are the annual Proceedings of SIGGRAPH [SIGGRAPH], and the ACM Transactions on Graphics [TOGS]. Another more recent arrival is the Journal of Graphics Tools [jgt].

CHAP 2. Getting Started: Drawing Figures

Machines exist; let us then exploit them to create beauty , a modern beauty, while we are about it. For we live in the twentieth century.

Aldous Huxley

Goals of the Chapter

- To get started writing programs that produce pictures.
- To learn the basic ingredients found in every OpenGL program
- To develop some elementary graphics tools for drawing lines, polylines, and polygons.
- To develop tools that allow the user to control a program with the mouse and keyboard.

Preview

Section 2.1 discusses the basics of writing a program that makes simple drawings. The importance of device-independent programming is discussed, and the characteristics of windows-based and event-driven programs are described. Section 2.2 introduces the use of OpenGL as the device-independent application programmer interface (API) that is emphasized throughout the book and shows how to draw various graphics primitives. Sample drawings, such as a picture of the Big Dipper, a drawing of the Sierpinski gasket, and a plot of a mathematical function illustrate the use of OpenGL. Section 2.3 discusses how to make pictures based on polylines and polygons, and begins the building of a personal library of graphics utilities. Section 2.4 describes interactive graphics programming, whereby the user can indicate positions on the screen with the mouse or press keys on the keyboard to control the action of a program. The chapter ends with a number of case studies which embellish ideas discussed earlier and that delve deeper into the main ideas of the chapter.

2.1. Getting started making pictures.

Like many disciplines, computer graphics is mastered most quickly by doing it: by writing and testing programs that produce a variety of pictures. It is best to start with simple tasks. Once these are mastered you can try variations, see what happens, and move towards drawing more complex scenes.

To get started you need an environment that lets you write and execute programs. For graphics this environment must also include hardware to display pictures (usually a CRT display which we shall call the "screen"), and a library of software tools that your programs can use to perform the actual drawing of graphics primitives.

Every graphics program begins with some initializations; these establish the desired display mode, and set up a coordinate system for specifying points, lines, etc. Figure 2.1 shows some of the different variations one might encounter. In part a) the entire screen is used for drawing: the display is initialized by switching it into "graphics mode", and the coordinate system is established as shown. Coordinates x and y are measured in pixels, with x increasing to the right and y increasing downward.

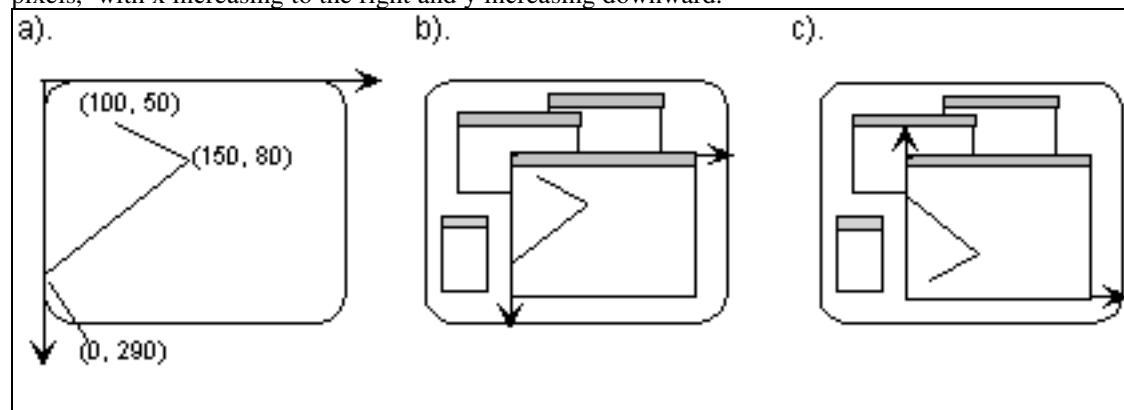


Figure 2.1. Some common varieties of display layouts.

In part b) a more modern "window-based" system is shown. It can support a number of different rectangular *windows* on the display screen at one time. Initialization involves creating and "opening" a new window (which we shall call the **screen window**¹) for graphics. Graphics commands use a coordinate system that is attached to the window: usually *x* increases to the right and *y* increases downward². Part c) shows a variation where the initial coordinate system is "right side up", with *y* increasing upward³.

Each system normally has some elementary drawing tools that help to get started. The most basic has a name like `setPixel(x, y, color)`: it sets the individual pixel at location (*x*, *y*) to the color specified by `color`. It sometimes goes by different names, such as `putPixel()`, `SetPixel()`, or `drawPoint()`. Along with `setPixel()` there is almost always a tool to draw a straight line, `line(x1, y1, x2, y2)`, that draws a line between (*x*₁, *y*₁) and (*x*₂, *y*₂). In other systems it might be called `drawLine()` or `Line()`. The commands

```
line(100, 50, 150, 80);
line(150, 80, 0, 290);
```

would draw the pictures shown in each system in Figure 2.1. Other systems have no `line()` command, but rather use `moveto(x, y)` and `lineto(x, y)`. They stem from the analogy of a pen plotter, where the pen has some **current position**. The notion is that `moveto(x, y)` moves the pen invisibly to location (*x*, *y*), thereby setting the current position to (*x*, *y*); `lineto(x, y)` draws a line from the current position to (*x*, *y*), then updates the current position to this (*x*, *y*). Each command moves the pen from its current position to a new position. The new position then becomes the current position. The pictures in Figure 2.1 would be drawn using the commands

```
moveto(100, 50);
lineto(150, 80);
lineto(0, 290);
```

For a particular system the energetic programmer can develop a whole toolkit of sophisticated functions that utilize these elementary tools, thereby building up a powerful library of graphics routines. The final graphics applications are then written making use of this personal library.

An obvious problem is that each graphics display uses different basic commands to "drive it", and every environment has a different collection of tools for producing the graphics primitives. This makes it difficult to *port* a program from one environment to another (and sooner or later everyone is faced with reconstructing a program in a new environment): the programmer must build the necessary tools on top of the new environment's library. This may require major alterations in the overall structure of a library or application, and significant programmer effort.

2.1.1. Device Independent Programming, and OpenGL.

It is a boon when a uniform approach to writing graphics applications is made available, such that the *same* program can be compiled and run on a variety of graphics environments, with the guarantee that it will produce nearly identical graphical output on each display. This is known as **device independent** graphics programming. OpenGL offers such a tool. Porting a graphics program only requires that you install the appropriate OpenGL libraries on the new machine; the application itself requires no change: it calls the same functions in this library with the same parameters, and the same graphical results are produced. The OpenGL way of creating graphics has been adopted by a large number of industrial companies, and OpenGL libraries exist for all of the important graphics environments⁴.

OpenGL is often called an "application programming interface" (API): the interface is a collection of routines that the programmer can call, along with a model of how the routines work together to produce graphics. The programmer "sees" only the interface, and is therefore shielded from having to cope with the specific hardware or software idiosyncrasies on the resident graphics system.

¹ The word "window" is overused in graphics: we shall take care to distinguish the various instances of the term.

² Example systems are unix workstations using X Windows, an IBM pc running Windows 95 using the basic Windows Application Programming Interface, and an Apple Macintosh using the built-in QuickDraw library.

³ An example is any window-based system using OpenGL.

⁴ Appendix 1 discusses how to obtain and get started with OpenGL in different environments.

OpenGL is at its most powerful when drawing images of complex three dimensional (3D) scenes, as we shall see. It might be viewed as overkill for simple drawings of 2D objects. But it works well for 2D drawing, too, and affords a *unified* approach to producing pictures. We start by using the simpler constructs in OpenGL, capitalizing for simplicity on the many default states it provides. Later when we write programs to produce elaborate 3D graphics we tap into OpenGL's more powerful features.

Although we will develop most of our graphics tools using the power of OpenGL, we will also “look under the hood” and examine how the classical graphics algorithms work. It is important to see how such tools might be implemented, even if for most applications you use the ready-made OpenGL versions. In special circumstances you may wish to use an alternative algorithm for some task, or you may encounter a new problem that OpenGL does not solve. You also may need to develop a graphics application that does not use OpenGL at all.

2.1.2. Windows-based programming.

As described above, many modern graphics systems are *windows-based*, and manage the display of multiple overlapping *windows*. The user can move the windows around the screen using the mouse, and can resize them. Using OpenGL we will do our drawing in one of these windows, as we saw in Figure 2.1c.

Event-driven programming.

Another property of most windows-based programs is that they are *event-driven*. This means that the program responds to various events, such as a mouse click, the press of a keyboard key, or the resizing of a screen window. The system automatically manages an *event queue*, which receives messages that certain events have occurred, and deals with them on a *first-come first-served basis*. The programmer organizes a program as a collection of *callback functions* that are executed when events occur. A callback function is created for each type of event that might occur. When the system removes an event from the queue it simply executes the *callback function* associated with the type of that event. For programmers used to building programs with a “do this, then do this,...” structure some rethinking is required. The new structure is more like: “do nothing until an event occurs, then do the specified thing”.

The method of associating a callback function with an event type is often quite system dependent. But OpenGL comes with a *Utility Toolkit* (see Appendix 1), which provides tools to assist with event management. For instance

```
glutMouseFunc(myMouse); // register the mouse action function
```

registers the function myMouse() as the function to be executed when a mouse event occurs. The prefix “glut” indicates it is part of the OpenGL Utility Toolkit. The programmer puts code in myMouse() to handle all of the possible mouse actions of interest.

Figure 2.2 shows a skeleton of an example main() function for an event-driven program. We will base most of our programs in this book on this skeleton. There are four principle types of events we will work with, and a “glut” function is available for each:

```
void main()
{
    initialize things5
    create a screen window
    glutDisplayFunc(myDisplay); // register the redraw function
    glutReshapeFunc(myReshape); // register the reshape function
    glutMouseFunc(myMouse); // register the mouse action function
    glutKeyboardFunc(myKeyboard); // register the keyboard action function
    perhaps initialize other things
    glutMainLoop(); // enter the unending main loop
}
```

all of the callback functions are defined here

Figure 2.2. A skeleton of an event-driven program using OpenGL.

⁵ Notes shown in italics in code fragments are pseudocode rather than actual program code. They suggest the actions that real code substituted there should accomplish.

- `glutDisplayFunc(myDisplay);` Whenever the system determines that a screen window should be **redrawn it issues a “redraw” event.** This happens when the window is first opened, and when the window is exposed by moving another window off of it. Here the function `myDisplay()` is registered as the callback function for a redraw event.
- `glutReshapeFunc(myReshape);` **Screen windows can be reshaped by the user,** usually by dragging a corner of the window to a new position with the mouse. (Simply moving the window does not produce a reshape event.) Here **the function `myReshape()` is registered with the “reshape” event.** As we shall see, `myReshape()` is automatically passed arguments that report the new width and height of the reshaped window.
- `glutMouseFunc(myMouse);` **When one of the mouse buttons is pressed or released a mouse event is issued.** Here `myMouse()` is registered as the function to be called when a mouse event occurs. `myMouse()` is automatically passed arguments that describe the mouse location and the nature of the button action.
- `glutKeyboardFunc(myKeyboard);` **This registers the function `myKeyboard()` with the event of pressing or releasing some key on the keyboard.** `myKeyboard()` is automatically passed arguments that tell which key was pressed. Conveniently, it is also passed data as to the location of the mouse at the time the key was pressed.

If a particular program does not use mouse interaction, the corresponding callback function need not be registered or written. Then mouse clicks have no effect in the program. The same is true for programs that have no keyboard interaction.

The final function shown in Figure 2.2 is `glutMainLoop()`. When this is executed the program draws the initial picture and enters an unending loop, in which it simply waits for events to occur. (A program is normally terminated by clicking in the “go away” box that is attached to each window.)

2.1.3. Opening a Window for Drawing.

The first task is to open a screen window for drawing. This can be quite involved, and is system dependent. Because OpenGL functions are device independent, they provide no support for window control on specific systems. But the OpenGL Utility Toolkit introduced above *does* include functions to open a window on whatever system you are using.

Figure 2.3 fleshes out the skeleton above to show the entire `main()` function for a program that will draw graphics in a screen window. The first five function calls use the toolkit to open a window for drawing with OpenGL. In your first graphics programs you can just copy these as is: later we will see what the various arguments mean and how to substitute others for them to achieve certain effects. The first five functions initialize and display the screen window in which our program will produce graphics. We give a brief description of what each one does.

```
// appropriate #includes go here - see Appendix 1

void main(int argc, char** argv)
{
    glutInit(&argc, argv); // initialize the toolkit
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB); // set the display mode
    glutInitWindowSize(640, 480); // set window size
    glutInitWindowPosition(100, 150); // set the window position on screen
    glutCreateWindow("my first attempt"); // open the screen window

    // register the callback functions
    glutDisplayFunc(myDisplay);
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);

    myInit(); // additional initializations as necessary
    glutMainLoop(); // go into a perpetual loop
}
```

Figure 2.3. Code using the OpenGL utility toolkit to open the initial window for drawing.

- `glutInit(&argc, argv);` This function initializes the toolkit. Its arguments are the standard ones for passing command line information; we will make no use of them here.
- `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);` This function specifies how the display should be initialized. The built-in constants `GLUT_SINGLE` and `GLUT_RGB`, which are OR'd together, indicate that a single display buffer should be allocated and that colors are specified using desired amounts of red, green, and blue. (Later we will alter these arguments: for example, we will use double buffering for smooth animation.)
- `glutInitWindowSize(640, 480);` This function specifies that the screen window should initially be 640 pixels wide by 480 pixels high. When the program is running the user can resize this window as desired.
- `glutInitWindowPosition(100, 150);` This function specifies that the window's upper left corner should be positioned on the screen 100 pixels from the left edge and 150 pixels down from the top. When the program is running the user can move this window wherever desired.
- `glutCreateWindow("my first attempt");` This function actually opens and displays the screen window, putting the title "my first attempt" in the title bar.

The remaining functions in `main()` register the callback functions as described earlier, perform any initializations specific to the program at hand, and start the main event loop processing. The programmer (you) must implement each of the callback functions as well as `myInit()`.

2.2. Drawing Basic Graphics Primitives.

We want to develop programming techniques for drawing a large number of geometric shapes that make up interesting pictures. The drawing commands will be placed in the callback function associated with a redraw event, such as the `myDisplay()` function mentioned above.

We first must establish the coordinate system in which we will describe graphical objects, and prescribe where they will appear in the screen window. Computer graphics programming, it seems, involves an ongoing struggle with defining and managing different coordinate systems. So we start simply and work up to more complex approaches.

We begin with an intuitive coordinate system. It is tied directly to the coordinate system of the screen window (see Figure 2.1c), and measures distances in pixels. Our first example screen window, shown in Figure 2.4, is 640 pixels wide by 480 pixels high. The *x*-coordinate increases from 0 at the left edge to 639 at the right edge. The *y*-coordinate increases from 0 at the bottom edge to 479 at the top edge. We establish this coordinate system later, after examining some basic primitives.

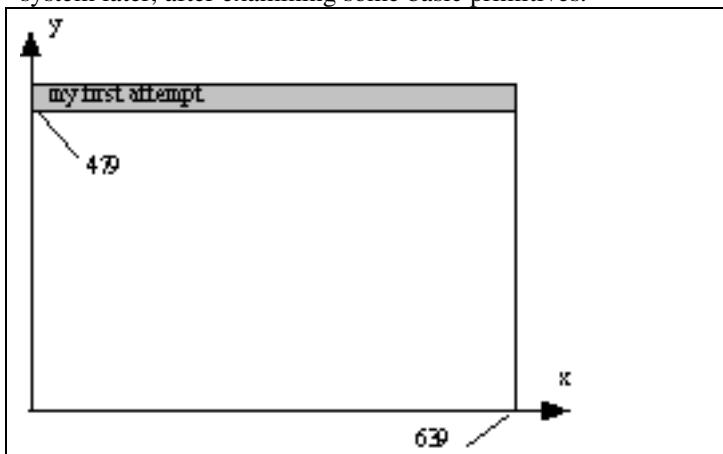


Figure 2.4. The initial coordinate system for drawing.

OpenGL provides tools for drawing all of the output primitives described in Chapter 1. Most of them, such as points, lines, polylines, and polygons, are defined by one or more *vertices*. To draw such objects in OpenGL you pass it a list of vertices. The list occurs between the two OpenGL function calls `glBegin()` and `glEnd()`. The argument of `glBegin()` determines which object is drawn. For instance, Figure 2.5 shows three points drawn in a window 640 pixels wide and 480 pixels high. These dots are drawn using the command sequence:

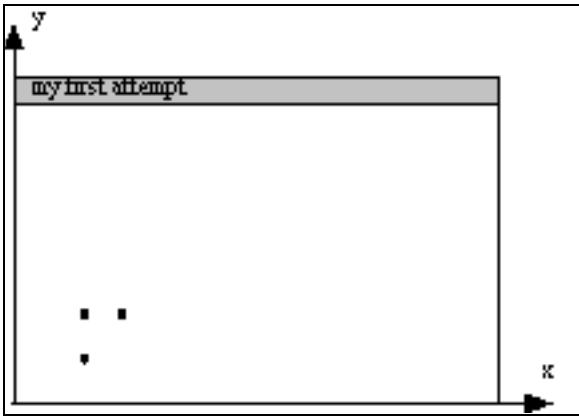


Figure 2.5. Drawing three dots.

```
glBegin(GL_POINTS);
    glVertex2i(100, 50);
    glVertex2i(100, 130);
    glVertex2i(150, 130);
glEnd();
```

The constant `GL_POINTS` is built-into OpenGL. To draw other primitives you replace `GL_POINTS` with `GL_LINES`, `GL_POLYGON`, etc. Each of these will be introduced in turn.

As we shall see later, these commands send the vertex information down a “graphics pipeline”, in which they go through several processing steps (look ahead to Figure ????). For present purposes just think of them as being sent more or less directly to the coordinate system in the screen window.

Many functions in OpenGL like `glVertex2i()` have a number of variations. The variations distinguish the number and type of arguments passed to the function. Figure 2.6 shows how such function calls are formatted.

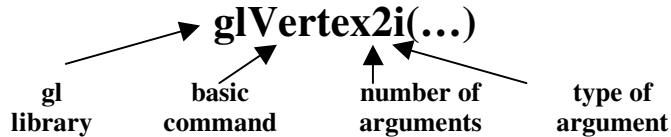


Figure 2.6. Format of OpenGL commands.

The prefix “`gl`” indicates a function from the OpenGL library (as opposed to “`glut`” for the utility toolkit). It is followed by the basic command root, then the number of arguments being sent to the function (this number will often be 3 and 4 in later contexts), and finally the type of argument, (`i` for an integer, `f` for a floating point value, etc., as we describe below). When we wish to refer to the basic command without regard to the specifics of its arguments we will use an asterisk, as in `glVertex*()`.

To generate the same three dot picture above, for example, you could pass it floating point values instead of integers, using:

```
glBegin(GL_POINTS);
    glVertex2d(100.0, 50.0);
    glVertex2d(100.0, 130.0);
    glVertex2d(150.0, 130.0);
glEnd();
```

On OpenGL Data Types.

OpenGL works internally with specific data types: for instance, functions such as `glVertex2i()` expect integers of a certain size (32 bits). It is well known that some systems treat the C or C++ data type `int` as a 16 bit quantity, whereas others treat it as a 32 bit quantity. There is no standard size for a `float` or `double` either. To insure that OpenGL functions receive the proper data types it is wise to use the built-in type names

like `GLint` or `GLfloat` for OpenGL types. The OpenGL types are listed in Figure 2.7. Some of these types will not be encountered until later in the book.

suffix	data type	typical C or C++ type	OpenGL type name
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating point	float	GLfloat, GLclampf
d	64-bit floating point	double	GLdouble, GLclampd
ub	8-bit unsigned number	unsigned char	GLubyte, GLboolean
us	16-bit unsigned number	unsigned short	GLushort
ui	32-bit unsigned number	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

Figure 2.7. Command suffixes and argument data types.

As an example, a function using suffix `i` expects a 32-bit integer, but your system might translate `int` as a 16-bit integer. Therefore if you wished to encapsulate the OpenGL commands for drawing a dot in a generic function such as `drawDot()` you might be tempted to use:

```
void drawDot(int x, int y)           ← danger: passes int's
{   // draw dot at integer point (x, y)
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}
```

which passes `int`'s to `glVertex2i()`. This will work on systems that use 32-bit `int`'s, but might cause trouble on those that use 16-bit `int`'s. It is much safer to write `drawDot()` as in Figure 2.8, and to use `GLint`'s in your programs. When you recompile your programs on a new system `GLint`, `GLfloat`, etc. will be associated with the appropriate C++ types (in the OpenGL header `GL.h` – see Appendix 1) for that system, and these types will be used consistently throughout the program.

```
void drawDot(GLint x, GLint y)
{   // draw dot at integer point (x, y)
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}
```

Figure 2.8. Encapsulating OpenGL details in a generic function `drawDot()`⁶.

The OpenGL “State”.

OpenGL keeps track of many *state variables*, such as the current “size” of points, the current color of drawing, the current background color, etc. The value of a state variable remains active until a new value is given. The size of a point can be set with `glPointSize()`, which takes one floating point argument. If its argument is 3.0 the point is usually drawn as a square three pixels on a side. For additional details on this and other OpenGL functions consult appropriate OpenGL documentation (some of which is on-line; see Appendix 1). The drawing color can be specified using

```
glColor3f(red, green, blue);
```

where the values of red, green, and blue vary between 0.0 and 1.0. For example, some of the colors listed in Figure 1.3.24??? could be set using:

```
glColor3f(1.0, 0.0, 0.0);      // set drawing color to red
glColor3f(0.0, 0.0, 0.0);      // set drawing color to black
glColor3f(1.0, 1.0, 1.0);      // set drawing color to white
glColor3f(1.0, 1.0, 0.0);      // set drawing color to yellow
```

⁶ Using this function instead of the specific OpenGL commands makes a program more readable. It is not unusual to build up a personal collection of such utilities.

The background color is set with `glClearColor(red, green, blue, alpha)`, where `alpha` specifies a degree of transparency and is discussed later (use 0.0 for now.) To clear the entire window to the background color, use `glClear(GL_COLOR_BUFFER_BIT)`. The argument `GL_COLOR_BUFFER_BIT` is another constant built into OpenGL.

Establishing the Coordinate System.

Our method for establishing our initial choice of coordinate system will seem obscure here, but will become clearer in the next chapter when we discuss windows, viewports, and clipping. Here we just take the few required commands on faith. The `myInit()` function in Figure 2.9 is a good place to set up the coordinate system. As we shall see later, OpenGL routinely performs a large number of transformations. It uses matrices to do this, and the commands in `myInit()` manipulate certain matrices to accomplish the desired goal. The `glOrtho2D()` routine sets the transformation we need for a screen window that is 640 pixels wide by 480 pixels high.

```
void myInit(void)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 640.0, 0, 480.0);
}
```

Figure 2.9. Establishing a simple coordinate system.

Putting it together: A Complete OpenGL program.

Figure 2.10 shows a complete program that draws the lowly three dots of Figure 2.5. It is easily extended to draw more interesting objects as we shall see. The initialization in `myInit()` sets up the coordinate system, the point size, the background color, and the drawing color. The drawing is encapsulated in the callback function `myDisplay()`. As this program is non-interactive, no other callback functions are used. `glFlush()` is called after the dots are drawn to insure that all data is completely processed and sent to the display. This is important in some systems that operate over a network: data is buffered on the host machine and only sent to the remote display when the buffer becomes full or a `glFlush()` is executed.

```

    glutDisplayFunc(myDisplay);      // register redraw function
    myInit();                      // initialization
    glutMainLoop();                // go into a perpetual loop
}

```

Figure 2.10. A complete OpenGL program to draw three dots.

2.2.1. Drawing Dot Constellations.

A “dot constellation” is some pattern of dots or points. We describe several examples of interesting dot constellations that are easily produced using the basic program in Figure 2.10. In each case the appropriate function is named in `glutDisplayFunc()` as the callback function for the redraw event. You are strongly encouraged to implement and test each example, in order to build up experience.

Example 2.2.1. The Big Dipper.

Figure 2.11 shows a pattern of eight dots representing the Big Dipper, a familiar sight in the night sky.

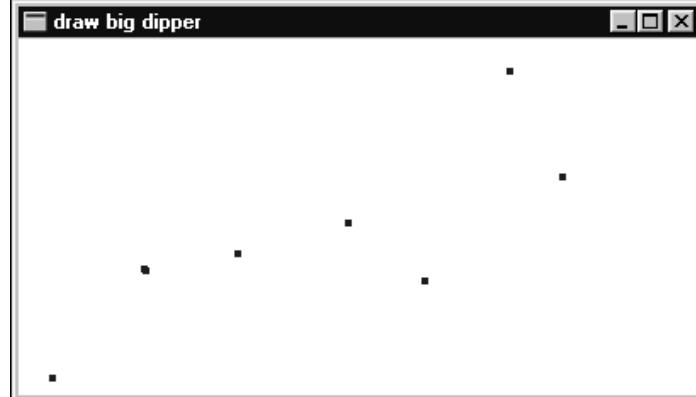


Figure 2.11. Two simple Dot Constellations.

The names and “positions” of the eight stars in the Big Dipper (for one particular view of the night sky), are given by: {Dubhe, 289, 190}, {Merak, 320, 128}, {Phecda, 239, 67}, {Megrez, 194, 101}, {Alioth, 129, 83}, {Mizar, 75, 73}, {Alcor, 74, 74}, {Alkaid, 20, 10}. Since so few data points are involved it is easy to list them explicitly, or “**hard-wire**” them into the code. (When many dots are to be drawn, it is more convenient to store them in a file, and then have the program read them from the file and draw them. We do this in a later chapter.) These points can replace the three points specified in Figure 2.10. It is useful to experiment with this constellation, trying different point sizes, as well as different background and drawing colors.

Example 2.2.2. Drawing the Sierpinski Gasket.

Figure 2.12 shows the Sierpinski gasket. Its dot constellation is generated *procedurally*, which means that each successive dot is determined by a procedural rule. Although the rule here is very simple, the final pattern is a fractal (see Chapter 8)! We first approach the rules for generating the Sierpinski gasket in an intuitive fashion. In Case Study 2.2 we see that it is one example of an *iterated function system*.

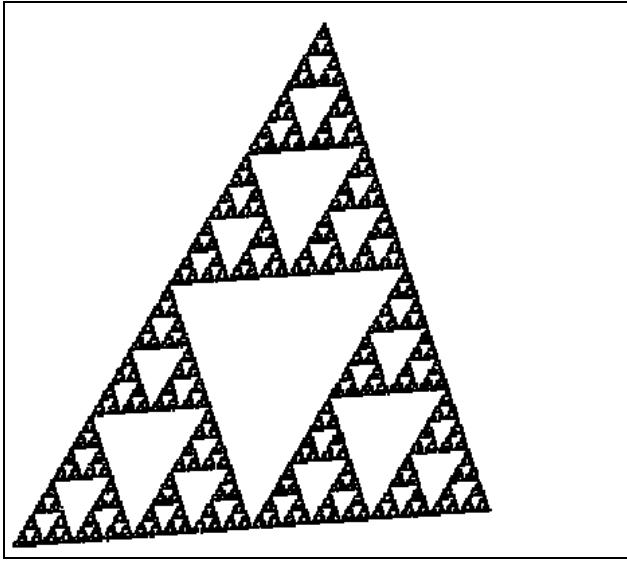


Figure 2.12. The Sierpinski Gasket. (file: fig2.12.bmp)

The Sierpinski gasket is produced by calling `drawDot()` many times with dot positions $(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots$ determined by a simple algorithm. Denote the k -th point $p_k = (x_k, y_k)$. Each point is based on the previous point p_{k-1} . The procedure is:

1. Choose three fixed points T_0, T_1 , and T_2 to form some triangle, as shown in Figure 2.13a.
2. Choose the initial point p_0 to be drawn by selecting one of the points T_0, T_1 , and T_2 at random.

Now iterate steps 3-5 until the pattern is satisfactorily filled in:

3. Choose one of the three points T_0, T_1 , and T_2 at random; call it T .
4. Construct the next point p_k as the **midpoint**⁷ between T and the previously found point p_{k-1} . Hence

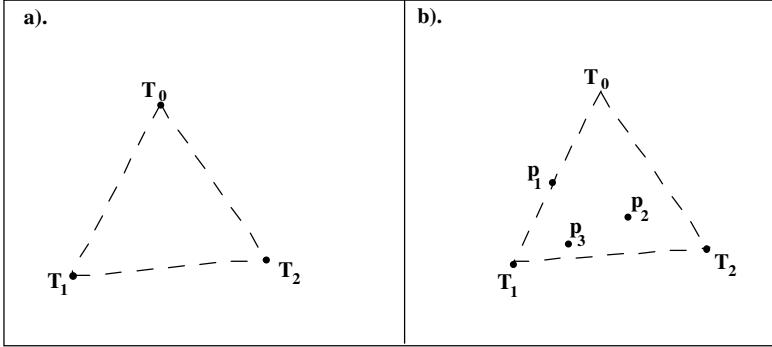


Figure 2.13. Building the Sierpinski gasket.

$p_k = \text{midpoint of } p_{k-1} \text{ and } T$,

5. Draw p_k using `drawDot()`.

Figure 2.13b shows a few iterations of this: Suppose the initial point p_0 happens to be T_0 , and that T_1 is chosen next. Then p_1 is formed so that it lies halfway between T_0 and T_1 . Suppose T_2 is chosen next, so p_2 lies halfway between p_1 and T_2 . Next suppose T_1 is chosen again, so p_3 is formed as shown, etc. This process goes on generating and drawing points (conceptually forever), and the pattern of the Sierpinski gasket quickly emerges.

⁷ To find the midpoint between 2 points, say (3, 12) and (5, 37) simply *average* their x and y components individually: add them and divide by 2. So the midpoint of (3,12) and (5,37) is $((3+5)/2, (12+37)/2) = (4, 24)$.

It is convenient to define a simple class `GLintPoint` that describes a point whose coordinates are integers⁸:

```
class GLintPoint{
public:
    GLint x, y;
};
```

We then build and initialize an array of three such points `T[0]`, `T[1]`, and `T[2]` to hold the three corners of the triangle using `GLintPoint T[3] = {{10,10}, {300,30}, {200, 300}}`. There is no need to store each point p_k in the sequence as it is generated, since we simply want to draw it and then move on. So we set up a variable `point` to hold this changing point. At each iteration `point` is updated to hold the new value.

We use `i = random(3)` to choose one of the points `T[i]` at random. `random(3)` returns one of the values 0, 1, or 2 with equal likelihood. It is defined as⁹

```
int random(int m) { return rand() % m; }
```

Figure 2.14 shows the remaining details of the algorithm, which generates 1000 points of the Sierpinski gasket.

```
void Sierpinski(void)
{
    GLintPoint T[3]= {{10,10}, {300,30}, {200, 300}};

    int index = random(3);           // 0, 1, or 2 equally likely
    GLintPoint point = T[index];    // initial point
    drawDot(point.x, point.y);     // draw initial point
    for(int i = 0; i < 1000; i++)   // draw 1000 dots
    {
        index = random(3);
        point.x = (point.x + T[index].x) / 2;
        point.y = (point.y + T[index].y) / 2;
        drawDot(point.x, point.y);
    }
    glFlush();
}
```

Figure 2.14. Generating the Sierpinski Gasket.

Example 2.2.3. Simple “Dot Plots”.

Suppose you wish to learn the behavior of some mathematical function $f(x)$ as x varies. For example, how does

$$f(x) = e^{-x} \cos(2\pi x)$$

vary for values of x between 0 and 4? A quick plot of $f(x)$ versus x , such as that shown in Figure 2.15, can reveal a lot.

⁸ If C rather than C++ is being used, a simple `struct` is useful here: `typedef struct{GLint x, y;}GLintPoint;`

⁹ Recall that the standard function `rand()` returns a pseudorandom value in the range 0 to 32767. The modulo function reduces it to a value in the range 0 to 2.

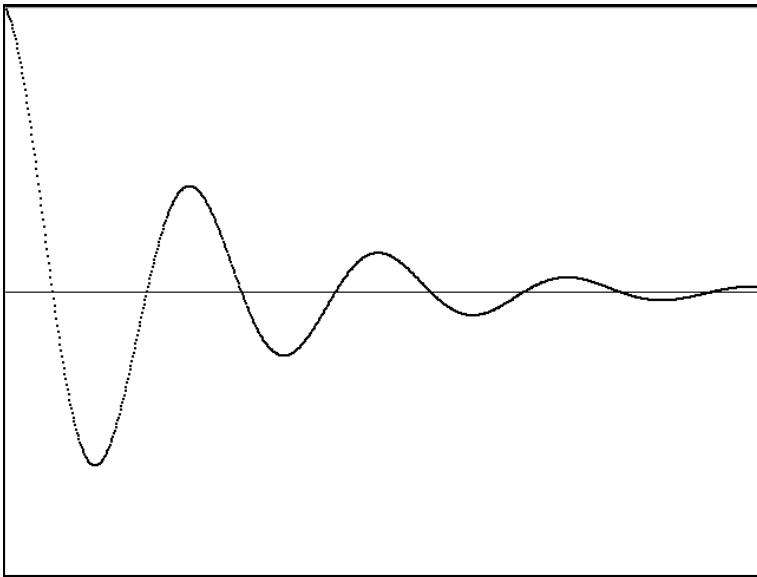


Figure 2.15. A “dot plot” of $e^x \cos(2\pi x)$ versus x . (file: fig2.15.bmp)

To plot this function, simply “sample” it at a collection of equispaced x -values, and plot a dot at each coordinate pair $(x_i, f(x_i))$. Choosing some suitable increment, say 0.005, between consecutive x -values the process is basically:

```
glBegin(GL_POINTS);
    for(GLdouble x = 0; x < 4.0 ; x += 0.005)
        glVertex2d(x, f(x));
glEnd();
glFlush();
```

But there is a problem here: the picture produced will be impossibly tiny because values of x between 0 and 4 map to the first four pixels at the bottom left of the screen window. Further, the negative values of $f()$ will lie below the window and will not be seen at all. We therefore need to scale and position the values to be plotted so they cover the screen window area appropriately. Here we do it by brute force, in essence picking some values to make the picture show up adequately on the screen. Later we develop a general procedure that copes with these adjustments, the so-called procedure of mapping from world coordinates to window coordinates.

- **Scaling x :** Suppose we want the range from 0 to 4 to be scaled so that it covers the entire width of the screen window, given in pixels by `screenWidth`. Then we need only scale all x -values by `screenWidth/4`, using

```
sx = x * screenWidth / 4.0;
```

which yields 0 when x is 0 , and `screenWidth` when x is 4.0, as desired.

- **Scaling, and shifting y :** The values of $f(x)$ lie between -1.0 and 1.0, so we must scale and shift them as well. Suppose we set the screen window to have height `screenHeight` pixels. Then to place the plot in the center of the window scale by `screenHeight / 2` and shift up by `screenHeight / 2`:

```
sy = (y + 1.0) * screenHeight / 2.0;
```

As desired, this yields 0 when y is -1.0, and `screenHeight` when y is 1.0.

Note that the conversions from x to sx , and from y to sy , are of the form:

$$\begin{aligned} sx &= A * x + B \\ sy &= C * y + D \end{aligned} \tag{2.1}$$

for properly chosen values of the constants A , B , C , and D . A and C perform scaling; B and D perform shifting. This scaling and shifting is basically a form of “affine transformation”. We study affine transformations in depth in Chapter 5. They provide a more consistent approach that maps any specified range in x and y to the screen window.

We need only set the values of A , B , C , and D appropriately, and draw the dot -plot using:

```

GLdouble A, B, C, D, x;
A = screenWidth / 4.0;
B = 0.0;
C = screenHeight / 2.0;
D = C;
glBegin(GL_POINTS);
for(x = 0; x < 4.0 ; x += 0.005)
    glVertex2d(A * x + B, C * f(x) + D);
glEnd();
glFlush();

```

Figure 2.16 shows the entire program to draw the dot plot, to illustrate how the various ingredients fit together. The initializations are very similar to those for the program that draws three dots in Figure 2.10. Notice that the width and height of the screen window are defined as constants, and used where needed in the code.

```

myInit();
glutMainLoop();           // go into a perpetual loop
}

```

Figure 2.16. A complete program to draw the “dot plot” of a function.

Practice Exercise 2.2.2. Dot plots for any function $f()$. Consider drawing a dot plot of the function $f(x)$ as in Example 2.2.4, where it is known that as x varies from x_{low} to x_{high} , $f(x)$ takes on values between y_{low} to y_{high} . Find the appropriate scaling and translation factors so that the dots will lie properly in a screen window with width W pixels and height H pixels.

2.3. Making Line-Drawings.

Hamlet: Do you see yonder cloud that's almost in shape of a camel?
Polonius: By the mass, and 'tis like a camel, indeed.
Hamlet: Methinks it is like a weasel.
William Shakespeare, Hamlet

As discussed in Chapter 1, line drawings are fundamental in computer graphics, and almost every graphics system comes with “driver” routines to draw straight lines. OpenGL makes it easy to draw a line: use `GL_LINES` as the argument to `glBegin()`, and pass it the two end points as vertices. Thus to draw a line between (40, 100) and (202, 96) use:

```

glBegin(GL_LINES);      // use constant GL_LINES here
    glVertex2i(40, 100);
    glVertex2i(202, 96);
glEnd();

```

This code might be encapsulated for convenience in the routine `drawLineInt()`:

```

void drawLineInt(GLint x1, GLint y1, GLint x2, GLint y2)
{
    glBegin(GL_LINES);
    glVertex2i(x1, y1);
    glVertex2i(x2, y2);
    glEnd();
}

```

and an alternate routine, `drawLineFloat()` could be implemented similarly (how?).

If more than two vertices are specified between `glBegin(GL_LINES)` and `glEnd()` they are taken in pairs and a separate line is drawn between each pair. The tic-tac-toe board shown in Figure 2.17a would be drawn using:

- a). thin lines b). thick lines c). stippled lines



Figure 2.17. Simple picture built from four lines.

```

glBegin(GL_LINES);
    glVertex2i(10, 20); // first horizontal line
    glVertex2i(40, 20)
    glVertex2i(20, 10); // first vertical line
    glVertex2i(20, 40);
    <four more calls to glVertex2i() here for other two lines>
glEnd();
glFlush();

```

OpenGL provides tools for setting the attributes of lines. A line’s color is set in the same way as for points, using `glColor3f()`. Figure 2.17b shows the use of thicker lines, as set by `glLineWidth(4.0)`. The

default thickness is 1.0. Figure 2.17c shows stippled (dotted and dashed) lines. The details of stippling are addressed in Case Study 2.5 at the end of this chapter.

2.3.1. Drawing Polylines and Polygons.

Recall from Chapter 1 that a **polyline** is a collection of line segments joined end to end. It is described by an ordered list of points, as in:

$$p_0 = (x_0, y_0), p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n). \quad (2.3.1)$$

In OpenGL a polyline is called a “line strip”, and is drawn by specifying the vertices in turn between `glBegin(GL_LINE_STRIP)` and `glEnd()`. For example, the code:

```
glBegin(GL_LINE_STRIP); // draw an open polyline
    glVertex2i(20,10);
    glVertex2i(50,10);
    glVertex2i(20,80);
    glVertex2i(50,80);
glEnd();
glFlush();
```

produces the polyline shown in Figure 2.18a. Attributes such as color, thickness and stippling may be applied to polylines in the same way they are applied to single lines. If it is desired to connect the last point with the first point to make the polyline into a polygon simply replace `GL_LINE_STRIP` with `GL_LINE_LOOP`. The resulting polygon is shown in Figure 2.18b.

Figure 2.18. A polyline and a polygon.

Polygons drawn using `GL_LINE_LOOP` cannot be filled with a color or pattern. To draw filled polygons you use `glBegin(GL_POLYGON)`, as described later.

Example 2.3.1. Drawing Line Graphs. In Example 2.2.3 we looked at plotting a function $f(x)$ versus x with a sequence of dots at positions $(x_i, f(x_i))$. A line graph is a straightforward extension of this: the dots are simply joined by line segments to form a polyline. Figure 2.19 shows an example, based on the function:

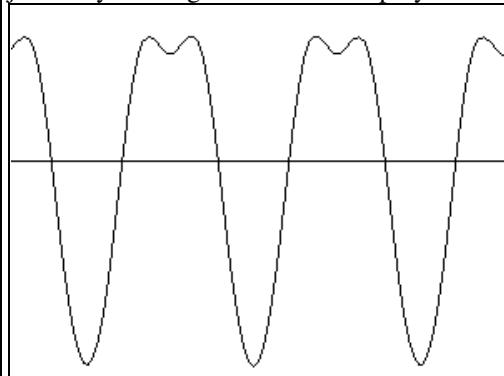


Figure 2.19. A plot of a mathematical formula.

$$f(x) = 300 - 100 \cos(2\pi x/100) + 30 \cos(4\pi x/100) + 6 \cos(6\pi x/100)$$

as x varies in steps of 3 for 100 steps. A blow-up of this figure would show a sequence of connected line segments; in the normal size picture they blend together and appear as a smoothly varying curve.

The process of plotting a function with line segments is almost identical to that for producing a dot plot: the program of Figure 2.17 can be used with only slight adjustments. We must scale and shift the lines being drawn here, to properly place the lines in the window. This requires the computation of the constants A , B , C and D in the same manner as we did before (see Equation 2.1). Figure 2.20 shows the changes necessary for the inner drawing loop in the `myDisplay()` function.

< Calculate constants A, B, C and D for scaling and shifting >

```

glBegin(GL_LINE_STRIP);
  for(x = 0; x <= 300; x += 3)
    glVertex2d(A * x + B, C * f(x) + D);
glEnd();
glFlush;

```

Figure 2.20. Plotting a function using a line graph.

Example 2.3.2. Drawing Polyline stored in a file.

Most interesting pictures made up of polylines contain a rather large number of line segments. It's convenient to store a description of the polylines in a file, so that the picture can be redrawn at will. (Several interesting examples may be found on the Internet - see the Preface.)

It's not hard to write a routine that draws the polylines stored in a file. Figure 2.21 shows an example of what might be drawn.

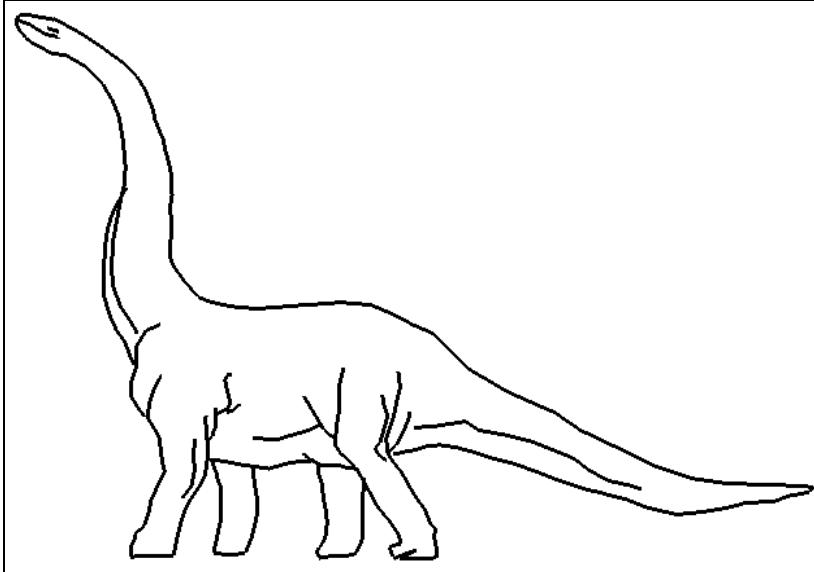


Figure 2.21. Drawing polyline stored in a file. (file: fig2.21.bmp)

Suppose the file `dino.dat` contains a collection of polylines, in the following format (the comments are not part of the file):

21	number of polylines in the file
4	number of points in the first polyline
169 118	first point of first polyline
174 120	second point of first polyline
179 124	
178 126	
5	number of points in the second polyline
298 86	first point of second polyline
304 92	
310 104	
314 114	
314 119	
29	
32 435	
10 439	
. . .	etc.

(The entire file is available on the web site for this book. See the preface.) Figure 2.22 shows a routine in C++ that will open such a file, and then draw each of the polylines it contains. The file having the name contained in the string `fileName` is read in and each polyline is drawn. The routine could be used in place of `myDisplay()` in Figure 2.17 as the callback function for the redraw event. The values of A , B , C and D would have to be chosen judiciously to scale the polylines properly. We develop a general approach to do this in Chapter 3.

```
void drawPolyLineFile(char * fileName)
```

```

{
    fstream inStream;
    inStream.open(fileName, ios ::in); // open the file
    if(inStream.fail())
        return;
    glClear(GL_COLOR_BUFFER_BIT); // clear the screen
    GLint numpolys, numLines, x ,y;
    inStream >> numpolys; // read the number of polylines
    for(int j = 0; j < numpolys; j++) // read each polyline
    {
        inStream >> numLines;
        glBegin(GL_LINE_STRIP); // draw the next polyline
        for (int i = 0; i < numLines; i++)
        {
            inStream >> x >> y; // read the next x, y pair
            glVertex2i(x, y);
        }
        glEnd();
    }
    glFlush();
    inStream.close();
}

```

Figure 2.22. Drawing polylines stored in a file.

This version of `drawPolyLineFile()` does very little error checking. If the file cannot be opened — perhaps the wrong name is passed to the function — the routine simply returns. If the file contains bad data, such as real values where integers are expected, the results are unpredictable. The routine as given should be considered only as a starting point for developing a more robust version.

Example 2.3.3. Parameterizing Figures.

Figure 2.23 shows a simple house consisting of a few polylines. It can be drawn using code shown partially in Figure 2.24. (What code would be suitable for drawing the door and window?)

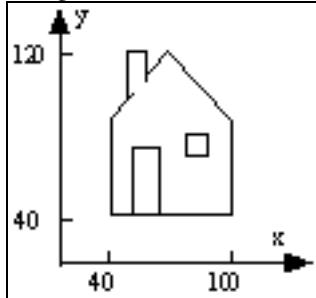


Figure 2.23. A House.

```

void hardwiredHouse(void)
{
    glBegin(GL_LINE_LOOP);
        glVertex2i(40, 40); // draw the shell of house
        glVertex2i(40, 90);
        glVertex2i(70, 120);
        glVertex2i(100, 90);
        glVertex2i(100, 40);
    glEnd();
    glBegin(GL_LINE_STRIP);
        glVertex2i(50, 100); // draw the chimney
        glVertex2i(50, 120);
        glVertex2i(60, 120);
        glVertex2i(60, 110);
    glEnd();
    . . . // draw the door
    . . . // draw the window
}

```

Figure 2.24. Drawing a house with “hard-wired” dimensions.

This is not a very flexible approach. The position of each endpoint is hard-wired into this code, so `hardwiredHouse()` can draw only one house in one size and one location. More flexibility is achieved if we **parameterize** the figure, and pass the parameter values to the routine. In this way we can draw **families** of objects, which are distinguished by different parameter values. Figure 2.25 shows this approach. The parameters specify the location of the peak of the roof, the width of the house, and its height. The details of drawing the chimney, door, and window are left as an exercise.

```
void parameterizedHouse(GLintPoint peak, GLint width, GLint height)
    // the top of house is at the peak; the size of house is given
    // by height and width
{
    glBegin(GL_LINE_LOOP);
        glVertex2i(peak.x, peak.y); // draw shell of house
        glVertex2i(peak.x + width / 2, peak.y - 3 * height / 8);
        glVertex2i(peak.x + width / 2, peak.y - height);
        glVertex2i(peak.x - width / 2, peak.y - height);
        glVertex2i(peak.x - width / 2, peak.y - 3 * height / 8);
    glEnd();
    draw chimney in the same fashion
    draw the door
    draw the window
}
```

Figure 2.25. Drawing a parameterized house.

This routine may be used to draw a “village” as shown in Figure 2.26, by making successive calls to `parameterizedHouse()` with different parameter values. (How is a house “flipped” upside down? Can all of the houses in the figure be drawn using the routine given?)

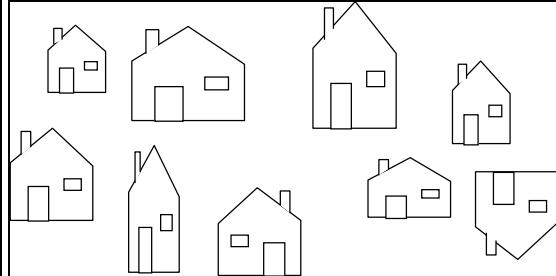


Figure 2.26. A “village” of houses drawn using `parameterizedHouse()`.

Example 2.3.4. Building a Polyline Drawer.

As we shall see, some applications compute and store the vertices of a polyline in a list. It is natural, therefore, to add to our growing toolbox of routines a function that accepts the list as a parameter and draws the corresponding polyline. The list might be in the form of an array, or a linked list. We show here the array form, and define the class to hold it in Figure 2.27.

```
class GLintPointArray{
    const int MAX_NUM = 100;
    public:
        int num;
        GLintPoint pt[MAX_NUM];
};
```

Figure 2.27. Data type for a linked list of vertices.

Figure 2.28 shows a possible implementation of the polyline drawing routine. It also takes a parameter `closed`: if `closed` is nonzero the last vertex in the polyline is connected to the first vertex. The value of `closed` sets the argument of `glBegin()`. The routine simply sends each vertex of the polyline to OpenGL.

```
void drawPolyLine(GLintPointArray poly, int closed)
{
    glBegin(closed ? GL_LINE_LOOP : GL_LINE_STRIP);
        for(int i = 0; i < poly.num; i++)
            glVertex2i(poly.pt[i].x, poly.pt[i].y);
    glEnd();
    glFlush();
}
```

Figure 2.28. A linked list data type, and drawing a polyline or polygon.

2.3.3. Line Drawing using `moveto()` and `lineto()`.

As we noted earlier a number of graphics systems provide line drawing tools based on the functions `moveto()` and `lineto()`. These functions are so common it is important to be familiar with their use. We shall fashion our own `moveto()` and `lineto()` that operate by calling OpenGL tools. In Chapter 3 we shall also dive “under the hood” to see how you would build `moveto()` and `lineto()` based on first principles, if a powerful library like OpenGL were not available.

Recall that `moveto()` and `lineto()` manipulate the position of a hypothetical pen, whose position is called the **current position**, or *CP*. We can summarize the effects of the two functions as:

```
moveto(x, y):    set CP to (x, y)
lineto(x, y):    draw a line from C2970P to (x, y), and then update CP to (x, y)
```

A line from (x_1, y_1) to (x_2, y_2) is therefore drawn using the two calls `moveto(x1, y1); lineto(x2, y2)`. A polyline based on the list of points $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$ is easily drawn using:

```
moveto(x[0], y[0]);
for(int i = 1; i < n; i++)
    lineto(x[i], y[i]);
```

It is straightforward to build `moveto()` and `lineto()` on top of OpenGL. To do this we must define and maintain our own *CP*. For the case of integer coordinates the implementation shown in Figure 2.29 would do the trick.

```
GLintPoint CP;           // global current position

//<<<<<<<<< moveto >>>>>>>>>
void moveto(GLint x, GLint y)
{
    CP.x = x; CP.y = y; // update the CP
}
//<<<<<<<<< lineTo >>>>>>>>>>
void lineto(GLint x, GLint y)
{
    glBegin(GL_LINES); // draw the line
    glVertex2i(CP.x, CP.y);
    glVertex2i(x, y);
    glEnd();
    glFlush();
    CP.x = x; CP.y = y; // update the CP
}
```

Figure 2.29. Defining `moveto()` and `lineto()` in OpenGL.

2.3.4. Drawing Aligned Rectangles.

A special case of a polygon is the **aligned rectangle**, so called because its sides are aligned with the coordinate axes. We could create our own function to draw an aligned rectangle (how?), but OpenGL provides the ready-made function:

```
glRecti(GLint x1, GLint y1, GLint x2, GLint y2);
// draw a rectangle with opposite corners (x1, y1) and (x2, y2);
// fill it with the current color;
```

This command draws the aligned rectangle based on two given points. In addition the rectangle is filled with the current color. Figure 2.30 shows what is drawn by the code:

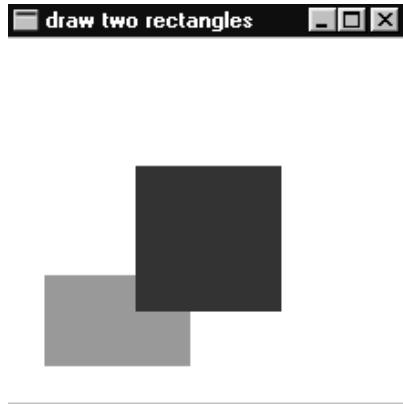


Figure 2.30. Two aligned rectangles filled with colors.

```
glClearColor(1.0,1.0,1.0,0.0); // white background
glClear(GL_COLOR_BUFFER_BIT); // clear the window
	glColor3f(0.6,0.6,0.6); // bright gray
	glRecti(20,20,100,70);
	glColor3f(0.2,0.2,0.2); // dark gray
	glRecti(70, 50, 150, 130);
	glFlush();
```

Notice that the second rectangle is “painted over” the first one. We examine other “drawing modes” in Chapter 10.

Figure 2.31 shows two further examples. Part a) is a “flurry” of randomly chosen aligned rectangles, that might be generated by code such as¹⁰:

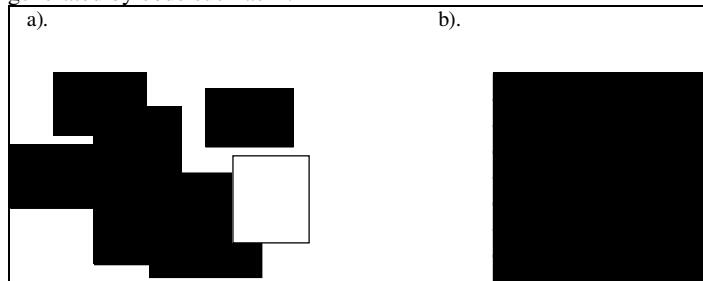


Figure 2.31. a). Random Flurry of rectangles. b). a checkerboard.

```
void drawFlurry(int num, int numColors, int Width, int Height)
// draw num random rectangles in a Width by Height rectangle
{
    for (int i = 0; i < num; i++)
    {
        GLint x1 = random(Width); // place corner randomly
        GLint y1 = random(Height);
        GLint x2 = random(Width); // pick the size so it fits
        GLint y2 = random(Height);
        GLfloat lev = random(10)/10.0; // random value, in range 0 to 1
        glColor3f(lev,lev,lev); // set the gray level
        glRecti(x1, y1, x2, y2); // draw the rectangle
    }
    glFlush();
}
```

Part b) is the familiar checkerboard, with alternating gray levels. The exercises ask you to generate it.

2.3.5. Aspect Ratio of an Aligned Rectangle.

¹⁰ Recall that `random(N)` returns a randomly-chosen value between 0 and $N - 1$ (see Appendix 3).

The principal properties of an aligned rectangle are its size, position, color, and “shape”. Its shape is embodied in its **aspect ratio**, and we shall be referring to the aspect ratios of rectangles throughout the book. The **aspect ratio** of a rectangle is simply the ratio of its width to its height¹¹:

$$\text{aspect ratio} = \frac{\text{width}}{\text{height}} \quad (2.2)$$

Rectangles with various aspect ratios are shown in Figure 2.32..

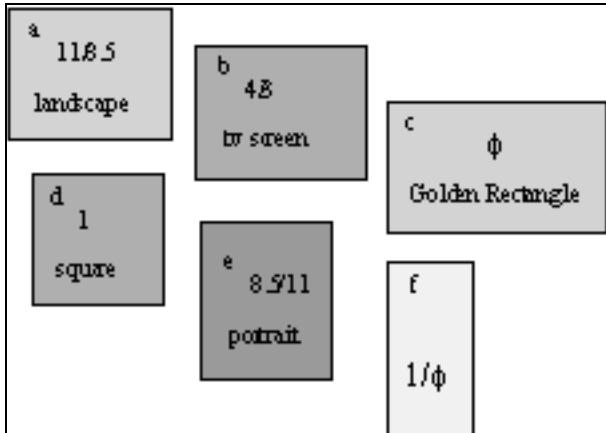


Figure 2.32. Examples of aspect ratios of aligned rectangles.

Rectangle *A* has the shape of a piece of 8.5 by 11 inch paper laid on its side in the so-called **landscape** orientation (i.e. width larger than height). It has an aspect ratio of 1.294. Rectangle *B* has the aspect ratio of a television screen, 4/3, and *C* is the famous **golden rectangle** described in Case Study 2.3. Its aspect ratio is close to $\phi = 1.618034$. Rectangle *D* is a square with aspect ratio equal to 1, and *E* has the shape of a piece of standard paper in **portrait** orientation, with an aspect ratio of .7727. Finally, *F* is tall and skinny with an aspect ratio of $1/\phi$.

Practice Exercises.

2.3.1. Drawing the checkerboard.

(Try your hand at this before looking at the answers.) Write the routine `checkerboard(int size)` that draws the checkerboard shown in Figure 2.31b. Place the checkerboard with its lower left corner at (0,0). Each of the 64 squares has length `size` pixels. Choose two nice colors for the squares. **Solution:** The ij -th square has lower left corner at $(i*size, j*size)$ for $i = 0..,7$ and $j = 0..,7$. The color can be made to alternate between (r_1, g_1, b_1) and (r_2, g_2, b_2) using

```
if((i + j)%2 ==0) // if i + j is even
    glColor3f( r1, g1, b1);
else
    glColor3f(r2, g2, b2);
```

2.3.2. Alternative ways to specify a rectangle. An aligned rectangle can be described in other ways than by two opposite corners. Two possibilities are:

- * its center point, height, and width;
- * its upper left corner, width, and aspect ratio.

Write functions `drawRectangleCenter()` and `drawRectangleCornerSize()` that pass these alternative parameters.

2.3.3. Different Aspect Ratios. Write a short program that draws a filled rectangle of aspect ratio R , where R is specified by the user. Initialize the display to a drawing space of 400 by 400. Arrange the size of the rectangle so that it is as large as possible. That is, if $R > 1$ it spans across the drawing space, and if $R < 1$ it spans from top to bottom.

2.3.4. Drawing the Parametrized house. Fill in the details of `parametrizedHouse()` in Figure 2.25 so that the door, window, and chimney are drawn in their proper proportions for given values of `height` and `width`.

¹¹Alert! Some authors define it as height / width.

2.3.5. Scaling and positioning a figure using parameters. Write the function `void drawDiamond(GLint center, int size)` that draws the simple diamond shown in Figure 2.33, centered at `center`, and having size `size`.

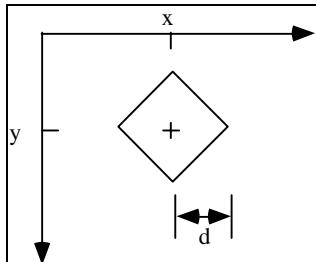


Figure 2.33. A simple diamond.

Use this function to draw a “flurry” of diamonds as suggested in Figure 2.34.

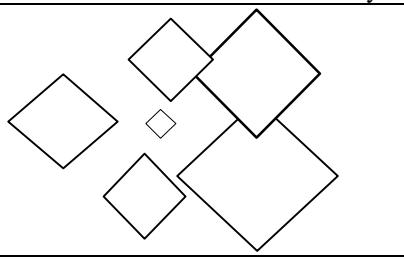


Figure 2.34. A flurry of diamonds.

2.3.6. Filling Polygons.

So far we can draw unfilled polygons in OpenGL, as well as aligned rectangles filled with a single solid color. OpenGL also supports filling more general polygons with a pattern or color. The restriction is that the polygons must be *convex*.

Convex polygon:

a polygon is convex if a line connecting any two points of the polygon lies entirely within the polygon.

Several polygons are shown in Figure 2.35. Of these only *D*, *E*, and *F* are convex. (Check that the definition of convexity is upheld for each of these polygons.) *D* is certainly convex: all triangles are. *A* is not even simple (recall Chapter 1) so it cannot be convex. Both *B* and *C* “bend inward” at some point. (Find two points on *B* such that the line joining them does not lie entirely inside *B*.)

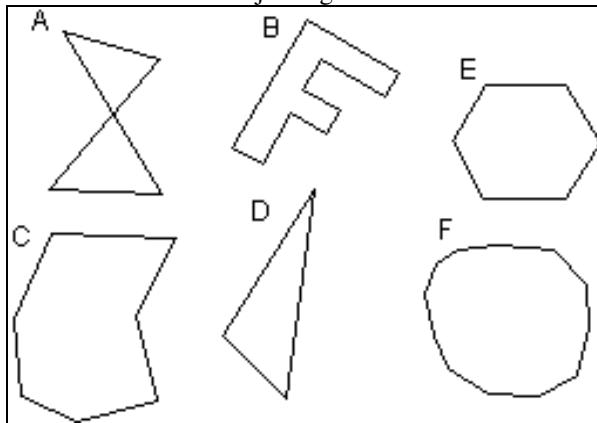


Figure 2.35. Convex and non-convex polygons.

To draw a convex polygon based on vertices $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ use the usual list of vertices, but place them between a `glBegin(GL_POLYGON)` and an `glEnd()`:

```
glBegin(GL_POLYGON);
    glVertex2f(x0, y0);
    glVertex2f(x1, y1);
    ...

```

```

glVertex2f(xn, yn);
glEnd();

```

It will be filled in the current color. It can also be filled with a stipple pattern – see Case Study 2.5, and later we will paint images into polygons as part of applying a texture.

Figure 2.36 shows a number of filled convex polygons. In Chapter 10 we will examine an algorithm for filling any polygon, convex or not.

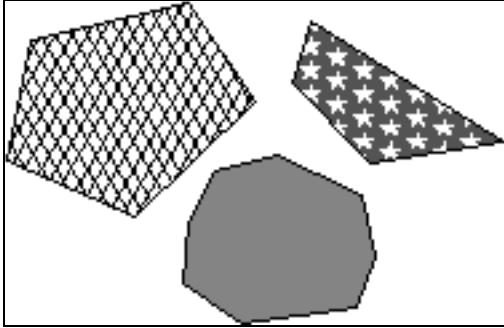


Figure 2.36. Several filled convex polygons.

2.3.7. Other Graphics Primitives in OpenGL.

OpenGL supports the drawing of five other objects as well. Figure 2.37 shows examples of each of them. To draw a particular one the constant shown with it is used in `glBegin()`.

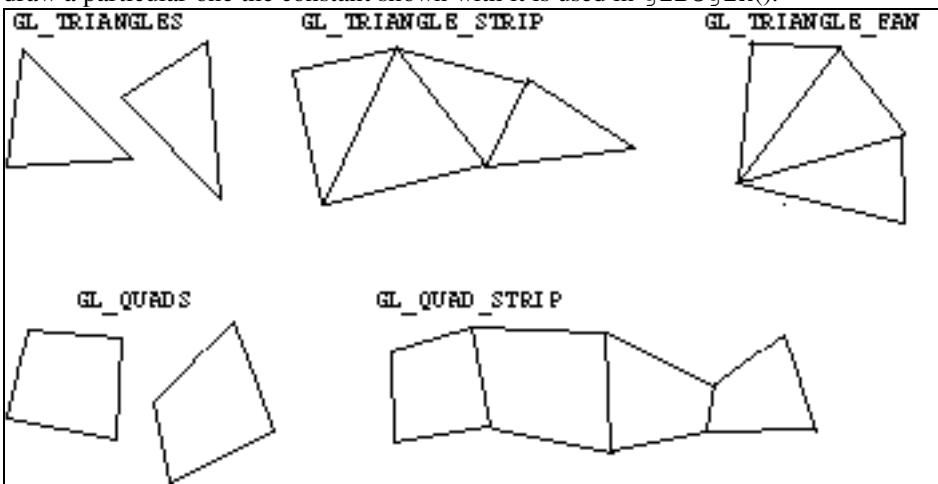


Figure 2.37. Other geometric primitive types.

The following list explains the function of each of the five constants:

- **GL_TRIANGLES**: takes the listed vertices three at a time, and draws a separate triangle for each;
- **GL_QUADS**: takes the vertices four at a time and draws a separate quadrilateral for each;
- **GL_TRIANGLE_STRIP**: draws a series of triangles based on triplets of vertices: v_0, v_1, v_2 , then v_2, v_1, v_3 , then v_2, v_3, v_4 , etc. (in an order so that all triangles are “traversed” in the same way; e.g. counterclockwise).
- **GL_TRIANGLE_FAN**: draws a series of connected triangles based on triplets of vertices: v_0, v_1, v_2 , then v_0, v_2, v_3 , then v_0, v_3, v_4 , etc.
- **GL_QUAD_STRIP**: draws a series of quadrilaterals based on foursomes of vertices: first v_0, v_1, v_3, v_2 , then v_2, v_3, v_5, v_4 , then v_4, v_5, v_7, v_6 (in an order so that all quadrilaterals are “traversed” in the same way; e.g. counterclockwise).

2.4. Simple Interaction with the mouse and keyboard.

Interactive graphics applications let the user control the flow of a program by natural human motions: pointing and clicking the mouse, and pressing various keyboard keys. The mouse position at the time of the click, or the identity of the key pressed, is made available to the application program and is processed as appropriate.

Recall that when the user presses or releases a mouse button, moves the mouse, or presses a keyboard key, an event occur. Using the OpenGL Utility Toolkit (GLUT) the programmer can register a callback function with each of these events by using the following commands:

- `glutMouseFunc(myMouse)` which registers `myMouse()` with the event that occurs when the mouse button is pressed or released;
- `glutMotionFunc(myMovedMouse)` which registers `myMovedMouse()` with the event that occurs when the mouse is moved while one of the buttons is pressed;
- `glutKeyboardFunc(myKeyboard)` which registers `myKeyboard()` with the event that occurs when a keyboard key is pressed.

We next see how to use each of these.

2.4.1. Mouse interaction.

How is data about the mouse sent to the application? You must design the callback function `myMouse()` to take four parameters, so that it has the prototype:

```
void myMouse(int button, int state, int x, int y);
```

When a mouse event occurs the system calls the registered function, supplying it with values for these parameters. The value of `button` will be one of:

```
GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON,
```

with the obvious interpretation, and the value of `state` will be one of: `GLUT_UP` or `GLUT_DOWN`. The values `x` and `y` report the position of the mouse at the time of the event. **Alert:** The `x` value is the number of pixels from the left of the window as expected, but the `y` value is the number of pixels *down* from the top of the window!

Example 2.4.1. Placing dots with the mouse.

We start with an elementary but important example. Each time the user presses down the left mouse button a dot is drawn in the screen window at the mouse position. If the user presses the right button the program terminates. The version of `myMouse()` shown next does the job. Because the `y`-value of the mouse position is the number of pixels from the top of the screen window, we draw the dot, not at `(x, y)`, but at `(x, screenHeight - y)`, where `screenHeight` is assumed here to be the height of the window in pixels.

```
void myMouse(int button, int state, int x, int y)
{
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        drawDot(x, screenHeight - y);
    else if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        exit(-1);
}
```

The argument of `-1` in the standard function `exit()` simply returns `-1` back to the operating system: It is usually ignored.

Example 2.4.2. Specifying a rectangle with the mouse.

Here we want the user to be able to draw rectangles whose dimensions are entered with the mouse. The user clicks the mouse at two points which specify opposite corners of an aligned rectangle, and the rectangle is drawn. The data for each rectangle need not be retained (except through the picture of the rectangle itself): each new rectangle replaces the previous one. The user can clear the screen by pressing the right mouse button.

The routine shown in Figure 2.38 stores the corner points in a `static` array `corner []`. It is made `static` so values are retained in the array between successive calls to the routine. Variable `numCorners` keeps track of how many corners have been entered so far: when this number reaches two the rectangle is drawn, and `numCorners` is reset to 0.

```
void myMouse(int button, int state, int x, int y)
{
```

```

static GLintPoint corner[2];
static int numCorners = 0;                                // initial value is 0
if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
{
    corner[numCorners].x = x;
    corner[numCorners].y = screenHeight - y;      // flip y coordinate
    numCorners++;                                // have another point
    if(numCorners == 2)
    {
        glRecti(corner[0].x, corner[0].y, corner[1].x, corner[1].y);
        numCorners = 0;                          // back to 0 corners
    }
}
else if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    glClear(GL_COLOR_BUFFER_BIT);                // clear the window
    glFlush();
}

```

Figure 2.38. A callback routine to draw rectangles entered with the mouse.

An alternative method for designating a rectangle uses a **rubber rectangle** that grows and shrinks as the user moves the mouse. This is discussed in detail in Section 10.3.3.

Example 2.4.3. Controlling the Sierpinski gasket with the mouse.

It is simple to extend the Sierpinski gasket routine described earlier so that the user can specify the three vertices of the initial triangle with the mouse. We use the same process as in the previous example: gather the three points in an array `corners[]`, and when three points are available draw the Sierpinski gasket. The meat of the `myMouse()` routine is therefore:

```

static GLintPoint corners[3];
static int numCorners = 0;
if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
{
    corner[numCorners].x = x;
    corner[numCorners].y = screenHeight - y;      // flip y coordinate
    if(++numCorners == 3)
    {
        Sierpinski(corners);                    // draw the gasket
        numCorners = 0;                        // back to 0 corners
    }
}

```

where `Sierpinski()` is the same as in Figure 2.15 except the three vertices of the triangle are passed as parameters.

Example 2.4.4. Create a polyline using the mouse.

Figure 2.39 shows a polyline being created with mouse clicks. Here, instead of having each new point replace the previous one we choose to retain all the points clicked for later use. The user enters a succession of points with the mouse, and each point is stored in the next available position of the array. If the array becomes full no further points are accepted. After each click of the mouse the window is cleared and the entire current polyline is redrawn. The polyline is reset to empty if the right mouse button is pressed.

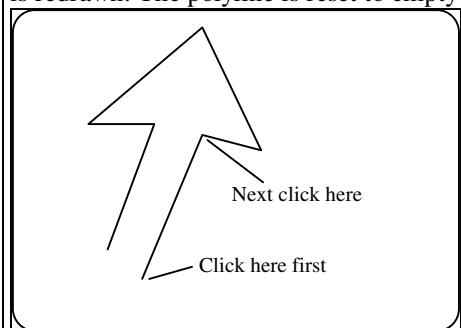


Figure 2.39. Interactive creation of a polyline.

Figure 2.40 shows one possible implementation. Note that `last` keeps track of the last index used so far in the array `List[]`; it is incremented as each new point is clicked, and set to `-1` to make the lists empty. If it were desirable to make use of the points in `List` outside of `myMouse()`, the variable `List` could be made global.

```
void myMouse(int button, int state, int x, int y)
{
#define NUM 20
static GLintPoint List[NUM];
    static int last = -1; // last index used so far

    // test for mouse button as well as for a full array
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN && last < NUM -1)
    {
        List[++last].x = x; // add new point to list
        List[ last].y = screenHeight - y; // window height is 480
        glClear(GL_COLOR_BUFFER_BIT); // clear the screen
        glBegin(GL_LINE_STRIP); // redraw the polyline
            for(int i = 0; i <= last; i++)
                glVertex2i(List[i].x, List[i].y);
        glEnd();
        glFlush();
    }
    else if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
last = -1; // reset the list to empty
}
```

Figure 2.40. A polyline drawer based on mouse clicks.

Mouse motion.

An event of a different type is generated when the mouse is moved (more than some minimal distance) while some button is held down. The callback function, say `myMovedMouse()`, is registered with this event using

```
glutMotionFunc(myMovedMouse);
```

The callback function must take two parameters and have the prototype: `myMovedMouse(int x, int y)`; The values of `x` and `y` are of course the position of the mouse when the event occurred.

Example 2.4.4. “Freehand” drawing with a fat brush.

Suppose we want to create a curve by sweeping the mouse along some trajectory with a button held down. In addition we want it to seem that the drawing “brush” has a square shape. This can be accomplished by designing `myMovedMouse()` to draw a square at the current mouse position:

```
void myMovedMouse(int mouseX, int mouseY)
{
    GLint x = mouseX; //grab the mouse position
    GLint y = screenHeight - mouseY; // flip it as usual
    GLint brushSize = 20;
    glRecti(x,y, x + brushSize, y + brushSize);
    glFlush();
}
```

2.4.2. Keyboard interaction.

As mentioned earlier, pressing a key on the keyboard queues a keyboard event. The callback function `myKeyboard()` is registered with this type of event through `glutKeyboardFunc(myKeyboard)`. It must have prototype:

```
void myKeyboard(unsigned int key, int x, int y);
```

The value of key is the ASCII value¹² of the key pressed. The values x and y report the position of the mouse at the time that the event occurred. (As before y measures the number of pixels down from the top of the window.)

The programmer can capitalize on the many keys on the keyboard to offer the user a large number of choices to invoke at any point in a program. Most implementations of myKeyboard() consist of a large switch statement, with a case for each key of interest. Figure 2.41 shows one possibility. Pressing ‘p’ draws a dot at the mouse position; pressing the left arrow key adds a point to some (global) list, but does no drawing¹³; pressing ‘E’ exits from the program. Note that if the user holds down the ‘p’ key and moves the mouse around a rapid sequence of points is generated to make a “freehand” drawing.

```
void myKeyboard(unsigned char theKey, int mouseX, int mouseY)
{
    GLint x = mouseX;
    GLint y = screenHeight - mouseY; // flip the y value as always
    switch(theKey)
    {
        case 'p':
            drawDot(x, y); // draw a dot at the mouse position
            break;
        case GLUT_KEY_LEFT: List[++last].x = x; // add a point
                            List[last].y = y;
            break;
        case 'E':
            exit(-1); //terminate the program
        default:
            break; // do nothing
    }
}
```

Figure 2.41. An example of the keyboard callback function.

2.5. Summary

The hard part in writing graphics applications is getting started: pulling together the hardware and software ingredients in a program to make the first few pictures. The OpenGL application programmer interface (API) helps enormously here, as it provides a powerful yet simple set of routines to make drawings. One of its great virtues is device independence, which makes it possible to write programs for one graphics environment, and use the same program without changes in another environment.

Most graphics applications are written today for a windows-based environment. The program opens a window on the screen that can be moved and resized by the user, and it responds to mouse clicks and key strokes. We saw how to use OpenGL functions that make it easy to create such a program.

Primitive drawing routines were applied to making pictures composed of dots, lines, polylines, and polygons, and were combined into more powerful routines that form the basis of one’s personal graphics toolkit. Several examples illustrated the use of these tools, and described methods for interacting with a program using the keyboard and mouse. The Case studies presented next offer additional programming examples that explore deeper into the topics discussed so far, or branch out to interesting related topics.

2.6. Case Studies.

It is best while using this text to try out new ideas as they are introduced, to solidify the ideas presented. This is particularly true in the first few chapters, since getting started with the first graphics programs often presents a hurdle. To focus this effort, each chapter ends with some **Case Studies** that describe programming projects that are both interesting in themselves, and concentrate on the ideas developed in the chapter.

¹² ASCII stands for American Standard Code for Information Interchange. Tables of ASCII values are readily available on the internet. Also see `ascii.html` in the web site for this book.

¹³ Names for the various “special” keyboard keys, such as the function keys, arrow keys, and “home”, may be found in the include file `glut.h`.

Some of the Case Studies are simple exercises that only require fleshing out some pseudocode given in the text, and then running the program through its paces. Others are much more challenging, and could be the basis of a major programming project within a course. It is always difficult to judge how much time someone else will need to accomplish any project. The “**Level of Effort**” that accompanies each Case Study is a rough guess at best.

Level of Effort:

I: a simple exercise. It could be assigned for the next class.

II: an intermediate exercise. It probably needs several days for completion¹⁴.

III: An advanced exercise. It would probably be assigned for two weeks or so ahead.

2.6.1. Case Study 2.1. Pseudo random Clouds of Dots.

(Level of Effort: II) The random number generator (RNG) `random(N)` (see Appendix 3) produces a value between 0 and $N-1$ each time it is called. It uses the standard C++ function `rand()` to generate values. Each value appears to be randomly selected, and to have no relation to its predecessors.

In fact the successive numbers that `rand()` produces are not generated randomly at all, but rather through a very regular mechanism where each number n_i is determined from its predecessor n_{i-1} by a specific formula. A typical formula is:

$$n_i = (n_{i-1} * A + B) \bmod N \quad (2.3)$$

where A , B , and N are suitably chosen constants. One set of numbers that works fairly well is: $A = 1103515245$, $B = 12345$, and $N = 32767$. Multiplying n_{i-1} by A and adding B forms a large value, and the modulo operation brings the value into the range 0 to $N-1$. The process begins with some “seed” value chosen for n_0 .

Because the numbers only give an appearance of randomness they are called **pseudo random** numbers. The choices of the values for A , B , and N are very important, and slightly different values give rise to very different characteristics in the sequence of numbers. More details can be found in [knuth, weiss98].

Scatter Plots.

Some experiments yield data consisting of many pairs of numbers (a_i, b_i) , and the goal is to infer visually how the a -values and b -values are “related”. For instance, a large number of people are measured, and one wonders if there is a strong correlation between a person’s height and weight.

A scatter plot can be used to give visual insight into the data. The data for each person is plotted as a dot at position $(height, weight)$ so only the `drawDot()` tool is needed. Figure 2.42 shows an example. It suggests that a person’s height and weight are roughly linearly related, although some people (such as A) are idiosyncratic, being very tall yet quite light.

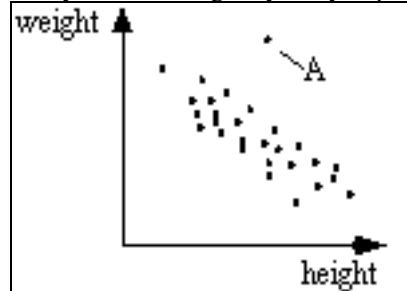


Figure 2.42. A scatter plot of people’s height versus weight.

¹⁴ A “day of programming” means several two hour “sessions”, with plenty of thinking (and resting) time between sessions. It also assumes a reasonably skilled programmer (with at least two semesters of programming in hand), who is familiar with the idiosyncrasies of the language and the platform being used. It does *not* allow for those dreadful hours we all know too well of being stuck with some obscure bug that presents a brick wall of frustration until it is ferreted out and squashed.

Here we use scatter plots to visually test the quality of a random number generator. Each time the function `random(N)` is called it returns a value in the range $0..N - 1$ that is apparently chosen at random, unrelated to values previously returned from `random(N)`. But are successive values truly unrelated?

One simple test builds a scatter plot based on pairs of successive values returned by `random(N)`. It calls `random(N)` twice in succession, and plots the first value against the second. This can be done using `drawDot()`:

```
for(int i = 0; i < num; i++)
    drawDot(random(N), random(N));
```

or in "raw" OpenGL by placing the `for` loop between `glBegin()` and `glEnd()`:

```
glBegin(GL_POINTS);
    for(int i = 0; i < num; i++)           // do it num times
        glVertex2i(random(N), random(N));
glEnd();
```

It is more efficient to do it the second way, which avoids the overhead associated with making many calls to `glBegin()` and `glEnd()`.

Figure 2.43 shows a typical plot that might result. There should be a “uniform” density of dots throughout the square, to reassure you that the values $0..N-1$ occur with about equal likelihood, and that there is no discernible dependence between one value and its successor.

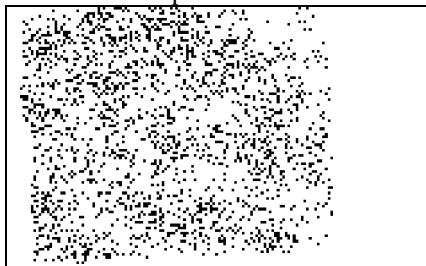


Figure 2.43. A constellation of 500 random dots.

Figure 2.44 shows what can happen with an inferior RNG. In a) there is too high a density of certain values, so the distribution is not uniform in $0..N-1$. In b) there is high correlation between a number and its successor: when one number is large the other tends to be small. And c) shows perhaps the worst situation of all: after a few dozen values have been generated the pattern *repeats*, and no new dots are generated!

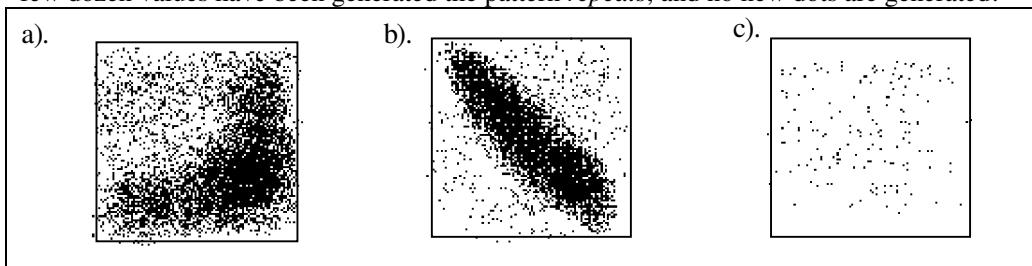


Figure 2.44. Scatter plots for inferior random number generators.

Plotting dot constellations such as these can provide a rough first check on the uniformity of the numbers generated. It is far from a thorough test, however [knuth].

Write a program that produces random dot plots, using some different RNG's to produce the (x, y) pairs. Try different constants A , B , and N in the basic RNG and see the effect this has on the dot constellations. (Warning: if the dot constellation suddenly “freezes” such that no new dots appear, it may be that the pattern of numbers is simply repeating.)

2.6.2. Case Study 2.2. Introduction to Iterated Function Systems.

David Hilbert, defending Cantor's set theory

(Level of Effort: II.) The repetitive operation of drawing the Sierpinski gasket is an example of an **iterated function system (IFS)**, which we shall encounter a surprising number of times throughout the book. Many interesting computer-generated figures (fractals, the Mandelbrot set, etc.) are based on variations of it.

A hand calculator provides a tool for experimenting with a simple IFS: Enter some (positive) number num and press the square root key. This produces a new number \sqrt{num} . Press the square root key again to take its square root, yielding $\sqrt{\sqrt{num}}$. Keep doing this forever..., or until satisfied. We are *iterating* with the square root function, and each result is used as the input for the next square root. An initial value of $num = 64$ yields the sequence: 64, 8, 2.8284, 1.68179,... (Is there a value to which this sequence converges?)

Figure 2.45 presents the system schematically, showing that each output value is *fed back* to have its square root formed, again and again.

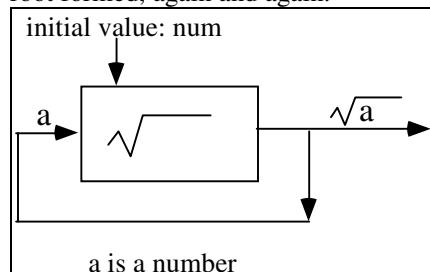


Figure 2.45. Taking the square root repetitively.

In this example the function being iterated is $f(x) = \sqrt{x}$, or symbolically $f(\cdot) = \sqrt{\cdot}$, the “square-rooter”. Other functions $f(\cdot)$ can be used instead, such as:

- $f(\cdot) = 2(\cdot)$; the “doubler” doubles its argument;
- $f(\cdot) = \cos(\cdot)$; the “cosiner”;
- $f(\cdot) = 4(\cdot)(1 - (\cdot))$ the “logistic” function, used in Chaos theory (see Chapter 3).
- $f(\cdot) = (\cdot)^2 + c$ for a constant c ; used to define the Mandelbrot set (see Chapter 8);

It is sometimes helpful to give a name to each number that emerges from the IFS. We call the k -th such number d_k , and say that the process begins at $k = 0$ by “injecting” the initial value d_0 into the system. Then the sequence of values generated by the IFS is:

$$\begin{aligned} d_0 \\ d_1 &= f(d_0) \\ d_2 &= f(f(d_0)) \\ d_3 &= f(f(f(d_0))) \\ \dots \end{aligned}$$

so d_3 is formed by applying function $f(\cdot)$ three times. This is called **the third iterate** of $f()$ applied to the initial value d_0 . More succinctly we can denote the **k -th iterate** of $f()$ by

$$d_k = f^{[k]}(d_0) \quad (2.4)$$

meaning the value produced after $f(\cdot)$ has been applied k times to d_0 . (Note: it does *not* mean the value $f(d_0)$ is raised to the k -th power.) We can also use the recursive form and say:

$$d_k = f(d_{k-1}) \text{ for } k = 1, 2, 3, \dots, \text{ for a given value of } d_0.$$

This sequence of values $d_0, d_1, d_2, d_3, d_4, \dots$ is called “the **orbit** of d_0 ” for the system.

Example: The orbit of 64 for the function $f(\cdot) = \sqrt{\cdot}$ is 64, 8, 2.8284, 1.68179,..., and the orbit of 10000 is 100, 10, 3.162278, 1.77828,... (What is the orbit of 0? What is the orbit of 0.1?)

Example: The orbit of 7 for the “doubler” $f(.) = 2 \cdot (.)$ is: 7, 14, 28, 56, 112, ... The k -th iterate is $7 * 2^k$.

Example: The orbit of 1 for $f(.) = \sin(.)$ can be found using a hand calculator: 1, .8414, .7456, .6784, ... , which very slowly approaches the value 0. (What is the orbit of 1 for $\cos(.)$? In particular, to what value does the orbit converge?)

Project 1: Plotting the Hailstone sequence.

Consider iterating the intriguing function $f(.)$:

$$f(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even} \\ 3x + 1 & \text{if } x \text{ is odd} \end{cases} \quad (2.5)$$

Even valued arguments are cut in half, whereas odd ones are enlarged. For example, the orbit of 17 is the sequence: 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 . . . Once a power of 2 is reached, the sequence falls “like a hailstone” to 1 and becomes trapped in a short repetitive cycle (which one?). An unanswered question in mathematics is:

Unanswered Question: Does *every* orbit fall to 1?

That is, does a positive integer exist, that when used as a starting point and iterated with the hailstone function, does *not* ultimately crash down to 1? No one knows, but the intricacies of the sequence have been widely studied (See [Hayes 1984] or numerous sources on the internet, such as [www.cecm.sfu.ca/organics/papers/lagarias/](http://www.cecm.sfu.ca/organics/papers/lagarias/>.)).

Write a program that plots the course of the sequence $y_k = f^{[k]}(y_0)$ versus k . The user gives a starting value y_0 between 1 and 4,000,000,000. (`unsigned long`'s will hold values of this size.) Each value y_k is plotted as the point (k, y_k) . Each plot continues until y_k reaches a value of 1 (if it does...).

Because the hailstone sequence can be very long, and the values of y_k can grow very large, it is essential to scale the values before they are displayed. Recall from Section 2.2 that appropriate values of A , B , C , and D are determined so that when the value (k, y_k) is plotted at screen coordinates:

$$\begin{aligned} sx &= (A * k + B) \\ sy &= (C * y_k + D) \end{aligned} \quad (2.6)$$

the entire sequence fits on the screen.

Note that you don't know how long the sequence will be, nor how large y_k will get, until after the sequence has been generated. A simple solution is to run the sequence invisibly first, keeping track of the largest value y_{Biggest} attained by y_k , as well as the number of iterations k_{Biggest} required for the sequence to reach 1. These values are then used to determine A , B , C , and D . The sequence is then re-run and plotted.

To improve the final plot:

- Draw horizontal and vertical axes;
- Plot the logarithm of y_k rather than y_k itself;

A Curious question: What is the largest y_{Biggest} and what is the largest k_{Biggest} encountered for any hailstone sequence with starting value between 1 and 1,000,000?

Iterating with functions that produce points.

Iterating numbers through some function $f(.)$ is interesting enough, but iterating *points* through a function is even more so, since we can use `drawDot()` to build patterns out of the different points that emerge. So we consider a function $f(p)$ that takes one point $p = (x, y)$ as input and produces another point as its output. Each newly formed point is fed back into the same function again to generate yet another new point, as suggested in Figure 2.46. Here p_{k+1} is used to create the k -th iterate $p_k = f^{[k]}(p_0)$, which is then fed back to produce p_{k+1} , etc.

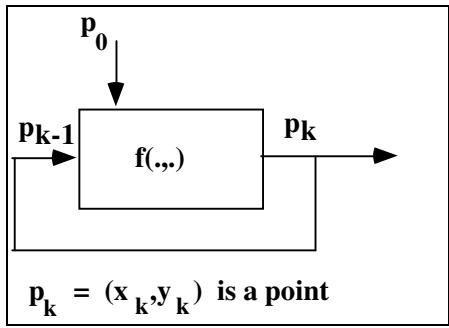


Figure 2.46. Iterated function sequence generator for points.

Once again, we call the sequence of points p_0, p_1, p_2, \dots the **orbit of p_0** .

Aside: The Sierpinski gasket seen as an IFS;

In terms of an IFS the k -th dot, p_k , of the Sierpinski gasket is formed from p_{k-1} using:

$$p_k = (p_{k-1} + T[\text{random}(3)]) / 2$$

where it is understood that the x and y components must be formed separately. Thus the function that is iterated is:

$$f(.) = ((.) + T[\text{random}(3)]) / 2$$

Project 2: The Gingerbread Man.

The “gingerbread man” shown in Figure 2.47 is based on another IFS, and it can be drawn as a dot constellation. It has become a familiar creature in chaos theory [peitgen88, gleick87, schroeder91] because it is a form of “strange attractor”: the successive dots are “attracted” into a region resembling a gingerbread man, with curious hexagonal holes.

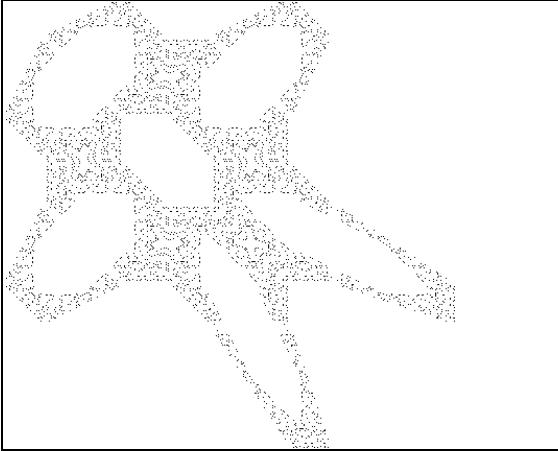


Figure 2.47. A Typical Gingerbread Man.

There is no randomness in the process that generates the gingerbread man; each new point q is formed from the previous point p according to the rules:

$$\begin{aligned} q.x &= M(1 + 2L) - p.y + |p.x - L| \\ q.y &= p.x; \end{aligned} \tag{2.7}$$

where constants M and L are carefully chosen to scale and position the gingerbread man on the display. (The values $M = 40$ and $L = 3$ might be good choices for a 640 by 480 pixel display.)

Write a program that allows the user to choose the starting point for the iterations with the mouse, and draws the dots for the gingerbread man. (If a mouse is unavailable, one good starting point is (115, 121).) Fix suitable values of M and L in the routine, but experiment with other values as well.

You will notice that for a given starting point only a certain number of dots appears before the pattern repeats (so it stops changing). Different starting points give rise to different patterns. Arrange your program so that you can add to the picture by inputting additional starting points with the mouse.

Practice Exercise 2.6.1. A Fixed point on the Gingerbread man. Show that this process has “fixed point”: $((1+L)M, (1+L)M)$. That is, the result of subjecting this point to the process of Equation 2.7 is the same point. (This would be a very uninteresting starting point for generating the gingerbread man!)

2.6.3. Case Study 2.3. The Golden Ratio and Other Jewels.

(Level of Effort: I.) The aspect ratio of a rectangle is an important attribute. Over the centuries, one aspect ratio has been particularly celebrated for its pleasing qualities in works of art: that of the golden rectangle. The golden rectangle is considered as the most pleasing of all rectangles, being neither too narrow nor too squat. It figures in the Greek Parthenon (see Figure 2.48), Leonardo da Vinci's Mona Lisa, Salvador Dali's The Sacrament of the Last Supper, and in much of M. C. Escher's works.

old Fig 3.18 (picture of Greek Parthenon inside golden rectangle)

Figure 2.48. The Greek Parthenon fitting within a Golden Rectangle.

The golden rectangle is based on a fascinating quantity, the golden ratio $\varphi = 1.618033989\ldots$. The value φ appears in a surprising number of places in computer graphics.

Figure 2.49 shows a golden rectangle, with sides of length φ and 1. Its shape has the unique property that if a square is removed from the rectangle, the piece that remains will again be a golden rectangle! What value must φ have to make this work? Note in the figure that the smaller rectangle has height 1 and so to be golden must have width $1/\varphi$. Thus

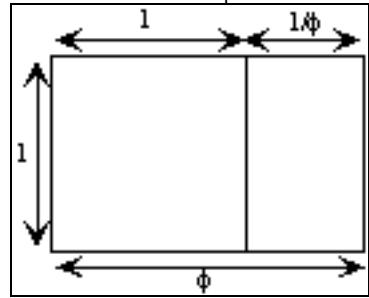


Figure 2.49. The Golden Rectangle.

$$\varphi = 1 + \frac{1}{\varphi} \quad (2.8)$$

which is easily solved to yield

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.618033989\ldots \quad (2.9)$$

This is approximately the aspect ratio of a standard 3-by-5 index card. From Equation 2.8 we see also that if 1 is subtracted from φ the reciprocal of φ is obtained: $1/\varphi = .618033989\ldots$. This is the aspect ratio of a golden rectangle lying on its short end.

The number φ is remarkable mathematically in many ways, two favorites being

$$\varphi = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \dots}}} \quad (2.10)}$$

and

$$\varphi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \dots}}} \quad (2.11)$$

These both are easy to prove (how?) and display a pleasing simplicity in the use of the single digit 1.

The idea that the golden rectangle contains a smaller version of itself suggests a form of “infinite regression” of figures...within figures...within figures... *ad infinitum*. Figure 2.50 demonstrates this. Keep removing squares from each remaining golden rectangle.

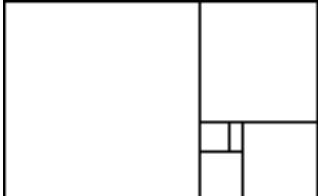


Figure 2.50. Infinite regressions of the golden rectangle.

Write an application that draws the regression of golden rectangles centered in a screen window 600 pixels wide by 400 pixels high. (First determine where and how big the largest golden rectangle is that will fit in this screen window. Your picture should regress down until the smallest rectangle is about one pixel in size.)

There is much more to be said about the golden ratio, and many delights can be found in [Gardner 1961], [Hill 1978], [Huntley 1970], and [Ogilvy 1969]. For instance, in the next chapter we see golden pentagrams, and in Chapter 9 we see that two of the platonic solids, the dodecahedron and the icosahedron, contain three mutually perpendicular golden rectangles!

Practice exercises.

2.6.2. Other golden things. Equation 2.10 shows φ as a repeated square root involving the number 1. What is the value of:

$$W = \sqrt{k + \sqrt{k + \sqrt{k + \sqrt{k + \dots}}}}$$

- 2.6.3. On φ and Golden Rectangles.** a). Show the validity of Equations 2.10 and 2.11.
 b). Find the point at which the two dotted diagonals shown in Figure 2.50 lie, and show that this is the point to which the sequence of golden rectangles converges.
 c). Use Equation 2.8 to derive the relationship:

$$\varphi^2 + \frac{1}{\varphi^2} = 3 \quad (2.12)$$

2.6.4. Golden orbits. The expressions in Equations 2.10 and 2.11 show that the golden ratio φ is the limiting value of applying certain functions again and again. The first function is $f(\cdot) = \sqrt{1 + (\cdot)}$. What is the second function? Viewing these expressions in terms of iterated functions systems, φ is seen to be the value to which orbits converge for some starting values. (The starting value is hidden in the “...” of the expressions.) Explore with a hand calculator what starting values one can use and still have the process converge to φ .

2.6.4. Case Study 2.4. Building and Using Polyline Files.

(Level of Effort: II) Complex pictures such as Figure 2.21 are based on a large collection of polylines. The data for the polylines are typically stored in a file, so the picture can be reconstructed at a later time by reading the polylines into a program and redrawing each line. A reasonable format for such a file was described in Section 2.3.1, and the routine `drawPolyLineFile()` was described that does the drawing.

The file, “dino.dat”, that stores the dinosaur in Figure 2.21 is available as `dino.dat` on the web site for this book (see the preface) Other polyline files are also available there.

- a). Write a program that reads polyline data from a file and draws each polyline in turn. Generate at least one interesting polyline file of your own on a text processor, and use your program to draw it.

b). Extend the program in the previous part to accept some other file formats. For instance, have it accept **differentially coded** x - and y - coordinates. Here the first point (x_1, y_1) of each polyline is encoded as above, but each remaining one (x_i, y_i) is encoded after subtracting the previous point from it: the file contains $(x_i - x_{i-1}, y_i - y_{i-1})$. In many cases there are fewer significant digits in the difference than in the original point values, allowing more compact files. Experiment with this format.

c). Adapt the file format above so that a “color value” is associated with each polyline in the file. This color value appears in the file on the same line as the number of points in the associated polyline. Experiment with several polyline files.

d). Adjust the polyline drawing routine so that it draws a closed polygon when a minus sign precedes the number of points in a polyline, as in:

```

-3           <-- negative: so this is a polygon
0            0           first point in this polygon
35           3           second point in this polygon
57           8           also connect this to the first point of the polygon
5            <-- positive, so leave it open as usual
0            1
12          21
23          34
.. etc

```

The first polyline is drawn as a triangle: its last point is connected to its first..

2.6.5. Case Study 2.5. Stippling of Lines and Polygons.

(Level of Effort: II) You often want a line to be drawn with a dot-dash pattern, or to have a polygon filled with a pattern representing some image. OpenGL provides convenient tools to do this.

Line Stippling.

It is straightforward to define a stipple pattern for use with line drawing. Once the pattern is specified, it is applied to subsequent line drawing as soon as it is enabled with:

```
glEnable(GL_LINE_STIPPLE);
```

and until it is disabled with

```
glDisable(GL_LINE_STIPPLE).
```

The function

```
glLineStipple(GLint factor, GLushort pattern);
```

defines the stipple pattern. The value of **pattern** (which is of type **GLushort** – an unsigned 16-bit quantity) is a sequence of 0’s and 1’s that define which dots along the line are drawn: a 1 prescribes that a dot is drawn; a 0 prescribes that it is not. The pattern is repeated as many times as necessary to draw the desired line. Figure 2.51 shows several examples. The pattern is given compactly in hexadecimal notation. For example, 0xEECC specifies the bit pattern 1110111011001100. The variable **factor** specifies how much to “enlarge” pattern: each bit in the pattern is repeated **factor** times. For instance, the pattern 0xEECC with factor 2 yields: 111111001111100111000011110000. When drawing a stippled polyline, as with `glBegin(GL_LINE_STRIP); glVertex*(); glVertex*(); glVertex*(); ... ; glEnd();`, the pattern continues from the end of one line segment to the beginning of the next, until the `glEnd()`.

pattern	factor	resulting stipple
0x00FF	1
0x00FF	2
0xAAAA	1
0xAAAA	2

Figure 2.51. Example stipple patterns.

Write a program that allows the user to type in a pattern (in hexadecimal notation) and a value for factor, and draws stippled lines laid down with the mouse.

Polygon Stippling.

It is also not difficult to define a stipple pattern for filling a polygon: but there are more details to cope with. After the pattern is specified, it is applied to subsequent polygon filling once it is enabled with `glEnable(GL_POLYGON_STIPPLE)`, until disabled with `glDisable(GL_POLYGON_STIPPLE)`.

The function

```
glPolygonStipple(const GLubyte * mask);
```

attaches the stipple pattern to subsequently drawn polygons, based on a 128 byte array `mask []`. These 128 bytes provide the bits for a bitmask that is 32 bits wide and 32 bits high. The pattern is “tiled” throughout the polygon (which is invoked with the usual `glBegin(GL_POLYGON); glVertex*(); . . . ; glEnd()`). The pattern is specified by an array definition such as:

```
GLubyte mask[] = {0xff, 0xfe, 0x34, ...};
```

The first four bytes prescribe the 32 bits across the bottom row, from left to right; the next 4 bytes give the next row up, etc. Figure 2.52 shows the result of filling a specific polygon with a “fly” pattern specified in the OpenGL “red book” [woo97].

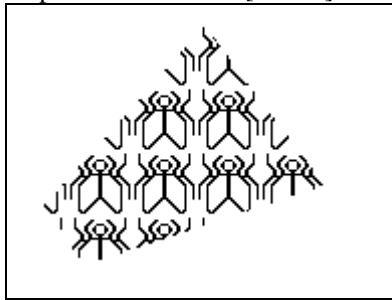


Figure 2.52. An example stippled polygon.

Write a program that defines an interesting stipple pattern for a polygon. It then allows the user to lay down a sequence of convex polygons with the mouse, and each polygon is filled with this pattern.

2.6.6. Case Study 2.6. Polyline Editor.

(Level of Effort: III) Drawing programs often allow one to enter polylines using a mouse, and then to *edit* the polylines until they present the desired picture. Figure 2.53a shows a house in the process of being drawn: the user has just clicked at the position shown, and a line has been drawn from the previous point to the mouse point.

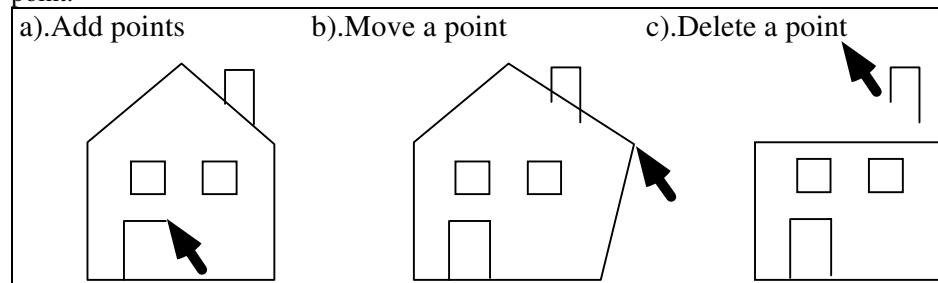


Figure 2.53. Creating and editing polylines.

Figure 2.53b shows the effect of moving a point. The user positions the mouse cursor near some polyline vertex, presses down the mouse button, and “drags” the chosen point to some other location before releasing the mouse button. Upon release, the previous lines connected to this point are erased, and new lines are drawn to it.

Figure 2.53c shows how a point is deleted from a polyline. The user clicks near some polyline vertex, and the two line segments connected to it are erased. Then the two “other” endpoints of the segments just erased are connected with a line segment.

Write and exercise a program that allows the user to enter and edit pictures made up of as many as 60 polylines. The user interacts by pressing keyboard keys and pointing/clicking with the mouse. The functionality of the program should include the “actions”:

- begin ('b') : (create a new polyline)
- delete ('d') : (delete the next point pointed to)
- move ('m') : (drag the point pointed to new location)
- refresh ('r') : (erase the screen and redraw all the polylines)
- quit ('q') : (exit from the program)

A list of polylines can be maintained in an array such as: `GLintPointArray polys[60]`. The verb `begin`, activated by pressing the key ‘b’, permits the user to create a new polyline, which is stored in the first available “slot” in array `polys`. The verb `delete` requires that the program identify which point of which polyline lies closest to the current mouse point. Once identified, the “previous” and “next” vertices in the chosen polyline are found. The two line segments connected to the chosen vertex are erased, and the previous and next vertices are joined with a line segment. The verb `move` finds the vertex closest to the current mouse point, and waits for the user to click the mouse a second time, at which point it moves the vertex to this new point.

What other functions might you want in a polyline editor? Discuss how you might save the array of polylines in a file, and read it in later. Also discuss what a reasonable mechanism might be for inserting a new point inside a polyline.

2.6.7. Case Study 2.7. Building and Running Mazes.

(Level of Effort: III) The task of finding a path through a maze seems forever fascinating (see [Ball and Coxeter 1974]). You can generate an elaborate maze on a computer, and use graphics to watch it be traversed. Figure 2.54 shows a rectangular maze having 100 rows and 150 columns. The goal is to find a path from the opening at the left edge to the opening at the right edge. Although you can traverse it manually by trial and error, it's more interesting to develop an algorithm to do it automatically.

use 1st Ed. Figure 3.38

Figure 2.54. A maze.

Write and exercise a program that a). generates and displays a rectangular maze of R rows and C columns, and b). finds (and displays) the path from start to end. The mazes are generated randomly but must be **proper**; that is, every one of the R -by- C cells is connected by a unique, albeit tortuous, path to every other cell. Think of a maze as a graph, as suggested by Figure 2.55. A node of the graph corresponds to each cell where either a path terminates or two paths meet, and each path is represented by a branch. So a node occurs at every cell for which there is a choice of which way to go next. For instance, when Q is reached there are three choices, whereas at M there are only two. The graph of a proper maze is “acyclic” and has a “tree” structure.

use 1st Ed. Figure 3.39

Figure 2.55. A simple maze and its graph.

How should a maze be represented? One way is to state for each cell whether its north wall is intact and its east wall is intact, suggesting the following data structure:

```
char northWall[R][C], eastWall[R][C];
```

If `northwall[i][j]` is 1, the ij -th cell has a solid upper wall; otherwise the wall is missing. The 0-th row is a phantom row of cells below the maze whose north walls comprise the bottom edge. Similarly, `eastwall[i][0]` specifies where any gaps appear in the left edge of the maze.

Generating a Maze. Start with all walls intact so that the maze is a simple grid of horizontal and vertical lines. The program draws this grid. An invisible “mouse” whose job is to “eat” through walls to connect adjacent cells,” is initially placed in some arbitrarily chosen cell. The mouse checks the four neighbor cells (above, below, left, and right) and for each asks whether the neighbor has all four walls intact. If not, the cell has previously been visited and so is already on some path. The mouse may detect several candidate cells that haven’t been visited: It chooses one randomly and eats through the connecting wall, saving the locations of the other candidates on a stack. The eaten wall is erased, and the mouse repeats the process. When it becomes trapped in a dead end—surrounded by visited cells—it pops an unvisited cell and continues. When the stack is empty, all cells in the maze have been visited. A “start” and “end” cell is then chosen randomly, most likely along some edge of the maze. It is delightful to watch the maze being formed dynamically as the mouse eats through walls. (Question: Might a queue be better than a stack to store candidates? How does this affect the order in which later paths are created?)

Running the Maze. Use a “backtracking” algorithm. At each step, the mouse tries to move in a random direction. If there is no wall, it places its position on a stack and moves to the next cell. The cell that the mouse is in can be drawn with a red dot. When it runs into a dead end, it can change the color of the cell to blue and backtrack by popping the stack. The mouse can even put a wall up to avoid ever trying the dead-end cell again.

Addendum: Proper mazes aren’t too challenging because you can always traverse them using the “shoulder-to-the-wall rule.” Here you trace the maze by rubbing your shoulder along the left-hand wall. At a dead end, sweep around and retrace the path, always maintaining contact with the wall. Because the maze is a “tree,” you will ultimately reach your destination. In fact, there can even be cycles in the graph and you still always find the end, as long as both the start and the end cells are on outer boundaries of the maze (why?). To make things more interesting, place the start and end cells in the interior of the maze and also let the mouse eat some extra walls (maybe randomly 1 in 20 times). In this way, some cycles may be formed that encircle the end cell and defeat the shoulder method.

2.9. For Further reading

Several books provide an introduction to using OpenGL. The “OpenGL Programming Guide” by Woo, Neider, and Davis [woo97] is an excellent source. There is also a wealth of information available on the internet. See, for instance, the OpenGL repository at: <http://www.opengl.org/>, and the complete manual for OpenGL at <http://www.sgi.com/software/opengl/manual.html>.

(For ECE660 - Fall, 1999)

CHAPTER 3. More Drawing Tools.

Computers are useless.
They can only give you answers.

Pablo Picasso
Even if you are on the right track, you'll
get run over if you just sit there.
Will Rogers

Goals of the Chapter

- Introduce viewports and clipping
- Develop the window to viewport transformation
- Develop a classical clipping algorithm
- Create tools to draw in world coordinates
- Develop a C++ class to encapsulate the drawing routines
- Develop ways to select windows and viewports for optimum viewing
- Draw complex pictures using relative drawing, and turtle graphics
- Build figures based on regular polygons and their offspring
- Draw arcs and circles.
- Describe parametrically defined curves and see how to draw them.

Preview.

Section 3.1 introduces world coordinates and the world window. Section 3.2 describes the window to viewport transformation. This transformation simplifies graphics applications by letting the programmer work in a reasonable coordinate system, yet have all pictures mapped as desired to the display surface. The section also discusses how the programmer (and user) choose the window and viewport to achieve the desired drawings. A key property is that the aspect ratios of the window and viewport must agree, or distortion results. Some of the choices can be automated. Section 3.3 develops a classical clipping algorithm that removes any parts of the picture that lie outside the world window.

Section 3.4 builds a useful C++ class called *Canvas* that encapsulates the many details of initialization and variable handling required for a drawing program. Its implementation in an OpenGL environment is developed. A programmer can use the tools in *Canvas* to make complex pictures, confident that the underlying data is protected from inadvertent mishandling.

Section 3.5 develops routines for relative drawing and “turtle graphics” that add handy methods to the programmer’s toolkit. Section 3.6 examines how to draw interesting figures based on regular polygons, and Section 3.7 discusses the drawing of arcs and circles. The chapter ends with several Case Studies, including the development of the *Canvas* class for a non-OpenGL environment, where all the details of clipping and the window to viewport transformation must be explicitly developed.

Section 3.8 describes different representations for curves, and develops the very useful parametric form, that permits straightforward drawing of complex curves. Curves that reside in both 2D space and 3D space are considered.

3.1. Introduction.

It is as interesting and as difficult to say a thing well as to paint it.
Vincent Van Gogh

In Chapter 2 our drawings used the basic coordinate system of the screen window: coordinates that are essentially in pixels, extending from 0 to some value `screenWidth - 1` in *x*, and from 0 to some value `screenHeight - 1` in *y*. This means that we can use only positive values of *x* and *y*, and the values must extend over a large range (several hundred pixels) if we hope to get a drawing of some reasonable size.

In a given problem, however, we may not want to think in terms of pixels. It may be much more natural to think in terms of x varying from, say, -1 to 1, and y varying from -100.0 to 20.0. (Recall how awkward it was to scale and shift values when making the dot plots in Figure 2.16.) Clearly we want to make a separation between the values we use in a program to *describe* the geometrical objects and the size and position of the *pictures* of them on the display.

In this chapter we develop methods that let the programmer/user describe objects in whatever coordinate system best fits the problem at hand, and to have the picture of the object automatically scaled and shifted so that it “comes out right” in the screen window. The space in which objects are described is called **world coordinates**. It is the usual Cartesian xy -coordinate system used in mathematics, based on whatever units are convenient.

We define a rectangular **world window**¹ in these world coordinates. The world window specifies which part of the “world” should be drawn. The understanding is that whatever lies inside the window should be drawn; whatever lies outside should be clipped away and not drawn.

In addition, we define a rectangular **viewport** in the screen window on the screen. A mapping (consisting of scalings and shiftings) between the world window and the viewport is established so that when all the objects in the world are drawn, the parts that lie inside the world window are automatically mapped to the inside of the viewport. So the programmer thinks in terms of “looking through a window” at the objects being drawn, and placing a “snapshot” of whatever is seen in that window into the viewport on the display. This window/viewport approach makes it much easier to do natural things like “zooming in” on a detail in the scene, or “panning around” a scene.

We first develop the mapping part that provides the automatic change of coordinates. Then we see how clipping is done.

3.2. World Windows and Viewports.

We use an example to motivate the use of world windows and viewports. Suppose you want to examine the nature of a certain mathematical function, the “*sinc*” function famous in the signal processing field. It is defined by

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x} \quad (3.1)$$

You want to know how it bends and wiggles as x varies. Suppose you know that as x varies from $-\infty$ to ∞ the value of $\text{sinc}(x)$ varies over much of the range -1 to 1, and that it is particularly interesting for values of x near 0. So you want a plot that is centered at (0, 0), and that shows $\text{sinc}(x)$ for closely spaced x -values between, say, -4.0 to 4.0. Figure 3.1 shows an example plot of the function. It was generated using the simple OpenGL display function (after a suitable world window and viewport were specified, of course):

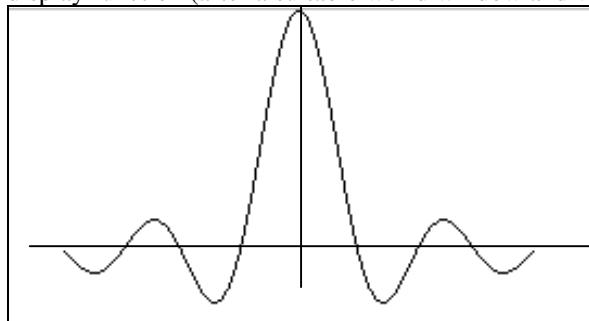


Figure 3.1. A plot of the “sinc” function.

```
void myDisplay(void)
```

¹ As mentioned, the term “window” has a bewildering set of meanings in graphics, which often leads to confusion. We will try to keep the different meanings clear by saying “world window”, “screen window”, etc., when necessary.

```

{
    glBegin(GL_LINE_STRIP);
    for(GLfloat x = -4.0; x < 4.0; x += 0.1)
    {
        GLfloat y = sin(3.14159 * x) / (3.14159 * x);
        glVertex2f(x, y);
    }
    glEnd();
    glFlush();
}

```

Note that the code in these examples operates in a *natural* coordinate system for the problem: x is made to vary in small increments from -4.0 to 4.0 . The key issue here is how the various (x, y) values become scaled and shifted so that the picture appears properly in the screen window.

We accomplish the proper scaling and shifting by setting up a world window and a viewport, and establishing a suitable mapping between them. The window and viewport are both aligned rectangles specified by the programmer. The window resides in world coordinates. The viewport is a portion of the screen window. Figure 3.2 shows an example world window and viewport. The notion is that whatever lies in the world window is scaled and shifted so that it appears in the viewport; the rest is clipped off and not displayed.

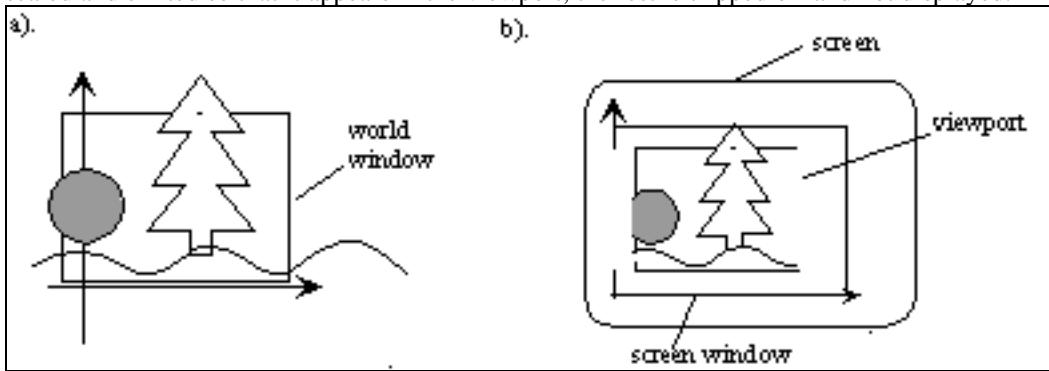


Figure 3.2. A world window and a viewport.

We want to describe not only how to “do it in OpenGL”, which is very easy, but also *how* it is done, to give insight into the low-level algorithms used. We will work with only a 2D version here, but will later see how these ideas extend naturally to 3D “worlds” viewed with a “camera”.

3.2.1. The mapping from the window to the viewport.

Figure 3.3 shows a world window and viewport in more detail. The world window is described by its *left*, *top*, *right*, and *bottom* borders as $W.l$, $W.t$, $W.r$, and $W.b$, respectively². The viewport is described likewise in the coordinate system of the screen window (opened at some place on the screen), by $V.l$, $V.t$, $V.r$, and $V.b$, which are measured in pixels.

²For the sake of brevity we use ‘l’ for ‘left’, ‘t’ for ‘top’, etc. in mathematical formulas.

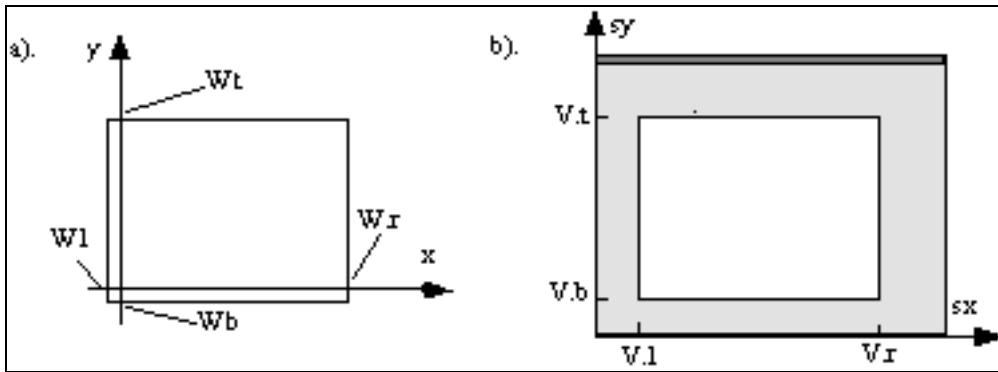


Figure 3.3. Specifying the window and viewport.

The world window can be of any size and shape and in any position, as long as it is an aligned rectangle. Similarly, the viewport can be any aligned rectangle, although it is of course usually chosen to lie entirely within the screen window. Further, the world window and viewport don't have to have the same aspect ratio, although distortion results if their aspect ratios differ. As suggested in Figure 3.4, distortion occurs because the figure in the window must be stretched to fit in the viewport. We shall see later how to set up a viewport with an aspect ratio that always matches that of the window, even when the user resizes the screen window.

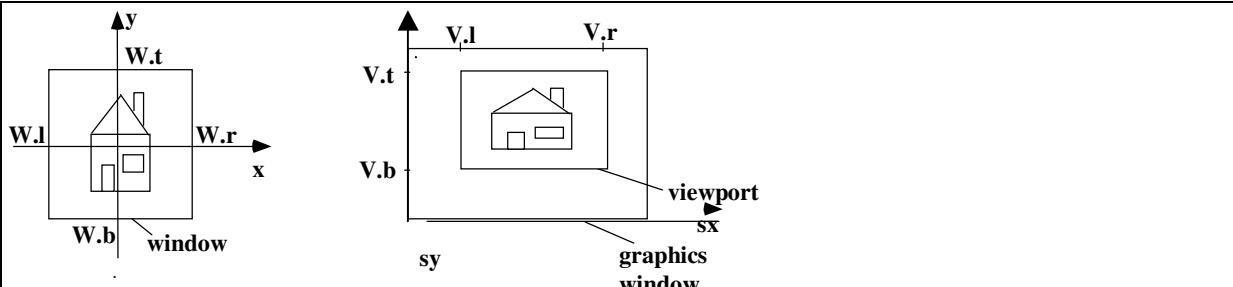


Figure 3.4. A picture mapped from a window to a viewport. Here some distortion is produced.

Given a description of the window and viewport, we derive a **mapping** or **transformation**, called the **window-to-viewport mapping**. This mapping is based on a formula that produces a point (sx, sy) in the screen window coordinates for any given point (x, y) in the world. We want it to be a “proportional” mapping, in the sense that if x is, say, 40% of the way over from the left edge of the window, then sx is 40% of the way over from the left edge of the viewport. Similarly if y is some fraction, f , of the window height from the bottom, sy must be the *same* fraction f up from the bottom of the viewport.

Proportionality forces the mappings to have a *linear* form:

$$\begin{aligned} sx &= A * x + C \\ sy &= B * y + D \end{aligned} \tag{3.2}$$

for some constants A, B, C and D . The constants A and B scale the x and y coordinates, and C and D shift (or *translate*) them.

How can A, B, C , and D be determined? Consider first the mapping for x . As shown in Figure 3.5, proportionality dictates that $(sx - V.l)$ is the same fraction of the total $(V.r - V.l)$ as $(x - W.l)$ is of the total $(W.r - W.l)$, so that

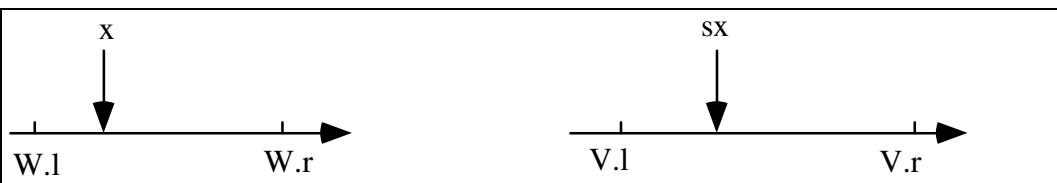


Figure 3.5. Proportionality in mapping x to sx .

$$\frac{sx - V.l}{V.r - V.l} = \frac{x - W.l}{W.r - W.l}$$

or

$$sx = \frac{V.r - V.l}{W.r - W.l} x + (V.l - \frac{V.r - V.l}{W.r - W.l} W.l)$$

Now identifying A as the part that multiplies x and C as the constant part, we obtain:

$$A = \frac{V.r - V.l}{W.r - W.l}, C = V.l - A \cdot W.l$$

Similarly, proportionality in y dictates that

$$\frac{sy - V.b}{V.t - V.b} = \frac{y - W.b}{W.t - W.b}$$

and writing sy as $B y + D$ yields:

$$B = \frac{V.t - V.b}{W.t - W.b}, D = V.b - B \cdot W.b$$

Summarizing, the **window to viewport transformation** is:

$$sx = A x + C, sy = B y + D$$

with (3.3)

$$A = \frac{V.r - V.l}{W.r - W.l}, C = V.l - A \cdot W.l$$

$$B = \frac{V.t - V.b}{W.t - W.b}, D = V.b - B \cdot W.b$$

The mapping can be used with *any* point (x, y) inside or outside the window. Points inside the window map to points inside the viewport, and points outside the window map to points outside the viewport.

(Important!) Carefully check the following properties of this mapping using Equation 3.3:

- a). if x is at the window's left edge: $x = W.l$, then sx is at the viewport's left edge: $sx = V.l$.
- b). if x is at the window's right edge then sx is at the viewport's right edge.
- c). if x is fraction f of the way across the window, then sx is fraction f of the way across the viewport.
- d). if x is outside the window to the left, ($x < W.l$), then sx is outside the viewport to the left ($sx < V.l$), and similarly if x is outside to the right.

Also check similar properties for the mapping from y to sy .

Example 3.2.1: Consider the window and viewport of Figure 3.6. The window has $(W.l, W.r, W.b, W.t) = (0, 2.0, 0, 1.0)$ and the viewport has $(V.l, V.r, V.b, V.t) = (40, 400, 60, 300)$.

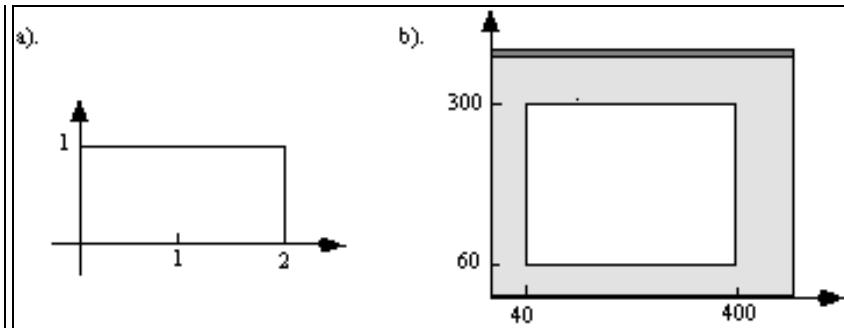


Figure 3.6. An example of a window and viewport.

Using the formulas in Equation 3.2.2 we obtain

$$A = 180, C = 40, \\ B = 240, D = 60$$

Thus for this example, the window to viewport mapping is:

$$sx = 180x + 40 \\ sy = 240y + 60$$

Check that this mapping properly maps various points of interest, such as:

- Each corner of the window is indeed mapped to the corresponding corner of the viewport. For example, (2.0, 1.0) maps to (400, 300).
- The center of the window (1.0, 0.5) maps to the center of the viewport (220, 180).

Practice Exercise 3.2.1. Building the mapping. Find values of A , B , C , and D for the case of a world window $10.0, 10.0, -6.0, 6.0$ and a viewport $(0, 600, 0, 400)$.

Doing it in OpenGL.

OpenGL makes it very easy to use the window to viewport mapping: it automatically passes each vertex it is given (via a `glVertex2*`() command) through a sequence of transformations that carry out the desired mapping. It also automatically clips off parts of objects lying outside the world window. All we need do is to set up these transformations properly, and OpenGL does the rest.

For 2D drawing the world window is set by the function `gluOrtho2D()`, and the viewport is set by the function `glViewport()`. These functions have prototypes:

```
void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
```

which sets the window to have lower left corner (`left, bottom`) and upper right corner (`right, top`), and

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

which sets the viewport to have lower left corner (`x, y`) and upper right corner (`x + width, y + height`).

By default the viewport is the entire screen window: if W and H are the width and height of the screen window, respectively, the default viewport has lower left corner at $(0, 0)$ and upper right corner at (W, H) .

Because OpenGL uses matrices to set up all its transformations, `gluOrtho2D()`³ must be preceded by two “set up” functions `glMatrixMode(GL_PROJECTION)` and `glLoadIdentity()`. (We discuss what is going on behind the scenes here more fully in Chapter 5.)

Thus to establish the window and viewport used in Example 3.2.1 we would use:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 2.0, 0.0, 1.0);      // sets the window
glViewport(40, 60, 360, 240);        // sets the viewport
```

Hereafter every point (x, y) sent to OpenGL using `glVertex2*(x, y)` undergoes the mapping of Equation 3.3, and edges are automatically clipped at the window boundary. (In Chapter 7 we see the details of how this is done in 3D, where it also becomes clear how the 2D version is simply a special case of the 3D version.)

It will make programs more readable if we encapsulate the commands that set the window into a function `setWindow()` as shown in Figure 3.7. We also show `setViewport()` that hides the OpenGL details of `glViewport(..)`. To make it easier to use, its parameters are slightly rearranged to match those of `setWindow()`, so they are both in the order `left, right, bottom, top`.

Note that for convenience we use simply the type `float` for the parameters to `setWindow()`. The parameters `left`, `right`, etc. are automatically cast to type `Gldouble` when they are passed to `gluOrtho2D()`, as specified by this function's prototype. Similarly we use the type `int` for the parameters to `setViewport()`, knowing the arguments to `glViewport()` will be properly cast.

```
----- setWindow -----
void setWindow(float left, float right, float bottom, float top)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(left, right, bottom, top);
}
----- setViewport -----
void setViewport(float left, float right, float bottom, float top)
{
    glViewport(left, bottom, right - left, top - bottom);
}
```

Figure 3.7. Handy functions to set the window and viewport.

It is worthwhile to look back and see what we used for a window and viewport in the early OpenGL programs given in Chapter 2. In Figures 2.10 and 2.17 the programs used:

1). in `main()`:

```
glutInitWindowSize(640, 480);      // set screen window size
```

which set the size of the screen window to 640 by 480. The default viewport was used since no `glViewport()` command was issued; the default viewport is the entire screen window.

2). in `myInit()`:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 640.0, 0.0, 480.0);
```

³ The root “ortho” appears because setting the window this way is actually setting up a so-called “orthographic” projection in 3D, as we'll see in Chapter 7.

This set the world window to the aligned rectangle with corners (0,0) and (640.0, 480.0), just matching the viewport size. So the underlying window to viewport mapping didn't alter anything. This was a reasonable first choice for getting started.

Example 3.2.2: Plotting the sinc function – revisited.

Putting these ingredients together, we can see what it takes to plot the *sinc()* function shape of Figure 3.1. With OpenGL it is just a matter of defining the window and viewport. Figure 3.8 shows the required code, assuming we want to plot the function from closely spaced *x*-values between -4.0 and 4.0, into a viewport with width 640 and height 480. (The window is set to be a little wider than the plot range to leave some cosmetic space around the plot.)

```
void myDisplay(void) // plot the sinc function, using world coordinates
{
    setWindow(-5.0, 5.0, -0.3, 1.0); // set the window
    setViewport(0, 640, 0, 480); // set the viewport
    glBegin(GL_LINE_STRIP);
    for(GLfloat x = -4.0; x < 4.0; x += 0.1) // draw the plot
        glVertex2f(x, sin(3.14159 * x) / (3.14159 * x));
    glEnd();
    glFlush();
}
```

Figure 3.8. Plotting the *sinc* function.

Example 3.2.3: Drawing polylines from a file.

In Chapter 2 we drew the dinosaur shown in Figure 3.9 using the routine `drawPolylineFile("dino.dat")` of Figure 2.22. The polyline data for the figure was stored in a file "dino.dat". The world window and viewport had not yet been introduced, so we just took certain things on faith or by default, and luckily still got a picture of the dinosaur.

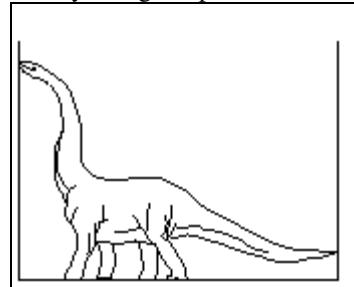


Figure 3.9. The dinosaur inside its world window.

Now we can see why it worked: the world window we used happened to enclose the data for the dinosaur (see Case Study 2.4): All of the polylines in `dino.dat` lie inside a rectangle with corners (0, 0) and (640, 480), so none are clipped with this choice of a window.

Armed with tools for setting the window and viewport, we can take more control of the situation. The next two examples illustrate this.

Example 3.2.4. Tiling the screen window with the dinosaur motif.

To add some interest, we can draw a number of copies of the dinosaur in some pattern. If we lay them side by side to cover the entire screen window it's called **tiling** the screen window. The picture that is copied at different positions is often called a **motif**. Tiling a screen window is easily achieved by using a different viewport for each instance of the motif. Figure 3.10a shows a tiling involving 25 copies of the motif. It was generated using:

a) .	b) .
------	------

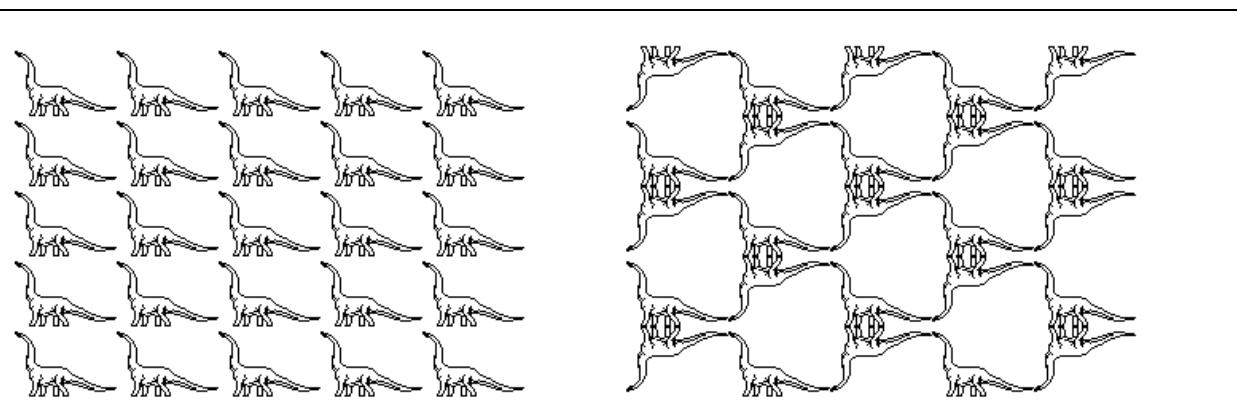


Figure 3.10. Tiling the display with copies of the dinosaur.

```

setWindow(0, 640.0, 0, 480.0);           // set a fixed window
for(int i = 0; i < 5; i++)               // for each column
    for(int j = 0; j < 5; j++)           // for each row
    {
        glViewport(i * 64, j * 44, 64, 44); // set the next viewport
        drawPolylineFile("dino.dat");       // draw it again
    }
}

```

(It's easier to use `glViewport()` here than `setViewport()`. What would the arguments to `setViewport()` be if we chose to use it instead?) Each copy is drawn in a viewport 64 by 48 pixels in size, whose aspect ratio 64/48 matches that of the world window. This draws each dinosaur without any distortion.

Figure 3.10b shows another tiling, but here alternate motifs are flipped upside down to produce an intriguing effect. This was done by flipping the window upside down every other iteration: interchanging the `top` and `bottom` values in `setWindow()`⁴. (Check that this flip of the window properly affects B and D in the window to viewport transformation of Equation 3.3 to flip the picture in the viewport.) Then the preceding double loop was changed to:

```

for(int i = 0; i < 5; i++)
    for(int j = 0; j < 5; j++)
    {
        if((i + j) % 2 == 0)           // if (i + j) is even
            setWindow(0.0, 640.0, 0.0, 480.0); // right side up window
        else
            setWindow(0.0, 640.0, 480.0, 0.0); // upside down window
        glViewport(i * 64, j * 44, 64, 44); // set the next viewport
        drawPolylineFile("dino.dat");       // draw it again
    }
}

```

Example 3.2.5. Clipping parts of a figure.

A picture can also be *clipped* by proper setting of the window. OpenGL automatically clips off parts of objects that lie outside the world window. Figure 3.11a shows a figure consisting of a collection of hexagons of different sizes, each slightly rotated relative to its neighbor. Suppose it is drawn by executing some function `hexSwirl()`. (We see how to write `hexSwirl()` in Section 3.6.) Also shown in part a are two boxes that indicate different choices of a window. Parts b and c show what is drawn if these boxes are used for the world windows. It is important to keep in mind that the *same* entire object is drawn in each case, using the code:

⁴ It might seem easier to invert the viewport, but OpenGL does not permit a viewport to have a negative height.

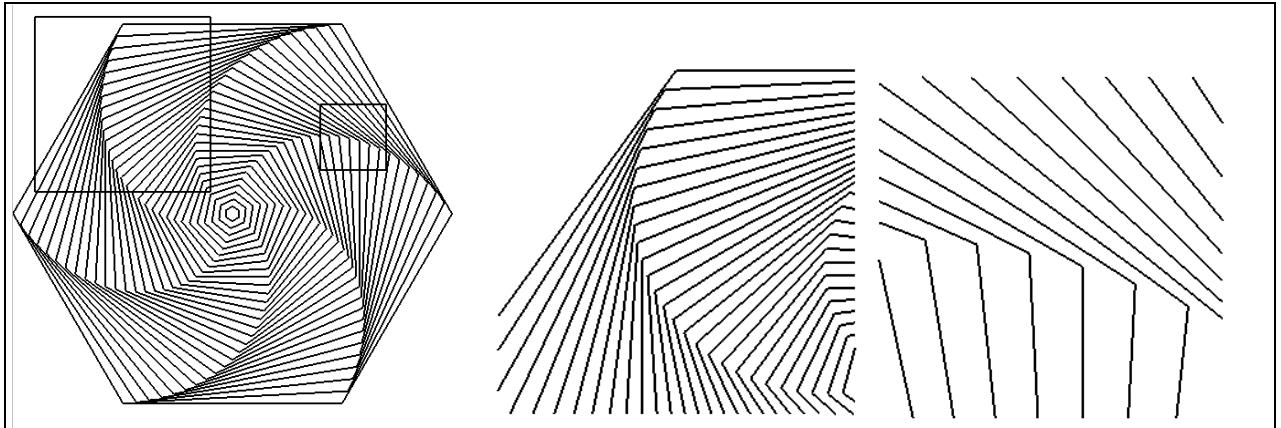


Figure 3.11. Using the window to clip parts of a figure.

```
setWindow(...); // the window is changed for each picture
setViewport(...); // use the same viewport for each picture
hexSwirl(); // the same function is called
```

What is *displayed*, on the other hand, depends on the setting of the window.

Zooming and roaming.

The example in Figure 3.11 points out how changing the window can produce useful effects. Making the window smaller is much like **zooming in** on the object with a camera. Whatever is in the window must be stretched to fit in the fixed viewport, so when the window is made smaller there must be greater enlargement of the portion inside. Similarly making the window larger is equivalent to **zooming out** from the object. (Visualize how the dinosaur would appear if the window were enlarged to twice the size it has in Figure 3.9.) A camera can also **roam** (sometimes called “pan”) around a scene, taking in different parts of it at different times. This is easily accomplished by shifting the window to a new position.

Example 3.2.6. Zooming in on a figure in an animation.

Consider putting together an animation where the camera zooms in on some portion of the hexagons in figure 3.11. We make a series of pictures, often called **frames**, using a slightly smaller window for each one. When the frames are displayed in rapid succession the visual effect is of the camera zooming in on the object.

Figure 3.12 shows a few of the windows used: they are concentric and have a fixed aspect ratio, but their size diminishes for each successive frame. Visualize what is drawn in the viewport for each of these windows.

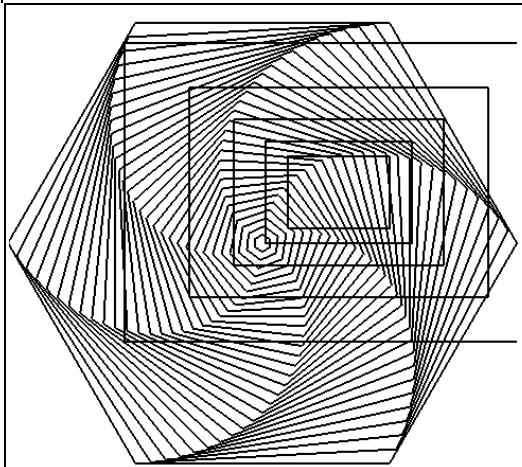


Figure 3.12. Zooming in on the swirl of hexagons. (file: fig3.12.bmp)

A skeleton of the code to achieve this is shown in Figure 3.13. For each new frame the screen is cleared, the window is made smaller (about a fixed center, and with a fixed aspect ratio), and the figure within the window is drawn in a fixed viewport.

```
float cx = 0.3, cy = 0.2; //center of the window
float H, W = 1.2, aspect = 0.7; // window properties
set the viewport
for(int frame = 0; frame < NumFrames; frame++) // for each frame
{
    clear the screen           // erase the previous figure
    W *= 0.7;                 // reduce the window width
    H = W * aspect;           // maintain the same aspect ratio
    setWindow(cx - W, cx + W, cy - H, cy + H); //set the next window
    hexSwirl();               // draw the object
}
```

Figure 3.13. Making an animation.

Achieving a Smooth Animation.

The previous approach isn't completely satisfying, because of the time it takes to draw each new figure. What the user sees is a repetitive cycle of:

- Instantaneous erasure of the current figure;
- A (possibly) slow redraw of the new figure.

The problem is that the user sees the line-by-line creation of the new frame, which can be distracting. What the user would like to see is a repetitive cycle of:

- A steady display of the current figure;
- Instantaneous replacement of the current figure by the *finished* new figure;

The trick is to draw the new figure "somewhere else" while the user stares at the current figure, and then to move the completed new figure instantaneously onto the user's display. OpenGL offers **double-buffering** to accomplish this. Memory is set aside for an extra screen window which is not visible on the actual display, and all drawing is done to this buffer. (The use of such "off-screen memory" is discussed fully in Chapter 10.) The command `glutSwapBuffers()` then causes the image in this buffer to be transferred onto the screen window visible to the user.

To make OpenGL reserve a separate buffer for this, use `GLUT_DOUBLE` rather than `GLUT_SINGLE` in the routine used in `main()` to initialize the display mode:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB); // use double buffering
```

The command `glutSwapBuffers()` would be placed directly after `drawPolylineFile()` in the code of Figure 3.13. Then, even if it takes a substantial period for the polyline to be drawn, at least the image will change abruptly from one figure to the next in the animation, producing a much smoother and visually comfortable effect.

Practice Exercise 3.2.2. Whirling swirls. As another example of clipping and tiling, Figure 3.14a shows the swirl of hexagons with a particular window defined. The window is kept fixed in this example, but the viewport varies with each drawing. Figure 3.14b shows a number of copies of this figure laid side by side to tile the display. Try to pick out the individual swirls. (Some of the swirls have been flipped: which ones?) The result is dazzling to the eye, in part due to the eye's yearning to synthesize many small elements into an overall pattern.

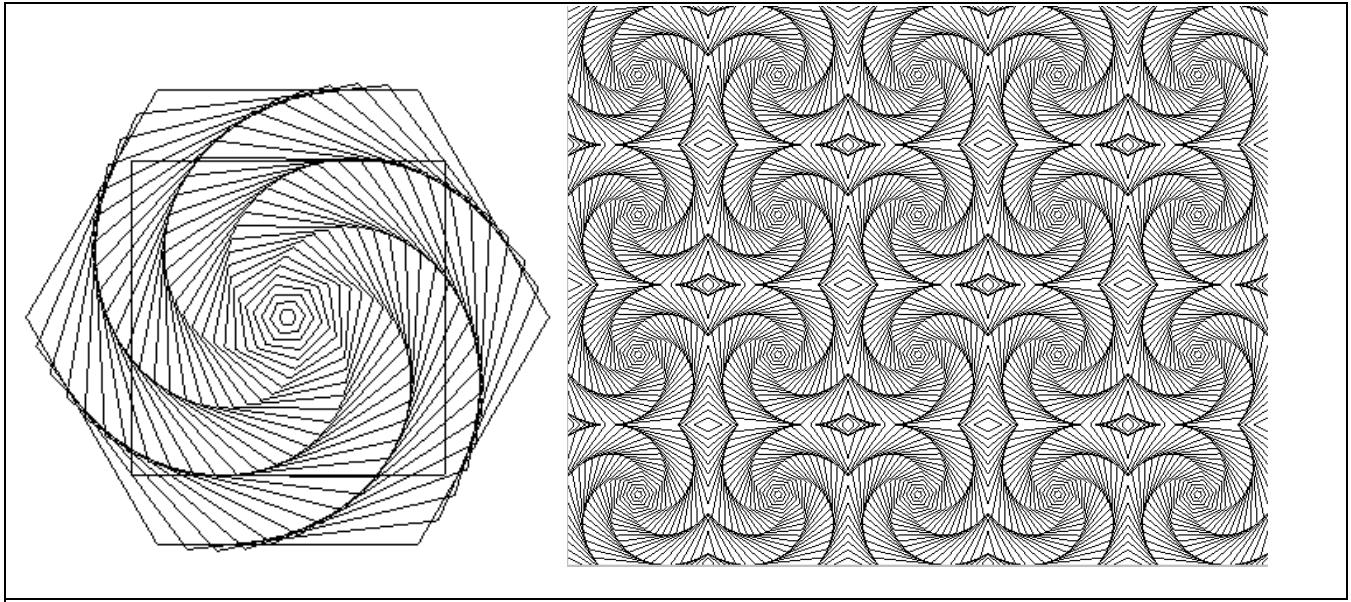


Figure 3.14. a). Whirling hexagons in a fixed window. b). A tiling formed using many viewports.

Except for the flipping, the code shown next creates this pattern. Function `myDisplay()` sets the window once, then draws the clipped swirl again and again in different viewports.

```
void myDisplay(void)
{
    clear the screen
    setWindow(-0.6, 0.6, -0.6, 0.6); // the portion of the swirl to draw
    for(int i = 0; i < 5; i++)          // make a pattern of 5 by 4 copies
        for(int j = 0; j < 4; j++)
    {
        int L = 80; // the amount to shift each viewport
        setViewport(i * L, L + i * L, j * L, L + j * L); // the next viewport
        hexSwirl();
    }
}
```

Type this code into an OpenGL environment, and experiment with the figures it draws. Taking a cue from a previous example, determine how to flip alternating figures upside down.

3.2.2. Setting the Window and Viewport Automatically.

We want to see how to choose the window and viewport in order to produce appropriate pictures of a scene. In some cases the programmer (or possibly the user at run-time) can input the window and viewport specifications to achieve a certain effect; in other cases one or both of them are set up automatically, according to some requirement for the picture. We discuss a few alternatives here.

Setting of the Window.

Often the programmer does not know where or how big the object of interest lies in world coordinates. The object might be stored in a file like the dinosaur earlier, or it might be generated procedurally by some algorithm whose details are not known. In such cases it is convenient to let the application determine a good window to use.

The usual approach is to find a window that includes the entire object: to achieve this the object's extent must be found. The **extent** (or **bounding box**) of an object is the aligned rectangle that just covers it. Figure 3.15 shows a picture made up of several line segments. The extent of the figure, shown as a dashed line, is (*left, right, bottom, top*) = (0.36, 3.44, -0.51, 1.75).

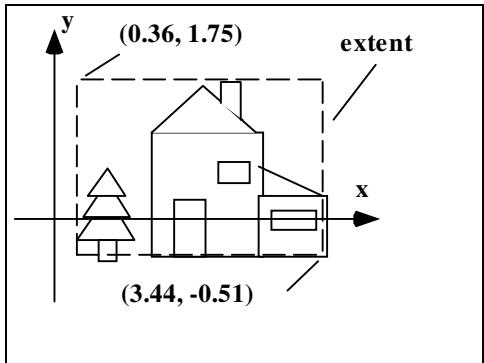


Figure 3.15. Using the Extent as the Window.

How can the extent be computed for a given object? If all the endpoints of its lines are stored in an array $pt[i]$, for $i = 0, 2, \dots, n-1$ the extent can be computed by finding the extreme values of the x - and y - coordinates in this array. For instance, the left side of the extent is the smallest of the values $pt[i].x$. Once the extent is known, the window can be made identical to it.

If, on the other hand, an object is procedurally defined, there may be no way to determine its extent ahead of time. In such a case the routine may have to be run twice:

- Pass 1:** Execute the drawing routine, but do no actual drawing; just compute the extent. Then set the window.
Pass 2: Execute the drawing routine again. Do the actual drawing.

Automatic setting of the viewport to Preserve Aspect Ratio.

Suppose you want to draw the largest undistorted version of a figure that will fit in the screen window. For this you need to specify a viewport that has the same aspect ratio as the world window. A common wish is to find the *largest* such viewport that will fit inside the screen window on the display.

Suppose the aspect ratio of the world window is known to be R , and the screen window has width W and height H . There are two distinct situations: the world window may have a larger aspect ratio than the screen window ($R > W/H$), or it may have a smaller aspect ratio ($R < W/H$). The two situations are shown in Figure 3.16.

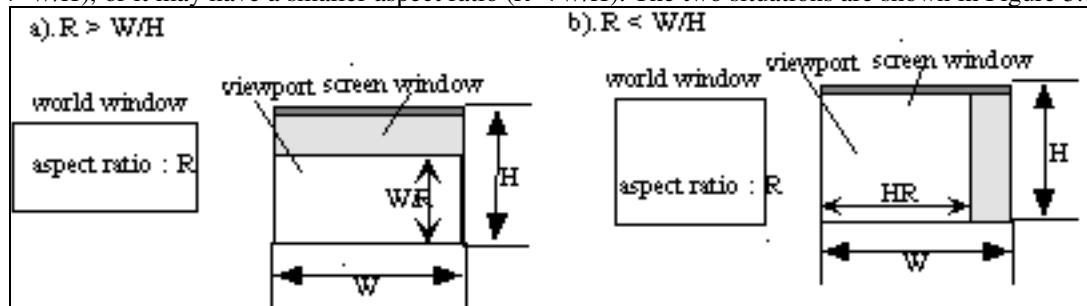


Figure 3.16. Possible aspect ratios for the world and screen windows.

Case a): $R > W/H$. Here the world window is short and stout relative to the screen window, so the viewport with a matching aspect ratio R will extend fully across the screen window, but will leave some unused space above or below. At its largest, therefore, it will have width W and height W/R , so the viewport is set using (check that this viewport does indeed have aspect ratio R):

```
setViewport(0, 0, 0, W/R);
```

Case b): $R < W/H$. Here the world window is tall and narrow relative to the screen window, so the viewport of matching aspect ratio R will reach from the top to the bottom of the screen window, but will leave some unused space to the left or right. At its largest it will have height H but width HR , so the viewport is set using:

```
setViewport(0, H * R, 0, H);
```

Example 3.2.7: A tall window. Suppose the window has aspect ratio $R = 1.6$ and the screen window has $H = 200$ and $W = 360$, and hence $W/H = 1.8$. Therefore Case b) applies, and the viewport is set to have a height of 200 pixels and a width of 320 pixels.

Example 3.2.8: A short window. Suppose $R = 2$ and the screen window is the same as in the example above. Then case a) applies, and the viewport is set to have a height of 180 pixels and a width of 360 pixels.

Resizing the screen window, and the resize event.

In a windows-based system the user can resize the screen window at run-time, typically by dragging one of its corners with the mouse. This action generates a **resize** event that the system can respond to. There is a function in the OpenGL utility toolkit, `glutReshape()` that specifies a function to be called whenever this event occurs:

```
glutReshape(myReshape); //specifies the function called on a resize event
```

(This statement appears in `main()` along with the other calls that specify callback functions.) The registered function is also called when the window is first opened. It must have the prototype:

```
void myReshape(GLsizei W, GLsizei H);
```

When it is executed the system automatically passes it the new width and height of the screen window, which it can use in its calculations. (`GLsizei` is a 32 bit integer – see Figure 2.7.)

What should `myReshape()` do? If the user makes the screen window bigger the previous viewport could still be used (why?), but it might be desired to increase the viewport to take advantage of the larger window size. If the user makes the screen window smaller, crossing any of the boundaries of the viewport, you almost certainly want to recompute a new viewport.

Making a matched viewport.

One common approach is to find a new viewport that a) fits in the new screen window, and b) has the same aspect ratio as the world window. “Matching” the aspect ratios of the viewport and world window in this way will prevent distortion in the new picture. Figure 3.17 shows a version of `myReshape()` that does this: it finds the largest “matching” viewport (matching the aspect ratio, R , of the window), that will fit in the new screen window. The routine obtains the (new) screen window width and height through its arguments. Its code is a simple embodiment of the result in Figure 3.16.

```
void myReshape(GLsizei W, GLsizei H)
{
    if(R > W/H) // use (global) window aspect ratio
        setViewport(0, W, 0, W/R);
    else
        setViewport(0, H * R, 0, H);
}
```

Figure 3.17. Using a reshape function to set the largest matching viewport upon a resize event.

Practice Exercises.

3.2.3. Find the bounding box for a polyline. Write a routine that computes the extent of the polyline stored in the array of points `pt[i]`, for $i = 0, 2, \dots, n-1$.

3.2.4. Matching the Viewport. Find the matching viewport for a window with aspect ratio .75 when the screen window has width 640 and height 480.

3.2.5. Centering the viewport. (Don't skip this one!) Adjust the `myReshape()` routine above so that the viewport, rather than lying in the lower left corner of the display, is centered both vertically and horizontally in the screen window.

3.2.6. How to squash a house. Choose a window and a viewport so that a square is squashed to half its proper height. What are the coefficients A , B , C , and D in this case?

3.2.7. Calculation of the mapping. Find the coefficients A , B , C , and D of the window to viewport mapping for a window given by $(-600, 235, -500, 125)$ and a viewport $(20, 140, 30, 260)$. Does distortion occur for figures drawn in the world? Change the right border of the viewport so that distortion will not occur.

3.3. Clipping Lines.

Clipping is a fundamental task in graphics, needed to keep those parts of an object that lie outside a given region from being drawn. A large number of clipping algorithms have been developed. In an OpenGL environment each object is automatically clipped to the world window using a particular algorithm (which we examine in detail in Chapter 7 for both 2D and 3D objects.)

Because OpenGL clips for you there may be a temptation to skip a study of the clipping process. But the ideas that are used to develop a clipper are basic and arise in diverse situations; we will see a variety of approaches to clipping in later chapters. And it's useful to know how to pull together a clipper as needed when a tool like OpenGL is not being used.

We develop a clipping algorithm here that clips off outlying parts of each line segment presented to it. This algorithm can be incorporated in a line-drawing routine if we do not have the benefit of the clipping performed by OpenGL. An implementation of a class that draws clipped lines is developed in Case Study 3.3.

3.3.1. Clipping a Line.

In this section we describe a classic line-clipping algorithm, the Cohen-Sutherland clipper, that computes which part (if any) of a line segment with endpoints p_1 and p_2 lies inside the world window, and reports back the endpoints of that part.

We'll develop the routine `clipSegment(p1, p2, window)` that takes two 2D points and an aligned rectangle. It clips the line segment defined by endpoints p_1 and p_2 to the window boundaries. If any portion of the line remains within the window, the new endpoints are placed in p_1 and p_2 , and 1 is returned (indicating some part of the segment is visible). If the line is completely clipped out, 0 is returned (no part is visible).

Figure 3.18 shows a typical situation covering some of the many possible actions for a clipper. `clipSegment()` does one of four things to each line segment:

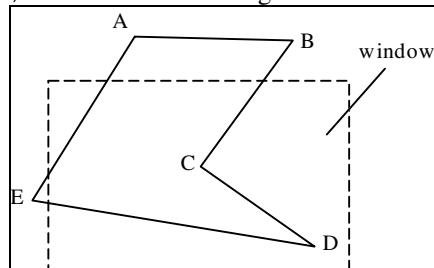


Figure 3.18. Clipping Lines at window boundaries.

- If the entire line lies within the window, (e.g. segment CD): it returns 1.
- If the entire line lies outside the window, (e.g. segment AB): it returns 0.
- If one endpoint is inside the window and one is outside (e.g. segment ED): the function clips the portion of the segment that lies outside the window and returns 1.
- If both endpoints are outside the window, but a portion of the segment passes through it, (e.g. segment AE): it clips both ends and returns 1.

There are many possible arrangements of a segment with respect to the window. The segment can lie to the left, to the right, above, or below the window; it can cut through any one (or two) window edges, and so on. We therefore need an organized and efficient approach that identifies the prevailing situation and computes new endpoints for the clipped segment. Efficiency is important because a typical picture contains thousands of line

segments, and each must be clipped against the window. The Cohen–Sutherland algorithm provides a rapid divide-and-conquer attack on the problem. Other clipping methods are discussed beginning in Chapter 4.

3.3.2. The Cohen-Sutherland Clipping Algorithm

The Cohen–Sutherland algorithm quickly detects and dispenses with two common cases, called “trivial accept” and “trivial reject”. As shown in Figure 3.19, both endpoints of segment

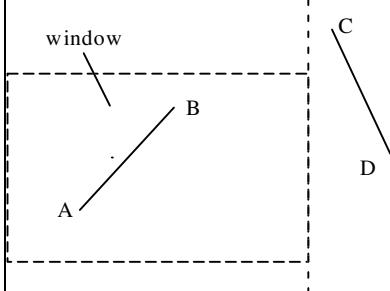


Figure 3.19. Trivial acceptance or rejection of a line segment.

AB lie within window W , and so the whole segment AB must lie inside. Therefore AB can be “*trivially accepted*”: it needs no clipping. This situation occurs frequently when a large window is used that encompasses most of the line segments. On the other hand, both endpoints C and D lie entirely to one side of W , and so segment CD must lie entirely outside. It is *trivially rejected*, and nothing is drawn. This situation arises frequently when a small window is used with a dense picture that has many segments outside the window.

Testing for a trivial accept or trivial reject.

We want a fast way to detect whether a line segment can be trivially accepted or rejected. To facilitate this, an “inside-outside code word” is computed for each endpoint of the segment. Figure 3.20 shows how it is done. Point P is to the left and above the window W . These two facts are recorded in a code word for P : a T (for TRUE) is seen in the field for “is to the left of”, and “is above”. An F (for FALSE) is seen in the other two fields, “is to the right of”, and “is below”.

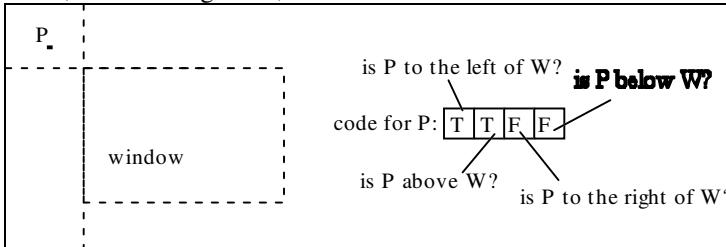


Figure 3.20. Encoding how point P is disposed with respect to the window.

For example, if P is inside the window its code is $FFFF$; if P is below but neither to the left nor right its code is $FFFT$. Figure 3.21 shows the nine different regions possible, each with its code.

TTFF	FTFF	FTTF
FFFF		FFTF
TFFT	FFFT	FFTT

Figure 3.21. Inside-outside codes for a point.

We form a code word for each of the endpoints of the line segment being tested. The conditions of trivial accept and reject are easily related to these code words:

- Trivial accept: Both code words are FFFF;
- Trivial reject: the code words have an F in the *same* position: both points are to the left of the window, or both are above, etc.

The actual formation of the code words and tests can be implemented very efficiently using the bit manipulation capabilities of C/C++, as we describe in Case Study 3.3.

Chopping when there is neither trivial accept nor reject.

The Cohen-Sutherland algorithm uses a divide-and-conquer strategy. If the segment can neither be trivially accepted nor rejected it is broken into two parts at one of the window boundaries. One part lies outside the window and is discarded. The other part is potentially visible, so the entire process is repeated for this segment against another of the four window boundaries. This gives rise to the strategy:

```
do{
    form the code words for p1 and p2
    if (trivial accept) return 1;
    if (trivial reject) return 0;
    chop the line at the "next" window border; discard the "outside" part;
} while(1);
```

The algorithm terminates after at most four times through the loop, since at each iteration we retain only the portion of the segment that has “survived” testing against previous window boundaries, and there are only four such boundaries. After at most four iterations trivial acceptance or rejection is assured.

How is the chopping at each boundary done? Figure 3.22 shows an example involving the right edge of the window.

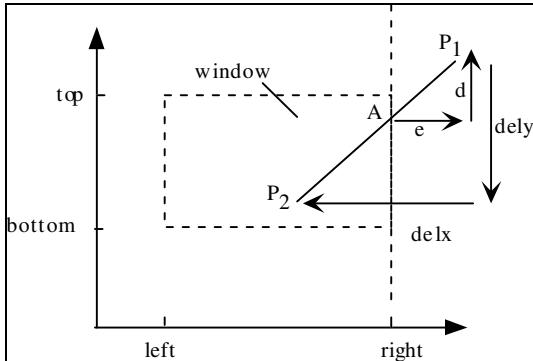


Figure 3.22. Clipping a segment against an edge.

Point A must be computed. Its *x*-coordinate is clearly *w.right*, the right edge position of the window. Its *y*-coordinate requires adjusting *p1.y* by the amount *d* shown in the figure. But by similar triangles

$$\frac{d}{dely} = \frac{e}{delx}$$

where *e* is *p1.x - w.right* and:

$$\begin{aligned} delx &= p2.x - p1.x; \\ dely &= p2.y - p1.y; \end{aligned} \tag{3.4}$$

are the differences between the coordinates of the two endpoints. Thus *d* is easily determined, and the new *p1.y* is found by adding an increment to the old as

$$p1.y += (w.right - p1.x) * dely / delx \tag{3.5}$$

Similar reasoning is used for clipping against the other three edges of window.

In some of the calculations the term `dely/delx` occurs, and in others it is `delx/dely`. One must always be concerned about dividing by zero, and in fact `delx` is zero for a vertical line, and `dely` is 0 for a horizontal line. But as discussed in the exercises the perilous lines of code are never executed when a denominator is zero, so division by zero will not occur.

These ideas are collected in the routine `clipSegment()` shown in Figure 3.23. The endpoints of the segment are passed by reference, since changes made to the endpoints by `clipSegment()` must be visible in the calling routine. (The type `Point2` holds a 2D point, and the type `RealRect` holds an aligned rectangle. Both types are described fully in Section 3.4.)

```
int clipSegment(Point2& p1, Point2& p2, RealRect W)
{
    do {
        if (trivial accept) return 1; // some portion survives
        if (trivial reject) return 0; // no portion survives

        if (p1 is outside)
        {
            if (p1 is to the left) chop against the left edge
            else if (p1 is to the right) chop against the right edge
            else if (p1 is below) chop against the bottom edge
            else if (p1 is above) chop against the top edge
        }
        else                                // p2 is outside
        {
            if (p2 is to the left) chop against the left edge
            else if (p2 is to the right) chop against the right edge
            else if (p2 is below) chop against the bottom edge
            else if (p2 is above) chop against the top edge
        }
    } while(1);
}
```

Figure 3.23. The Cohen-Sutherland line clipper (pseudocode).

Each time through the `do` loop the code for each endpoint is recomputed and tested. When trivial acceptance and rejection fail, the algorithm tests whether `p1` is outside, and if so it clips that end of the segment to a window boundary. If `p1` is inside then `p2` must be outside (why?) so `p2` is clipped to a window boundary.

This version of the algorithm clips in the order left, then right, then bottom, and then top. The choice of order is immaterial if segments are equally likely to lie anywhere in the world. A situation that requires all four clips is shown in Figure 3.24. The first clip

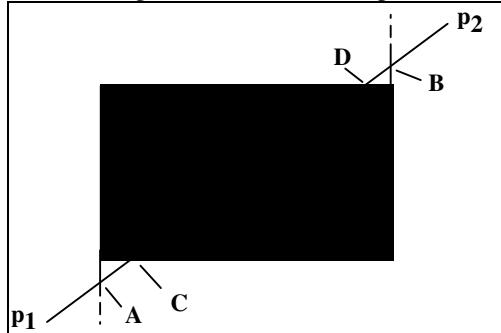


Figure 3.24. A segment that requires four clips.

changes p_1 to A ; the second alters p_2 to B ; the third finds p_1 still outside and below and so changes A to C ; and the last changes p_2 to D . For any choice of ordering for the chopping tests, there will always be a situation in which all four clips are necessary.

Clipping is a fundamental operation that has received a lot of attention over the years. Several other approaches have been developed. We examine some of them in the Case Studies at the end of this chapter, and in Chapter 4.

3.3.2. Hand Simulation of `clipSegment()`.

Go through the clipping routine by hand for the case of a window given by $(left, right, bottom, top) = (30, 220, 50, 240)$ and the following line segments:

- 1). $p_1=(40,140), p_2=(100,200);$
- 2). $p_1=(10,270), p_2=(300,0);$
- 3). $p_1=(20,10), p_2=(20,200);$
- 4). $p_1=(0,0), p_2=(250,250);$

In each case determine the endpoints of the clipped segment, and for a visual check, sketch the situation on graph paper.

3.4. Developing the Canvas Class.

“One must not always think that feeling is everything.
Art is nothing without form.”
Gustave Flaubert

There is significant freedom in working in world coordinates, and having primitives be clipped and properly mapped from the window to the viewport. But this freedom must be managed properly. There are so many interacting ingredients (points, rectangles, mappings, etc.) in the soup now we should encapsulate them and restrict how the application programmer accesses them, to avoid subtle bugs. We should also insure that the various ingredients are properly initialized.

It is natural to use classes and the data hiding they offer. So we develop a class called *Canvas* that provides a handy drawing canvas on which to draw the lines, polygons, etc. of interest. It provides simple methods to create the desired screen window and to establish a world window and viewport, and it insures that the window to viewport mapping is well defined. It also offers the routines `moveTo()` and `lineTo()` that many programmers find congenial, as well as the useful “turtle graphics” routines we develop later in the chapter.

There are many ways to define the *Canvas* class: the choice presented here should be considered only as a starting point for your own version. We implement the class in this section using OpenGL, exploiting all of the operations OpenGL does automatically (such as clipping). But in Case Study 3.4 we describe an entirely different implementation (based on Turbo C++ in a DOS environment), for which we have to supply all of the tools. In particular an implementation of the Cohen Sutherland clipper is used.

3.4.1. Some useful Supporting Classes.

It will be convenient to have some common data types available for use with *Canvas* and other classes. We define them here as classes⁵, and show simple constructors and other functions for handling objects of each type. Some of the classes also have a `draw` function to make it easy to draw instances of the class. Other member functions (methods) will be added later as the need arises. Some of the methods are implemented directly in the class definitions; the implementation of others is requested in the exercises, and only the declaration of the method is given.

class Point2: A point having real coordinates.

The first supporting class embodies a single point expressed with floating point coordinates. It is shown with two constructors, the function `set()` to set the coordinate values, and two functions to retrieve the individual coordinate values.

```
class Point2
{
```

⁵ Students preferring to write in C can define similar types using `struct`'s.

```

public:
    Point2() {x = y = 0.0f;}      // constructor1
    Point2(float xx, float yy) {x = xx; y = yy;} // constructor2
    void set(float xx, float yy) {x = xx; y = yy;}
    float getX() {return x;}
    float getY() {return y;}
    void draw(void) { glBegin(GL_POINTS); // draw this point
                      glVertex2f((GLfloat)x, (GLfloat)y);
                      glEnd();}

private:
    float x, y;
};

```

Note that values of `x` and `y` are cast to the type `GLfloat` when `glVertex2f()` is called. This is most likely unnecessary since the type `GLfloat` is defined on most systems as `float` anyway.

class IntRect: An aligned rectangle with integer coordinates.

To describe a viewport we need an aligned rectangle having integer coordinates. The class `IntRect` provides this.

```

class IntRect
{
public:
    IntRect() {l = 0; r = 100; b = 0; t = 100;}// constructors
    IntRect(int left, int right, int bottom, int top)
        {l = left; r = right; b = bottom; t = top;}
    void set(int left, int right, int bottom, int top)
        {l = left; r = right; b = bottom; t = top;}
    void draw(void); // draw this rectangle using OpenGL
private:
    int l, r, b, t;
};

```

class RealRect: An aligned rectangle with real coordinates.

A world window requires the use of an aligned rectangle having real values for its boundary position . (This class is so similar to `IntRect` some programmers would use templates to define a class that could hold either integer or real coordinates.)

```

class RealRect
{
    same as intRect except use float instead of int
};

```

Practice Exercise 3.4.1. Implementing the classes. Flesh out these classes by adding other functions you think would be useful, and by implementing the functions, such as `draw()` for `intRect`, that have only been declared above.

3.4.2. Declaration of Class Canvas.

We declare the interface for `Canvas` in `Canvas.h` as shown in Figure 3.25. Its data members include the current position, a window, a viewport, and the window to viewport mapping.

```

class Canvas {
public:
    Canvas(int width, int height, char* windowTitle); // constructor
    void setWindow(float l, float r, float b, float t);
    void setViewport(int l, int r, int b, int t);
    IntRect getViewport(void); // divulge the viewport data
    RealRect getWindow(void); // divulge the window data
};

```

```
    float getWindowAspectRatio(void);
    void clearScreen();
    void setBackgroundColor(float r, float g, float b);
    void setColor(float r, float g, float b);
    void lineTo(float x, float y);
    void lineTo(Point2 p);
    void moveTo(float x, float y);
    void moveTo(Point2 p);
    others later
private:
    Point2 CP;           // current position in the world
    IntRect viewport;   // the current window
    RealRect window;    // the current viewport
    others later
};
```

Figure 3.25. The header file `Canvas.h`.

The *Canvas* constructor takes the width and height of the screen window along with the title string for the window. As we show below it creates the screen window desired, performing all of the appropriate initializations. *Canvas* also includes functions to set and return the dimensions of the window and the viewport, and to control the drawing and background color. (There is no explicit mention of data for the window to viewport mapping in this version, as this mapping is managed “silently” by OpenGL. In Case Study 3.4 we add members to hold the mapping for an environment that requires it.). Other functions shown are versions of *lineTo()* and *moveTo()* that do the actual drawing (in world coordinates, of course). We add “relative drawing tools” in the next section.

Figure 3.26 shows how the *Canvas* class might typically be used in an application. A single global object *cvs* is created, which initializes and opens the desired screen window. It is made global so that callback functions such as *display()* can “see” it. (We cannot pass *cvs* as a parameter to such functions, as their prototypes are fixed by the rules of the OpenGL utility toolkit.) The *display()* function here sets the window and viewport, and then draws a line, using *Canvas* member functions. Then a rectangle is created and drawn using its own member function.

Figure 3.26. Typical usage of the *Canvas* class.

The main() routine doesn't do any initialization: this has all been done in the *Canvas* constructor. The routine main() simply sets the drawing and background colors, registers function `display()`, and enters the main event loop. (Could these OpenGL-specific functions also be "buried" in *Canvas* member functions?) Note that this application makes almost no OpenGL-specific calls, so it could easily be ported to another environment (which used a different implementation of *Canvas*, of course).

3.4.3. Implementation of Class Canvas.

We show next some details of an implementation of this class when OpenGL is available. (Case Study 3.4 discusses an alternate implementation.) The constructor, shown in Figure 3.27, passes the desired width and height (in pixels) to `glutInitWindowSize()`, and the desired title string to `glutCreateWindow()`. Some fussing must be done to pass `glutInit()` the arguments it needs, even though they aren't used here. (Normally `main()` passes `glutInit()` the command line arguments, as we saw earlier. This can't be done here since we will use a global `Canvas` object, `cvs`, which is constructed before `main()` is called.)

```
//<<<<<<<< Canvas constructor >>>>>>>>>>
Canvas:: Canvas(int width, int height, char* windowTitle)
{
    char* argv[1];           // dummy argument list for glutInit()
    char dummyString[8];
    argv[0] = dummyString; // hook up the pointer
    int argc = 1;           // to satisfy glutInit()

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(width, height);
    glutInitWindowPosition(20, 20);
    glutCreateWindow(windowTitle); // open the screen window
    setWindow(0, (float)width, 0, (float)height); //default world wi
    setViewport(0, width, 0, height); // default viewport
    CP.set(0.0f, 0.0f);           // initialize the CP to (0, 0)
}
```

Figure 3.27. The constructor for *Canvas* – OpenGL version.

Figure 3.28 shows the implementation of some of the remaining *Canvas* member functions. (Others are requested in the exercises.) Function `moveTo()` simply updates the current position; `lineTo()` sends the *CP* as the first vertex, and the new point (x, y) as the second vertex. Note that we don't need to use the window to viewport mapping explicitly here, since OpenGL automatically applies it. The function `setWindow()` passes its arguments to `gluOrtho2D()` – after properly casting their types – and loads them into *Canvas*'s window.

```
//<<<<<<<<<<<<<<< moveTo >>>>>>>>>>>>>
void Canvas:: moveTo(float x, float y)
{
    CP.set(x, y);
}
//<<<<<<<<<<<<<< lineTo >>>>>>>>>>>>>>>>
void Canvas:: lineTo(float x, float y)
{
    glBegin(GL_LINES);
        glVertex2f((GLfloat)CP.x, (GLfloat)CP.y);
        glVertex2f((GLfloat)x, (GLfloat)y);           // draw the line
    glEnd();
    CP.set(x, y); // update the CP
    glFlush();
}
//<<<<<<<<<<<<<< set Window >>>>>>>>>>>>>>
void Canvas:: setWindow(float l, float r, float b, float t)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
```

```

    gluOrtho2D((GLdouble)l, (GLdouble)r, (GLdouble)b, (GLdouble)t);
    window.set(l, r, b, t);
}

```

Figure 3.28. Implementation of some *Canvas* member functions.

Practice Exercises.

3.4.2. Flesh out each of the member functions:

- void setViewport(int l, int r, int b, int t);
- IntRect getViewport(void);
- RealRect getWindow(void);
- void clearScreen(void);
- void setBackgroundColor(float r, float g, float b);
- void setColor(float r, float g, float b);
- void lineTo(Point2 p);
- void moveTo(Point2 p);
- float getWindowAspectRatio(void)

3.4.3. Using *Canvas* for a simulation: Fibonacci numbers. The growth in the size of a rabbit population is said to be modeled by the following equation [gardner61]:

$$y_k = y_{k-1} + y_{k-2}$$

where y_k is the number of bunnies at the k -th generation. This model says that the number in this generation is the sum of the numbers in the previous two generations. The initial populations are $y_0 = 1$ and $y_1 = 1$. Successive values of y_k are formed by substituting earlier values, and the resulting sequence is the well-known **Fibonacci sequence**; 1, 1, 2, 3, 5, 8, 13. . . . A plot of the sequence y_k versus k reveals the nature of this growth pattern. Use the *Canvas* class to write a program that draws such a plot for a sequence of length N . Adjust the size of the plot appropriately for different N . (The sequence grows very rapidly, so you may instead wish to plot the logarithm of y_k versus k .) Also plot the sequence of ratios $p_k = y_k / y_{k-1}$ and watch how quickly this ratio converges to the golden ratio.

3.4.4. Another Simulation: sinusoidal sequences. The following difference equation generates a sinusoidal sequence:

$$y_k = a \cdot y_{k-1} - y_{k-2} \quad \text{for } k = 1, 2, \dots$$

where a is a constant between 0 and 2; y_k is 0 for $k < 0$; and $y_0 = 1$ (see [oppenheim83]). In general, one cycle consists of S points if we set $a = 2 \cdot \cos(2\pi/S)$. A good picture results with $S = 40$. Write a routine that draws sequences generated in this fashion, and test it for various values of S .

3.5. Relative Drawing.

If we add just a few more drawing tools to our tool bag (which is the emerging class *Canvas*) certain drawing tasks become much simpler. It is often convenient to have drawing take place at the current position (*CP*), and to describe positions relative to the *CP*. We develop functions, therefore, whose parameters specify *changes* in position: the programmer specifies how far to go along each coordinate to the next desired point.

3.5.1. Developing *moveRel()* and *lineRel()*.

Two new routines are *moveRel()* and *lineRel()*. The function *moveRel()* is easy: it just “moves” the *CP* through the displacement (dx , dy). The function *lineRel*(*float dx*, *float dy*) does this too, but it first draws a line from the old *CP* to the new one. Both functions are shown in Figure 3.29.

```

void Canvas :: moveRel(float dx, float dy)
{
    CP.set(CP.x + dx, CP.y + dy);
}

```

```

}

void Canvas :: lineRel(float dx, float dy)
{
    float x = CP.x + dx, y = CP.y + dy;
    lineTo(x, y);
    CP.set(x, y);
}

```

Figure 3.29. The functions `moveRel()` and `lineRel()`.

Example 3.5.1. An arrow marker. Markers of different shapes can be placed at various points in a drawing to add emphasis. Figure 3.30 shows pentagram markers used to highlight the data points in a line graph.

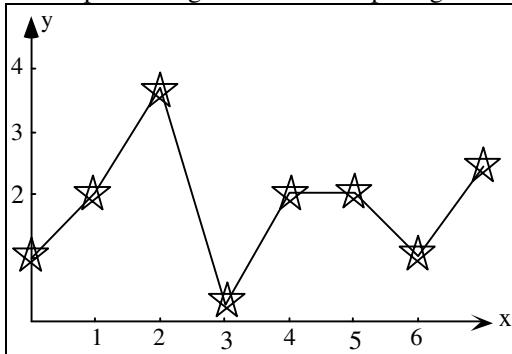


Figure 3.30. Placing markers for emphasis.

Because the same figure is drawn at several different points it is convenient to be able to say simply `drawMarker()` and have it be drawn at the *CP*. Then the line graph of Figure 3.30 can be drawn along with the markers using code suggested by the pseudocode:

```

moveTo(first data point);
drawMarker();           // draw a marker there
for(each remaining data point)
{
    lineTo(the next point); // draw the next line segment
    drawMarker();           // draws it at the CP
}

```

Figure 3.31 shows an arrow-shaped marker, drawn using the routine in Figure 3.32. The arrow is positioned with its uppermost point at the *CP*. For flexibility the arrow shape is parameterized through four size parameters *f*, *h*, *t*, and *w* as shown. Function `arrow()` uses only `lineRel()`, and no reference is made to absolute positions. Also note that although the *CP* is altered while drawing is going on, at the end the *CP* has been set back to its initial position. Hence the routine produces no “side effects” (beyond the drawing itself).

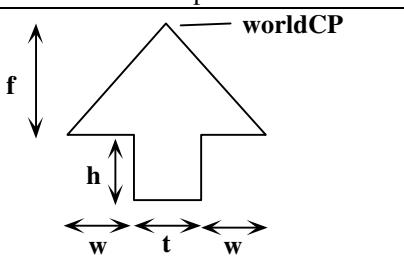


Figure 3.31. Model of an arrow.

```

void arrow(float f, float h, float t, float w)
{ // assumes global Canvas object: cvs
    cvs.lineRel(-w - t / 2, -f);           // down the left side
    cvs.lineRel(w, 0);
}

```

```

    cvs.lineRel(0, -h);
    cvs.lineRel(t, 0);                                // across
    cvs.lineRel(0, h);                                // back up
    cvs.lineRel(w, 0);
    cvs.lineRel(-w - t / 2, f);
}

```

Figure 3.32. Drawing an arrow using relative moves and draws.

3.5.2. Turtle Graphics.

The last tool we add for now is surprisingly convenient. It keeps track not only of “where we are” with the *CP*, but also “the direction in which we are headed”. This is a form of **turtlegraphics**, which has been found to be a natural way to program in graphics⁶. The notion is that a “turtle”, which is conceptually similar to the pen in a pen plotter, migrates over the page, leaving a trail behind itself which appears as a line segment. The turtle is positioned at the *CP*, headed in a certain direction called the **current direction**, *CD*. *CD* is the number of degrees measured counterclockwise (CCW) from the positive *x*-axis.

It is easy to add functionality to the *Canvas* class to “control the turtle”. First, *CD* is added as a private data member. Then we add three methods:

- 1). *turnTo(float angle)*. Turn the turtle to the given angle, implemented as:

```
void Canvas:: turnTo(float angle) {CD = angle;}
```

- 2). *turn(float angle)*. Turn the turtle through *angle* degrees counterclockwise:

```
void Canvas:: turn(angle){CD += angle;}
```

Use a negative argument to make a right turn. Note that a *turn* is a relative direction change: we don’t specify a direction, only a change in direction. This simple distinction provides enormous power in drawing complex figures with the turtle.

- 3). *forward(float dist, int isVisible)*. Move the turtle forward in a straight line from the *CP* through a distance *dist* in the current direction *CD*, and update the *CP*. If *isVisible* is nonzero a visible line is drawn; otherwise nothing is drawn.

Figure 3.33 shows that in going forward in direction *CD* the turtle just moves in *x* through the amount $dist * \cos(\pi * CD/180)$ and in *y* through the amount $dist * \sin(\pi * CD/180)$, so the implementation of *forward()* is immediate:

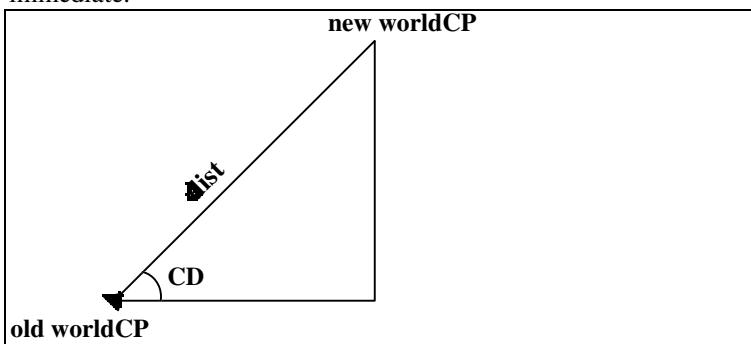


Figure 3.33. Effect of the *forward()* routine.

```
void Canvas:: forward(float dist, int isVisible)
{
```

⁶Introduced by Seymour Papert at MIT as part of the LOGO language for teaching children how to program. See e.g. [Abel81]

```

        const float RadPerDeg = 0.017453393; //radians per degree
        float x = CP.x + dist * cos(RadPerDeg * CD);
        float y = CP.y + dist * sin(RadPerDeg * CD);
        if(isVisible)
            lineTo(x, y);
        else
            moveTo(x, y);
    }
}

```

Turtle graphics makes it easy to build complex figures out of simpler ones, as we see in the next examples.

Example 3.5.2. Building a figure upon a hook motif. The 3-segment “hook” motif shown in Figure 3.34a can be drawn using the commands:

```

forward(3 * L, 1); // L is the length of the short sides
turn(90);
forward(L, 1);
turn(90);
forward(L, 1);
turn(90);

```

for some choice of L . Suppose that procedure `hook()` encapsulates these instructions. Then the shape in Figure 3.34b is drawn using four repetitions of `hook()`. The figure can be positioned and oriented as desired by choices of the initial CP and CD .

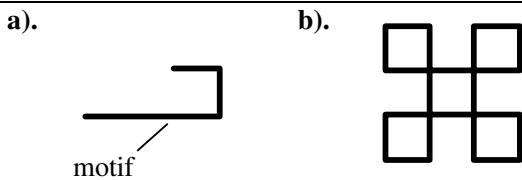


Figure 3.34. Building a figure out of several turtle motions.

Example 3.5.3. Polyspirals. A large family of pleasing figures called *polyspirals* can be generated easily using turtlegraphics. A polyspiral is a polyline where each successive segment is larger (or smaller) than its predecessor by a fixed amount, and oriented at some fixed angle to the predecessor. A polyspiral is rendered by the following pseudocode:

```

for(<some number of iterations>)
{
    forward(length,1);      // draw a line in the current direction
    turn(angle);           // turn through angle degrees
    length += increment;  // increment the line length
}

```

Each time a line is drawn both its length and direction are incremented. If `increment` is 0, the figure neither grows nor shrinks.. Figure 3.35 shows several polyspirals. The implementation of this routine is requested in the exercises.

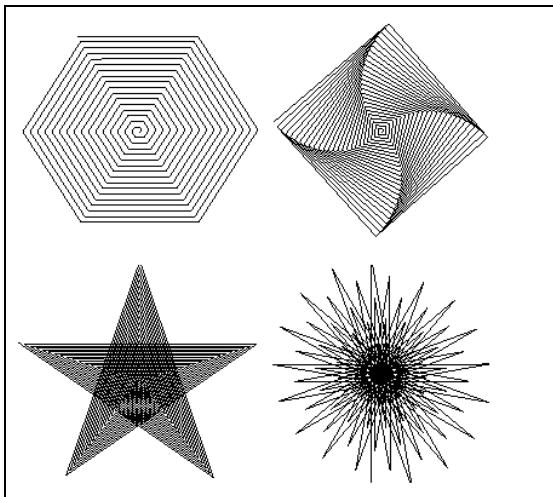


Figure 3.35. Examples of polyspirals. Angles are: a). 60, b). 89.5, c). -144, d). 170.

Practice Exercises.

3.5.1. Drawing Turtle figures. Provide routines that use turtle motions to draw the three figures shown in Figure 3.36. Can the turtle draw the shape in part c without “lifting the pen” and without drawing any line twice?

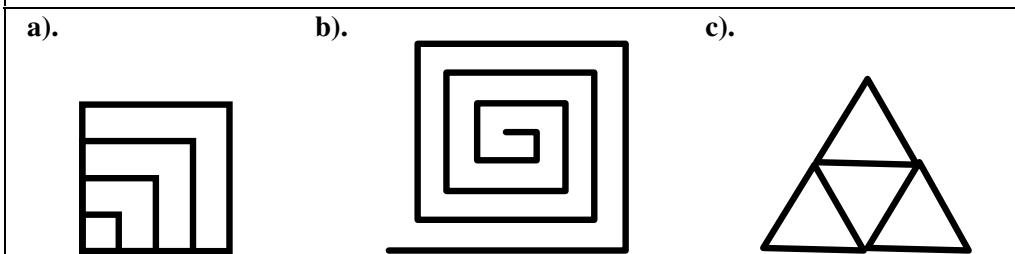


Figure 3.36. Other Simple Turtle Figures.

3.5.2. Drawing a well-known logo. Write a routine that makes a turtle draw the outline of the logo shown in Figure 3.37. (It need not fill the polygons.)

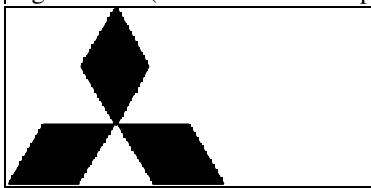


Figure 3.37. A famous logo.

3.5.3. Driving the Turtle with Strings. We can use a shorthand notation to describe a figure. Suppose

F	means <code>forward(d, 1);</code>	{for some distance d}
L	means <code>turn(60);</code>	{left turn}
R	means <code>turn(-60).</code>	{right turn}

What does the following sequence of commands produce?

`FLFLFLRFLFLFLRFLFLFR.` (See Chapter 9 for a generalization of this that produces fractals!)

3.5.4. Drawing Meanders. A meander⁷ is a pattern like that in Figure 3.38a, often made up of a continuous line meandering along some path. One frequently sees meanders on Greek vases, Chinese plates, or floor tilings from various countries. The motif for the meander here is shown in Figure 3.38b. After each motif is drawn the turtle is turned (how much?) to prepare it for drawing the next motif.

⁷Based on the name Maeander (which has modern name Menderes), a winding river in Turkey [Janson 86].

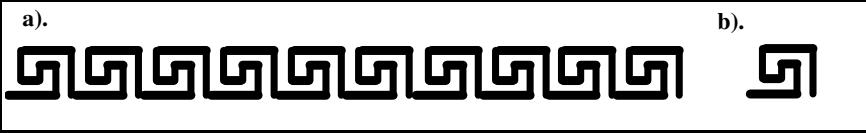


Figure 3.38. Example of a meander.

Write a routine that draws this motif, and a routine that draws this meander. (Meanders are most attractive if the graphics package at hand supports the control of line thickness -- as OpenGL does -- so that `forward()` draws thick lines.) A dazzling variety of more complex meanders can be designed, as suggested in later exercises. A meander is a particular type of **frieze** pattern. Friezes are studied further in Chapter ???.

3.5.5. Other Classes of Meanders. Figure 3.39 shows two additional types of meanders. Write routines that employ turtle graphics to draw them.

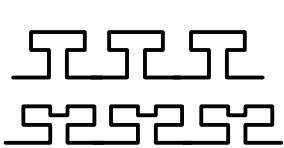


Figure 3.39. Additional figures for meanders.

3.5.6. Drawing Elaborate Meanders. Figure 3.40 shows a sequence of increasingly complex motifs for meanders. Write routines that draw a meander for each of these motifs. What does the “next most complicated” motif in this sequence look like, and what is the general principal behind constructing these motifs?

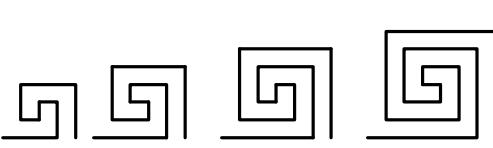


Figure 3.40. Hierarchy of meander motifs.

3.5.7. Implementing polyspiral. Write the routine `polyspiral(float length, float angle, float incr, int num)` that draws a polyspiral consisting of `num` segments, the first having length `length`. After each segment is drawn `length` is incremented by `incr` and the turtle turns through angle `angle`.

3.5.8. Is a Polyspiral an IFS? Can a polyspiral be described in terms of an iterated function system as defined in Chapter 2? Specify the function that is iterated by the turtle at each iteration.

3.5.9. Recursive form for Polyspiral(). Rewrite `polyspiral()` in a recursive form, so that `polyspiral()` with argument `dist` calls `polyspiral()` with argument `dist+inc`. Put a suitable stopping criterion in the routine.

3.6. Figures based on Regular Polygons.

*To generalize is to be an idiot.
William Blake*

“Bees...by virtue of certain geometrical forethought...know that the hexagon is greater than the square and triangle, and will hold more honey for the same expenditure of material.”

Pappus of Alexandria

The regular polygons form a large and important family of shapes, often encountered in computer graphics. We need efficient ways to draw them. In this section we examine how to do this, and how to create a number of figures that are variations of the regular polygon.

3.6.1. The Regular Polygons.

First recall the definition of a regular polygon:

Definition: A polygon is **regular** if it is simple, if all its sides have equal lengths, and if adjacent sides meet at equal interior angles.

As discussed in Chapter 1, a polygon is **simple** if no two of its edges cross each other (more precisely: only adjacent edges can touch, and only at their shared endpoint). We give the name **n-gon** to a regular polygon having n sides. Familiar examples are the 4-gon (a square), a 5-gon (a regular pentagon), 8-gon (a regular octagon), and so on. A 3-gon is an equilateral triangle. Figure 3.41 shows various examples. If the number of sides of an n -gon is large the polygon approximates a circle in appearance. In fact this is used later as one way to implement the drawing of a circle.

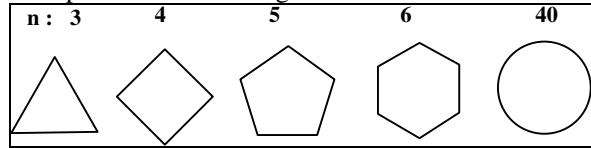


Figure 3.41. Examples of n -gons.

The vertices of an n -gon lie on a circle, the so-called “parent circle” of the n -gon, and their locations are easily calculated. The case of the hexagon is shown in Figure 3.42 where the vertices lie equispaced every 60° around the circle. The parent circle of radius R (not shown) is centered at the origin, and the first vertex P_0 has been placed on the positive x -axis. The other vertices follow accordingly, as $P_i = (R \cos(i \cdot a), R \sin(i \cdot a))$, for $i = 1, \dots, 5$, where a is $2\pi/6$ radians. Similarly, the vertices of the general n -gon lie at:

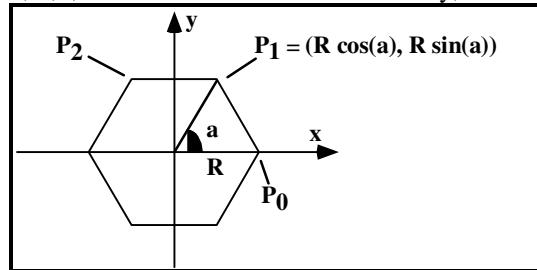


Figure 3.42. Finding the vertices of an 6-gon.

$$P_i = (R \cos(2\pi i / n), R \sin(2\pi i / n)), \text{ for } i = 0, \dots, n-1 \quad (3.6)$$

It's easy to modify this n -gon. To center it at position (cx, cy) we need only add cx and cy to the x - and y -coordinates, respectively. To scale it by factor S we need only multiply R by S . To rotate through angle A we need only add A to the arguments of $\cos()$ and $\sin()$. More general methods for performing geometrical transformations are discussed in Chapter 6.

It is simple to implement a routine that draws an n -gon, as shown in Figure 3.43. The n -gon is drawn centered at (cx, cy) , with radius $radius$, and is rotated through $rotAngle$ degrees.

```
void ngon(int n, float cx, float cy, float radius, float rotAngle)
{
    // assumes global Canvas object, cvs
    if(n < 3) return;                                // bad number of sides
    double angle = rotAngle * 3.14159265 / 180;    // initial angle
    double angleInc = 2 * 3.14159265 / n;           //angle increment
    cvs.moveTo(radius + cx, cy);
    for(int k = 0; k < n; k++) // repeat n times
    {
        angle += angleInc;
        cvs.lineTo(radius * cos(angle) + cx, radius * sin(angle) + cy);
    }
}
```

Figure 3.43. Building an n -gon in memory.

Example 3.6.1: A Turtle-driven n-gon. It is also simple to draw an n -gon using turtlegraphics. Figure 3.44 shows how to draw a regular hexagon. The initial position and direction of the turtle is indicated by the small triangle. The turtle simply goes forward six times, making a CCW turn of 60 degrees between each move:

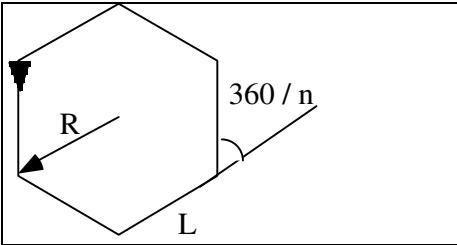


Figure 3.44. Drawing a hexagon.

```
for (i = 0; i < 6; i++)
{
    cvs.forward(L, 1);
    cvs.turn(60);
}
```

One vertex is situated at the initial CP , and both CP and CD are left unchanged by the process. Drawing the general n -gon, and some variations of it, is discussed in the exercises.

3.6.2. Variations on n-gons.

Interesting variations based on the vertices of an n -gon can also be drawn. The n -gon vertices may be connected in various ways to produce a variety of figures, as suggested in Figure 3.45. The standard n -gon is drawn in Figure 3.45a by connecting adjacent vertices, but Figure 3.45b shows a **stellation** (or star-like figure) formed by connecting every other vertex. And Figure 3.45c shows the interesting **rosette**, formed by connecting each vertex to every other vertex. We discuss the rosette next. Other figures are described in the exercises.

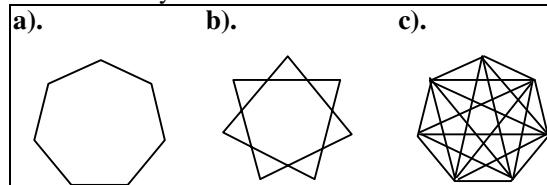


Figure 3.45. A 7-gon and its offspring. a). the 7-gon, b). a stellation, c). a “7-rosette”.

Example 3.6.2. The rosette, and the Golden 5-rosette.

The **rosette** is an n -gon with each vertex joined to every other vertex. Figure 3.46 shows 5-, 11-, and 17-rosettes. A rosette is sometimes used as a test pattern for computer graphics devices. Its orderly shape readily reveals any distortions, and the resolution of the device can be determined by noting the amount of “crowding” and blurring exhibited by the bundle of lines that meet at each vertex.

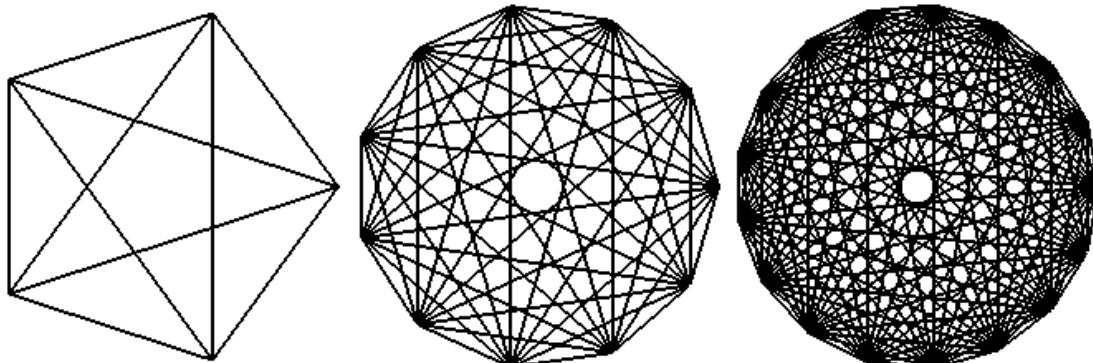


Figure 3.46. The 5-, 11-, and 17-rosettes.

Rosettes are easy to draw: simply connect every vertex to every other. In pseudocode this looks like

```
void Rosette(int N, float radius)
{
    Point2 pt[big enough value for largest rosette];
    generate the vertices pt[0], . . . , pt[N-1], as in Figure 3.43
    for(int i = 0; i < N - 1; i++)
        for(int j = i + 1; j < N; j++)
    {
        cvs.moveTo(pt[i]); // connect all the vertices
        cvs.lineTo(pt[j]);
    }
}
```

The 5-rosette is particularly interesting because it embodies many instances of the golden ratio ϕ (recall Chapter 2). Figure 3.47a shows a 5-rosette, which is made up of an outer pentagon and an inner pentagram. The Greeks saw a mystical significance in this figure. Its segments have an interesting relationship: Each segment is ϕ times longer than the next smaller one (see the exercises). Also, because the edges of the star pentagram form an inner pentagon, an infinite regression of pentagrams is possible, as shown in Figure 3.47b.

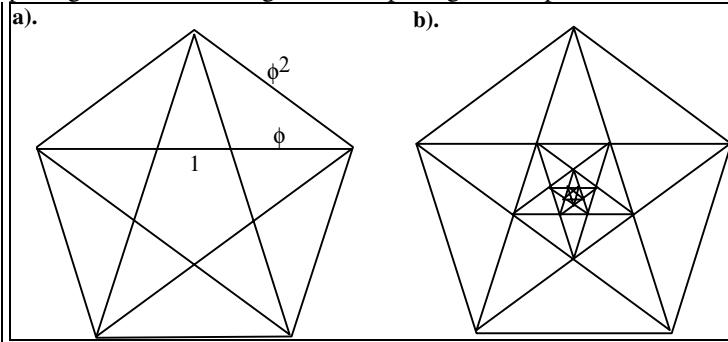


Figure 3.47. 5-rosette and Infinite regressions - pentagons and pentagrams.

Example 3.6.3. Figures based on two concentric n-gons.

Figures 3.48 shows some shapes built upon two concentric parent circles, the outer of radius R , and the inner of radius fR for some fraction f . Each figure uses a variation of an n -gon whose radius alternates between the inner and outer radii. Parts a) and b) show familiar company logos based on 6-gons and 10-gons. Part c) is based on the 14-gon, and part d) shows the inner circle explicitly.

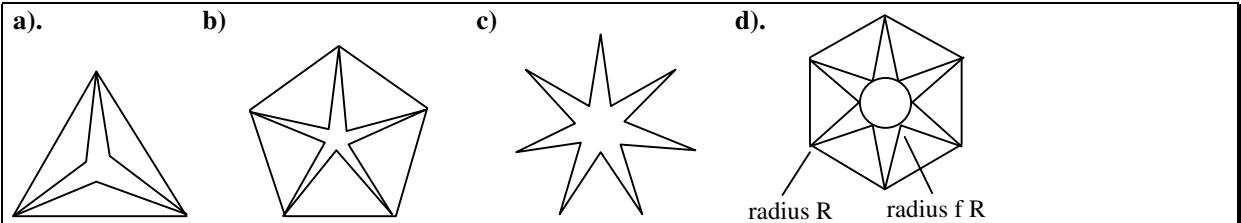


Figure 3.48. A family of Famous Logos.

Practice Exercises.

3.6.1. Stellations and rosettes. The pentagram is drawn by connecting “every other” point as one traverses around a 5-gon. Extend this to an arbitrary odd-valued n -gon and develop a routine that draws this so-called “stellated” polygon. Can it be done with a single initial `moveTo()` followed only by `lineTo()`’s (that is, without “lifting the pen”)? What happens if n is even?

3.6.2. How Many Edges in an N -rosette? Show that a rosette based on an N -gon, an N -rosette, has $N(N - 1) / 2$ edges. This is the same as the number of “clinks” one hears when N people are seated around a table and everybody clinks glasses with everyone else.

3.6.3. Prime Rosettes. If a rosette has a prime number N of sides, it can be drawn without “lifting the pen,” that is, by using only `lineTo()`. Start at vertex v_0 and draw to each of the others in turn: v_1, v_2, v_3, \dots until v_0 is again reached and the polygon is drawn. Then go around again drawing lines, but skip a vertex each time – that is, increment the index by 2 – thereby drawing to v_2, v_4, \dots, v_0 . This will require going around twice to arrive back at v_0 . (A *modulo* operation is performed on the indices so that their values remain between 0 and $N-1$.)

Then repeat this, incrementing by 3: $v_3, v_6, v_0, \dots, v_0$. Each repeat draws exactly N lines. Because there are $N(N - 1) / 2$ lines in all, the process repeats $(N - 1) / 2$ times. Because the number of vertices is a prime, no pattern is ever repeated until the drawing is complete. Develop and test a routine that draws prime rosettes in this way.

3.6.4. Rosettes with an odd number of sides. If n is prime we know the n -rosette can be drawn as a single polyline without “lifting the pen”. It can also be drawn as a single polyline for any *odd* value of n . Devise a method that does this.

3.6.5. The Geometry of the Star Pentagram. Show that the length of each segment in the 5-rosette stands in the golden ratio to that of the next smaller one. One way to tackle this is to show that the triangles of the star pentagram are “golden triangles” with an inner angle of $\pi / 5$ radians. Show that $2 * \cos(\pi / 5) = \phi$ and $2 * \cos(2\pi / 5) = 1 / \phi$. Another approach uses only two families of similar triangles in the pentagram and the relation $\phi^3 = 2\phi + 1$ satisfied by ϕ .

3.6.6. Erecting Triangles on n -gon legs. Write a routine that draws figures like the logo in part a of Figure 3.48 for any value of f , positive or negative. What is a reasonable geometric interpretation of negative f ?

3.6.7. Drawing the Star with Relative Moves and Draws. Write a routine to draw a pentagram that uses only relative moves and draws, centering the star at the *CP*.

3.6.8. Draw a pattern of stars. Write a routine to draw the pattern of 21 stars shown in Figure 3.49. The small stars are positioned at the vertices of an n -gon.

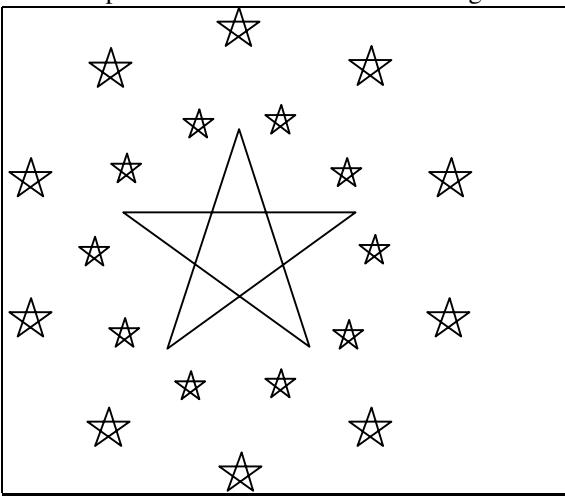


Figure 3.49. A star pattern.

3.6.9. New points on the “7-gram”. Figure 3.50 shows a figure formed from the 7 points of a 7-gon, centered at the origin. The first point lies at $(R, 0)$. Instead of connecting consecutive points around the 7-gon, two intermediary points are skipped. (This is a form of “stellation” of an n -gon.) Find the coordinates of point P , where two of the edges intersect.

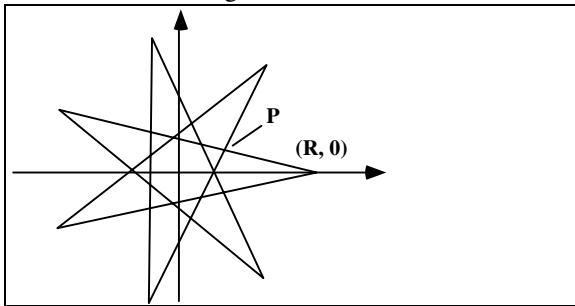


Figure 3.50. A “7-gram”.

3.6.10. Turtle drawings of the n -gon. Write `turtleNgon(int numSides, float length)` that uses turtlegraphics to draw an n -gon with `numSides` sides and a side of length `length`.

3.6.11. Polygons sharing an edge. Write a routine that draws n -gon's, for $n = 3, \dots, 12$, on a common edge, as in Figure 3.51.

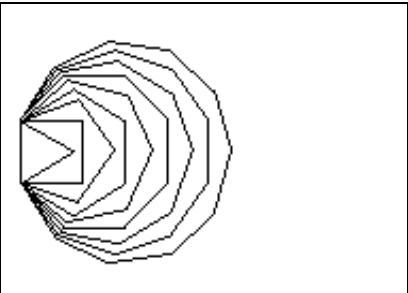


Figure 3.51. N -gons sharing a common edge.

3.6.12. A more elaborate figure. Write a routine that draws the shape in Figure 3.52 by drawing repeated hexagons rotated relative to one another.

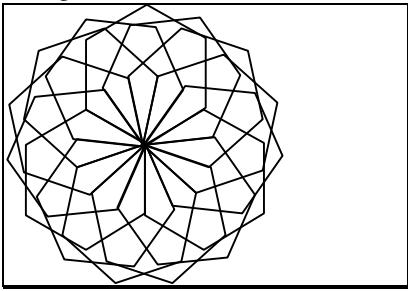


Figure 3.52. Repeated use of turtle commands.

3.6.13. Drawing a famous logo. The esteemed logo shown in Figure 3.53 consists of three instances of a motif, rotated a certain amount with respect to each other. Show a routine that draws this shape using turtlegraphics.

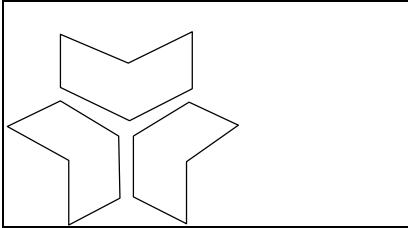


Figure 3.53. Logo of the University of Massachusetts.

3.6.14. Rotating Pentagons: animation. Figure 3.54 shows a pentagram oriented with some angle of rotation within a pentagon, with corresponding vertices joined together. Write a program that “animates” this figure. The configuration is drawn using some initial angle A of rotation for the pentagram. After a short pause it is erased and then redrawn but with a slightly larger angle A . This process repeats until a key is pressed.

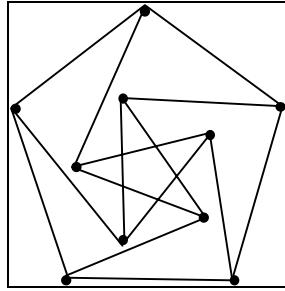


Figure 3.54. Rotating penta-things.

3.7. Drawing Circles and Arcs.

Drawing a circle is equivalent to drawing an n -gon that has a large number of vertices. The n -gon resembles a circle (unless it is scrutinized too closely). The routine `drawCircle()` shown in Figure 3.55 draws a 50-sided n -gon, by simply passing its parameters on to `ngon()`. It would be more efficient to write `drawCircle()` from scratch, basing it on the code of Figure 3.43.

```
void drawCircle(Point2 center, float radius)
{
    const int numVerts = 50;      // use larger for a better circle
    ngon(numVerts, center.getX(), center.getY(), radius, 0);
}
```

Figure 3.55. Drawing a circle based on a 50-gon.

3.7.1. Drawing Arcs.

Many figures in art, architecture, and science involve arcs of circles placed in pleasing or significant arrangements. An arc is conveniently described by the position of the center, c , and radius, R , of its “parent” circle, along with its beginning angle a and the angle b through which it “sweeps”. Figure 3.56 shows such an arc. We assume that if b is positive the arc sweeps in a CCW direction from a . If b is negative it sweeps in a CW fashion. A circle is a special case of an arc, with a sweep of 360° .

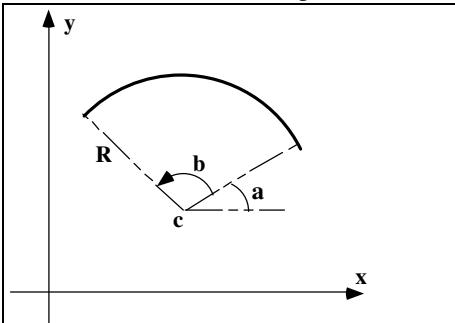


Figure 3.56. Defining an arc.

We want a routine, `drawArc()`, that draws an arc of a circle. The function shown in Figure 3.57 approximates the arc by part of an n -gon, using `moveTo()` and `lineTo()`. Successive points along the arc are found by computing a `cos()` and `sin()` term each time through the main loop. If `sweep` is negative the angle automatically decreases each time through.

```
void drawArc(Point2 center, float radius, float startAngle, float sweep)
{   // startAngle and sweep are in degrees
    const int n = 30; // number of intermediate segments in arc
    float angle = startAngle * 3.14159265 / 180; // initial angle in radians
    float angleInc = sweep * 3.14159265 / (180 * n); // angle increment
    float cx = center.getX(), cy = center.getY();
    cvs.moveTo(cx + radius * cos(angle), cy + radius * sin(angle));
    for(int k = 1; k < n; k++, angle += angleInc)
        cvs.lineTo(cx + radius * cos(angle), cy + radius * sin(angle));
}
```

Figure 3.57. Drawing an arc of a circle.

The `CP` is left at the last point on the arc. (In some cases one may wish to omit the initial `moveTo()` to the first point on the arc, so that the arc is connected to whatever shape was being drawn when `drawArc()` is called.)

A much faster arc drawing routine is developed in Chapter 5 that avoids the repetitive calculation of so many `sin()` and `cos()` functions. It may be used freely in place of the procedure here.

With `drawArc()` in hand it is a simple matter to build the routine `drawCircle(Point2 center, float radius)` that draws an entire circle (how?).

The routine `drawCircle()` is called by specifying a center and radius, but there are other ways to describe a circle, which have important applications in interactive graphics and computer-aided design. Two familiar ones are:

- 1). **The center is given, along with a point on the circle.** Here `drawCircle()` can be used as soon as the radius is known. If c is the center and p is the given point on the circle, the radius is simply the distance from c to p , found using the usual Pythagorean Theorem.
- 2). **Three points are given through which the circle must pass.** It is known that a unique circle passes through any three points that don't lie in a straight line. Finding the center and radius of this circle is discussed in Chapter 4.

Example 3.7.1. Blending Arcs together. More complex shapes can be obtained by using parts of two circles that are tangent to one another. Figure 3.58 illustrates the underlying principle. The two circles are tangent at point A, where they share tangent line L. Because of this the two arcs shown by the thick curve “blend” together seamlessly at A with no visible break or corner. Similarly the arc of a circle blends smoothly with any tangent line, as at point B.

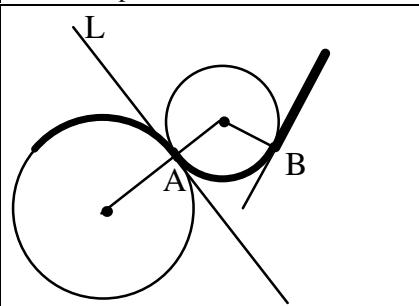
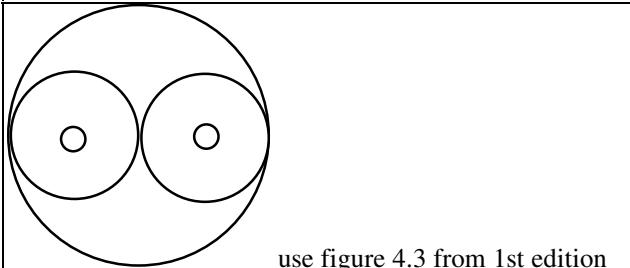


Figure 3.58. Blending arcs using tangent circles.

Practice Exercises.

3.7.1. Circle Figures in Philosophy. In Chinese philosophy and religion the two principles of yin and yang interact to influence all creatures' destinies. Figure 3.59 shows the exquisite yin-yang symbol. The dark portion, yin, represents the feminine aspect, and the light portion, yang, represents the masculine. Describe in detail the geometry of this symbol, supposing it is centered in some coordinate system.



use figure 4.3 from 1st edition

Figure 3.59. The yin-yang symbol.

3.7.2. The Seven Pennies. Describe the configuration shown in Figure 3.60 in which six pennies fit snugly around a center penny. Use symmetry arguments to explain why the fit is exact; that is, why each of the outer pennies *exactly* touches its three neighbors.

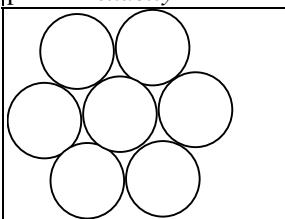


Figure 3.60. The seven circles.

3.7.3. A famous logo. Figure 3.61 shows a well-known automobile logo. It is formed by erecting triangles inside an equilateral triangle, but the outer triangle is replaced by two concentric circles. After determining the “proper” positions for the three inner points, write a routine to draw this logo.

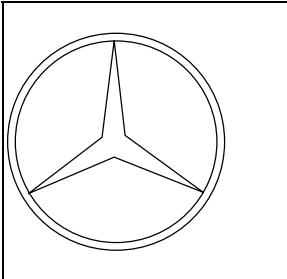


Figure 3.61. A famous logo.

3.7.4. Drawing clocks and such. Circles and lines may be made tangent in a variety of ways to create pleasing smooth curves, as in Figure 3.62a. Figure 3.62b shows the underlying lines and circles. Write a routine that draws this basic clock shape

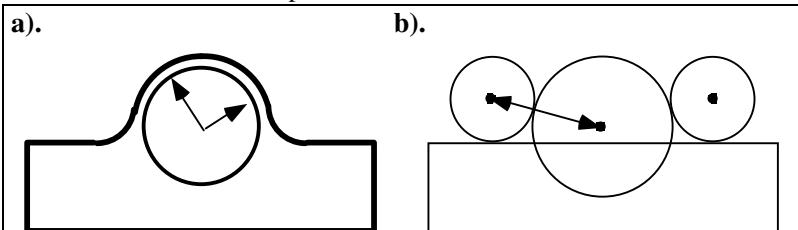


Figure 3.62. Blending arcs to form smooth curves.

3.7.5. Drawing rounded rectangles. Figure 3.63 shows an aligned rectangle with rounded corners. The rectangle has width W and aspect ratio R , and each corner is described by a quarter-circle of radius $r = g W$ for some fraction g . Write a routine `drawRoundRect(float W, float R, float g)` that draws this rectangle centered at the CP . The CP should be left at the center when the routine exits.

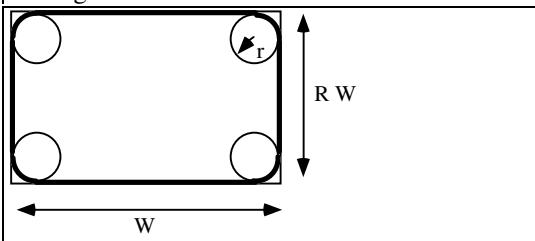


Figure 3.63. A rounded rectangle.

3.7.6. Shapes involving arcs. Figure 3.64 shows two interesting shapes that involve circles or arcs. One is similar to the Atomic Energy Commission symbol (How does it differ from the standard symbol?). Write and test two routines that draw these figures.

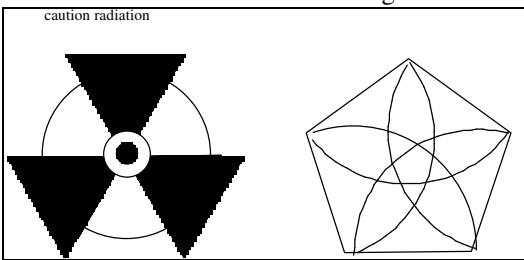


Figure 3.64. Shapes based on arcs.

3.7.7. A tear drop. A “tear drop” shape that is used in many ornamental figures is shown in Figure 3.65a. As shown in part b) it consists of a circle of given radius R snuggled down into an angle ϕ . What are the coordinates of the circle’s center C for a given R and ϕ ? What are the initial angle of the arc, and its sweep? Develop a routine to draw a tear drop at any position and in any orientation.

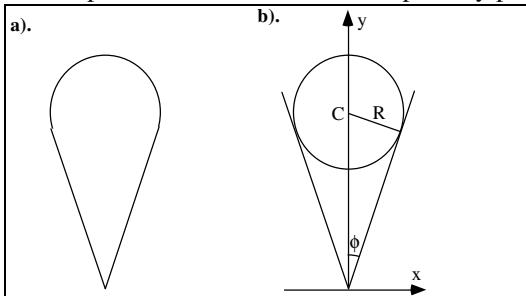


Figure 3.65. The tear drop and its construction.

3.7.8. Drawing Patterns of Tear Drops. Figure 3.66 show some uses of the tear drop. Write a routine that draws each of them.

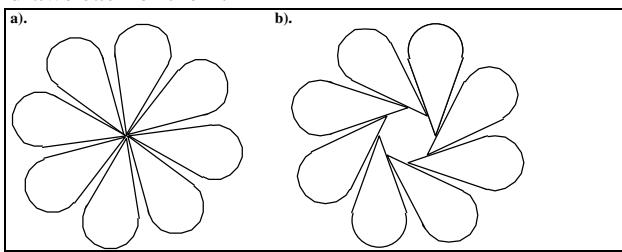


Figure 3.66. Some figures based on the tear drop.

3.7.9. Pie Charts. A sector is closely related to an arc: each end of the arc is connected to the center of the circle. The familiar pie chart is formed by drawing a number of sectors. A typical example is shown in Figure 3.67. Pie charts are used to illustrate how a whole is divided into parts, as when a pie is split up and distributed. The eye quickly grasps how big each “slice” is relative to the others. Often one or more of the slices is “exploded” away from the pack as well, as shown in the figure. Sectors that are exploded are simply shifted slightly away from the center of the pie chart in the proper direction

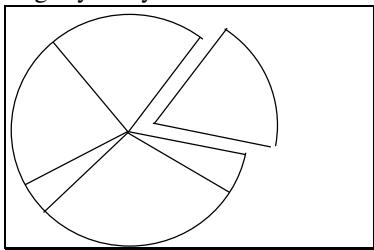


Figure 3.67. A pie chart.

To draw a pie chart we must know the relative sizes of the slices. Write and test a routine that accepts data from the user and draws the corresponding pie chart. The user enters the fraction of the pie each slice represents, along with an ‘e’ if the slice is to be drawn exploded, or an ‘n’ otherwise.

3.8. Using the Parametric form for a curve.

There are two principal ways to describe the shape of a curved line: implicitly and parametrically. The **implicit form** describes a curve by a function $F(x, y)$ that provides a relationship between the x and y coordinates: the point (x, y) lies on the curve if and only if it satisfies:

$$F(x, y) = 0 \quad \text{condition for } (x, y) \text{ to lie on the curve} \quad (3.7)$$

For example, the straight line through points A and B has implicit form:

$$F(x, y) = (y - A_y)(B_x - A_x) - (x - A_x)(B_y - A_y) \quad (3.8)$$

and the circle with radius R centered at the origin has implicit form:

$$F(x, y) = x^2 + y^2 - R^2 \quad (3.9)$$

A benefit of using the implicit form is that you can easily test whether a given point lies on the curve: simply evaluate $F(x, y)$ at the point in question. For certain classes of curves it is meaningful to speak of an inside and an outside of the curve, in which case $F(x, y)$ is also called the **inside-outside function**, with the understanding that

$F(x, y) = 0$	for all (x, y) on the curve	(3.10)
$F(x, y) > 0$	for all (x, y) outside the curve	
$F(x, y) < 0$	for all (x, y) inside the curve	

(Is $F(x, y)$ of Equation 3.9 a legitimate inside-outside function for the circle?)

Some curves are **single-valued** in x , in which case there is a function $g(\cdot)$ such that all points on the curve satisfy $y = g(x)$. For such curves the implicit form may be written $F(x, y) = y - g(x)$. (What is $g(\cdot)$ for the line of Equation 3.8?) Other curves are single-valued in y , (so there is a function $h(\cdot)$ such that points on the curve satisfy $x = h(y)$). And some curves are not single-valued at all: $F(x, y) = 0$ cannot be rearranged into either of the forms $y = g(x)$ nor $x = h(y)$. The circle, for instance, can be expressed as:

$$y = \pm\sqrt{R^2 - x^2} \quad (3.11)$$

but here there are two functions, not one.

3.8.1. Parametric Forms for Curves.

A parametric form for a curve produces different points on the curve based on the value of a parameter. Parametric forms can be developed for a wide variety of curves, and they have much to recommend them, particularly when one wants to draw or analyze the curve. A parametric form suggests the movement of a point through time, which we can translate into the motion of a pen as it sweeps out the curve. The path of the particle traveling along the curve is fixed by two functions, $x(\cdot)$ and $y(\cdot)$, and we speak of $(x(t), y(t))$ as the **position** of the particle at time t . The curve itself is the totality of points “visited” by the particle as t varies over some interval. For any curve, therefore, if we can dream up suitable functions $x(\cdot)$ and $y(\cdot)$ they will represent the curve concisely and precisely.

The familiar Etch-a-Sketch⁸ shown in Figure 3.68 provides a vivid analogy. As knobs are turned, a stylus hidden in the box scrapes a thin visible line across the screen. One knob controls the horizontal position, and the other directs the vertical position of the stylus. If the knobs are turned in accordance with $x(t)$ and $y(t)$, the **parametric curve** is swept out. (Complex curves require substantial manual dexterity.)

1st Ed. Figure 7.11

Figure 3.68. Etch-a-Sketch drawings of parametric curves. (Drawing by Suzanne Casiello.)

Examples: The line and the ellipse.

The straight line of Equation 3.8 passes through points A and B . We choose a parametric form that visits A at $t = 0$ and B at $t = 1$, obtaining:

$$\begin{aligned} x(t) &= A_x + (B_x - A_x)t \\ y(t) &= A_y + (B_y - A_y)t \end{aligned} \quad (3.12)$$

⁸Etch-a-Sketch is a trademark of Ohio Art.

Thus the point $P(t) = (x(t), y(t))$ “sweeps through” all of the points on the line between A and B as t varies from 0 to 1 (check this out).

Another classic example is the **ellipse**, a slight generalization of the circle. It is described parametrically by

$$\begin{aligned} x(t) &= W \cos(t) \\ y(t) &= H \sin(t) \end{aligned} \quad , \text{ for } 0 \leq t \leq 2\pi \quad (3.13)$$

Here W is the “half-width”, and H the “half-height” of the ellipse. Some of the geometric properties of the ellipse are explored in the exercises. When W and H are equal the ellipse is a circle of radius W .

Figure 3.69 shows this ellipse, along with the component functions $x(.)$ and $y(.)$.

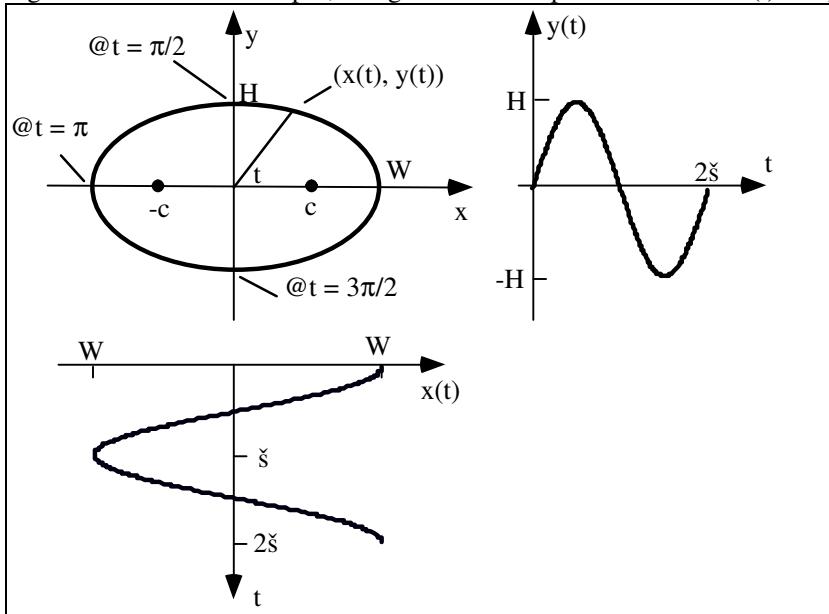


Figure 3.69. An ellipse described parametrically.

As t varies from 0 to 2π the point $P(t) = (x(t), y(t))$ moves once around the ellipse starting (and finishing) at $(W, 0)$. The figure shows where the point is located at various “times” t . It is useful to visualize drawing the ellipse on an Etch-a-Sketch. The knobs are turned back and forth in an undulating pattern, one mimicking $W \cos(t)$ and the other $H \sin(t)$. (This is surprisingly difficult to do manually.)

• Finding an implicit form from a parametric form - “implicitization”.

Suppose we want to check that the parametric form in Equation 3.13 truly represents an ellipse. How do we find the implicit form from the parametric form? The basic step is to combine the two equations for $x(t)$ and $y(t)$ to somehow eliminate the variable t . This provides a relationship that must hold for *all* t . It isn’t always easy to see how to do this — there are no simple guidelines that apply for all parametric forms. For the ellipse, however, square both x/W and y/H and use the well-known fact $\cos(t)^2 + \sin(t)^2 = 1$ to obtain the familiar equation for an ellipse:

$$\left(\frac{x}{W}\right)^2 + \left(\frac{y}{H}\right)^2 = 1 \quad (3.14)$$

The following exercises explore properties of the ellipse and other “classical curves”. They develop useful facts about the **conic sections**, which will be used later. Read them over, even if you don’t stop to solve each one.

| Practice Exercises

3.8.1. On the geometry of the Ellipse. An ellipse is the set of all points for which the sum of the distances to two foci is constant. The point $(c, 0)$ shown in Figure 3.69 forms one “focus”, and $(-c, 0)$ forms the other. Show that H , W , and c are related by: $W^2 = H^2 + c^2$.

3.8.2. How eccentric. The **eccentricity**, $e = c / W$, of an ellipse is a measure of how non circular the ellipse is, being 0 for a true circle. As interesting examples, the planets in our solar system have very nearly circular orbits, with e ranging from $1/143$ (Venus) to $1/4$ (Pluto). Earth’s orbit exhibits $e = 1/60$. As the eccentricity of an ellipse approaches 1, the ellipse flattens into a straight line. But e has to get very close to 1 before this happens. What is the ratio H / W of height to width for an ellipse that has $e = 0.99$?

3.8.3. The other Conic Sections.

The ellipse is one of the three conic sections, which are curves formed by cutting (“sectioning”) a circular cone with a plane, as shown in Figure 3.70. The conic sections are:

- ellipse: if the plane cuts one “nappe” of the cone;
- hyperbola: if the plane cuts both nappes
- parabola: if the plane is parallel to the side of the cone;

Figure 3.70. The classical conic sections.

The parabola and hyperbola have interesting and useful geometric properties. Both of them have simple implicit and parametric representations.

Show that the following parametric representations are consistent with the implicit forms given:

• **Parabola:** Implicit form: $y^2 - 4ax = 0$

$$\begin{aligned} x(t) &= at^2 \\ y(t) &= 2at \end{aligned} \tag{3.15}$$

• **Hyperbola:** Implicit form: $(x/a)^2 - (y/b)^2 = 1$

$$\begin{aligned} x(t) &= a \sec(t) \\ y(t) &= b \tan(t) \end{aligned} \tag{3.16}$$

What range in the parameter t is used to sweep out this hyperbola? Note: A hyperbola is defined as the locus of all points for which the *difference* in its distances from two fixed foci is a constant. If the foci here are at $(-c, 0)$ and $(+c, 0)$, show that a and b must be related by $c^2 = a^2 + b^2$.

3.8.2. Drawing curves represented parametrically.

It is straightforward to draw a curve when its parametric representation is available. This is a major advantage of the parametric form over the implicit form. Suppose a curve C has the parametric representation $P(t) = (x(t), y(t))$ as t varies from 0 to T (see Figure 3.71a). We want to draw a good approximation to it, using only straight lines. Just take **samples** of $P(t)$ at closely spaced “instants”. A sequence $\{t_i\}$ of times are chosen, and for each t_i the position $P_i = P(t_i) = (x(t_i), y(t_i))$ of the curve is found. The curve $P(t)$ is then approximated by the polyline based on this sequence of points P_i , as shown in Figure 3.71b.

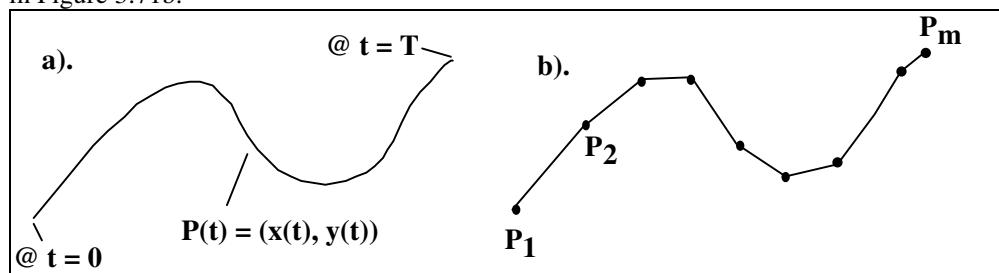


Figure 3.71. Approximating a curve by a polyline.

Figure 3.72 shows a code fragment that draws the curve $(x(t), y(t))$ when the desired array of sample times $t[i]$ is available.

```

// draw the curve (x(t), y(t)) using
// the array t[0],...,t[n-1] of "sample-times"

glBegin(GL_LINES);
    for(int i = 0; i < n; i++)
        glVertex2f((x(t[i])), y(t[i]));
glEnd();

```

Figure 3.72. Drawing the ellipse using points equispaced in t .

If the samples are spaced sufficiently close together, the eye will naturally blend together the line segments and will see a smooth curve. Samples must be closely spaced in t -intervals where the curve is “wiggling” rapidly, but may be placed less densely where the curve is undulating slowly. The required “closeness” or “quality” of the approximation depends on the situation.

Code can often be simplified if it is needed only for a specific curve. The ellipse in Equation 3.13 can be drawn using n equispaced values of t with:

```

#define TWOPI 2 * 3.14159265
glBegin(GL_LINES);
    for(double t = 0; t <= TWOPI; t += TWOPI/n)
        glVertex2f(W * cos(t), H * sin(t));
glEnd();

```

For drawing purposes, parametric forms circumvent all of the difficulties of implicit and explicit forms. Curves can be multi-valued, and they can self-intersect any number of times. Verticality presents no special problem: $x(t)$ simply becomes constant over some interval in t . Later we see that drawing curves that lie in 3D space is just as straightforward: three functions of t are used, and the point at t on the curve is $(x(t), y(t), z(t))$.

Practice Exercises.

3.8.4. An Example Curve. Compute and plot by hand the points that would be drawn by the fragment above for $W = 2$, $H = 1$, at the 5 values of $t = 2\pi i/9$, for $i = 0, 1, \dots, 4$.

3.8.5. Drawing a logo. A well-known logo consists of concentric circles and ellipses, as shown in Figure 3.73. Suppose you have a drawing tool: `drawEllipse(W, H, color)` that draws the ellipse of Equation 3.13 filled with color `color`. (Assume that as each color is drawn it completely obscures any previously drawn color.) Choose suitable dimensions for the ellipses in the logo and give the sequence of commands required to draw it.

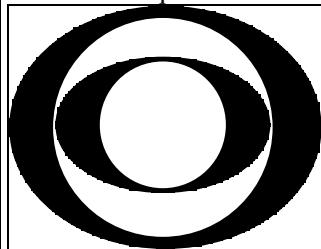


Figure 3.73. A familiar “eye” made of circles and ellipses.

Some specific examples of curves used in computer graphics will help to cement the ideas.

3.8.3. Superellipses

An excellent variation of the ellipse is the **superellipse**, a family of ellipse-like shapes that can produce good effects in many drawing situations. The implicit formula for the superellipse is

$$\left(\frac{x}{W}\right)^n + \left(\frac{y}{H}\right)^n = 1 \quad (3.17)$$

where n is a parameter called the *bulge*. Looking at the corresponding formula for the ellipse in Equation 3.14, the superellipse is seen to become an ellipse when $n = 2$. The superellipse has the following parametric representation:

$$\begin{aligned}x(t) &= W \cos(t) |\cos(t)|^{2/n-1} \\y(t) &= H \sin(t) |\sin(t)|^{2/n-1}\end{aligned}\tag{3.18}$$

for $0 \leq t \leq 2\pi$. The exponent on the $\sin()$ and $\cos()$ is really $2/n$, but the peculiar form as shown is used to avoid trying to raise a negative number to a fractional power. A more precise version avoids this. Check that this form reduces nicely to the equation for the ellipse when $n = 2$. Also check that the parametric form for the superellipse is consistent with the implicit equation.

Figure 3.74a shows a family of **supercircles**, special cases of superellipses for which $W = H$. Figure 3.74b shows a scene composed entirely of superellipses, suggesting the range of shapes possible.

1st Ed. Figures 4.16 and 4.17 together

Figure 3.74. Family of supercircles. b). Scene composed of superellipses.

For $n > 1$ the bulge is outward, whereas for $n < 1$ it is inward. When $n = 1$, it becomes a square. (In Chapter 6 we shall look at three-dimensional “superquadrics,” surfaces that are sometimes used in CAD systems to model solid objects.)

Superellipses were first studied in 1818 by the French physicist Gabriel Lamé. More recently in 1959, the extraordinary inventor Piet Hein (best known as the originator of the Soma cube and the game Hex) was approached with the problem of designing a traffic circle in Stockholm. It had to fit inside a rectangle (with $W/H = 6/5$) determined by other roads, and had to permit smooth traffic flow as well as be pleasing to the eye. An ellipse proved to be too pointed at the ends for the best traffic patterns, and so Piet Hein sought a fatter curve with straighter sides and dreamed up the superellipse. He chose $n = 2.5$ as the most pleasing bulge. Stockholm quickly accepted the superellipse motif for its new center. The curves were “strangely satisfying, neither too rounded nor too orthogonal, a happy blend of elliptical and rectangular beauty” [Gardner75, p. 243]. Since that time, superellipse shapes have appeared in furniture, textile patterns, and even silverware. More can be found out about them in the references, especially in [Gardner75] and [Hill 79b].

The **superhyperbola** can also be defined [Barr81]. Just replace $\cos(t)$ by $\sec(t)$, and $\sin(t)$ by $\tan(y)$, in Equation 3.18. When $n = 2$, the familiar hyperbola is obtained. Figure 3.75 shows example superhyperbolas. As the bulge n increases beyond 2, the curve bulges out more and more, and as it decreases below 2, it bulges out less and less, becoming straight for $n = 1$ and pinching inward for $n < 1$.

1st Ed. Figure 9.14.

Figure 3.75. The superhyperbola family.

3.8.4. Polar Coordinate Shapes

Polar coordinates may be used to represent many interesting curves. As shown in Figure 3.76, each point on the curve is represented by an angle θ and a radial distance r . If r and θ are each made a function of t , then as t varies the curve $(r(t), \theta(t))$ is swept out. Of course this curve also has the Cartesian representation $(x(t), y(t))$ where:

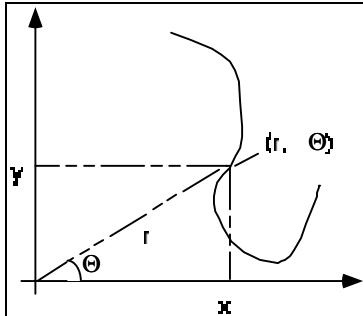


Figure 3.76. Polar coordinates.

$$\begin{aligned}x(t) &= r(t) \cos(\theta(t)) \\y(t) &= r(t) \sin(\theta(t)).\end{aligned}\tag{3.19}$$

But a simplification is possible for a large number of appealing curves. In these instances the radius r is expressed directly as a function of θ , and the parameter that “sweeps” out the curve is θ itself. For each point (r, θ) the corresponding Cartesian point (x, y) is given by

$$\begin{aligned}x &= f(\theta) \cdot \cos(\theta) \\y &= f(\theta) \cdot \sin(\theta)\end{aligned}\tag{3.20}$$

Curves given in polar coordinates can be generated and drawn as easily as any others: The parameter is θ , which is made to vary over an interval appropriate to the shape. The simplest example is a circle with radius K : $f(\theta) = K$. The form $f(\theta) = 2K \cos(\theta)$ is another simple curve (which one?). Figure 3.77 shows some shapes that have simple expressions in polar coordinates:

1st Ed. Figure 4.19

Figure 3.77. Examples of curves with simple polar forms..

- *Cardioid*: $f(\theta) = K(1 + \cos(\theta))$.
- *Rose curves*: $f(\theta) = K \cos(n \theta)$, where n specifies the number of petals in the rose. Two cases are shown.
- *Archimedean spiral*: $f(\theta) = K \cdot \theta$.

In each case, constant K gives the overall size of the curve. Because the cardioid is periodic, it can be drawn by varying θ from 0 to 2π . The rose curves are periodic when n is an integer, and the Archimedean spiral keeps growing forever as θ increases from 0. The shape of this spiral has found wide use as a cam to convert rotary motion to linear motion (see [Yates46] and [Seggern90]).

The **conic sections** (ellipse, parabola, and hyperbola) all share the following polar form:

$$f(\theta) = \frac{1}{1 \pm e \cdot \cos(\theta)}\tag{3.21}$$

where e is the eccentricity of the conic section. For $e = 1$ the shape is a parabola; for $0 \leq e < 1$ it is an ellipse; and for $e > 1$ it is a hyperbola.

• The Logarithmic Spiral

The **logarithmic spiral** (or “equiangular spiral”) $f(\theta) = Ke^{a\theta}$, shown in Figure 3.78a, is also of particular interest [Coxeter61]. This curve cuts all radial lines at a constant angle α , where $a = \cot(\alpha)$. This is the only spiral that has the same shape for any change of scale: Enlarge a photo of such a spiral any amount, and the enlarged

1st Ed Figures 4.20 and 4.21

Figure 3.78. The logarithmic spiral and b). chambered nautilus

spiral will fit (after a rotation) exactly on top of the original. Similarly, rotate a picture of an equiangular spiral, and it will seem to grow larger or smaller [Steinhaus69]⁹. This preservation of shape seems to be used by some animals such as the mollusk inside a chambered nautilus (see Figure 3.78b). As the animal grows, its shell also grows along a logarithmic spiral in order to provide a home of constant shape [Gardner61].

Other families of curves are discussed in the exercises and Case Studies, and an exhaustive listing and characterization of interesting curves is given in [yates46, seggern90, shikin95].

3.8.5. 3D Curves.

Curves that meander through 3D space may also be represented parametrically, and will be discussed fully in later chapters. To create a parametric form for a 3D curve we invent three functions $x(\cdot)$, $y(\cdot)$, and $z(\cdot)$, and say the curve is “at” $P(t) = (x(t), y(t), z(t))$ at time t .

Some examples are:

The helix: The circular helix is given parametrically by:

$$\begin{aligned} x(t) &= \cos(t) \\ y(t) &= \sin(t) \\ z(t) &= bt \end{aligned} \tag{3.22}$$

for some constant b . It is illustrated in Figure 3.79 as a stereo pair. See the Preface for viewing stereo pairs. If you find this unwieldy, just focus on one of the figures.

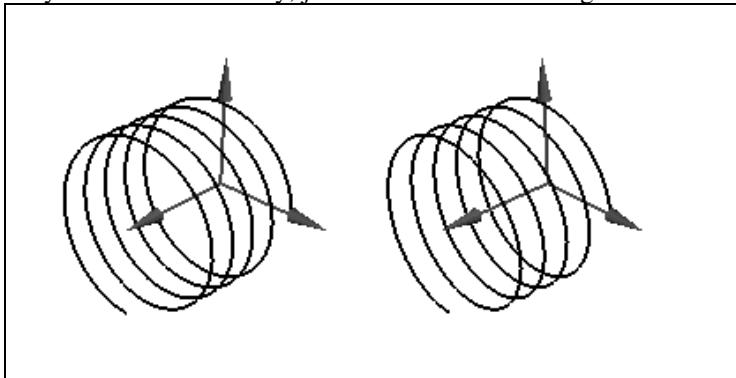


Figure 3.79. The helix, displayed as a stereo pair.

Many variations on the circular helix are possible, such as the elliptical helix $P(t) = (W \cos(t), H \sin(t), bt)$, and the conical helix $P(t) = (t \cos(t), t \sin(t), bt)$ (sketch these). Any 2D curve $(x(t), y(t))$ can of course be converted to a helix by appending $z(t) = bt$, or some other form for $z(t)$.

The toroidal spiral. A toroidal spiral, given by

$$\begin{aligned} x(t) &= (a \sin(ct) + b) \cos(t) \\ y(t) &= (a \sin(ct) + b) \sin(t) \\ z(t) &= a \cos(ct) \end{aligned} \tag{3.23}$$

⁹This curve was first described by Descartes in 1638. Jacob Bernoulli (1654---1705) was so taken by it that his tombstone in Basel, Switzerland, was engraved with it, along with the inscription *Eadem mutata resurgo*: “Though changed I shall arise the same.”

is formed by winding a string about a torus (doughnut). Figure 3.80 shows the case $c = 10$, so the string makes 10 loops around the torus. We examine tubes based on this spiral in Chapter 6.

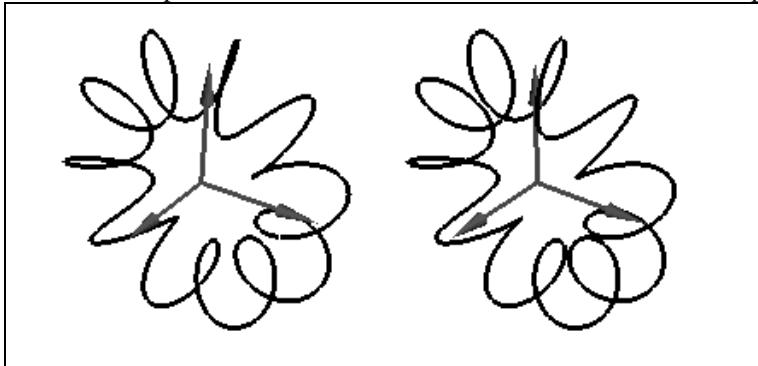


Figure 3.80. A toroidal spiral, displayed as a stereo pair.

Practice Exercises

3.8.6. Drawing superellipses. Write a routine `drawSuperEllipse(...)` that draws a superellipse. It takes as parameters c , the center of the superellipse, size parameters W and H , the bulge n , and m , the number of “samples” of the curve to use in fashioning the polyline approximation.

3.8.7. Drawing polar forms. Write routines to draw an n -petaled rose and an equiangular spiral.

3.8.8. Golden Cuts. Find the specific logarithmic spiral that makes “golden cuts” through the intersections of the infinite regression of golden rectangles, as shown in Figure 3.81 (also recall Chapter 2). How would a picture like this be drawn algorithmically?

1st Ed. Figure 4.22

Figure 3.81. The spiral and the golden rectangle.

3.8.9. A useful implicit form function. Define a suitable implicit form for the rose curve defined earlier in polar coordinate form: $f(\theta) = K \cos(n \theta)$.

3.8.10. Inside-outside functions for polar curves. Discuss whether there is a single method that will yield a suitable inside-outside function for *any* curve given in polar coordinate form as in Equation 3.20. Give examples or counter-examples.

3.9. Summary of the Chapter.

In this chapter we developed several tools that make it possible for the applications programmer to “think” and work directly in the most convenient “world” coordinate system for the problem at hand. Objects are defined (“modeled”) using high precision real coordinates, without concern for “where” or “how big” the picture of the object will be on the screen. These concerns are deferred to a later selection of a window and a viewport – either manually or automatically – that define both how much of the object is to be drawn, and how it is to appear on the display. This approach separates the modeling stage from the viewing stage, allowing the programmer or user to focus at each phase on the relevant issues, undistracted by details of the display device.

The use of windows makes it very easy to “zoom” in or out on a scene, or “roam” around to different parts of a scene. Such actions are familiar from everyday life with cameras. The use of viewports allows the programmer to place pictures or collections of pictures at the desired spots on the display in order to compose the final picture. We described techniques for insuring that the window and viewport have the same aspect ratio, in order to prevent distortion.

Clipping is a fundamental technique in graphics, and we developed a classical algorithm for clipping line segments against the world window. This allows the programmer to designate which portion of the picture will actually be rendered: parts outside the window are clipped off. OpenGL automatically performs this clipping, but in other environments a clipper must be incorporated explicitly.

We developed the *Canvas* class to encapsulate many underlying details, and provide the programmer with a single uniform tool for fashioning drawing programs. This class hides the OpenGL details in convenient

routines such as `setWindow()`, `setViewport()`, `moveTo()`, `lineTo()`, and `forward()`, and insures that all proper initializations are carried out. In a Case Study we implement `Canvas` for a more basic non-OpenGL environment, where explicit clipping and window-to-viewport mapping routines are required. Here the value of data-hiding within the class is even more apparent.

A number of additional tools were developed for performing relative drawing and turtle graphics, and for creating drawings that include regular polygons, arcs and circles. The parametric form for a curve was introduced, and shown to be a very natural description of a curve. It makes it simple to draw a curve, even those that are multi-valued, cross over themselves, or have regions where the curve moves vertically.

3.10. Case Studies.

One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.
Bertrand Russell

3.10.1. Case Study 3.1. Studying the Logistic Map and Simulation of Chaos.

(Level of Effort: II) Iterated function systems (IFS's) were discussed at the end of Chapter 2. Another IFS provides a fascinating look into the world of **chaos** (see [Gleick87, Hofs85]), and requires proper setting of a window and viewport. A sequence of values is generated by the repeated application of a function $f(\cdot)$, called the **logistic map**. It describes a parabola:

$$f(x) = 4 \lambda x (1 - x) \quad (3.24)$$

where λ is some chosen constant between 0 and 1. Beginning at a given starting point, x_0 , between 0 and 1, function $f(\cdot)$ is applied iteratively to generate the **orbit** (recall its definition in Chapter 2):

$$x_k = f^{[k]}(x_0)$$

How does this sequence behave? A world of complexity lurks here. The action can be made most vivid by displaying it graphically in a certain fashion, as we now describe. Figure 3.82 shows the parabola $y = 4 \lambda x (1 - x)$ for $\lambda = 0.7$ as x varies from 0 to 1.

1st Ed. Figure 3.28

Figure 3.82. The logistic map for $\lambda = 0.7$.

The starting point $x_0 = 0.1$ is chosen here, and at $x = 0.1$ a vertical line is drawn up to the parabola, showing the value $f(x_0) = 0.252$. Next we must apply the function to the new value $x_1 = 0.252$. This is shown visually by moving horizontally over to the line $y = x$, as illustrated in the figure. Then to evaluate $f(\cdot)$ at this new value a line is again drawn up vertically to the parabola. This process repeats forever as in other IFS's. From the previous position (x_{k-1}, x_k) a horizontal line is drawn to (x_k, x_k) from which a vertical line is drawn to (x_k, x_{k+1}) . The figure shows that for $\lambda = 0.7$, the values quickly converge to a stable “attractor,” a fixed point so that $f(x) = x$. (What is its value for $\lambda = 0.7$? This attractor does not depend on the starting point; the sequence always converges quickly to a final value.)

If λ is set to small values, the action will be even simpler: There is a single attractor at $x = 0$. But when the “ λ -knob” is increased, something strange begins to happen. Figure 3.83a shows what results when $\lambda = 0.85$. The “orbit” that represents the sequence falls into an endless repetitive cycle, never settling down to a final value. There are several attractors here, one at each vertical line in the limit cycle shown in the figure. And when λ is increased beyond the critical value $\lambda = 0.892486418\dots$ the process becomes truly chaotic.

1st Ed. Figure 3.29

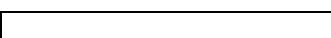


Figure 3.83. The logistic map for a). $\lambda = 0.85$ and b). $\lambda = 0.9$.

The case of $\lambda = 0.9$ is shown in Figure 3.83b. For most starting points the orbit is still periodic, but the number of orbits observed between the repeats is extremely large. Other starting points yield truly aperiodic motion, and very small changes in the starting point can lead to very different behavior. Before the truly remarkable character of this phenomenon was first recognized by Mitchell Feigenbaum in 1975, most researchers believed that very small adjustments to a system should produce correspondingly small changes in its behavior and that simple systems such as this could not exhibit arbitrarily complicated behavior. Feigenbaum's work spawned a new field of inquiry into the nature of complex nonlinear systems, known as chaos theory [Gleick87]. It is intriguing to experiment with this logistic map.

Write and exercise a program that permits the user to study the behavior of repeated iterations of the logistic map, as shown in Figure 3.83. Set up a suitable window and viewport so that the entire logistic map can be clearly seen. The user gives the values of x_0 and λ and the program draws the limit cycles produced by the system.

3.10.2. Case Study 3.2. Implementation of the Cohen Sutherland Clipper in C/C++.

(Level of Effort: II) The basic flow of the Cohen Sutherland algorithm was described in Section 3.3.2. Here we flesh out some details of its implementation in C or C++, exploiting for efficiency the low-level bit manipulations these languages provide.

We first need to form the “inside-outside” code words that report how a point P is positioned relative to the window (see Figure 3.20). A single 8-bit word `code` suffices: four of its bits are used to capture the four pieces of information. Point P is tested against each window boundary in turn; if it lies outside this boundary, the proper bit of `code` is set to 1 to represent TRUE. Figure 3.84 shows how this can be done. `code` is initialized to 0, and then its individual bits are set as appropriate using a bit-wise OR operation. The values 8, 4, 2, and 1 are simple masks. For instance, since 8 in binary is 00001000, bit-wise OR-ing a value with 8 sets the fourth bit from the right end to 1.

```
unsigned char code = 0;           // initially all bits are 0
...
if(P.x < window.l)      code |= 8;      // set bit 3
if(P.y > window.t)      code |= 4;      // set bit 2
if(P.x > window.r)      code |= 2;      // set bit 1
if(P.y < window.b)      code |= 1;      // set bit 0
```

Figure 3.84. Setting bits in the “inside-outside code word” for a point P .

In the clipper both endpoints P_1 and P_2 (see Figure 3.22) are tested against the window, and their code words `code1` and `code2` are formed. We then must test for “trivial accept” and “trivial reject”.

- **trivial accept:** Both endpoints are inside, so both codes `code1` and `code2` are identically 0. In C/C++ this is quickly determined using the bit-wise OR: a trivial accept occurs if `(code1 | code2)` is 0.

- **trivial reject:** A trivial reject occurs if both endpoints lie outside the window *on the same side*: both to the left of the window, both above, both below, or both to the right. This is equivalent to their codes having at least one 1 in the *same* bit position. For instance if `code1` is 0110 and `code2` is 0100 then P_1 lies both above and to the right of the window, while P_2 lies above but neither to the left nor right. Since both points lie above, no part of the line can lie inside the window. So trivial rejection is easily tested using the bit-wise AND of `code1` and `code2`: if they have some 1 in the same position then `code1 & code2` does also, and `(code1 & code2)` will be nonzero.

Chopping when there is neither trivial accept nor reject.

Another implementation issue is efficient chopping of the portion of a line segment that lies outside the window, as in Figure 3.22. Suppose it is known that point P with code word `code` lies outside the window. The

individual bits of `code` can be tested to see on which side of the window `P` lies, and the chopping can be accomplished as in Equation 3.5. Figure 3.85 shows a chop routine that finds the new point (such as A in Figure 3.22) and replaces `P` with it. It uses the bit-wise AND of `code` with a mask to determine where `P` lies relative to the window.

```
ChopLine(Point2 &P, unsigned char code)
{
    if(code & 8){      // to the Left
        P.y += (window.l - P.x) * dely / delx;
        P.x = window.l;
    }
    else if(code & 2){      // to the Right
        P.y += (window.r - P.x) * dely / delx;
        P.x = window.r;
    }
    else if(code & 1){      // below
        P.x += (window.b - P.y) * delx / dely;
        P.y = window.b;
    }
    else if(code & 4){      // above
        P.x += (window.t - P.y) * delx / dely;
        P.y = window.t;
    }
}
```

Figure 3.85. Chopping the segment that lies outside the window.

Write a complete implementation of the Cohen Sutherland algorithm, putting together the pieces described here with those in Section 3.3.2. If you do this in the context of a *Canvas* class implementation as discussed in the next Case Study, consider how the routine should best access the private data members of the window and the points involved, and develop the code accordingly.

Test the algorithm by drawing a window and a large assortment of randomly chosen lines , showing the parts that lie inside the window in red, and those that lie outside in black.

Practice Exercises.

3.10.1. Why will a “divide by zero” never occur? Consider a *vertical* line segment such that `delx` is zero. Why is the code `P.y += (window.l - P.x) * dely / delx` that would cause a divide by zero never reached? Similarly explain why each of the four statements that compute `delx/dely` or `dely/delx` are never reached if the denominator happens to be zero.

3.10.2. Do two chops in the same iteration? It would seem to improve performance if we replaced lines such as “`else if(code & 2)`” with “`if(c & 2)`” and tried to do two line “chops” in succession. Show that this can lead to erroneous endpoints being computed, and hence to disaster.

3.10.3. Case Study 3.3. Implementing Canvas for Turbo C++.

(Level of Effort: III) It is interesting to develop a drawing class like *Canvas* in which all the details are worked out, to see how the many ingredients go together. Sometimes it is even necessary to do this, as when a supporting library like OpenGL is not available. We design *Canvas* here for a popular graphics platform that uses Borland’s Turbo C++.

We want an implementation of the *Canvas* class that has essentially the same interface as that in Figure 3.25. Figure 3.86 shows the version we develop here (omitting parts that are simple repeats of Figure 3.25). The constructor takes a desired width and height but no title, since Turbo C++ does not support titled screen windows. There are several new private data members that internally manage clipping and the window to viewport mapping.

```
class Canvas {
public:
    Canvas(int width, int height); // constructor
    setWindow(), setViewport(), lineTo(), etc .. as before
```

```
private:
    Point2 CP;           // current position in the world
    IntRect viewport;   // the current window
    RealRect window;    // the current viewport
    float mapA, mapB, mapC, mapD; // data for the window to viewport mapping
    void makeMap(void); // builds the map
    int screenWidth, screenHeight;
    float delx, dely;      // increments for clipper
    char code1, code2;     // outside codes for clipper
    void ChopLine(tPoint2 &p, char c);
    int clipSegment(tPoint2 &p1, tPoint2 &p2);
};
```

Figure 3.86. Interface for the Canvas class for Turbo C++.

Implementation of the *Canvas* class.

We show some of the *Canvas* member functions here, to illustrate what must be done to manage the window to viewport mapping and clipping ourselves.

1). The Canvas constructor.

The constructor is passed the desired width and height of the screen. Turbo C++ is placed in graphics mode at the highest resolution supported by the graphics system. The actual screen width and height available is tested, and if it is less than was requested, the program terminates. Then a default window and viewport are established, and the window to viewport mapping is built (inside `setViewport()`.)

```
Canvas:: Canvas(int width, int height)
{
    int gdriver = DETECT, gmode; //Turbo C++ : use best resolution screen
    initgraph(&gdriver, &gmode, ""); // go to "graphics" mode
    screenWidth = getmaxx() + 1; // size of available screen
    screenHeight = getmaxy() + 1;
    assert(screenWidth >= width); // as wide as asked for?
    assert(screenHeight >= height); // as high as asked for?
    CP.set(0.0, 0.0);
    window.set(-1.0,1.0,-1.0,1.0); // default window
    setViewport(0,screenWidth, 0, screenHeight); // sets default map too
}
```

2). Setting the window and viewport and the mapping.

Whenever either the window or viewport is set, the window to viewport mapping is updated to insure that it is current. A degenerate window of zero height causes an error. The mapping uses window and viewport data to compute the four coefficients A , B , C , and D required.

```
//<<<<<<<<< set Window >>>>>>>>>>>>>
void Canvas:: setWindow(float l, float r, float b, float t)
{
    window.set(l, r, b, t);
    assert(t != b); //degenerate !
    makeMap(); // update the mapping
}
//<<<<<<<<< setViewport >>>>>>>>>>>>>
void Canvas:: setViewport(int l, int r, int b, int t)
{
    viewport.set(l, r, b, t);
    makeMap(); // update the mapping
}
//<<<<<<<< makeMap >>>>>>>>>>>>>
void Canvas:: makeMap(void)
{           // set mapping from window to viewport
```

```

intRect vp = getViewport(); // local copy of viewport
RealRect win = getWindow(); // local copy of window
float winWid = win.r - win.l;
float winHt = win.t - win.b;
assert(winWid != 0.0); assert(winHt != 0.0); // degenerate!
mapA = (vp.r - vp.l)/winWid; // fill in mapping values
mapB = vp.l - map.A * win.l;
mapC = (vp.t - vp.b)/winHt;
mapD = vp.b - map.B * win.b;
}

```

3). `moveTo()`, and `lineTo()` with clipping.

The routine `moveTo()` converts its point from world coordinates to screen coordinates, and calls the Turbo C++ specific `moveto()` to update the internal current position maintained by Turbo C++. It also updates *Canvas*' world coordinate *CP*. Routine `lineTo()` works similarly, but it must first determine which part if any of the segment lies within the window. To do this it uses `clipSegment()` described in Section 3.3 and in Case Study 3.2, which returns the `first` and `second` endpoints of the inside portion. If so it moves to `first` and draws a line to `second`. It finishes with a `moveTo()` to insure that the *CP* will be current (both the *Canvas CP* and the internal Turbo C++ *CP*).

`ChopLine` and `ClipSegment` are same as in Case Study 3.2.

```

//<<<<<<<<<<<< moveTo >>>>>>>>>>>>
void Canvas:: moveTo(float x, float y)
{
    int sx = (int)(mapA * x + mapC);
    int sy = (int)(mapB * y + mapD);
    moveto(sx, sy); // a Turbo C++ routine
    CP.set(x, y);
}

//<<<<<<<<<<<< lineTo >>>>>>>>>
void Canvas:: lineTo(float x, float y)
{ // Draw a line from CP to (x,y), clipped to the window
    Point2 first = CP; // initial value of first
    Point2 second(x, y); // initial value of second
    if(clipSegment(first, second)) // any part inside?
    {
        moveTo(first.x, first.y); // to world CP
        int sx = (int)(mapA * second.x + mapC);
        int sy = (int)(mapB * second.y + mapD);
        lineto(sx,sy); // a Turbo C++ routine
    }
    moveTo(x, y); // update CP
}

```

Write a full implementation of the `Canvas` class for Turbo C++ (or a similar environment that requires you to implement clipping and mapping). Cope appropriately with setting the drawing and background colors (this is usually quite system-specific). Test your class by using it in an application that draws polyspirals as specified by the user.

3.10.4. Case Study 3.4. Drawing Arches.

(Level of Effort: II) Arches have been used throughout history in architectural compositions. Their structural strength and ornamental beauty make them very important elements in structural design, and a rich variety of shapes have been incorporated into cathedrals, bridges, doorways, etc.

Figure 3.87 shows two basic arch shapes. The arch in part a) is centered at the origin, and has a width of $2W$. The arch begins at height H above the base line. Its principal element is a half-circle with a radius $R = W$. The ratio H/W can be adjusted according to taste. For instance, H/W might be related to the golden ratio.

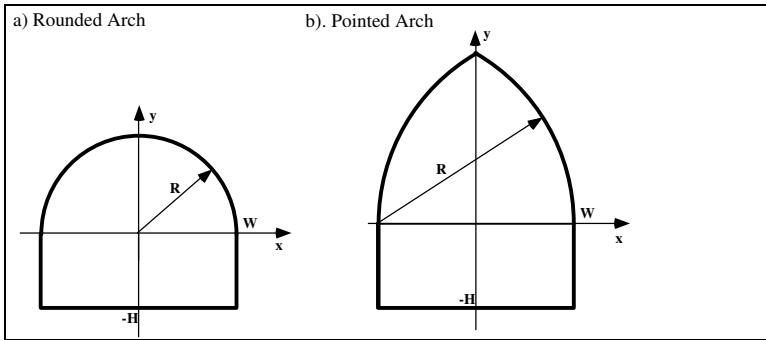


Figure 3.87. Two basic arch forms.

Figure 3.73b shows an idealized version of the second most famous arch shape, the **pointed** or “equilateral” arch, often seen in cathedrals¹⁰. Here two arcs of radius $R = 2W$ meet directly above the center. (Through what angle does each arc sweep?)

The **ogee**¹¹ (or “keel”) arch is shown in Figure 3.88. This arch was introduced about 1300 AD, and was popular in architectural structures throughout the late Middle Ages. Circles of radius fR rest on top of a rounded arch of radius R for some fraction f . This fixes the position of the two circles. (What are the coordinates of point C ?) On each side two arcs blend together to form a smooth pointed top. It is interesting to work out the parameters of the various arcs in terms of W and f .

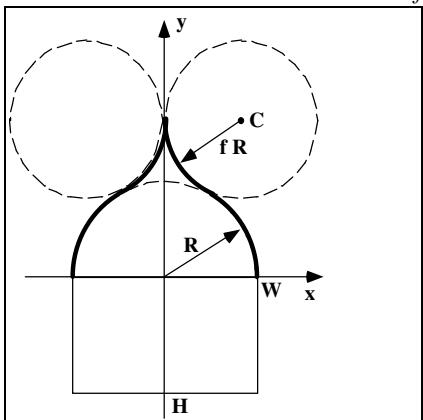


Figure 3.88. The Ogee arch.

Develop routines that can draw each of the arch types described above. Also write an application that draws an interesting collection of such arches in a castle, mosque, or bridge of your design.

3.10.5. Case Study 3.5. Some Figures used in Physics and Engineering.

(Level of Effort: II) This Case Study works with a collection of interesting pictures that arise in certain topics within physics and engineering. The first illustrates a physical principal of circles intersecting at right angles; the second creates a chart that can be used to study electromagnetic phenomena; the third develops symbols that are used in designing digital systems.

1). Electrostatic Fields. The pattern of circles shown in Figure 3.89 is studied in physics and electrical engineering, as the electrostatic field lines that surround electrically charged wires. It also appears in mathematics in connection with the analytic functions of a complex variable. In Chapter 5 these families also are found when we examine a fascinating set of transformations, “inversions in a circle.” Here we view them simply as an elegant array of circles and consider how to draw them.

¹⁰From J. Fleming, H. Honour, N. Pevsner: **Dictionary of Architecture**. Penguin Books, London 1980

¹¹From the old French *ogive* meaning an S-shaped curve.

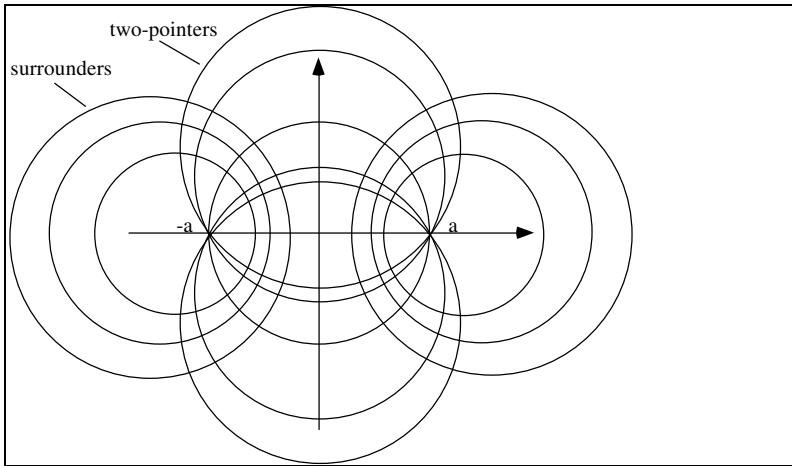


Figure 3.89. Families of orthogonal circles..

There are two families of circles, which we will call “two-pointers” and “surrounders”. The two-pointers family consists of circles that pass through two given points. Suppose the two points are $(-a, 0)$ and $(a, 0)$. The two-pointers can be distinguished by some parameter m , and for each value of m two different circles are generated (see Figure 3.75). The circles have centers and radii given by:

$$\text{center} = (0, \pm a \sqrt{m^2 - 1}) \quad \text{and} \quad \text{radius} = am$$

as m varies from 1 to infinity.

Circles in the surrounders family surround one of the points $(-a, 0)$ or $(a, 0)$. The centers and radii of the surrounders are also distinguished by a parameter n and have the values

$$\text{center} = (\pm an, 0) \quad \text{and} \quad \text{radius} = a\sqrt{n^2 - 1}$$

as n varies from 1 to infinity. The surrounders circles are also known as “circles of Appolonius,” and they arise in problems of pursuit [Ball & Coxeter]. The distances from any point on a circle of Appolonius to the points $(-a, 0)$ and $(a, 0)$ have a constant ratio. (What is this ratio in terms of a and n ?)

The “surrounder” family is intimately related to the two-pointer family: Every surrounders circle “cuts” through every two-pointer circle at a right angle. The families of circles are thus said to be **orthogonal** to one another.

Write and exercise a program that draws the two families of orthogonal circles. Choose sets of values of m and n so that the picture is well balanced and pleasing.

2). Smith Charts. Another pattern of circles is found in Smith charts, familiar in electrical engineering in connection with electromagnetic transmission lines. Figure 3.90 shows the two orthogonal families found in Smith charts. Here all members of the families pass through a common point $(1, 0)$. Circles in family A have centers at $(1 - m, 0)$ and radii m , and circles in family B have centers at $(1, \pm n)$ and radii n , where both m and n vary from 0 to π . Write and exercise a program that draws these families of circles.

1st Ed. Figure 4.31

Figure 3.90. The Smith Chart.

3). Logic Gates for Digital Circuits. Logic gates are familiar to scientists and engineers who study basic electronic circuits found in computers. Each type of gate is symbolized in a circuit diagram by a characteristic shape, several of which are based on arcs of circles. Figure 3.91a shows the shape of the so-called

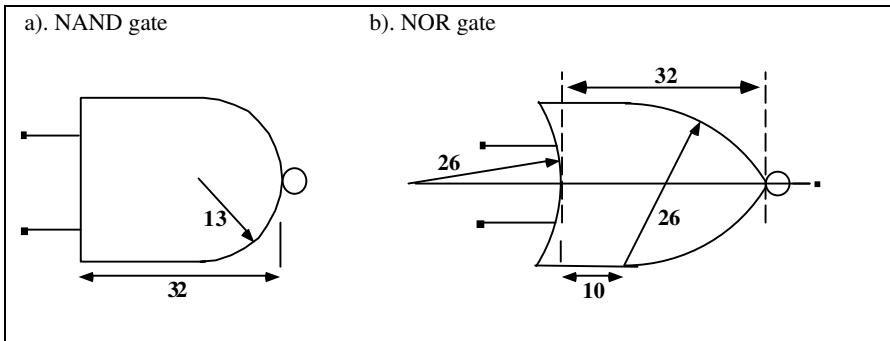


Figure 3.91. Standard Graphic Symbol for the Nand and Nor Gates.

NAND gate, according to a world-wide standard¹². The NAND gate is basically a rounded arch placed on its side. The arc has radius 13 units relative to the other elements, so the NAND gate must be 26 units in height.

Figure 3.91b shows the standard symbol for a NOR gate. It is similar to a pointed arch turned on its side. Three arcs are used, each having a radius of 26 units. (The published standard as shown has an error in it, that makes it impossible for certain elements to fit together. What is the error?)

Write a program that can draw both of these circuit types at any size and position in the world. (For the NOR gate find and implement a reasonable correction to the error in Figure 3.77b.) Also arrange matters so that your program can draw these gates rotated by 90° , 180° , or 270° .

3.10.6. Case Study 3.6. Tilings.

(Level of Effort: II) Computer graphics offers a powerful tool for creating pleasing pictures based on geometric objects. One of the most intriguing types of pictures are those that apparently repeat forever in all directions. They are called variously **tilings**, and **repeat patterns**. They are studied in greater detail in Chapter ???.

A). Basic Tilings. Figure 3.92 shows a basic tiling. A **motif**, in this case four quarter circles in a simple arrangement, is designed in a square region of the world. To draw a tiling over the plane based on this motif, a collection of viewports are created side by side that cover the display surface, and the motif is drawn once inside each viewport.

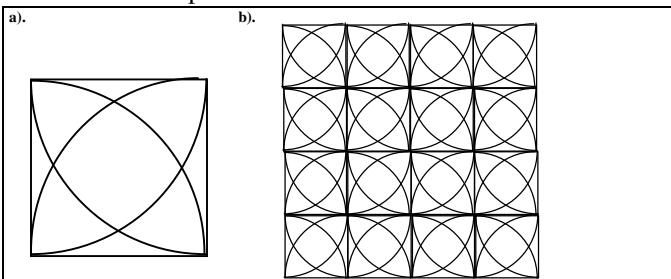


Figure 3.92. A motif and the resulting tiling.

Write a program that:

- chooses a square window in the world, and draws some interesting motif (possibly clipping portions of it, as in Figure 3.14);
- successively draws the picture in a set of viewports that abut one another and together cover the display surface.

Exercise your program with at least two motifs.

¹²The Institute of Electrical and Electronic Engineers (IEEE) publishes many things, including standard definitions of terminology and graphic shapes of circuit elements. These drawings are taken from the standard document: IEEE Std. 91-1984 .

B). Truchet Tiles. A slight variation of the method above selects successive motifs randomly from a “pool” of candidate motifs. Figure 3.93a shows the well-known Truchet tiles¹³, which are based on two quarter circles centered at opposite corners of a square. Tile 0 and tile 1 differ only by a 90° rotation.

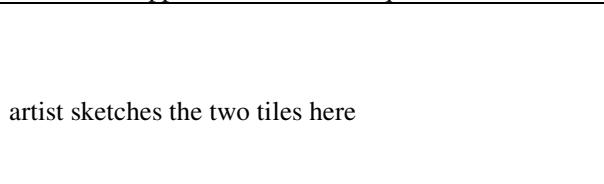


Figure 3.93. Truchet Tiles. a). the two tiles. b). A truchet pattern

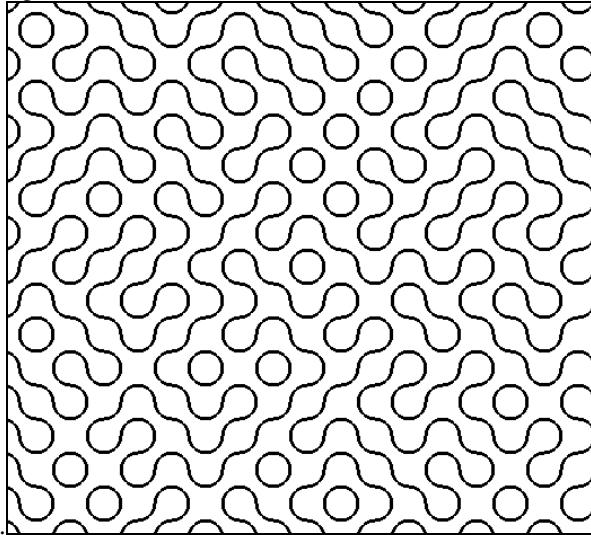


Figure 3.93.b.

Write an application that draws Truchet tiles over the entire viewport. Each successive tile uses tile 0 or tile 1, selected at random.

Curves other than arcs can be used as well, as suggested in Figure 3.94. What conditions should be placed on the angle with which each curve meets the edge of the tile in order to avoid sharp corners in the resulting curve? This notion can also be extended to include more than two tiles.

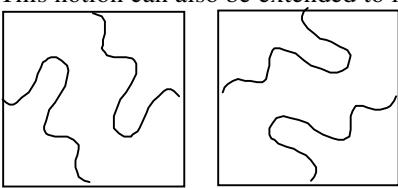


Figure 3.94. Extension of Truchet tiles.

Extend the program above so that it introduces random selections of two or more motifs, and exercise it on the motifs you have designed. Design motifs that “blend” together properly.

3.10.7. Case Study 3.7. Playful Variations on a Theme.

(Estimate of time required: four hours). In Section 3.8 we discussed how to draw a curve represented parametrically by $P(t)$: take a succession of instants $\{t_i\}$ and connect the successive "samples" $(x(t_i), y(t_i))$ by straight lines. A wide range of pictures can be created by varying the way in which the samples are taken. We suggest some possibilities here.

¹³Smith, C. “The Tiling Patterns of Sebastian Truchet and the topology of structural hierarchy.” Leonardo, 20:4, pp: 373-385, 1987. (refd in Pickover, p.386)

Write a program that draws each of the four shapes:

- a). an ellipse
- b). a hyperbola
- c). a logarithmic spiral
- d). a 5-petal rose curve

for each of the methods described below for obtaining t-samples.

1). Unevenly Spaced Values of t. Instead of using a constant increment between values of t when sampling the functions $x()$ and $y()$, use a varying increment. It is interesting to experiment with different choices to see what visual effects can be achieved. Some possibilities for a sequence of $(n+1)$ t -values between 0 and T (suitably chosen for the curve shape at hand) are

- $t_i = T\sqrt{i/n}$: The samples cluster closer and closer together as i increases.
- $t_i = T(i/n)^2$: The samples spread out as i increases.
- $t_i = T(i/n) + A \sin(ki/n)$ The samples cyclically cluster together or spread apart. Constants A and k are chosen to vary the amount and speed of the variation.

2). Randomly Selected t-Values. The t -values can be chosen randomly as in

$$\bullet t_i = \text{randChoose}(0, T)$$

Here $\text{randChoose}(0, T)$ is a function (devised by you) that returns a value randomly selected from the range 0 to T each time it is called. (See Appendix 3 for a basic random number generator.)

Figure 3.95 shows the polyline generated in this fashion for points on an ellipse. It is interesting to watch such a picture develop on a display. A flurry of seemingly unrelated lines first appears, but soon the eye detects some order in the chaos and “sees” an elliptical “envelope” emerging around the cloud of lines.

1st Ed. Figure 4.26

Figure 3.95. A random ellipse polyline.

Alternatively, a sequence of increasing t-values can be used, generated by

$$\bullet t_i = t_{i-1} + \text{randChoose}(0, r)$$

where r is some small positive value.

3). Connecting Vertices in Different Orders

In a popular children’s game, pins are driven into a board in some pattern, and a piece of thread is woven around the pins in some order. The t -values here define the positions of the pins in the board, and `worldLineTo()` plays the role of the thread.

The samples of $P(t)$ are prestored in a suitable array $P[i]$, $i = 0, 1, \dots, n$. The polyline is drawn by sequencing in an interesting way through values of i . That is, the sequence i_0, i_1, \dots is generated from values between 0 and n , and for each index i_k a call to `worldLineTo(P[i_k])` is made. Some possibilities are:

- “Random Deal”: the sequence i_0, i_1, \dots is a random permutation of the values 0,1,..,n, as in dealing a fixed set of cards from a shuffled deck.
- Every pair of points is connected by a straight line. So every pair of values in the range 0,1,..,n appears in adjacent spots somewhere in the sequence i_0, i_1, \dots . The prime rosette of Chapter 5 gave one example, where lines were drawn connecting each point to every other.

- One can also draw “webs,” as suggested in Figure 3.96. Here the index values cycle many times through the possible values, skipping by some M each time. This is easily done by forming the next index from the previous one using $i = (i + M) \bmod (n+1)$.

1st Ed. Figure 4.27

Figure 3.96. Adding webs to a curve.

3.10.8. Case Study 3.8. Circles Rolling around Circles.

(Level of Effort: II) Another large family of interesting curves can be useful in graphics. Consider the path traced by a point rigidly attached to a circle as the circle rolls around another fixed circle [thomas53, Yates46]. These are called **trochoids**, and Figure 3.97 shows how they are generated. The tracing point is attached to the rolling circle (of radius b) at the end of a rod k units from the center. The fixed circle has radius a . There are two basic kinds: When the circle rolls externally (Figure 3.97a), an **epitrochoid** is generated, and when it rolls internally (Figure 3.97b), a **hypotrochoid** is generated. The children’s game Spirograph¹⁴ is a familiar tool for drawing trochoids, which have the following parametric forms:

1st Ed. Figure 4.23

Figure 3.97. Circles rolling around circles.

The epitrochoid:

$$\begin{aligned}x(t) &= (a+b)\cos(2\pi t) - k\cos\left(2\pi\frac{(a+b)t}{b}\right) \\y(t) &= (a+b)\sin(2\pi t) - k\sin\left(2\pi\frac{(a+b)t}{b}\right)\end{aligned}\tag{3.24}$$

The hypotrochoid:

$$\begin{aligned}x(t) &= (a-b)\cos(2\pi t) + k\cos\left(2\pi\frac{(a-b)t}{b}\right) \\y(t) &= (a-b)\sin(2\pi t) - k\sin\left(2\pi\frac{(a-b)t}{b}\right)\end{aligned}\tag{3.25}$$

An ellipse results from the hypotrochoid when $a = 2b$ for any k .

When the tracing point lies on the rolling circle ($k = b$) these shapes are called **cycloids**. Some familiar special cases of cycloids are

Epicycloids:

$$\begin{aligned}\text{Cardioid: } &b = a \\ \text{Nephroid: } &2b = a\end{aligned}$$

Hypocycloids:¹⁵

$$\begin{aligned}\text{Line segment: } &2b = a \\ \text{Deltoid: } &3b = a \\ \text{Astroid: } &4b = a.\end{aligned}$$

Some of these are shown in Figure 3.98. Write a program that can draw both epitrochoids and hypotrochoids. The user can choose which family to draw, and can enter the required parameters. Exercise the program to draw each of the special cases listed above.

1st Ed. Figure 4.24.

Figure 3.98. Examples of cycloids: a) nephroid, b) $a/b = 10$, c) deltoid, d) astroid.

¹⁴A trademark of Kenner Products.

¹⁵Note that the astroid is also a superellipse! It has a bulge of 2/3.

3.10.9. Case Study 3.9. Superellipses.

(Level of Effort: I) Write and exercise a program to draw superellipses. To draw each superellipse, the user indicates opposite corners of its bounding, and types a value for the bulge, whereupon the specified superellipse is drawn.

(Optional). Extend the program so that it can draw rotated superellipses. The user types an angle after typing the bulge.

3.11. For Further Reading.

When getting started with graphics it is very satisfying to write applications that produce fascinating curves and patterns. This leads you to explore the deep connection between mathematics and the visual arts. Many books are available that offer guidance and provide myriad examples. McGregor and Watt's THE ART OF GRAPHICS FOR THE IBM PC, [mcgregor86] offers many algorithms for creating interesting patterns. Some particularly noteworthy books on curves and geometry are Jay Kapraff's CONNECTIONS [kapraff91], Dewdney's THE ARMCHAIR UNIVERSE [dewdney88], Stan Ogilvy's EXCURSIONS IN GEOMETRY[ogilvy69], Pedoe's GEOMETRY AND THE VISUAL ARTS [pedoe76], Roger Sheperd's MIND SIGHTS [shep90], and the series of books on mathematical excursions by Martin Gardner, (such as TIME TRAVEL [gardner88] and PENROSE TILES TO TRAPDOOR CIPHERS [gardner89]). Coxeter has written elegant books on geometry, such as INTRODUCTION TO GEOMETRY[Coxeter69] and MATHEMATICAL RECREATIONS AND ESSAYS [ball74], and Hoggar's MATHEMATICS FOR COMPUTER GRAPHICS [hoggar92] discusses many features of iterated function systems.

(for ECE660, Fall, 1999)

Chapter 4. Vectors Tools for Graphics.

*“The knowledge at which geometry aims is knowledge of the eternal,
and not of aught perishing and transient.”*

Plato

*For us, whose shoulders sag under the weight of the heritage of Greek thought
and who walk in the paths traced out by the heroes of the Renaissance,
a civilization without mathematics is unthinkable.*

Andre Weil

“Let us grant that the pursuit of mathematics is a divine madness of the human spirit.”

Alfred North Whitehead

“All that transcend geometry, transcends our comprehension”.

Blaise Pascal

Goals of the Chapter

- To review vector arithmetic, and to relate vectors to objects of interest in graphics.
- To relate geometric concepts to their algebraic representations.
- To describe lines and planes parametrically.
- To distinguish points and vectors properly.
- To exploit the dot product in graphics topics.
- To develop tools for working with objects in 3D space, including the cross product of two vectors.

Preview

This chapter develops a number of useful tools for dealing with geometric objects encountered in computer graphics. Section 4.1 motivates the use of vectors in graphics, and describes the principal coordinate systems used. Section 4.2 reviews the basic ideas of vectors, and describes the key operations that vectors allow. Although most results apply to any number of dimensions, vectors in 2D and 3D are stressed. Section 4.3 reviews the powerful dot product operation, and applies it to a number of geometric tasks, such as performing orthogonal projections, finding the distance from a point to a line, and finding the direction of a ray “reflected” from a shiny surface. Section 4.4 reviews the cross product of two vectors, and discusses its important applications in 3D graphics.

Section 4.5 introduces the notion of a coordinate frame and homogeneous coordinates, and stresses that points and vectors are significantly different types of geometric objects. It also develops the two principal mathematical representations of a line and a plane, and shows where each is useful. It also introduces affine combinations of points and describes an interesting kind of animation known as “tweening”. A preview of Bezier curves is described as an application of tweening.

Section 4.6 examines the central problem of finding where two line segments intersect, which is vastly simplified by using vectors. It also discusses the problem of finding the unique circle determined by three points. Section 4.7 discusses the problem of finding where a “ray” hits a line or plane, and applies the notions to the clipping problem. Section 4.8 focuses on clipping lines against convex polygons and polyhedra, and develops the powerful Cyrus-Beck clipping algorithm.

The chapter ends with Case Studies that extend these tools and provide opportunities to enrich your graphics programming skills. Tasks include processing polygons, performing experiments in 2D “ray tracing”, drawing rounded corners on figures, animation by tweening, and developing advanced clipping tools.

4.1 Introduction.

In computer graphics we work, of course, with objects defined in a three dimensional world (with 2D objects and worlds being just special cases). All objects to be drawn, and the “cameras” used to draw them, have shape, position, and orientation. We must write computer programs that somehow describe these objects, and describe how light bounces around illuminating them, so that the final pixel values on the display can be computed.

Think of an animation where a camera flies through a hilly scene containing various buildings, trees, roads, and cars. What does the camera “see”? It all has to be converted ultimately to numbers. It’s a tall order.

The two fundamental sets of tools that come to our aid in graphics are *vector analysis* and *transformations*. By studying them in detail we develop methods to describe the various geometric objects we will encounter, and we learn how to convert geometric ideas to numbers. This leads to a collection of crucial algorithms that we can call upon in graphics programs.

In this chapter we examine the fundamental operations of vector algebra, and see how they are used in graphics; transformations are addressed in Chapter 5. We start at the beginning and develop a number of important tools and methods of attack that will appear again and again throughout the book. If you have previously studied vectors much of this chapter will be familiar, but the numerous applications of vector analysis to geometric situations should still be scrutinized. The chapter might strike you as a mathematics text. But having it all collected in one place, and related to the real problems we encounter in graphics, may be found useful.

Why are vectors so important?

A preview of some of some situations where vector analysis comes to the rescue might help to motivate the study of vectors. Figure 4.1 shows three geometric problems that arise in graphics. Many other examples could be given.

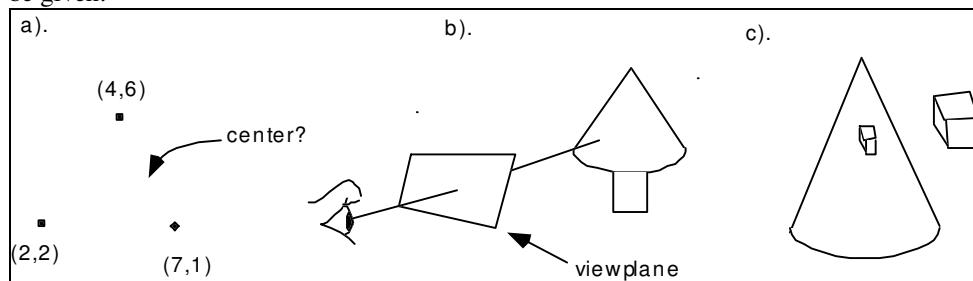


Figure 4.1. Three sample geometric problems that yield readily to vector analysis.

Part a) shows a computer-aided design problem: the user has placed three points on the display with the mouse, and wants to draw the unique circle that passes through them. (Can you visualize this circle?). For the coordinates given where is the center of the circle located? We see in Section 4.6 that this problem is thorny without the use of vectors, but almost trivial when the right vector tools are used.

Part b) shows a camera situated in a scene that contains a Christmas tree. The camera must form an image of the tree on its “viewplane” (similar to the film plane of a physical camera), which will be transferred to a screen window on the user’s display. Where does the image of the tree appear on this plane, and what is its exact shape? To answer this we need a detailed study of perspective projections, which will be greatly aided by the use of vector tools. (If this seems too easy, imagine that you are developing an animation, and the camera is zooming in on the sphere along some trajectory, and rotating as it does so. Write a routine that generates the whole sequence of images!)

Part c) shows a shiny cone in which the reflection of a cube can be seen. Given the positions of the cone, cube, and viewing camera, where *exactly* does the reflected image appear, and what is its color and shape? When studying ray tracing in Chapter 15 we will make extensive use of vectors, and we will see that this problem is readily solved.

Some Basics.

All points and vectors we work with are defined relative to some coordinate system. Figure 4.2 shows the coordinate systems that are normally used. Each system has an *origin* called ϑ and some axes emanating from ϑ . The axes are usually oriented at right angles to one another. Distances are marked along each axis, and a

point is given coordinates according to how far along each axis it lies. Part a) shows the usual two-dimensional system. Part b) shows a *right handed* 3D coordinate system, and part c) shows a *left handed* coordinate system.

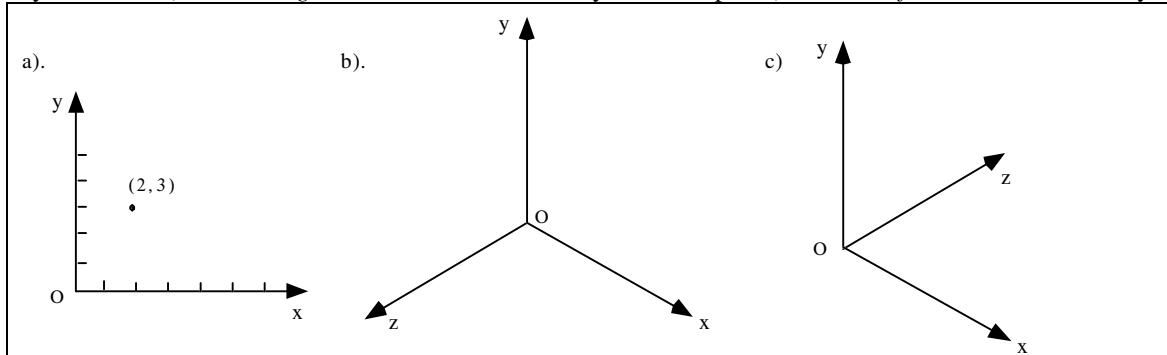


Figure 4.2. The familiar two- and three-dimensional coordinate systems.

In a right handed system, if you rotate your *right* hand around the z -axis by sweeping from the positive x -axis around to the positive y -axis, as shown in the figure, your thumb points along the positive z -axis. In a left handed system, you must do this with your *left* hand to make your thumb point along the positive z -axis. Right-handed systems are more familiar and are conventionally used in mathematics, physics, and engineering discussions. In this text we use a right-handed system when setting up models for objects. But left-handed systems also have a natural place in graphics, when dealing with viewing systems and “cameras”.

We first look at the basics of vectors, how one works with them, and how they are useful in graphics. In Section 4.5 we return to fundamentals and show an important distinction between points and vectors that, if ignored, can cause great difficulties in graphics programs.

4.2. Review of Vectors.

“Not only Newton’s laws, but also the other laws of physics, so far as we know today, have the two properties which we call invariance under translation of axes and rotation of axes. These properties are so important that a mathematical technique has been developed to take advantage of them in writing and using physical laws.. called vector analysis.”

Richard Feynman

Vector arithmetic provides a unified way to express geometric ideas algebraically. In graphics we work with vectors of two, three, and four dimensions, but many results need only be stated once and they apply to vectors of any dimension. This makes it possible to bring together the various cases that arise in graphics together into a single expression, which can be applied to a broad variety of tasks.

Viewed geometrically, vectors are objects having length and direction. They correspond to various physical entities such as force, displacement, and velocity. A vector is often drawn as an arrow of a certain length pointing in a certain direction. It is valuable to think of a vector geometrically as a *displacement* from one point to another.

Figure 4.3 uses vectors to show how the stars in the Big Dipper are moving over time [kerr79]. The current location of each star is shown by a point, and a vector shows the velocity of each star. The “tip” of each arrow shows the point where its star will be located in 50,000 years: producing a very different Big Dipper indeed!

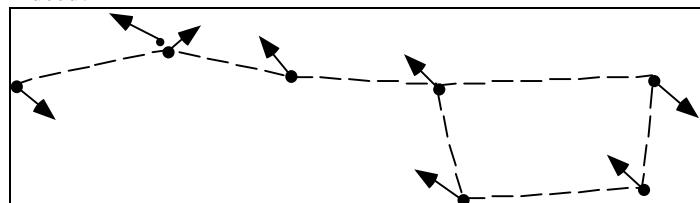


Figure 4.3. The Big Dipper now and in AD 50,000.

Figure 4.4a shows, in a 2D coordinate system, the two points $P = (1, 3)$ and $Q = (4, 1)$. The displacement from P to Q is a vector \mathbf{v} having components $(3, -2)$,¹ calculated by subtracting the coordinates of the points individually. To “get from” P to Q we shift down by 2 and to the right by 3. Because a vector is a displacement it has size and direction but no inherent location: the two arrows labeled \mathbf{v} in the figure are in fact the same vector. Figure 4.4b shows the corresponding situation in three dimensions: \mathbf{v} is the vector from point P to point Q . One often states:

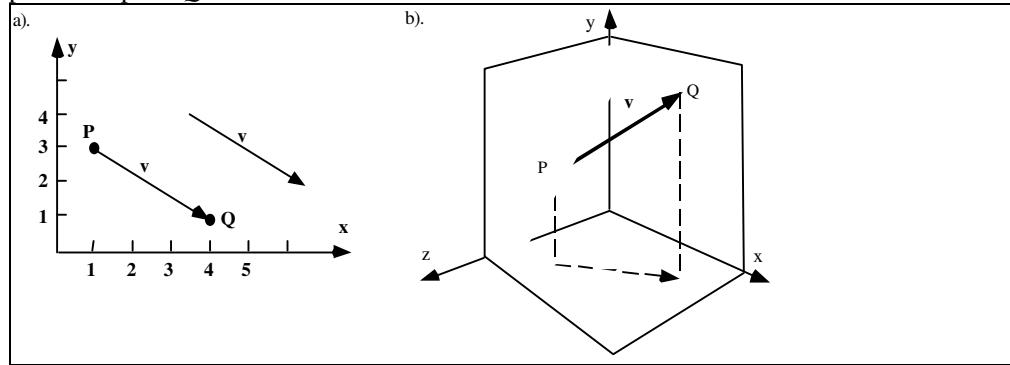


Figure 4.4. A vector as a displacement.

- The **difference** between two points is a vector: $\mathbf{v} = Q - P$;

Turning this around, we also say that a point Q is formed by displacing point P by vector \mathbf{v} ; we say that \mathbf{v} “offsets” P to form Q . Algebraically, Q is then the **sum**: $Q = P + \mathbf{v}$.

- The **sum** of a point and a vector is a point: $P + \mathbf{v} = Q$.

At this point we represent a vector through a list of its components: an n -dimensional vector is given by an *n-tuple*:

$$\mathbf{w} = (w_1, w_2, \dots, w_n) \quad (4.1)$$

Mostly we will be interested in 2D or 3D vectors as in $\mathbf{r} = (3.4, -7.78)$ or $\mathbf{t} = (33, 142.7, 89.1)$. Later when it becomes important we will explore the distinction between a vector and its *representation*, and in fact will use a slightly expanded notation to represent vectors (and points). Writing a vector as a *row matrix* like $\mathbf{t} = (33, 142.7, 89.1)$ fits nicely on the page, but when it matters we will instead write vectors as *column matrices*:

$$\mathbf{r} = \begin{pmatrix} 3.4 \\ -7.78 \end{pmatrix}, \text{ or } \mathbf{t} = \begin{pmatrix} 33 \\ 142.7 \\ 89.1 \end{pmatrix}$$

It matters when we want to multiply a point or a vector by a matrix, as we shall see in Chapter 5.

4.2.1. Operations with vectors.

Vectors permit two fundamental operations: you can add them, and you can multiply them by **scalars** (real numbers)². So if \mathbf{a} and \mathbf{b} are two vectors, and s is a scalar, it is meaningful to form both $\mathbf{a} + \mathbf{b}$ and the product $s\mathbf{a}$. For example, if $\mathbf{a} = (2, 5, 6)$ and $\mathbf{b} = (-2, 7, 1)$, we can form the two vectors:

$$\mathbf{a} + \mathbf{b} = (0, 12, 7)$$

¹Upper case letters are conventionally used for points, and boldface lower case letters for vectors.

²There are also systems where the scalars can be complex numbers; we do not work with them here.

$$6 \mathbf{a} = (12, 30, 36)$$

always performing the operations *componentwise*. Figure 4.5 shows a two-dimensional example, using $\mathbf{a} = (1, -1)$ and $\mathbf{b} = (2, 1)$. We can represent the addition of two vectors graphically in two different ways. In Figure 4.5a we show both vectors “starting” at the same point, thereby forming two sides of a parallelogram. The sum of the vectors is then a diagonal of this parallelogram, the diagonal that emanates from the binding point of the vectors. This view — the “parallelogram rule” for adding vectors — is the natural picture for forces acting at a point: The diagonal gives the resultant force.

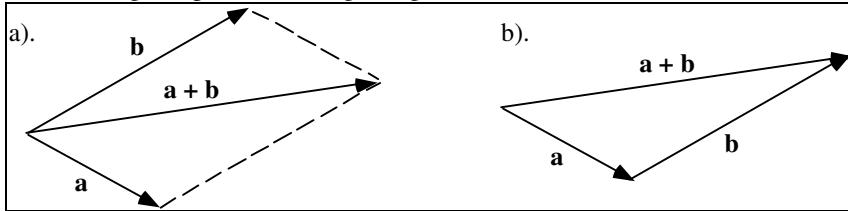


Figure 4.5. The sum of two vectors.

Alternatively, in Figure 4.5b we show one vector starting at the head of the other (i.e., place the tail of \mathbf{b} at the head of \mathbf{a}) and draw the sum as emanating from the tail of \mathbf{a} to the head of \mathbf{b} . The sum completes the triangle, which is the simple addition of one displacement to another. The components of the sum are clearly the sums of the components of its parts, as the algebra dictates.

Figure 4.6 shows the effect of scaling a vector. For $s = 2.5$ the vector $s \mathbf{a}$ has the same direction as \mathbf{a} but is 2.5 times as long. When s is negative, the direction of $s \mathbf{a}$ is opposite that of \mathbf{a} : The case $s = -1$ is shown in the figure.

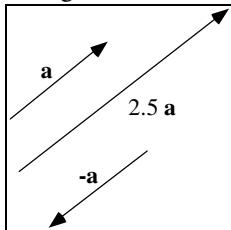


Figure 4.6. Scaling a vector.

Subtraction follows easily once adding and scaling have been established: $\mathbf{a} - \mathbf{c}$ is simply $\mathbf{a} + (-\mathbf{c})$. Figure 4.7 shows the geometric interpretation of this operation, forming the difference of \mathbf{a} and \mathbf{c} as the sum of \mathbf{a} and $-\mathbf{c}$ (Figure 4.7b). Using the parallelogram rule, this sum is seen to be equal to the vector that

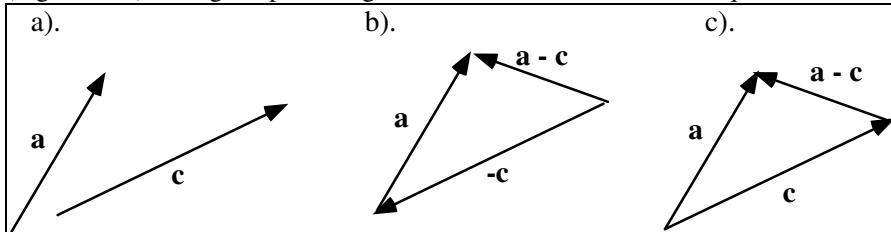


Figure 4.7. Subtracting vectors.

emanates from the head of \mathbf{c} and terminates at the head of \mathbf{a} (Figure 4.7c). This is recognized as one diagonal of the parallelogram constructed using \mathbf{a} and \mathbf{c} . Note too that it is the “other” diagonal from the one that represents the sum $\mathbf{a} + \mathbf{c}$.

4.2.2. Linear Combinations of Vectors.

With methods in hand for adding and scaling vectors, we can define a linear combination of vectors. To form a **linear combination** of two vectors, \mathbf{v} and \mathbf{w} , (having the same dimension) we scale each of them by some scalars, say a and b , and add the weighted versions to form the new vector, $a \mathbf{v} + b \mathbf{w}$. The more general definition for combining m such vectors is:

Definition:

A **linear combination** of the m vectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m$ is a vector of the form

$$\mathbf{w} = a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + \dots + a_m \mathbf{v}_m \quad (4.2)$$

where a_1, a_2, \dots, a_m are scalars.

For example, the linear combination $2(3, 4, -1) + 6(-1, 0, 2)$ forms the vector $(0, 8, 10)$. In later chapters we shall deal with rather elaborate linear combinations of vectors, especially when representing curves and surfaces using spline functions.

Two special types of linear combinations, “affine” and “convex” combinations, are particularly important in graphics.

Affine Combinations of Vectors.

A linear combination is an **affine combination** if the coefficients a_1, a_2, \dots, a_m add up to 1. Thus the linear combination in Equation 4.2 is affine if:

$$a_1 + a_2 + \dots + a_m = 1 \quad (4.3)$$

For example, $3\mathbf{a} + 2\mathbf{b} - 4\mathbf{c}$ is an affine combination of \mathbf{a}, \mathbf{b} , and \mathbf{c} , but $3\mathbf{a} + \mathbf{b} - 4\mathbf{c}$ is not. The coefficients of an affine combination of two vectors \mathbf{a} and \mathbf{b} are often forced to sum to 1 by writing one as some scalar t and the other as $(1-t)$:

$$(1-t)\mathbf{a} + (t)\mathbf{b} \quad (4.4)$$

Affine combinations of vectors appear in various contexts, as do affine combinations of points, as we see later.

Convex Combinations of Vectors.

Convex combinations have an important place in mathematics, and numerous applications in graphics. A **convex combination** arises as a further restriction on an affine combination. Not only must the coefficients of the linear combination sum to one; each one must also be nonnegative. The linear combination of Equation 4.2.2 is **convex** if:

$$a_1 + a_2 + \dots + a_m = 1, \quad (4.5)$$

and $a_i \geq 0$, for $i = 1, \dots, m$. As a consequence all a_i must lie between 0 and 1. (Why?).

Thus $.3\mathbf{a}+.7\mathbf{b}$ is a convex combination of \mathbf{a} and \mathbf{b} , but $1.8\mathbf{a}-.8\mathbf{b}$ is not. The set of coefficients a_1, a_2, \dots, a_m is sometimes said to form a **partition of unity**, suggesting that a unit amount of “material” is partitioned into pieces. Convex combinations frequently arise in applications when one is making a unit amount of some brew and can combine only positive amounts of the various ingredients. They appear in unexpected contexts. For instance, we shall see in Chapter 8 that “spline” curves are in fact convex combinations of certain vectors, and in our discussion of color in Chapter 12 we shall find that colors can be considered as vectors, and that any color of unit brightness may be considered to be a convex combination of three primary colors!

We will find it useful to talk about the “set of all convex combinations” of a collection of vectors. Consider the set of all convex combinations of the two vectors \mathbf{v}_1 and \mathbf{v}_2 . It is the set of all vectors

$$\mathbf{v} = (1 - a)\mathbf{v}_1 + a\mathbf{v}_2 \quad (4.6)$$

as the parameter a is allowed to vary from 0 to 1 (why?) What is this set? Rearranging the equation, \mathbf{v} is seen to be:

$$\mathbf{v} = \mathbf{v}_1 + a (\mathbf{v}_2 - \mathbf{v}_1) \quad (4.7)$$

Figure 4.8a shows this to be the vector that is \mathbf{v}_1 plus some fraction of $\mathbf{v}_2 - \mathbf{v}_1$, so the tip of \mathbf{v} lies on the line joining \mathbf{v}_1 and \mathbf{v}_2 . As a varies from 0 to 1, \mathbf{v} takes on all the positions on the line from \mathbf{v}_1 to \mathbf{v}_2 , and only those.

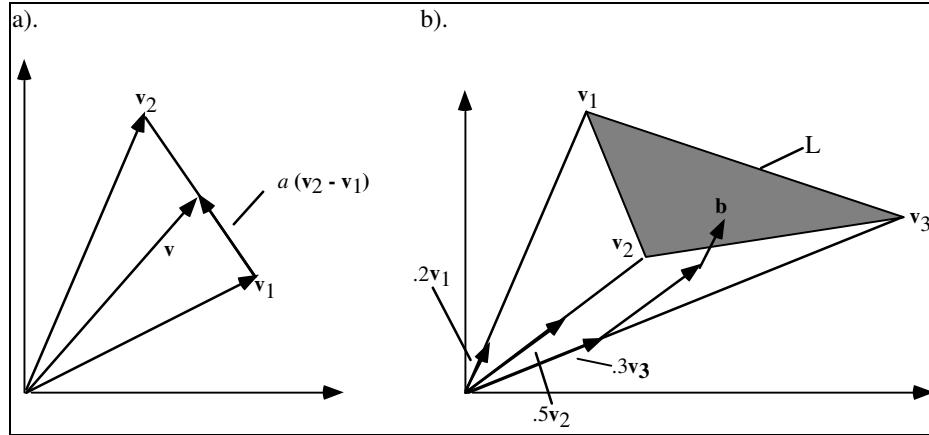


Figure 4.8. The set of vectors representable by convex combinations.

Figure 4.8b shows the set of all convex combinations of three vectors. Choose two parameters a_1 and a_2 , both lying between 0 and 1, and form the following linear combination:

$$\mathbf{q} = a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + (1 - a_1 - a_2) \mathbf{v}_3 \quad (4.8)$$

where we also insist that a_1 plus a_2 does not exceed one. This is a convex combination, since none of the coefficients is ever negative and they sum to one. Figure 4.9 shows the three position vectors $\mathbf{v}_1 = (2, 6)$, $\mathbf{v}_2 = (3, 3)$, and $\mathbf{v}_3 = (7, 4)$. By the proper choices of a_1 and a_2 , any vector lying within the shaded triangle of vectors can be represented, and no vectors outside this triangle can be reached. The vector $\mathbf{b} = .2 \mathbf{v}_1 + .5 \mathbf{v}_2 + .3 \mathbf{v}_3$, for instance, is shown explicitly as the vector sum of the three weighted ingredients. (Note how it is built up out of “portions” of the three constituent vectors.) So the set of all convex combinations of these three vectors “spans” the shaded triangle. The proof of this is requested in the exercises.

If $a_2 = 0$, any vector in the line L that joins \mathbf{v}_1 and \mathbf{v}_3 can be “reached” by the proper choice of a_1 . For example, the vector that is 20 percent of the way from \mathbf{v}_1 to \mathbf{v}_3 along L is given by $.8 \mathbf{v}_1 + 0 \mathbf{v}_2 + .2 \mathbf{v}_3$.

4.2.3. The Magnitude of a vector, and unit vectors.

If a vector \mathbf{w} is represented by the n -tuple (w_1, w_2, \dots, w_n) , how might its magnitude (equivalently, its length or size) be defined and computed? We denote the magnitude by $|\mathbf{w}|$ and define it as the distance from its tail to its head. Based on the Pythagorean theorem, this becomes

$$|\mathbf{w}| = \sqrt{w_1^2 + w_2^2 + \dots + w_n^2} \quad (4.9)$$

For example, the magnitude of $\mathbf{w} = (4, -2)$ is $\sqrt{20}$, and that of $\mathbf{w} = (1, -3, 2)$ is $\sqrt{14}$. A vector of zero length is denoted as $\mathbf{0}$. Note that if \mathbf{w} is the vector from point A to point B , then $|\mathbf{w}|$ will be the distance from A to B (why?).

It is often useful to scale a vector so that the result has a length equal to one. This is called **normalizing** a vector, and the result is known as a **unit vector**. For example, we form the normalized version of \mathbf{a} , denoted $\hat{\mathbf{a}}$, by scaling it with the value $1/|\mathbf{a}|$:

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{|\mathbf{a}|} \quad (4.10)$$

Clearly this is a unit vector: $|\hat{\mathbf{a}}| = 1$ (why?), having the same direction as \mathbf{a} . For example, if $\mathbf{a} = (3, -4)$, then $|\mathbf{a}| = 5$ and the normalized version is $\hat{\mathbf{a}} = (\frac{3}{5}, \frac{-4}{5})$. At times we refer to a unit vector as a **direction**. Note that any vector can be written as its magnitude times its direction: If $\hat{\mathbf{a}}$ is the normalized version of \mathbf{a} , vector \mathbf{a} may always be written $\mathbf{a} = |\mathbf{a}| \hat{\mathbf{a}}$.

Practice Exercises.

4.2.1. Representing Vectors as linear combinations. With reference to Figure 4.9, what values, or range of values, for a_1 and a_2 create the following sets?

- a. \mathbf{v}_1 .
- b. The line joining \mathbf{v}_1 and \mathbf{v}_2 .
- c. The vector midway between \mathbf{v}_2 and \mathbf{v}_3 .
- d. The centroid of the triangle.

4.2.2. The set of all convex combinations. Show that the set of all convex combinations of three vectors \mathbf{v}_1 , \mathbf{v}_2 , and \mathbf{v}_3 is the set of vectors whose tips lie in the “triangle” formed by the tips of the three vectors. Hint: Each point in the triangle is a combination of \mathbf{v}_1 and some point lying between \mathbf{v}_2 and \mathbf{v}_3 .

4.2.3. Factoring out a scalar. Show how scaling a vector \mathbf{v} by a scalar s changes its length. That is, show that: $|s\mathbf{v}| = |s||\mathbf{v}|$. Note the dual use of the magnitude symbol $|\cdot|$, once for a scalar and once for a vector.

4.2.4. Normalizing Vectors. Normalize each of the following vectors:

- a). $(1, -2, .5)$; b). $(8, 6)$; c). $(4, 3)$

4.3. The Dot Product.

There are two other powerful tools that facilitate working with vectors: the dot (or inner) product, and the cross product. The dot product produces a scalar; the cross product works only on three dimensional vectors and produces another vector. In this section we review the basic properties of the dot product, principally to develop the notion of perpendicularity. We then work with the dot product to solve a number of important geometric problems in graphics. Then the cross product is introduced, and used to solve a number of 3D geometric problems.

The **dot product** of two vectors is simple to define and compute. For two-dimensional vectors, (a_1, a_2) and (b_1, b_2) , it is simply the scalar whose value is $a_1b_1 + a_2b_2$. Thus to calculate it, multiply corresponding components of the two vectors, and add the results. For example, the dot product of $(3, 4)$ and $(1, 6)$ is 27, and that of $(2, 3)$ and $(9, -6)$ is 0.

The definition of the dot product generalizes easily to n dimensions:

Definition: The Dot Product

The dot product d of two n -dimensional vectors $\mathbf{v} = (v_1, v_2, \dots, v_n)$ and $\mathbf{w} = (w_1, w_2, \dots, w_n)$ is denoted as $\mathbf{v} \cdot \mathbf{w}$ and has the value

$$d = \mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^n v_i w_i \quad (4.11)$$

Example 4.3.1:

- The dot product of $(2, 3, 1)$ and $(0, 4, -1)$ is 11.
- $(2, 2, 2, 2) \cdot (4, 1, 2, 1.1) = 16.2$.
- $(1, 0, 1, 0, 1) \cdot (0, 1, 0, 1, 0) = 0$.
- $(169, 0, 43) \cdot (0, 375.3, 0) = 0$.

4.3.1. Properties of the Dot Product

The dot product exhibits four major properties that we frequently exploit and that follow easily (see the exercises) from its basic definition:

- | | |
|-----------------|--|
| 1. Symmetry: | $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$ |
| 2. Linearity: | $(\mathbf{a} + \mathbf{c}) \cdot \mathbf{b} = \mathbf{a} \cdot \mathbf{b} + \mathbf{c} \cdot \mathbf{b}$ |
| 3. Homogeneity: | $(s\mathbf{a}) \cdot \mathbf{b} = s(\mathbf{a} \cdot \mathbf{b})$ |
| 4. | $ \mathbf{b} ^2 = \mathbf{b} \cdot \mathbf{b}$ |

The first states that the order in which the two vectors are combined does not matter: the dot product is **commutative**. The next two proclaim that the dot product is **linear**; that is, the dot product of a sum of vectors can be expressed as the sum of the individual dot products, and scaling a vector scales the value of the dot product. The last property is also useful, as it asserts that taking the dot product of a vector with itself yields the **square of the length** of the vector. It appears frequently in the form $|\mathbf{b}| = \sqrt{\mathbf{b} \cdot \mathbf{b}}$.

The following manipulations show how these properties can be used to simplify an expression involving dot products. The result itself will be used in the next section.

Example 4.3.2: Simplification of $|\mathbf{a} - \mathbf{b}|^2$.

Simplify the expression for the length (squared) of the difference of two vectors, \mathbf{a} and \mathbf{b} , to obtain the following relation:

$$|\mathbf{a} - \mathbf{b}|^2 = |\mathbf{a}|^2 - 2\mathbf{a} \cdot \mathbf{b} + |\mathbf{b}|^2 \quad (4.12)$$

The derivation proceeds as follows: Give the name C to the expression $|\mathbf{a} - \mathbf{b}|^2$. By the fourth property, C is the dot product:

$$C = |\mathbf{a} - \mathbf{b}|^2 = (\mathbf{a} - \mathbf{b}) \cdot (\mathbf{a} - \mathbf{b}).$$

Using linearity: $C = \mathbf{a} \cdot (\mathbf{a} - \mathbf{b}) - \mathbf{b} \cdot (\mathbf{a} - \mathbf{b})$.

Using symmetry and linearity to simplify this further: $C = \mathbf{a} \cdot \mathbf{a} - 2\mathbf{a} \cdot \mathbf{b} + \mathbf{b} \cdot \mathbf{b}$.

Using the fourth property above to obtain $C = |\mathbf{a}|^2 - 2\mathbf{a} \cdot \mathbf{b} + |\mathbf{b}|^2$ gives the desired result.

By replacing the minus with a plus in this relation, the following similar and useful relation emerges:

$$|\mathbf{a} + \mathbf{b}|^2 = |\mathbf{a}|^2 + 2\mathbf{a} \cdot \mathbf{b} + |\mathbf{b}|^2 \quad (4.13)$$

4.3.2. The Angle Between Two Vectors.

The most important application of the dot product is in finding the angle between two vectors, or between two intersecting lines. Figure 4.9 shows the 2D case, where vectors \mathbf{b} and \mathbf{c} lie at angles ϕ_b and ϕ_c , relative to the x -axis. Now from elementary trigonometry:

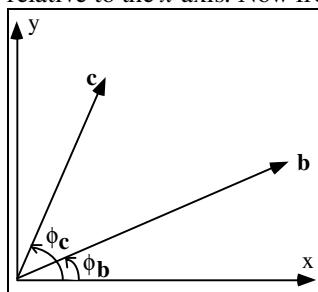


Figure 4.9. Finding the angle between two vectors.

$$\mathbf{b} = (|\mathbf{b}| \cos \varphi_{\mathbf{b}}, |\mathbf{b}| \sin \varphi_{\mathbf{b}})$$

$$\mathbf{c} = (|\mathbf{c}| \cos \varphi_{\mathbf{c}}, |\mathbf{c}| \sin \varphi_{\mathbf{c}}).$$

Thus their dot product is

$$\begin{aligned}\mathbf{b} \cdot \mathbf{c} &= |\mathbf{b}| |\mathbf{c}| \cos \varphi_{\mathbf{c}} \cos \varphi_{\mathbf{b}} + |\mathbf{b}| |\mathbf{c}| \sin \varphi_{\mathbf{c}} \sin \varphi_{\mathbf{b}} \\ &= |\mathbf{b}| |\mathbf{c}| \cos(\varphi_{\mathbf{c}} - \varphi_{\mathbf{b}})\end{aligned}$$

so we have, for any two vectors \mathbf{b} and \mathbf{c} :

$$\mathbf{b} \cdot \mathbf{c} = |\mathbf{b}| |\mathbf{c}| \cos(\theta) \quad (4.14)$$

where θ is the angle from \mathbf{b} to \mathbf{c} . Thus $\mathbf{b} \cdot \mathbf{c}$ varies as the cosine of the angle from \mathbf{b} to \mathbf{c} . The same result holds for vectors of three, four, or any number of dimensions.

To obtain a slightly more compact form, divide through both sides by $|\mathbf{b}| |\mathbf{c}|$ and use the unit vector notation $\hat{\mathbf{b}} = \mathbf{b} / |\mathbf{b}|$ to obtain

$$\cos(\theta) = \hat{\mathbf{b}} \cdot \hat{\mathbf{c}} \quad (4.15)$$

This is the desired result: The cosine of the angle between two vectors \mathbf{b} and \mathbf{c} is the dot product of their normalized versions.

Example 4.3.3. Find the angle between $\mathbf{b} = (3, 4)$ and $\mathbf{c} = (5, 2)$.

Solution: Form $|\mathbf{b}| = 5$ and $|\mathbf{c}| = 5.385$ so that $\hat{\mathbf{b}} = (3/5, 4/5)$ and $\hat{\mathbf{c}} = (.9285, .3714)$. The dot product $\hat{\mathbf{b}} \cdot \hat{\mathbf{c}} = .85422 = \cos(\theta)$, so that $\theta = 31.326^\circ$. This can be checked by plotting the two vectors on graph paper and measuring the angle between them.

4.3.3. The Sign of $\mathbf{b} \cdot \mathbf{c}$, and Perpendicularity.

Recall that $\cos(\theta)$ is **positive** if $|\theta|$ is less than 90° , **zero** if $|\theta|$ equals 90° , and **negative** if $|\theta|$ exceeds 90° . Because the dot product of two vectors is proportional to the cosine of the angle between them, we can therefore observe immediately that two vectors (of any nonzero length) are

- less than 90° apart if $\mathbf{b} \cdot \mathbf{c} > 0$;
- exactly 90° apart if $\mathbf{b} \cdot \mathbf{c} = 0$;
- more than 90° apart if $\mathbf{b} \cdot \mathbf{c} < 0$;

This is indicated by Figure 4.10. The sign of the dot product is used in many algorithmic tests.

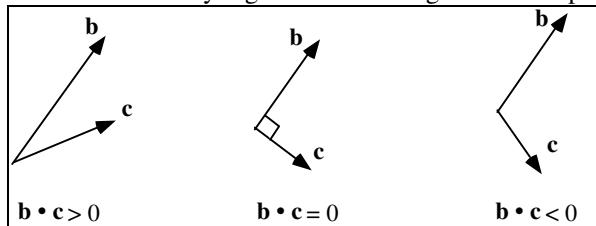


Figure 4.10. The sign of the dot product.

The case in which the vectors are 90° apart, or **perpendicular**, is of special importance.

Definition:

Vectors \mathbf{b} and \mathbf{c} are perpendicular if $\mathbf{b} \cdot \mathbf{c} = 0$.
(4.17)

Other names for “perpendicular” are **orthogonal** and **normal**, and we shall use all three interchangeably.

The most familiar examples of orthogonal vectors are those aimed along the axes of 2D and 3D coordinate systems, as shown in Figure 4.11. In part a) the 2D vectors $(1, 0)$ and $(0, 1)$ are mutually perpendicular unit vectors. The 3D versions are so commonly used they are called the “standard unit vectors” and are given names **i**, **j**, and **k**.

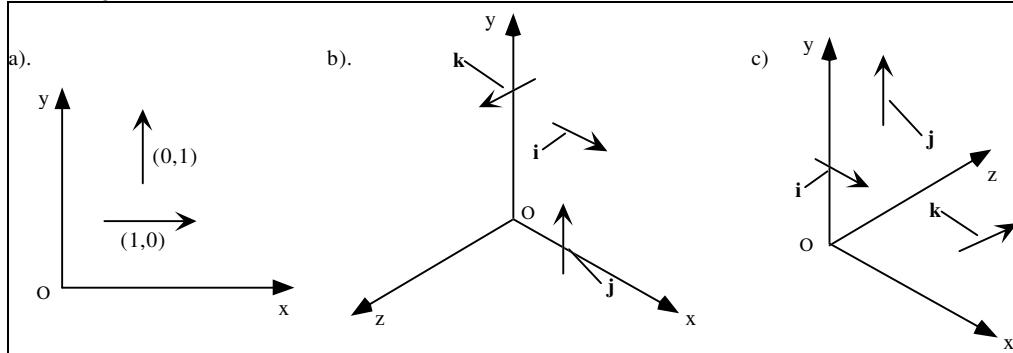


Figure 4.11. The standard unit vectors.

Definition:

The **standard unit vectors** in 3D have components:

$$\mathbf{i} = (1, 0, 0), \quad \mathbf{j} = (0, 1, 0), \quad \text{and } \mathbf{k} = (0, 0, 1). \quad (4.18)$$

Part b) of the figure shows them for a right-handed system, and part c) shows them for a left-handed system. Note that **k** always points in the positive z direction.

Using these definitions any 3D vector such as (a, b, c) can be written in the alternative form:

$$(a, b, c) = a \mathbf{i} + b \mathbf{j} + c \mathbf{k} \quad (4.19)$$

Example 4.3.4. Notice that $\mathbf{v} = (2, 5, -1)$ is clearly the same as $2(1, 0, 0) + 5(0, 1, 0) - 1(0, 0, 1)$, which is recognized as $2\mathbf{i} + 5\mathbf{j} - \mathbf{k}$.

This form presents a vector as a sum of separate elementary component vectors, so it simplifies various pencil-and-paper calculations. It is particularly convenient when dealing with the cross product, discussed in Section 4.4.

Practice Exercises.

4.3.1. Alternate proof of $\mathbf{b} \cdot \mathbf{c} = |\mathbf{b}| |\mathbf{c}| \cos \theta$. Note that **b** and **c** form two sides of a triangle, and the third side is $\mathbf{b} - \mathbf{c}$. Use the law of cosines to obtain the square of the length of $\mathbf{b} - \mathbf{c}$ in terms of the lengths of **b** and **c** and the cosine of θ . Compare this with Equation 4.13 to obtain the desired result.

4.3.2. Find the Angle. Calculate the angle between the vectors $(2, 3)$ and $(-3, 1)$, and check the result visually using graph paper. Then compute the angle between the 3D vectors $(1, 3, -2)$ and $(3, 3, 1)$.

4.3.3. Testing for Perpendicularity. Which pairs of the following vectors are perpendicular to one another: $(3, 4, 1)$, $(2, 1, 1)$, $(-3, -4, 1)$, $(0, 0, 0)$, $(1, -2, 0)$, $(4, 4, 4)$, $(0, -1, 4)$, and $(2, 2, 1)$?

4.3.4. Pythagorean Theorem. Refer to Equations 4.12 and 4.13. For the case in which **a** and **b** are perpendicular, these expressions have the same value, which seems to make no sense geometrically. Show that it works all right, and relate the result to the Pythagorean theorem.

4.3.4. The 2D “Perp” Vector.

Suppose the 2D vector \mathbf{a} has components (a_x, a_y) . What vectors are perpendicular to it? One way to obtain such a vector is to interchange the x - and y - components and negate one of them.³ Let $\mathbf{b} = (-a_y, a_x)$. Then the dot product $\mathbf{a} \cdot \mathbf{b}$ equals 0 so \mathbf{a} and \mathbf{b} are indeed perpendicular. For instance, if $\mathbf{a} = (4, 7)$ then $\mathbf{b} = (-7, 4)$ is a vector normal to \mathbf{a} . There are infinitely many vectors normal to any \mathbf{a} , since any scalar multiple of \mathbf{b} , such as $(-21, 12)$ and $(7, -4)$ is also normal to \mathbf{a} . (Sketch several of them for a given \mathbf{a} .)

It is convenient to have a symbol for one *particular* vector that is normal to a given 2D vector \mathbf{a} . We use the symbol \perp (pronounced “perp”) for this.

Definition: Given $\mathbf{a} = (a_x, a_y)$, $\mathbf{a}^\perp = (-a_y, a_x)$ is the **couterclockwise**

(4.20)

perpendicular to \mathbf{a} .

Note that \mathbf{a} and \mathbf{a}^\perp have the same length: $|\mathbf{a}| = |\mathbf{a}^\perp|$. Figure 4.12a shows an arbitrary vector \mathbf{a} and the resulting \mathbf{a}^\perp . Note that moving from the \mathbf{a} direction to direction \mathbf{a}^\perp requires a left turn. (Making a right turn is equivalent to turning in the direction $-\mathbf{a}^\perp$.)

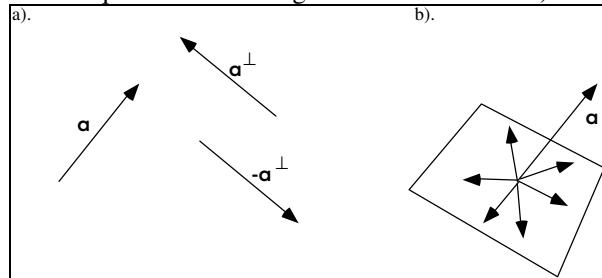


Figure 4.12. The vector \mathbf{a}^\perp perpendicular to \mathbf{a} .

We show in the next section how this notation can be put to good use. Figure 4.12b shows that in three dimensions no single vector lies in “the” direction perpendicular to a given 3D vector \mathbf{a} , since any of the vectors lying in the plane perpendicular to \mathbf{a} will do. However, the cross product developed later will provide a simple tool for dealing with such vectors.

Practice Exercises.

4.3.5. Some Pleasant Properties of \mathbf{a}^\perp . It is useful in some discussions to view the “perp” symbol \perp as an operator that performs a “rotate 90° left” operation on its argument, so that \mathbf{a}^\perp is the vector produced by applying the \perp to vector \mathbf{a} , much as \sqrt{x} is the value produced by applying the square root operator to x .

Viewing \perp in this way, show that it enjoys the following properties:

- Linearity: $(\mathbf{a} + \mathbf{b})^\perp = \mathbf{a}^\perp + \mathbf{b}^\perp$ and $(A\mathbf{a})^\perp = A\mathbf{a}^\perp$ for any scalar A ;
- $\mathbf{a}^{\perp\perp} = (\mathbf{a}^\perp)^\perp = -\mathbf{a}$ (two perp’s make a reversal)

4.3.6. The “perp dot” product. Interesting things happen when we dot the “perp” of a vector with another vector, as in $\mathbf{a}^\perp \cdot \mathbf{b}$. We call this the “perp dot product” [hill95]. Use the basic definition of \mathbf{a}^\perp above to show:

- $\mathbf{a}^\perp \cdot \mathbf{b} = a_x b_y - a_y b_x$ (value of the perp dot product)
- $\mathbf{a}^\perp \cdot \mathbf{a} = 0$, $(\mathbf{a}^\perp \text{ is perpendicular to } \mathbf{a})$
- $|\mathbf{a}^\perp|^2 = |\mathbf{a}|^2$, $(\mathbf{a}^\perp \text{ and } \mathbf{a} \text{ have the same length})$ (4.21)
- $\mathbf{a}^\perp \cdot \mathbf{b} = -\mathbf{b}^\perp \cdot \mathbf{a}$, (antisymmetric)

³ This is equivalent to the familiar fact that perpendicular lines have slopes that are negative reciprocals of one another. In Chapter 5 we see the “interchange and negate” operation arise naturally in connection with a rotation of 90 degrees.

The fourth fact shows that the perp dot product is “antisymmetric”: moving the \perp from one vector to the other reverses the sign of the dot product. Other useful properties of the perp dot product will be discussed as they are needed.

4.3.7. Calculate one. Compute $\mathbf{a} \cdot \mathbf{b}$ and $\mathbf{a}^\perp \cdot \mathbf{b}$ for $\mathbf{a} = (3,4)$ and $\mathbf{b} = (2,1)$.

4.3.8. It's a determinant. Show that $\mathbf{a}^\perp \cdot \mathbf{b}$ can be written as the determinant (for definitions of matrices and determinants see Appendix 2):

$$\mathbf{a}^\perp \cdot \mathbf{b} = \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix}$$

4.3.9. Other goodies.

- Show that $(\mathbf{a}^\perp \cdot \mathbf{b})^2 + (\mathbf{a} \cdot \mathbf{b})^2 = |\mathbf{a}|^2 |\mathbf{b}|^2$.
- Show that if $\mathbf{a} + \mathbf{b} + \mathbf{c} = \mathbf{0}$ then $\mathbf{a}^\perp \cdot \mathbf{b} = \mathbf{b}^\perp \cdot \mathbf{c} = \mathbf{c}^\perp \cdot \mathbf{a}$.

4.3.5. Orthogonal Projections, and the Distance from a Point to a Line.

Three geometric problems arise frequently in graphics applications: **projecting** a vector onto a given vector, **resolving** a vector into its components in one direction and another, and finding the distance between a point and a line. All three problems are simplified if we use the perp vector and the perp dot product.

Figure 4.13a shows the basic ingredients. We are given two points A and C , and a vector \mathbf{v} . These questions arise:

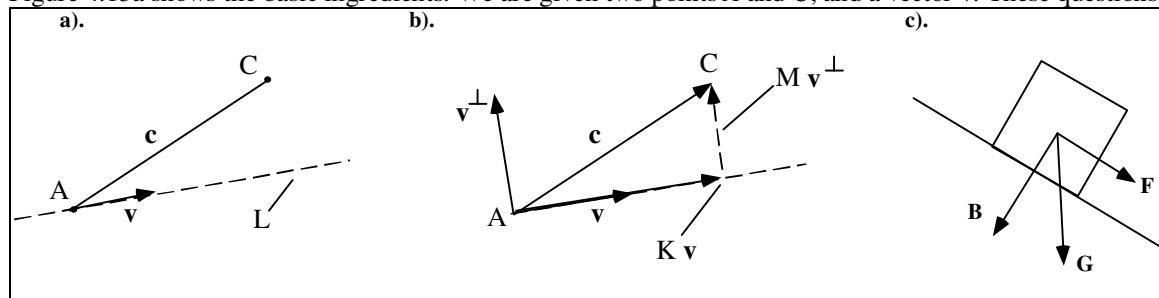


Figure 4.13. Resolving a vector into two orthogonal vectors.

- How far is the point C from the line L that passes through A in the direction \mathbf{v} ?
- If we drop a perpendicular from C onto L , where does it hit L ?
- How do we decompose the vector $\mathbf{c} = C - A$ into a part along the line L and a part perpendicular to L ?

Figure 4.13.b defines some additional quantities: \mathbf{v}^\perp is the vector \mathbf{v} rotated 90 degrees CCW. Dropping a perpendicular from C onto line L we say that the vector \mathbf{c} is **resolved** into the portion $K\mathbf{v}$ along \mathbf{v} and the portion $M\mathbf{v}^\perp$ perpendicular to \mathbf{v} , where K and M are some constants to be determined. Then we have

$$\mathbf{c} = K\mathbf{v} + M\mathbf{v}^\perp \quad (4.22)$$

Given \mathbf{c} and \mathbf{v} we want to solve for K and M . Once found, we say that the **orthogonal projection** of \mathbf{c} onto \mathbf{v} is $K\mathbf{v}$, and that the distance from C to the line is $|M\mathbf{v}^\perp|$.

Figure 4.13.c shows a situation where these questions might arise. We wish to analyze how the gravitational force vector \mathbf{G} acts on the block to pull it down the incline. To do this we must resolve \mathbf{G} into the force \mathbf{F} acting along the incline and the force \mathbf{B} acting perpendicular to the incline. That is, find \mathbf{F} and \mathbf{B} such that $\mathbf{G} = \mathbf{F} + \mathbf{B}$.

Equation 4.22 is really two equations: the left and right hand sides must agree for the x -components and they also must agree for the y -components. There are two unknowns K and M . So we have two equations in two unknowns, and Cramer's rule can be applied. But who remembers Cramer's rule? We use a trick

here that is easy to remember and immediately reveals the solution. It is equivalent to Cramer's rule, but simpler to apply.

The trick in solving two equations in two unknowns is to eliminate one of the variables. We do this by forming the dot product of both sides with the vector \mathbf{v} :

$$\mathbf{c} \cdot \mathbf{v} = K \mathbf{v} \cdot \mathbf{v} + M \mathbf{v}^\perp \cdot \mathbf{v} \quad (4.23)$$

Happily, the term $\mathbf{v}^\perp \cdot \mathbf{v}$ vanishes, (why?), yielding K immediately:

$$K = \frac{\mathbf{c} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}}.$$

Similarly “dot” both sides of Equation 4.3.12 with \mathbf{v}^\perp to obtain M :

$$M = \frac{\mathbf{c} \cdot \mathbf{v}^\perp}{\mathbf{v} \cdot \mathbf{v}}$$

where we have used the third property in Equation 4.21. Putting these together we have

$$\mathbf{c} = \left(\frac{\mathbf{v} \cdot \mathbf{c}}{\|\mathbf{v}\|^2} \right) \mathbf{v} + \left(\frac{\mathbf{v}^\perp \cdot \mathbf{c}}{\|\mathbf{v}\|^2} \right) \mathbf{v}^\perp \quad (\text{resolving } \mathbf{c} \text{ into } \mathbf{v} \text{ and } \mathbf{v}^\perp) \quad (4.24)$$

This equality holds for any vectors \mathbf{c} and \mathbf{v} . The part along \mathbf{v} is known as the **orthogonal projection** of \mathbf{c} onto the vector \mathbf{v} . The second term gives the “difference term” explicitly and compactly. Its size is the distance from C to the line:

$$\text{distance} = \left| \frac{\mathbf{v}^\perp \cdot \mathbf{c}}{\|\mathbf{v}\|^2} \mathbf{v}^\perp \right| = \frac{|\mathbf{v}^\perp \cdot \mathbf{c}|}{\|\mathbf{v}\|},$$

(Check that the second form really equals the first). Referring to Figure 4.13b we can say: **the distance from a point C to the line through A in the direction \mathbf{v} is:**

$$\text{distance} = \frac{|\mathbf{v}^\perp \cdot (\mathbf{C} - \mathbf{A})|}{\|\mathbf{v}\|}. \quad (4.25)$$

Example 4.3.5. Find the orthogonal projection of the vector $\mathbf{c} = (6, 4)$ onto $\mathbf{a} = (1, 2)$. (Sketch the relevant vectors.) **Solution:** Evaluate the first term in Equation 4.24, obtaining the vector $(14, 28)/5$.

Example 4.3.6: How far is the point $C = (6, 4)$ from the line that passes through $(1, 1)$ and $(4, 9)$? **Solution:** Set $A = (1, 1)$, use $\mathbf{v} = (4, 9) - (1, 1) = (3, 8)$, and evaluate *distance* in Equation 4.25. The result is:
 $d = 31/\sqrt{73}$.

Practice Exercises.

4.3.10. Resolve it. Express vector $\mathbf{g} = (4, 7)$ as a linear combination of $\mathbf{b} = (3, 5)$ and \mathbf{b}^\perp . How far is $(4, 2) + \mathbf{g}$ from the line through $(4, 2)$ that moves in the direction \mathbf{b} ?

4.3.11. A Block pulled down an incline. A block rests on an incline tilted 30° from the horizontal. Gravity exerts a force of one newton on the block. What is the force that is “trying” to move the block along the incline?

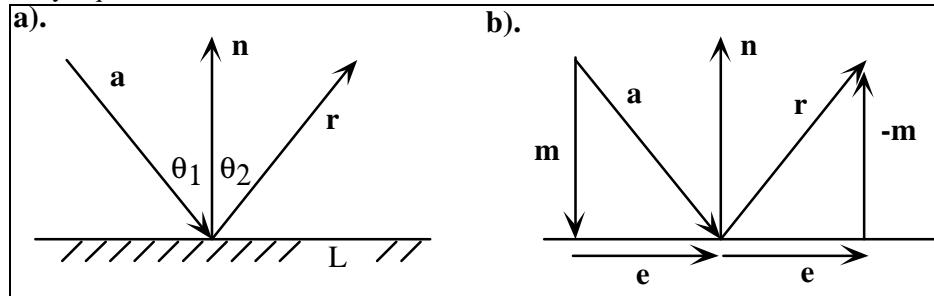
4.3.12. How far is it? How far from the line through $(2, 5)$ and $(4, -1)$ does the point $(6, 11)$ lie? Check your result on graph paper.

4.3.6. Applications of Projection: Reflections.

To display the reflection of light from a mirror, or the behavior of billiard balls bouncing off one another, we need to find the direction that an object takes upon being reflected at a given surface. In a case study at the end of this chapter we describe an application to trace a ray of light as it bounces around inside a reflective chamber, or a billiard ball as it bounces around a pool table. At each bounce a reflection is made to a new direction, as derived in this section.

When light reflects from a mirror we know that the angle of reflection must equal the angle of incidence. We next show how to use vectors and projections to compute this new direction. We can think in terms of two-dimensional vectors for simplicity, but because the derivation does not explicitly state the dimension of the vectors involved, the same result applies in three dimensions for reflections from a surface.

Figure 4.14a shows a ray having direction \mathbf{a} , hitting line L , and reflecting in (as yet unknown) direction \mathbf{r} . The vector \mathbf{n} is perpendicular to the line. Angle θ_1 in the figure must equal angle θ_2 . How is \mathbf{r} related to \mathbf{a} and \mathbf{n} ? Figure 4.14b shows \mathbf{a} resolved into the portion \mathbf{m} along \mathbf{n} and the portion \mathbf{e} orthogonal to \mathbf{n} . Because of symmetry, \mathbf{r} has the same component \mathbf{e} orthogonal to \mathbf{n} , but the opposite component along \mathbf{n} , and so $\mathbf{r} = \mathbf{e} - \mathbf{m}$. Because $\mathbf{e} = \mathbf{a} - \mathbf{m}$, this gives $\mathbf{r} = \mathbf{a} - 2\mathbf{m}$. Now \mathbf{m} is the orthogonal projection of \mathbf{a} onto \mathbf{n} , so by Equation 4.24 \mathbf{m} is



4.14. Reflection of a ray from a surface.

$$\mathbf{m} = \frac{\mathbf{a} \cdot \mathbf{n}}{|\mathbf{n}|^2} \mathbf{n} = (\mathbf{a} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} \quad (4.26)$$

(recall $\hat{\mathbf{n}}$ is the unit length version of \mathbf{n}) and so we obtain the result

$$\mathbf{r} = \mathbf{a} - 2(\mathbf{a} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} \quad (\text{direction of the reflected ray}) \quad (4.27)$$

In three dimensions physics demands that the reflected direction \mathbf{r} must lie in the plane defined by \mathbf{n} and \mathbf{a} . The expression for \mathbf{r} above indeed supports this, as we show in Chapter five.

Example 4.3.7. Let $\mathbf{a} = (4, -2)$ and $\mathbf{n} = (0, 3)$. Then Equation 4.27 yields $\mathbf{r} = (4, 2)$, as expected. Both the angle of incidence and reflection are equal to $\tan^{-1}(2)$.

Practice Exercises.

4.3.13. Find the Reflected Direction. For $\mathbf{a} = (2, 3)$ and $\mathbf{n} = (-2, 1)$, find the direction of the reflection.

4.3.14. Lengths of the Incident and Reflected Vectors. Using Equation 4.27 and properties of the dot product, show that $|\mathbf{r}| = |\mathbf{a}|$.

4.4. The Cross Product of Two Vectors.

The **cross product** (also called the **vector product**) of two vectors is another vector. It has many useful properties, but the one we use most often is that it is perpendicular to both of the given vectors. The cross product is defined only for three-dimensional vectors.

Given the 3D vectors $\mathbf{a} = (a_x, a_y, a_z)$ and $\mathbf{b} = (b_x, b_y, b_z)$, their cross product is denoted as $\mathbf{a} \times \mathbf{b}$. It is defined in terms of the standard unit vectors \mathbf{i} , \mathbf{j} , and \mathbf{k} (see Equation 4.18) by

Definition of $\mathbf{a} \times \mathbf{b}$:

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - a_z b_y) \mathbf{i} + (a_z b_x - a_x b_z) \mathbf{j} + (a_x b_y - a_y b_x) \mathbf{k} \quad (4.28)$$

(It can actually be derived from more fundamental principles: See the exercises.) As this form is rather difficult to remember, it is often written as an easily remembered determinant (see Appendix 2 for a review of determinants).

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} \quad (4.29)$$

Remembering how to form the cross product thus requires only remembering how to form a determinant.

Example 4.4.1. For $\mathbf{a} = (3, 0, 2)$ and $\mathbf{b} = (4, 1, 8)$, direct calculation shows that $\mathbf{a} \times \mathbf{b} = -2\mathbf{i} - 16\mathbf{j} + 3\mathbf{k}$. What is $\mathbf{b} \times \mathbf{a}$?

From this definition one can easily show the following algebraic properties of the cross product:

- $\mathbf{i} \times \mathbf{j} = \mathbf{k}$
 - 1. $\mathbf{j} \times \mathbf{k} = \mathbf{i}$
 - $\mathbf{k} \times \mathbf{i} = \mathbf{j}$
 - 2. $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$ (antisymmetry)
 - 3. $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$ (linearity)
 - 4. $(s\mathbf{a}) \times \mathbf{b} = s(\mathbf{a} \times \mathbf{b})$ (homogeneity)
- (4.30)

These equations are true in both left-handed and right-handed coordinate systems. Note the logical (alphabetical) ordering of ingredients in the equation $\mathbf{i} \times \mathbf{j} = \mathbf{k}$, which also provides a handy mnemonic device for remembering the direction of cross products.

Practice Exercises.

4.4.1. Demonstrate the Four Properties. Prove each of the preceding four properties given for the cross product.

4.4.2. Derivation of the Cross Product. The form in Equation 4.28, presented as a definition, can actually be derived from more fundamental ideas. We need only assume that:

- a. The cross product operation is linear.
- b. The cross product of a vector with itself is 0.
- c. $\mathbf{i} \times \mathbf{j} = \mathbf{k}$, $\mathbf{j} \times \mathbf{k} = \mathbf{i}$, and $\mathbf{k} \times \mathbf{i} = \mathbf{j}$.

By writing $\mathbf{a} = a_x \mathbf{i} + a_y \mathbf{j} + a_z \mathbf{k}$ and $\mathbf{b} = b_x \mathbf{i} + b_y \mathbf{j} + b_z \mathbf{k}$, apply these rules to derive the proper form for $\mathbf{a} \times \mathbf{b}$.

4.4.3. Is $\mathbf{a} \times \mathbf{b}$ perpendicular to \mathbf{a} ? Show that the cross product of vectors \mathbf{a} and \mathbf{b} is indeed perpendicular to \mathbf{a} .

4.4.4. Vector Products. Find the vector $\mathbf{b} = (b_x, b_y, b_z)$ that satisfies the cross product relation $\mathbf{a} \times \mathbf{b} = \mathbf{c}$, where $\mathbf{a} = (2, 1, 3)$ and $\mathbf{c} = (2, -4, 0)$. Is there only one such vector?

4.4.5. Nonassociativity of the Cross Product. Show that the cross product is not associative. That is, that $\mathbf{a} \times (\mathbf{b} \times \mathbf{c})$ is not necessarily the same as $(\mathbf{a} \times \mathbf{b}) \times \mathbf{c}$.

4.4.6. Another Useful Fact. Show by direct calculation on the components that the length of the cross product has the form:

$$|\mathbf{a} \times \mathbf{b}| = \sqrt{|\mathbf{a}|^2 |\mathbf{b}|^2 - (\mathbf{a} \cdot \mathbf{b})^2}$$

4.4.1. Geometric Interpretation of the Cross Product.

By definition the cross product $\mathbf{a} \times \mathbf{b}$ of two vectors is another vector, but how is it related geometrically to the others, and why is it of interest? Figure 4.15 gives the answer. The cross product $\mathbf{a} \times \mathbf{b}$ has the following useful properties (whose proofs are requested in the exercises):

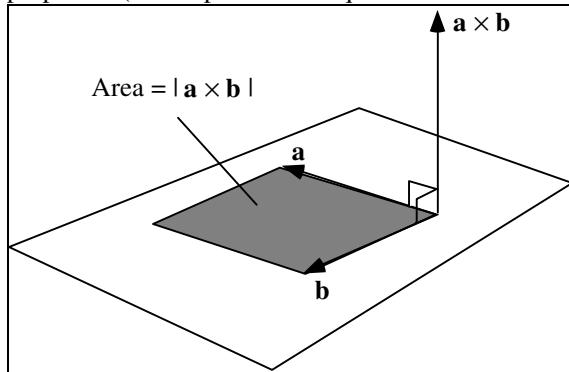


Figure 4.15. Interpretation of the cross product.

1. $\mathbf{a} \times \mathbf{b}$ is perpendicular (orthogonal) to both \mathbf{a} and \mathbf{b} .
2. The length of $\mathbf{a} \times \mathbf{b}$ equals the area of the parallelogram determined by \mathbf{a} and \mathbf{b} . This area is equal to

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}| |\mathbf{b}| \sin(\theta) \quad (4.31)$$

where θ is the angle between \mathbf{a} and \mathbf{b} , measured from \mathbf{a} to \mathbf{b} or \mathbf{b} to \mathbf{a} , whichever produces an angle less than 180 degrees. As a special case, $\mathbf{a} \times \mathbf{b} = 0$ if, and only if, \mathbf{a} and \mathbf{b} have the same or opposite directions or if either has zero length. What is the magnitude of the cross product if \mathbf{a} and \mathbf{b} are perpendicular?

3. The sense of $\mathbf{a} \times \mathbf{b}$ is given by the right-hand rule when working in a right-handed system. For example, twist the fingers of your right hand from \mathbf{a} to \mathbf{b} , and then $\mathbf{a} \times \mathbf{b}$ will point in the direction of your thumb. (When working in a left-handed system, use your left hand instead.) Note that $\mathbf{i} \times \mathbf{j} = \mathbf{k}$ supports this.

Example 4.4.2. Let $\mathbf{a} = (1, 0, 1)$ and $\mathbf{b} = (1, 0, 0)$. These vectors are easy to visualize, as they both lie in the x, z -plane. (Sketch them.) The area of the parallelogram defined by \mathbf{a} and \mathbf{b} is easily seen to be 1. Because $\mathbf{a} \times \mathbf{b}$ is orthogonal to both \mathbf{a} and \mathbf{b} , we expect it to be parallel to the y -axis and hence be proportional to $\pm \mathbf{j}$. In either a right-handed or a left-handed system, sweeping the fingers of the appropriate hand from \mathbf{a} to \mathbf{b} reveals a thumb pointed along the positive y -axis. Direct calculation based on Equation 4.28 confirms all of this: $\mathbf{a} \times \mathbf{b} = \mathbf{j}$.

|Practice Exercise 4.4.7. Proving the Properties. Prove the three properties given above for the cross product.

4.4.2. Finding the Normal to a Plane.

As we shall see in the next section, we sometimes must compute the components of the normal vector \mathbf{n} to a plane.

If the plane is known to pass through three specific points, the cross product provides the tool to accomplish this.

Any three points, P_1, P_2, P_3 , determine a unique plane, as long as the points don't lie in a straight line. Figure 4.16 shows this situation.

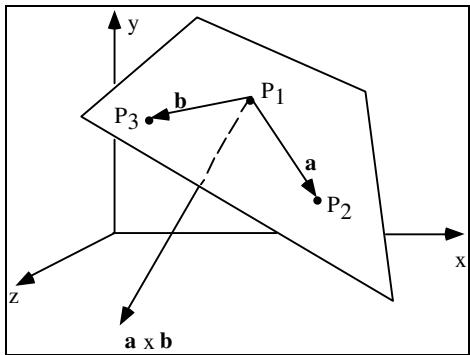


Figure 4.16. Finding the plane through three given points.

To find the normal vector, build two vectors, $\mathbf{a} = \mathbf{P}_2 - \mathbf{P}_1$ and $\mathbf{b} = \mathbf{P}_3 - \mathbf{P}_1$. Their cross product, $\mathbf{n} = \mathbf{a} \times \mathbf{b}$, must be normal to both \mathbf{a} and \mathbf{b} , so it is normal to every line in the plane (why?). It is therefore the desired normal vector. (What happens if the three points do lie in a straight line?) Any scalar multiple of this cross product is also a normal vector, including $\mathbf{b} \times \mathbf{a}$, which points in the opposite direction.

Example 4.4.3. Find the normal vector to the plane that passes through the points $(1, 0, 2)$, $(2, 3, 0)$, and $(1, 2, 4)$.

Solution: By direct calculation, $\mathbf{a} = (2, 3, 0) - (1, 0, 2) = (1, 3, -2)$, and $\mathbf{b} = (1, 2, 4) - (1, 0, 2) = (0, 2, 2)$, and so their cross product $\mathbf{n} = (10, -2, 2)$.

Note: Since a cross product involves the subtraction of various quantities (see Equation 4.28), this method for finding \mathbf{n} is vulnerable to numerical inaccuracies, especially when the angle between \mathbf{a} and \mathbf{b} is small. We develop a more robust method later for finding normal vectors in practice.

Practice Exercises.

4.4.8. Does the choice of points matter? Is the same plane obtained as in Example 4.4.3 if we use the points in a different order, say, $\mathbf{a} = (1, 0, 2) - (2, 3, 0)$ and $\mathbf{b} = (1, 2, 4) - (2, 3, 0)$? Show that the same plane does result.

4.4.9. Finding Some Planes. For each of the following triplets of points, find the normal vector to the plane (if it exists) that passes through the triplet.

- $P_1 = (1, 1, 1)$, $P_2 = (1, 2, 1)$, $P_3 = (3, 0, 4)$
- $P_1 = (8, 9, 7)$, $P_2 = (-8, -9, -7)$, $P_3 = (1, 2, 1)$
- $P_1 = (6, 3, -4)$, $P_2 = (0, 0, 0)$, $P_3 = (2, 1, -1)$
- $P_1 = (0, 0, 0)$, $P_2 = (1, 1, 1)$, $P_3 = (2, 2, 2)$

4.4.10. Finding the normal vectors. Calculate the normal vectors to each of the faces of the two objects shown in Figure 4.17. The cube has vertices $(\pm 1, \pm 1, \pm 1)$ and the tetrahedron has vertices $(0, 0, 0)$, $(0, 0, 1)$, $(1, 0, 0)$, and $(0, 1, 0)$.

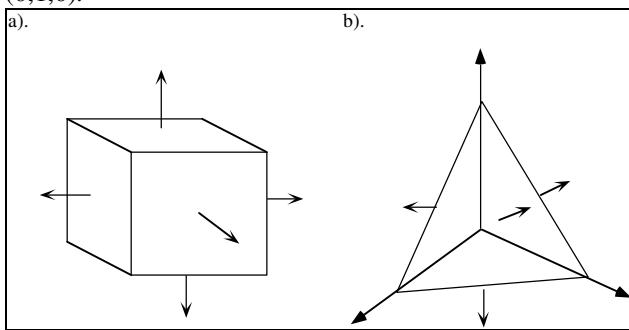


Figure 4.17. Finding the normal vectors to faces.

4.5. Representations of Key Geometric Objects.

In the preceding sections we have discussed some basic ideas of vectors and their application to important geometric problems that arise in graphics. Now we develop the fundamental ideas that facilitate working

with lines and planes, which are central to graphics, and whose “straightness” and “flatness” makes them easy to represent and manipulate.

What does it mean to “represent” a line or plane, and why is it important? The goal is to come up with a formula or equation that distinguishes points that lie on the line from those that don’t. This might be an equation that is satisfied by all points on the line, and only those points. Or it might be a function that returns different points in the line as some parameter is varied. The representation allows one to test such things as: is point P on the line?, or where does the line *intersect* another line or some other object. Very importantly, a line lying in a plane divides the plane into two parts, and we often need to ask whether point P lies on one side or the other of the line.

In order to deal properly with lines and planes we must, somewhat unexpectedly, go back to basics and review how points and vectors differ, and how each is represented. The need for this arises because, to represent a line or plane we must “add points together”, and “scale points”, operations that for points are nonsensical. To see what is really going on we introduce the notion of a coordinate frame, that makes clear the significant difference between a point and a vector, and reveals in what sense it is legitimate to “add points”. The use of coordinate frames leads ultimately to the notion of “homogeneous coordinates”, which is a central tool in computer graphics, and greatly simplifies many algorithms. We will make explicit use of coordinate frames in only a few places in the book, most notably when changing coordinate systems and “flying” cameras around a scene (see Chapters 5, 6, and 7)⁴. But even when not explicitly mentioned, an underlying coordinate frame will be present in every situation.

4.5.1. Coordinate Systems and Coordinate Frames.

One doesn’t discover new lands without consenting to lose sight of the shore for a very long time.

Andre Gide

When discussing vectors in previous sections we say, for instance, that a vector $\mathbf{v} = (3, 2, 7)$, meaning it is a certain 3-tuple. We say the same for a point, as in point $P = (5, 3, 1)$. This makes it seem that points and vectors are the same thing. But points and vectors are very different creatures: points have location but no size or direction; vectors have size and direction but no location.

What we mean by $\mathbf{v} = (3, 2, 7)$, of course, is that the vector \mathbf{v} has “components” $(3, 2, 7)$ in the underlying coordinate system. Similarly, $P = (5, 3, 1)$ means point P has coordinates $(5, 3, 1)$ in the underlying coordinate system. Normally this confusion between the object and its representation presents no problem. The problem arises when there is more than one coordinate system (a very common occurrence in graphics), and when you transform points or vectors from one system into another.

We usually think of a coordinate system as three “axes” emanating from an origin, as in Figure 4.2b. But in fact a coordinate system is “located” somewhere in “the world”, and its axes are best described by three vectors that point in mutually perpendicular directions. In particular it is important to make explicit the “location” of the coordinate system. So we extend the notion of a 3D coordinate system⁵ to that of a 3D coordinate “frame.” A **coordinate frame** consists of a specific point, ϑ , called the *origin*, and three mutually perpendicular unit vectors⁶, \mathbf{a} , \mathbf{b} , and \mathbf{c} . Figure 4.18 shows a coordinate frame “residing” at some point ϑ within “the world”, with its vectors \mathbf{a} , \mathbf{b} , and \mathbf{c} drawn so they appear to emanate from ϑ like axes.

⁴ This is an area where graphics programmers can easily go astray: their programs produce pictures that look OK for simple situations, and become mysteriously and glaringly wrong when things get more complex.

⁵ The ideas for a 2D system are essentially identical.

⁶ In more general contexts the vectors need not be mutually perpendicular, but rather only “linearly independent” (such that, roughly, none of them is a linear combination of the other two). The coordinate frames we work with will always have perpendicular axis vectors.



Figure 4.18. A coordinate frame positioned in “the world”.

Now to represent a vector \mathbf{v} we find three numbers, (v_1, v_2, v_3) such that

$$\mathbf{v} = v_1 \mathbf{a} + v_2 \mathbf{b} + v_3 \mathbf{c} \quad (4.32)$$

and say that \mathbf{v} “has the representation” (v_1, v_2, v_3) in this system.

On the other hand, to represent a point, P , we view its location as an offset from the origin by a certain amount: we represent the vector $P - \mathcal{V}$ by finding three numbers (p_1, p_2, p_3) such that:

$$P - \mathcal{V} = p_1 \mathbf{a} + p_2 \mathbf{b} + p_3 \mathbf{c}$$

and then equivalently write P itself as:

$$P = \mathcal{V} + p_1 \mathbf{a} + p_2 \mathbf{b} + p_3 \mathbf{c} \quad (4.33)$$

The representation of P is not just a 3-tuple, but a 3-tuple along with an origin. P is “at” a location that is offset from the origin by $p_1 \mathbf{a} + p_2 \mathbf{b} + p_3 \mathbf{c}$. The basic idea is to make the origin of the coordinate system *explicit*. This becomes important only when there is more than one coordinate frame, and when transforming one frame into another.

Note that when we earlier defined the standard unit vectors \mathbf{i} , \mathbf{j} , and \mathbf{k} as $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, respectively, we were actually defining their *representations* in an underlying coordinate frame. Since by Equation 4.32 $\mathbf{i} = 1\mathbf{a} + 0\mathbf{b} + 0\mathbf{c}$, vector \mathbf{i} is actually just \mathbf{a} itself! It’s a matter of naming: whether you are talking about the vector or about its representation in a coordinate frame. We usually don’t bother to distinguish them.

Note that you can’t explicitly say where \mathcal{V} is, or cite the directions of \mathbf{a} , \mathbf{b} , and \mathbf{c} : To do so requires having some other coordinate frame in which to represent this one. In terms of its own coordinate frame, \mathcal{V} has the representation $(0, 0, 0)$, \mathbf{a} has the representation $(1, 0, 0)$, etc.

The homogeneous representation of a point and a vector.

It is useful to represent both points and vectors using the *same* set of basic underlying objects, $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathcal{V})$. From Equations 4.32 and 4.33 the vector $\mathbf{v} = v_1 \mathbf{a} + v_2 \mathbf{b} + v_3 \mathbf{c}$ then needs the four coefficients $(v_1, v_2, v_3, 0)$ whereas the point $P = p_1 \mathbf{a} + p_2 \mathbf{b} + p_3 \mathbf{c} + \mathcal{V}$ needs the four coefficients $(p_1, p_2, p_3, 1)$. The fourth component designates whether the object does or does not include \mathcal{V} . We can formally write any \mathbf{v} and P using a matrix multiplication (multiplying a row vector by a column vector - see Appendix 2):

$$\mathbf{v} = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \vartheta) \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix} \quad (4.34)$$

$$P = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \vartheta) \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} \quad (4.35)$$

Here the row matrix captures the nature of the coordinate frame, and the column vector captures the representation of the specific object of interest. Thus vectors and points have different representations: there is a fourth component of 0 for a vector and 1 for a point. This is often called the **homogeneous representation**.⁷ The use of homogeneous coordinates is one of the hallmarks of computer graphics, as it helps to keep straight the distinction between points and vectors, and provides a compact notation when working with affine transformations. It pays off in a computer program to represent the points and vectors of interest in homogeneous coordinates as 4-tuples, by appending a 1 or 0⁸. This is particularly true when we must convert between one coordinate frame and another in which points and vectors are represented.

It is simple to convert between the “ordinary” representation of a point or vector (a 3-tuple for 3D objects or a 2-tuple for 2D objects) and the homogeneous form:

To go from ordinary to homogeneous coordinates:

if it's a point append a 1;
if it's a vector, append a 0;

To go from homogeneous coordinates to ordinary coordinates:

If it's a vector its final coordinate is 0. Delete the 0.
If it's a point its final coordinate is 1 Delete the 1.

OpenGL uses 4D homogeneous coordinates for all its vertices. If you send it a 3-tuple in the form (x, y, z) , it converts it immediately to $(x, y, z, 1)$. If you send it a 2D point (x, y) , it first appends a 0 for the z-component and then a 1, to form $(x, y, 0, 1)$. All computations are done within OpenGL in 4D homogeneous coordinates.

Linear Combinations of Vectors .

Note how nicely some things work out in homogeneous coordinates when we combine vectors coordinate-wise: all the definitions and manipulations are consistent:

- The difference of two points $(x, y, z, 1)$ and $(u, v, w, 1)$ is $(x - u, y - v, z - w, 0)$, which is, as expected, a vector.
- The sum of a point $(x, y, z, 1)$ and a vector $(d, e, f, 0)$ is $(x + d, y + e, z + f, 1)$, another point;
- Two vectors can be added: $(d, e, f, 0) + (m, n, r, 0) = (d + m, e + n, f + r, 0)$ which produces another vector;
- It is meaningful to scale a vector: $3(d, e, f, 0) = (3d, 3e, 3f, 0)$;

⁷ Actually we are only going part of the way in this discussion. As we see in Chapter 7 when studying projections, homogeneous coordinates in that context permit an additional operation, which makes them truly “homogeneous”. Until we examine projections this operation need not be introduced.

⁸ In the 2D case, points are 3-tuples $(p_1, p_2, 1)$ and vectors are 3-tuples $(v_1, v_2, 0)$.

- It is meaningful to form *any* linear combination of vectors. Let the vectors be $\mathbf{v} = (v_1, v_2, v_3, 0)$ and $\mathbf{w} = (w_1, w_2, w_3, 0)$. Then using arbitrary scalars a and b , we form $a\mathbf{v} + b\mathbf{w} = (av_1 + bw_1, av_2 + bw_2, av_3 + bw_3, 0)$, which is a legitimate vector.

Forming a linear combination of vectors is well defined, but does it make sense for points? The answer is no, except in one special case, as we explore next.

4.5.2. Affine Combinations of Points.

Consider forming a linear combination of two points, $P = (P_1, P_2, P_3, 1)$ and $R = (R_1, R_2, R_3, 1)$, using the scalars f and g :

$$fP + gR = (fP_1 + gR_1, fP_2 + gR_2, fP_3 + gR_3, f + g)$$

We know this is a legitimate vector if $f + g = 0$ (why?). But we shall see that it is *not* a legitimate point unless $f + g = 1$! Recall from Equation 4.2 that when the coefficients of a linear combination sum to 1 it is called an “affine” combination. So we see that the only linear combination of points that is legitimate is an affine combination. Thus, for example, the object $0.3P + 0.7R$ is a legitimate point, as are $2.7P - 1.7R$ and the midpoint $0.5P + 0.5R$, but $P + R$ is not a point. For three points, P , R , and Q we can form the legal point $0.3P + 0.9R - 0.2Q$, but not $P + Q - 0.9R$.

Fact: any affine combination of points is a legitimate point.

But what's wrong geometrically with forming *any* linear combination of two points, say

$$E = fP + gR \quad (4.36)$$

when $f + g$ is different from 1? The problem arises if we shift the origin of the coordinate system [Goldman85]. Suppose the origin is shifted by vector \mathbf{u} , so that P is altered to $P + \mathbf{u}$ and R is shifted to $R + \mathbf{u}$. If E is a legitimate point, it too must be shifted to the new point $E' = E + \mathbf{u}$. But instead we have

$$E' = fP + gR + (f + g)\mathbf{u}$$

which is *not* $E + \mathbf{u}$ unless $f + g = 1$.

The failure of a simple sum $P_1 + P_2$ of two points to be a true point is shown in Figure 4.19. Points P_1 and P_2 are shown represented in two coordinate systems, one offset from the other. Viewing each point as the head of a vector bound to its origin, we see that the sum $P_1 + P_2$ yields two different points in the two systems. Therefore $P_1 + P_2$ depends on the choice of coordinate system. Note, by way of contrast, that the affine combination $0.5(P_1 + P_2)$ does *not* depend on this choice.

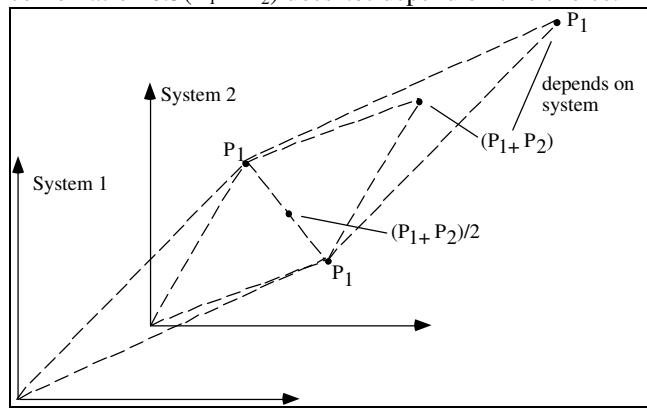


Figure 4.19. Adding points is not legal.

A Point plus a Vector is an Affine Combination of Points.

There is another way of examining affine sums of points that is interesting on its own, and also leads to a useful tool in graphics. It doesn't require the use of homogeneous coordinates.

Consider forming a point as a point A offset by a vector \mathbf{v} that has been scaled by scalar t : $A + t\mathbf{v}$. This is the sum of a point and a vector so it is a legitimate point. If we take as vector \mathbf{v} the difference between some other point B and A : $\mathbf{v} = B - A$ then we have the point P :

$$P = A + t(B - A) \quad (4.37)$$

which is also a legitimate point. But now rewrite it algebraically as:

$$P = tB + (1 - t)A \quad (4.38)$$

and it is seen to be an affine combination of points (why?). This further legitimizes writing affine sums of points. In fact, any affine sum of points can be written as a point plus a vector (see the exercises). If you are ever uncomfortable writing an affine sum of points as in Equation 4.38 (a form we will use often), simply understand that it *means* the point given by Equation 4.37.

Example 4.5.1: The centroid of a triangle. Consider the triangle T with vertices A , B , and C shown in Figure 4.20. We use the ideas above to show that the three **medians** of T meet at a point that lies $2/3$ of the way along each median. This is the centroid (center of gravity⁹) of T .

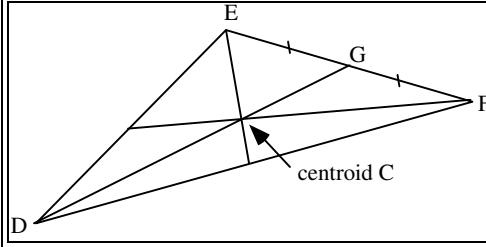


Figure 4.20. The centroid of a triangle as an affine combination.

By definition the median from D is the line from D to the midpoint of the opposite side. Thus $G = (E + F)/2$. We first ask where the point that is $2/3$ of the way from D to G lies? Using the parametric form the desired point must be $D + (G - D)t$ with $t = 2/3$, which yields the affine combination C given by

$$C = \frac{D + E + F}{3}$$

(Try it!) Here's the cute part [pedoe70]. Since this result is *symmetrical* in D , E , and F , it must also be $2/3$ of the way along the median from E , and $2/3$ of the way along the median from F . Hence the 3 medians meet there, and C is the centroid.

This result generalizes nicely for a regular polygon of N sides: the centroid is simply the average of the N vertex locations, another affine combination. For an arbitrary polygon the formula is more complex

Practice Exercises.

4.5.1. Any affine combination of points is legitimate. Consider three scalars a , b , and c that sum to one, and three points A , B , and C . The affine combination $aA + bB + cC$ is a legal point because using $c = 1 - a - b$ it is seen to be the same as $aA + bB + (1 - a - b)C = C + a(A - C) + b(B - C)$, the sum of a point and two vectors (check this out!). To generalize: Given the affine combination of points $w_1A_1 + w_2A_2 + \dots + w_nA_n$, where $w_1 + w_2 + \dots + w_n = 1$, show that it can be written as a point plus a vector, and is therefore a legitimate point.

4.5.2. Shifting the coordinate system [Goldman85]. Consider the general situation of forming a linear combination of m points:

⁹The reference to gravity arises because if a thin plate is cut in the shape of T , the plate hangs level if suspended by a thread attached at the centroid. Gravity pulls equally on all sides of the centroid, so the plate is balanced.

$$E = \sum_{i=1}^m a_i P_i$$

We ask whether E is a point, a vector, or nothing at all? By considering the effect of a shift in each P_i by \mathbf{u} show that E is “shifted” to $E' = E + S \mathbf{u}$, where S is the sum of the coefficients:

$$S = \sum_{i=1}^m a_i$$

Show that:

- i). E is a point if $S = 1$.
- ii). E is a vector if $S = 0$.
- iii). E is meaningless for other values of S .

4.5.3. Linear Interpolation of two points.

The affine combination of points expressed in Equation 4.33:

$$P = A(I - t) + Bt$$

performs **linear interpolation** between the points A and B . That is, the x -component $P_x(t)$ provides a value that is fraction t of the way between the value A_x and B_x , and similarly for the y -component (and in 3D the z -component). This is a sufficiently important operation to warrant a name, and `lerp()` (for linear interpolation) has become popular. In one dimension, `lerp(a, b, t)` provides a number that is the fraction t of the way from a to b . Figure 4.21 provides a simple implementation of `lerp()`.

```
float lerp(float a, float b, float t)
{
    return a + (b - a) * t; // return a float
}
```

Figure 4.21. Linear interpolation effected by `lerp()`.

Similarly, one often wants to compute the point $P(t)$ that is fraction t of the way along the straight line from point A to point B . This point is often called the “tween” (for “in-between”) at t of points A and B . Each component of the resulting point is formed as the `lerp()` of the corresponding components of A and B . A procedure

```
Point2 canvas:: Tween(Point2 A, Point2 B, float t) // tween A and B
```

is easily written (how?) to implement tweening. A 3D version is almost the same.

Example 4.5.2. Let $A = (4, 9)$ and $B = (3, 7)$. Then `Tween(A, B, t)` returns the point $(4 - t, 9 - 2t)$, so that `Tween(A, B, 0.4)` returns $(3.6, 8.1)$. (Check this on graph paper.)

4.5.3. “Tweening” for Art and Animation.

Interesting animations can be created that show one figure being “tweened” into another. It’s simplest if the two figures are polylines (or families of polylines) based on the same number of points. Suppose the first figure, A , is based on the polyline with points A_i , and the second polyline, B , is based on points B_i , for $i = 0, \dots, n-1$. We can form the polyline $P(t)$, called the “tween at t ”, by forming the points:

$$P_i(t) = (1 - t) A_i + t B_i$$

If we look at a succession of values for t between 0 and 1, say, $t = 0, 0.1, 0.2, \dots, 0.9, 1.0$, we see that this polyline begins with the shape of A and ends with the shape of B , but in between it is a blend of the two

shapes. For small values of t it looks like A , but as t increases it warps (smoothly) towards a shape close to B . For $t = 0.25$, for instance, point $P_1(0.25)$ of the tween is 25% of the way from A to B .

Figure 4.22 shows a simple example, in which polyline A has the shape of a house, and polyline B has the shape of the letter ‘T’. The point R on the house corresponds to point S on the ‘T’. The various tweens of point R on the house and point S on the T lie on the line between R and S . The tween for $t = 1/2$ lies at the midpoint of RS . The in between polylines show the shapes of the tweens for $t = 0, 0.25, 0.5, 0.75$, and 1.0 .

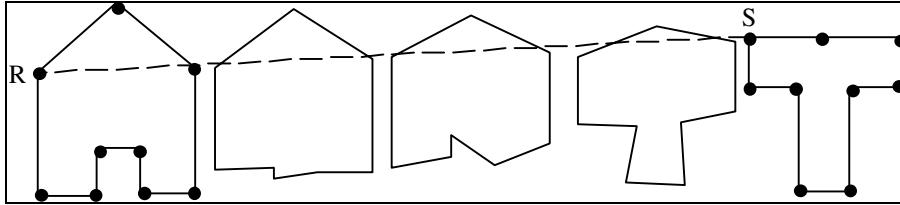


Figure 4.22. Tweening a "T" into a house.

Figure 4.23 shows `drawTween()`, that draws a tween of two polylines A and B , each having n vertices, at the specified value of t .

```
void canvas:: drawTween(Point2 A[], Point2 B[], int n, float t)
{   // draw the tween at time t between polylines A and B
    for(int i = 0; i < n; i++)
    {
        Point2 P;
        P = Tween(A[i], B[i],t);
        if(i == 0)      moveTo(P.x, P.y);
        else           lineTo(P.x, P.y);
    }
}
```

Figure 4.23. Tweening two Polylines.

`drawTween()` could be used in an animation loop that tweens A and B back and forth, first as t increases from 0 to 1, then as t decreases back to 0, etc. Double buffering, as discussed in Chapter 3, is used to make the transition from one displayed tween to the next instantaneous.

```
for(t = 0.0, delT = 0.1; ; t += delT) // tween back and forth forever
{
    <clear the buffer>
    drawTween(A, B, n, t);
    glutSwapBuffers();
    if( t >= 1.0 || t <= 0.0) delT = - delT; // reverse the flow of t
}
```

Figure 4.24 shows an artistic use of this technique based on two sets of polylines. Three tweens are shown (what values of t are used?). Because the two sets of polylines are drawn sufficiently far apart, there is room to draw the tweens between them with no overlap, so that all five pictures fit nicely on one frame.

see Figure 7.11 from first edition

Figure 4.24. From man to woman. (Courtesy of Marc Infield.)

Susan E. Brennan of Hewlett Packard in Palo Alto, California, has produced caricatures of famous figures using this method (see [dewdney88]). Figure 4.25 shows an example. The second and fourth faces are based on digitized points for Elizabeth Taylor and John F. Kennedy. The third face is a tween, and the other three are based on **extrapolation**. That is, values of t larger than 1 are used, so that the term $(1 - t)$ is negative. Extrapolation can produce caricature-like distortions, in some sense “going to the other side” of polyline B from polyline A . Values of t less than 0 may also be used, with a similar effect.

see Figure 7.12 from 1st edition: Elizabeth Taylor to J.F. Kennedy

Figure 4.25. Face Caricature: Tweening and extrapolation. (Courtesy of Susan Brennan.)

Tweening is used in the film industry to reduce the cost of producing animations such as cartoons. In earlier days an artist had to draw 24 pictures for each second of film, because movies display 24 frames per second. With the assistance of a computer, however, an artist need draw only the first and final pictures, called **key-frames**, in certain sequences and let the others be generated automatically. For instance, if the characters are not moving too rapidly in a certain one-half-second portion of a cartoon, the artist can draw and digitize the first and final frames of this portion, and the computer can create 10 tweens using linear interpolation, thereby saving a great deal of the artist's time. See the case study at the end of this chapter for a programming project that produces these effects.

Practice Exercises.

4.5.3. A Limiting Case of Tweening. What is the effect of tweening when all of the points A_i in polyline A are the same? How is polyline B distorted in its appearance in each tween?

4.5.4. An Extrapolation. Polyline A is a square with vertices $(1, 1)$, $(-1, 1)$, $(-1, -1)$, $(1, -1)$, and polyline B is a wedge with vertices $(4, 3)$, $(5, -2)$, $(4, 0)$, $(3, -2)$. Sketch (by hand) the shape $P(t)$ for $t = -1, -0.5, 0.5$, and 1.5 .

4.5.5. Extrapolation Versus Tweening. Suppose that five polyline pictures are displayed side by side. From careful measurement you determine that the middle three are in-betweens of the first and the last, and you calculate the values of t used. But someone claims that the last is actually an extrapolation of the first and the fourth. Is there any way to tell whether this is true? If it is an extrapolation, can the value of t used be determined? If so, what is it?

4.5.4. Preview: Quadratic and cubic tweening, and Bezier Curves.

In Chapter 8 we address the problem of designing complex shapes called Bezier curves. It is interesting to note here that the underlying idea is simply tweening between a collection of points. With linear interpolation above we “partition unity” into the pieces $(1 - t)$ and t , and use these pieces to “weight” the points A and B . We can extend this to quadratic interpolation by partitioning unity into three pieces. Just rewrite 1 as

$$1 = ((1-t) + t)^2$$

and expand it to produce the three pieces $(1 - t)^2$, $2(1 - t)t$, and t^2 . They obviously sum to one, so they can be used to form the affine combination of points A , B , and C :

$$P(t) = (1 - t)^2 A + 2(1 - t)t B + t^2 C$$

This is the “Bezier curve” for the points A , B , and C . Figure 4.26a shows the shape of $P(t)$ as t varies from 0 to 1. It flows smoothly from A to C . (Notice that the curve misses the middle point.) Going further, one can expand $((1 - t) + t)^3$ into four pieces (which ones?) which can be used to do “cubic interpolation” between four points A , B , C and D , as shown in Figure 4.26b.

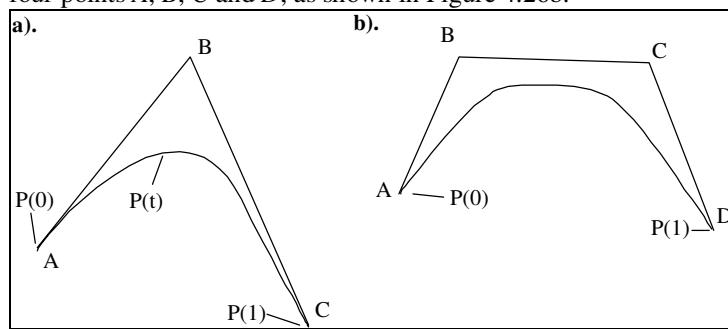


Figure 4.26. Bezier curves as Tweening.

Practice Exercise 4.5.6. Try it out. Draw three points A , B , and C on a piece of graph paper. For each of the values $t = 0, .1, .2, \dots, .9, 1$ compute the position of $P(t)$ in Equation 4.38, and draw the polyline that passes through these points. Is it always a parabola?

4.5.5. Representing Lines and Planes.

We now turn to developing the principal forms in which lines and planes are represented mathematically. It is quite common to find data structures within a graphics program that capture a line or plane using one of these forms.

Lines in 2D and 3D space.

A **line** is defined by two points, say C and B (see Figure 4.27a). It is infinite in length, passing through the points and extending forever in both directions. A **line segment** (**segment** for short) is also defined by two points, its **endpoints**, but extends only from one endpoint to the other (Figure 4.27b). Its **parent** line is the infinite line that passes through its endpoints. A **ray** is “semi-infinite.” It is specified by a point and a direction. It “starts” at a point and extends infinitely far in a given direction (Figure 4.27c).

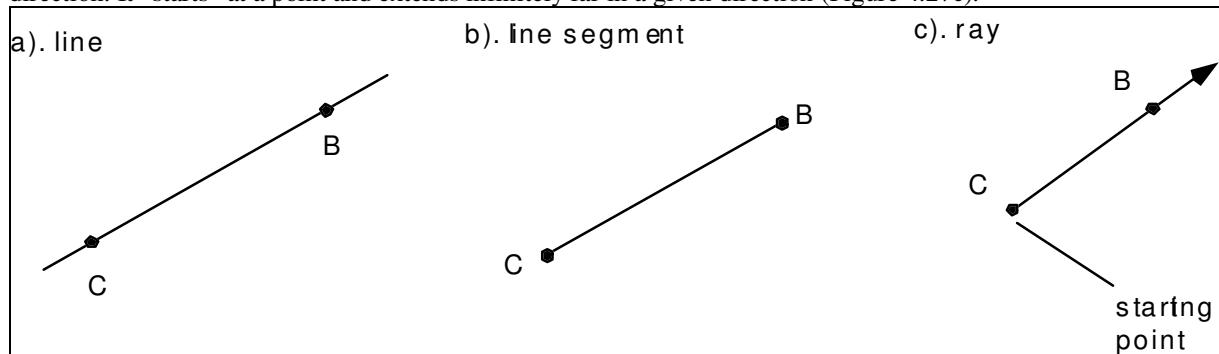


Figure 4.27. Lines, segments, and rays.

These objects are very familiar, yet it is useful to collect their important representations and properties in one spot. We also describe the most important representation of all for a line in computer graphics, the **parametric representation**.

The parametric representation of a line.

The construction in Equations 4.32 and 4.33 is very useful, because as t varies the point P traces out all of the points on the straight line defined by C and B . The construction therefore gives us a way to name and compute any point along this line.

This is done using a **parameter** t that distinguishes one point on the line from another. Call the line L , and give the name $L(t)$ to the position associated with t . Using $\mathbf{b} = B - C$ we have:

$$L(t) = C + \mathbf{b} t \quad (4.39)$$

As t varies so does the position of $L(t)$ along the line. (One often thinks of t as “time”, and uses language such as: “at time 0 ...”, “as time goes on...”, or “later” to describe different parts of the line.) Figure 4.28 shows vector \mathbf{b} and the line L passing through C and B . (A 2D version is shown but the 3D version uses the same ideas.) Note where $L(t)$ is located for various values of t . If $t = 0$, $L(0)$ evaluates to C so at $t = 0$ we are “at” point C . At $t = 1$ then $L(1) = C + (B - C) = B$. As t varies we add a longer or shorter version of \mathbf{b} to the point C , resulting in a new point along the line. If t is larger than 1 this point lies somewhere on the opposite side of C from B , and when t is less than 0 it lies on the side of C opposite from B .

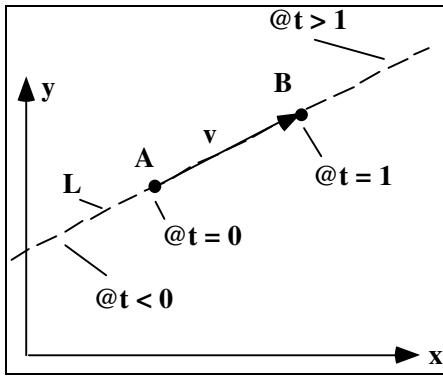


Figure 4.28. Parametric representation $L(t)$ of a line.

For a fixed value of t , say $t = 0.6$, Equation 4.39 gives a formula for exactly one point along the line through C and B : the particular point $L(0.6)$. Thus it is a description of a point. But since one can view it as a function of t that generates the coordinates of *every* point on L as t varies, it is called the **parametric representation of line L** .

The line, ray, and segment of Figure 4.26 are all represented by the same $L(t)$ of Equation 4.39. They differ parametrically only in the values of t that are relevant:

$$\begin{aligned} \text{segment : } & 0 \leq t \leq 1 \\ \text{ray: } & 0 \leq t < \infty \\ \text{line : } & -\infty < t < \infty \end{aligned} \tag{4.40}$$

The ray “starts” at C when $t = 0$ and passes through B at $t = 1$, then continues forever as t increases. C is often called the “starting point” of the ray.

A very useful fact is that $L(t)$ lies “fraction t of the way” between C and B when t lies between 0 and 1. For instance, when $t = 1/2$ the point $L(0.5)$ is the **midpoint** between C and B , and when $t = 0.3$ the point $L(0.3)$ is 30% of the way from C to B . This is clear from Equation 4.39 since $|L(t) - C| = |\mathbf{b}| |t|$ and $|B - C| = |\mathbf{b}|$, so the value of $|t|$ is the ratio of the distances $|L(t) - C|$ to $|B - C|$, as claimed.

One can also speak of the “speed” with which the point $L(t)$ “moves” along line L . Since it covers distance $|\mathbf{b}| t$ in time t it is moving at constant speed $|\mathbf{b}|$.

Example 4.5.2. A line in 2D. Find a parametric form for the line that passes through $C = (3, 5)$ and $B = (2, 7)$. **Solution:** Build vector $\mathbf{b} = B - C = (-1, 2)$ to obtain the parametric form $L(t) = (3 - t, 2 + 2t)$.

Example 4.5.3. A line in 3D. Find a parametric form for the line that passes through $C = (3, 5, 6)$ and $B = (2, 7, 3)$. **Solution:** Build vector $\mathbf{b} = B - C = (-1, 2, -3)$ to obtain the parametric form $L(t) = (3 - t, 2 + 2t, 6 - 3t)$.

Other parametrizations for a straight line are possible, although they are rarely used. For instance, the point $W(t)$ given by

$$W(t) = C + \mathbf{b}t^3$$

also “sweeps” over every point on L . It lies at C when $t = 0$, and reaches B when $t = 1$. Unlike $L(t)$, however, $W(t)$ “accelerates” along its path from C to B .

Point normal form for a line (the implicit form).

This is the same as the equation for a line, but we rewrite it in a way that better reveals the underlying geometry. The familiar equation of a line in 2D has the form

$$fx + gy = 1 \tag{4.41}$$

where f and g are some constants. The notion is that every point (x, y) that satisfies this equation lies on the line, so it provides a condition for a point to be on the line. Note: This is true only for a line in 2D; a line in 3D requires two equations. So, unlike the parametric form that works perfectly well in 2D and 3D, the point normal form only applies to lines in 2D.

This equation can be written using a dot product: $(f, g) \cdot (x, y) = 1$, so for every point on a line a certain dot product must have the same value. We examine the geometric interpretation of the “vector” (f, g) , and in so doing develop the “point normal” form of a line. It is very useful in such tasks as clipping, hidden line elimination, and ray tracing. Formally the point normal form makes no mention of dimensionality: A line in 2D has a point normal form, and a plane in 3D has one.

Suppose that we know line L passes through points C and B , as in Figure 4.29. What is its point normal form? If we can find a vector \mathbf{n} that is perpendicular to the line, then for any point $R = (x, y)$ on the line the vector $R - C$ must be perpendicular to \mathbf{n} , so we have the condition on R :

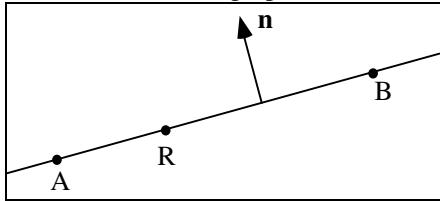


Figure 4.29. Finding the point normal form for a line.

$$\mathbf{n} \cdot (R - C) = 0 \quad (\text{point normal form}) \quad (4.42)$$

This is the **point normal** equation for the line, expressing that a certain dot product must turn out to be zero for *every* point R on the line. It employs as data *any* point lying on the line, and *any* normal vector to the line.

We still must find a suitable \mathbf{n} . Let $\mathbf{b} = B - C$ denote the vector from C to B . Then \mathbf{b}^\perp will serve well as the desired \mathbf{n} . For purposes of building the point normal form, any scalar multiple of \mathbf{b}^\perp works just as well for \mathbf{n} .

Example 4.5.4. Find the point normal form. Suppose line L passes through points $C = (3, 4)$ and $B = (5, -2)$. Then $\mathbf{b} = B - C = (2, -6)$ and $\mathbf{b}^\perp = (6, 2)$ (sketch this). Choosing C as the point on the line, the point normal form is: $(6, 2) \cdot ((x, y) - (3, 4)) = 0$, or $6x + 2y = 26$. Both sides of the equation can be divided by 26 (or any other nonzero number) if desired.

It's also easy to find the normal to a line given the equation of the line, say, $fx + gy = 1$. Writing this once again as $(f, g) \cdot (x, y) = 1$ it is clear that the normal \mathbf{n} is simply (f, g) (or any multiple thereof). For instance, the line given by $5x - 2y = 7$ has normal vector $(5, -2)$, or more generally $K(5, -2)$ for any nonzero K .

It's also straightforward to find the parametric form for a line if you are given its point normal form.

Suppose it is known that line L has point normal form $\mathbf{n} \cdot (P - C) = 0$, where \mathbf{n} and C are given explicitly.

The parametric form is then $L(t) = C + \mathbf{n}^\perp t$ (why?). You can also obtain the parametric form if the equation of the line is given. a). find the normal \mathbf{n} as in the previous paragraph, and b). find a point (C_x, C_y) on the line by choosing any value for C_x and use the equation to find the corresponding C_y .

Moving from each representation to the others.

We have described three different ways to characterize a line. Each representation uses certain data that distinguishes one line from another. This is the data that would be stored in a suitable data structure within a program to capture the specifics of each line being stored. For instance, the data associated with the representation that specifies a line parametrically as in $C + \mathbf{b}t$ would be the point C and the direction \mathbf{b} . We summarize this by saying the relevant data is $\{C, \mathbf{b}\}$.

The three representations and their data are:

- The two point form: say C and B ; data = $\{C, B\}$
- The parametric form: $C + \mathbf{b}t$; data = $\{C, \mathbf{b}\}$.
- The point normal (implicit) form (in 2D only): $\mathbf{n} \cdot (P - C) = 0$; data = $\{C, \mathbf{n}\}$.

Note that a point C on the line is common to all three forms. Figure 4.30 shows how the data in each representation can be obtained from the data in the other representations. For instance, given $\{C, \mathbf{b}\}$ of the parametric form, the normal \mathbf{n} of the point normal form is obtained simply as \mathbf{b}^\perp .

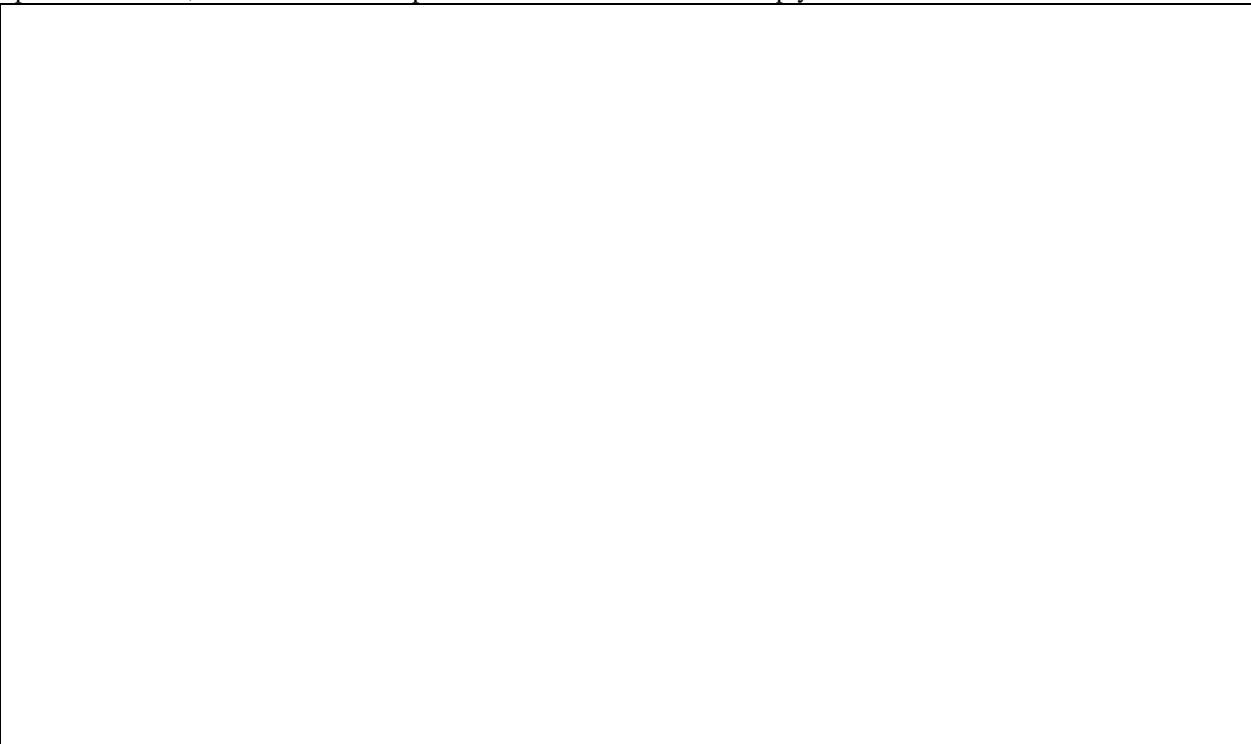


Figure 4.30. Moving between representations of a line.

Practice Exercise 4.5.5. Find the point normal form. Find the point normal form for the line that passes through $(-3, 4)$ and $(6, -1)$. Sketch the line and its normal vector on graph paper.

Planes in 3D space.

Because there is such a heavy use of polygons in 3D graphics, planes seem to appear everywhere. A polygon (a “face” of an object) lies in its “parent” plane, and we often need to clip objects against planes, or find the plane in which a certain face lies.

Planes, like lines, have three fundamental forms: the three-point form, the parametric representation and the point normal form. We examined the three-point form in Section 4.4.2.

The parametric representation of a plane.

The parametric form for a plane is built on three ingredients: one of its points, C , and two (nonparallel) vectors, \mathbf{a} and \mathbf{b} , that lie in the plane, as shown in Figure 4.31. If we are given the three (non-collinear) points A , B , and C in the plane, then take $\mathbf{a} = A - C$ and $\mathbf{b} = B - C$.

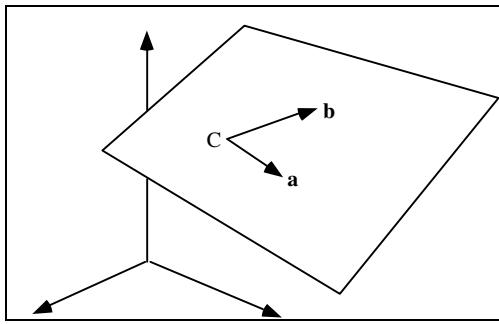


Figure 4.31. Defining a plane parametrically.

To construct a parametric form for this plane, note that any point in the plane can be represented by a vector sum: C plus some multiple of \mathbf{a} plus some multiple of \mathbf{b} . Using parameters s and t to specify the “multiples” we have $C + s \mathbf{a} + t \mathbf{b}$. This provides the desired parametric form $P(s, t)$

$$P(s, t) = C + \mathbf{a} s + \mathbf{b} t \quad (4.43)$$

Given any values of s and t we can identify the corresponding point on the plane. For example, the position “at” $s = t = 0$ is C itself, and that at $s = 1$ and $t = -2$ is $P(1, -2) = C + \mathbf{a} - 2 \mathbf{b}$.

Note that two parameters are involved in the parametric expression for a surface, whereas only one parameter is needed for a curve. In fact if one of the parameters is fixed, say $s = 3$, then $P(3, t)$ is a function of one variable and represents a straight line: $P(3, t) = (C + 3 \mathbf{a}) + \mathbf{b} t$.

It is sometimes handy to arrange the parametric form into its “component” form by collecting terms

$$P(s, t) = (C_x + a_x s + b_x t, C_y + a_y s + b_y t, C_z + a_z s + b_z t). \quad (4.44)$$

We can rewrite the parametric form in Equation 4.43 explicitly in terms of the given points A , B , and C : just use the definitions of \mathbf{a} and \mathbf{b} :

$$P(s, t) = C + s(A - C) + t(B - C)$$

which can be rearranged into the *affine combination* of points:

$$P(s, t) = sA + tB + (1 - s - t)C \quad (4.45)$$

Example 4.5.6. Find a parametric form given three points in a plane. Consider the plane passing through $A = (3, 3, 3)$, $B = (5, 5, 7)$, and $C = (1, 2, 4)$. From Equation 4.43 it has parametric form $P(s, t) = (1, 2, 4) + (2, 1, -1)s + (4, 3, 3)t$. This can be rearranged to the component form: $P(s, t) = (1 + 2s + 4t)\mathbf{i} + (2s + 3t)\mathbf{j} + (4s + 3t)\mathbf{k}$, or to the affine combination form $P(s, t) = s(3, 3, 3) + t(5, 5, 7) + (1 - s - t)(1, 2, 4)$.

The point normal form for a plane.

Planes can also be represented in point normal form, and the classic equation for a plane emerges at once.

Figure 4.32 shows a portion of plane P in three dimensions. A plane is completely specified by giving a single point, $B = (b_x, b_y, b_z)$, that lies within it, and the normal direction, $\mathbf{n} = (n_x, n_y, n_z)$, to the plane. Just as the normal vector to a line in two dimensions orients the line, the normal to a plane orients the plane in space.

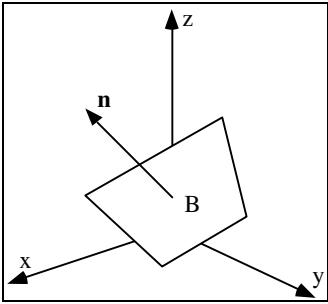


Figure 4.32. Determining the equation of a plane.

The normal \mathbf{n} is understood to be perpendicular to any line lying in the plane. For an arbitrary point $R = (x, y, z)$ in the plane, the vector from R to B must be perpendicular to \mathbf{n} , giving:

$$\mathbf{n} \cdot (R - B) = 0 \quad (4.46)$$

This is the point normal equation of the plane. It is identical in form to that for the line: a dot product set equal to 0. All points in a plane form vectors with B that have the same dot product with the normal vector. By spelling out the dot product and using $\mathbf{n} = (n_x, n_y, n_z)$, we see that the point normal form is the traditional equation for a plane:

$$n_x x + n_y y + n_z z = D \quad (4.47)$$

where $D = \mathbf{n} \cdot (B - 0)$. For example, if given the equation for a plane such as $5x - 2y + 8z = 2$, you know immediately that the normal to this plane is $(5, -2, 8)$ or any multiple of this. (How do you find a point in this plane?)

Example 4.5.7. Find a point normal form. Let plane P pass through $(1, 2, 3)$ with normal vector $(2, -1, -2)$.

Its point normal form is $(2, -1, -2) \cdot ((x, y, z) - (1, 2, 3)) = 0$. The equation for the plane may be written out as $2x - y - 2z = 6$.

Example 4.5.8. Find a parametric form given the equation of the plane. Find a parametric form for the plane $2x - y + 3z = 8$. **Solution:** By inspection the normal is $(2, -1, 3)$. There are many parametrizations; we need only find one. For C , choose any point that satisfies the equation; $C = (4, 0, 0)$ will do. Find two (noncollinear) vectors, each having a dot product of 0 with $(2, -1, 3)$; some hunting finds that $\mathbf{a} = (1, 5, 1)$ and $\mathbf{b} = (0, 3, 1)$ will work. Thus the plane has parametric form $P(s, t) = (4, 0, 0) + (1, 5, 1)s + (0, 3, 1)t$.

Example 4.5.9. Finding two noncollinear vectors. Given the normal \mathbf{n} to a plane, what is an easy way to find two noncollinear vectors \mathbf{a} and \mathbf{b} that are both perpendicular to \mathbf{n} ? (In the previous exercise we just invented two that work.) Here we use the fact that the cross product of *any* vector with \mathbf{n} is normal to \mathbf{n} . So we take a simple choice such as $(0, 0, 1)$, and construct \mathbf{a} as its cross product with \mathbf{n} :

$$\mathbf{a} = (0, 0, 1) \times \mathbf{n} = (-n_y, n_x, 0)$$

(Is this indeed normal to \mathbf{n} ?). We can use the same idea to form \mathbf{b} that is normal to both \mathbf{n} and \mathbf{a} :

$$\mathbf{b} = \mathbf{n} \times \mathbf{a} = (-n_x n_z, -n_y n_z, n_x^2 + n_y^2)$$

(Check that $\mathbf{b} \perp \mathbf{a}$ and $\mathbf{b} \perp \mathbf{n}$.) So \mathbf{b} is certainly not collinear with \mathbf{a} .

We apply this method to the plane $(3, 2, 5) \cdot (R - (2, 7, 0)) = 0$. Set $\mathbf{a} = (0, 0, 1) \times \mathbf{n} = (-2, 3, 0)$ and $\mathbf{b} = (-15, -10, 13)$. The plane therefore has parametric form:

$$P(s, t) = (2 - 2s - 15t, 7 + 3s - 10t, 13t).$$

Check: Is $P(s, t) - C = (-2s - 15t, -3s - 10t, 13t)$ indeed normal to \mathbf{n} for every s and t ?

Practice Exercise 4.5.7. Find the Plane. Find a parametric form for the plane coincident with the y, z -plane.

Moving from each representation to the others.

Just as with lines, it is useful to be able to move between the three representations of a plane, to manipulate the data that describes a plane into the form best suited to a problem.

For a plane, the three representations and their data are:

- The three point form: say C , B , and A ; data = $\{C, B, A\}$
- The parametric form: $C + \mathbf{as} + \mathbf{bt}$; data = $\{C, \mathbf{a}, \mathbf{b}\}$.
- The point normal (implicit) form: $\mathbf{n} \cdot (P - C) = 0$; data = $\{C, \mathbf{n}\}$.

A point C on the plane is common to all three forms. Figure 4.33 shows how the data in each representation can be obtained from the data in the other representations. Check each one carefully. Most of these cases have been developed explicitly in Section 4.4.2 and this section. Some are developed in the exercises. The trickiest is probably the calculation in Example 4.5.10. Another that deserves some explanation is finding three points in a plane when given the point normal form. One point, C , is already known. The other two are found using special values in the point normal form itself, which is the equation $n_x x + n_y y + n_z z = \mathbf{n} \cdot C$. Choose, for convenience, $A = (0, 0, a_z)$, and use the equation to determine that $a_z = \mathbf{n} \cdot C / n_z$. Similarly, choose $B = (0, b_y, 0)$, and use the equation to find $b_y = \mathbf{n} \cdot C / n_y$.

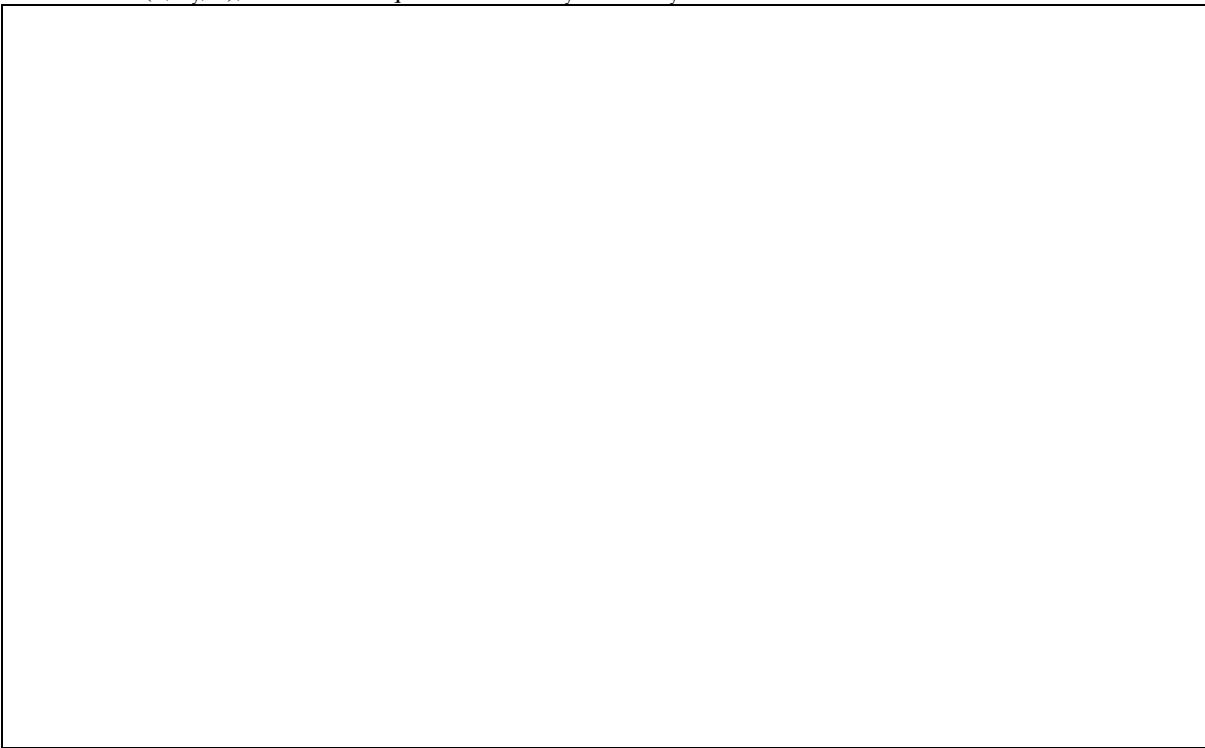


Figure 4.33. Moving between representations of a plane.

Planar Patches.

Just as we can restrict the parameter t in the representation of a line to obtain a ray or a segment, we can restrict the parameters s and t in the representation of a plane.

In the parametric form of Equation 4.43 the values for s and t can range from $-\infty$ to ∞ , and thus the plane can extend forever. In some situations we want to deal with only a “piece” of a plane, such as a parallelogram that lies in it. Such a piece is called a **planar patch**, a term that invites us to imagine the plane as a quilt of many patches joined together. Later we examine curved surfaces made up of patches which are not necessarily planar. Much of the practice of modeling solids involves piecing together patches of various shapes to form the skin of an object.

A planar patch is formed by restricting the range of allowable parameter values for s and t . For instance, one often restricts s and t to lie only between 0 and 1. The patch is positioned and oriented in space by appropriate choices of \mathbf{a} , \mathbf{b} , and C . Figure 4.34a shows the available range of s and t as a square in **parameter space**, and Figure 4.34b shows the patch that results from this restriction in object space.

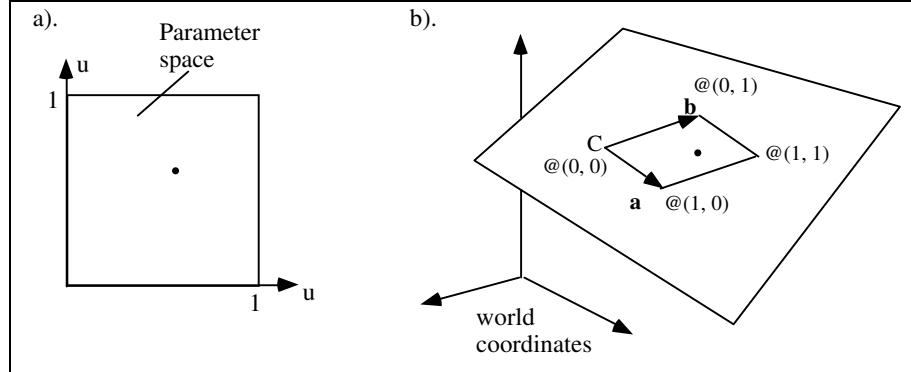


Figure 4.34. Mapping between two spaces to define a planar patch.

To each point (s, t) in parameter space there corresponds one 3D point in the patch $P(s, t) = C + \mathbf{as} + \mathbf{bt}$. The patch is a parallelogram whose corners correspond to the four corners of parameter space and are situated at

$$\begin{aligned} P(0, 0) &= C; \\ P(1, 0) &= C + \mathbf{a}; \\ P(0, 1) &= C + \mathbf{b}; \\ P(1, 1) &= C + \mathbf{a} + \mathbf{b}. \end{aligned} \quad (4.48)$$

The vectors \mathbf{a} and \mathbf{b} determine both the size and the orientation of the patch. If \mathbf{a} and \mathbf{b} are perpendicular, the grid will become rectangular, and if in addition \mathbf{a} and \mathbf{b} have the same length, the grid will become square. Changing C just shifts the patch without changing its shape or orientation.

Example 4.5.10. Make a patch. Let $C = (1, 3, 2)$, $\mathbf{a} = (1, 1, 0)$, and $\mathbf{b} = (1, 4, 2)$. Find the corners of the planar patch. **Solution:** From the preceding table we obtain the four corners: $P(0, 0) = (1, 3, 2)$, $P(0, 1) = (2, 7, 4)$, $P(1, 0) = (2, 4, 2)$, and $P(1, 1) = (3, 8, 4)$.

Example 4.5.11. Characterize a Patch. Find \mathbf{a} , \mathbf{b} , and C that create a square patch of length 4 on a side centered at the origin and parallel to the x , z -plane. **Solution:** The corners of the patch are at $(2, 0, 2)$, $(2, 0, -2)$, $(-2, 0, 2)$, and $(-2, 0, -2)$. Choose any corner, say $(2, 0, -2)$, for C . Then \mathbf{a} and \mathbf{b} each have length 4 and are parallel to either the x - or the z -axis. Choose $\mathbf{a} = (-4, 0, 0)$ and $\mathbf{b} = (0, 0, 4)$.

Practice Exercise 4.5.8. Find a Patch. Find point C and some vectors \mathbf{a} and \mathbf{b} that create a patch having the four corners $(-4, 2, 1)$, $(1, 7, 4)$, $(-2, -2, 2)$, and $(3, 3, 5)$.

4.6. Finding the Intersection of two Line Segments.

We often need to compute where two line segments in 2D space intersect. It appears in many other tasks, such as determining whether or not a polygon is simple. Its solution will illustrate the power of parametric forms and dot products.

The Problem: Given two line segments, determine whether they intersect, and if they do, find their point of intersection.

Suppose one segment has endpoints A and B and the other segment has endpoints C and D . As shown in Figure 4.35 the two segments can be situated in many different ways: They can miss each other (a and b), overlap in one point (c and d), or even overlap over some region (e). They may or may not be parallel. We need an organized approach that handles all of these possibilities.

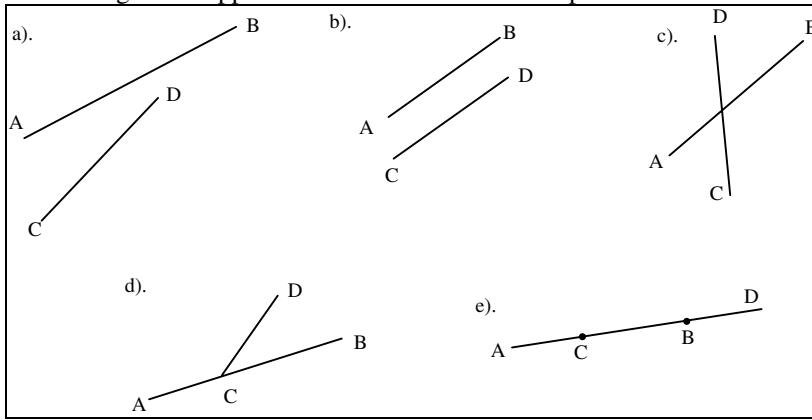


Figure 4.35. Many cases for two line segments.

Every line segment has a **parent line**, the infinite line of which it is part. Unless two parent lines are parallel they will intersect at some point. We first locate this point.

We set up parametric representations for each of the line segments in question. Call AB the segment from A to B . Then

$$AB(t) = A + \mathbf{b} t \quad (4.49)$$

where for convenience we define $\mathbf{b} = B - A$. As t varies from 0 to 1 all points on the finite line segment are visited. If t is allowed to vary from $-\infty$ to ∞ the entire parent line is swept out.

Similarly we call the segment from C to D by the name CD , and give it parametric representation (using a new parameter, say, u)

$$CD(u) = C + \mathbf{d} u,$$

where $\mathbf{d} = D - C$. We use different parameters for the two lines, t for one and u for the other, in order to describe different points on the two lines independently. (If the same parameter were used, the points on the two lines would be locked together.)

For the parent lines to intersect, there must be specific values of t and u for which the two equations above are equal:

$$A + \mathbf{b}t = C + \mathbf{d}u$$

Defining $\mathbf{c} = C - A$ for convenience we can write this condition in terms of three known vectors and two (unknown) parameter values:

$$\mathbf{b}t = \mathbf{c} + \mathbf{d}u \quad (4.50)$$

This provides two equations in two unknowns, similar to Equation 4.22. We solve it the same way: dot both sides with \mathbf{d}^\perp to eliminate the term in d , giving $\mathbf{d}^\perp \cdot \mathbf{b}t = \mathbf{d}^\perp \cdot \mathbf{c}$. There are two main cases: the term $\mathbf{d}^\perp \cdot \mathbf{b}$ is zero or it is not.

Case 1: The term $\mathbf{d}^\perp \cdot \mathbf{b}$ is not Zero.

Here we can solve for t obtaining:

$$t = \frac{\mathbf{d}^\perp \cdot \mathbf{c}}{\mathbf{d}^\perp \cdot \mathbf{b}} \quad (4.51)$$

Similarly “dot” both sides of Equation 4.50 with \mathbf{b}^\perp to obtain (after using one additional property of perp-dot products—which one?):

$$u = \frac{\mathbf{b}^\perp \cdot \mathbf{c}}{\mathbf{d}^\perp \cdot \mathbf{b}} \quad (4.52)$$

Now we know that the two parent lines intersect, and we know where. But this doesn’t mean that the line segments themselves intersect. If t lies outside the interval $[0, 1]$, segment AB doesn’t “reach” the other segment, with similar statements if u lies outside of $[0, 1]$. If both t and u lie between 0 and 1 the line segments *do* intersect at some point, I . The location of I is easily found by substituting the value of t in Equation 4.49:

$$I = A + \left(\frac{\mathbf{d}^\perp \cdot \mathbf{c}}{\mathbf{d}^\perp \cdot \mathbf{b}} \right) \mathbf{b} \quad (\text{the intersection point}) \quad (4.53)$$

Example 4.6.1: Given the endpoints $A = (0, 6)$, $B = (6, 1)$, $C = (1, 3)$, and $D = (5, 5)$, find the intersection if it exists. **Solution:** $\mathbf{d}^\perp \cdot \mathbf{b} = -32$, so $t = 7/16$ and $u = 13/32$ which both lie between 0 and 1, and so the segments do intersect. The intersection lies at $(x, y) = (21/8, 61/16)$. This result may be confirmed visually by drawing the segments on graph paper and measuring the observed intersection.

Case 2: The term $\mathbf{d}^\perp \cdot \mathbf{b}$ is Zero.

In this case we know \mathbf{d} and \mathbf{b} are parallel (why?). The segments might still overlap, but this can happen only if the parallel parent lines are identical. A test for this is developed in the exercises.

The exercises discuss developing a routine that performs the complete intersection test on two line segments.

Practice Exercises.

4.6.1. When the parent lines overlap. We explore case 2 above, where the term $\mathbf{d}^\perp \cdot \mathbf{b} = 0$, so the parent lines are parallel. We must determine whether the parent lines are identical, and if so whether the segments themselves overlap.

To test whether the parent lines are the same, see whether C lies on the parent line through A and B .

a). Show that the equation for this parent line is $b_x(y - A_y) - b_y(x - A_x) = 0$.

We then substitute C_x for x and C_y for y and see whether the left-hand side is sufficiently close to zero (i.e. its size is less than some tolerance such as 10^{-8}). If not, the parent lines do not coincide, and no intersection exists. If the parents lines are the same, the final test is to see whether the segments themselves overlap.

b). To do this, show how to find the two values t_c and t_d at which this line through A and B reaches C and D , respectively. Because the parent lines are identical, we can use just the x -component. Segment AB begins at 0 and ends at 1, and by examining the ordering of the four values 0, 1, t_c , and t_d , we can readily determine the relative positions of the two lines.

c). Show that there is an overlap unless both t_c and t_d are less than 0 or both are larger than 1. If there is an overlap, the endpoints of the overlap can easily be found from the values of t_c and t_d .

d). Given the endpoints $A = (0, 6)$, $B = (6, 2)$, $C = (3, 4)$, and $D = (9, 0)$, determine the nature of any intersection.

4.6.2. The Algorithm for determining the intersection. Write the routine `segIntersect()` that would be used in the context: `if(segIntersect(A, B, C, D, InterPt)) <do something>` It takes four points representing the two segments, and returns 0 if the segments do not intersect, and 1 if they do. If they do intersect the location of the intersection is placed in `interPt`. It returns -1 if the parent lines are identical.

4.6.3. Testing the Simplicity of a Polygon. Recall that a polygon P is simple if there are no edge intersections except at the endpoints of adjacent edges. Fashion a routine `int isSimple(Polygon P)` that takes a brute force approach and tests whether any pair of edges of the list of vertices of the polygon intersect, returning 0 if so, and 1 if not so. (`Polygon` is some suitable class for describing a polygon.) This is a simple algorithm but not the most efficient one. See [moret91] and [preparata85] for more elaborate attacks that involve some sorting of edges in x and y .

4.6.4. Line Segment Intersections. For each of the following segment pairs, determine whether the segments intersect, and if so where.

- | | | | |
|-------------------|------------------|--------------------|------------------|
| 1. $A = (1, 4)$, | $B = (7, 1/2)$, | $C = (7/2, 5/2)$, | $D = (7, 5)$; |
| 2. $A = (1, 4)$, | $B = (7, 1/2)$, | $C = (5, 0)$, | $D = (0, 7)$; |
| 3. $A = (0, 7)$, | $B = (7, 0)$, | $C = (8, -1)$, | $D = (10, -3)$; |

4.6.1. Application of Line Intersections: the circle through three points.

Suppose a designer wants a tool that draws the unique circle that passes through three given points. The user specifies three points A , B , and C , on the display with the mouse as suggested in Figure 4.36a, and the circle is drawn automatically as shown in Figure 4.36b. The unique circle that passes through three points is called the **excircle** or **circumscribed circle**, of the triangle defined by the points. Which circle is it? We need a routine that can calculate its center and radius.

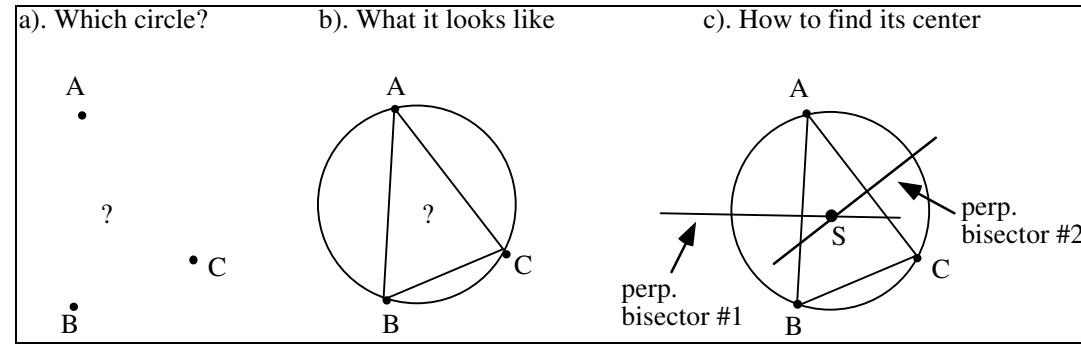


Figure 4.36. Finding the excircle.

Figure 4.35c shows how to find it. The center S of the desired circle must be equidistant from all three vertices, so it must lie on the **perpendicular bisector** of *each* side of triangle ABC (The perpendicular bisector is the locus of all points that are equidistant from two given points.). Thus we can determine S if we can compute where two of the perpendicular bisectors intersect.

We first show how to find a parametric representation of the perpendicular bisector of a line segment. Figure 4.37 shows a segment S with endpoints A and B . Its perpendicular bisector L is the infinite line that passes through the midpoint M of segment S , and is oriented perpendicular to it. But we know that midpoint M is given by $(A + B)/2$, and the direction of the normal is given by $(B - A)^\perp$, so the perpendicular bisector has parametric form:

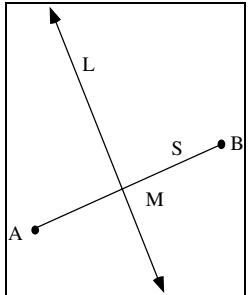


Figure 4.37. The perpendicular bisector of a segment.

$$L(t) = \frac{1}{2}(A + B) + (B - A)^{\perp}t \quad (\text{the perpendicular bisector of } AB) \quad (4.54)$$

Now we are in a position to compute the excircle of three points. Returning to Figure 4.35b we seek the intersection S of the perpendicular bisectors of AB and AC . For convenience we define the vectors:

$$\begin{aligned} \mathbf{a} &= B - A \\ \mathbf{b} &= C - B \\ \mathbf{c} &= A - C \end{aligned} \quad (4.55)$$

To find the perpendicular bisector of AB we need the midpoint of AB and a direction perpendicular to AB . The midpoint of AB is $A + \mathbf{a} / 2$ (why?). The direction perpendicular to AB is \mathbf{a}^{\perp} . So the parametric form for the perpendicular bisector is $A + \mathbf{a} / 2 + \mathbf{a}^{\perp}t$. Similarly the perpendicular bisector of AC is $A - \mathbf{c} / 2 + \mathbf{c}^{\perp}u$, using parameter u . Point S lies where these meet, at the solution of:

$$\mathbf{a}^{\perp}t = \mathbf{b} / 2 + \mathbf{c}^{\perp}u$$

(where we have used $\mathbf{a} + \mathbf{b} + \mathbf{c} = \mathbf{0}$). To eliminate the term in u take the dot product of both sides with \mathbf{c} , and obtain $t = 1/2 (\mathbf{b} \cdot \mathbf{c}) / (\mathbf{a}^{\perp} \cdot \mathbf{c})$. To find S use this value for t in the representation of the perpendicular bisector: $A + \mathbf{a} / 2 + \mathbf{a}^{\perp}t$, which yields the simple *explicit* form¹⁰:

$$S = A + \frac{1}{2} \left(\mathbf{a} + \frac{\mathbf{b} \cdot \mathbf{c}}{\mathbf{a}^{\perp} \cdot \mathbf{c}} \mathbf{a}^{\perp} \right) \quad (\text{center of the excircle}) \quad (4.56)$$

The radius of the excircle is the distance from S to any of the three vertices, so it is $|S - A|$. Just form the magnitude of the last term in Equation 4.56. After some manipulation (check this out) we obtain:

$$\text{radius} = \frac{|\mathbf{a}|}{2} \sqrt{\left(\frac{\mathbf{b} \cdot \mathbf{c}}{\mathbf{a}^{\perp} \cdot \mathbf{c}} \right)^2 + 1} \quad (\text{radius of the excircle}) \quad (4.57)$$

Once S and the radius are known, we can use `drawCircle()` from Chapter 3 to draw the desired circle.

Example 4.6.2. Find the perpendicular bisector L of the segment AB having endpoints $A = (3, 5)$ and $B = (9, 3)$.

Solution: By direct calculation, midpoint $M = (6, 4)$, and $(B - A)^{\perp} = (2, 6)$, so L has representation $L(t) = (6 + 2t, 4 + 6t)$. It is useful to plot both S and L to see this result.

¹⁰Other closed form expressions for S have appeared previously, e.g. in [goldman90] and [lopex92]

Every triangle also has an **inscribed circle**, which is sometimes necessary to compute in a computer-aided design context. A case study examines how to do this, and also discusses the beguiling **nine-point circle**.

Practice Exercise 4.6.5. A Perpendicular Bisector. Find a parametric expression for the perpendicular bisector of the segment with endpoints $A = (0, 6)$ and $B = (4, 0)$. Plot the segment and the line.

4.7. Intersections of Lines with Planes, and Clipping.

The task of finding the intersection of a line with another line or with a plane arises in a surprising variety of situations in graphics. We have already seen one approach in Section 4.6, that finds where two line segments intersect. That approach used parametric representations for both the line segments, and solved two simultaneous equations.

Here we develop an alternative method that works for both lines and planes. It represents the intersecting line by a parametric representation, and the line or plane being intersected in a point normal form. It is very direct and clearly reveals what is going on. We develop the method once, and then apply the results to the problem of clipping a line against a convex polygon in 2D, or a convex polyhedron in 3D. In Chapter 7 we see that this is an essential step in viewing 3D objects. In Chapter 14 we use the same intersection technique to get started in ray tracing.

In 2D we want to find where a line intersects another line; in 3D we want to find where a line intersects a plane. Both of these problems can be solved at once because the formulation is in terms of dot products, and the same expressions arise whether the involved vectors are 2D or 3D. (We also address the problem of finding the intersection of two planes in the exercises: it too is based on dot products.)

Consider a line described parametrically as $R(t) = A + \mathbf{c} t$. We also refer to it as a “ray”. We want to compute where it intersects the object characterized by the point normal form $\mathbf{n} \cdot (P - B) = 0$. In 2D this is a line; in 3D it is a plane. Point B lies on it, and vector \mathbf{n} is normal to it. Figure 4.38a shows the ray hitting a line, and part b) shows it hitting a plane. We want to find the location of the “hit point”.

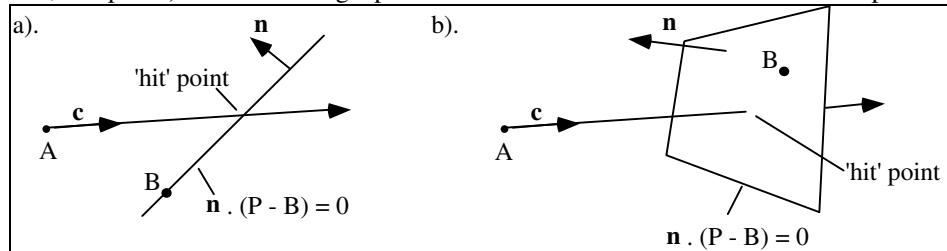


Figure 4.38. Where does a ray hit a line or a plane?

Suppose it hits at $t = t_{\text{hit}}$, the “hit time”. At this value of t the line and ray must have the same coordinates, so $A + \mathbf{c} t_{\text{hit}}$ must satisfy the equation of the point normal form of the line or plane. Therefore we substitute this unknown “hit point” into the point normal equation to obtain a condition on t_{hit} :

$$\mathbf{n} \cdot (A + \mathbf{c} t_{\text{hit}} - B) = 0.$$

This may be rewritten as

$$\mathbf{n} \cdot (A - B) + \mathbf{n} \cdot \mathbf{c} t_{\text{hit}} = 0,$$

which is a linear equation in t_{hit} . Its solution is:

$$t_{\text{hit}} = \frac{\mathbf{n} \cdot (B - A)}{\mathbf{n} \cdot \mathbf{c}} \quad (\text{hit time — 2D and 3D cases}) \quad (4.58)$$

As always with a ratio of terms we must examine the eventuality that the denominator of t_{hit} is zero. This occurs when $\mathbf{n} \cdot \mathbf{c} = 0$, or when the ray is aimed parallel to the plane, in which case there is no hit at all.¹¹

When the hit time has been computed, it is simple to find the location of the hit point : Substitute t_{hit} into the representation of the ray:

$$\text{"hit" point: } P_{\text{hit}} = A + \mathbf{c}t_{\text{hit}} \quad (\text{hit spot — 2D and 3D cases}) \quad (4.59)$$

In the intersection problems treated below we will also need to know generally which direction the ray strikes the line or plane: "along with" the normal \mathbf{n} or "counter to" \mathbf{n} . (This will be important because we will need to know whether the ray is exiting from an object or entering it.) Figure 4.39 shows the two possibilities for a ray hitting a line. In part a) the angle between the ray's direction, \mathbf{c} , and \mathbf{n} is less than 90° so we say the ray is aimed "along with" \mathbf{n} . In part b) the angle is greater than 90° so the ray is aimed "counter to" \mathbf{n} .

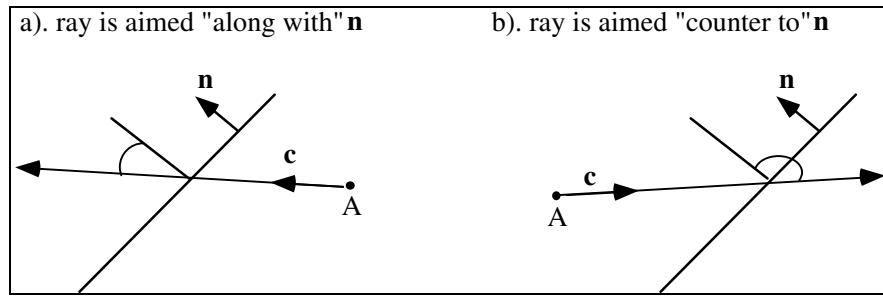


Figure 4.39. The direction of the ray is "along" or "against" \mathbf{n} .

It is easy to test which of these possibilities occurs, since the *sign* of $\mathbf{n} \cdot \mathbf{c}$ tells immediately whether the angle between \mathbf{n} and \mathbf{c} is less than or greater than 90° . Putting these ideas together, we have the three possibilities:

- if $\mathbf{n} \cdot \mathbf{c} > 0$ the ray is aimed "along with" the normal;
- if $\mathbf{n} \cdot \mathbf{c} = 0$ the ray is parallel to the line
- if $\mathbf{n} \cdot \mathbf{c} < 0$ the ray is aimed "counter to" the normal

Practice Exercises.

4.7.1. Intersections of rays with lines and planes. Find when and where the ray $A + \mathbf{c}t$ hits the object $\mathbf{n} \cdot (P - B) = 0$ (lines in the 2D or planes in the 3D).

- $A = (2, 3)$, $\mathbf{c} = (4, -4)$, $\mathbf{n} = (6, 8)$, $B = (7, 7)$.
- $A = (2, -4, 3)$, $\mathbf{c} = (4, 0, -4)$, $\mathbf{n} = (6, 9, 9)$, $B = (-7, 2, 7)$.
- $A = (2, 0)$, $\mathbf{c} = (0, -4)$, $\mathbf{n} = (0, 8)$, $B = (7, 0)$.
- $A = (2, 4, 3)$, $\mathbf{c} = (4, 4, -4)$, $\mathbf{n} = (6, 4, 8)$, $B = (7, 4, 7)$.

4.7.2. Rays hitting Planes. Find the point where the ray $(1, 5, 2) + (5, -2, 6)t$ hits the plane $2x - 4y + z = 8$.

4.7.3. What is the intersection of two planes? Geometrically we know that two planes intersect in a straight line. But which line? Suppose the two planes are given by $\mathbf{n} \cdot (P - A) = 0$ and $\mathbf{m} \cdot (P - B) = 0$. Find the parametric form of the line in which they intersect. You may find it easiest to:

- First obtain a parametric form for one of the planes: say, $C + \mathbf{a}s + \mathbf{b}t$ for the second plane.
- Then substitute this form into the point normal form for the first plane, thereby obtaining a linear equation that relates parameters s and t .
- Solve for s in terms of t , say $s = E + Ft$. (Find expressions for E and F .)
- Write the desired line as $C + \mathbf{a}(E + Ft) + \mathbf{b}t$.

4.8. Polygon Intersection Problems.

¹¹If the numerator is also 0 the ray lies entirely in the line (2D) or plane (3D). (why?).

We know polygons are the fundamental objects used in both 2D and 3D graphics. In 2D graphics their straight edges make it easy to describe them and draw them. In 3D graphics, an object is often modeled as a polygonal “mesh”: a collection of polygons that fit together to make up its “skin”. If the skin forms a closed surface that encloses some space the mesh is called a polyhedron. We study meshes and polyhedra in depth in Chapter 6.

Figure 4.40 shows a 2D polygon and a 3D polyhedron that we might need to analyze or render in a graphics application. Three important questions that arise are:



Figure 4.40. Intersection problems of a line and a polygonal object.

- a). Is a given point P inside or outside the object?
- b). Where does a given ray R first intersect the object?
- c). Which part of a given line L lies inside the object, and which part lies outside?

As a simple example, which part(s) of the line $3y - 2x = 6$ lie inside the polygon whose vertices are $(0, 3)$, $(-2, -2)$, $(-5, 0)$, $(0, -7)$, $(1, 1)$?

4.8.1. Working with convex polygons and polyhedra.

The general case of intersecting a line with any polygon or polyhedron is quite complex; we address it in Section 4.8.4. Things are much simpler when the polygon or polyhedron is convex. They are simpler because a convex polygon is completely described by a set of “bounding lines”; in 3D a convex polyhedron is completely described by a set of “bounding planes”. So we need only test the line against a set of unbounded lines or planes.

Figure 4.41 illustrates this for the 2D case. Part a) shows a convex pentagon, and part b) shows the bounding lines L_0 , L_1 , etc. of the pentagon. Each bounding line defines two half spaces: the inside half space that contains the polygon, and the outside half space that shares no points with the polygon. Part c) of the figure shows a portion of the outside half space associated with the bounding line L_2 .

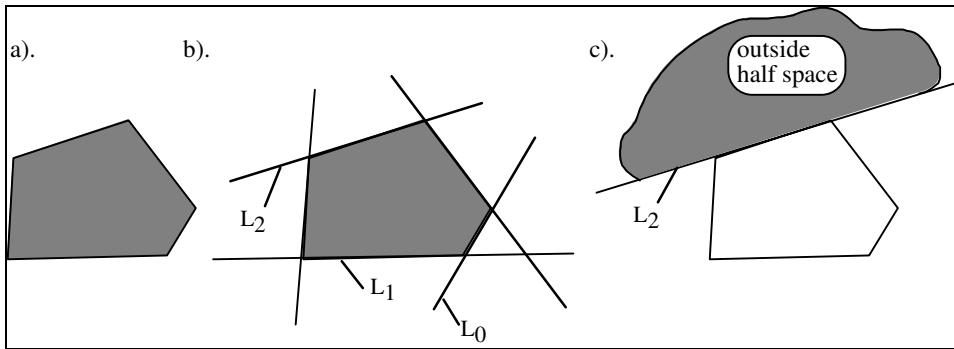


Figure 4.41. Convex polygons and polyhedra.

Example 4.8.1. Finding the bounding lines. Figure 4.42a shows a unit square. There are four bounding lines, given by $x = 1$, $x = -1$, $y = 1$, and $y = -1$. In addition, for each bounding line we can identify the outward normal vector: the one that points into the outside half space of the bounding line. The outward normal vector for the line $y = 1$ is of course $\mathbf{n} = (0, 1)$. (What are the other three?)

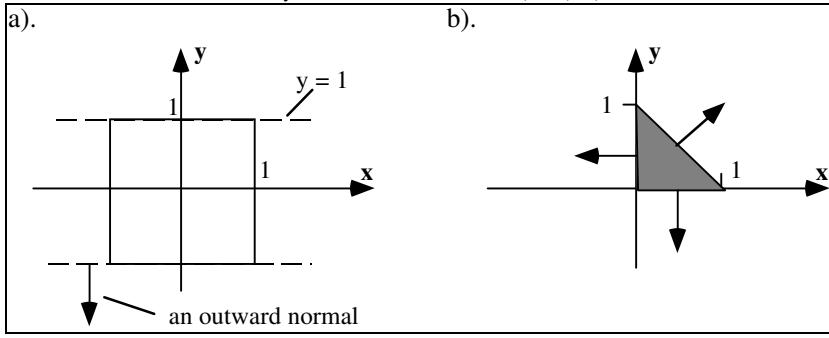


Figure 4.42. Examples of convex polygons.

The triangle in part b) has three bounding lines. (What is the equation for each line?) The point normal form for each of the three lines is given next; in each case it uses the outward normal (check this):

$$\begin{aligned} (-1, 0) \cdot (P - (0, 0)) &= 0; \\ (0, -1) \cdot (P - (0, 0)) &= 0; \\ (1, 1) \cdot (P - (1, 0)) &= 0; \end{aligned}$$

The big advantage in dealing with convex polygons is that we perform intersection tests only on infinite lines, and don't need to check whether an intersection lies "beyond" an endpoint — recall the complexity of the intersection tests in Section 4.7. In addition the point normal form can be used, which simplifies the calculations.

For a convex polyhedron in 3D, each plane has an inside and an outside half space, and an outward pointing normal vector. The polyhedron is the intersection of all the inside half spaces, (the set of all points that are simultaneously in the inside half space of every bounding plane).

4.8.2. Ray Intersections and Clipping for Convex Polygons.

We developed a method in Section 4.7 that finds where a ray hits an individual line or plane. We can use this method to find where a ray hits a convex polygon or polyhedron.

The Intersection Problem. Where does the ray $A + ct$ hit polygon P ?

Figure 4.43 shows a ray $A + \mathbf{c}t$ intersecting polygon P . We want to know all of the places where the ray hits P . Because P is convex the ray hits P exactly twice: It enters once and exits once. Call the values of t at which it enters and exits t_{in} and t_{out} , respectively. The ray intersection problem is to compute the values of t_{in} and t_{out} . Once these hit times are known we of course know the hit points themselves:

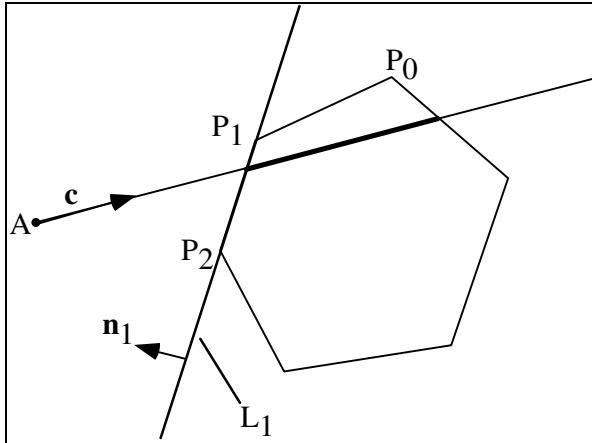


Figure 4.43. Ray $A + \mathbf{c}t$ intersecting a convex polygon.

$$\begin{aligned} \text{Entering hit point: } & A + \mathbf{c} t_{\text{in}} \\ \text{Exiting hit point: } & A + \mathbf{c} t_{\text{out}} \end{aligned} \quad (4.61)$$

The ray is inside P for all t in the interval $[t_{\text{in}}, t_{\text{out}}]$.

Note that finding t_{in} and t_{out} not only solves the intersection problem, but also the clipping problem. If we know t_{in} and t_{out} we know which part of the line $A + \mathbf{c}t$ lies inside P . Usually the clipping problem is stated as:

The Clipping problem: For the two points A and C which part of segment AC lies inside P ?

Figure 4.44 shows several possible situations. Part a) shows the case where A and C both lie outside P , but there is a portion of the segment AC that lies inside P .

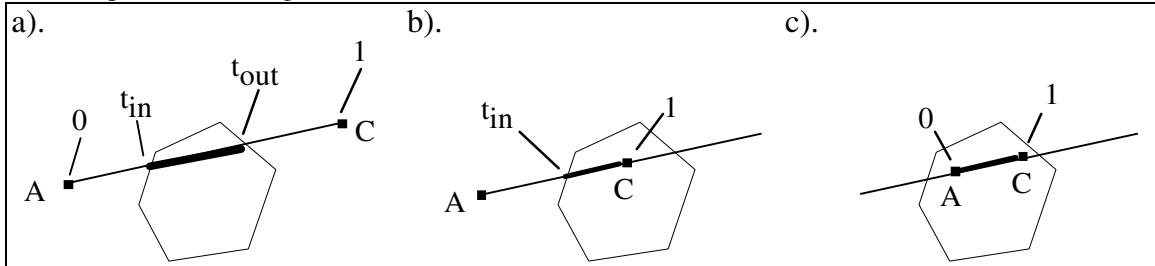


Figure 4.44. A segment clipped by a polygon.

If we consider segment AC as part of a ray given by $A + \mathbf{c}t$ where $\mathbf{c} = C - A$, then point A corresponds to the point on the ray at $t = 0$, and C corresponds to the point at $t = 1$. These “ray times” are labeled in the figure. To find the clipped segment we compute t_{in} and t_{out} as described above. The segment that “survives” clipping has end points $A + \mathbf{c} t_{\text{in}}$ and $A + \mathbf{c} t_{\text{out}}$. In Figure 4.44b point C lies inside P and so t_{out} is larger than 1. The clipped segment has end points $A + \mathbf{c} t_{\text{in}}$ and C . In part c) both A and C lie inside P , so the clipped segment is the same: AC .

In general we compute t_{in} , and compare it to 0. The larger of the values 0 and t_{in} is used as the “time” for the first end point of the clipped segment. Similarly, the smaller of the values 1 and t_{out} is used to find the second end point. So the end points of the clipped segment are:

$$\begin{aligned} A' &= A + \mathbf{c} \max(0, t_{\text{in}}) \\ C' &= A + \mathbf{c} \min(t_{\text{out}}, 1) \end{aligned} \quad (4.62)$$

Now how are t_{in} and t_{out} computed? We must consider each of the bounding lines of P in turn, and find where the ray $A + \mathbf{c}t$ intersects it. We suppose each bounding line is stored in point normal form as the pair

$\{B, \mathbf{n}\}$, where B is some point on the line and \mathbf{n} is the *outward pointing normal* for the line: it points to the outside of the polygon. Because it is outward pointing the test of Equation 4.60 translates to:

- | | |
|--------------------------------------|---------------------------------|
| if $\mathbf{n} \cdot \mathbf{c} > 0$ | the ray is exiting from P ; |
| if $\mathbf{n} \cdot \mathbf{c} = 0$ | the ray is parallel to the line |
| if $\mathbf{n} \cdot \mathbf{c} < 0$ | the ray is entering P |
- (4.63)

For each bounding line we find:

- The hit time of the ray with the bounding line (use Equation 4.58);
- Whether the ray is entering or exiting the polygon (use Equation 4.63)

If the ray is entering, we know that the time at which the ray ultimately enters P (if it enters it at all) cannot be *earlier* than this newly found hit time. We keep track of the “earliest possible entering time” as t_{in} . For each entering hit time, t_{hit} , we replace t_{in} by $\max(t_{in}, t_{hit})$. Similarly we keep track of the *latest* possible exit time as t_{out} , and for each exiting hit we replace t_{out} by $\min(t_{out}, t_{hit})$.

It helps to think of the interval $[t_{in}, t_{out}]$ as the **candidate interval** of t , the interval of t inside of which the ray *might* lie inside the object. Figure 4.45 shows an example for the clipping problem. We know the point $A + \mathbf{c}t$ *cannot* be inside P for any t in the candidate interval. As each bounding line is tested, the candidate interval gets reduced as t_{in} is increased or t_{out} is decreased: pieces of it get “chopped” off. To get started we initialize t_{in} to 0 and t_{out} to 1 for the line clipping problem, so the candidate interval is $[0,1]$.

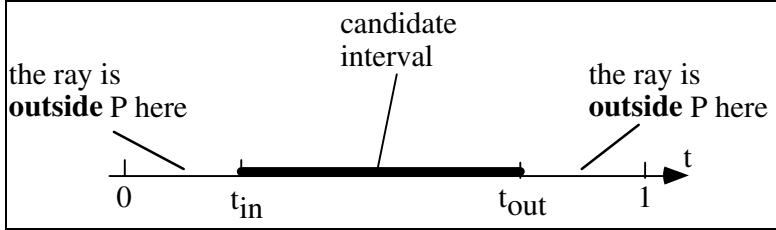


Figure 4.45. The candidate interval for a hit.

The algorithm is then:

- 1). Initialize the candidate interval to $[0,1]$ ¹².
- 2). For each bounding line, use Equation 4.58 to find the hit time t_{hit} and determine whether it's an entering or exiting hit:
 - if it's an entering hit, set $t_{in} = \max(t_{in}, t_{hit})$
 - if it's an exiting hit, set $t_{out} = \min(t_{out}, t_{hit})$

If at any point t_{in} becomes greater than t_{out} we know the ray misses P entirely, and testing is terminated
 3). If candidate interval is not empty, then from Equation 4.62 the segment from $A + \mathbf{c}t_{in}$ to $A + \mathbf{c}t_{out}$ is known to lie inside P . For the line clipping problem these are the endpoints of the clipped line. For the ray intersection problem we know the entering and exiting points of the ray.

Note that we stop further testing as soon the candidate interval vanishes. This is called an **early out**: if we determine early in the processing that the ray is outside of the polygon, we save time by immediately exiting from the test.

Figure 4.46 shows a specific example of clipping: we seek the portion of segment AC that lies in polygon P . We initialize t_{in} to 0 and t_{out} to 1. The ray “starts” at A at $t = 0$ and proceeds to point C , reaching it at $t = 1$. We test it against each bounding line L_0, L_1, \dots , in turn and update t_{in} and t_{out} as necessary.

¹² For the ray intersection problem, where the ray extends infinitely far in both directions, we set $t_{in} = -\infty$ and $t_{out} = \infty$. In practice t_{in} is set to a large negative value, and t_{out} to a large positive value.

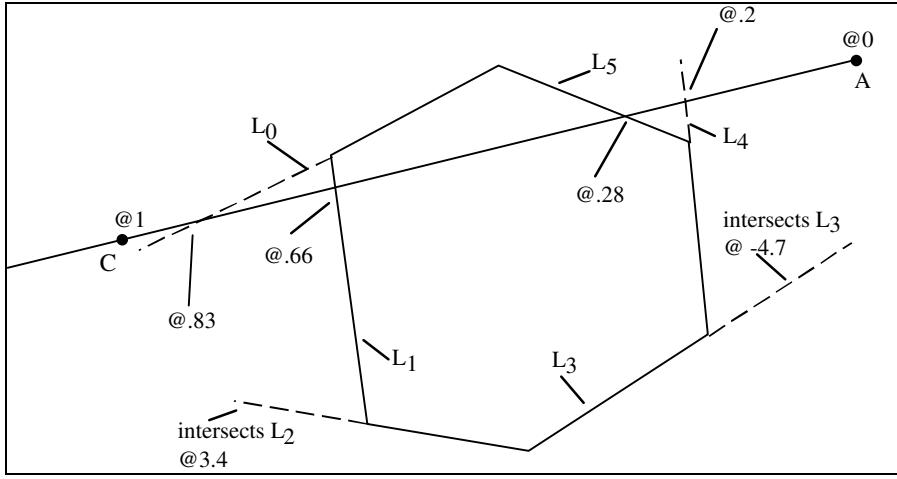


Figure 4.46. Testing when a ray lies inside a convex polygon.

Suppose when we test it against line L_0 we find an exiting hit at $t = 0.83$. This sets t_{out} to 0.83, and the candidate interval is now $[0, 0.83]$. We then test it against L_1 and find an exiting hit at $t = 0.66$. This reduces the candidate interval to $[0, 0.66]$. The test against L_2 gives an exiting hit at $t = 3.4$. This tells us nothing new: we already know the ray is outside for $t > 0.66$. The test against L_3 gives an entering hit at $t = -0.47$. So we set t_{in} to -0.47, and the candidate interval is $[-0.47, 0.66]$. The test with L_4 gives an entering hit at $t = 0.2$, so t_{in} is updated to 0.2. Finally, testing against L_5 gives an entering hit at $t = 0.28$, and we are done. The candidate interval is $[0.28, 0.66]$. In fact the ray *is* inside P for all t between 0.28 and 0.66.

Figure 4.47 shows the sequence of updates to t_{in} and t_{out} that occur as each of the lines above is tested.

line test	t_{in}	t_{out}
0	0	0.83
1	0	0.66
2	0	0.66
3	0	0.66
4	0.2	0.66
5	0.28	0.66

Figure 4.47. Updates on the values of t_{in} and t_{out} .

4.8.3. The Cyrus-Beck Clipping Algorithm.

We build a routine from these ideas, that performs the clipping of a line segment against any convex polygon. The method was originally developed by Cyrus and Beck [cyrus78]. Later a highly efficient clipper for rectangular windows was devised by Liang and Barsky [liang84] based on similar ideas. It is discussed in a Case Study at the end of this chapter.

The routine that implements the Cyrus-Beck clipper has interface:

```
int CyrusBeckClip(Line& seg, LineList& L);
```

Its parameters are the line segment, `seg`, to be clipped (which contains the first and second endpoints named `seg.first` and `seg.second`) and the list of bounding lines of the polygon. It clips `seg` against each line in `L` as described above, and places the clipped segment back in `seg`. (This is why `seg` must be passed by reference.) The routine returns:

- 0 if no part of the segment lies in P (the candidate interval became empty);
- 1 if some part of the segment does lie in P .

Figure 4.48 shows pseudocode for the Cyrus Beck algorithm. The types `LineSegment`, `LineList`, and `Vector2` are suitable data types to hold the quantities in question (see the exercises). Variables `numer` and `denom` hold the numerator and denominator for t_{hit} of Equation 4.48:

$$\begin{aligned} numer &= \mathbf{n} \cdot (\mathbf{B} - \mathbf{A}) \\ denom &= \mathbf{n} \cdot \mathbf{c} \end{aligned} \tag{4.64}$$

```

int CyrusBeckClip(LineSegment& seg, LineList L)
{
    double numer, denom; // used to find hit time for each line
    double tIn = 0.0, tOut = 1.0;
    Vector2 c, tmp;
    form vector: c = seg.second - seg.first
    for(int i = 0; i < L.num; i++) // chop at each bounding line
    {
        form vector tmp = L.line[i].pt - first
        numer = dot(L.line[i].norm, tmp);
        denom = dot(L.line[i].norm, c);
        if(!chopCI(numer, denom, tIn, tOut)) return 0; // early out
    }
    // adjust the endpoints of the segment; do second one 1st.
    if (tOut < 1.0) // second endpoint was altered
    {
        seg.second.x = seg.first.x + c.x * tOut;
        seg.second.y = seg.first.y + c.y * tOut;
    }
    if (tIn > 0.0) // first endpoint was altered
    {
        seg.first.x = seg.first.x + c.x * tIn;
        seg.first.y = seg.first.y + c.y * tIn;
    }
    return 1; // some segment survives
}

```

Figure 4.48. Cyrus-Beck Clipper for a Convex Polygon, 2D case (pseudocode).

Note that the value of `seg.second` is updated first, since we must use the old value of `seg.first` in the update calculation for both `seg.first` and `seg.second`.

The routine `chopCI()` is shown in Figure 4.49. It uses `numer` and `denom` of Equation 4.64 to calculate the hit time at which the ray hits a bounding line, uses Equation 4.63 to determine whether the ray is entering or exiting the polygon, and “chops” off the piece of the candidate interval CI that is thereby found to be outside the polygon.

```

int chopCI(double& tIn, double& tOut, double numer, double
denom)
{
    double tHit;
    if (denom < 0) // ray is entering
    {
        tHit = numer / denom;
        if (tHit > tOut) return 0; // early out
        else if (tHit > tIn) tIn = tHit; // take larger t
    }
    else if(denom > 0) // ray is exiting
    {
        tHit = numer / denom;
        if(tHit < tIn) return 0; // early out
        if(tHit < tout) tOut = tHit; // take smaller t
    }
    else // denom is 0: ray is parallel
    if(numer <= 0) return 0; // missed the line
    return 1; // CI is still non-empty
}

```

Figure 4.49. Clipping against a single bounding line.

If the ray is parallel to the line it could lie entirely in the inside half space of the line, or entirely out of it. It turns out that $\text{numer} = \mathbf{n} \cdot (\mathbf{B} - \mathbf{A})$ is exactly the quantity needed to tell which of these cases occurs. See the exercises.

The 3D case: Clipping a line against a Convex Polyhedron.

The Cyrus Beck clipping algorithm works in three dimensions in exactly the same way. In 3D the edges of the window become planes defining a convex region in three dimensions, and the line segment is a line suspended in space. `ChopCI()` needs no changes at all (since it uses only the values of dot products - through `numer` and `denom`). The data types in `CyrusBeckClip()` must of course be extended to 3D types, and when the endpoints of the line are adjusted the z-component must be adjusted as well.

Practice Exercises.

4.8.2. Data types for variable in the Cyrus Beck Clipper. Provide useful definitions for data types, either as struct's or classes, for `LineSegment`, `LineList`, and `Vector2` used in the Cyrus Beck clipping algorithm.

4.8.3. What does $\text{numer} \leq 0$ do?

Sketch the vectors involved in value of `numer` in `chopCI()` and show that when the ray $\mathbf{A} + \mathbf{c}t$ moves parallel to the bounding line $\mathbf{n} \cdot (\mathbf{P} - \mathbf{B}) = 0$, it lies wholly in the inside half space of the line if and only if $\text{numer} > 0$.

4.8.4. Find the Clipped Line. Find the portion of the segment with endpoints (2, 4) and (20, 8) that lies within the quadrilateral window with corners at (0, 7), (9, 9), (14, 4), and (2, 2).

4.8.4. Clipping against arbitrary polygons.

We saw how to clip a line segment against a convex polygon in the previous section. We generalize this to a method for clipping a segment against *any* polygon.

The basic problem is to find where the ray $\mathbf{A} + \mathbf{c}t$ lies inside polygon P given by the vertex list P_0, P_1, \dots, P_N . Figure 4.50 shows an example.

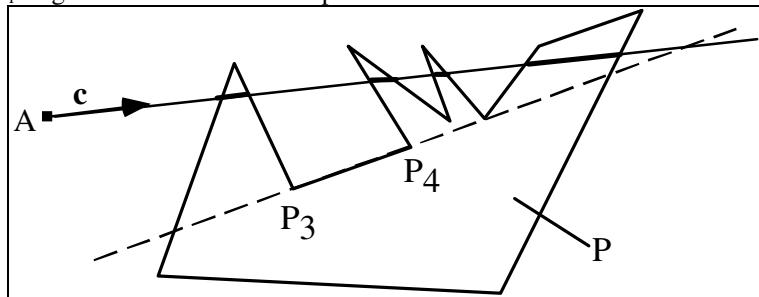


Figure 4.50. Where is a ray inside an arbitrary polygon P ?

It is clear that the ray can enter and exit from P multiple times in general, and that the result of clipping a segment against P may result in a *list* of segments rather than a single one. Also, of course, P is no longer described by a collection of infinite bounding lines in point normal form; we must work with the N finite segments such as $P_3 P_4$ that form its edges.

The problem is close to the problem we dealt with in Section 4.7: finding the intersection of two line segments. Now we are intersecting one line segment with the sequence of line segments associated with P .

We represent each edge of P parametrically (rather than in point normal form). For instance, the edge $P_3 P_4$ is represented as $P_3 + \mathbf{e}_3 u$ where $\mathbf{e}_3 = P_4 - P_3$ is the **edge vector** associated with P_3 . In general, the i -th edge is given by $P_i + \mathbf{e}_i u$, for u in $[0,1]$ and $i = 0, 1, \dots, N-1$ where $\mathbf{e}_i = P_{i+1} - P_i$, and as always we equate P_N with P_0 .

Recall from Section 4.7 that the ray $\mathbf{A} + \mathbf{c}t$ hits the i -th edge when t and u have the proper values to make $\mathbf{A} + \mathbf{c}t = P_i + \mathbf{e}_i u$. Calling vector $\mathbf{b}_i = P_i - \mathbf{A}$ we seek the solution (values of t and u) of

$$\mathbf{c}t = \mathbf{b}_i + \mathbf{e}_i u$$

Equations 4.51 and 4.52 hold the answers. When converted to the current notation we have:

$$t = \frac{\mathbf{e}_i^\perp \cdot \mathbf{b}_i}{\mathbf{e}_i^\perp \cdot \mathbf{c}} \quad \text{and} \quad u = \frac{\mathbf{c}^\perp \cdot \mathbf{b}_i}{\mathbf{e}_i^\perp \cdot \mathbf{c}}.$$

If $\mathbf{e}_i^\perp \cdot \mathbf{c}$ is 0 the i -th edge is parallel to the ray direction \mathbf{c} and there is no intersection. There is a true intersection with the i -th edge only if u falls in the interval $[0,1]$.

We need to find all of the legitimate hits of the ray with edges of P , and place them in a list of the hit times. Call this list `hitList`. Then pseudocode for the process would look like:

```
initialize hitList to empty
for(int i = 0; i < N; i++)      // for each edge of P
{
    build bi, ei for the i-th edge
    solve for t, u
    if(u lies in [0,1])
        add t to the hitList
}
```

What we do now with this list depends on the problem at hand.

The ray intersection problem. (Where does the ray *first* hit P ?)

This is solved by finding the smallest value of t , t_{\min} , in `theList`. The hit spot is, as always, $A + \mathbf{c} t_{\min}$.

The line clipping problem.

For this we need the sequence of t -intervals in which the ray is inside P . This requires sorting `theList` and then taking the t -values in pairs. The ray enters P at the first time in each pair, and exits from P at the second time of each pair.

Example 4.9.2. Clip AB to polygon P. Suppose the line to be clipped is AB as shown in Figure 4.51, for which $A = (1, 1)$ and $B = (8, 2)$.

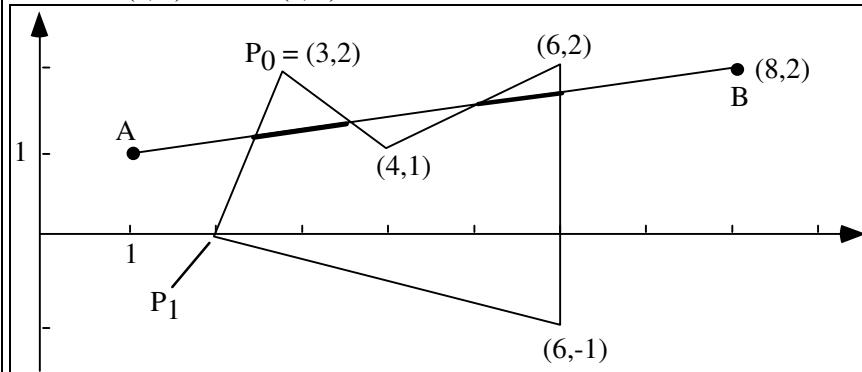


Figure 4.51. Clipping a line against a polygon.

P is given by the vertex list: $(3, 2), (2, 0), (6, -1), (6, 2), (4, 1)$. Taking each edge in turn we get for the values of t and u at the intersections:

edge	u	t
0	0.3846	0.2308
1	-0.727	-0.2727
2	0.9048	0.7142
3	0.4	0.6

4	0.375	0.375
---	-------	-------

The hit with edge 1 occurs at t outside of $[0,1]$ so it is discarded. We sort the remaining t -values and arrive at the sorted hit list: $\{0.2308, 0.375, 0.6, 0.7142\}$. Thus the ray enters P at $t = 0.2308$, exits it at $t = 0.375$, re-enters it at $t = 0.6$, and exits it for the last time at $t = 0.7142$.

Practice exercise 4.9.4. Clip a line. Find the portions of the line from $A = (1, 3.5)$ to $B = (9, 3.5)$ that lie inside the polygon with vertex list: $(2, 4), (3, 1), (4, 4), (3, 3)$.

4.8.5. More Advanced Clipping.

Clipping algorithms are fundamental to computer graphics, and a number of efficient algorithms have been developed. We have examined two approaches to clipping so far. The **Cohen Sutherland** clipping algorithm, studied in Chapter 2, clips a line against an aligned rectangle. The **Cyrus-Beck** clipper generalizes this to clipping a line against any convex polygon or polyhedron. But situations arise where one needs more sophisticated clipping. We mention two such methods here, and develop details of both in Case Studies at the end of this chapter.

The **Sutherland-Hodgman** clipper is similar to the Cyrus-Beck method, performing clipping against a convex polygon. But instead of clipping a single line segment, it clips an entire polygon (which needn't be convex) against the convex polygon. Most importantly, its output is again a *polygon* (or possibly a set of polygons). It can be important to retain the polygon structure during clipping since the clipped polygons may need to be filled with a pattern or color. This is not possible if the edges of the polygon are clipped individually.

The **Weiler-Atherton** clipping algorithm clips any polygon, P , against *any* other polygon, W , convex or not. It can output the part of P that lies inside W (**interior clipping**) or the part of P that lies outside W (**exterior clipping**). In addition, both P and W can have “holes” in them. As might be expected, this algorithm is somewhat more complex than the others we have examined, but its power makes it a welcome addition to one's toolbox in a variety of applications.

4.9. Summary of the Chapter.

Vectors provide a convenient way to express many geometric relations, and the operations they support provide a powerful way to manipulate geometric objects algebraically. Many computer graphics algorithms are simplified and made more efficient through the use of vectors. Because most vector operations are expressed the same way independent of the dimensionality of the underlying space, it is possible to derive results that are equally true in 2D or 3D space.

The dot product of two vectors is a fundamental quantity that simplifies finding the length of a vector and the angle between two vectors. It can be used to find such things as the orthogonal projection of one vector onto another, the location of the center of the excircle of three points, and the direction of a reflected ray. It is often used to test whether two vectors are orthogonal to one another, and more generally to test when they are pointing less than, or more than, 90° from each other. It is also useful to work with a 2D vector \mathbf{a}^\perp that lies 90° to the left of a given vector \mathbf{a} . In particular the dot product $\mathbf{a}^\perp \cdot \mathbf{b}$ reports useful information about how \mathbf{a} and \mathbf{b} are disposed relative to each other.

The cross product also reveals information about the angle between two vectors in 3D, and in addition evaluates to a vector that is perpendicular to them both. It is often used to find a vector that is normal to a plane.

In the process of developing an algorithm it is crucial to have a concise representation of the graphical objects involved. The two principal forms are the parametric representation, and the implicit form. The parametric representation “visits” each of the points on the object as a parameter is made to vary, so the parameter “indexes into” different points on the object. The implicit form expresses an equation that all points on the object, and only those, must satisfy. It is often given in the form $f(x, y) = 0$ in 2D, or $f(x, y, z) = 0$ in 3D, where $f()$ is some function. The value of $f()$ for a given point not only tells when the point is on

the object, but when a point lies off of the object the sign of $f()$ can reveal on *which* side of the object the point lies. In this chapter we addressed finding representations of the two fundamental “flat” objects in graphics: lines and planes. For such objects both the parametric form and implicit form are linear in their arguments. The implicit form can be revealingly written as the dot product of a normal vector and a vector lying within the object.

It is possible to form arbitrary linear combinations of vectors, but not of points. For points only affine combinations are allowed, or else chaos reigns if the underlying coordinate system is ever altered, as it frequently is in graphics. Affine combinations of points are useful in graphics, and we showed that they form the basis of “tweening” for animations and for Bezier curves.

The parametric form of a line or ray is particularly useful for such tasks as finding where two lines intersect or where a ray hits a polygon or polyhedron. These problems are important in themselves, and they also underlie clipping algorithms that are so prominent in graphics. The Cyrus-Beck clipper, which finds where a line expressed parametrically shares the same point in space as a line or plane expressed implicitly, addresses a larger class of problems than the Cohen Sutherland clipper of Chapter 2, and will be seen in action in several contexts later.

In the Case Studies that are presented next, the vector tools developed so far are applied to some interesting graphics situations, and their power is seen even more clearly. Whether or not you intend to carry out the required programming to implement these mini-projects, it is valuable to read through them and imagine what process you would pursue to solve them.

4.10. Case Studies.

4.10.1. Case Study 4.1: Animation with Tweening.

(Level of Effort: II.) Devise two interesting polylines, such as A and B as shown in Figure 4.52. Ensure that A and B have the same number of points, perhaps by adding an artificial extra point in the top segment of B.

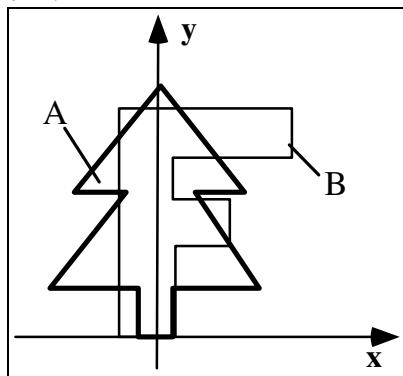


Figure 4.52. Tweening two polylines.

- Develop a routine similar to routine `drawTween(A, B, n, t)` of Figure 4.23 that draws the tween at t of the polylines A and B.
- Develop a routine that draws a sequence of “tweens” between A and B as t varies from 0 to 1, and experiment with it. Use the double buffering offered by OpenGL to make the animation smooth.
- Extend the routine so that after t increases gradually from 0 to 1 it decreases gradually back to 0 and then repeats, so the animation repeatedly shows A mutating into B then back into A. This should continue until a key is pressed.

d). Arrange so that the user can enter two polylines with the mouse, following which the polylines are tweened as just described. The user presses key ‘A’ and begins to lay down points to form polyline A, then presses key ‘B’ and lays down the points for polyline B. Pressing ‘T’ terminates that process and begins the tweening, which continues until the user types ‘Q’. Allow for the case where the user inputs a different number of points for A than for B: your program automatically creates the required number of extra points along line segments (perhaps at their midpoints) of the polyline having fewer points.

4.10.2. Case Study 4.2. Circles Galore.

(Level of Effort: II.). Write an application that allows the user to input the points of a triangle with a mouse. The program then draws the triangle along with its **inscribed circle**, **excircle**, and **9-point circle**, each in a different color. Arrange matters so the user can then move vertices of the triangle to new locations with the mouse, whereupon the new triangle with its three circles are redrawn.

We saw how to draw the excircle in Section 4.6.1. Here we show how to find the inscribed circle and the nine-point circle.

The inscribed circle. This is the circle that just snugs up inside the given triangle, and is tangent to all three sides¹³. Figure 4.53a shows a triangle ABC along with its inscribed circle.

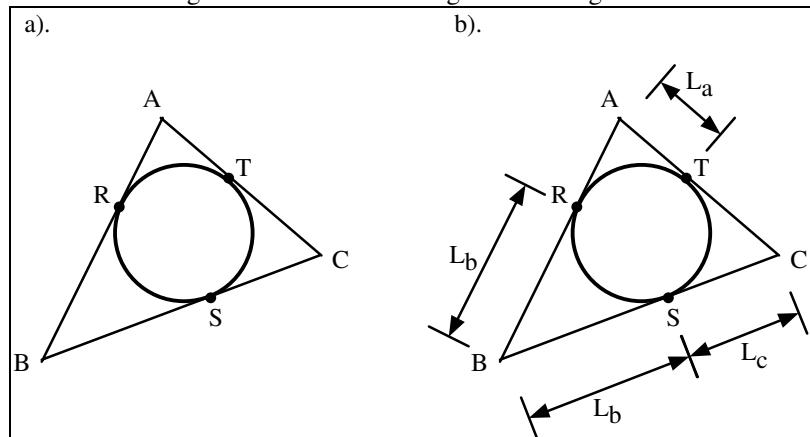


Figure 4.53. The inscribed circle of ABC is the excircle of RST .

As was the case with the excircle, the hard part is finding the center of the inscribed circle. A straightforward method¹⁴ recognizes that the inscribed circle of ABC is simply the excircle of a different set of three points, RST as shown in Figure 4.53a.

We need only find the locations of R , S , and T and then use the excircle method of Section 4.6.1. Figure 4.53b shows the distances of R , S , and T from A , B , and C . By the symmetry of a circle the distances $|B - R|$ and $|B - S|$ must be equal, and there are two other pairs of lines that have the same length. We therefore have (using the definitions of Equation 4.55 for **a**, **b**, and **c**):

$$|\mathbf{a}| = L_b + L_a, \quad |\mathbf{b}| = L_b + L_c, \quad |\mathbf{c}| = L_a + L_c$$

which can be combined to solve for L_a and L_b :

$$2L_a = |\mathbf{a}| + |\mathbf{c}| - |\mathbf{b}|, \quad 2L_b = |\mathbf{a}| + |\mathbf{b}| - |\mathbf{c}|$$

so L_a and L_b are now known. Thus R , S , and T are given by:

¹³Note: finding the incircle also solves the problem of finding the unique circle that is tangent to 3 noncollinear lines in the plane.

¹⁴Suggested by Russell Swan.

$$R = A + L_a \frac{\mathbf{a}}{|\mathbf{a}|} \quad (4.65)$$

$$S = B + L_b \frac{\mathbf{b}}{|\mathbf{b}|}$$

$$T = A - L_a \frac{\mathbf{c}}{|\mathbf{c}|}$$

(Check these expressions!)

Encapsulate the calculation of R , S , and T from A , B , and C in a simple routine having usage
`getTangentPoints(A, B, C, R, S, T)`. The advantage here is that if we have a routine
`excircle()` that takes three points and computes the center and radius of the excircle defined by them,
we can use the *same* routine to find the inscribed circle. Experiment with these tools.

The nine-point circle.

For any triangle, there are nine particularly distinguished points:

- the midpoints of the 3 sides;
- the feet of the 3 altitudes;
- the midpoints of the lines joining the orthocenter (where the 3 altitudes meet) to the vertices.

Remarkably, a single circle passes through all nine points! Figure 4.54 shows **the 9-point circle**¹⁵ for an example triangle. The nine-point circle is perhaps most easily drawn as the excircle of the midpoints of the sides of the triangle.

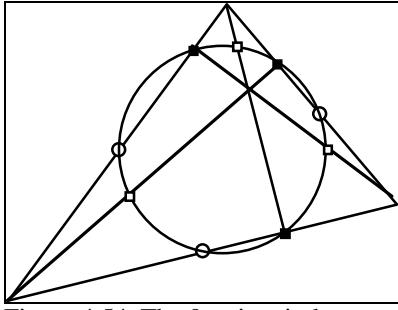


Figure 4.54. The 9-point circle.

4.10.3. Case Study 4.3. Is point Q inside convex polygon P ?

(Level of Effort: II.) Suppose you are given the specification of a convex polygon, P . Then given a point Q you are asked to determine whether or not Q lies inside P . But from the discussion on convex polygons in Section 4.8.1 we know this is equivalent to asking whether Q lies on the inside half space of *every* bounding line of P . For each bounding line L_i we need only test whether the vector $Q - P_i$ is more than 90° away from the outward pointing normal.

Fact: Q lies in P if $(Q - P_i) \cdot n_i < 0$ for $i = 0, 1, \dots, N-1$. (4.66)

Figure 4.55 illustrates the test for the particular bounding line that passes through P_1 and P_2 . For the case of point Q , which lies inside P , the angle with n_1 is greater than 90° . For the case of point Q' which lies outside P the angle is less than 90° .

¹⁵“This circle is the first really exciting one to appear in any course on elementary geometry.” Daniel Pedoe. *Circles*, Pergamon Press, New York, 1957

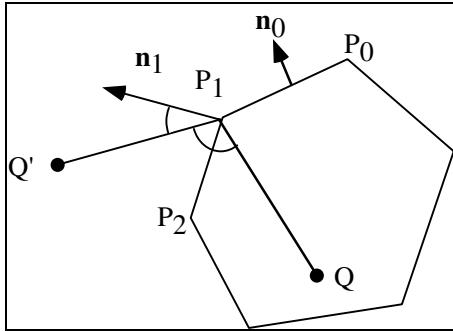


Figure 4.55. Is point Q inside polygon P ?

Write and test a program that allows the user to:

- lay down the vertices of a convex polygon, P , with the mouse;
- successively lay down test points, Q , with the mouse;
- prints “is inside” or “is not inside” depending on whether the point Q is or is not inside P .

4.10.4. Case Study 4.4. Reflections in a Chamber (2D Ray Tracing)

(Level of Effort: II.) This case study applies some of the tools and ideas introduced in this chapter to a fascinating yet simple simulation. The simulation performs a kind of ray tracing, based in a 2D world for easy visualization. Three dimensional ray tracing is discussed in detail in Chapter 14.

This simulation traces the path of a single tiny “pinball” as it bounces off various walls inside a “chamber.” Figure 4.56a shows a cross section of a convex chamber W that has six walls and contains three convex “pillars”. The pinball begins at point S and moves in a straight line in direction \mathbf{c} until it hits a barrier, whereupon it “reflects” off the barrier and moves in a new direction, again in a straight line. It continues to do this forever. Figure 4.56b shows an example of the polyline path that a ray traverses.

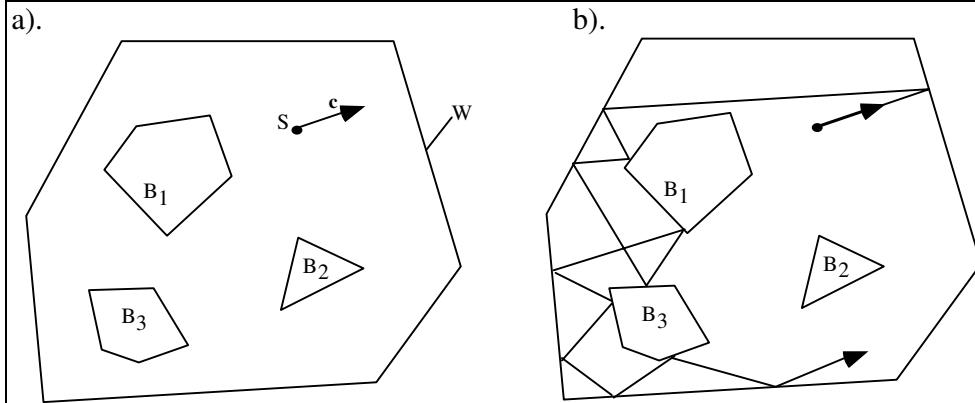


Figure 4.56. A 2D ray-tracing experiment.

For any given position S and direction \mathbf{c} of the ray, tracing its path requires two operations:

- Finding the first wall of the chamber “hit” by the ray;
- Finding the new direction the ray will take as it reflects off this first line.

Both of these operations have been discussed in the chapter. Note that as each new ray is created, its start point is always on some wall, the “hit point” of the previously hit wall.

We represent the chamber by a list of convex polygons, $pillar_0, pillar_1, \dots$, and arrange that $pillar_0$ is the “chamber” inside which the action takes place. The pillars are stored in suitable arrays of points. For each ray beginning at S and moving in direction \mathbf{c} , the entire array of pillars is scanned, and the intersection of the ray with each pillar is determined. This test is done using the Cyrus-Beck algorithm of Section 4.8.3. If

there is a hit with a pillar, the “hit time” is taken to be the time at which the ray “enters” the pillar. We encapsulate this test in the routine:

```
int rayHit(Ray thisRay, int which, double& tHit);
```

that calculates the hit time t_{Hit} of the ray $thisRay$ against $\text{pillar}_{\text{which}}$ and returns 1 if the ray hits the pillar, and 0 if it misses. A suitable type for `Ray` is `struct{Point2 startPt; Vector2 dir;}` or the corresponding class; it captures the starting point S and direction \mathbf{c} of the ray.

We want to know which pillar the ray hits first. This is done by keeping track of the earliest hit time as we scan through the list of pillars. Only positive hit times need to be considered: negative hit times correspond to hits at spots in the opposite direction from the ray’s travel. When the earliest hit point is found, the ray is drawn from S to it.

We must find the direction of the reflected ray as it moves away from this latest hit spot. The direction \mathbf{c}' of the reflected ray is given in terms of the direction \mathbf{c} of the incident ray by Equation 4.27:

$$\mathbf{c}' = \mathbf{c} - 2(\mathbf{c} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \quad (4.67)$$

where $\hat{\mathbf{n}}$ is the unit normal to the wall of the pillar that was hit. If a pillar inside the chamber was hit we use the outward pointing normal; if the chamber itself was hit, we use the inward pointing normal.

Write and exercise a program that draws the path of a ray as it reflects off the inner walls of chamber W and the walls of the convex pillars inside the chamber. Arrange to read in the list of pillars from an external file and to have the user specify the ray’s starting position and direction. (Also see Chapter 7 for the “elliptipool” 2D ray tracing simulation.)

4.10.5. Case Study 4.5. Cyrus-Beck Clipping.

(Level of Effort: II.) Write and exercise a program that clips a collection of lines against a convex polygon. The user specifies the polygon by laying down a sequence of points with the mouse (pressing key ‘C’ to terminate the polygon and begin clipping). Then a sequence of lines is generated, each having randomly chosen end points.

For each such line, the whole line is first drawn in red, then the portion that lies inside the polygon is drawn in blue.

4.10.6. Case Study 4.6. Clipping a polygon against a convex polygon — Sutherland Hodgman Clipping.

(Level of Effort: III.) Clipping algorithms studied so far clip individual line segments against polygons. When instead a *polygon* is clipped against a window it can be fragmented into several polygons in the clipping process, as suggested in Figure 4.57a. The polygon may need to be filled with a color or pattern, which means that each of the clipped fragments must be associated with that pattern, as suggested in Figure 4.57b. Therefore a clipping algorithm must keep track of edges ab , cd , and so on, and must fashion a new polygon (or polygons) out of the original one. It is also important that an algorithm not retain extraneous edges such as bc as part of the new polygon, as such edges would be displayed when they should in fact be invisible.

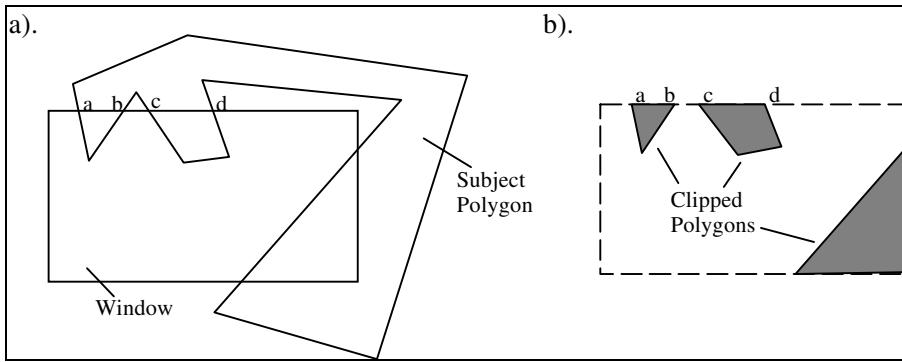


Figure 4.57. Clipping a polygon against a polygon.

The polygon to be clipped will be called the “subject” polygon, S . The polygon against which S is clipped will be called the “clip” polygon, C . How do we clip polygon S , represented by a vertex list, against polygon C , to generate a collection of vertex lists that properly represent the set of clipped polygons?

We examine here the **Sutherland–Hodgman** clipping algorithm. This method is quite simple and clips any subject polygon (convex or not) against a convex clip polygon. The algorithm can leave extraneous edges that must be removed later.

Because of the many different cases that can arise, we need an organized method for keeping track of the clipping process. The Sutherland–Hodgman algorithm takes a divide-and-conquer approach: It breaks a difficult problem into a set of simpler ones. It is built on the Cyrus-Beck approach, but must work with a *list* of vertices - that represent a polygon - rather than a simple pair of vertices.

Like the Cyrus-Beck algorithm this method clips polygon S against each bounding line of polygon C in turn, leaving only the part that is inside C . Once all of the edges of C have been used this way, S will have been clipped against C as desired. Figure 4.58 shows the algorithm in action for

1st edition Figure A6.2 on page 716.

Figure 4.58. Sutherland–Hodgman polygon clipping.

a seven-sided subject polygon S and a rectangular clip polygon C . We will describe each step in the process for this example. S is characterized by the vertex list $a\ b\ c\ d\ e\ f\ g$. S is clipped against the top, right, bottom, and left edges of C in turn, and at each stage a new list of vertices is generated from the old. This list describes one or more polygons and is passed along as the subject polygon for clipping against the next edge of C .

The basic operation, then, is to clip the polygon(s) described by an input vertex list V against the current clip edge of C and produce an output vertex list. To do this, traverse V , forming successive edges with pairs of adjacent vertices. Each such edge E has a first and a second endpoint we call s and p , respectively. There are four possible situations for endpoints s and p : s and p can both be inside, both can be outside, or they can be on opposite sides of the clip edge. In each case, certain points are output to (appended onto) the new vertex list, as shown in Figure 4.59.

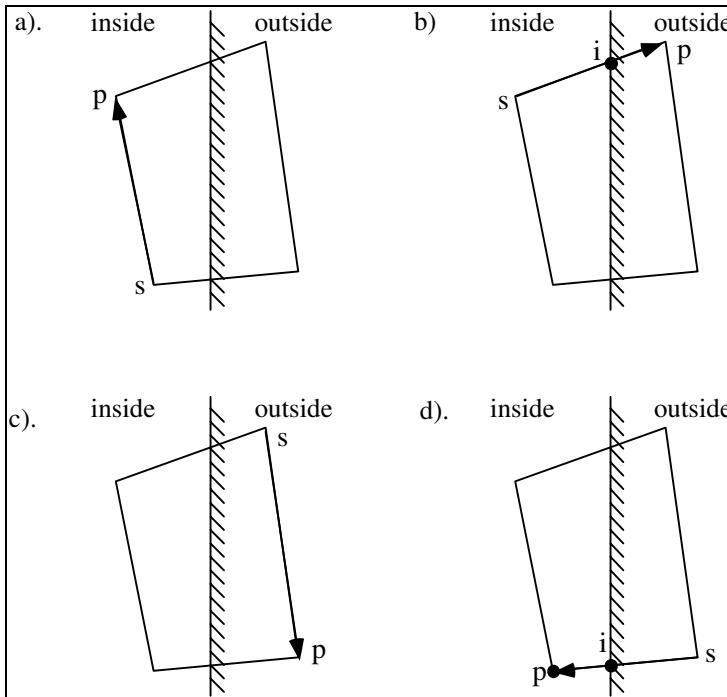


Figure 4.59. Four cases for each edge of S .

- Both s and p are inside: p is output.
- s is inside and p is outside. Find the intersection i and output it.
- Both s and p are outside. Nothing is output.
- s is outside and p is inside. Find intersection i , and output i and then p .

Now follow the progress of the Sutherland–Hodgman algorithm in Figure 4.58. Consider clipping S against the top edge of C . The input vertex list for this phase is $a\ b\ c\ d\ e\ f\ g$. The first edge from the list is taken for convenience as that from g to a , the edge that “wraps around” from the end of the list to its first element. Thus point s is g and point p is a here. Edge g, a , meaning the edge from g to a , intersects the clip edge at a new point “ I ”, which is output to the new list. (The output list from each stage in the algorithm is shown below the subsequent figure in Figure 4.58.) The next edge in the input list is a, b . Since both endpoints are above the clipping edge, nothing is output. The third edge, b, c , generates two output points, 2 and c , and the fourth edge, c, d , outputs point d . This process continues until the last edge, f, g , is tested, producing g . The new vertex list for the next clipping stage is therefore $1\ 2\ c\ d\ e\ f\ g$. It is illuminating to follow the example in Figure 4.58 carefully in its entirety to see how the algorithm works.

Notice that extraneous edges 3, 6 and 9, 10 are formed that connect the three polygon fragments formed in the clipping algorithm. Such edges can cause problems in some polygon filling algorithms. It is possible but not trivial to remove these offending edges [sutherland74].

Task: Implement the Sutherland–Hodgman clipping algorithm, and test it on a variety of sample polygons. The user lays down the convex polygon C with the mouse, then lays down the subject polygon S with the mouse. It is drawn in red as it is being laid down. Clipping is then performed, and the clipped polygon(s) are drawn in blue.

4.10.7. Case Study 4.7. Clipping a Polygon against another — Weiler Atherton Clipping.

(Level of Effort: III). This method provides the most general clipping mechanism of all we have studied. It clips any subject polygon against any (possibly non-convex) clip polygon. The polygons may even contain holes.

The Sutherland-Hodgman algorithm examined in Case Study 4.6 exploits the convexity of the clipping polygon through the use of inside-outside half-spaces. In some applications, such as hidden surface removal and rendering shadows, however, one must clip one concave polygon against another. Clipping is more complex in such cases. The Weiler–Atherton approach clips any polygon against any other, even when they have holes. It also allows one to form the set theoretic **union**, **intersection**, and **difference** of two polygons, as we discuss in Case Study 4.8.

We start with a simple example, shown in Figure 4.60. Here two concave polygons, *SUBJ* and *CLIP*, are represented by the vertex lists, (a, b, c, d) and (A, B, C, D) , respectively. We adopt the convention here of listing vertices so that the interior of the polygon is to the right of each edge as we move cyclically from vertex to vertex through the list. For instance, the interior of *SUBJ* lies to the right of the edge from *c* to *d* and to the right of that from *d* to *a*. This is akin to listing vertices in “clockwise” order.

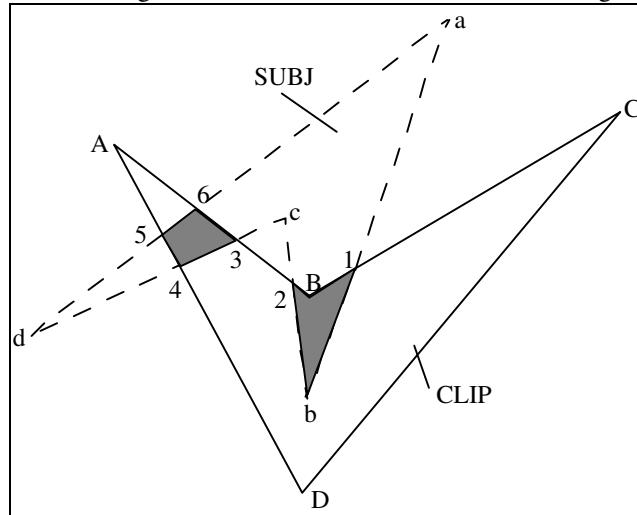


Figure 4.60 .Weiler–Atherton clipping.

All of the intersections of the two polygons are identified and stored in a list (see later). For the example here, there are six such intersections. Now to clip *SUBJ* against *CLIP*, traverse around *SUBJ* in the “forward direction” (i.e., so that its interior is to the right) until an “entering” intersection is found: one for which *SUBJ* is moving from the outside to the inside of *CLIP*. Here we first find 1, and it goes to an output list that records the clipped polygon(s).

The process is now simple to state in geometric terms: Traverse along *SUBJ*, moving segment by segment, until an intersection is encountered (2 in the example). The idea now is to turn away from following *SUBJ* and to follow *CLIP* instead. There are two ways to turn. Turn so that *CLIP* is traversed in its forward direction. This keeps the inside of both *SUBJ* and *CLIP* to the right. Upon finding an intersection, turn and follow along *SUBJ* in its forward direction, and so on. Each vertex or intersection encountered is put on the output list. Repeat the “turn and jump between polygons” process, traversing each polygon in its forward direction, until the first vertex is revisited. The output list at this point consists of $(1, b, 2, B)$.

Now check for any other entering intersections of *SUBJ*. Number 3 is found and the process repeats, generating output list $(3, 4, 5, 6)$. Further checks for entering intersections show that they have all been visited, so the clipping process terminates, yielding the two polygons $(1, b, 2, B)$ and $(3, 4, 5, 6)$. An organized way to implement this “follow in the forward direction and jump” process is to build the two lists

SUBLIST: $a, 1, b, 2, c, 3, 4, d, 5, 6$
CLIPLIST: $A, 6, 3, 2, B, 1, C, D, 4, 5$

that traverse each polygon (so that its interior is to the right) and list both vertices and intersections in the order they are encountered. (What should be done if no intersections are detected between the two

polygons?) Therefore traversing a polygon amounts to traversing a list, and jumping between polygons is effected by jumping between lists.

Notice that once the lists are available, there is very little geometry in the process—just a “point outside polygon” test to properly identify an entering vertex. The proper direction in which to traverse each polygon is embedded in the ordering of its list. For the preceding example, the progress of the algorithm is traced in Figure 4.61.

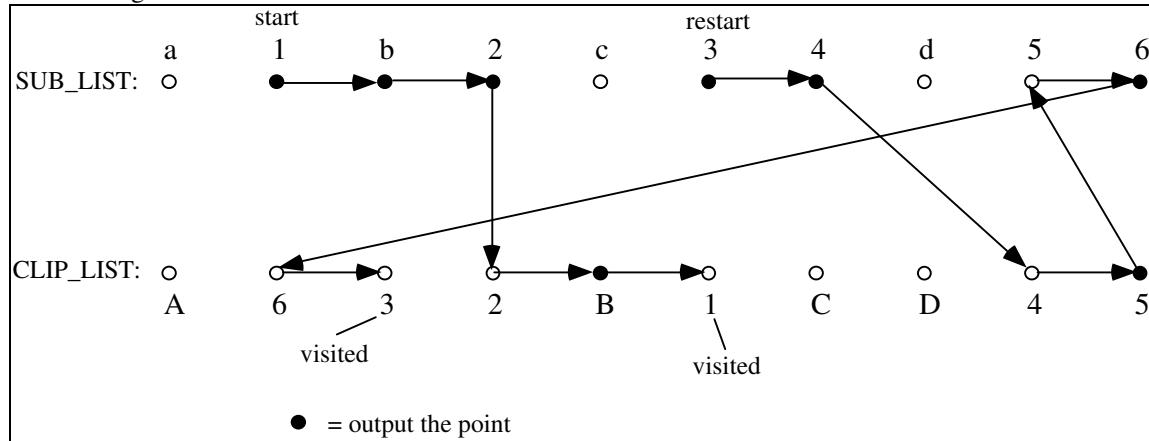


Figure 4.61. Applying the Weiler–Atherton method.

A more complex example involving polygons with holes is shown in Figure 4.62. The

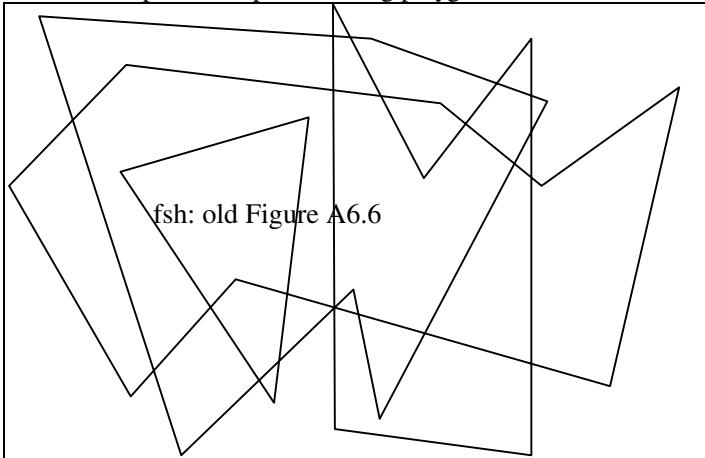


Figure 4.62. Weiler–Atherton clipping: polygons with holes.

vertices that describe holes are also listed in order such that the interior of the polygon lies to the right of an edge. (For holes this is sometimes called “counterclockwise order.”) The same rule is used as earlier: Turn and follow the other polygon in its forward direction. Beginning with entering intersection I , the polygon $(1, 2, 3, 4, 5, i, 6, H)$ is formed. Then, starting with entering intersection 7 , the polygon $(7, 8, 9, c, 10, F)$ is created. What entering intersection should be used to generate the third polygon? It is a valuable exercise to build $SUBJLIST$ and $CLIPLIST$ and to trace through the operation of the method for this example.

As with many algorithms that base decisions on intersections, we must examine the preceding method for cases where edges of $CLIP$ and $SUBJ$ are parallel and overlap over a finite segment.

Task: Implement the Weiler–Atherton clipping algorithm, and test it on a variety of polygons. Generate $SUBJ$ and $CLIP$ polygons, either in files or by letting the user lay down polygons with the mouse. In your implementation carefully consider how the algorithm will operate in situations such as the following:

- Some edges of $SUBJ$ and $CLIP$ are parallel and overlap over a finite segment,
- $SUBJ$ or $CLIP$ or both are nonsimple polygons,

- Some edges of $SUBJ$ and $CLIP$ overlap only at their endpoints,
- $CLIP$ and $SUBJ$ are disjoint,
- $SUBJ$ lies entirely within a hole of $CLIP$.

4.10.8. Case Study 4.8. Boolean Operations on Polygons.

(Level of Effort: III.) If we view polygons as sets of points (the set of all points on the boundary or in the interior of the polygon), then the result of the previous clipping operation is the **intersection** of the two polygons, the set of all points that are in both $CLIP$ and $SUBJ$. The polygons output by the algorithm consist of points that lie both within the original $SUBJ$ and within the $CLIP$ polygons. Here we generalize from intersections to other set theoretic operations on polygons, often called “Boolean” operations. Such operations arise frequently in modeling [mortenson85] as well as in graphics (see Chapter 14). In general, for any two sets of points A and B, the three set theoretic operations are

- intersection: $A \cap B = \{ \text{all points in both } A \text{ and } B \}$
- union: $A \cup B = \{ \text{all points in } A \text{ or in } B \text{ or both} \}$
- difference: $A - B = \{ \text{all points in } A \text{ but not } B \}$

with a similar definition for the set difference $B - A$. Examples of these sets are shown in Figure 4.63.

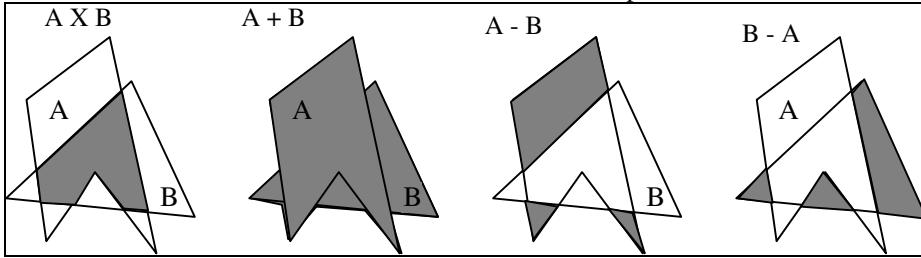


Figure 4.63. Polygons formed by boolean operations on polygons.

It is not hard to adjust the Weiler-Atherton method, which already performs intersections, to perform the union and difference operations on polygons A and B.

1. *Computing the union of A and B.* Traverse around A in the forward direction until an exiting intersection is found: one for which A is moving from the inside to the outside of B. Output the intersection and traverse along A until another intersection with B is found. Now turn to follow B in its forward direction. At each subsequent intersection, output the vertex and turn to follow the other polygon in its forward direction. Upon returning to the initial vertex, look for other exiting intersections that have not yet been visited.

2. *Computing the difference A - B(outside clipping).* Whereas finding the intersection of two polygons results in clipping one against the other, the difference operation “shields” one polygon from another. That is, the difference $SUBJ - CLIP$ consists of the parts of $SUBJ$ that lie outside $CLIP$. No parts of $SUBJ$ are drawn that lie within the border of $CLIP$, so the region defined by $CLIP$ is effectively protected, or shielded.

Traverse around A until an entering intersection into B is found. Turn to B, following it in the reverse direction, (so that B's interior is to the left). Upon reaching another intersection, jump to A again. At each intersection, jump to the other polygon, always traversing A in the forward direction and B in the reverse direction. Some examples of forming the union and difference of two polygons are shown in Figure 4.64. The three set operations generate the following polygons:

$POLYA \cup POLYB$:

- 4, 5, g, h(a hole)
- 8, B, C, D, 1, b, c, d
- 2, 3, i, j(a hole)
- 6, H, E, F, 7, f(a hole)

POLYA - POLYB:
4, 5, 6, *H*, *E*, *F*, 7, *e*, 8, *B*, *C*, *D*, *I*, a
2, 3, *k*

POLYB - POLYA:
1, *b*, *c*, *d*, 8, 5, *g*, *h*, 4, *A*, 3, *i*, *j*, 2
7, *f*, 6, *G*

1st edition Figure A6.8.

Figure 4.64. Forming the union and difference of two polygons.

Notice how the holes (*E*, *F*, *G*, *H*) and (*k*, *i*, *j*) in the polygons are properly handled, and that the algorithm generates holes as needed (holes are polygons listed in counterclockwise fashion).

Task: Adapt the Weiler–Atherton method so that it can form the union and difference of two polygons, and exercise your routines on a variety of polygons. Generate *A* and *B* polygons, either in files or algorithmically, to assist in the testing. Draw the polygons *A* and *B* in two different colors, and the result of the operation in a third color.

4.11. For Further Reading

Many books provide a good introduction to vectors. A favorite is Hoffmann’s ABOUT VECTORS. The GRAPHICS GEMS series [gems] provides an excellent source of new approaches and results in vector arithmetic and geometric algorithms by computer graphics practitioners. Three excellent example articles are Alan Paeth’s “A Half-Angle Identity for Digital Computation: The Joys of the Half Tangent” [paeth91], Ron Goldman “Triangles”[goldman90], and Lopez-Lopez’s “Triangles Revisited” [lopez92]. Two books that delve more deeply into the nature of geometric algorithms are Moret and Shapiro’s ALGORITHMS FROM P TO NP [moret91] and Preparata and Shamos’s COMPUTATIONAL GEOMETRY, AN INTRODUCTION [preparata85].

Chapter 5. Transformations of Objects

“Minus times minus is plus, the reason for this we need not discuss”
W.H.Auden

*“If I eat one of these cakes“, she thought,
”it’s sure to make some change in my size.“
So she swallowed one ...
and was delighted to find that she began shrinking directly.*
Lewis Carroll, Alice in Wonderland

*Many of the brightly colored tile-covered walls and floors of the
Alhambra in Spain show us that the Moors were masters
in the art of filling a plane with similar interlocking figures,
bordering each other without gaps.
What a pity that their religion forbade them to make images!*
M. C. Escher

Goals of the Chapter

- Develop tools for transforming one picture into another.
- Introduce the fundamental concepts of affine transformations, which perform combinations of rotations, scalings, and translations.
- Develop functions that apply affine transformations to objects in computer programs
- Develop tools for transforming coordinate frames.
- See how to set up a camera to render a 3D scene using OpenGL
- Learn to design scenes in the Scene Design language SDL, and to write programs that read SDL files and draw the scenes they describe.

Preview.

Section 5.1 motivates the use of 2D and 3D transformations in computer graphics, and sets up some basic definitions. Section 5.2 defines 2D affine transformations and establishes terminology for them in terms of a matrix. The notation of coordinate frames is used to keep clear what objects are being altered and how. The section shows how elementary affine transformations can perform scaling, rotation, translation, and shearing. Section 5.2.5 shows that you can combine as many affine transformations as you wish, and the result is another affine transformation, also characterized by a matrix. Section 5.2.7 discusses key properties of all affine transformations — most notably that they preserve straight lines, planes, and parallelism — and shows why they are so prevalent in computer graphics.

Section 5.3 extends these ideas to 3D affine transformations, and shows that all of the basic properties hold here as well. 3D transformations are more complex than 2D ones, however, and more difficult to visualize, particularly when it comes to 3D rotations. So special attention is paid to describing and combining various rotations.

Section 5.4 discusses the relationship between transforming points and transforming coordinate systems. Section 5.5 shows how transformations are managed within a program when OpenGL is available, and how transformations can greatly simplify many operations commonly needed in a graphics program. Modeling transformations and the use of the “current transformation” are motivated through a number of examples. Section 5.6 discusses modeling 3D scenes and drawing them using OpenGL. A “camera” is defined that is positioned and oriented so that it takes the desired snapshot of the scene. The section discusses how transformations are used to size and position objects as desired in a scene. Some example 3D scenes are modeled and rendered, and the code required to do it is examined. This section also introduces a Scene Description language, SDL, and shows how to write an application that can draw any scene described in the language. This

requires the development of a number of classes to support reading and parsing SDL files, and creating lists of objects that can be rendered. These classes are available from the book's web site.

The chapter ends with a number of Case Studies that elaborate on the main ideas and provide opportunities to work with affine transformations in graphics programs. One case study asks you to develop routines that perform transformations when OpenGL is not available. Also described there are ways to decompose an affine transformation into its elementary operations, and the development of a fast routine to draw arcs of circles that capitalizes on the equivalence between a rotation and three successive shears.

5.1. Introduction.

The main goal in this chapter is to develop techniques for working with a particularly powerful family of transformations called *affine transformations*, both with pencil and paper and in a computer program, with and without OpenGL. These transformations are a fundamental cornerstone of computer graphics, and are central to OpenGL as well as most other graphics systems. They are also a source of difficulty for many programmers because it is often difficult to get them right.

One particularly delicate area is the confusion of points and vectors. Points and vectors seem very similar, and are often expressed in a program using the same data type, perhaps a list of three numbers like (3.0, 2.5, -1.145) to express them in the current coordinate system. But this practice can lead to disaster in the form of serious bugs that are very difficult to ferret out, principally because points and vectors do *not* transform the same way. We need a way to keep them straight, which is offered by using *coordinate frames* and appropriate homogeneous coordinates as introduced in Chapter 4.

5.2. Introduction to Transformations.

The universe is full of magical things, patiently waiting for our wits to grow sharper.
E. Phillpotts

We have already seen some examples of transformations, at least in 2D. In Chapter 3, for instance, the window to viewport transformation was used to scale and translate objects situated in the world window to their final size and position in the viewport.

We want to build on those ideas, and gain more flexible control over the size, orientation, and position of objects of interest. In the following sections we develop the required tools, based on the powerful **affine transformation**, which is a staple in computer graphics. We operate in both two and three dimensions.

Figure 5.1a shows two versions of a simple house, drawn before and after each of its points has been transformed. In this case the house has been scaled down in size, rotated a small amount, and then moved up and to the right. The overall transformation here is a combination of three more elementary ones: scaling, rotation, and translation. Figure 5.1b shows a 3D house before and after it is similarly transformed: each 3D point in the house is subjected by the transformation to a scaling, a rotation, and a translation.

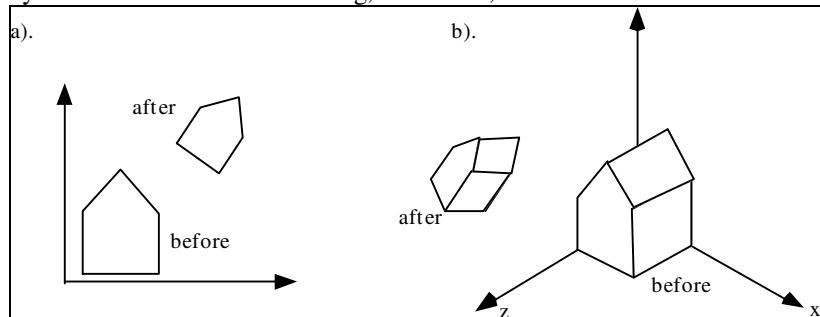


Figure 5.1. Drawings of objects before and after they are transformed.

Transformations are very useful in a number of situations:

- a). We can compose a “scene” out of a number of objects, as in Figure 5.2. Each object such as the arch is most easily designed (once for all) in its own “master” coordinate system. The scene is then fashioned by placing a number of “instances” of the arch at different places and with different sizes, using the proper transformation for each.

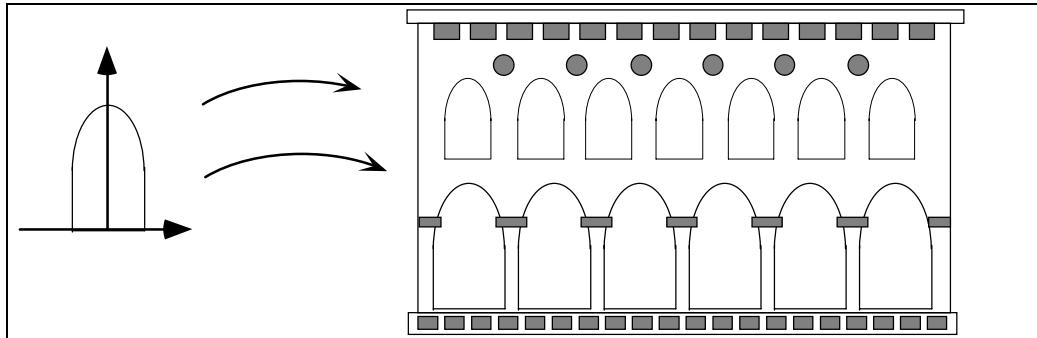


Figure 5.2. Composing a picture from many instances of a simple form.

Figure 5.3 shows a 3D example, where the scene is composed of many instances of cubes that have been scaled and positioned in a “city”.

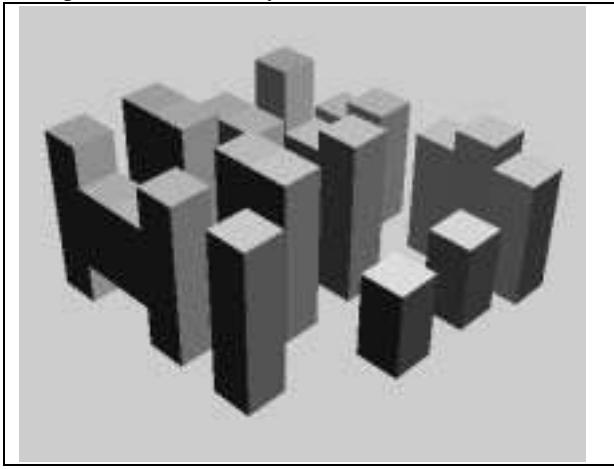


Figure 5.3. Composing a 3D scene from primitives.

- b). Some objects, such as the snowflake shown in Figure 5.4, exhibit certain symmetries. We can design a single “motif” and then fashion the whole shape by appropriate reflections, rotations, and translations of the motif.

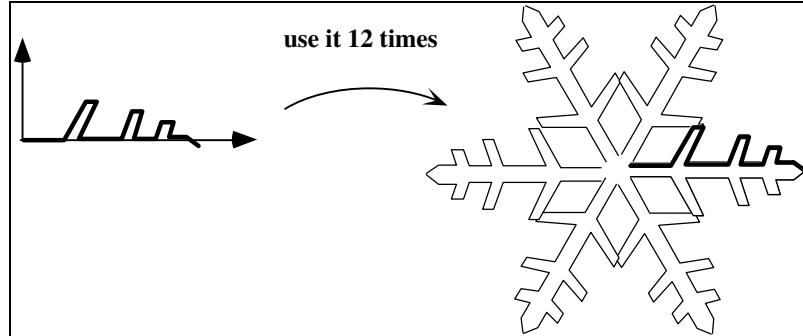


Figure 5.4. Using a “motif” to build up a figure.

- c). A designer may want to view an object from different vantage points, and make a picture from each one. The scene can be rotated and viewed with the same camera. But as suggested in Figure 5.5 it’s more natural to leave

the scene alone and move the camera to different orientations and positions for each snapshot. Positioning and reorienting a camera can be carried out through the use of 3D affine transformations.

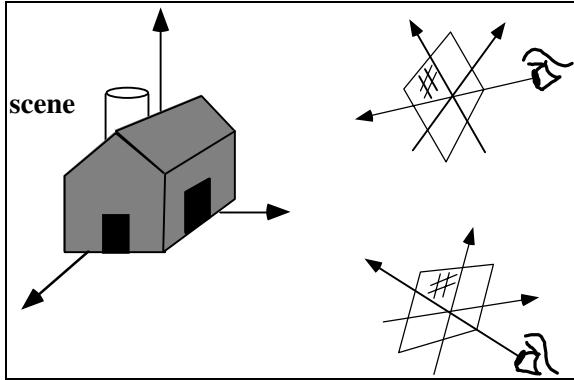


Figure 5.5. Viewing a scene from different points of view.

d). In a computer animation several objects must move relative to one another from frame to frame. This can be achieved by shifting and rotating their local coordinate systems as the animation proceeds. Figure 5.6 shows an example.

author supplied

Figure 5.6. Animating by transforming shapes.

Where are we headed? Using Transformations with OpenGL.

The first few sections of this chapter present the basic concepts of affine transformations, and show how they produce certain geometric effects such as scaling, rotations, and translations, both in 2D and 3D space.

Ultimately, of course, the goal is to produce graphical drawings of objects that have been transformed to the proper size, orientation, and position so they produce the desired scene. A number of graphics platforms, including OpenGL, provide a “graphics pipeline”, or sequence of operations that are applied to all points that are “sent through it”. A drawing is produced by processing each point.

Figure 5.7 shows a simplified view of the OpenGL graphics pipeline. An application “sends it” a sequence of points P_1, P_2, P_3, \dots using the commands like the now familiar:

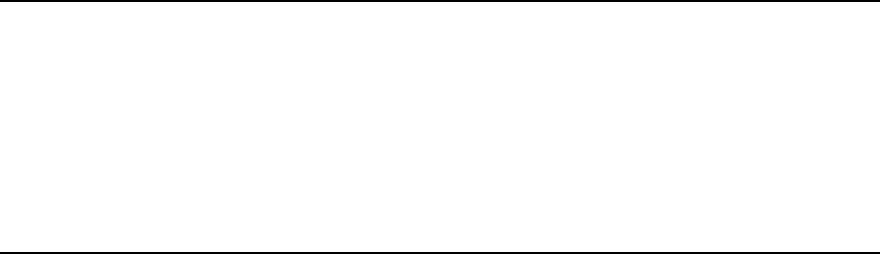


Figure 5.7. The OpenGL pipeline.

```
glBegin(GL_LINES);
    glVertex3f(...); // send P1 through the pipeline
    glVertex3f(...); // send P2 through the pipeline
    glVertex3f(...); // send P3 through the pipeline
    ...
glEnd();
```

As shown in the figure these points first encounter a transformation called the “current transformation” (“CT”), which alters their values into a different set of points, say Q_1, Q_2, Q_3, \dots . Just as the original points P_i describe some geometric object, the points Q_i describe the transformed version of the same object. These points are then sent through additional steps, and ultimately are used to draw the final image on the display.

The current transformation therefore provides a crucial tool in the manipulation of graphical objects, and it is essential for the application programmer to know how to adjust the CT so that the desired transformations are produced. After developing the underlying theory of affine transformations, we turn in Section 5.5 to showing how this is done.

Object transformations versus coordinate transformations.

There are two ways to view a transformation: as an **object transformation** or as a **coordinate transformation**.

An object transformation alters the coordinates of each point on the object according to some rule, leaving the underlying coordinate system fixed. A coordinate transformation defines a new coordinate system in terms of the old one, then represents all of the object's points in this new system. The two views are closely connected, and each has its advantages, but they are implemented somewhat differently. We shall first develop the central ideas in terms of object transformations, and then relate them to coordinate transformations.

5.2.1. Transforming Points and Objects.

We look here at the general idea of a transformation, and then specialize to affine transformations.

A transformation alters each point, P , in space (2D or 3D) into a new point, Q , using a specific formula or algorithm. Figure 5.8 shows 2D and 3D examples.

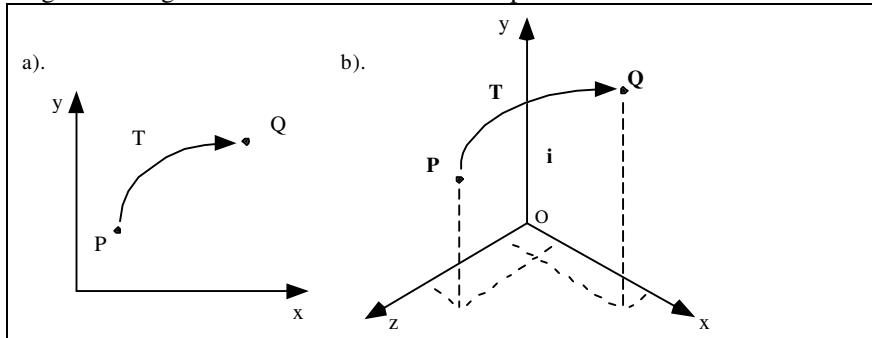


Figure 5.8. Mapping points into new points.

As Figure 5.8 illustrates, an arbitrary point P in the plane is **mapped** to Q . We say Q is the **image** of P under the mapping T . Part a) shows a 2D point P being mapped to a new point Q ; part b) shows a 3D point P being mapped to a new Q . We transform an object by transforming each of its points, using, of course, the same function $T()$ for each point. We can map whole collections of points at once. The collection might be all the points on a line or circle. The **image** of line L under T , for instance, consists of the images of all the individual points of L .¹

Most mappings of interest are continuous, so the image of a straight line is still a connected curve of some shape, although it's not necessarily a straight line. Affine transformations, however, do preserve lines as we shall see: The image under T of a straight line is also a straight line. Most of this chapter will focus on affine transformations, but other kinds can be used to create special effects. Figure 5.9, for instance, shows a complex warping of a figure that cannot be achieved with an affine transformation. This transformation might be used for visual effect, or to emphasize important features of an object.

old Fig 11.2 Da Vinci warped peculiarly

Figure 5.9. A complex warping of a figure.

To keep things straight we use an explicit coordinate frame when performing transformations. Recall from Chapter 4 that a coordinate frame consists of a particular point, \mathbf{v} , called the origin, and some mutually perpendicular vectors (called \mathbf{i} and \mathbf{j} in the 2D case; \mathbf{i} , \mathbf{j} , and \mathbf{k} in the 3D case) that serve as the axes of the coordinate frame.

¹More formally, if S is a set of points, its **image** $T(S)$ is the set of all points $T(P)$ where P is some point in S .

Take the 2D case first, as it is easier to visualize. In whichever coordinate frame we are using, point P and Q have the representations \tilde{P} and \tilde{Q} given by:

$$\tilde{P} = \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}, \tilde{Q} = \begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix}$$

Recall that this means the point P “is at” location $P = P_x \mathbf{i} + P_y \mathbf{j} + \mathbf{0}$, and similarly for Q . P_x and P_y are familiarly called the “coordinates” of P . The transformation operates on the representation \tilde{P} and produces the representation \tilde{Q} according to some function, $T()$,

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = T \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} \quad (5.1)$$

or more succinctly,

$$\tilde{Q} = T(\tilde{P}). \quad (5.2)$$

The function $T()$ could be complicated, as in

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(P_x) e^{-P_y} \\ \frac{\ln(P_y)}{1+P_x^2} \\ 1 \end{pmatrix}$$

and such transformations might have interesting geometric effects, but we restrict ourselves to much simpler families of functions, those that are *linear* in P_x and P_y . This property characterizes the affine transformations.

5.2.2. The Affine Transformations.

What is algebra, exactly? Is it those three-cornered things?
J. M. Barrie

Affine transformations are the most common transformations used in computer graphics. Among other things they make it easy to scale, rotate, and reposition figures. A succession of affine transformations can easily be combined into a simple overall affine transformation, and affine transformations permit a compact matrix representation.

Affine transformations have a simple form: the coordinates of Q are linear combinations of those of P :

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11}P_x + m_{12}P_y + m_{13} \\ m_{21}P_x + m_{22}P_y + m_{23} \\ 1 \end{pmatrix} \quad (5.3)$$

for some six given constants m_{11} , m_{12} , etc. Q_x consists of portions of both of P_x and P_y , and so does Q_y . This “cross fertilization” between the x - and y -components gives rise to rotations and shears.

The affine transformation of Equation 5.3 has a useful matrix representation that helps to organize your thinking:²

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} \quad (5.4)$$

(Just multiply this out to see that it's the same as Equation 5.3. In particular, note how the third row of the matrix forces the third component of Q to be 1.) For an affine transformation the third row of the matrix is always $(0, 0, 1)$.

Vectors can be transformed as well as points. Recall that if vector V has coordinates V_x and V_y then its coordinate frame representation is a column vector with a third component of 0. When transformed by the same affine transformation as above the result is

$$\begin{pmatrix} W_x \\ W_y \\ 0 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} V_x \\ V_y \\ 0 \end{pmatrix} \quad (5.5)$$

which is clearly another vector: its third component is always 0.

Practice Exercise 5.2.1. Apply the transformation. An affine transformation is specified by the matrix:

$$\begin{pmatrix} 3 & 0 & 5 \\ -2 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Find the image Q of point $P = (1, 2)$.

Solution: $\begin{pmatrix} 8 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 & 0 & 5 \\ -2 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$.

5.2.3. Geometric Effects of Elementary 2D Affine Transformations.

What geometric effects are produced by affine transformations? They produce combinations of four elementary transformations: (1) a translation, (2) a scaling, (3) a rotation, and (4) a shear.

Figure 5.10 shows an example of the effect of each kind of transformation applied individually.

1st Ed. Figure 11.5

Figure 5.10. Transformations of a map: a) translation, b) scaling, c)rotation, d) shear.

Translation.

You often want to translate a picture into a different position on a graphics display. The translation part of the affine transformation arises from the third column of the matrix

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & m_{13} \\ 0 & 1 & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} \quad (5.6)$$

²See Appendix 2 for a review of matrices.

or simply

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} P_x + m_{13} \\ P_y + m_{23} \\ 1 \end{pmatrix}$$

so in ordinary coordinates $Q = P + \mathbf{d}$, where “offset vector” \mathbf{d} has components (m_{13}, m_{23}) .

For example, if the offset vector is $(2, 3)$, every point will be altered into a new point that is two units farther to the right and three units above the original point. The point $(1, -5)$, for instance, is transformed into $(3, -2)$, and the point $(0, 0)$ is transformed into $(2, 3)$.

Scaling.

A scaling changes the size of a picture and involves two scale factors, S_x and S_y , for the x - and y -coordinates, respectively:

$$(Q_x, Q_y) = (S_x P_x, S_y P_y)$$

Thus the matrix for a scaling by itself is simply

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (5.7)$$

Scaling in this fashion is more accurately called **scaling about the origin**, because each point P is moved S_x times farther from the origin in the x -direction, and S_y times farther from the origin in the y -direction. If a scale factor is negative, then there is also a **reflection** about a coordinate axis. Figure 5.11 shows an example in which the scaling $(S_x, S_y) = (-1, 2)$ is applied to a collection of points. Each point is both reflected about the y -axis and scaled by 2 in the y -direction.

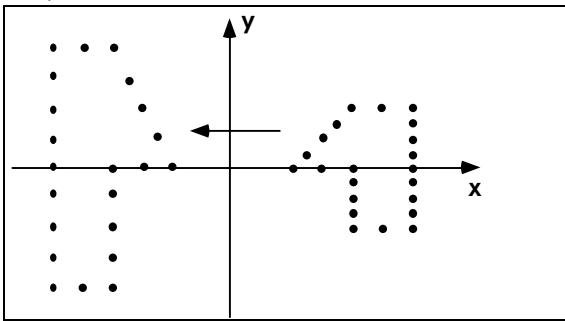


Figure 5.11. A scaling and a reflection.

There are also “pure” reflections, for which each of the scale factors is $+1$ or -1 . An example is

$$T(P_x, P_y) = (-P_x, P_y) \quad (5.8)$$

which produces a mirror image of a picture by “flipping” it horizontally about the y -axis, replacing each occurrence of x with $-x$. (What is the matrix of this transformation?)

If the two scale factors are the same, $S_x = S_y = S$, the transformation is a **uniform scaling**, or a magnification about the origin, with magnification factor $|S|$. If S is negative, there are reflections about both axes. A point is

moved outward from the origin to a position $|S|$ times farther away from the origin. If $|S| < 1$, the points will be moved closer to the origin, producing a reduction (or “demagnification”). If, on the other hand, the scale factors are not the same, the scaling is called a **differential scaling**.

Practice Exercise 5.2.2. Sketch the effect. A pure scaling affine transformation uses scale factors $S_x = 3$ and $S_y = -2$. Find the image of each of the three objects in Figure 5.12 under this transformation, and sketch them. (Make use of the facts - to be proved later - that an affine transformations maps straight lines to straight lines, and ellipses to ellipses.)

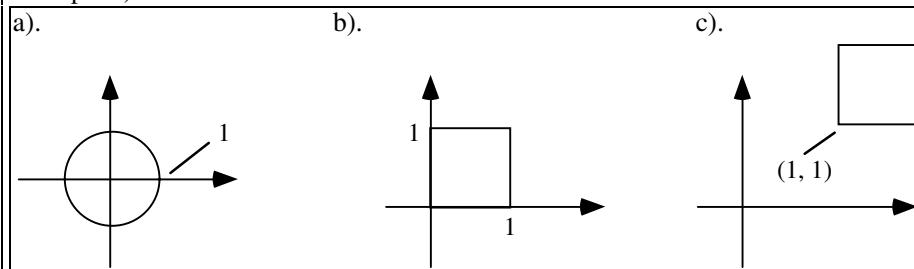


Figure 5.12. Objects to be scaled.

Rotation.

A fundamental graphics operation is the rotation of a figure about a given point through some angle. Figure 5.13 shows a set of points rotated about the origin through an angle of $\theta = 60^\circ$.

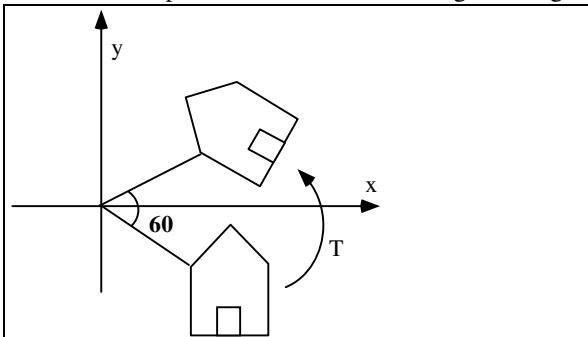


Figure 5.13. Rotation of points through an angle of 60° .

When $T(\cdot)$ is a rotation about the origin, the offset vector \mathbf{d} is zero and $Q = T(P)$ has the form

$$\begin{aligned} Q_x &= P_x \cos(\theta) - P_y \sin(\theta) \\ Q_y &= P_x \sin(\theta) + P_y \cos(\theta) \end{aligned} \tag{5.9}$$

As we derive next, this form causes positive values of θ to perform a counterclockwise (CCW) rotation. In terms of its matrix form, a pure rotation about the origin is given by

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.10}$$

Example 5.2.1. Find the transformed point, Q , caused by rotating $P = (3, 5)$ about the origin through an angle of 60° . **Solution:** For an angle of 60° , $\cos(\theta) = .5$ and $\sin(\theta) = .866$, and Equation 5.9 yields $Q_x = (3)(0.5) - (5)(0.866) = -2.83$ and $Q_y = (3)(0.866) + (5)(0.5) = 5.098$. Check this on graph paper by swinging an arc of 60° from $(3, 5)$ and reading off the position of the mapped point. Also check numerically that Q and P are at the same distance from the origin. (What is this distance?)

Derivation of the Rotation Mapping.

We wish to demonstrate that Equation 5.9 is correct. Figure 5.14 shows how to find the coordinates of a point Q that results from rotating point P about the origin through an angle θ . If P is at a distance R from the origin, at some angle ϕ , then $P = (R \cos(\phi), R \sin(\phi))$. Now Q must be at the same distance as P , and at angle $\theta + \phi$. Using trigonometry, the coordinates of Q are

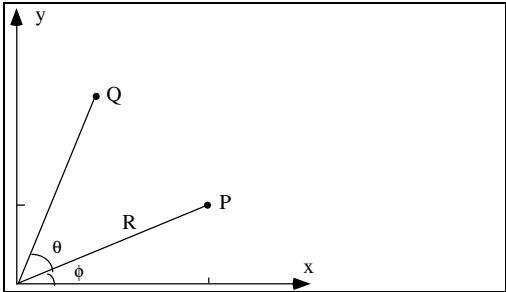


Figure 5.14. Derivation of the rotation mapping.

$$Q_x = R \cos(\theta + \phi)$$

$$Q_y = R \sin(\theta + \phi)$$

Substitute into this equation the two familiar trigonometric relations:

$$\cos(\theta + \phi) = \cos(\theta) \cos(\phi) - \sin(\theta) \sin(\phi)$$

$$\sin(\theta + \phi) = \sin(\theta) \cos(\phi) + \cos(\theta) \sin(\phi)$$

and use $P_x = R \cos(\phi)$ and $P_y = R \sin(\phi)$ to obtain Equation 5.9.

Practice Exercise 5.2.3. Rotate a Point. Use Equation 5.9 to find the image of each of the following points after rotation about the origin:

- a). $(2, 3)$ through an angle of -45°
- b). $(1, 1)$ through an angle of -180° .
- c). $(60, 61)$ through an angle of 4° .

In each case check the result on graph paper, and compare numerically the distances of the original point and its image from the origin.

Solution: a). $(3.5355, .7071)$, b). $(-1, -1)$, c). $(55.5987, 65.0368)$.

Shearing.

An example of shearing is illustrated in Figure 5.15 is a shear “in the x -direction” (or “along x ”). In this case the y -coordinate of each point is unaffected, whereas each x -coordinate is translated by an amount that increases linearly with y . A shear in the x -direction is given by

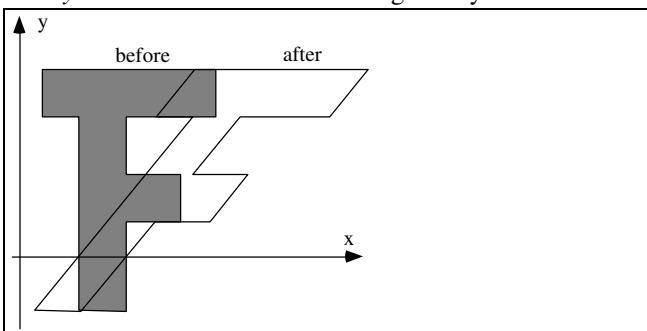


Figure 5.15. An example of shearing.

$$Q_x = P_x + h P_y$$

$$Q_y = P_y$$

where the coefficient h specifies what fraction of the y -coordinate of P is to be added to the x -coordinate. The quantity h can be positive or negative. Shearing is sometimes used to make italic letters out of regular letters. The matrix associated with this shear is:

$$\begin{pmatrix} 1 & h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (5.11)$$

One can also have a shear “along y ”, for which $Q_x = P_x$ and $Q_y = g P_x + P_y$ for some value g , so that the matrix is given by

$$\begin{pmatrix} 1 & 0 & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (5.12)$$

Example 5.2.2: Into which point does $(3, 4)$ shear when $h = .3$ in Equation 5.11? **Solution:** $Q = (3 + (.3)4, 4) = (4.2, 4)$.
Example 5.2.3: Let $g = 0.2$ in Equation 5.12. To what point does $(6, -2)$ map? **Solution:** $Q = (6, 0.2 \cdot 6 - 2) = (6, -0.8)$.

A more general shear “along” an arbitrary line is discussed in a Case Study at the end of the chapter. A notable feature of a shear is that its matrix has a determinant of 1. As we see later this implies that the area of a figure is unchanged when it is sheared.

Practice Exercise 5.2.4. Shearing Lines. Consider the shear for which $g = .4$ and $h = 0$. Experiment with various sets of three collinear points to build some assurance that the sheared points are still collinear. Then, assuming that lines do shear into lines, determine into what objects the following line segments shear:

- the horizontal segment between $(-3, 4)$ and $(2, 4)$;
- the horizontal segment between $(-3, -4)$ and $(2, -4)$;
- the vertical segment between $(-2, 5)$ and $(-2, -1)$;
- the vertical segment between $(2, 5)$ and $(2, -1)$;
- the segment between $(-1, -2)$ and $(3, 2)$;

Into what shapes do each of the objects in Figure 5.2.12 shear?

5.2.4. The Inverse of an Affine Transformation

Most affine transformations of interest are **nonsingular**, which means that the determinant of M in Equation 5.4, which evaluates to³

$$\det M = m_{11}m_{22} - m_{12}m_{21} \quad (5.13)$$

is nonzero. Notice that the third column of M , which represents the amount of translation, does not affect the determinant. This is a direct consequence of the two zeroes appearing in the third row of M . We shall make special note on those rare occasions that we use singular transformations.

³ See Appendix 2 for a review of determinants.

It is reassuring to be able to undo the effect of a transformation. This is particularly easy to do with nonsingular affine transformations. If point P is mapped into point Q according to $Q = MP$, simply premultiply both sides by the **inverse** of M , denoted M^{-1} , and write

$$P = M^{-1} Q \quad (5.14)$$

The inverse of M is given by⁴

$$M^{-1} = \frac{1}{\det M} \begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix} \quad (5.15)$$

We therefore obtain the following matrices for the elementary inverse transformations:

- **Scaling** (use M as found in Equation 5.7):

$$M^{-1} = \begin{pmatrix} \frac{1}{S_x} & 0 & 0 \\ 0 & \frac{1}{S_y} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- **Rotation** (use M as found in Equation 5.10):

$$M^{-1} = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- **Shearing** (using the version of M in Equation 5.11):

$$M^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -h & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- **Translations**: The inverse transformation simply subtracts the offset rather than adds it.

$$M^{-1} = \begin{pmatrix} 1 & 0 & -m_{13} \\ 0 & 1 & -m_{23} \\ 0 & 0 & 1 \end{pmatrix}$$

Practice Exercises.

5.2.5. What Is the Inverse of a Rotation? Show that the inverse of a rotation through θ is a rotation through $-\theta$. Is this reasonable geometrically? Why?

5.2.6. Inverting a Shear. Is the inverse of a shear also a shear? Show why or why not.

5.2.7. An Inverse Matrix. Compute the inverse of the matrix

$$M = \begin{pmatrix} 3 & 2 & 1 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

⁴ See Appendix 2 for a review of inverse matrices.

5.2.5. Composing Affine Transformations.

Progress might have been all right once , but it has gone on too long.
Ogden Nash

It's rare that we want to perform just one elementary transformation; usually an application requires that we build a compound transformation out of several elementary ones. For example, we may want to

- translate by (3, - 4)
- then rotate through 30^0
- then scale by (2, - 1)
- then translate by (0, 1.5)
- and finally rotate through $- 30^0$

How do these individual transformations combine into one overall transformation? The process of applying several transformations in succession to form one overall transformation is called **composing** (or **concatenating**) the transformations. As we shall see, when two affine transformations are composed, the resulting transformation is (happily) also affine.

Consider what happens when two 2D transformations, $T_1()$ and $T_2()$, are composed. As suggested in Figure 5.16, $T_1()$ maps P into Q , and $T_2()$ maps Q into point W . What is the transformation, $T()$, that maps P directly into W ? That is, what is the nature of $W = T_2(Q) = T_2(T_1(P))$?

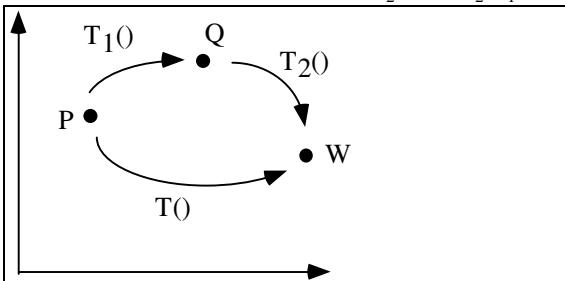


Figure 5.16. The composition of two transformations.

Suppose the two transformations are represented by the matrices \tilde{M}_1 and \tilde{M}_2 . Thus \tilde{P} is first transformed to the point $\tilde{M}_1 \tilde{P}$ which is then transformed to $\tilde{M}_2 (\tilde{M}_1 \tilde{P})$. By associativity this is just $(\tilde{M}_2 \tilde{M}_1) \tilde{P}$, and so we have

$$\tilde{W} = \tilde{M} \tilde{P} \quad (5.16)$$

where the overall transformation is represented by the single matrix

$$\tilde{M} = \tilde{M}_2 \tilde{M}_1. \quad (5.17)$$

When homogeneous coordinates are used, composing affine transformations is accomplished by simple matrix multiplication. Notice that the matrices appear in *reverse* order to that in which the transformations are applied: if we first apply T_1 with matrix \tilde{M}_1 , and then apply T_2 with matrix \tilde{M}_2 to the result, the overall transformation has matrix $\tilde{M}_2 \tilde{M}_1$, with the “second” matrix appearing first in the product as you read from left to right. (Just the opposite order will be seen when we transform coordinate systems.)

By applying the same reasoning, any number of affine transformations can be composed simply by multiplying their associated matrices. In this way, transformations based on an arbitrary succession of rotations, scalings, shears, and translations can be formed and captured in a single matrix.

Example 5.2.4. Build one. Build a transformation that

- rotates through 45 degrees;
- then scales in x by 1.5 and in y by -2;
- then translates through $(3, 5)$.

Find the image under this transformation of the point $(1, 2)$.

Solution: Construct the three matrices and multiply them in the proper order (first one last, etc.) to form:

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1.5 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} .707 & -.707 & 0 \\ .707 & .707 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1.06 & -1.06 & 3 \\ -1.414 & -1.414 & 5 \\ 0 & 0 & 1 \end{pmatrix}$$

Now to transform point $(1, 2)$, enlarge it to the triple $(1, 2, 1)$, multiply it by the composite matrix to obtain $(1.94, 0.758, 1)$, and drop the one to form the image point $(1.94, 0.758)$. It is instructive to use graph paper, and to perform each of this transformations in turn to see how $(1, 2)$ is mapped.

5.2.6. Examples of Composing 2D Transformations.

*Art is the imposing of a pattern on experience,
and our aesthetic enjoyment is recognition of the pattern.
Alfred North Whitehead*

We examine some important examples of composing 2D transformations, and see how they behave.

Example 5.2.5. Rotating About an Arbitrary Point.

So far all rotations have been about the origin. But suppose we wish instead to rotate points about some other point in the plane. As suggested in Figure 5.17, the desired “pivot” point is $V = (V_x, V_y)$, and we wish to rotate points such as P through angle θ to position Q . To do this we must relate the rotation about V to an elementary rotation about the origin.

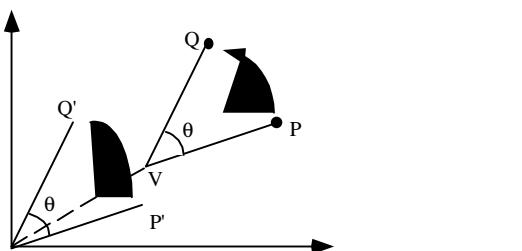


Figure 5.17. Rotation about a point.

Figure 5.2.17 shows that if we first translate all points so that V coincides with the origin, then a rotation about the origin (which maps P' to Q') will be appropriate. Once done, the whole plane is shifted back to restore V to its original location. The rotation therefore consists of the following three elementary transformations:

1. Translate point P through vector $v = (-V_x, -V_y)$;
2. Rotate about the origin through angle θ ;
3. Translate P back through v .

Creating a matrix for each elementary transformation, and multiplying them out produces:

$$\begin{pmatrix} 1 & 0 & V_x \\ 0 & 1 & V_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -V_x \\ 0 & 1 & -V_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & d_x \\ \sin(\theta) & \cos(\theta) & d_y \\ 0 & 0 & 1 \end{pmatrix}$$

where the overall translation components are

$$d_x = -\cos(\theta)V_x + \sin(\theta)V_y + V_x$$

$$d_y = -\sin(\theta)V_x - \cos(\theta)V_y + V_y$$

Because the same $\cos(\theta)$ and $\sin(\theta)$ terms appear in this result as in a rotation about the origin, we see that a rotation about an arbitrary point is equivalent to a rotation about the origin followed by a complicated translation through (d_x, d_y) .

As a specific example, we find the transformation that rotates points through 30° about $(-2, 3)$, and determine to which point the point $(1, 2)$ maps. A 30° rotation uses $\cos(\theta) = 0.866$ and $\sin(\theta) = 0.5$. The offset vector is then $(1.232, 1.402)$, and so the transformation applied to any point (P_x, P_y) is

$$Q_x = 0.866 P_x - 0.5 P_y + 1.232$$

$$Q_y = 0.5 P_x + 0.866 P_y + 1.402$$

Applying this to $(1, 2)$ yields $(1.098, 3.634)$. This is the correct result, as can be checked by sketching it on graph paper. (Do it!)

Example 5.2.6. Scaling and Shearing about arbitrary “pivot” points.

In a similar manner we often want to scale all points about some pivot point other than the origin. Because the elementary scaling operation of Equation 5.13 scales points about the origin, we do the same “shift-transform-unshift” sequence as for rotations. This and generalizing the shearing operation are explored in the exercises.

Example 5.2.7. Reflections about a tilted line.

Consider the line through the origin that makes an angle of β with the x -axis, as shown in Figure 5.18. Point A reflects into point B , and each house shown reflects into the other. We want to develop the transformation that reflects any point P about this axis, to produce point Q . Is this an affine transformation?

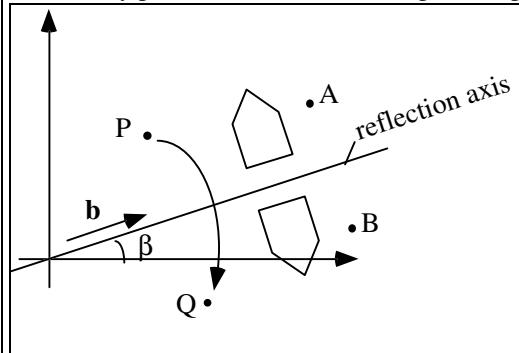


Figure 5.18. Reflecting about a tilted axis.

To show that it is affine, we build it out of three parts:

- A rotation through angle $-\beta$ (so the axis coincides with the x -axis);
- A reflection about the x -axis;
- A rotation back through β that “restores” the axis.

Each of these is represented by a matrix, so the overall transformation is given by the product of the three matrices, so it *is* affine. Check that each of the steps is properly represented in the following three matrices, and that the product is also correct:

$$\begin{pmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} c^2 - s^2 & -2cs & 0 \\ -2cs & s^2 - c^2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

where c stands for $\cos(\beta)$ and s for $\sin(\beta)$. Using trigonometric identities, the final matrix can be written (check this out!)

$$\begin{pmatrix} \cos(2\beta) & \sin(2\beta) & 0 \\ \sin(2\beta) & -\cos(2\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \{\text{a reflection about the axis at angle } \beta\} \quad (5.18)$$

This has the general look of a rotation matrix, except the angle has been doubled and minus signs have crept into the second column. But in fact it is the matrix for a reflection about the axis at angle β .

Practice Exercises.

Exercise 5.2.8. The classic: the Window to Viewport Transformation.

We developed this transformation in Chapter 3. Rewriting Equation 3.2 in the current notation we have:

$$\tilde{M} = \begin{pmatrix} A & 0 & C \\ 0 & B & D \\ 0 & 0 & 1 \end{pmatrix}$$

where the ingredients A , B , C , and D depend on the window and viewport and are given in Equation 3.3. Show that this transformation is composed of:

- A translation through $(-W.l, -W.b)$ to place the lower left corner of the window at the origin;
- A scaling by (A, B) to size things.
- A translation through $(V.l, V.b)$ to move the corner of the viewport to the desired position.

5.2.9. Alternative Form for a Rotation About a Point. Show that the transformation of Figure 5.17 can be written out as

$$Q_x = \cos(\theta)(P_x - V_x) - \sin(\theta)(P_y - V_y) + V_x$$

$$Q_y = \sin(\theta)(P_x - V_x) + \cos(\theta)(P_y - V_y) + V_y$$

This form clearly reveals that the point is first translated by $(-V_x, -V_y)$, rotated, and then translated by (V_x, V_y) .

5.2.10. Where does it end up? Where is the point $(8, 9)$ after it is rotated through 50° about the point $(3, 1)$? Find the M matrix.

5.2.11. Seeing it two ways. On graph paper place point $P = (4, 7)$ and the result Q of rotating P about $V = (5, 4)$ through 45° . Now rotate P about the origin through 45° to produce Q' , which is clearly different from Q . The difference between them is $V - VM$. Show the point $V - VM$ in the graph, and check that $Q - Q'$ equals $V - VM$.

5.2.12. What if the axis doesn't go through the origin? Find the affine transformation that produces a reflection about the line given parametrically by $L(t) = A + bt$. Show that it reduces to the result in Equation 5.20 when $A + bt$ does pass through the origin.

5.2.13. Reflection in $x = y$. Show that a reflection about the line $x = y$ is equivalent to a reflection in x followed by a 90° rotation.

5.2.14. Scaling About an Arbitrary Point. Fashion the affine transformation that scales points about a pivot point, (V_x, V_y) . Test the overall transformation on some sample points, to confirm that the scaling operation is correct. Compare this with the transformation for rotation about a pivot point.

5.2.15. Shearing Along a Tilted Axis. Fashion the transformation that shears a point along the axis described by vector \mathbf{u} tilted at angle θ , as shown in Figure 5.19. Point P is shifted along \mathbf{u} an amount that is fraction f of the displacement d of P from the axis.

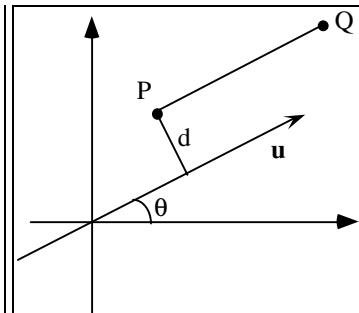


Figure 5.19. Shearing along a tilted axis.

5.2.16. Transforming Three Points. An affine transformation is completely determined by specifying what it does to three points. To illustrate this, find the affine transformation that converts triangle C with vertices $(-3, 3)$, $(0, 3)$, and $(0, 5)$ into equilateral triangle D with vertices $(0, 0)$, $(2, 0)$, and $(1, \sqrt{3})$, as shown in Figure 5.20.

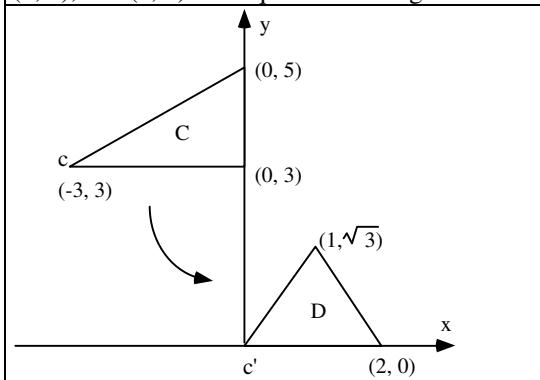


Figure 5.20. Converting one triangle into another.

Do this by a sequence of three elementary transformations:

1. Translate C down by 3 and right by 3 to place vertex c at c' .
2. Scale in x by $2/3$ and in y by $\sqrt{3}/2$ so C matches D in width and height.
3. Shear by $1/\sqrt{3}$ in the x -direction to align the top vertex of C with that of D .

Check that this transformation does in fact transform triangle C into triangle D . Also find the inverse of this transformation and show that it converts triangle D back into triangle C .

5.2.17. Fixed Points of an Affine Transformation. The point F is a *fixed point* of the affine transformation $T(p) = Mp$ if $T(F) = F$. That is, if F satisfies $F M = F$.

- a). Show that when the third column of M is $(0,0,1)$ - such that there is no translation, the origin is always a fixed point of T .
- b). Show that if F is a fixed point of T then for *any* P we have $T(P) = M(P - F) + F$.
- c). Show that F must always satisfy: $F = d(I - M)^{-1}$. Does every affine transformation have a fixed point?
- d). What is a fixed point for a rotation about point V ? Show that it satisfies the relationship in part b above.
- e). What is the fixed point for a scaling with scale factors S_x and S_y , about point V ?
- f). Consider the “5-th iterate” of $T()$ applied to P , given by $R = T(T(T(T(T(P))))$. (Recall IFS’s described at the end of Chapter 1). Use the result in b) to show a simple form for output R in terms of fixed point F of $T()$: $R = (P - F)M^5 + F$.

5.2.18. Finding Matrices. Give the explicit form of the 3-by-3 matrix representing each of the following transformations:

- a. Scaling by a factor of 2 in the x -direction and then rotating about $(2, 1)$.
- b. Scaling about $(2, 3)$ and following by translation through $(1, 1)$.
- c. Shearing in x by 30%, scaling by 2 in x , and then rotating about $(1, 1)$ through 30° .

5.2.19. Normalizing a Box. Find the affine transformation that maps the box with corners $(0, 0)$, $(2, 1)$, $(0, 5)$, and $(-2, 4)$ into the square with corners $(0, 0)$, $(1, 0)$, $(1, 1)$, and $(0, 1)$. Sketch the boxes.

5.2.20. Some Transformations Commute. Show that uniform scaling **commutes** with rotation, in that the resulting transformation does not depend on the order in which the individual transformations are applied. Show that two translations commute, as do two scalings. Show that differential scaling does not commute with rotation.

5.2.21. Reflection plus a rotation. Show that a reflection in x followed by a reflection in y is the same as a rotation by 180° .

5.2.22. Two Successive Rotations. Suppose that $R(\theta)$ denotes the transformation that produces a rotation through angle θ . Show that applying $R(\theta_1)$ followed by $R(\theta_2)$ is equivalent to applying the single rotation $R(\theta_1 + \theta_2)$. Thus successive rotations are additive.

5.2.23. A Succession of Shears. Find the composition of a pure shear along the x -axis followed by a pure shear along the y -axis. Is this still a shear? Sketch by hand an example of what happens to a square centered at the origin when subjected to a simultaneous shear versus a succession of shears along the two axes.

5.2.8. Some Useful Properties of Affine Transformations.

We have seen how to represent 2D affine transformations with matrices, how to compose complex transformations from a sequence of elementary ones, and the geometric effect of different 2D affine transformations. Before moving on to 3D transformations it is useful to summarize some general properties of affine transformations. These properties are easy to establish, and because no reference is made of the dimensionality of the objects being transformed, they apply equally well to 3D affine transformations. The only fact about 3D transformations we need at this point is that, like their 2D counterparts, they can be represented in homogeneous coordinates by a matrix.

1). Affine transformations *preserve* affine combinations of points.

We know that an affine combination of two points P_1 and P_2 is the point

$$W = a_1 P_1 + a_2 P_2, \quad \text{where } a_1 + a_2 = 1$$

What happens when we apply an affine transformation $T()$ to this point W ? We claim $T(W)$ is the *same* affine combination of the transformed points, that is:

$$\text{Claim: } T(a_1 P_1 + a_2 P_2) = a_1 T(P_1) + a_2 T(P_2), \quad (5.19)$$

For instance, $T(0.7 (2, 9) + 0.3 (1, 6)) = 0.7 T((2, 9)) + 0.3 T((1, 6))$.

The truth of this is simply a matter of linearity. Using homogeneous coordinates, the point $T(W)$ is $\tilde{M}\tilde{W}$, and we can do the following steps using linearity of matrix multiplication :

$$\tilde{M}\tilde{W} = \tilde{M}(a_1 \tilde{P}_1 + a_2 \tilde{P}_2) = a_1 \tilde{M}\tilde{P}_1 + a_2 \tilde{M}\tilde{P}_2$$

which in ordinary coordinates is just $a_1 T(P_1) + a_2 T(P_2)$ as claimed. The property that affine combinations of points are preserved under affine transformations seems fairly elementary and abstract, but it turns out to be pivotal. It is sometimes taken as the *definition* of what an affine transformation is.

2). Affine transformations *preserve* lines and planes.

Affine transformations preserve collinearity and “flatness”, and so the image of a straight line is another straight line. To see this, recall that the parametric representation $L(t)$ of a line through A and B is itself an affine combination of A and B :

$$L(t) = (1 - t) A + t B$$

This is an affine combination of points, so by the previous result the image of $L(t)$ is the same affine combination of the images of A and B :

$$Q(t) = (1 - t) T(A) + t T(B), \quad (5.20)$$

This is another straight line passing through $T(A)$ and $T(B)$. In computer graphics this vastly simplifies drawing transformed line segments: We need only compute the two transformed endpoints $T(A)$ and $T(B)$ and then draw a straight line between them! This saves having to transform *each* of the points along the line, which is obviously impossible.

The argument is the same to show that a plane is transformed into another plane. Recall from Equation 4.45 that the parametric representation for a plane can be written as an affine combination of points:

$$P(s, t) = sA + tB + (1 - s - t)C$$

When each point is transformed this becomes:

$$T(P(s, t)) = sT(A) + tT(B) + (1 - s - t)T(C)$$

which is clearly also the parametric representation of some plane.

Preservation of collinearity and “flatness” guarantees that polygons will transform into polygons, and planar polygons (those whose vertices all lie in a plane) will transform into planar polygons. In particular, triangles will transform into triangles.

3). Parallelism of lines and planes is preserved.

If two lines or planes are parallel, their images under an affine transformation are also parallel. This is easy to show. We first do it for lines. Take an arbitrary line $A + \mathbf{b}t$ having direction \mathbf{b} . It transforms to the line given in homogeneous coordinates by $\tilde{M}(\tilde{A} + \tilde{\mathbf{b}}t) = \tilde{M}\tilde{A} + (\tilde{M}\tilde{\mathbf{b}})t$ which has direction vector $\tilde{M}\tilde{\mathbf{b}}$. This new direction does *not* depend on point A . Thus two different lines $A_1 + \mathbf{b}t$ and $A_2 + \mathbf{b}t$ that have the same direction will transform into two lines both having the direction $\tilde{M}\tilde{\mathbf{b}}$, so they are parallel. An important consequence of this property is that *parallelograms map into other parallelograms*.

The same argument applies to planes: its direction vectors (see Equation 4.43) transform into new direction vectors whose values do not depend on the location of the plane. A consequence of this is that parallelepipeds⁵ map into other parallelepipeds.

Example 5.2.8. How is a grid transformed?

Because affine transformations map parallelograms into parallelograms they are rather limited in how much they can alter the shape of geometrical objects. To illustrate this apply any 2D affine transformation T to a unit square grid, as in Figure 5.21. Because a grid consists of two sets of parallel lines, T maps the square grid to another grid consisting of two sets of parallel lines. Think of the grid “carrying along” whatever objects are defined in the grid, to get an idea of how the objects are warped by the transformation. This is all that an affine transformation can do: warp figures in the same way that one grid is mapped into another. The new lines can be tilted at any angle; they can be any (fixed) distance apart; and the two new axes need not be perpendicular. And of course the whole grid can be positioned anywhere in the plane.

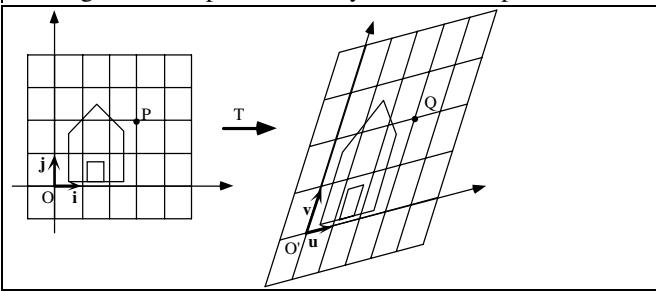


Figure 5.21. A transformed grid.

⁵ As we see later, a parallelepiped is the 3D analog of a parallelogram: it has six sides that occur in pairs of parallel faces.

The same result applies in 3D: all a 3D affine transformation can do is map a cubical grid into a grid of parallelipeds.

4). The Columns of the Matrix reveal the Transformed Coordinate Frame.

It is useful to examine the columns of the matrix M of an affine transformation, for they prescribe how the coordinate frame is transformed. Suppose the matrix M is given by

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} = (\mathbf{m}_1 : \mathbf{m}_2 : m_3) \quad (5.21)$$

so its columns are \mathbf{m}_1 , \mathbf{m}_2 , and m_3 . The first two columns are vectors (their third component is 0) and the last column is a point (its third component is a 1). As always the coordinate frame of interest is defined by the origin ϑ , and the basis vectors \mathbf{i} and \mathbf{j} , which have representations:

$$\vartheta = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \mathbf{i} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \text{and } \mathbf{j} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Notice that vector \mathbf{i} transforms into the vector \mathbf{m}_1 (check this out):

$$\mathbf{m}_1 = M\mathbf{i}$$

and similarly \mathbf{j} maps into \mathbf{m}_2 and ϑ maps into the point m_3 . This is illustrated in Figure 5.22a. The coordinate frame $(\mathbf{i}, \mathbf{j}, \vartheta)$ transforms into the coordinate frame $(\mathbf{m}_1, \mathbf{m}_2, m_3)$, and these new objects are precisely the columns of the matrix.

a).

b).

Figure 5.22. The transformation forms a new coordinate frame.

The axes of the new coordinate frame are not necessarily perpendicular, nor must they be unit length. (They are still perpendicular if the transformation involves only rotations and uniform scalings.) Any point $P = P_x\mathbf{i} + P_y\mathbf{j} + \vartheta$ transforms into $Q = P_x\mathbf{m}_1 + P_y\mathbf{m}_2 + m_3$. It is sometimes very revealing to look at the matrix of an affine transformation in this way.

Example 5.2.9. Rotation about a point. The transformation explored in Example 5.2.5 is a rotation of 30° about the point $(-2, 3)$. This yielded the matrix:

$$\begin{pmatrix} .866 & -.5 & 1.232 \\ .5 & .866 & 1.402 \\ 0 & 0 & 1 \end{pmatrix}$$

As shown in Figure 5.22b the coordinate frame therefore maps into the new coordinate frame with origin at $(1.232, 1.402, 1)$ and coordinate axes given by the vectors $(0.866, 0.5, 0)$ and $(-0.5, 0.866, 0)$. Note that these axes are still perpendicular, since only a rotation is involved.

5). Relative Ratios Are Preserved.

Affine transformations have yet another useful property. Consider a point P that lies at the fraction t of the way between two given points, A and B , as shown in Figure 5.23. Apply affine transformation $T()$ to A , B , and P . We claim the transformed point, $T(P)$, also lies the *same* fraction t of the way between the images $T(A)$ and $T(B)$. This is not hard to show (see the exercises).

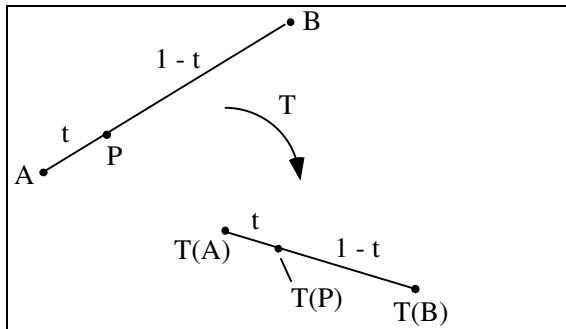


Figure 5.23. Relative ratios are preserved.

As a special case, midpoints of lines map into midpoints. This result pops out a nice geometric result: the diagonals of any parallelogram bisect each other. (Proof: any parallelogram is an affine-transformed square (why?), and the diagonals of a square bisect each other, so the diagonals of a parallelogram also bisect each other.) The same applies in 3D space: the diagonals of any parallelepiped bisect each other.

Interesting Aside. In addition to preserving lines, parallelism, and relative ratios, affine transformations also preserve ellipses and ellipsoids, as we see in Chapter 8!

6). Effect of Transformations on the Areas of Figures.

In CAD applications it is often important to compute the area or volume of an object. For instance, how is the area of a polygon affected when all of its vertices are subjected to an affine transformation? It is clear geometrically that neither translations nor rotations have any effect on the area of a figure, but scalings certainly do, and shearing might.

The result is simple, and is developed in the exercises: When the 2D transformation with matrix M is applied to an object, its area is multiplied by the *magnitude of the determinant* of M :

$$\frac{\text{area after transformation}}{\text{area before transformation}} = |\det M| \quad (5.22)$$

In 2D the determinant of M in Equation 5.6 is $m_{11}m_{22} - m_{12}m_{21}$.⁶ Thus for a pure scaling as in Equation 5.10, the new area is S_xS_y times the original area, whereas for a shear along one axis the new area is the same as the original area! Equation 5.21 also confirms that a rotation does not alter the area of a figure, since $\cos^2(\theta) + \sin^2(\theta) = 1$.

In 3D similar arguments apply, and we can conclude that the volume of a 3D object is scaled by $|\det M|$ when the object is transformed by the 3D transformation based on matrix M .

Example 5.2.9: The Area of an Ellipse. What is the area of the ellipse that fits inside a rectangle with width W and height H ? **Solution:** This ellipse can be formed by scaling the unit circle $x^2 + y^2 = 1$ by the scale factors $S_x = W$ and $S_y = H$, a transformation for which the matrix M has determinant WH . The unit circle is known to have area π , and so the ellipse has area πWH .

7). Every Affine Transformation is Composed of Elementary Operations.

We can construct complex affine transformations by composing a number of elementary ones. It is interesting to turn the question around and ask, what elementary operations “reside in” a given affine transformation?

⁶ The determinant of the homogeneous coordinate version is the same (why?)

Basically a matrix \tilde{M} may be factored into a product of elementary matrices in various ways. One particular way of factoring the matrix \tilde{M} associated with a 2D affine transformation, elaborated upon in Case Study 5.3, yields the result:

$$\tilde{M} = (\text{shear}) (\text{scaling}) (\text{rotation}) (\text{translation})$$

That is, any 3 by 3 matrix \tilde{M} that represents a 2D affine transformation can be written as the product of (reading right to left) a translation matrix, a rotation matrix, a scaling matrix, and a shear matrix. The specific ingredients of each matrix are given in the Case Study.

In 3D things are somewhat more complicated. The 4 by 4 matrix \tilde{M} that represents a 3D affine transformation can be written as:

$$\tilde{M} = (\text{scaling}) (\text{rotation}) (\text{shear}_1) (\text{shear}_2) (\text{translation}),$$

the product of (reading right to left) a translation matrix, a shear matrix, another shear matrix, a rotation matrix, and a scaling matrix. This result is developed in Case study 5.???

Practice Exercises.

5.2.24 Generalizing the argument. Show that if W is an affine combination of the N points P_i , $i = 1, \dots, N$, and $T()$ is an affine transformation, then $T(W)$ is the same affine combination of the N points $T(P_i)$, $i = 1, \dots, N$.

5.2.25. Show that relative ratios are preserved. Consider P given by $A + \mathbf{b}t$ where $\mathbf{b} = B - A$. Find the distances $|P - A|$ and $|P - B|$ from P to A and B respectively, showing that they lie in the ratio t to $1 - t$. Is this true if t lies outside of the range 0 to 1? Do the same for the distances $|T(P) - T(A)|$ and $|T(P) - T(B)|$.

5.2.26. Effect on Area. Show that a 2D affine transformation causes the area of a figure to be multiplied by the factor given in Equation 5.27. Hint: View a geometric figure as made up of many very small squares, each of which is mapped into a parallelogram, and then find the area of this parallelogram.

5.3. 3D Affine Transformations.

The same ideas apply to 3D affine transformations as apply to 2D affine transformations, but of course the expressions are more complicated, and it is considerably harder to visualize the effect of a 3D transformation.

Again we use coordinate frames, and suppose that we have an origin ϑ and three mutually perpendicular axes in the directions \mathbf{i} , \mathbf{j} , and \mathbf{k} (see Figure 5.8). Point P in this frame is given by $P = \vartheta + P_x\mathbf{i} + P_y\mathbf{j} + P_z\mathbf{k}$, and so has as the representation

$$\tilde{P} = \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Suppose $T()$ is an affine transformation that transforms point P to point Q . Then just as in the 2D case $T()$ is represented by a matrix \tilde{M} which is now 4 by 4:

$$\tilde{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.23)$$

and we can say that the representation of point Q is found by multiplying P by matrix \tilde{M} :

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \tilde{M} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} \quad (5.24)$$

Notice that once again for an affine transformation the final row of the matrix is a string of zeroes followed a lone one. (This will cease to be the case when we examine projective matrices in Chapter 7.)

5.3.1. The Elementary 3D Transformations.

We consider the nature of elementary 3D transformations individually, and then compose them into general 3D affine transformations.

Translation.

For a pure translation, the matrix \tilde{M} has the simple form.

$$\begin{pmatrix} 1 & 0 & 0 & m_{14} \\ 0 & 1 & 0 & m_{24} \\ 0 & 0 & 1 & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Check that $Q = MP$ is simply a shift in Q by the vector $\mathbf{m} = (m_{14}, m_{24}, m_{34})$.

Scaling.

Scaling in three dimensions is a direct extension of the 2D case, having a matrix given by:

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.25)$$

where the three constants S_x , S_y , and S_z cause scaling of the corresponding coordinates. Scaling is about the origin, just as in the 2D case. Figure 5.24 shows the effect of scaling in the z -direction by 0.5 and in the x -direction by a factor of two.

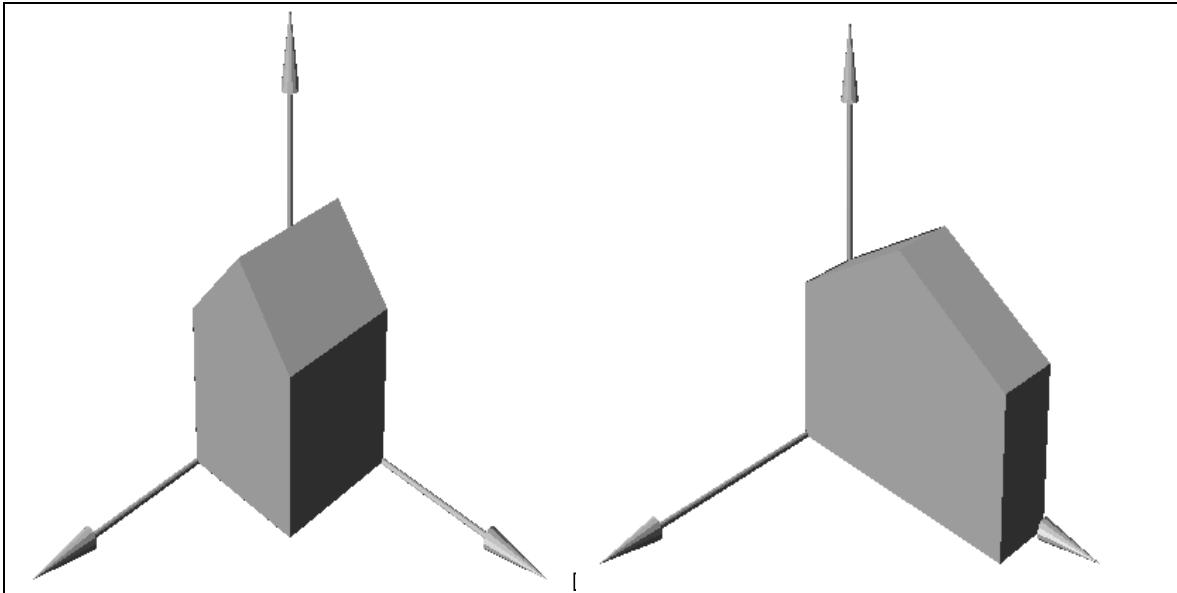


Figure 5.24. Scaling the basic barn.

Notice that this figure shows various lines before and after being transformed. It capitalizes on the important fact that straight lines transform to straight lines.

Shearing.

Three-dimensional shears appear in greater variety than do their two-dimensional counterparts. The matrix for the simplest elementary shear is the identity matrix with one zero term replaced by some value, as in

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ f & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.26)$$

which produces $Q = (P_x, fP_x + P_y, P_z)$; that is, P_y is offset by some amount proportional to P_x , and the other components are unchanged. This causes an effect similar to that in 2D shown in Figure 5.15. Goldman [goldman??] has developed a much more general form for a 3D shear, which is described in Case Study 5.???

Rotations.

Rotations in three dimensions are common in graphics, for we often want to rotate an object or a camera in order to obtain different views. There is a much greater variety of rotations in three than in two dimensions, since we must specify an axis about which the rotation occurs, rather than just a single point. One helpful approach is to decompose a rotation into a combination of simpler ones.

Elementary rotations about a coordinate axis.

The simplest rotation is a rotation about one of the coordinate axes. We call a rotation about the x -axis an “ x -roll”, a rotation about the y -axis a “ y -roll”, and one about the z -axis a “ z -roll”. We present individually the matrices that produce an x -roll, a y -roll, and a z -roll. In each case the rotation is through an angle, β , about the given axis. We define positive angles using a “looking inward” convention:

Positive values of β cause a counterclockwise (CCW) rotation about an axis as one looks inward from a point on the positive axis toward the origin.

The three basic positive rotations are illustrated in Figure 5.25.⁷

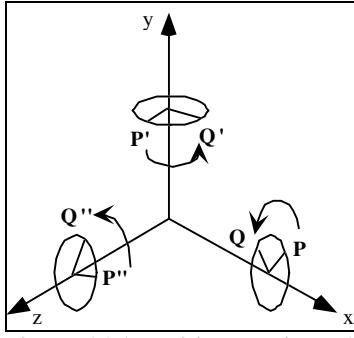


Figure 5.25. Positive rotations about the three axes.

This formulation is also consistent with our notion of 2D rotations: a positive rotation in two dimensions is equivalent to a z -roll as we look at the xy -plane from a point on the positive z -axis.

Notice what happens with this convention for the particular case of a 90° rotation:

- For a z -roll, the x -axis rotates to the y -axis.
- For an x -roll, the y -axis rotates to the z -axis.
- For a y -roll, the z -axis rotates to the x -axis.

The following three matrices represent transformations that rotate points through angle β about a coordinate axis. We use the suggestive notation $R_x(\cdot)$, $R_y(\cdot)$, and $R_z(\cdot)$ to denote x -, y -, and z -rolls, respectively. The parameter is the angle through which points are rotated, given in radians, and c stands for $\cos(\beta)$ and s for $\sin(\beta)$.

1. An x -roll:

$$R_x(\beta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.27)$$

2. A y -roll:

$$R_y(\beta) = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.28)$$

3. A z -roll:

⁷In a left-handed system the sense of a rotation through a positive β would be CCW looking **outward** along the positive axis from the origin. This formulation is used by some authors.

$$R_z(\beta) = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.29)$$

Note that 12 of the terms in each matrix are the zeros and ones of the identity matrix. They occur in the row and column that correspond to the axis about which the rotation is being made (e.g., the first row and column for a x -roll). They guarantee that the corresponding coordinate of the point being transformed will not be altered. The c and s terms always appear in a rectangular pattern in the other rows and columns.

Aside: Why is the y-roll different? The $-s$ term appears in the lower row for the x - and z -rolls, but in the upper row for the y -roll. Is a y -roll inherently different in some way? This question is explored in the exercises.

Example 5.3.1. Rotating the barn. Figure 5.26 shows a “barn” in its original orientation (part a), and after a -70° x -roll (part b), a 30° y -roll (part c), and a -90° z -roll (part d).

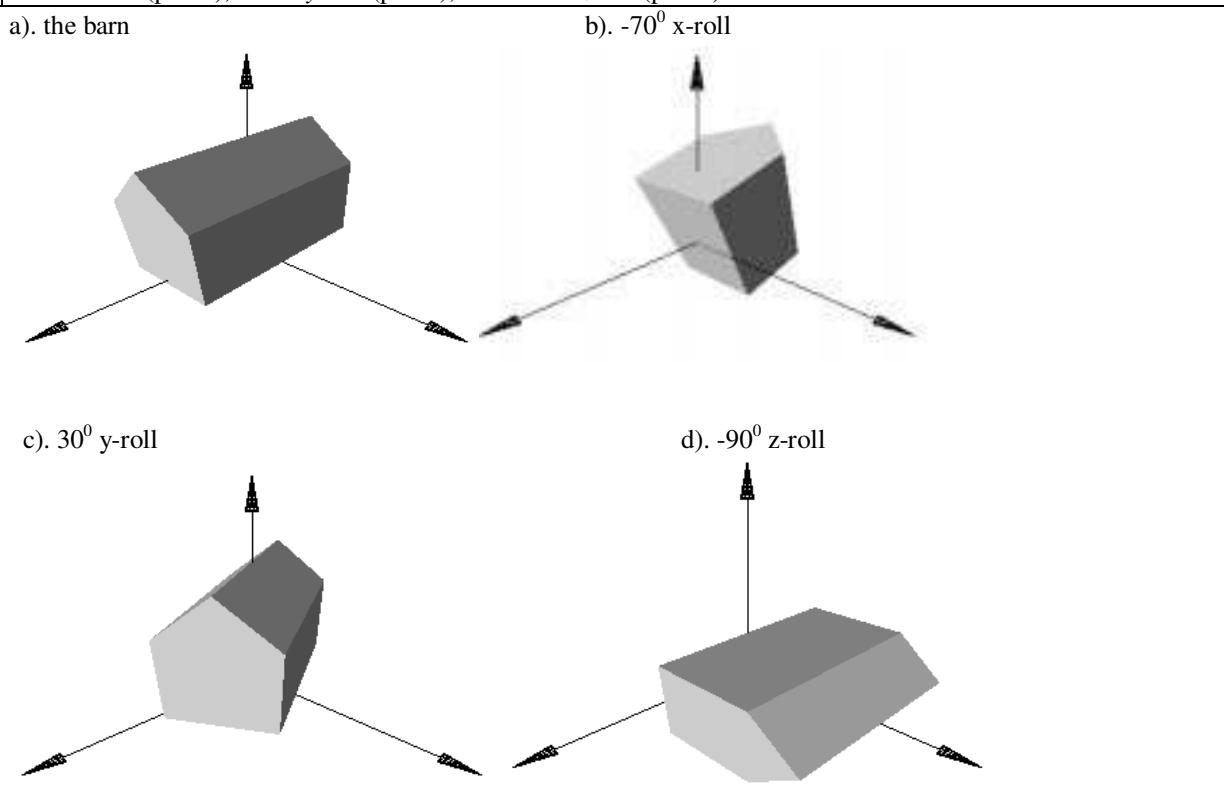


Figure 5.26. Rotating the basic barn.

Example 5.3.2. Rotate the point $P = (3, 1, 4)$ through 30° about the y -axis. **Solution:** Using Equation 9.9.10 with $c = .866$ and $s = .5$, P is transformed into

$$Q = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 4.6 \\ 1 \\ 1.964 \\ 1 \end{pmatrix}$$

As expected, the y -coordinate of the point is not altered.

Practice Exercises.

5.3.1. Visualizing the 90° Rotations. Draw a right-handed 3D system and convince yourself that a 90° rotation (CCW looking toward the origin) about each axis rotates the other axes into one another, as specified in the preceding list. What is the effect of rotating a point on the x -axis about the x -axis?

5.3.2. Rotating the Basic Barn. Sketch the basic barn after each vertex has experienced a 45° x -roll. Repeat for y - and z -rolls.

5.3.3. Do a Rotation. Find the image Q of the point $P = (1, 2, -1)$ after a 45° y -roll. Sketch P and Q in a 3D coordinate system and show that your result is reasonable.

5.3.4. Testing 90° Rotations of the Axes. This exercise provides a useful trick for remembering the form of the rotation matrices. Apply each of the three rotation matrices to each of the standard unit position vectors, \mathbf{i} , \mathbf{j} , and \mathbf{k} , using a 90° rotation. In each case discuss the effect of the transformation on the unit vector.

5.3.5. Is a y-roll indeed different? The minus sign in Equation 5.28 seems to be in the wrong place: on the lower s rather than the upper one. Here you show that Equations 5.27-29 are in fact consistent. It's just a matter of how things are ordered. Think of the three axes x , y , and z as occurring cyclically: $x \rightarrow y \rightarrow z \rightarrow x \rightarrow y \dots$, etc. If we are discussing a rotation about some "current" axis (x -, y -, or z -) then we can identify the "previous" axis and the "next" axis. For instance, if x - is the current axis, then the previous one is z - and the next is y - . Show that with this naming all three types of rotations use the same equations: $Q_{\text{curr}} = P_{\text{curr}}$, $Q_{\text{next}} = c P_{\text{next}} - s P_{\text{prev}}$, and $Q_{\text{prev}} = s P_{\text{next}} + c P_{\text{prev}}$. Write these equations out for each of the three possible "current" axes.

5.3.2. Composing 3D Affine Transformations.

Not surprisingly, 3D affine transformations can be composed, and the result is another 3D affine transformation. The thinking is exactly parallel to that which led to Equation 5.17 in the 2D case. The matrix that represents the overall transformation is the product of the individual matrices M_1 and M_2 that perform the two transformations, with M_2 premultiplying M_1 :

$$\tilde{M} = \tilde{M}_2 \tilde{M}_1. \quad (5.30)$$

Any number of affine transformations can be composed in this way, and a single matrix results that represents the overall transformation.

Figure 5.27 shows an example, where a barn is first transformed using some M_1 , then that transformed barn is again transformed using M_2 . The result is the same as the barn transformed once using M_2M_1 .

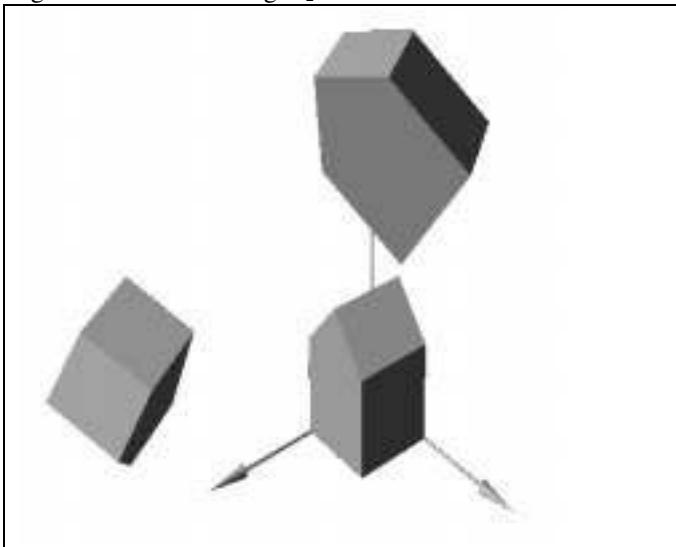


Figure 5.27. Composing 3D affine transformations. (file: fig5.27.bmp)

5.3.3. Combining Rotations.

Results! Why, man, I have gotten a lot of results.

I know several thousand things that won't work.

Thomas A. Edison

One of the most important distinctions between 2D and 3D transformations is the manner in which rotations combine. In 2D two rotations, say $R(\beta_1)$ and $R(\beta_2)$, combine to produce $R(\beta_1 + \beta_2)$, and the order in which they are combined makes no difference. In 3D the situation is much more complicated, because rotations can be about different axes. The order in which two rotations about different axes are performed *does* matter: 3D rotation matrices do **not** commute. We explore some properties of 3D rotations here, investigating different ways that a rotation can be represented, and see how to create rotations that do a certain job.

It's very common to build a rotation in 3D by composing three elementary rotations: an x -roll followed by a y -roll, and then a z -roll. Using the notation of Equations 5.27-29 for each individual roll, the overall rotation is given by

$$M = R_x(\beta_x)R_y(\beta_y)R_z(\beta_z) \quad (5.31)$$

In this context the angles β_1 , β_2 , and β_3 are often called **Euler⁸ angles**. One form of **Euler's Theorem** asserts that *any* 3D rotation can be obtained by three rolls about the x -, y -, and z -axes, so any rotation can be written as in Equation 5.32 for the appropriate choice of Euler angles. This implies that it takes three values to completely specify a rotation.

Example 5.3.3. What is the matrix associated with an x -roll of 45^0 followed by a y -roll of 30^0 followed by a z -roll of 60^0 ? Direct multiplication of the three component matrices (in the proper "reverse" order) yields:

$$\begin{pmatrix} .5 & -.866 & 0 & 0 \\ .866 & .5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} .866 & 0 & .5 & .0 \\ 0 & 1 & 0 & 0 \\ -.5 & 0 & .866 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & .707 & -.707 & 0 \\ 0 & .707 & .707 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} .433 & -.436 & .789 & 0 \\ .75 & .66 & -.047 & 0 \\ -.5 & .612 & .612 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Some people use a different ordering of "rolls" to create a complicated rotation. For instance, they might express a rotation as $R_y(\beta_1)R_z(\beta_2)R_x(\beta_3)$: first a y -roll then a z -roll then an x -roll. Because rotations in 3D do not commute this requires the use of different Euler angles β_1 , β_2 , and β_3 to create the same rotation as in Equation 5.32. There are 12 possible orderings of the three individual rolls, and each uses different values for β_1 , β_2 , and β_3 .

Rotations About an Arbitrary Axis.

When using Euler angles we perform a sequence of x -, y -, and z -rolls, that is, rotations about a coordinate axis. But it can be much easier to work with rotations if we have a way to rotate about an axis that points in an arbitrary direction. Visualize the earth, or a toy top, spinning about a tilted axis. In fact, Euler's theorem states that every rotation can be represented as one of this type:

Euler's Theorem: *Any rotation (or sequence of rotations) about a point is equivalent to a single rotation about some axis through that point.⁹*

What is the matrix for such a rotation, and can we work with it conveniently?

Figure 5.28 shows an axis represented by vector \mathbf{u} , and an arbitrary point P that is to be rotated through angle β about \mathbf{u} to produce point Q .

⁸ Leonhard Euler, 1707-1783, a Swiss mathematician of extraordinary ability who made important contributions to all branches of mathematics.

⁹ This is sometimes stated: Given two rectangular coordinate systems with the same origin and arbitrary directions of axes, one can always specify a line through the origin such that one coordinate system goes into the other by a rotation about this line. [gellert75]

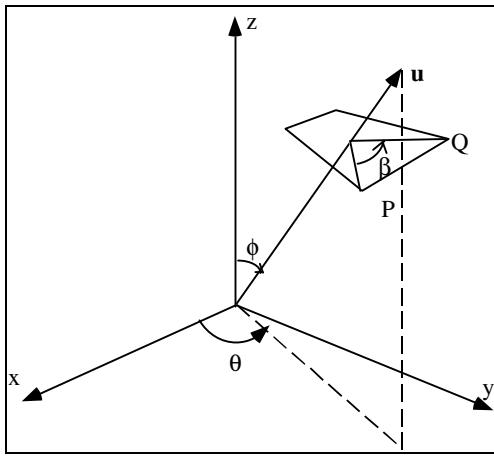


Figure 5.28. Rotation about an axis through the origin.

Because \mathbf{u} can have any direction, it would seem at first glance to be very difficult to find a single matrix that represents such a rotation. But in fact it can be found in two rather different ways, a classic way and a constructive way.

1). The classic way. Decompose the required rotation into a sequence of known steps:

1. Perform two rotations so that \mathbf{u} becomes aligned with the z -axis.
2. Do a z -roll through angle β .
3. Undo the two alignment rotations to restore \mathbf{u} to its original direction.

This is reminiscent of rotating about a point in two dimensions: The first step prepares the situation for a simpler known operation; the simple operation is done; and finally the preparation step is undone. The result (discussed in the exercises) is that the transformation requires the multiplication of five matrices:

$$R_{\mathbf{u}}(\beta) = R_Z(-\theta) R_Y(-\phi) R_Z(\beta) R_Y(\phi) R_Z(\theta) \quad (5.32)$$

each being a rotation about one of the coordinate axes. This is tedious to do by hand but is straightforward to carry out in a program. However, expanding out the product gives little insight into how the ingredients go together.

2). The constructive way. Using some vector tools we can obtain a more revealing expression for the matrix $R_{\mathbf{u}}(\beta)$. This approach has become popular recently, and versions of it are described by several authors in GEMS I [glass90]. We adapt the derivation of Maillot [mail90].

Figure 5.29 shows the axis of rotation \mathbf{u} , and we wish to express the operation of rotating point P through angle β into point Q . The method, spelled out in Case Study 5.5, effectively establishes a 2D coordinate system in the plane of rotation as shown. This defines two orthogonal vectors \mathbf{a} and \mathbf{b} lying in the plane, and as shown in Figure 5.29b point Q is expressed as a linear combination of them. The expression for Q involves dot products and cross products of various ingredients in the problem. But because each of the terms is linear in the coordinates of P , it can be rewritten as P times a matrix.

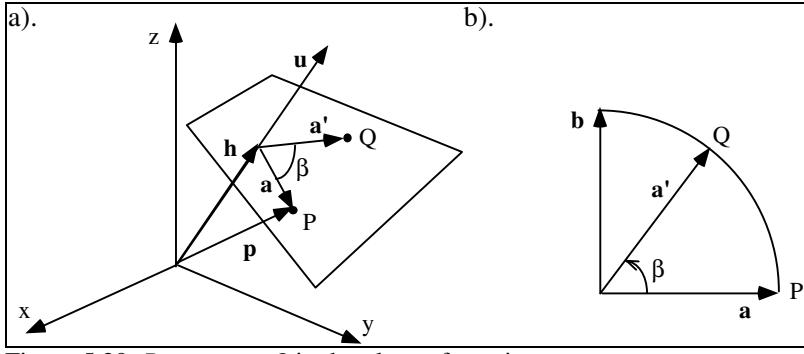


Figure 5.29. P rotates to Q in the plane of rotation.

The final result is the matrix:

$$R_u(\beta) = \begin{pmatrix} c + (1-c)u_x^2 & (1-c)u_yu_x - su_z & (1-c)u_zu_x + su_y & 0 \\ (1-c)u_xu_y + su_z & c + (1-c)u_y^2 & (1-c)u_zu_y - su_x & 0 \\ (1-c)u_xu_z - su_y & (1-c)u_yu_z + su_x & c + (1-c)u_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.33)$$

where $c = \cos(\beta)$, and $s = \sin(\beta)$, and (u_x, u_y, u_z) are the components of the unit vector \mathbf{u} . This looks more complicated than it is. In fact, as we see later, there is so much structure in the terms that, given an arbitrary rotation matrix, we can find the specific axis and angle that produces the rotation (which proves Euler's theorem).

As we see later, OpenGL provides a function to create a rotation about an arbitrary axis:

```
glRotated(angle, ux, uy, uz);
```

Example 5.3.4. Rotating about an axis. Find the matrix that produces a rotation through 45° about the axis $\mathbf{u} = (1,1,1)/\sqrt{3} = (0.577, 0.577, 0.577)$. **Solution:** For a 45° rotation, $c = s = 0.707$, and filling in the terms in Equation 5.33 we obtain:

$$R_u(45^\circ) = \begin{pmatrix} .8047 & -.31 & .5058 & 0 \\ .5058 & .8047 & -.31 & 0 \\ -.31 & .5058 & .8047 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This has a determinant of 1 as expected. Figure 5.30 shows the basic barn, shifted away from the origin, before it is rotated (dark), after a rotation through 22.5° (medium), and after a rotation of 45° (light).

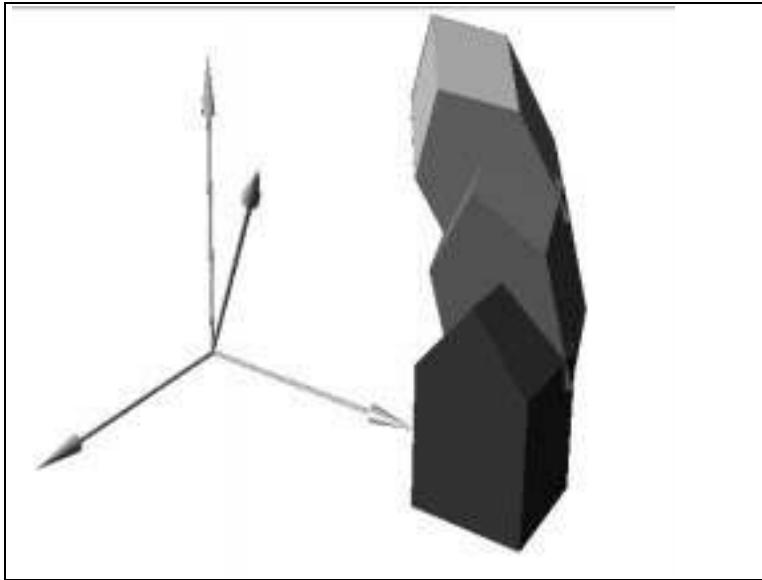


Figure 5.30. The basic barn rotated about axis \mathbf{u} .

Finding the Axis and Angle of Rotation.

Euler's theorem guarantees that any rotation is equivalent to a rotation about some axis. It is useful, when presented with some rotation matrix, to determine the specific axis and angle. That is, given values m_{ij} for the matrix:

$$R_u(\beta) = \begin{pmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

extract the angle β and the unit vector \mathbf{u} .

This is surprisingly easy to do by examining Equation 5.33[watt92]. First note that the trace of $R_u(\beta)$, that is the sum of the three diagonal elements, is $3c + (1 - c)(u_x^2 + u_y^2 + u_z^2) = 1 + 2\cos(\beta)$. So we can solve for $\cos(\beta)$ directly:

$$\cos(\beta) = \frac{1}{2}(m_{11} + m_{22} + m_{33} - 1).$$

Take the arc cosine of this value to obtain β , and use it to find $s = \sin(\beta)$ as well. Now see that pairs of elements of the matrix combine to reveal the individual components of \mathbf{u} :

$$\begin{aligned} u_x &= \frac{m_{32} - m_{23}}{2\sin(\beta)} \\ u_y &= \frac{m_{13} - m_{31}}{2\sin(\beta)} \\ u_z &= \frac{m_{21} - m_{12}}{2\sin(\beta)} \end{aligned} \tag{5.34}$$

Example 5.3.5. Find the axis and angle. Pretend you don't know the underlying axis and angle for the rotation matrix in Example 5.3.3, and solve for it. The trace is 2.414, so $\cos(\beta) = 0.707$, β must be 45° , and $\sin(\beta) = 0.707$. Now calculate each of the terms in Equation 5.35: they all yield the value 0.577, so $\mathbf{u} = (1, 1, 1)/\sqrt{3}$, just as we expected.

Practice Exercises.

5.3.6. Which ones commute? Consider two affine transformations T_1 and T_2 . Is T_1T_2 the same as T_2T_1 when:

- They are both pure translations?
- They are both scalings?
- They are both shears?
- One is a rotation and one a translation?
- One is a rotation and one is a scaling?
- One is a scaling and one is a shear?

5.3.7. Special cases of rotation about a general axis \mathbf{u} . It always helps to see that a complicated result collapses to a familiar one in special cases. Check that this happens in Equation 5.34 when \mathbf{u} is itself

- the x -axis, \mathbf{i} ;
- the y -axis, \mathbf{j} ;
- the z -axis, \mathbf{k} .

5.3.8. Classic Approach to Rotation about an Axis. Here we suggest how to find the rotations that cause \mathbf{u} to become aligned with the z -axis. (See Appendix 2 for a review of spherical coordinates.) Suppose the direction of \mathbf{u} is given by the spherical coordinate angles ϕ and θ as indicated in Figure 5.27. Align \mathbf{u} with the z -axis by a z -roll through $-\theta$: this swings \mathbf{u} into the xz -plane to form the new axis, \mathbf{u}' (sketch this). Use Equation 5.29 to obtain $R_Z(-\theta)$. Second, a y -roll through $-\phi$ completes the alignment process. With \mathbf{u} aligned along the z -axis, do the desired z -roll through angle β , using Equation 5.29. Finally, the alignment rotations must be undone to restore the axis to its original direction. Use the inverse matrices to $R_Y(-\phi)$ and $R_Z(-\theta)$, which are $R_Y(\phi)$ and $R_Z(\theta)$, respectively. First undo the y -roll and then the z -roll. Finally, multiply these five elementary rotations to obtain Equation 5.34. Work out the details, and apply them to find the matrix M that performs a rotation through angle 35° about the axis situated at $\theta=30^\circ$ and $\phi=45^\circ$. Show that the final result is:

$$\tilde{M} = \begin{pmatrix} .877 & -.366 & .281 & 0 \\ .445 & .842 & -.306 & 0 \\ -.124 & .396 & .910 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5.3.9. Orthogonal Matrices. A matrix is **orthogonal** if its columns are mutually orthogonal unit-length vectors. Show that each of the three rotation matrices given in Equations 5.27-29 is orthogonal. What is the determinant of an orthogonal matrix? An orthogonal matrix has a splendid property: *Its inverse is identical to its transpose* (also see Appendix 2). Show why the orthogonality of the columns guarantees this. Find the inverse of each of the three rotation matrices above, and show that the inverse of a rotation is simply a rotation in the opposite direction.

5.3.10. The Matrix Is Orthogonal. Show that the complicated rotation matrix in Equation 5.34 is orthogonal.

5.3.11. Structure of a rotation matrix. Show that for a 3×3 rotation M the three rows are pair-wise orthogonal, and the third is the cross product of first two.

5.3.12. What if the axis of rotation does not pass through the origin? If the axis does not pass through the origin but instead is given by $S + \mathbf{u}r$ for some point S , then we must first translate to the origin through $-S$, apply the appropriate rotation, and then translate back through S . Derive the overall matrix that results.

5.3.4. Summary of Properties of 3D Affine Transformations.

The properties noted for affine transformations in Section 5.2.8 apply, of course, to 3D affine transformations. Stated in terms of any 3D affine transformation $T(\cdot)$ having matrix M , they are:

- **Affine transformations preserve affine combinations of points.** If $a + b = 1$ then $aP + bQ$ is a meaningful 3D point, and $T(aP + bQ) = aT(P) + bT(Q)$.
- **Affine transformations preserve lines and planes.** Straightness is preserved: The image $T(L)$ of a line L in 3D space is another straight line; the image $T(W)$ of a plane W in 3D space is another plane.
- **Parallelism of lines and planes is preserved.** If W and Z are parallel lines (or planes), then $T(W)$ and $T(Z)$ are also parallel.

- **The Columns of the Matrix reveal the Transformed Coordinate Frame.** If the columns of M are the vectors $\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3$, and the point m_4 , the transformation maps the frame $(\mathbf{i}, \mathbf{j}, \mathbf{k}, \vartheta)$ to the frame $(\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, m_4)$.
- **Relative Ratios Are Preserved.** If P is fraction f of the way from point A to point B , then $T(P)$ is also fraction f of the way from point $T(A)$ to $T(B)$.
- **Effect of Transformations on the Areas of Figures.** If 3D object D has volume V , then its image $T(D)$ has volume $|det M| V$, where $|det M|$ is the absolute value of the determinant of M .
- **Every Affine Transformation is Composed of Elementary Operations.** A 3D affine transformation may be decomposed into a composition of elementary transformations. This can be done in several ways.

5.4. Changing Coordinate Systems.

There's another way to think about affine transformations. In many respect it is a more natural approach when modeling a scene. Instead of viewing an affine transformation as producing a different point in a fixed coordinate system, you think of it as producing a new coordinate system in which to represent points.

Aside: a word on notation: To make things fit better on the printed page, we shall sometimes use the notation

$(P_x, P_y, 1)^T$ in place of $\begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$. (Also see Appendix 2.) The superscript T denotes the **transpose**, so we are simply writing the column vector as a transposed row vector.

Suppose we have a 2D coordinate frame #1 as shown in Figure 5.31, with origin ϑ and axes \mathbf{i} and \mathbf{j} . Further suppose we have an affine transformation $T(\cdot)$ represented by matrix M . So $T(\cdot)$ transforms coordinate frame #1 into coordinate frame #2, with new origin $\vartheta' = T(\vartheta)$, and new axes $\mathbf{i}' = T(\mathbf{i})$ and $\mathbf{j}' = T(\mathbf{j})$.

Figure 5.31. Transforming a coordinate frame.

Now let P be a point with representation $(c, d, 1)^T$ in the new system #2. What are the values of a and b in its representation $(a, b, 1)^T$ in the original system #1? The answer: just multiply $(c, d, 1)^T$ by M :

$$\begin{pmatrix} a \\ b \\ 1 \end{pmatrix} = M \begin{pmatrix} c \\ d \\ 1 \end{pmatrix} \quad (5.35)$$

Summarizing: Suppose coordinate system #2 is formed from coordinate system #1 by the affine transformation M . Further suppose that point $P = (P_x, P_y, P_z, 1)$ are the coordinates of a point P expressed in system #2. Then the coordinates of P expressed in system #1 are MP .

This may seem obvious to some readers, but in case it doesn't, a derivation is developed in the exercises. This result also holds for 3D systems, of course, and we use it extensively when calculating how 3D points are transformed as they are passed down the graphics pipeline.

Example 5.4.1. Rotating a coordinate system. Consider again the transformation of Example 5.2.5 that rotates points through 30° about the point $(-2, 3)$. (See Figure 5.22.) This transformation maps the origin ϑ and axes \mathbf{i} and \mathbf{j} into the system #2 as shown in that figure. Now consider the point P with coordinates $(P_x, P_y, 1)^T$ in the *new* coordinate system. What are the coordinates of this point expressed in the *original* system #1? The answer is simply MP . For instance, $(1, 2, 1)^T$ in the new system lies at $M(1, 2, 1)^T = (1.098, 3.634, 1)^T$ in the original system. (Sketch this in the figure.) Notice that the point $(-2, 3, 1)^T$, the center of rotation of the transformation, is

a fixed point of the transformation: $M(2, 3, 1)^T = (2, 3, 1)^T$. Thus if we take $P = (-2, 3, 1)^T$ in the new system, it maps to $(-2, 3, 1)^T$ in the original system (check this visually).

Successive Changes in a Coordinate frame.

Now consider forming a transformation by making two successive changes of the coordinate system. What is the overall effect? As suggested in Figure 5.32, system #1 is converted to system #2 by transformation $T_1(\cdot)$, and system #2 is then transformed to system #3 by transformation $T_2(\cdot)$. Note that system #3 is transformed relative to #2.

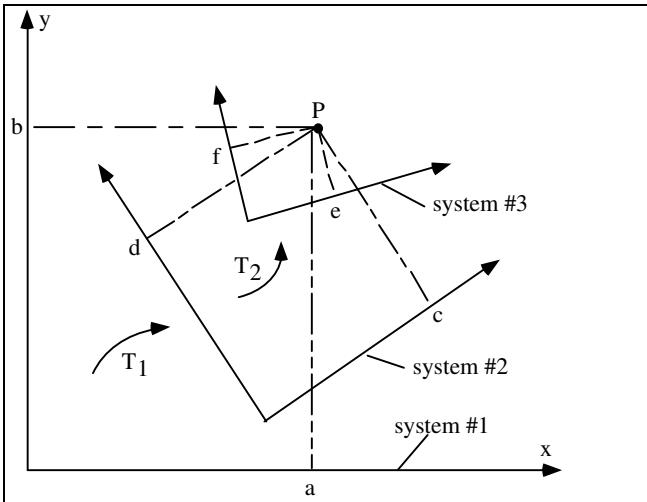


Figure 5.32. Transforming a coordinate system twice.

Again the question is: if point P has representation $(e, f, 1)^T$ with respect to system #3, what are its coordinates $(a, b, 1)^T$ with respect to the original system #1?

To answer this just work backwards and collect the effects of each transformation. In terms of system #2 the point P has coordinates $(c, d, 1)^T = M_2(e, f, 1)^T$. And in terms of system #1 the point $(c, d, 1)^T$ has coordinates $(a, b, 1)^T = M_1(c, d, 1)^T$. Putting these together:

$$\begin{pmatrix} a \\ b \\ 1 \end{pmatrix} = M_1 \begin{pmatrix} c \\ d \\ 1 \end{pmatrix} = M_1 M_2 \begin{pmatrix} e \\ f \\ 1 \end{pmatrix} \quad (5.36)$$

The essential point is that when determining the desired coordinates $(a, b, 1)^T$ from $(e, f, 1)^T$ we first apply M_2 and then M_1 , just the opposite order as when applying transformations to points.

We summarize this fact for the case of three successive transformations. The result generalizes immediately to any number of transformations.

Transforming points. To apply a sequence of transformations $T_1(\cdot), T_2(\cdot), T_3(\cdot)$ (in that order) to a point P , form the matrix:

$$M = M_3 \times M_2 \times M_1$$

Then P is transformed to MP . To compose each successive transformation M_i you must premultiply by M_i .

Transforming the coordinate system. To apply a sequence of transformations $T_1(\cdot), T_2(\cdot), T_3(\cdot)$ (in that order) to the coordinate system, form the matrix:

$$M = M_1 \times M_2 \times M_3$$

Then a point P expressed in the transformed system has coordinates MP in the original system. To compose each additional transformation M_i you must *postmultiply* by M_i .

How OpenGL operates.

We shall see in the next section that OpenGL provides tools for successively applying transformations in order to build up an overall “current transformation”. In fact OpenGL is organized to *postmultiply* each new transformation matrix to combine it with the current transformation. Thus it will often seem more natural to the modeler to think in terms of successively transforming the coordinate system involved, as the order in which these transformations is carried out is the *same* as the order in which OpenGL computes them.

Practice Exercises.

5.4.1. How transforming a coordinate system relates to transforming a point.

We wish to show the result in Equation 5.35. To do this, show each of the following steps.

- Show why the point P with representation $(c, d, 1)^T$ used in system #2 lies at $c\mathbf{i}' + d\mathbf{j}' + \mathbf{v}'$.
- We want to find where this point lies in system #1. Show that the representation (in system #1) of \mathbf{i}' is $M(1, 0, 0)^T$, that of \mathbf{j}' is $M(0, 1, 0)^T$, and that of \mathbf{v}' is $M(0, 0, 1)^T$.
- Show that therefore the representation of the point $c\mathbf{i}' + d\mathbf{j}' + \mathbf{v}'$ is $cM(1, 0, 0)^T + dM(0, 1, 0)^T + M(0, 0, 1)^T$.
- Show that this is the same as $M(c, 0, 0)^T + M(0, d, 0)^T + M(0, 0, 1)$ and that this is $M(c, d, 1)^T$, as claimed.

5.4.2. Using elementary examples. Figure 5.33 shows the effect of four elementary transformations of a coordinate system. In each case the original system with axes x and y is transformed into the new system with axes x' and y' .

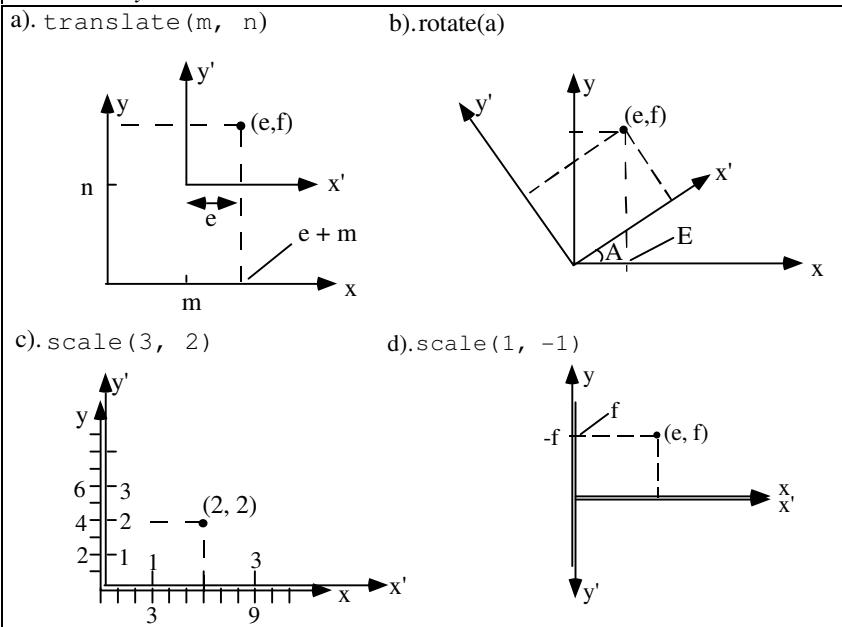


Figure 5.33. Elementary changes between coordinate systems.

- Part a) shows the effect of a translation through (m, n) . Show that point (e, f) in the new system lies at $(e + m, f + n)$ in the original system.
- Part b) shows the effect of a rotation about the origin through A degrees. Show that the point (e, f) in the new system lies at $(e \cos(a) - f \sin(a), e \sin(a) + f \cos(a))$, where $a = \pi A / 180$ radians.
- Part c) shows the effect of a scaling of the axes by $(3, 2)$. To make the figure clearer the new and old axes are shown slightly displaced. Show that a point (e, f) in the new system lies at $(3e, 2f)$ in the original system.
- Part d) shows a special case of scaling, a reflection about the x -axis. Show that the point (e, f) lies in the original system at $(e, -f)$.

5.5. Using Affine Transformations in a Program.

We want to see how to apply the theory of affine transformations in a program to carry out scaling, rotating, and translating of graphical objects. We also investigate how it is done when OpenGL is used. We look at 2D examples first as they are easier to visualize, then move on to 3D examples.

To set the stage, suppose you have a routine `house()` that draws the house #1 in Figure 5.34. But you wish to draw the version #2 shown that has been rotated through -30° and then translated through $(32, 25)$. This is a frequently encountered situation: an object is defined at a convenient size and position, but we want to draw it (perhaps many times) at different sizes, orientations, and locations.

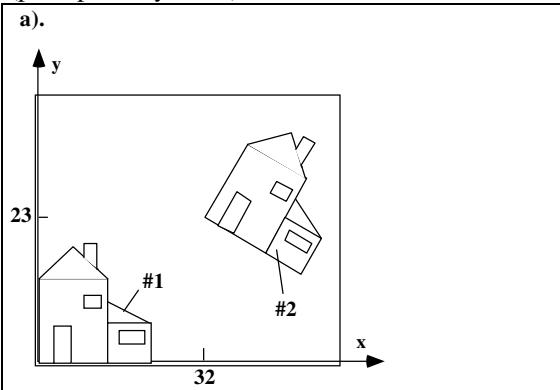


Figure 5.34. Drawing a rotated and translated house.

As we discussed in Chapter 3, `house()` would draw the various polylines of the figure. If it were written in “raw” OpenGL it might consist of a large number of chunks like:

```
glBegin(GL_LINES);
    glVertex2d(V[0].x, V[0].y);
    glVertex2d(V[1].x, V[1].y);
    glVertex2d(V[2].x, V[2].y);
    .... // the remaining points
glEnd();
```

based on some array `V[]` of points. Or if we use the `Canvas` class developed in Chapter 3 there would be a number of calls to `moveTo()` and `lineTo()` as in (using the global `canvas` object `cvs`):

```
cvs.moveTo(V[0]);
cvs.lineTo(V[1]);
cvs.lineTo(V[2]);
... // the remaining points
```

In either case we would set up a world window and a viewport with calls like:

```
cvs.setWindow(...);
cvs.setViewport(...);
```

and we would be assured that all vertex positions `V[i]` are “quietly” converted from world coordinates to screen window coordinates by the underlying window to viewport transformation.

But how do we arrange matters so that house #2 is drawn instead? There is the hard way and the easy way.

The hard way.

With this approach we construct the matrix for the desired transformation, say `M`, and build a routine, say `transform2D()`, that transforms one point into another, such that:

```
Q = transform2D(M, P);
```

The routine produces $\tilde{Q} = \tilde{M}\tilde{P}$. To apply the transformation to each point $V[i]$ in `house()` we must adjust the source code above as in

```
cvs.moveTo(transform2D(M, V[0])); // move to the transformed point
cvs.lineTo(transform2D(M, V[1]));
cvs.lineTo(transform2D(M, V[2]));
...
...
```

so that the *transformed* points are sent to `moveTo()` and `lineTo()`. This is workable if the source code for `house()` is at hand. But it is cumbersome at best, and *not possible* at all if the source code for `house()` is not available. It also requires tools to create the matrix M in the first place.

The easy way.

We cause the desired transformation to be applied automatically to each vertex. Just as we know the window to viewport mapping is “quietly” applied to each vertex as part of `moveTo()` and `lineTo()`, we can have an additional transformation be quietly applied as well. It is often called the **current transformation**, CT . We enhance `moveTo()` and `lineTo()` in the Canvas class so that they first quietly apply this transformation to the argument vertex, and then apply the window to viewport mapping. (Clipping is performed at the world window boundary as well.)

Figure 5.35 provides a slight elaboration of the graphics pipeline we introduced in Figure 5.7. When `glVertex2d()` is called with argument V , the vertex V is first transformed by the CT to form point Q . Q is then passed through the window to viewport mapping to form point S in the screen window. (As we see later, clipping is also performed, “inside” this last mapping process.)

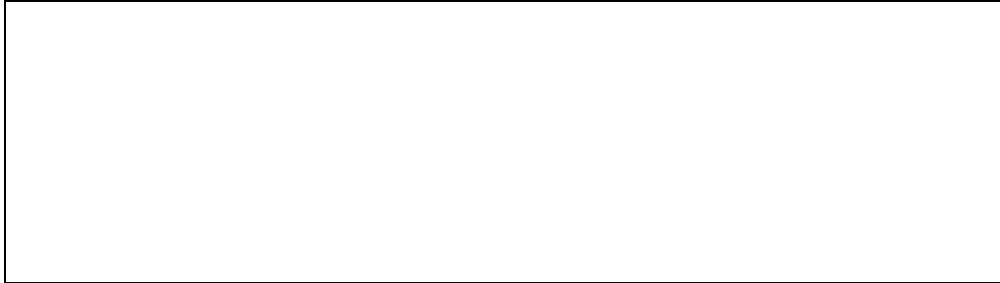


Figure 5.35. The current transformation is applied to vertices.

How do we extend `moveTo()` and `lineTo()` so they quietly carry out this additional mapping? (i.e. how do we rewrite these functions in the Canvas class?) If you are not using OpenGL you must write code that actually performs the transformation; this is described in Case Study 5.1. If you are using OpenGL it is done automatically! OpenGL maintains a so-called **modelview matrix**, and every vertex that is passed down the graphics pipeline is multiplied by it. We need only set up the modelview matrix to embody the desired transformation.

OpenGL works entirely in 3D, so its modelview matrix produces 3D transformations. Here we work with the modelview matrix in a restricted way to perform 2D transformations. Later we use its full power. Figure 5.36 shows how we restrict the 3D transformations to carry out the desired 2D transformations. The main idea is that 2D drawing is done in the xy -plane: the z -coordinate is understood to be zero. Therefore when we transform 2D points we set the part of the underlying 3D transformation that affects the z -coordinate so that it has no effect at all. For example, rotating about the origin in 2D is equivalent to rotating about the z -axis in 3D, as shown in the figure. Further, although a scaling in 3D takes three scale factors, S_x , S_y , and S_z to scale in the x , y , and z dimensions, respectively, we set the scale factor $S_z = 1$.



Figure 5.36. 2D drawing takes place in the xy -plane.

The principal routines for altering the modelview matrix are `glRotated()`¹⁰, `glScaled()`, and `glTranslated()`. These don't set the *CT* directly; instead each *postmultiplies* the *CT* (the modelview matrix) by a particular matrix, say M , and puts the result back into the *CT*. That is, each of these routines creates a matrix M as required for the new transformation, and performs:

$$CT = CT^* M \quad (5.37)$$

The order is important. As we saw earlier, applying $CT * M$ to a point is equivalent to first performing the transformation embodied in M , followed by performing the transformation dictated by the previous value of CT . Or if we are thinking in terms of transforming the coordinate system, it is equivalent to performing one additional transformation to the existing current coordinate system.

OpenGL routines for applying transformations in the 2D case are:

- `glScaled(sx, sy, 1.0);` Postmultiply CT by a matrix that performs a scaling by sx in x and by sy in y ; Put the result back in CT . No scaling in z is done.
 - `glTranslated(dx, dy, 0);` Postmultiply CT by a matrix that performs a translation by dx in x and by dy in y ; Put the result back in CT . No translation in z is done.
 - `glRotated(angle, 0, 0, 1);` Postmultiply CT by a matrix that performs a rotation through $angle$ degrees about the z -axis (indicated by $(0, 0, 1)$)¹¹. Put the result back in CT .

Since these routines only compose a transformation with the *CT*, we need some way to get started: to initialize the *CT* to the identity transformation. OpenGL provides `glLoadIdentity()`. And because these functions can be set to work on any of the matrices that OpenGL supports, we must inform OpenGL which matrix we are altering. This is accomplished using `glMatrixMode(GL_MODELVIEW)`.

Figure 5.37 shows suitable definitions of four new methods of the *Canvas* class that manage the *CT* and allow us to build up arbitrarily complex 2D transformations. Their pleasing simplicity is possible because OpenGL is doing the hard work.

```
//<<<<<<< initCT >>>>>>>>>>
void Canvas:: initCT(void)
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();           // set CT to the identity matrix
}
//<<<<<<<< scale2D >>>>>>>>>>>>>
void Canvas:: scale2D(double sx, double sy)
{
    glMatrixMode(GL_MODELVIEW);
    glScaled(sx, sy, 1.0); // set CT to CT * (2D scaling)
}
//<<<<<<<< translate2D >>>>>>>>>>>
void Canvas:: translate2D(double dx, double dy)
{
```

¹⁰ The suffix ‘d’ indicates that its arguments are doubles. There is also the version `glRotatef()` that takes float arguments.

¹¹ Here, as always, positive angles produce CCW rotations.

```

    glMatrixMode(GL_MODELVIEW);
    glTranslated(dx, dy, 1.0); // set CT to CT * (2D translation)
}
//<<<<<<<<< rotate2D >>>>>>>>>>>>>
void Canvas::rotate2D(double angle)
{
    glMatrixMode(GL_MODELVIEW);
    glRotated(angle, 0.0, 0.0, 1.0); // set CT to CT * (2D rotation)
}

```

Figure 5.37. Routines to manage the *CT* for 2D transformations.

We are now in a position to use 2D transformations. Returning to drawing version #2 of the house in Figure 5.34, we next show the code that first rotates the house through -30° and then translates it through $(32, 25)$. Notice that, to get the ordering straight, it calls the operations in opposite order to the way they are applied: *first* the translation operation, and *then* the rotation operation.

```

cvs.setWindow(...);
cvs.setViewport(...);      // set the window to viewport mapping
cvs.initCT();            // get started with the identity transformation
house();                 // draw the untransformed house first
cvs.translate2D(32, 25); // CT now includes translation
cvs.rotate2D(-30.0);     // CT now includes translation and rotation
house();                 // draw the transformed house

```

Notice that we can scale, rotate, and position the house in any manner we choose, and never need to “go inside” the routine `house()` or alter it. (In particular, the source code for `house()` need not be available.)

Some people find it more natural to think in terms of transforming the coordinate system. As shown in Figure 5.38 they would think of first translating the coordinate system through $(32, 25)$ to form system #2, and then rotating *that* system through -30° to obtain coordinate system #3. Because OpenGL applies transformations in the order that coordinate systems are altered, the code for doing it this way first calls `cvs.translate2D(32, 25)` and then calls `cvs.rotate2D(-30.0)`. This is, of course, *identical* to the code obtained doing it the other way, but it has been arrived at through a different thinking process.

Figure 5.38. The same transformation viewed as a sequence of coordinate system changes.

We give some further examples to show how easily the *CT* is manipulated to produce various effects.

Example 5.5.1. Capitalizing on rotational symmetry.

Figure 5.39a shows a star made of stripes that seem to interlock with one another. This is easy to draw using `rotate2D()`. Suppose that routine `starMotif()` draws a part of the star, the polygon shown in Figure 5.39b. (Determining the positions of this polygon’s vertices is challenging, and is addressed in Case Study 5.2.) To draw the whole star we just draw the motif five times, each time rotating the motif through an additional 72° :

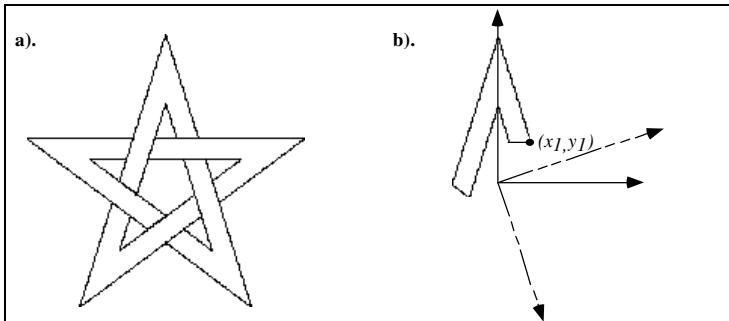


Figure 5.39. Using successive rotations of the coordinate system.

```
for(int count = 0; count < 5; count++)
{
    starMotif();
    cvs.rotate2D(72.0); // concatenate another rotation
}
```

Visualize what is happening during each of these steps.

Example 5.5.2. Drawing snowflakes.

The beauty of a snowflake arises in good measure from its high degree of symmetry. A snowflake has six identical spokes oriented 60° apart, and each spoke is symmetrical about its own axis. It is easy to produce a complex snowflake by designing one half of a spoke, and drawing it 12 times. Figure 5.40a shows a snowflake, based on the motif shown in Figure 5.40b. The motif is a polyline that meanders around above the positive x -axis. (To avoid any overlap with other parts of the snowflake, the polyline is kept below the 30° line shown in the figure.) Suppose the routine `flakeMotif()` draws this polyline.

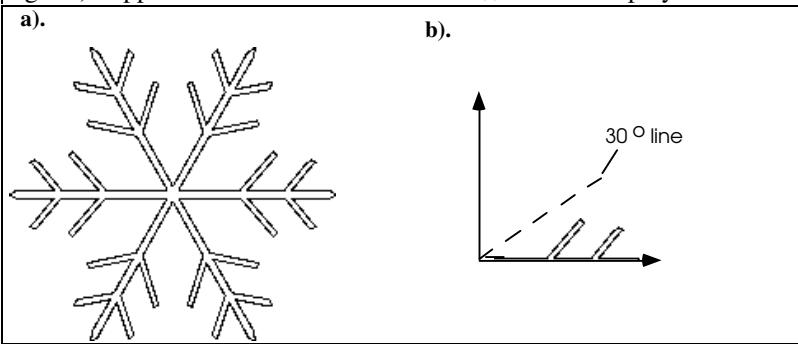


Figure 5.40. Designing a Snowflake.

Each spoke of the snowflake is a combination of the motif and a reflected version. A reflection about the x -axis is achieved by the use of `scale2D(1, -1)` (why?), so the motif plus its reflection can be drawn using

```
flakeMotif(); // draw the top half
cvs.scale2D(1.0, -1.0); // flip it vertically
flakeMotif(); // draw the bottom half
cvs.scale2D(1.0, -1.0); // restore the original axis
```

To draw the entire snowflake just do this six times, with an intervening rotation of 60° :

```
void drawFlake()
{
    for(int count = 0; count < 6; count++) // draw a snowflake
    {
        flakeMotif();
        cvs.scale2D(1.0, -1.0);
        flakeMotif();
```

```

        cvs.scale2D(1.0,-1.0);
        cvs.rotate2D(60.0);                                // concatenate a 60 degree rotation
    }
}

```

Example 5.5.3. A Flurry of Snowflakes.

A flurry of snowflakes like that shown in Figure 5.41 can be drawn by drawing the flake repeatedly at random positions, as in:

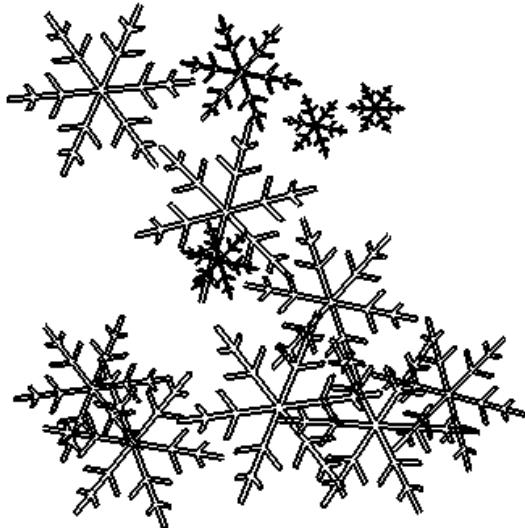


Figure 5.41. A flurry of snowflakes.

```

while(!bored)
{
    cvs.initCT();
    cvs.translate2D(random amount, random amount);
    drawFlake();
}

```

Notice that the *CT* has to be initialized each time, to prevent the translations from accumulating.

Example 5.5.4. Making patterns from a motif.

Figure 5.42 shows two configurations of the dinosaur motif. The dinosaurs are distributed around a circle in both versions, but in one case each dinosaur is rotated so that its feet point toward the origin, and in the other all the dinosaurs are upright. In both cases a combination of rotations and translations is used to create the pattern. It is interesting to see how the ordering of the operations affects the picture.

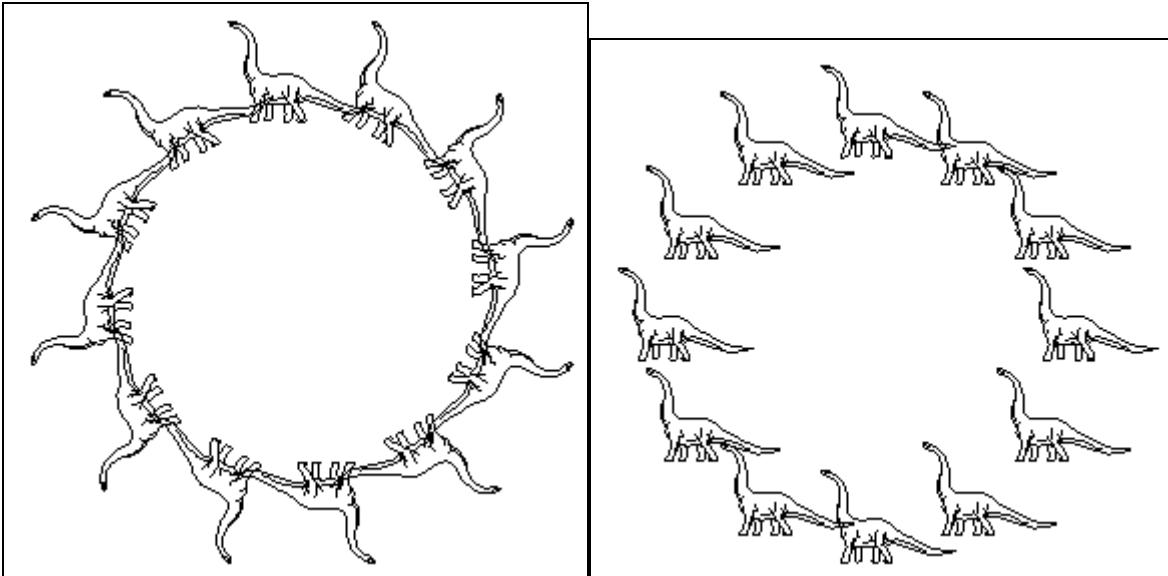


Figure 5.42. Two patterns based on a motif. a). each motif is rotated separately. b). all motifs are upright.

Suppose that `drawDino()` draws an upright dinosaur centered at the origin. In part a) the coordinate system for each motif is first rotated about the origin through a suitable angle, and then this coordinate system is translated along its y-axis by H units as shown in the following code. Note that the *CT* is reinitialized each time through the loop so that the transformations don't accumulate. (Think through the transformations you would use if instead you took the point of view of transforming points of the motif.)

```
const int numMotifs = 12;
for(int i = 0; i < numMotifs; i++)
{
    cvs.initCT(); // init CT at each iteration
    cvs.rotate2D(i * 360 / numMotifs); // rotate
    cvs.translate2D(0.0, H); // shift along y-axis
    drawDino();
}
```

An easy way to keep the motifs upright as in part b) is to “pre-rotate” each motif before translating it. If a particular motif is to appear finally at 120° , it is first rotated (while still at the origin) through -120° , then translated up by H units, and then rotated through 120° . What adjustments to the preceding code will achieve this?

5.5.1. Saving the *CT* for later use.

A program can involve rather lengthy sequences of calls to `rotate2D()`, `scale2D()`, and `translate2D()`. These functions make “additional” or “relative” changes to the *CT*, but sometimes we may need to “back up” to some prior *CT* in order to follow a different path of transformations for the next instance of a picture. In order to “remember” the desired *CT* we make a copy of it and store it in a convenient location. Then at a later point we can restore this matrix as the *CT*, effectively returning to the transformation that was in effect at that point. We may even want to keep a collection of “prior” *CT*’s, and return to selected ones at key moments.

To do this you can work with a **stack of transformations**, as suggested by Figure 5.43. The top matrix on the stack is the actual *CT*, and operations like `rotate2D()` compose their transformation with it in the manner described earlier. To save this *CT* for later use a copy of it is made and “pushed” onto the stack using a routine `pushCT()`. This makes the top two items on the stack identical. The top item can now be altered further with additional calls to `scale2D()` and the like. To return to the previous *CT* the top item is simply “popped” off the stack using `popCT()`, and discarded. This way we can return to the most recent *CT*, the next most recent *CT*, and so forth, in a last-in, first-out order.

a). before b). after pushCT() c). after rotate2D() d). after popCT()

Figure 5.43. Manipulating a stack of *CT*'s.

The implementation of `pushCT()` and `popCT()` is simple when OpenGL is being used, since OpenGL has routines `glPushMatrix()` and `glPopMatrix()` to manage several different stacks of matrices. Figure 5.44 shows the required functions. Note that each of them must inform OpenGL which matrix stack is being affected.

```
void Canvas:: pushCT(void)
{
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();           // push a copy of the top matrix
}
void Canvas:: popCT(void)
{
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();           // pop the top matrix from the stack
}
```

Figure 5.44. Routines to save and restore CT's.

Example 5.5.5: Tilings made easy.

Many beautiful designs called **tilings** appear on walls, pottery, and fabric. They are based on the repetition of a basic motif both horizontally and vertically. Consider tiling the window with some motif, as suggested in Figure 5.45. The motif is drawn centered in its own coordinate system as shown in part a) using some routine `drawMotif()`. Copies of the motif are drawn L units apart in the x-direction, and D units apart in the y-direction, as shown in part b).

a). the motif b). the tiling

Figure 5.45. A tiling based on a motif.

Figure 5.46 shows how easily the coordinate system can be manipulated in a double loop to draw the tiling. The *CT* is restored after drawing each row, so it returns to the start of that row, ready to move up to start the next row. In addition, the whole block of code is surrounded with a `pushCT()` and a `popCT()`, so that after the tiling has been drawn the *CT* is returned to its initial value, in case more drawing needs to be done.

```
cvs.pushCT();           // so we can return here
cvs.translate2D(W, H);   // position for the first motif
for(row = 0; row < 3; row++) // draw each row
{
    cvs.pushCT();
    for(col = 0; col < 4; col++)// draw the next row of motifs
    {
        motif();
        cvs.translate2D(L, 0); // move to the right
    }
    cvs.popCT();           // back to the start of this row
    cvs.translate2D(0, D);  // move up to the next row
}
```

```

}
cvs.popCT();                                // back to where we started

```

Figure 5.46. Drawing a hexagonal tiling.

Example 5.5.6. Using modeling transformations in a CAD program. Some programs must draw many instances of a small collection of shapes. Figure 5.47 shows the example of a CAD program that analyzes the behavior of an interconnection of digital logic gates. The user can construct a circuit by “picking and placing” different gates at different places in the work area, possibly with different sizes and orientations. Each picture of the object in the scene is called an **instance** of the object. A single definition of the object is given in a coordinate system that is convenient for that object shape, called its **master coordinate system**. The transformation that carries the object from its own master coordinate system to world coordinates to produce an instance is often called a **modeling transformation**.

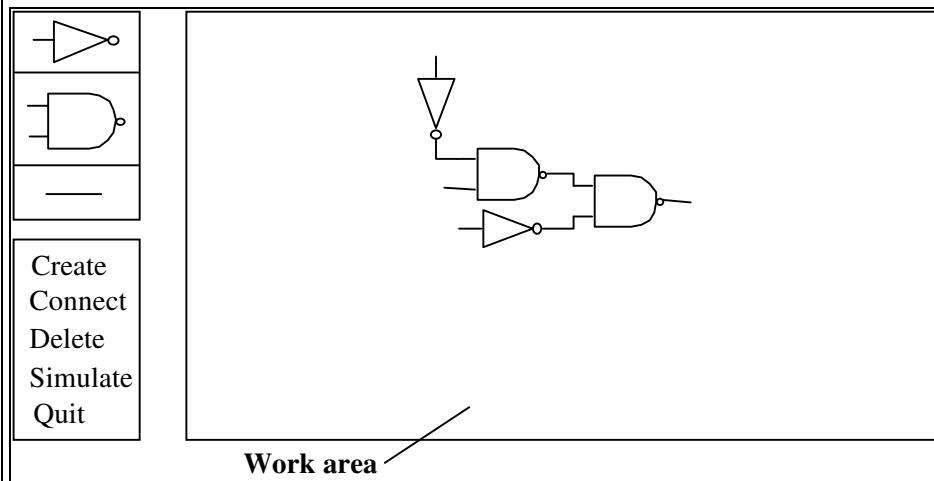


Figure 5.47. Creating instances in a pick-and-place application.

Figure 5.48 shows two logic gates, each defined once in its own master coordinate system. As the user creates each instance of one of these gates, the appropriate modeling transformation is generated that orients and positions the instance. The transformation might be stored simply as a set of parameters, say, S , A , dx , dy , with the understanding that the modeling transformation would always consist of:

1. A scaling by factor S ;
2. A rotation through angle A
3. A translation through (dx, dy)

performed in that order. A list is kept of the gates in the circuit, along with the transformation parameters of each gate.

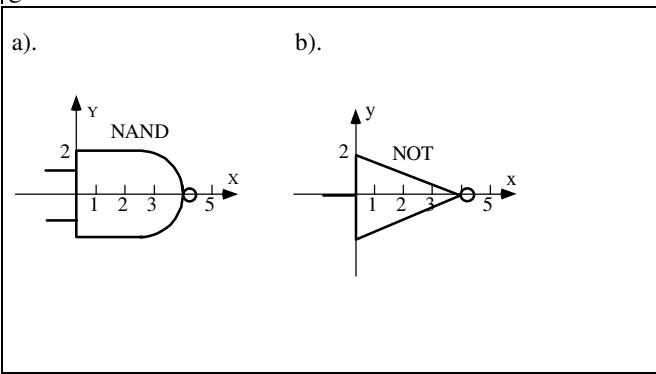


Figure 5.48. Each gate type is defined in its own coordinate system.

Whenever the drawing must be refreshed, each instance is drawn in turn, with the proper modeling transformation applied. Code to do this might look like:

```
clear the screen
for(i = 0; i < numberOfGates; i++) // for each gate
{
    pushCT(); // remember the CT
    translate2D(dx[i], dy[i]); // apply the transformation
    rotate2D(A[i]);
    scale2D( S[i], S[i]);
    drawGate(type[i]); // draw one of the two types
    popCT(); // restore the CT
}
```

The CT is pushed before drawing each instance, so that it can be restored after the instance has been drawn. The modeling transformation for the instance is determined by its parameters, and then one of the two gate shapes is drawn. The necessary code has a simple organization, because the burden of sizing, orienting, and positioning each instance has been passed to the underlying tools that maintain the *CT* and its stack.

Practice Exercises.

5.5.1. Developing the Transformations. Supposing OpenGL were not available, detail how you would write the routines that perform elementary coordinate system changes:

```
void scale2D(double sx, double sy);
void translate2D(double dx, double dy);
```

5.5.2. Implementing the transformation stack. Define, in the absence of OpenGL, appropriate data types for a stack of transformations, and write the routines `pushCT()` and `popCT()`.

5.5.3. A hexagonal tiling. A hexagonal pattern provides a rich setting for tilings, since regular hexagons fit together neatly as in a beehive. Figure 5.49 shows 9 columns of stacked 6-gons. Here the hexagons are shown empty, but we could draw interesting figures inside them.

- Show that the length of a hexagon with radius R is also R .
- Show that the centers of adjacent hexagons in a column are separated vertically by $\sqrt{3} R$ and adjacent columns are separated horizontally by $3 R / 2$.
- Develop code that draws this hexagonal tiling, using `pushCT()` and `popCT()` and suitable transformations to keep track of where each row and column start.

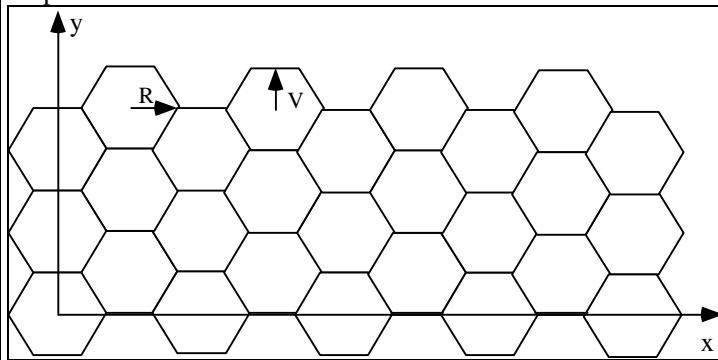


Figure 5.49. A simple hexagonal tiling.

5.6. Drawing 3D Scenes with OpenGL

We introduced the drawing of 2D objects in Chapter 3, developing a simple class *Canvas*. This class provides functions that establish a window and viewport, and that do line drawing through `moveTo()` and `lineTo()`. So far in this chapter we have added the notion of the *CT*, and provided functions that perform 2D rotations, scalings, and translations. These 2D transformations are really just special cases of 3D transformations: they basically ignore the third dimension.

In this section we examine how 3D transformations are used in an OpenGL-based program. The main emphasis is on transforming objects in order to orient and position them as desired in a 3D scene. Not surprisingly it is all done with matrices, and OpenGL provides the necessary functions to build the required matrices. Further, the matrix stack maintained by OpenGL makes it easy to set up a transformation for one object, and then “back up” to a previous transformation, in preparation for transforming another object.

It is very satisfying to build a program that draws different scenes using a collection of 3D transformations. Experimenting with such a program also improves your ability to visualize what the various 3D transformations do. OpenGL makes it easy to set up a “camera” that takes a “snapshot” of the scene from a particular point of view. The camera is created with a matrix as well, and we study in detail in Chapter 7 how this is done. Here we just use an OpenGL tool to set up a reasonable camera, so that attention can be focussed on transforming objects. Granted we are using a tool before seeing exactly how it operates, but the payoff is high: you can make impressive pictures of 3D scenes with a few simple calls to OpenGL functions.

5.6.1. An overview of the viewing process and the graphics pipeline.

All of our 2D drawing so far has actually used a special case of 3D viewing, based on a simple “parallel projection”. We have been using the “camera” suggested in Figure 5.50. The “eye” that is viewing the scene looks along the z-axis at the “window”, a rectangle lying in the xy-plane. The **view volume** of the camera is a rectangular parallelepiped, whose four side walls are determined by the border of the window, and whose other two walls are determined by a **near plane** and a **far plane**. Points lying inside the view volume are projected onto the window along lines parallel to the z-axis. This is equivalent to simply ignoring their z-component, so that the 3D point $(x_1 y_1, z_1)$ projects to $(x_1, y_1, 0)$. Points lying outside the view volume are clipped off. A separate **viewport transformation** maps the projected points from the window to the viewport on the display device.



Figure 5.50. Simple viewing used in OpenGL for 2D drawing.

Now we move into 3D graphics and place 3D objects in a scene. For the examples here we continue to use a parallel projection. (The more realistic perspective projection, for which more remote objects appear smaller than nearby objects, is described in Chapter 7.) Therefore we use the same camera as in Figure 5.50, but allow it to have a more general position and orientation in the 3D scene, in order to produce better views of the scene.

Figure 5.51 shows such a camera immersed in a scene, The scene consists of block, part of which lies outside the view volume. The image produced by this camera is also shown.



Figure 5.51. A camera to produce parallel views of a scene.

We saw in the previous section that OpenGL provides the three functions `glScaled(..)`, `glRotated(..)`, and `glTranslated(..)` for applying modeling transformations to a shape. The block in Figure 5.51 is in fact a cube that has been stretched, rotated, and shifted as shown. OpenGL also provides functions for defining the view volume and its position in the scene.

The graphics pipeline implemented by OpenGL does its major work through matrix transformations, so we first give insight into what each of the matrices in the pipeline does. At this point it is important only to grasp the basic idea of how each matrix operates: in Chapter 7 we give a detailed discussion. Figure 5.52 shows the pipeline (slightly simplified). Each vertex of an object is passed through this pipeline with a call such as `glVertex3d(x, y, z)`. The vertex is multiplied by the various matrices shown, it is clipped if necessary, and if it survives clipping it is ultimately mapped onto the viewport. Each vertex encounters three matrices:



Figure 5.52. The OpenGL pipeline (slightly simplified).

- The **modelview matrix**;
- The **projection matrix**;
- The **viewport matrix**;

The **modelview matrix** basically provides what we have been calling the *CT*. It combines two effects: the sequence of modeling transformations applied to objects, and the transformation that orients and positions the camera in space (hence its peculiar name *modelview*). Although it is a single matrix in the actual pipeline, it is easier to think of it as the product of two matrices, a modeling matrix M , and a viewing matrix V . The modeling matrix is applied first, and then the viewing matrix, so the modelview matrix is in fact the product VM (why?).

Figure 5.53 suggests what the M and V matrices do, for the situation introduced in Figure 5.51, where a camera “looks down” on a scene consisting of a block. Part a shows a unit cube centered at the origin. A modeling transformation based on M scales, rotates, and translates the cube into block shown in part b. Part b also shows the relative position of the camera’s view volume.

a). before model b). after model c) after model view

Figure 5.53. Effect of the modelview matrix in the graphics pipeline. a). Before the transformations. b). After the modeling transformation. c). After the modelview transformation.

The V matrix is now used to rotate and translate the block into a new position. The specific transformation used is that which would carry the camera from its position in the scene to its “generic” position, with the eye at the origin and the view volume aligned with the z -axis, as shown in part c. The vertices of the block are now positioned (that is, their coordinates have the proper values) so that projecting them onto a plane such as the near plane yields the proper values for displaying the projected image. So the matrix V in fact effects a change of coordinates of the scene vertices into the camera’s coordinate system. (Camera coordinates are sometimes also called **eye coordinates**.)

In the camera coordinate system the edges of the view volume are parallel to the x -, y -, and z -axes. The view volume extends from *left to right* in x , from *bottom to top* in y , and from *-near to -far* in z , as shown. When the vertices of the original cube have passed through the entire modelview matrix, they are located as shown in part c.

The **projection matrix** scales and shifts each vertex in a particular way, so that all those that lie inside the view volume will lie inside a *standard cube* that extends from -1 to 1 in each dimension¹². (When perspective projections are being used this matrix does quite a bit more, as we see in Chapter 7.) This matrix effectively squashes the view volume into the cube centered at the origin. This cube is a particularly efficient boundary against which to clip objects, as we see in Chapter 7. Scaling the block in this fashion might badly distort it, of course, but this distortion will be compensated for in the viewport transformation. The projection matrix also reverses the sense of the z -axis, so that increasing values of z now represent increasing values of depth of a point from the eye. Figure 5.54 shows how the block is transformed into a different block by this transformation.

Figure 5.54. Effect of the projection matrix (for parallel projections).

(Notice that the view volume of the camera need never be created as an object itself. It is defined only as that particular shape that the projection matrix converts into the standard cube!)

Clipping is now performed, which eliminates the portion of the block that lies outside the standard cube.

Finally, the **viewport matrix** maps the surviving portion of the block into a “3D viewport”. This matrix maps the standard cube into a block shape whose x and y values extend across the viewport (in screen coordinates), and whose z -component extends from 0 to 1 and retains a measure of the depth of point, as shown in Figure 5.55.

¹² Coordinates in this system are sometimes called *Normalized Device Coordinates*.

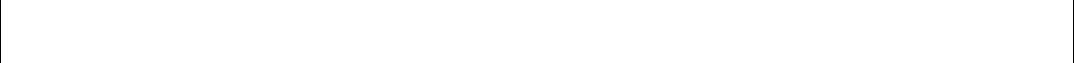


Figure 5.55. Effect of the viewport transformation.

This overview has outlined how the whole OpenGL graphics pipeline operates, showing that transformations are a central ingredient. Each vertex of an object is subjected to a sequence of transformations that carry it from world coordinates into eye coordinates, into a neutral system specially designed for clipping, and finally into the right coordinate system for proper display. Each transformation is effected by a matrix multiplication.

Aside: Some important details of the pipeline have been suppressed in this first overview. When perspective projections are used we need to include a “perspective division” that capitalizes on a property of homogeneous coordinates. And of course many interesting details lurk in the last step of actually rendering the image, such as computing the color of pixels “in between” vertices, and checking for proper hidden surface removal. These are all addressed in later chapters.

5.6.2. Some OpenGL Tools for Modeling and Viewing.

We now see what functions OpenGL provides for modeling and setting the camera, and how to use them.

1). Three functions are used to set modeling transformations.

The following functions are normally used to modify the modelview matrix, so the modelview matrix is first made “current” by executing: `glMatrixMode(GL_MODELVIEW);`

- `glScaled(sx, sy, sz);` Postmultiply the current matrix by a matrix that performs a scaling by `sx` in `x`, by `sy` in `y`, and by `sz` in `z`; Put the result back in the current matrix.
- `glTranslated(dx, dy, dz);` Postmultiply the current matrix by a matrix that performs a translation by `dx` in `x`, by `dy` in `y`, and by `dz` in `z`; Put the result back in the current matrix.
- `glRotated(angle, ux, uy, uz);` Postmultiply the current matrix by a matrix that performs a rotation through `angle` degrees about the axis that passes through the origin and the point (ux, uy, uz) .¹³ Put the result back in the current matrix. Equation 5.33 shows the matrix used to perform the rotation.

2). Setting the camera in OpenGL (for a parallel projection).

`glOrtho(left, right, bott, top, near, far);` Establishes as a view volume a parallelepiped that extends from¹⁴ `left` to `right` in `x`, from `bott` to `top` in `y`, and from `-near` to `-far` in `z`. (Since this definition operates in eye coordinates, the camera’s eye is at the origin, looking down the negative `z`-axis.) This function creates a matrix and postmultiplies the current matrix by it. (We show in Chapter 7 precisely what values this matrix contains.)

Thus to set the projection matrix use:

```
glMatrixMode(GL_PROJECTION);           // make the projection matrix current
glLoadIdentity();                     // set it to the identity matrix
glOrtho(left,right,bottom,top,near,far); // multiply it by the new matrix
```

Notice the minus signs above: because the default camera is located at the origin looking down the negative `z`-axis, using a value of `2` for `near` means to place the near plane at $z = -2$, that is, `2` units in front of the eye. Similarly, using `20` for `far` places the far plane `20` units in front of the eye.

3). Positioning and aiming the camera.

¹³ Positive values of `angle` produce rotations that are CCW as one looks along the axis from the point (ux, uy, uz) towards the origin.

¹⁴ All parameters of `glOrtho()` are of type `GLdouble`.

OpenGL offers a function that makes it easy to set up a basic camera:

```
gluLookAt(eye.x, eye.y, eye.z, look.x, look.y, look.z, up.x, up.y, up.z);  
Creates the view matrix and postmultiplies the current matrix by it.
```

It takes as parameters the eye position, `eye`, of the camera and the look-at point, `look`. It also takes an approximate “up” direction, `up`. Since the programmer knows where an interesting part of the scene is situated, it is usually straightforward to choose reasonable values for `eye` and `lookAt` for a good first view. And `up` is most often set to $(0, 1, 0)$ to suggest an “up” direction parallel to the y -axis. Later we develop more powerful tools for setting up a camera and for interactively “flying it” in an animation.

We want this function to set the V part of the modelview matrix VM . So it is invoked before any modeling transformations are added, since subsequent modeling transformations will postmultiply the modelview matrix. So to use `gluLookAt()` follow the sequence:

```
glMatrixMode(GL_MODELVIEW);           // make the modelview matrix current  
glLoadIdentity();                   // start with a unit matrix  
gluLookAt(eye.x, eye.y, eye.z,      // the eye position  
          look.x, look.y, look.z,     // the “look at” point  
          up.x, up.y, up.z)         // the up direction
```

We discuss how this function operates in Chapter 7, and also develop more flexible tools for establishing the camera. For those curious about what values `gluLookAt()` actually places in the modelview matrix, see the exercises.

Example 5.6.1: Set up a typical camera. Cameras are often set to “look down” on the scene from some nearby position. Figure 5.56 shows the camera with its eye situated at `eye = (4,4,4)` looking at the origin with `lookAt = (0,1,0)`. The up direction is set to `up = (0, 1, 0)`. Suppose we also want the view volume to have a width of 6.4, a height of 4.8 (so its aspect ratio is 640/480), and to set `near` to 1 and `far` to 50. This camera would be established using:

Figure 5.56. Setting a camera with `gluLookAt()`.

```
glMatrixMode(GL_PROJECTION); // set the view volume  
glLoadIdentity();  
glOrtho(-3.2, 3.2, -2.4, 2.4, 1, 50);  
glMatrixMode(GL_MODELVIEW); // place and aim the camera  
glLoadIdentity();  
gluLookAt(2, 4, 5, 0, 0, 0, 0, 1, 0);  
The exercises show the specific values that get placed in the modelview matrix.
```

Practice Exercises.

5.6.1. What does `gluLookAt()` do? We know that `gluLookAt()` builds a matrix that converts world coordinates into eye coordinates. Figure 5.57 shows the camera as a coordinate system suspended in the world, with its origin at `eye`, and oriented according to its three mutually perpendicular unit vectors \mathbf{u} , \mathbf{v} , and \mathbf{n} . The eye is “looking” in the direction $-\mathbf{n}$. `gluLookAt()` uses the parameters `eye`, `look`, and `up` to create \mathbf{u} , \mathbf{v} , and \mathbf{n} according to

Figure 5.57*. Converting from world to camera coordinates.

$$\mathbf{n} = \mathbf{eye} - \mathbf{look}$$

$$\mathbf{u} = \mathbf{up} \times \mathbf{n}$$

(5.38)

$$\mathbf{v} = \mathbf{n} \times \mathbf{u}$$

and then normalizes all three of these to unit length. It builds the matrix

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where the point d has components $(d_x, d_y, d_z) = (-\mathbf{eye} \cdot \mathbf{u}, -\mathbf{eye} \cdot \mathbf{v}, -\mathbf{eye} \cdot \mathbf{n})$.

- Show that \mathbf{u} , \mathbf{v} , and \mathbf{n} are mutually perpendicular.
- Show that matrix V properly converts world coordinates to eye coordinate by showing that it maps \mathbf{eye} into the origin $(0,0,0,1)^T$, \mathbf{u} into $\mathbf{i} = (1,0,0,0)$, \mathbf{v} into $\mathbf{j} = (0,1,0)$, and \mathbf{n} into $\mathbf{k} = (0,0,1)$.
- Show for the case $\mathbf{eye} = (4, 4, 4)$, $\mathbf{lookAt} = (0,1,0)$, and $\mathbf{up} = (0,1,0)$ that the resulting matrix is:

$$V = \begin{pmatrix} .70711 & 0 & -.70711 & 0 \\ -.3313 & .88345 & -.3313 & -.88345 \\ .6247 & .4685 & .6247 & -6.872 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5.6.2. Inquiring of the values in a matrix in OpenGL. Print out the values in the modelview matrix to test some of the assertions made about how it is formed. To see what is stored in the modelview matrix in OpenGL define an array `GLfloat mat[16]` and use `glGetFloatv(GL_MODELVIEW_MATRIX, mat)` which copies into `mat[]` the 16 values in the modelview matrix. `M[i][j]` is copied into the element `mat[4j+i]`, for $i, j = 0, 1, \dots, 3$.

5.6.3. Drawing Elementary Shapes Provided by OpenGL

We see in the next chapter how to create our own 3D objects, but at this point we need some 3D objects to draw, in order to experiment with setting and using cameras. The GLUT provides several ready-made 3D objects. These include a sphere, a cone, a torus, the five Platonic solids (discussed in Chapter 6), and the famous teapot. Each is available as a “wireframe” model and as a “solid” model with faces that can be shaded.

- **cube:** `glutWireCube(GLdouble size);` Each side is of length `size`
- **sphere:** `glutWireSphere(GLdouble radius, GLint nSlices, GLint nStacks)`
- **torus:** `glutWireTorus(GLdouble inRad, GLdouble outRad, GLint nSlices, GLint nStacks)`
- **teapot:** `glutWireTeapot(GLdouble size)`

There is also a `glutSolidCube()`, `glutSolidSphere()`, etc., that we use later. The shape of the torus is determined by inner radius `inRad` and outer radius `outRad`. The sphere and torus are approximated by polygonal faces, and you can adjust parameters `nSlices` and `nStacks` to specify how many faces to use in the approximation. `nSlices` is the number of subdivisions around the z -axis, and `nStacks` is the number of ‘bands’ along the z -axis, as if the shape were a stack of `nStacks` disks.

Four of the Platonic solids (the fifth is the cube, already presented):

- **tetrahedron:** `glutWireTetrahedron()`
- **octahedron:** `glutWireOctahedron()`
- **dodecahedron:** `glutWireDodecahedron()`
- **icosahedron:** `glutWireIcosahedron()`

All of the shapes above are centered at the origin.

- **cone:** `glutWireCone(GLdouble baseRad, GLdouble height, GLint nSlices, GLint nStacks)`

- **tapered cylinder:** `gluCylinder(GLUquadricObj * qobj, GLdouble baseRad, GLdouble topRad, GLdouble height, GLint nSlices, GLint nStacks)`

The axes of the cone and tapered cylinder coincide with the z -axis. Their bases rest on the $z = 0$ plane, and they extend to $z = \text{height}$ along the z -axis. The radius of the cone and tapered cylinder at $z = 0$ is given by `baseRad`. The radius of the tapered cylinder at $z = \text{height}$ is `topRad`.

The **tapered cylinder** is actually a *family* of shapes, distinguished by the value of `topRad`. When `topRad` is 1 there is no tapering; this is the classic **cylinder**. When `topRad` is 0 the tapered cylinder is identical to the **cone**.

Note that drawing the tapered cylinder in OpenGL requires some extra work, because it is a special case of a quadric surface, as we shall see in Chapter 6. To draw it you must 1) define a new quadric object, 2). set the drawing style (GLU_LINE for a wireframe, GLU_FILL for a solid rendering), and 3). draw the object:

```
GLUquadricObj * qobj = gluNewQuadric();           // make a quadric object
gluQuadricDrawStyle(qobj,GLU_LINE);                // set style to wireframe
gluCylinder(qobj, baseRad, topRad, nSlices, nStacks); // draw the cylinder
```

Figure 5.58. Shapes available in the GLUT.

We next employ some of these shapes in two substantial examples that focus on using affine transformations to model and view a 3D scene. The complete program to draw each scene is given. A great deal of insight can be obtained if you enter these programs and produce the figures, and then see the effect of varying the various parameters.

Example 5.6.2: A scene composed of wireframe objects.

Figure 5.59 shows a scene with several objects disposed at the corners of a unit cube. The cube has one corner at the origin. Seven objects appear at various corners of the cube, all drawn as wireframes.

The camera is given a view volume that extends from -2 to 2 in y , with an aspect ratio of `aspect = 640/480`. Its near plane is at $N = 0.1$, and its far plane is at $F = 100$. This is accomplished using:

```
glOrtho(-2.0* aspect, 2.0* aspect, -2.0, 2.0, 0.1, 100);
```

The camera is positioned with `eye = (2, 2.2, 3)`, `lookAt = (0, 0, 0)`, and `up = (0, 1, 0)` (parallel to the y -axis), using:

```
gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

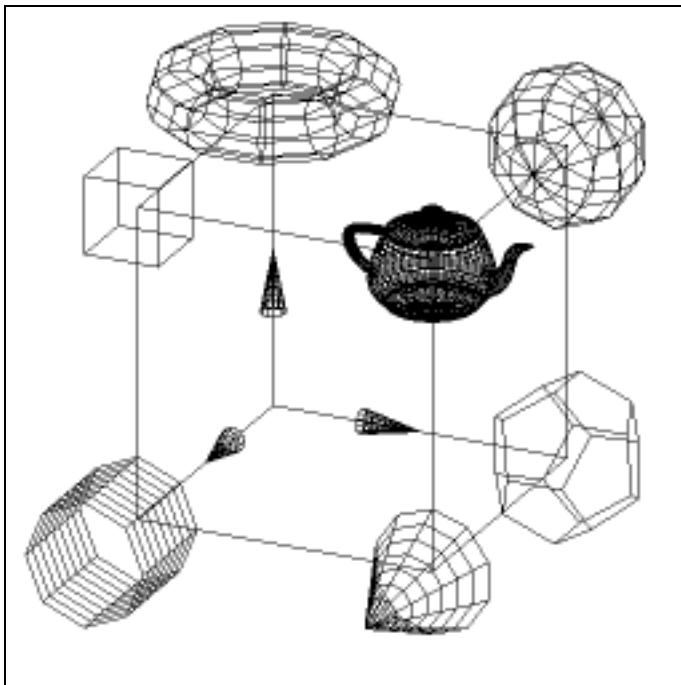


Figure 5.59. Wireframe drawing of various primitive shapes.

Figure 5.60 shows the complete program to produce the drawing. The `main()` routine initializes a 640 by 480 pixel screen window, sets the viewport and background color, and specifies `displayWire()` as the display function to be called to perform the drawing. In `displayWire()` the camera shape and position are established first. Then each object is drawn in turn. Most objects need their own modeling matrix, in order to rotate and position them as desired. Before each modeling transformation is established, a `glPushMatrix()` is used to member the current transformation, and after the object has been drawn this prior current transformation is restored with a `glPopMatrix()`. Thus the code to draw each object is imbedded in a `glPushMatrix()`, `glPopMatrix()` pair. Check each transformation carefully to see that it places its object at the proper place.

Also shown in the figure are the x -, y -, and z - axes drawn with conical arrow heads. Displaying the underlying coordinate system can help to orient the viewer. To draw the x -axis, the z -axis is rotated 90° about the y -axis to form a rotated system, and the axis is redrawn in its new orientation. Note that this axis is drawn without immersing it in a `glPushMatrix()`, `glPopMatrix()` pair, so the next rotation to produce the y -axis takes place in the already rotated coordinate system. Check that it's the proper rotation.


```

glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB );
glutInitWindowSize(640, 480);
glutInitWindowPosition(100, 100);
glutCreateWindow("Transformation testbed - wireframes");
glutDisplayFunc(displayWire);
glClearColor(1.0f, 1.0f, 1.0f, 0.0f); // background is white
glViewport(0, 0, 640, 480);
glutMainLoop();
}

```

Figure 5.60. Complete program to draw Figure 5.65 using OpenGL.

Notice that the sides of the large cube that are parallel in 3D are also displayed as parallel. This is a result of using a parallel projection. The cube looks slightly unnatural because we are used to seeing the world with a perspective projection. As we see in Chapter 7, if a perspective projection were used instead, these parallel edges would not be drawn parallel.

Example 5.6.3. A 3D scene rendered with shading.

We develop a somewhat more complex scene to illustrate further the use of modeling transformations. We also show how easily OpenGL makes it to draw much more realistic drawings of solid objects by incorporating shading, along with proper hidden surface removal.

Two views of a scene are shown in Figure 5.61. Both views use a camera set by `gluLookAt(2.3, 1.3, 2, 0, 0.25, 0, 0.0, 1.0, 0.0)`. Part a uses a large view volume that encompasses the whole scene; part b uses a small view volume that encompasses only a small portion of the scene, thereby providing a close-up view.

The scene contains three objects resting on a table in the corner of a “room”. Each of the three walls is made by flattening a cube into a thin sheet, and moving it into position. (Again, they look somewhat unnatural due to the use of a parallel projection.) The “jack” is composed of three stretched spheres oriented at right angles plus six small spheres at their ends.

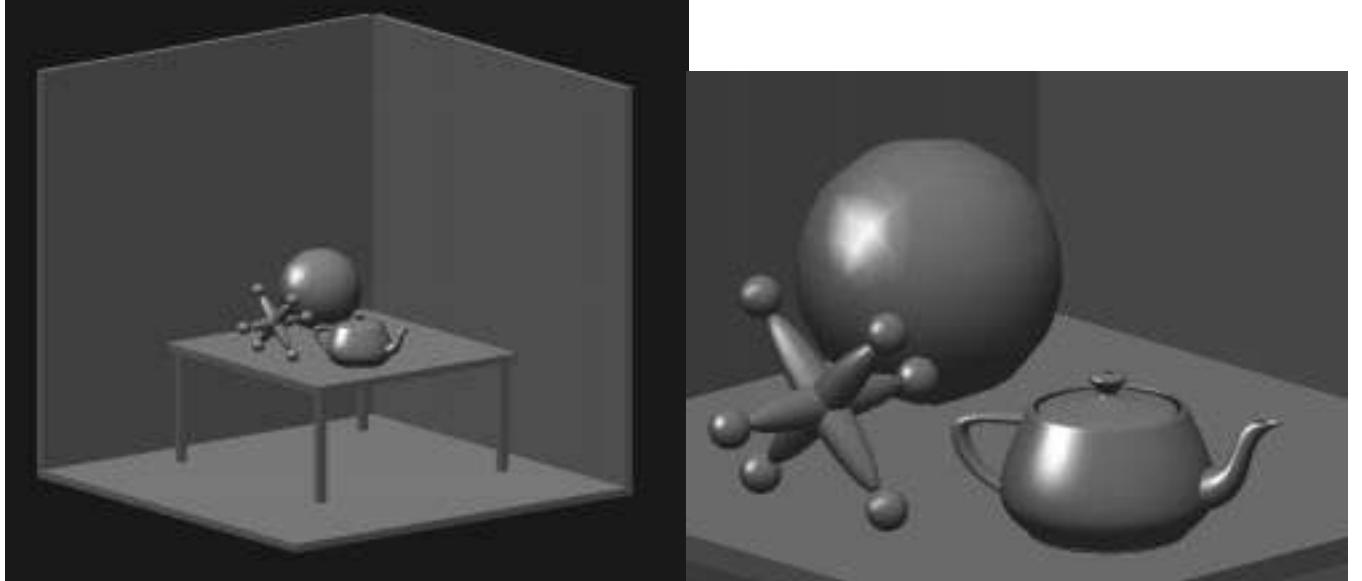


Figure 5.61. A simple 3D scene - a). Using a large view volume. b). Using a small view volume.

The table consists of a table top and four legs. Each of the table’s five pieces is a cube that has been scaled to the desired size and shape. The layout for the table is shown in Figure 5.62. It is based on four parameters that characterize the size of its parts: `topWidth`, `topThick`, `legLen`, and `legThick`. A routine `tableLeg()` draws each leg, and is called four times within the routine `table()` to draw the legs in the four different locations. The different parameters used produce different modeling transformations within `tableLeg()`. As always, a `glPushMatrix()`, `glPopMatrix()` pair surrounds the modeling functions to isolate their effect.

- | | |
|-----|-----|
| a). | b). |
|-----|-----|

Figure 5.62. Designing the table.

The complete code for this program is shown in Figure 5.63. Note that the “solid” version of each shape, such as `glutSolidSphere()`, is used here, rather than the “wire” version. Check the assorted transformations used to orient and position each object in the scene. Check the jack model particularly. This example is designed to whet your appetite for trying out scenes on your own, practicing with transformations.

The code also shows the various things that must be done to create shaded images. The position and properties of a light source must be specified, along with certain properties of the objects’ surfaces, in order to describe how they reflect light. Since we discuss the shading process in Chapter 8 we just present the various function calls here; using them as shown will generate shading.

```
#include <windows.h>
#include <iostream.h>
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>
//<<<<<<<<< wall >>>>>>>>>>>>
void wall(double thickness)
{ // draw thin wall with top = xz-plane, corner at origin
    glPushMatrix();
    glTranslated(0.5, 0.5 * thickness, 0.5);
    glScaled(1.0, thickness, 1.0);
    glutSolidCube(1.0);
    glPopMatrix();
}
//<<<<<<<<< tableLeg >>>>>>>>>>>>>
void tableLeg(double thick, double len)
{
    glPushMatrix();
    glTranslated(0, len/2, 0);
    glScaled(thick, len, thick);
    glutSolidCube(1.0);
    glPopMatrix();
}
//<<<<<<<<<<< jack part >>>>>>>>>>
void jackPart()
{ // draw one axis of the unit jack - a stretched sphere
    glPushMatrix();
    glScaled(0.2,0.2,1.0);
    glutSolidSphere(1,15,15);
    glPopMatrix();
    glPushMatrix();
    glTranslated(0,0,1.2); // ball on one end
    glutSolidSphere(0.2,15,15);
    glTranslated(0,0, -2.4);
    glutSolidSphere(0.2,15,15); // ball on the other end
    glPopMatrix();
}
//<<<<<<<<<< jack >>>>>>>>>>>>>>>
void jack()
{ // draw a unit jack out of spheroids
    glPushMatrix();
    jackPart();
    glRotated(90.0, 0, 1, 0);
    jackPart();
    glRotated(90.0, 1,0,0);
    jackPart();
    glPopMatrix();
}
//<<<<<<<<<<<<<<< table >>>>>>>>>>>>>>
```



```

    table(0.6, 0.02, 0.02, 0.3); // draw the table
    glPopMatrix();
    wall(0.02); // wall #1: in xz-plane
    glPushMatrix();
    glRotated(90.0, 0.0, 0.0, 1.0);
    wall(0.02); // wall #2: in yz-plane
    glPopMatrix();
    glPushMatrix();
    glRotated(-90.0, 1.0, 0.0, 0.0);
    wall(0.02); // wall #3: in xy-plane
    glPopMatrix();
    glFlush();
}
//<<<<<<<<<<< main >>>>>>>>>>>>>>>>>>>>>
void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("shaded example - 3D scene");
    glutDisplayFunc(displaySolid);
    glEnable(GL_LIGHTING); // enable the light source
    glEnable(GL_LIGHT0);
    glShadeModel(GL_SMOOTH);
    glEnable(GL_DEPTH_TEST); // for hidden surface removal
    glEnable(GL_NORMALIZE); // normalize vectors for proper shading
    glClearColor(0.1f, 0.1f, 0.1f, 0.0f); // background is light gray
    glViewport(0, 0, 640, 480);
    glutMainLoop();
}

```

Figure 5.63. Complete program to draw the shaded scene.

Practice Exercises.

5.6.3. Inquiring of the values in a matrix in OpenGL. Test some of the assertions above that put specific values into the modelview matrix. You can see what is stored in the modelview matrix in OpenGL by defining an array `GLfloat mat[16]` and using `glGetFloatv(GL_MODELVIEW_MATRIX, mat)` which copies into `mat[]` the 16 values in the modelview matrix. $M[i][j]$ is copied into the element `mat[4j+i]`, for $i, j = 0, 1, \dots, 3$.

5.6.4. Reading a Scene Description from a File.

In the previous examples the scene was described through specific OpenGL calls that transform and draw each object, as in

```

glTranslated(0.25, 0.42, 0.35);
glutSolidSphere(0.1, 15, 15); // draw a sphere

```

The objects in the scene were therefore “hard-wired” into the program. This way of specifying a scene is cumbersome and error-prone. It is a boon when the designer can specify the objects in a scene through a simple language, and place the description in a file. The drawing program becomes a (much simpler) general-purpose program: it reads a scene file at run-time and draws whatever objects are encountered in the file.

The Scene Description Language (SDL) described in Appendix 4 provides such a tool. We define a `Scene` class – also described in Appendix 4, and available on the book’s web site – that supports the reading of an SDL file and the drawing of the objects described in the file. It is very simple to use the `Scene` class in an application: a global `Scene` object is created:

```
Scene scn; // create a scene object
```

and the `read()` method of the class is called to read in a scene file:

```
scn.read("simple.dat"); // read the scene file & build an object list
```

Figure 5.64 shows the data structure for the `scn` object, after the following simple SDL file has been read.

Figure 5.64. An object of the Scene class.

```
! simple.dat: a simple scene having one light and four shapes
background 0 0 1          ! give the scene a blue background
light 2 9 8 1 1 1         ! put a white light at (9, 9, 9)
diffuse .9 .1 .1          ! make the following objects reddish
translate 3 5 -2 sphere    ! put a sphere at 3 5 -2
translate -4 -6 8 cone     ! put a cone in the scene
translate 1 1 1 cube       ! add a cube
diffuse 0 1 0              ! make the following objects green
translate 40 5 2 scale .2 .2 .2 sphere !add a tiny sphere
```

The first line is a comment; comments extend to the end of the line. This scene has a bright blue background color (*red, green, blue*) = (0, 0, 1), a bright white (1, 1, 1) light situated at (2, 9, 8), and four objects: two spheres, a cone and a cube. The `light` field points to the list of light sources, and the `obj` field points to the object list. Each shape object has its own affine transformation M that describes how it is scaled, rotated, and positioned in the scene. It also contains various data fields that specify its material properties, that are important when we want to render it faithfully as discussed in Chapter 8. Only the `diffuse` field is shown in the figure.

Once the light list and object list have been built, the application can render the scene:

```
scn.makeLightsOpenGL(),
scn.drawSceneOpenGL(); // render the scene using OpenGL
```

The first instruction passes a description of the light sources to OpenGL. The second uses the method `drawSceneOpenGL()` to draw each object in the object list. The code for this method is very simple:

```
void Scene :: drawSceneOpenGL()
{
    for(GeomObj* p = obj; p ; p = p->next)
        p->drawOpenGL(); // draw it
}
```

It moves a pointer through the object list, calling `drawOpenGL()` for each object in turn. It's a nice example of using polymorphism which is a foundation-stone of object-oriented programming: Each different shape "knows" how to draw itself: it has a method `drawOpenGL()` that calls the appropriate routine for that shape. So when `p` points to a sphere the `drawOpenGL()` routine for a sphere is called automatically; when `p` points to a cone the `drawOpenGL()` routine for a cone is called, etc. Figure 5.65 shows the methods for the `Sphere` and `Cone` classes; they differ only in the final OpenGL drawing routine that is called. Each first passes the object's material properties to OpenGL, then updates the modelview matrix with the object's specific affine transformation. The original modelview matrix is pushed and later restored, to protect it from being affected after this object has been drawn.

```
void Sphere :: drawOpenGL()
{
    tellMaterialsGL(); //pass material data to OpenGL
    glPushMatrix();
    glMultMatrixf(transf.m); // load this object's matrix
    glutSolidSphere(1.0,10,12); // draw a sphere
    glPopMatrix();
}
void Cone :: drawOpenGL()
```

```

{
    tellMaterialsGL(); //pass material data to OpenGL
    glPushMatrix();
    glMultMatrixf(transf.m); // load this object's matrix
    glutSolidCone(1.0, 1.0, 10, 12); // draw a cone
    glPopMatrix();
}

```

Figure 5.65. the drawOpenGL() methods for two shapes.

Figure 5.66 shows the program that reads an SDL file and draws the scene. It is very short, (but of course the code for the classes Scene, Shape, etc. must be loaded as well). It reads the particular SDL file myScene1.dat, which recreates the same scene as in Figure 5.63. Note that by simply changing the SDL file that is read this program can draw *any* scene described in SDL, without any changes in its code.

```

#include "basicStuff.h"
#include "Scene.h"
//##### GLOBALS #####
Scene scn; // construct the scene object
//<<<<<<<<<<< displaySDL >>>>>>>>>>>>>>>>>
void displaySDL(void)
{
    glMatrixMode(GL_PROJECTION); //set the camera
    glLoadIdentity();
    double winHt = 1.0; // half-height of the window
    glOrtho(-winHt*64/48.0, winHt*64/48.0, -winHt, winHt, 0.1, 100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(2.3, 1.3, 2, 0, 0.25, 0, 0.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT); // clear screen
    scn.drawSceneOpenGL();
} // end of display
//<<<<<<<<<<<< main >>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("read and draw an SDL scene");
    glutDisplayFunc(displaySDL);
    glShadeModel(GL_SMOOTH);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_NORMALIZE);
    glViewport(0, 0, 640, 480);
    scn.read("myScene1.dat"); //read the SDL file and build the objects
    glEnable(GL_LIGHTING);
    scn.makeLightsOpenGL(); // scan the light list and make OpenGL lights
    glutMainLoop();
}

```

Figure 5.66. Drawing a scene read in from an SDL file.

The SDL file that describes the scene of Figure 5.63 is shown in Figure 5.67. It defines the jack shape of nine spheres by first defining a jackPart and then using it three times, as explained in Appendix 4. Similarly a leg of the table is first defined as a unit, and then used four times.

```

! - myScene1.dat
light 20 60 30 .7 .7 .7 !put a light at (20,60,30),color:(.7, .7, .7)
ambient .7 .7 .7 ! set material properties for all of the objects
diffuse .6 .6 .6
specular 1 1 1
exponent 50

def jackPart{ push scale .2 .2 1 sphere pop

```

```

push translate 0 0 1.2 scale .2 .2 .2 sphere pop
push translate 0 0 -1.2 scale .2 .2 .2 sphere pop
}

def jack{ push use jackPart
rotate 90 0 1 0 use jackPart
rotate 90 1 0 0 use jackPart pop
}

def wall{push translate 1 .01 1 scale 1 .02 1 cube pop}
def leg {push translate 0 .15 0 scale .01 .15 .01 cube pop}

def table{
push translate 0 .3 0 scale .3 .01 .3 cube pop !table top
push
translate .275 0 .275 use leg
translate 0 0 -.55 use leg
translate -.55 0 .55 use leg
translate 0 0 -.55 use leg pop
}
!now add the objects themselves
push translate .4 .4 .6 rotate 45 0 0 1 scale .08 .08 .08 use jack pop
push translate .25 .42 .35 scale .1 .1 .1 sphere pop
push translate .6 .38 .5 rotate 30 0 1 0 scale .08 .08 .08 teapot pop
push translate 0.4 0 0.4 use table pop

use wall
push rotate 90 0 0 1 use wall pop
push rotate -90 1 0 0 use wall pop

```

Figure 5.67. The SDL file to create the scene of Figure 5.63.

With a scene description language like SDL available, along with tools to read and parse it, the scene designer can focus on creating complex scenes without having to work at the application code level. The scene being developed can be edited and tested again and again until it is right. The code developer puts the initial effort into constructing an application that can render any scene describable in SDL.

5.7. Summary of the Chapter.

Affine transformations are a staple in computer graphics, for they offer a unified tool for manipulating graphical objects in important ways. A designer always needs to scale, orient, and position objects in order to compose a scene as well as an appropriate view of the scene, and affine transformations make this simple to manage in a program.

Affine transformations convert one coordinate frame into another, and when homogeneous coordinates are used, an affine transformation is captured in a single matrix form. A sequence of such transformations can be combined into a single transformation whose matrix is simply the product of the individual transformation matrices. Significantly, affine transformations preserve straightness, so the image of a line is another line, and the image of a plane is a plane. This vastly simplifies working with lines and planes in a program, where one benefits from the simple representations of a line (its two endpoints) and a plane (by three points or four coefficients). In addition, parallelism is preserved, so that parallelograms map to parallelograms, and in 3D parallelepipeds map to parallelepipeds. This makes it simpler to visualize the geometric effects of affine transformations.

Three dimensional affine transformations are much more complex than their 2D counterparts, particularly when it comes to visualize a combination of rotations. A given rotation can be viewed as three elementary rotations through Euler angles, or a rotation about some axis, or simply as a matrix that has special properties. (Its columns are orthogonal unit vectors). It is often important to move between these different forms.

OpenGL and other graphics packages offer powerful tools for manipulating and applying transformations. In OpenGL all points are passed through several transformations, and the programmer can exploit this to define and

manipulate a “camera”, as well as to size and position different objects into a scene. Two of the matrices used by OpenGL (the modelview and viewport transformations) define affine transformations, whereas the projection matrix normally defines a perspective transformation, to be examined thoroughly in Chapter 7. OpenGL also maintains a stack of transformations, which make it easy for the scene designer to control the dependency of one object’s position on that of another, and to create objects that are composed of several related parts.

The SDL language, along with the Scene and Shape classes, make it much simpler to separate programming issues from scene design issues. An application is developed once that can draw any scene described by a list of light sources and a list of geometric objects. This application is then used over and over again with different scene files. A key task in the scene design process is applying the proper geometric transformations to each object. Since a certain amount of trial and error is usually required, it is convenient to be able to express these transformations in a concise and readable way.

The next section presents a number of Case Studies that elaborate on the main ideas of the chapter and suggest ways to practice with affine transformations in a graphics program. These range from plunging deeper into the theory of transformations to actual modeling and rendering of objects such as electronic CAD circuits and robots.

5.8 Case Studies.

Case study 5.1. Doing your own transforming by the CT in Canvas.

(Level of Effort: II). It’s easy to envision situations where you must implement the transformation mechanism yourself, rather than rely on OpenGL to do it. In this Case Study you add the support of a current transformation to the Canvas class for 2D drawing. This involves writing several functions, those to initialize and alter the *CT* itself:

```
void Canvas:: initCT(void);    // init CT to unit transformation
void Canvas:: scale2D(double sx, double sy);
void Canvas:: translate2D(double dx, double dy);
void Canvas:: rotate2D(double angle);
```

as well as others that are incorporated into `moveTo()` and `lineTo()` so that all points sent to them are “silently” transformed before being used. For extra benefit add the stack mechanism for the CT as well, along with functions `pushCT()` and `popCT()`. Exercise your new tools on some interesting 2D modeling and drawing examples.

Case Study 5.2. Draw the star of Fig 5.39. using multiple rotations.

(Level of Effort: I). Develop a function that draws the polygon in figure 5.39b that is one fifth of the star. Use it with rotation transformations to draw the whole star.

Case Study 5.3. Decomposing a 2D Affine Transformation.

(Level of Effort: II). We have seen that some affine transformations can be expressed as combinations of others. Here we “decompose” any 2D affine transformation into a combination of scalings, rotations, and translations. We also show that a rotation can be performed by three successive shears, which gives rise to a very fast arc drawing routine. Some results on what lurks in any 3D affine transformation are also discussed. We just summarize results here.

Because an affine transformation is a linear transformation followed by an offset we omit the translation part of the affine transformation, and focus on what a linear transformation is. So we use 2 by 2 matrices here for simplicity.

a). Two 2D linear transformations.

Consider the 2 by 2 matrix M that represents a 2D linear transformation. Matrix M can always be factored into a rotation, a scaling , and a shear. Call the four scalars in M by the names a, b, c , and d for brevity. Verify by direct multiplication that M is the product of the three matrices [martin82]:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{ac+bd}{R^2} & 1 \end{pmatrix} \begin{pmatrix} R & 0 \\ 0 & \frac{ad-bc}{R} \end{pmatrix} \begin{pmatrix} \frac{a}{R} & \frac{b}{R} \\ -\frac{b}{R} & \frac{a}{R} \end{pmatrix} \quad (5.39)$$

where $R = \sqrt{a^2 + b^2}$. The leftmost matrix on the right hand side is recognized as a shear, the middle one as a scaling, and the rightmost one as a rotation (why?).

Thus *any* 2D affine transformation is a rotation followed by a scaling followed by a shear followed by a translation.

An alternative decomposition, based on the so-called “Gram-Schmidt process”, is discussed in Case Study 5.5.

Example 5.8.1. Decompose the matrix $M = \begin{pmatrix} 4 & -3 \\ 2 & 7 \end{pmatrix}$ into the product of shear, scaling, and rotation matrices.

Solution: Check the following, obtained by direct substitution in Equation 5.39:

$$M = \begin{pmatrix} 4 & -3 \\ 2 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -13/25 & 1 \end{pmatrix} \begin{pmatrix} 5 & 0 \\ 0 & 34/5 \end{pmatrix} \begin{pmatrix} 4/5 & -3/5 \\ 3/5 & 4/5 \end{pmatrix}$$

b). A 2D Rotation is three shears.

Matrices can be factored in different ways. In fact a rotation matrix can be written as the product of three shear matrices! [Paeth90] This leads to a particularly fast method for performing a series of rotations, as we will see.

Equation 5.40 shows a rotation represented as three successive shears. It can be verified by direct multiplication. It demonstrates that a rotation is a shear in y followed by a shear in x , followed by a repetition of the first shear.

$$\begin{pmatrix} \cos(a) & \sin(a) \\ -\sin(a) & \cos(a) \end{pmatrix} = \begin{pmatrix} 1 & \tan(a/2) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\sin(a) & 1 \end{pmatrix} \begin{pmatrix} 1 & \tan(a/2) \\ 0 & 1 \end{pmatrix} \quad (5.40)$$

Calling $T = \tan(a/2)$, and $S = \sin(a)$ show that we can write the sequence of operations that rotate point (x, y) as¹⁵:

$$\begin{aligned} x' &= T * y + x; && \{\text{first shear}\} \\ y' &= y; \end{aligned}$$

$$\begin{aligned} x'' &= x'; && \{\text{second shear}\} \\ y'' &= y' - S * x'; \end{aligned}$$

$$\begin{aligned} x''' &= T * y'' + x'; && \{\text{third shear}\} \\ y''' &= y''; \end{aligned}$$

using primes to distinguish new values from old. But operations like $x'' = x'$ do nothing, so the primes are unnecessary and this sequence reduces to:

$$\begin{aligned} x &= x + T * y; \\ y &= y - S * x; && \{\text{actual operations for the three shears}\} \\ x &= x + T * y; \end{aligned}$$

¹⁵Note that $\sin(a)$ may be found quickly from $\tan(a/2)$ by one multiplication and one division (see Appendix 2): $S = (2T)/(1 + T^2)$

If we have only one rotation to perform there is no advantage to going about it this way. However, it becomes very efficient if we need to do a succession of rotations through the same angle. Two places where we need to do this are 1). calculating the vertices of an n -gon — which are equispaced points around a circle, and 2). computing the points along the arc of a circle.

Figure 5.68 shows a code fragment for calculating the positions of n points around a circle. It loads the successive values $(\cos(i * 2\pi/n + b), \sin(i * 2\pi/n + b))$ into the array of points $p[]$.

```
T = tan(PI/n);           // tangent of half angle16
S = 2 * T/(1 + T * T); // sine of angle
p[0].x = sin(b);        // initial angle, vertex 0
p[0].y = cos(b);
for (int i = 1; i < n; i++)
{
    p[i].y = p[i-1].x * T + p[i-1].y; //1st shear
    p[i].x = p[i-1].x - S * p[i-1].y; //2nd shear
    p[i].y = p[i-1].x * T + p[i-1].y; //3rd shear
}
```

Figure 5.68. Building the vertices of an n -gon efficiently.

Figure 5.69 shows how to use shears to build a fast arc drawer. It does the same job as `drawArc()` in Chapter 3, but much more efficiently, since it avoids the repetitive computation of `sin()` and `cos()`.

```
void drawArc2(RealPoint c, double R,
              double startangle, double sweep) // in degrees
{
#define n 30
#define RadPerDeg .01745329
    double delang = RadPerDeg * sweep / n;
    double T = tan(delang/2);           // tan. of half angle
    double S = 2 * T/(1 + T * T);     // sine of half angle
    double snR = R * sin(RadPerDeg * startangle);
    double csR = R * cos(RadPerDeg * startangle);
    moveTo(c.x + csR, c.y + snR);
    for(int i = 1; i < n; i++)
    {
        snR += T * csR;      // build next snR, csR pair
        csR -= S * snR;
        snR += T * csR;
        lineTo(c.x + csR, c.y + snR);
    }
}
```

Figure 5.69. A fast arc drawer.

Develop a test program that uses this routine to draw arcs. Compare its efficiency with arc drawers that compute each vertex using trigonometry.

c). Is a shear “fundamental”?

One sometimes reads in the graphics literature that the fundamental elementary transformations are the rotation, scaling, and translation; shears seem to be second class citizens. This attitude may stem from the fact that any shear can be decomposed into a combination of rotations and scalings. The following equation may be verified by multiplying out the three matrices on the right [Greene90]:

$$\begin{pmatrix} 1 & 0 \\ a - \frac{1}{a} & 1 \end{pmatrix} = \frac{1}{\sqrt{1+a^2}} \begin{pmatrix} 1 & -a \\ a & 1 \end{pmatrix} \begin{pmatrix} a & 0 \\ 0 & 1/a \end{pmatrix} \begin{pmatrix} a & 1 \\ -1 & a \end{pmatrix} \frac{1}{\sqrt{1+a^2}} \quad (5.41)$$

¹⁶ Note: Some C environments do not support `tan()` directly. Use `sin()/cos()` - of course first checking that this denominator is not zero.

The middle matrix is a scaling, while the outer two matrices (when combined with the scale factors shown) are rotations. For the left-hand rotation associate $1/\sqrt{1+a^2}$ with $\cos(\alpha)$ and $-a/\sqrt{1+a^2}$ with $\sin(\alpha)$ for some angle α . Thus $\tan(\alpha) = -a$. Similarly for the right-hand rotation associate $\cos(\beta)$ and $\sin(\beta)$ for angle β , where $\tan(\beta) = 1/a$. Note that α and β are related: $\beta = \alpha + \pi/2$ (why?).

Using this decomposition, write the shear in Equation 5.41 as a “rotation then a scaling then a rotation” to conclude that every 2D affine transformation is the following sequence of elementary transformations:

$$\text{Any affine transformation} = \text{Scale} * \text{Rotation} * \text{Scale} * \text{Rotation} * \text{Translation} \quad (5.42)$$

Practice Exercises

5.8.1. A “golden” decomposition. Consider the special case of a “unit shear” where the term $a - 1/a$ in Equation 5.3.5 is 1. What value must a have? Determine the two angles α and β associated with the rotations.

Solution: Since a must satisfy $a = 1 + 1/a$, a is the golden ratio ϕ !. Thus

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} \phi & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\beta) & \sin(\beta) \\ -\sin(\beta) & \cos(\beta) \end{pmatrix} \quad (5.43)$$

where $\alpha = \tan^{-1}(\phi) = 58.28^\circ$ and $\beta = \tan^{-1}(1/\phi) = 31.72^\circ$.

5.8.2. Unit Shears. Show that any shear in x contains within it a unit shear. Decompose the shear given by

$$\begin{pmatrix} 1 & 0 \\ h & 1 \end{pmatrix}$$

into the product of a scaling, a unit shear, and another scaling.

5.8.3. Seeing It Graphically. Drawing upon the ideas in Exercise 5.8.1, draw a rectangle on graph paper; rotate it through -58.28° ; scale it by $(\phi, 1/\phi)$; and finally rotate it through 31.72° . Sketch each intermediate result, and show that the final result is the same parallelogram obtained when the original rectangle undergoes a unit shear.

5.8.4. Decompose a Transformation. Decompose the transformation

$$Q_x = 3P_x - 2P_y + 5$$

$$Q_y = 4P_x + P_y - 6$$

into a product of rotations, scalings, and translations.

5.8.5. One reflection can always be in the x-axis. Show that a rotation through angle A about the origin may always be effected by a reflection in the x -axis followed by a reflection in a line at angle $A/2$.

5.8.6. Isometries. A particularly important family of affine transformations in the study of symmetry are the **isometries** (“same measure”), since they don’t alter the distance between two points and their images. For any two points P and Q the distance $|T(P) - T(Q)|$ is the same as the distance $|P - Q|$ when $T()$ is an isometry. Show that if $T()$ is affine with matrix M , then T is an isometry if and only if each row of M considered as a vector is of unit length, and the two rows are orthogonal.

Solution: Call $r = P - Q$. Then $|T(P) - T(Q)| = |PM + d - QM - d| = \|rM\| = |(r_{x1}m_{11} + r_{y1}m_{21}, r_{x2}m_{12} + r_{y2}m_{22})|$.

Equating $\|rM\|^2$ to $\|r\|^2$ requires $m_{11}^2 + m_{12}^2 = 1$, $m_{21}^2 + m_{22}^2 = 1$, and $m_{11}m_{21} + m_{22}m_{12} = 0$, as claimed.

5.8.7. Ellipses Are Invariant. Show that ellipses are invariant under an affine transformation. That is, if E is an ellipse and if T is an affine transformation, then the image $T(E)$ of the points in E also makes an ellipse. **Hint to Solution:** Any affine is a combination of rotations and scalings. When an ellipse is rotated it is still an ellipse, so only the non-uniform scalings could possibly destroy the ellipse property. So it is necessary only to show that an ellipse, when subjected to a non-uniform scaling, is still an ellipse.

5.8.8. What else is invariant? Consider what class of shapes (perhaps a broader class than ellipses) is invariant to affine transformations. If the equation $f(x,y)=0$ describes a shape, show that after transforming it with

transformation T , the new shape is described by all points that satisfy $g(x,y) = f(T^{-1}(x,y)) = 0$. Then show the details of this form when T is affine. Finally, try to describe the largest class of shapes which is preserved under affine transformations.

Case Study 5. 4. Generalized 3D Shears.

(Level of Effort:II) A shear can be more general than those discussed in Section 5.???. As suggested by Goldman [goldman 91] the ingredients of a shear are:

- A plane through the origin having unit normal vector \mathbf{m} ;
- A unit vector \mathbf{v} lying in the plane (thus perpendicular to \mathbf{m});
- An angle ϕ .

Then as shown in Figure 5.70 point P is sheared to point Q by shifting it in direction \mathbf{v} a certain amount. The amount is proportional to both the distance at which P lies from the plane, and to $\tan \phi$. Goldman shows that this shear has the matrix representation:

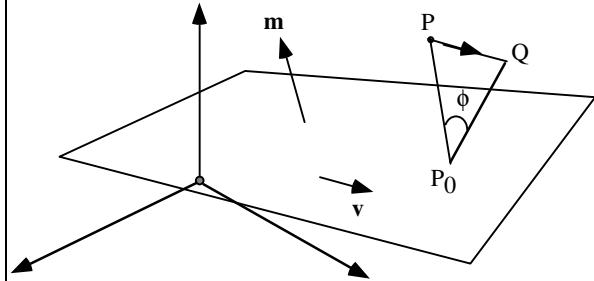


Figure 5.70. Defining a shear in 3D.

$$M = I + \tan(\phi) \begin{pmatrix} m_x v_x & m_x v_y & m_x v_z \\ m_y v_x & m_y v_y & m_y v_z \\ m_z v_x & m_z v_y & m_z v_z \end{pmatrix} \quad (5.44)$$

where I is the 3 by 3 identity matrix, and an offset vector of $\mathbf{0}$. Some details of the derivation are given in the exercises.

Example 5.8.2: Find the shear associated with the plane having unit normal vector $\mathbf{m} = (1,1,1)/\sqrt{3} = (0.577, 0.577, 0.577)$, unit vector $\mathbf{v} = (0, 0.707, -0.707)$, and angle $\phi = 30^\circ$. **Solution:** Note that \mathbf{v} lies in the plane as required (why?). Applying Equation 9.9.5 we obtain:

$$M = I + 0.577 \begin{pmatrix} 0 & 0.408 & -0.408 \\ 0 & 0.408 & -0.408 \\ 0 & 0.408 & -0.408 \end{pmatrix} = \begin{pmatrix} 1 & 0.235 & -0.235 \\ 0 & 1.235 & -0.235 \\ 0 & 0.235 & 0.764 \end{pmatrix}$$

Derive the shear matrix. We want to express the point Q in Figure 5.73 in terms of P and the ingredients of the shear. The (signed) distance of P from the plane is given by $P \cdot \mathbf{m}$ (considering P as a position vector pinned to the origin). a). Put the ingredients together to obtain:

$$Q = P + (P \cdot \mathbf{m}) \tan(\phi) \mathbf{v} \quad (5.45)$$

Note that points “on the other side” of the plane are sheared in the opposite direction, as we would expect.

Now we need to the second term as P times some matrix. Use Appendix 2 to show that $P \cdot \mathbf{m}$ is $P \mathbf{m}^T$, and therefore that $Q = P (I + \tan(\phi) \mathbf{m}^T \mathbf{v})$, and that the shear matrix is $(I + \tan(\phi) \mathbf{m}^T \mathbf{v})$. Show that the matrix $\mathbf{m}^T \mathbf{v}$ has the form:

$$\mathbf{m}^T \mathbf{v} = \begin{pmatrix} m_x \\ m_y \\ m_z \end{pmatrix} (v_x, v_y, v_z) = \begin{pmatrix} m_x v_x & m_x v_y & m_x v_z \\ m_y v_x & m_y v_y & m_y v_z \\ m_z v_x & m_z v_y & m_z v_z \end{pmatrix} \quad (5.46)$$

This is called the **outer product** (or **tensor product**) of \mathbf{m} with \mathbf{v} .

Practice Exercises.

5.8.9. Consistency with a simple shear. Express the matrix for the shear associated with direction $\mathbf{v} = (1, 0, 0)$ and the plane having normal vector $(0, 1, 0)$. Show that this is a form of the elementary shear matrix given in Equation 5.45, for any angle ϕ .

5.8.10. Find the shear. Compute the shear matrix for the shear that uses a normal $\mathbf{m} = 0.577(1, -1, 1)$, direction vector $\mathbf{v} = 0.707(1, 1, 0)$, and angle $\phi = 45^\circ$. Sketch the vectors and plane involved, and sketch how a cube centered at the origin with edges aligned with the coordinate axes is affected by the shear.

5.8.11. How Is a Three-Dimensional Shear Like a Two-Dimensional Translation in Homogeneous Coordinates?

Consider the specific shear:

$$Q = P \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t & s & 1 \end{pmatrix} \quad (5.47)$$

which alters the 3D point P by shearing it along x with factor t owing to z and along y with factor s owing to z . If P lies in the $z = 1$ plane with coordinates $P = (p_x, p_y, 1)$, show that P is transformed into $(p_x + t, p_y + s, 1)$. Hence it is simply shifted in x by amount t and in y by amount s . Show that for any point in the $z = 1$ plane, this shear is equivalent to a translation. Show that the matrix in Equation 5.48 is identical to the homogeneous coordinate form for a pure translation in two dimensions. This correspondence helps reinforce how homogeneous coordinates operate.

Case Study 5.5. Rotation about an Axis: The Constructive Approach.

(Level of Effort: II) You are asked to fill in the details in the following derivation of the rotation matrix $R_{\mathbf{u}}(\beta)$ of Equation 9.9.15. For simplicity we assume \mathbf{u} is a unit vector: $\|\mathbf{u}\| = 1$. Denote the position vector based on P by the name \mathbf{p} , so $\mathbf{p} = P - O$ where O is the origin of the coordinate system. Now project \mathbf{p} onto \mathbf{u} to obtain vector \mathbf{h} , as shown in Figure 9.9.8a.

- a). Show it has the form $\mathbf{h} = (\mathbf{p} \cdot \mathbf{u}) \mathbf{u}$. Define two perpendicular vectors \mathbf{a} and \mathbf{b} that lie in the plane of rotation: $\mathbf{a} = \mathbf{p} - \mathbf{h}$, and $\mathbf{b} = \mathbf{u} \times \mathbf{a}$
- b). Show that they are perpendicular, they have the same length, they both lie in the plane, and that $\mathbf{b} = \mathbf{u} \times (\mathbf{p} - \mathbf{h})$ simplifies to $\mathbf{u} \times \mathbf{p}$. This effectively establishes a 2D coordinate system in the plane of rotation. Now look onto the plane of rotation, as in Figure 9.9.8b. The rotation rotates \mathbf{a} to $\mathbf{a}' = \cos \beta \mathbf{a} + \sin \beta \mathbf{b}$, so the rotated point Q is given by:

$$Q = \mathbf{h} + \cos \beta \mathbf{a} + \sin \beta \mathbf{b}, \text{ or using the expressions for } \mathbf{a} \text{ and } \mathbf{b},$$

$$Q = \mathbf{p} \cos \beta + (1 - \cos \beta)(\mathbf{p} \cdot \mathbf{u}) \mathbf{u} + \sin \beta (\mathbf{u} \times \mathbf{p}) \quad (5.48)$$

This is a rather general result, showing how the rotated point Q can be decomposed into portions along \mathbf{h} and along two orthogonal axes lying in the plane of rotation.

This form for Q hardly looks like the multiplication of P by some matrix, but it is because each of the three terms is linear in \mathbf{p} . Convert each of the terms to the proper form as follows:

- c). Replace \mathbf{p} with P to obtain immediately $\mathbf{p}(\cos \beta) = P(\cos \beta) I$ where I is the 3 by 3 identity matrix.
- d). Use the result (Appendix 2) that a dot product $\mathbf{p} \cdot \mathbf{c}$ can be rewritten as P times a matrix: $P \mathbf{u}^T$, to show that $(\mathbf{p} \cdot \mathbf{u}) \mathbf{u} = P \mathbf{u}^T \mathbf{u}$ where $\mathbf{u}^T \mathbf{u}$ is an outer product similar to Equation 9.9.7.
- e). Use the fact developed in Appendix 2 that a cross product $\mathbf{u} \times \mathbf{p}$ can also be written as P times a matrix to show that $\mathbf{u} \times \mathbf{p} = P \text{Cross}(\mathbf{u})$ where matrix $\text{Cross}(\mathbf{c})$ is:

$$Cross(\mathbf{u}) = \begin{pmatrix} 0 & u_z & -u_y \\ -u_z & 0 & u_x \\ u_y & -u_x & 0 \end{pmatrix} \quad (5.49)$$

f). Put these together to obtain the matrix

$$R_u(\beta) = \cos \beta I + (1 - \cos \beta) \mathbf{u}^T \mathbf{u} + \sin \beta Cross(\mathbf{u}) \quad (5.50)^{17}$$

M is therefore the sum of three weighted matrices. It surely is easier to build than the product of five matrices as in the classic route.

g). Write this out to obtain Equation 5.33.

Case Study 5.6. Decomposing 3D Affine Transformations.

(Level of Effort:III) This Case Study looks at several broad families of affine transformations.

What is in a 3D Affine Transformation?

Once again we ignore the translation portion of an affine transformation and focus on the linear transformation part represented by 3×3 matrix M . What kind of transformations are “imbedded” in it? It is somewhat more complicated. Goldman[GEMSIII, p.108] shows that every such M is the product of a scaling S , a rotation R , and two shears H_1 and H_2 .

$$M = S R H_1 H_2 \quad (5.51)$$

Every 3D affine transformation, then, can be viewed as this sequence of elementary operations, followed by a translation. In Case Study 5.??? we explore the mathematics behind this form, and see how an actual decomposition might be carried out.

Useful Classes of Transformations.

It's useful to categorize affine transformations, according to what they “do” or “don't do” to certain properties of an object when it is transformed. We know they always preserve parallelism of edges of an object, but which transformations also preserve the length of each edge, and which ones preserve the angles between each pair of edges?

1). Rigid Body Motions.

It is intuitively clear that translating an object, or rotating it, will not change its shape or size. In addition, reflecting it about a plane has no effect on its shape or size. Since the shape is not affected by any one of these transformations alone, it is also not affected by an arbitrary composition of them. We denote by

$$T_{\text{rigid}} = \{\text{rotations, reflections, translations}\}$$

the collection of all affine transformations that consist of any sequence of rotations, reflections, and translations. These are known classically as the **rigid body motions**, since an object is moved rigidly from one position and orientation to another. Such transformations have *orthogonal* matrices in homogeneous coordinates. These are matrices for which the inverse is the same as the transpose:

$$\tilde{M}^{-1} = \tilde{M}^T$$

2). Angle-Preserving Transformations.

A uniform scaling (having equal scale factors $S_x = S_y = S_z$) expands or shrinks an object, but does so uniformly, so there is no change in the object's shape. Thus the angle between any two edges is unaffected. We can denote such a class as

¹⁷Goldman [gold90] reports the same form for M , and gives compact results for several other complex transformations.

$$T_{\text{angle}} = \{\text{rotations, reflections, translations, uniform scalings}\}$$

This class is larger than the rigid body motions, because it includes uniform scaling. It is an important class because, as we see in Chapter ???, lighting and shading calculations depend on the dot products between various vectors. If a certain transformation does not alter angles then it does not alter dot products, and lighting calculations can take place in either the transformed or the untransformed space.

(Estimate of time required: three hours.) Given a 3D affine transformation $\{M, \mathbf{d}\}$ we wish to see how it is composed of a sequence of elementary transformations. Following Goldman [GEMS III, p. 108] we develop the steps required to decompose the 3 by 3 matrix M into the product of a scaling S , a rotation R , and two shears H_1 and H_2 :

$$M = S R H_1 H_2 \quad (5.52)$$

You are asked to verify each step along the way, and to develop a routine that will produce the individual matrices S, R, H_1 and H_2 .

Suppose the matrix M has rows \mathbf{u}, \mathbf{v} , and \mathbf{w} , each a 3D vector:

$$M = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{w} \end{pmatrix}$$

Goldman's approach is based on the classical Gram-Schmidt orthogonalization procedure, whereby the rows of M are combined in such a way that they become mutually orthogonal and of unit length. The matrix composed of these rows is therefore orthogonal (see Practice Exercise 9.9.14) and so represents a rotation (or a rotation with a reflection). Goldman shows that the orthogonalization process is in fact two shears. The rest is detail.

The steps are as follows. Carefully follow each step, and do each of the tasks given in brackets.

1. Normalize \mathbf{u} to $\mathbf{u}^* = \mathbf{u}/S_1$, where $S_1 = \|\mathbf{u}\|$.
2. Subtract a piece of \mathbf{u}^* from \mathbf{v} so that what is left is orthogonal to \mathbf{u}^* : Call $\mathbf{b} = \mathbf{v} - d\mathbf{u}^*$ where $d = \mathbf{v} \cdot \mathbf{u}^*$. [Show that $\mathbf{b} \cdot \mathbf{u}^* = 0$.]
3. Normalize \mathbf{b} : set $\mathbf{v}^* = \mathbf{b}/S_2$, where $S_2 = \|\mathbf{b}\|$.
4. Set up some intermediate values: $m = \mathbf{w} \cdot \mathbf{u}^*$ and $n = \mathbf{w} \cdot \mathbf{v}^*$, $e = \sqrt{m^2 + n^2}$, and $\mathbf{r} = (m\mathbf{u}^* + n\mathbf{v}^*)/e$.
5. Subtract a piece of \mathbf{r} from \mathbf{w} so that what is left is orthogonal to both \mathbf{u}^* and \mathbf{v}^* : Call $\mathbf{c} = \mathbf{w} - e\mathbf{r}$. [Show that $\mathbf{c} \cdot \mathbf{u}^* = \mathbf{c} \cdot \mathbf{v}^* = 0$.]
6. Normalize \mathbf{c} : set $\mathbf{w}^* = \mathbf{c}/S_3$ where $S_3 = \|\mathbf{c}\|$.
7. The matrix

$$R = \begin{pmatrix} \mathbf{u}^* \\ \mathbf{v}^* \\ \mathbf{w}^* \end{pmatrix}$$

is therefore orthogonal, and so represents a rotation. (Compute its determinant: if it is -1 then simply replace \mathbf{w}^* with $-\mathbf{w}^*$.)

8. Define the shear matrix $H_1 = I + \frac{d}{S_2}(\mathbf{v}^* \otimes \mathbf{u}^*)$ (see Practice exercise 9.9.1), where

$(\mathbf{v}^* \otimes \mathbf{u}^*) = (\mathbf{v}^*)^T \mathbf{u}^*$ is the outer product of \mathbf{v}^* and \mathbf{u}^* .

9. Define the shear matrix $H_2 = I + \frac{e}{S_3}(\mathbf{w}^* \otimes \mathbf{r})$, where $(\mathbf{w}^* \otimes \mathbf{r}) = (\mathbf{w}^*)^T \mathbf{r}$.

10. [Show that $\mathbf{u}^* H_1 = \mathbf{u}^*$, $\mathbf{v}^* H_1 = \mathbf{v}^* + d\mathbf{u}^*/S_2 = \mathbf{v}/S_2$, and $\mathbf{w}^* H_1 = \mathbf{w}^*$. First show the property of the outer product that for any vectors \mathbf{a}, \mathbf{b} , and \mathbf{c} : $\mathbf{a}(\mathbf{b} \otimes \mathbf{c}) = (\mathbf{a} \cdot \mathbf{b})\mathbf{c}$. Then use this property, along with the orthogonality of \mathbf{u}^* , \mathbf{v}^* and \mathbf{w}^* , to show the three relations.]

11. [Show that $\mathbf{u}^* H_2 = \mathbf{u}^*$, $\mathbf{v}^* H_2 = \mathbf{v}^*$, $\mathbf{w}^* H_2 = \mathbf{w}^* + e\mathbf{r}/S_3 = \mathbf{w}/S_3$.]

12. [Put these together to show that $M = SRH_1H_2$.] S is defined to be

$$S = \begin{pmatrix} S_1 & 0 & 0 \\ 0 & S_2 & 0 \\ 0 & 0 & S_3 \end{pmatrix}$$

Note that the decomposition is not unique, since the vectors \mathbf{u} , \mathbf{v} , and \mathbf{w} could be orthogonalized in a different order. For instance, we could first form \mathbf{w}^* as $\mathbf{w}/\|\mathbf{w}\|$, then subtract a piece of \mathbf{v} from \mathbf{w}^* to make a vector orthogonal to \mathbf{w}^* , then subtract a vector from \mathbf{u} to make a vector orthogonal to the other two.

Write the routine:

```
void decompose(DBL m[3][3], DBL S[3][3], DBL R[3][3], DBL H1[3][3], DBL  
H2[3][3])
```

where `DBL` is defined as `double`, that takes matrix `m` and computes the matrices `S`, `R`, `H1` and `H2` as described above. Test your routine on several matrices.

Other ways to decompose a 3D transformation have been found as well. See for instance [thomas, GEMS II, p. 320] and [Shoemake, GEMS IV p. 207].

Case Study 5.7. Drawing 3D scenes described by SDL.

(Level of Effort:II) Develop a complete application that uses the `Scene`, `Shape`, `Affine4`, etc. classes and supports reading in and drawing the scene described in an SDL file. To assist you, use any classes provided on the book's web site. Flesh out any `drawOpenGL()` methods that are needed. Develop a scene file that contains descriptions of the jack, a wooden chair, and several walls.

5.9. For Further Reading

There are some excellent books on transformations that supplement the treatment here. George Martin's TRANSFORMATION GEOMETRY [martin82] is superb, as is Yaglom's GEOMETRIC TRANSFORMATIONS [yaglom62]. Hoggar also provides a fine chapter on vectors and transformations [hoggar92]. Several of Blinn's engaging articles in JIM BLINN'S CORNER - A TRIP DOWN THE GRAPHICS PIPELINE [blinn96] provide excellent discussions of homogeneous coordinates and transformations in computer graphics.

CHAP 6. Modeling Shapes with Polygonal Meshes

Try to learn something about everything and everything about something.

T.H. Huxley

Goals of the Chapter

- To develop tools for working with objects in 3D space
- To represent solid objects using polygonal meshes
- To draw simple wireframe views of mesh objects

Preview

Section 6.1 gives an overview of the 3D modeling process. Section 6.2 describes polygonal meshes that allow you to capture the shape of complex 3D objects in simple data structures. The properties of such meshes are reviewed, and algorithms are presented for viewing them.

Section 6.3 describes families of interesting 3D shapes, such as the Platonic solids, the Buckyball, geodesic domes, and prisms. Section 6.4 examines the family of extruded or “swept” shapes, and shows how to create 3D letters for flying logos, tubes and “snakes” that undulate through space, and surfaces of revolution.

Section 6.5 discusses building meshes to model solids that have smoothly curved surfaces. The meshes approximate some smooth “underlying” surface. A key ingredient is the computation of the normal vector to the underlying surface at any point. Several interesting families of smooth solids are discussed, including quadric and superquadric surfaces, ruled surfaces and Coons patches, explicit functions of two variables, and surfaces of revolution.

The Case Studies at the end of the chapter explore many ideas further, and request that you develop applications to test things out. Some cases studies are theoretical in nature, such as deriving the Newell formula for computing a normal vector, and examining the algebraic form for quadric surfaces. Others are more practical, such as reading mesh data from a file and creating beautiful 3D shapes such as tubes and arches.

6.1 Introduction.

In this chapter we examine ways to describe 3D objects using *polygonal meshes*. Polygonal meshes are simply collections of polygons, or “faces”, which together form the “skin” of the object. They have become a standard way of representing a broad class of solid shapes in graphics. We have seen several examples, such as the cube, icosahedron, as well as approximations of smooth shapes like the sphere, cylinder, and cone (see Figure 5.65). In this chapter we shall see many more examples. Their prominence in graphics stems from the simplicity of using polygons. Polygons are easy to represent (by a sequence of vertices) and transform, have simple properties (a single normal vector, a well-defined inside and outside, etc.), and are easy to draw (using a polygon-fill routine, or by mapping texture onto the polygon).

Many rendering systems, including OpenGL, are based on drawing objects by drawing a sequence of polygons. Each polygonal face is sent through the graphics pipeline (recall Figure 5.56), where its vertices undergo various transformations, until finally the portion of the face that survives clipping is colored in, or “shaded”, and shown on the display device.

We want to see how to design complicated 3D shapes by defining an appropriate set of faces. Some objects can be perfectly represented by a polygonal mesh, whereas others can only be approximated. The barn of Figure 6.1a, for example, naturally has flat faces, and in a rendering the edges between faces should be visible. But the cylinder in Figure 6.1b is supposed to have a smoothly rounded wall. This roundness cannot be achieved using only polygons: the individual flat faces are quite visible, as are the edges between them. There are, however, rendering techniques that make a mesh like this *appear* to be smooth, as in Figure 6.1c. We examine the details of so-called Gouraud shading in Chapter 8.

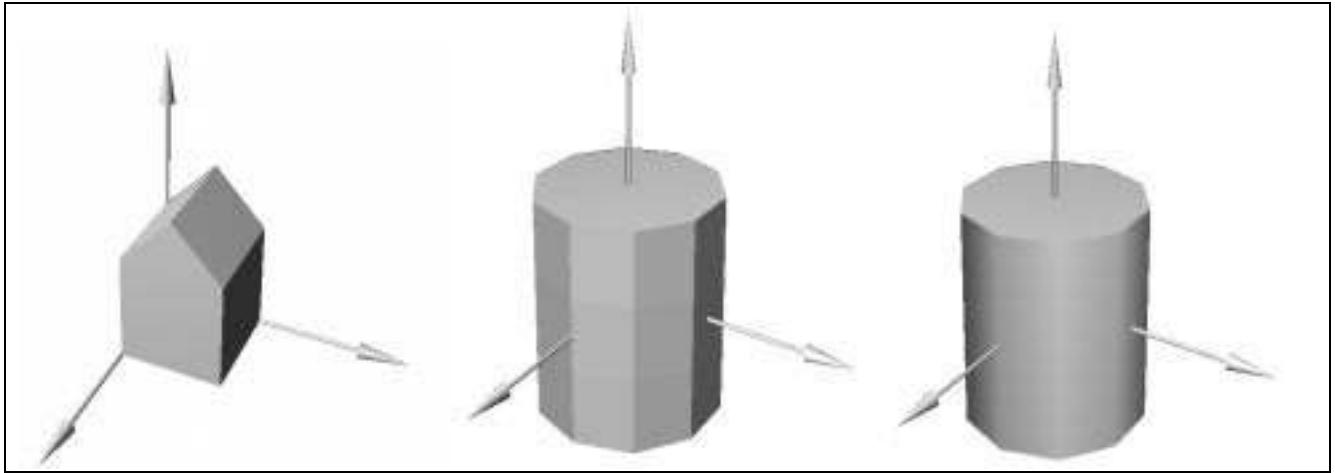


Figure 6.1. Various shapes modeled by meshes.

We begin by describing polygonal meshes in some generality, and seeing how to define and manipulate them in a program (with or without OpenGL). We apply these ideas to modeling polyhedra, which inherently have flat faces, and study a number of interesting families of polyhedra. We then tackle the problem of using a mesh to approximate a smoothly curved shape, and develop the necessary tools to create and manipulate such models.

6.2. Introduction to Solid Modeling with Polygonal Meshes.

I never forget a face, but in your case I'll make an exception.
Grouch Marx

We shall use meshes to model both solid shapes and thin “skins”. The object is considered to be **solid** if the polygonal faces fit together to enclose space. In other cases the faces fit together without enclosing space, and so they represent an infinitesimally thick surface. In both cases we call the collection of polygons a **Polygonal mesh** (or simply **mesh**).

A polygonal mesh is given by a list of polygons, along with information about the direction in which each polygon is “facing”. This directional information is often simply the **normal vector** to the plane of the face, and it is used in the shading process to determine how much light from a light source is scattered off the face. Figure 6.2 shows the normal vectors for various faces of the barn. As we examine in detail in Chapter 11, one component of the brightness of a face is assumed to be proportional to the cosine of the angle (shown as θ in the figure for the side wall of the barn) between the normal vector to a face and the vector to the light source. Thus the orientation of a surface with respect to light sources plays an important part in the final drawing.

Figure 6.2. The normal direction to a face determines its brightness.

Vertex normals versus face normals.

It turns out to be highly advantageous to associate a “normal vector” to each vertex of a face, rather than one vector to an entire face. As we shall see, this facilitates the clipping process as well as the shading process for smoothly curved shapes. For flat surfaces such as the wall of a barn, this means that each of the vertices V_1 , V_2 , V_3 , and V_4 that define the side wall of the barn will be associated with the *same* normal \mathbf{n}_1 , the normal vector to the side wall itself (see Figure 6.3a). But vertices of the front wall, such as V_5 , will use normal \mathbf{n}_2 . (Note that vertices V_1 and V_5 are located at the same point in space, but use different normals.)

Figure 6.3. Associating a “normal” with each vertex of each face.

For a smoothly curved surface such as the cylinder in Figure 6.3b, a different approach is used in order to permit shading that makes the surface appear smooth. Both vertex V_1 of face F_1 and vertex V_2 on face F_2 use the same normal \mathbf{n} , which is the vector perpendicular to the underlying smooth surface. We see how to compute this vector conveniently in Section 6.2.2.

6.2.1. Defining a polygonal mesh.

A polygonal mesh is a collection of polygons along with a normal vector associated with each vertex of each polygon. We begin with an example.

Example 6.2.1. The “basic barn”. Figure 6.4 shows a simple shape we call the “basic barn”. It has seven polygonal faces and a total of 10 vertices (each of which is shared by three faces). For convenience it has a square floor one unit on a side. (The barn would be scaled and oriented appropriately before being placed in a scene.) Because the barn is assumed to have flat walls there are only seven distinct normal vectors involved, the normal to each face as shown.



Figure 6.4. Introducing the basic barn.

There are various ways to store mesh information in a file or program. For the barn you could use a list of seven polygons, and list for each one where its vertices are located and where the normal for each of its vertices is pointing (a total of 30 vertices and 30 normals). But this would be quite redundant and bulky, since there are only 10 distinct vertices and seven distinct normals.

A more efficient approach uses three separate lists, a **vertex list**, **normal list**, and **face list**. The vertex list reports the locations of the distinct vertices in the mesh. The list of normals reports the directions of the distinct normal vectors that occur in the model. The face list simply indexes into these lists. As we see next, the barn is thereby captured with 10 vertices, seven normals, and a list of seven simple face descriptors.

The three lists work together: The vertex list contains locational or **geometric** information, the normal list contains **orientation** information, and the face list contains connectivity or **topological** information.

The vertex list for the barn is shown in Figure 6.5. The list of the seven distinct normals is shown in Figure 6.6. The vertices have indices 0 through 9 and the normals have indices 0 through 6. The vectors shown have already been normalized, since most shading algorithms require unit vectors. (Recall that a cosine can be found as the dot product between two unit vectors).

vertex	x	y	z
0	0	0	0
1	1	0	0
2	1	1	0
3	0.5	1.5	0
4	0	1	0
5	0	0	1
6	1	0	1
7	1	1	1
8	0.5	1.5	1
9	0	1	1

Figure 6.5. Vertex list for the basic barn.

normal	n_x	n_y	n_z
0	-1	0	0

1	-0.477	0.8944	0
2	0.447	0.8944	0
3	1	0	0
4	0	-1	0
5	0	0	1
6	0	0	-1

Figure 6.6. The list of distinct normal vectors involved.

face	vertices	associated normal
0 (left)	0,5,9,4	0,0,0,0
1 (roof left)	3,4,9,8	1,1,1,1
2 (roof right)	2,3,8,7	2,2,2,2
3 (right)	1,2,7,6	3,3,3,3
4 (bottom)	0,1,6,5	4,4,4,4
5 (front)	5,6,7,8,9	5,5,5,5,5
6 (back)	0,4,3,2,1	6,6,6,6,6

Figure 6.7. Face list for the basic barn.

Figure 6.7 shows the barn's face list: each face has a list of vertices and the normal vector associated with each vertex. To save space, just the indices of the proper vertices and normals are used. (Since each surface is flat all of the vertices in a face are associated with the same normal.) The list of vertices for each face begins with any vertex in the face, and then proceeds around the face vertex by vertex until a complete circuit has been made. There are two ways to traverse a polygon: clockwise and counterclockwise. For instance, face #5 above could be listed as (5,6,7,8,9) or (9,8,7,6,5). Either direction could be used, but we follow a convention that proves handy in practice:

- *Traverse the polygon counterclockwise as seen from outside the object.*

Using this order, if you traverse around the face by walking on the outside surface from vertex to vertex, the interior of the face is on your left. We later design algorithms that exploit this ordering. Because of it the algorithms are able to distinguish with ease the “front” from the “back” of a face.

The barn is an example of a “data intensive” model, where the position of each vertex is entered (maybe by hand) by the designer. In contrast, we see later some models that are generated algorithmically. Here it isn't too hard to come up with the vertices for the basic barn: the designer chooses a simple unit square for the floor, decides to put one corner of the barn at the origin, and chooses a roof height of 1.5 units. By suitable scaling these dimension can be altered later (although the relative height of the wall to the barn's peak, 1: 1.5, is forever fixed).

6.2.2. Finding the normal vectors.

It may be possible to set vertex positions by hand, but it is not so easy to calculate the normal vectors. In general each face will have three or more vertices, and a designer would find it challenging to jot down the normal vector. It's best to let the computer do it during the creation of the mesh model.

If the face is considered flat, as in the case of the barn, we need only find the normal vector to the face itself, and associate it with each of the face's vertices. One direct way uses the vector cross product to find the normal, as in Figure 4.16. Take any three adjacent points on the face, V_1 , V_2 , and V_3 , and compute the normal as their cross product $\mathbf{m} = (V_1 - V_2) \times (V_3 - V_2)$. This vector can now be normalized to unit length.

There are two problems with this simple approach. a). If the two vectors $V_1 - V_2$ and $V_3 - V_2$ are nearly parallel the cross product will be very small (why?), and numerical inaccuracies may result. b). As we see later, it may turn out that the polygon is not perfectly planar, i.e. that all of the vertices do not lie in the same plane. Thus

the surface represented by the vertices cannot be truly flat. We need to form some “average” value for the normal to the polygon, one that takes into consideration all of the vertices.

A robust method that solves both of these problems was devised by Martin Newell [newm79]. It computes the components m_x , m_y , m_z of the normal \mathbf{m} according to the formula:

$$\begin{aligned} m_x &= \sum_{i=0}^{N-1} (y_i - y_{\text{next}(i)})(z_i + z_{\text{next}(i)}) \\ m_y &= \sum_{i=0}^{N-1} (z_i - z_{\text{next}(i)})(x_i + x_{\text{next}(i)}) \\ m_z &= \sum_{i=0}^{N-1} (x_i - x_{\text{next}(i)})(y_i + y_{\text{next}(i)}) \end{aligned} \quad (6.1)$$

where N is the number of vertices in the face, (x_i, y_i, z_i) is the position of the i -th vertex, and where $\text{next}(j) = (j+1) \bmod N$ is the index of the “next” vertex around the face after vertex j , in order to take care of the “wrap-around” from the $(N-1)$ -st to the 0-th vertex. The computation requires only one multiplication per edge for each of the components of the normal, and no testing for collinearity is needed. This result is developed in Case Study 6.2, and C++ code is presented for it.

The vector \mathbf{m} computed by the Newell method could point toward the inside or toward the outside of the polygon. We also show in the Case study that if the vertices of the polygon are traversed (as i increases) in a CCW direction as seen from outside the polygon, then \mathbf{m} points toward the outside of the face.

Example 6.2.2: Consider the polygon with vertices $P_0 = (6, 1, 4)$, $P_1 = (7, 0, 9)$, and $P_2 = (1, 1, 2)$. Find the normal to this polygon using the Newell method. **Solution:** Direct use of the cross product gives $((7, 0, 9) - (6, 1, 4)) \times ((1, 1, 2) - (6, 1, 4)) = (2, -23, -5)$. Application of the Newell method yields the same result: $(2, -23, -5)$.

Practice Exercises.

6.2.1. Using the Newell Method.

For the three vertices $(6, 1, 4)$, $(2, 0, 5)$, and $(7, 0, 9)$, compare the normal found using the Newell method with that found using the usual cross product. Then use the Newell method to find (n_x, n_y, n_z) for the polygon having the vertices $(1, 1, 2)$, $(2, 0, 5)$, $(5, 1, 4)$, $(6, 0, 7)$. Is the polygon planar? If so, find its true normal using the cross product, and compare it with the result of the Newell method.

6.2.2. What about a non-planar polygon? Consider the quadrilateral shown in Figure 6.8 that has the vertices $(0, 0, 0)$, $(1, 0, 0)$, $(0, 0, 1)$, and $(1, a, 1)$. When a is nonzero this is a non-planar polygon. Find the “normal” to it using the Newell method, and discuss how good an estimate it is for different values of a .

Figure 6.8. A non-planar polygon.

6.2.3. Represent the “generic cube”. Make vertex, normal, and face lists for the “generic cube”, which is centered at the origin, has its edges aligned with the coordinate axes, and has edges of length two. Thus its eight vertices lie at the eight possible combinations of ‘+’ and ‘-‘ in $(\pm 1, \pm 1, \pm 1)$.

6.2.4. Faces with holes. Figure 6.9 shows how a face containing a hole can be captured in a face list. A pair of imaginary edges are added that bridge the gap between the circumference of the face and the hole, as suggested in the figure.

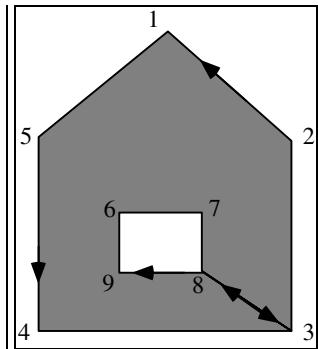


Figure 6.9. A face containing a hole.

The face is traversed so that (when walking along the outside surface) the interior of the face lies to the left. Thus a hole is traversed in the CW direction. Assuming we are looking at the face in the figure from its outside, the list of vertices would be: 5 4 3 8 9 6 7 8 3 2 1. Sketch this face with an additional hole in it, and give the proper list of vertices for the face. What normals would be associated with each vertex?

6.2.3. Properties of Meshes.

Given a mesh consisting of a vertex, normal, and face lists, we might wonder what kind of an object it represents. Some properties of interest are:

- **Solidity:** As mentioned above a mesh represents a solid object if its faces together enclose a positive and finite amount of space.
- **Connectedness:** A mesh is **connected** if every face shares at least one edge with some other face. (If a mesh is not connected it is usually considered to represent more than one object.)
- **Simplicity:** A mesh is **simple** if the object it represents is solid and has no holes through it; it can be deformed into a sphere without tearing. (Note that the term “simple” is being used here in quite a different sense from for a “simple” polygon.).
- **Planarity:** A mesh is planar if every face is a **planar** polygon: The vertices of each face then lie in a single plane. Some graphics algorithms work much more efficiently if a face is planar. Triangles are inherently planar, and some modeling software takes advantage of this by using only triangles. Quadrilaterals, on the other hand, may or may not be planar. The quadrilateral in Figure 6.8, for instance, is planar if and only if $a = 0$.
- **Convexity:** The mesh represents a **convex** object if the line connecting any two points within the object lies wholly inside the object. Convexity was first discussed in section 2.3.6 in connection with polygons. Figure 6.10 shows some convex and some non-convex objects. For each non-convex object an example line is shown whose endpoints lie in the object but which is not itself contained within the object.

Figure 6.10. Examples of convex and nonconvex 3D objects.

The basic barn happens to possess all of these properties (check this). For a given mesh some of these properties are easy to determine in a program, in the sense that a simple algorithm exists that does the job. (We discuss some below.) Other properties, such as solidity, are quite difficult to establish.

The polygonal meshes we choose to use to model objects in graphics may have some or all of these properties: the choice depends on how one is going to use the mesh. If the mesh is supposed to represent a physical object made of some material, perhaps to determine its mass or center of gravity, we may insist that it be at least

connected and solid. If we just want to draw the object, however, much greater freedom is available, since many objects can still be drawn, even if they are “non-physical”.

Figure 6.11 shows some examples of objects we might wish to represent by meshes. PYRAMID is made up of triangular faces which are necessarily planar. It is not only convex, in fact it has all of the properties above.

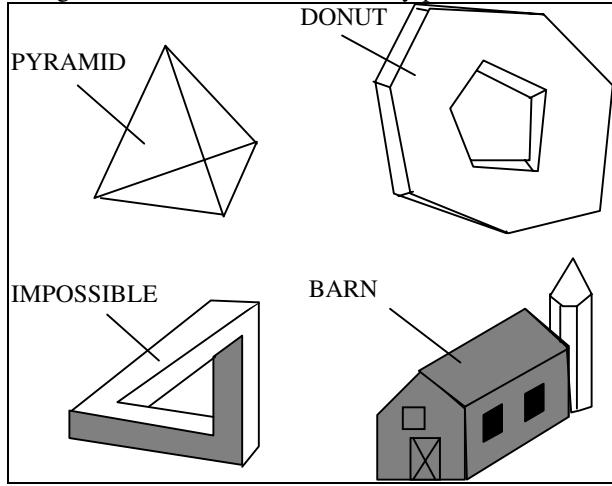


Figure 6.11. Examples of solids to be described by meshes.

DONUT is connected and solid, but it is neither simple (there is a hole through it) nor convex. Two of its faces themselves have holes. Whether or not its faces are planar polygons cannot be determined from a figure alone. Later we give an algorithm for determining planarity from the face and vertex lists.

IMPOSSIBLE cannot exist in space. Can it really be represented by a mesh?

BARN seems to be made up of two parts, but it could be contained in a single mesh. Whether it is connected would then depend on how the faces were defined at the juncture between the silo and the main building. BARN also illustrates a situation often encountered in graphics. Some faces have been added to the mesh that represent windows and doors, to provide “texture” to the object. For instance, the side of the barn is a rectangle with no holes in it, but two squares have been added to the mesh to represent windows. These squares are made to lie in the same plane as the side of the barn. This then is not a connected mesh, but it can still be displayed on a graphics device.

6.2.4. Mesh models for non-solid objects.

Figure 6.12 shows examples of other objects that can be characterized by polygonal meshes. These are surfaces, and are best thought of as infinitesimally thick “shells.”

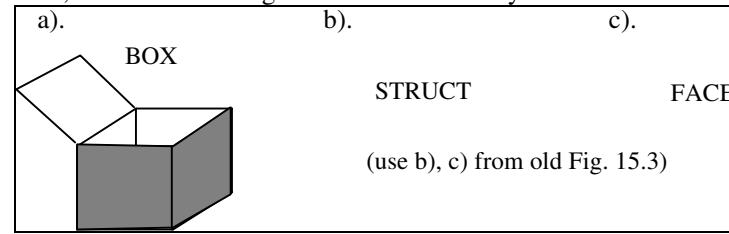


Figure 6.12. Some surfaces describable by meshes. (Part c is courtesy of the University of Utah.)

BOX is an open box whose lid has been raised. In a graphics context we might want to color the outside of BOX's six faces blue and their insides green. (What is obtained if we remove one face from PYRAMID above?)

Two complex surfaces, STRUCT and FACE, are also shown. For these examples the polygonal faces are being used to approximate a smooth underlying surface. In some situations the mesh may be all that is available for the object, perhaps from digitizing points on a person's face. If each face of the mesh is drawn as a shaded polygon, the picture will look artificial, as seen in FACE. Later we shall examine tools that attempt to draw the smooth underlying surface based only on the mesh model.

Many geometric modeling software packages construct a model from some object - a solid or a surface - that tries to capture the true shape of the object in a polygonal mesh. The problem of composing the lists can be difficult. As an example, consider creating an algorithm that generates vertex and face lists to approximate the shape of an engine block, a prosthetic limb, or a building. This area in fact is a subject of much ongoing research [Mant88], [Mort85]. By using a sufficient number of faces, a mesh can approximate the "underlying surface" to any degree of accuracy desired. This property of **completeness** makes polygon meshes a versatile tool for modeling.

6.2.5. Working with Meshes in a Program.

We want an efficient way to capture a mesh in a program that makes it easy to create and draw the object. Since mesh data is frequently stored in a file, we also need simple ways to read and write "mesh files".

It is natural to define a class *Mesh*, and to imbue it with the desired functionality. Figure 6.13 shows the declaration of the class *Mesh*, along with those of two simple helper classes, *VertexID* and *Face*¹. A *Mesh* object has a vertex list, a normal list, and a face list, represented simply by arrays *pt*, *norm*, and *face*, respectively. These arrays are allocated dynamically at runtime, when it is known how large they must be. Their lengths are stored in *numVerts*, *numNormals*, and *numFaces*, respectively. Additional data fields can be added later that describe various physical properties of the object such as weight and type of material.

```
//////////////////////////////////////////////////////////////////////// VertexID #####
class VertexID{
public:
    int vertIndex; // index of this vert in the vertex list
    int normIndex; // index of this vertex's normal
};

//////////////////////////////////////////////////////////////////////// Face #####
class Face{
public:
    int nverts; // number of vertices in this face
    VertexID *vert; // the list of vertex and normal indices
    Face(){nverts = 0; vert = NULL;} // constructor
    ~Face(){delete[] vert; nverts = 0;} // destructor
};

//////////////////////////////////////////////////////////////////////// Mesh #####
class Mesh{
private:
    int numverts; // number of vertices in the mesh
    Point3* pt; // array of 3D vertices
    int numNormals; // number of normal vectors for the mesh
    Vector3 *norm; // array of normals
    int numFaces; // number of faces in the mesh
    Face* face; // array of face data
    // ... others to be added later
public:
    Mesh(); // constructor
    ~Mesh(); // destructor
    int readFile(char * fileName); // to read in a filed mesh
    .. others ..
};
```

Figure 6.13. Proposed data type for a mesh.

¹ Definitions of the basic classes *Point3* and *Vertex3* have been given previously, and also appear in Appendix 3.

The Face data type is basically a list of vertices and the normal vector associated with each vertex in the face. It is organized here as an array of index pairs: the v-th vertex in the f-th face has position `pt[face[f].vert[v].vertIndex]` and normal vector `norm[face[f].vert[v].normIndex]`. This appears cumbersome at first exposure, but the indexing scheme is quite orderly and easy to manage, and it has the advantage of efficiency, allowing rapid “random access” indexing into the `pt[]` array.

Example 6.2.3. Data for the tetrahedron. Figure 6.14 shows the specific data structure for the tetrahedron shown, which has vertices at (0,0,0), (1,0,0), (0,1,0), and (0,0,1). Check the values reported in each field. (We discuss how to find the normal vectors later.)

Figure 6.14. Data for the tetrahedron.

The first task is to develop a method for drawing such a mesh object. It’s a matter of drawing each of its faces, of course. An OpenGL-based implementation of the `Mesh::draw()` method must traverse the array of faces in the mesh object, and for each face send the list of vertices and their normals down the graphics pipeline. In OpenGL you specify that subsequent vertices are associated with normal vector `m` by executing `glNormal3f(m.x, m.y, m.z)`². So the basic flow of `Mesh::draw()` is:

```
for (each face, f, in the mesh)
{
    glBegin(GL_POLYGON);
    for (each vertex, v, in face, f)
    {
        glNormal3f(normal at vertex v);
        glVertex3f(position of vertex v);
    }
    glEnd();
}
```

The implementation is shown in Figure 6.15.

```
void Mesh:: draw() // use OpenGL to draw this mesh
{
    for(int f = 0; f < numFaces; f++) // draw each face
    {
        glBegin(GL_POLYGON);
        for(int v = 0; v < face[f].nVerts; v++) // for each one..
        {
            int in = face[f].vert[v].normIndex ; // index of this normal
            int iv = face[f].vert[v].vertIndex ; // index of this vertex
            glNormal3f(norm[in].x, norm[in].y, norm[in].z);
            glVertex3f(pt[iv].x, pt[iv].y, pt[iv].z);
        }
        glEnd();
    }
}
```

Figure 6.15. Method to draw a mesh using OpenGL.

We also need methods to create a particular mesh, and to read a predefined mesh into memory from a file. We consider reading from and writing to files in Case Study 6.1. We next examine a number of interesting families of shapes that can be stored in a mesh, and see how to create them.

² For proper shading these vectors must be normalized. Otherwise place `glEnable(GL_NORMALIZE)` in the `init()` function. This requests that OpenGL automatically normalize all normal vectors.

Creating and drawing a Mesh object using SDL.

It is convenient in a graphics application to read in mesh descriptions using the SDL language introduced in Chapter 5. To make this possible simply derive the `Mesh` class from `Shape`, and add the method `drawOpenGL()`. Thus Figure 6.13 becomes:

```
class Mesh : public Shape {  
    //same as in Figure 6.13  
    virtual void drawOpenGL()  
{  
    tellMaterialsGL(); glPushMatrix(); // load properties into the object  
    glMultMatrixf(transf.m); draw(); // draw the mesh  
    glPopMatrix();  
}  
}; //end of Mesh class
```

The `Scene` class that reads SDL files is already set to accept the keyword `mesh`, followed by the name of the file that contains the mesh description. Thus to create and draw a pawn with a certain translation and scaling, use:

```
push translate 3 5 4 scale 3 3 3 mesh pawn.3vn pop
```

6.3. Polyhedra.

It is frequently convenient to restrict the data in a mesh so that it represents a **Polyhedron**. A very large number of solid objects of interest are indeed polyhedra, and algorithms for processing a mesh can be greatly simplified if they need only process meshes that represent a polyhedron.

Slightly different definitions for a polyhedron are used in different contexts, [Coxeter69, Courant & Robbins??, F&VD90,] but we use the following:

Definition

A **Polyhedron** is a connected mesh of simple planar polygons that encloses a finite amount of space.

So by definition a polyhedron represents a single solid object. This requires that:

- every edge is shared by exactly two faces;
- at least three edges meet at each vertex;
- faces do not interpenetrate: Two faces either do not touch at all, or they touch only along their common edge.

In Figure 6.11 PYRAMID is clearly a polyhedron. DONUT evidently encloses space, so it is a polyhedron if its faces are in fact planar. It is not a simple polyhedron, since there is a hole through it. In addition, two of its faces themselves have holes. Is IMPOSSIBLE a polyhedron? Why? If the texture faces are omitted BARN might be modeled as two polyhedra, one for the main part and one for the silo.

Euler's formula:

Euler's formula (which is very easy to prove; see for instance [courant and robbins, 61, p. 236]) provides a fundamental relationship between the number of faces, edges, and vertices (F , E , and V respectively) of a simple polyhedron:

$$V + F - E = 2 \quad (6.1)$$

For example, a cube has $V = 8$, $F = 6$, and $E = 12$.

A generalization of this formula is available if the polyhedron is not simple [F&VD p. 544]:

$$V + F - E = 2 + H - 2G \quad (6.2)$$

where H is the total number of holes occurring in faces, and G is the number of holes through the polyhedron. Figure 6.16a shows a “donut” constructed with a hole in the shape of another parallelepiped. The two ends are beveled off as shown. For this object $V = 16$, $F = 16$, $E = 32$, $H = 0$, and $G = 1$. Figure 6.16b shows a polyhedron having a hole A penetrating part way into it, and hole B passing through it.

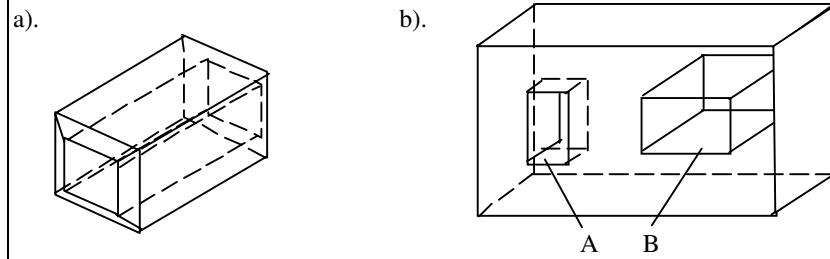


Figure 6.16. Polyhedron with holes.

Here $V = 24$, $F = 15$, $E = 36$, $H = 3$, and $G = 1$. These values satisfy the formula above in both cases.

The “structure” of a polyhedron is often nicely revealed by a **Schlegel diagram**. This is based on a view of the polyhedron from a point just outside the center of one of its faces, as suggested in Figure 6.17a. Viewing a cube in this way produces the Schlegel diagram shown in Figure 6.17b. The front face appears as a large polygon surrounding the rest of the faces.

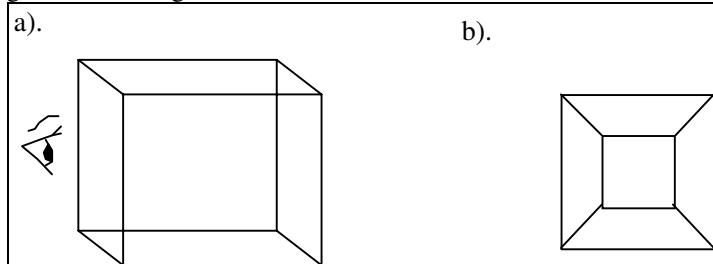


Figure 6.17. The Schlegel diagrams for a cube.

Figure 6.18 shows further examples. Part a) shows the Schlegel diagram of PYRAMID shown in Figure 6.11, and parts b) and c) show two quite different Schlegel diagrams for the basic barn. (Which faces are closest to the eye?).

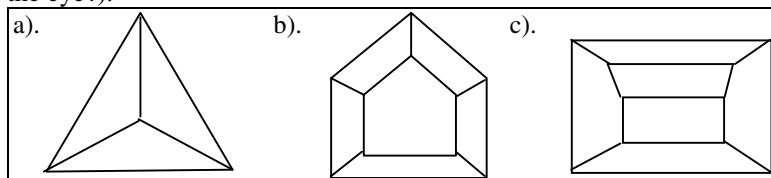


Figure 6.18. Schlegel diagrams for PYRAMID and the basic barn.

6.3.1. Prisms and Antiprisms.

A prism is a particular type of polyhedron that embodies certain symmetries, and therefore is quite simple to describe. As shown in Figure 6.19 a prism is defined by **sweeping** (or **extruding**) a polygon along a straight line, turning a 2D polygon into a 3D polyhedron. In Figure 6.19a polygon P is swept along vector \mathbf{d} to form the polyhedron shown in part b. When \mathbf{d} is perpendicular to the plane of P the prism is a **right prism**. Figure 6.19c shows some block letters of the alphabet swept to form prisms.

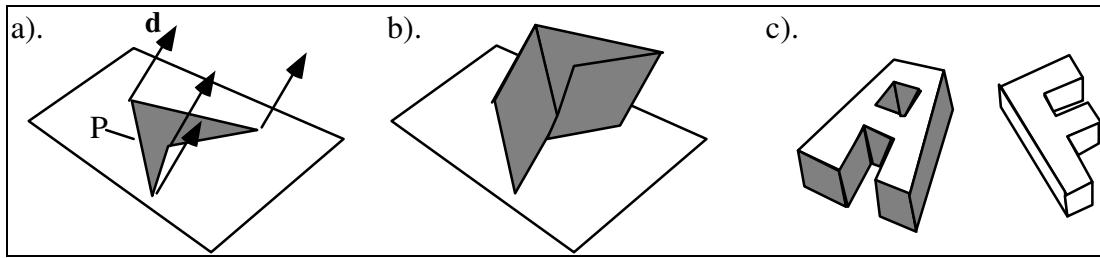


Figure 6.19. Forming a prism.

A **regular prism** has a regular polygon for its base, and squares for its side faces. A hexagonal version is shown in Figure 6.20a. A variation is the **antiprism**, shown in Figure 6.20b. This is not an extruded object. Instead the top n -gon is rotated through $180 / n$ degrees and connected to the bottom n -gon to form faces that are equilateral triangles. (How many triangular faces are there?) The regular prism and antiprism are examples of “semi-regular” polyhedra, which we examine further later.

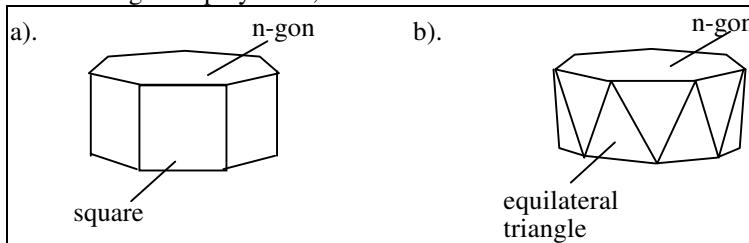


Figure 6.20. A regular prism and antiprism.

Practice Exercises

6.3.1. Build the lists for a prism.

Give the vertex, normal, and face lists for the prism shown in Figure 6.21. Assume the base of the prism (face #4) lies in the xy -plane, and vertex 2 lies on the z -axis at $z = 4$. Further assume vertex 5 lies 3 units along the x -axis, and that the base is an equilateral triangle.

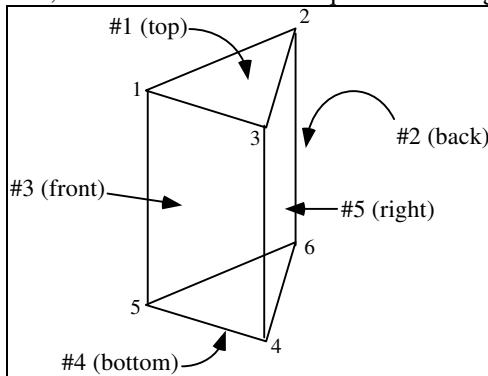


Figure 6.21. An example polyhedral object.

6.3.2. The unit cube. Build vertex, normal, and face lists for the **unit cube**: the cube centered at the origin with its edges aligned with the coordinate axes. Each edge has length 2.

6.3.3. An Antiprism. Build vertex, normal, and face lists for an anti-prism having a square as its top polygon.

6.3.4. Is This Mesh Connected? Is the mesh defined by the list of faces: (4, 1, 3), (4, 7, 2, 1), (2, 7, 5), (3, 4, 8, 7, 9) connected? Try to sketch the object, choosing arbitrary positions for the nine vertices. What algorithm might be used to test a face list for connectedness?

6.3.5. Build Meshes. For the DONUT, IMPOSSIBLE, and BARN objects in Figure 6.11, assign numbers to each vertex and then write a face list.

6.3.6. Schlegel diagrams. Draw Schlegel diagrams for the prisms of Figure 6.20 and Figure 6.21.

6.3.2. The Platonic Solids.

If all of the faces of a polyhedron are identical and each is a regular polygon, the object is a **regular polyhedron**. These symmetry constraints are so severe that only five such objects can exist, the **Platonic solids**³ shown in Figure 6.22 [Coxe61]. The Platonic solids exhibit a sublime symmetry and a fascinating array of properties. They make interesting objects of study in computer graphics, and often appear in solid modeling CAD applications.

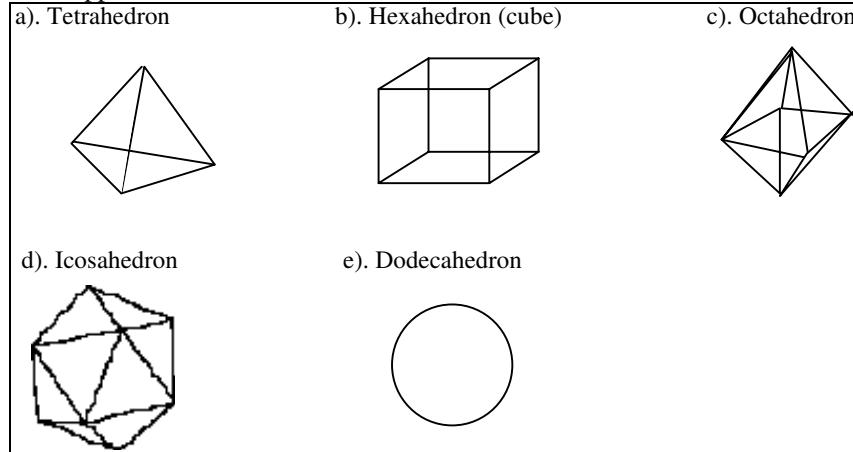


Figure 6.22. The five Platonic solids.

Three of the Platonic solids have equilateral triangles as faces, one has squares, and the dodecahedron has pentagons. The cube is a regular prism, and the octahedron is an antiprism (why?). The values of V , F , and E for each solid are shown in Figure 6.23. Also shown is the Schläfli⁴ symbol (p,q) for each solid. It states that each face is a p -gon and that q of them meet at each vertex.

<u>solid</u>	<u>V</u>	<u>F</u>	<u>E</u>	<u>Schlafli</u>
Tetrahedron	4	4	6	$(3, 3)$
Hexahedron	8	6	12	$(4, 3)$
Octahedron	6	8	12	$(3, 4)$
Icosahedron	12	20	30	$(3, 5)$
Dodecahedron	20	12	30	$(5, 3)$

Figure 6.23. Descriptors for the Platonic solids.

It is straightforward to build mesh lists for the cube and octahedron (see the exercises). We give example vertex and face lists for the tetrahedron and icosahedron below, and we discuss how to compute normal lists. We also show how to derive the lists for the dodecahedron from those of the icosahedron, making use of their duality which we define next.

• Dual polyhedra.

Each of the Platonic solids P has a **dual** polyhedron D . The vertices of D are the *centers* of the faces of P , so edges of D connect the midpoints of adjacent faces of P . Figure 6.24 shows the dual of each Platonic solid inscribed within it.

old A8.2 showing duals.

Figure 6.24. Dual Platonic solids.

- The dual of a tetrahedron is also a tetrahedron.
- The cube and octahedron are duals.

³Named in honor of Plato (427-347 bc) who commemorated them in his *Timaeus*. But they were known before this: a toy dodecahedron was found near Padua in Etruscan ruins dating from 500 BC.

⁴ Named for L. Schläfli, (1814-1895), a Swiss Mathematician.

- The icosahedron and dodecahedron are duals.

Duals have the same number, E , of edges, and V for one is F for the other. In addition, if (p, q) is the Schläfli symbol for one, then it is (q, p) for the other.

If we know the vertex list of one Platonic solid P , we can immediately build the vertex list of its dual D , since vertex k of D lies at the center of face k of P . Building D in this way actually builds a version that inscribes P .

To keep track of vertex and face numbering we use a **model**, which is fashioned by slicing along certain edges of each solid and “unfolding” it to lie flat, so that all the faces are seen from the outside. Models for three of the Platonic solids are shown in Figure 6.25.

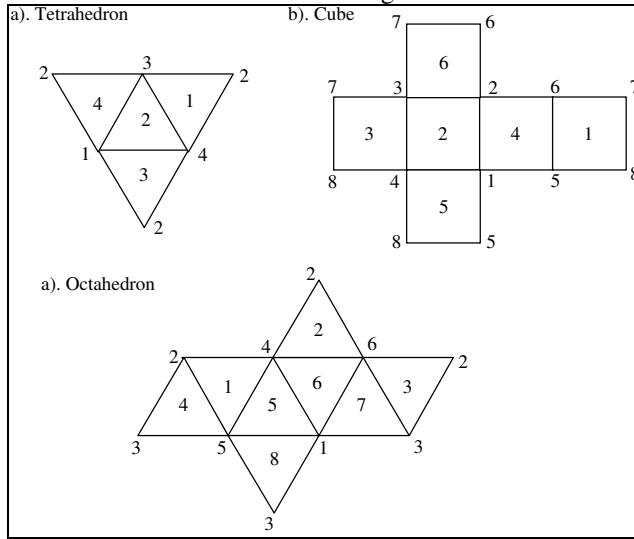


Figure 6.25. Models for the tetrahedron, cube, and octahedron.

Consider the dual pair of the cube and octahedron. Face 4 of the cube is seen to be surrounded by vertices 1, 5, 6, and 2. By duality vertex 4 of the octahedron is surrounded by faces 1, 5, 6, and 2. Note for the tetrahedron that since it is self-dual, the list of vertices surrounding the k -th face is identical to the list of faces surrounding the k -th vertex.

The position of vertex 4 of the octahedron is the center of face 4 of the cube. Recall from Chapter 5 that the center of a face is just the *average* of the vertices belonging to that face. So if we know the vertices V_1, V_5, V_6 , and V_2 for face 4 of the cube we have immediately that:

$$V_4 = \frac{1}{4}(V_1 + V_5 + V_6 + V_2) \quad (6.3)$$

Practice Exercises

6.3.7. The octahedron. Consider the octahedron that is the dual of the cube described in Exercise 9.4.3. Build vertex and face lists for this octahedron.

6.3.8. Check duality. Beginning with the face and vertex lists of the octahedron in the previous exercise, find its dual and check that this dual is (a scaled version of) the cube.

Normal Vectors for the Platonic Solids.

If we wish to build meshes for the Platonic solids we must compute the normal vector to each face. This can be done in the usual way using Newell’s method, but the high degree of symmetry of a Platonic solid offers a much simpler approach. Assuming the solid is centered at the origin, the normal vector to each face is the vector from the origin to the *center* of the face, which is formed as the average of the vertices. Figure 6.26 shows this for the octahedron. The normal to the face shown is simply:

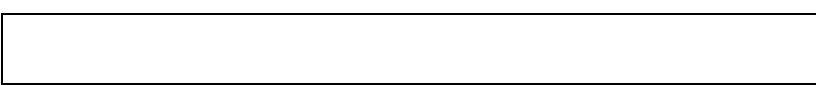


Figure 6.26. Using symmetry to find the normal to a face.

$$\mathbf{m} = (V_1 + V_2 + V_3) / 3 \quad (6.4)$$

(Note also: This vector is also the same as that from the origin to the appropriate vertex on the dual Platonic solid.)

• The Tetrahedron

The vertex list of a tetrahedron depends of course on how the tetrahedron is positioned, oriented, and sized. It is interesting that a tetrahedron can be inscribed in a cube (such that its four vertices lie in corners of the cube, and its four edges lie in faces of the cube). Consider the unit cube having vertices $(\pm 1, \pm 1, \pm 1)$, and choose the tetrahedron that has one vertex at $(1, 1, 1)$. Then it has vertex and face lists given in Figure 6.27 [blinn87].

vertex list				face list	
vertex	x	y	z	face number	vertices
0	1	1	1	0	1,2,3
1	1	-1	-1	1	0,3,2
2	-1	-1	1	2	0,1,3
3	-1	1	-1	3	0,2,1

Figure 6.27. Vertex List and Face List for a Tetrahedron.

• The Icosahedron

The vertex list for the icosahedron presents more of a challenge, but we can exploit a remarkable fact to make it simple. Figure 6.28 shows that three mutually perpendicular **golden rectangles** inscribe the icosahedron, and so a vertex list may be read directly from this picture. We choose to align each golden rectangle with a coordinate axis. For convenience, we size the rectangles so their longer edge extends from -1 to 1 along its axis. The shorter edge then extends from $-\tau$ to τ , where $\tau = (\sqrt{5} - 1) / 2 = 0.618\dots$ is the reciprocal of the golden ratio ϕ .

old Fig A8.4 golden rectangles in icosahedron.

Figure 6.28. Golden rectangles defining the icosahedron.

From this it is just a matter of listing vertex positions, as shown in Figure 6.29.

vertex	x	y	z
0	0	1	τ
1	0	1	$-\tau$
2	1	τ	0
3	1	$-\tau$	0
4	0	-1	$-\tau$
5	0	-1	τ
6	τ	0	1
7	$-\tau$	0	1
8	τ	0	-1
9	$-\tau$	0	-1
10	-1	τ	0
11	-1	$-\tau$	0

Figure 6.29. Vertex List for the Icosahedron.

A model for the icosahedron is shown in Figure 6.30. The face list for the icosahedron can be read directly off of it. (Question: what is the normal vector to face #8?)

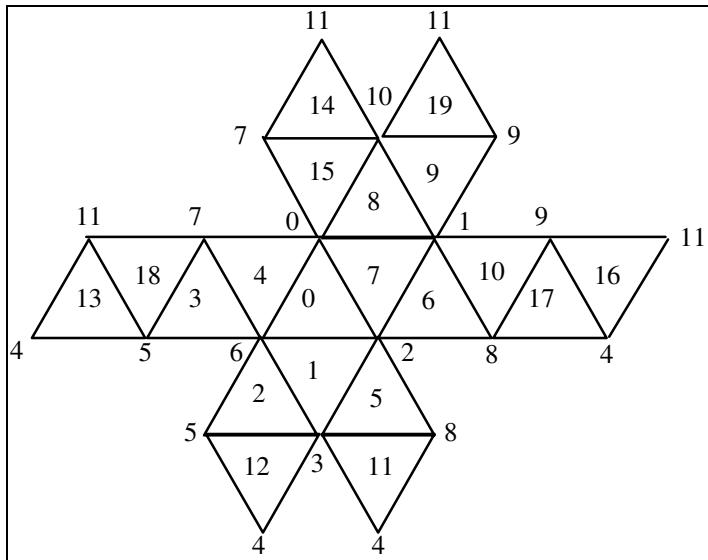


Figure 6.30. Model for the icosahedron.

Some people prefer to adjust the model for the icosahedron slightly into the form shown in Figure 6.31. This makes it clearer that an icosahedron is made up of an anti-prism (shown shaded) and two pentagonal pyramids on its top and bottom.

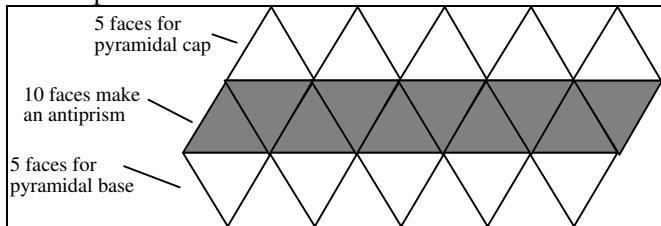


Figure 6.31. An icosahedron is an anti-prism with a cap and base.

• The Dodecahedron

The dodecahedron is dual to the icosahedron, so all the information needed to build lists for the dodecahedron is buried in the lists for the icosahedron. But it is convenient to see the model of the dodecahedron laid out, as in Figure 6.32.

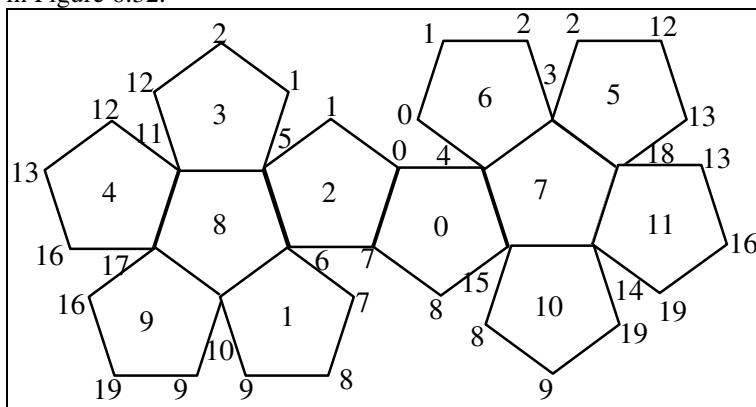


Figure 6.32. Model for the dodecahedron.

Using duality again, we know vertex k of the dodecahedron lies at the center of face k of the icosahedron: just average the three vertices of face k . All of the vertices of the dodecahedron are easily calculated in this fashion.

Practice Exercises

6.3.9. Icosahedral Distances. What is the radial distance of each vertex of the icosahedron above from the origin?

6.3.10. Vertex list for the dodecahedron. Build the vertex list and normal list for the dodecahedron.

6.3.3. Other interesting Polyhedra.

There are endless varieties of polyhedra (see for instance [Wenn71] and [Coxe63 polytopes]), but one class is particularly interesting. Whereas each Platonic solid has the same type of n -gon for all of its faces, the **Archimedean** (also **semi-regular**) solids have more than one kind of face, although they are still regular polygons. In addition it is required that every vertex is surrounded by the same collection of polygons in the same order.

For instance, the “truncated cube”, shown in Figure 6.33a, has 8-gons and 3-gons for faces, and around each vertex one finds one triangle and two 8-gons. This is summarized by associating the symbol 3·8·8 with this solid.

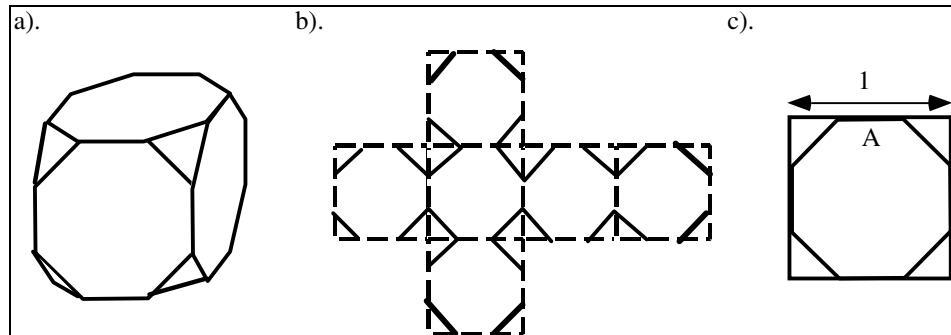


Figure 6.33. The Truncated Cube.

The truncated cube is formed by “slicing” off each corner of a cube in just the right fashion. The model for the truncated cube, shown in Figure 6.33b, is based on that of the cube. Each edge of the cube is divided into three parts, the middle part of length $A = 1/(1 + \sqrt{2})$ (Figure 6.33c), and the middle portion of each edge is joined to its neighbors. Thus if an edge of the cube has endpoints C and D , two new vertices V , and W are formed as the affine combinations

$$\begin{aligned} V &= \frac{1+A}{2}C + \frac{1-A}{2}D \\ W &= \frac{1-A}{2}C + \frac{1+A}{2}D \end{aligned} \tag{6.5}$$

Based on this it is straightforward to build vertex and face lists for the truncated cube (see the exercises and Case Study 6.3).

Given the constraints that faces must be regular polygons, and that they must occur in the same arrangement about each vertex, there are only 13 possible Archimedean solids discussed further in Case Study 6.10. Archimedean solids still enjoy enough symmetry that the normal vector to each face is found using the center of the face.

One Archimedean solid of particular interest is the truncated icosahedron $5\cdot6^2$ shown in Figure 6.34, which consists of regular hexagons and pentagons. The pattern is familiar from soccer balls used around the world. More recently this shape has been named the **Buckyball** after Buckminster Fuller because of his interest in geodesic structures similar to this. Crystallographers have recently discovered that 60 atoms of carbon can be

arranged at the vertices of the truncated icosahedron, producing a new kind of carbon molecule that is neither graphite nor diamond. The material has many remarkable properties, such as high-temperature stability and superconductivity [Brow90 Sci. Amer., also see Arthur hebard, Science Watch 1991], and has acquired the name **Fullerene**.

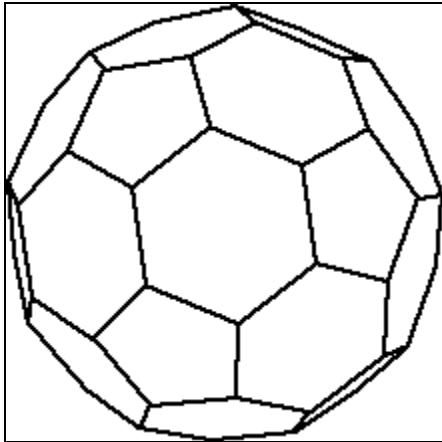


Figure 6.34. The Bucky Ball.

The Bucky ball is interesting to model and view on a graphics display. To build its vertex and face lists, draw the model of the icosahedron given in Figure 6.32 and divide each edge into 3 equal parts. This produces two new vertices along each edge, whose positions are easily calculated. Number the 60 new vertices according to taste to build the vertex list of a Bucky ball. Figure 6.35 shows a partial model of the icosahedron with the new vertices connected by edges. Note that each old face of the icosahedron becomes a hexagon, and that each old vertex of the icosahedron has been “snipped off” to form a pentagonal face. Building the face list is just a matter of listing what is seen in the model. Case Study 6.10 explores the Archimedean solids further.

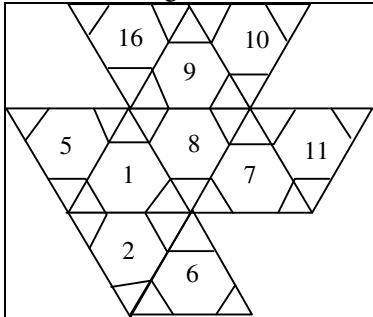


Figure 6.35. Building a Bucky Ball.

• Geodesic Domes.

Few things are harder to put up with than a good example.
Mark Twain

Although Buckminster Fuller was a pioneer along many lines, he is best known for introducing geodesic domes [fuller73]. They form an interesting class of polyhedra having many useful properties. In particular, a geodesic dome constructed out of actual materials exhibits extraordinary strength for its weight.

There are many forms a geodesic dome can take, and they all approximate a sphere by an arrangement of faces, usually triangular in shape. Once the sphere has been approximated by such faces, the bottom half is removed, leaving the familiar dome shape. An example is shown in Figure 6.36 based on the icosahedron.

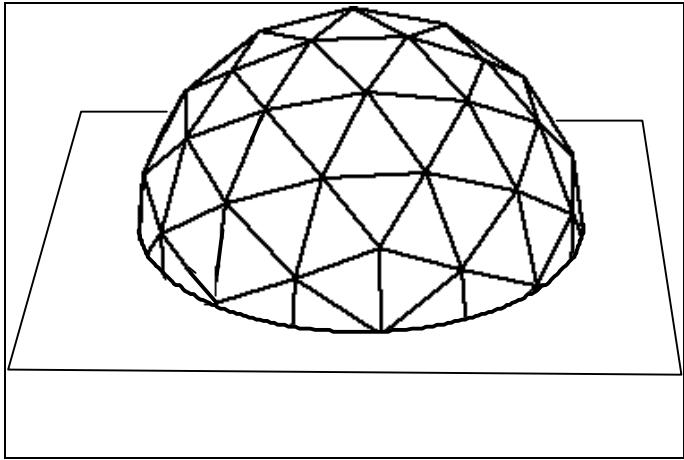


Figure 6.36. A geodesic dome.

To determine the faces, each edge of the icosahedron is subdivided into $3F$ equal parts, where Fuller called F the **frequency** of the dome. In the example $F = 3$, so each icosahedral edge is trisected to form 9 smaller triangular faces (see Figure 6.37a).

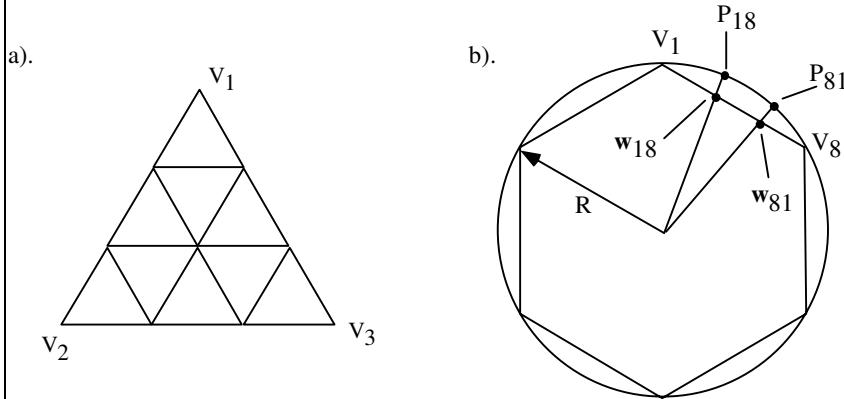


Figure 6.37. Building new vertices for the dome.

These faces do not lie in the plane of the original faces, however; first the new vertices are “projected outward” onto the surrounding sphere. Figure 6.37b presents a cross sectional view of the projection process. For example, the edge from V_1 to V_8 is subdivided to produce the two points, W_{18} and W_{81} . Vertex W_{18} is given by:

$$W_{18} = \frac{2}{3} V_1 + \frac{1}{3} V_8 . \quad (6.6)$$

(What is W_{81} ?). Projecting W_{18} onto the enclosing sphere of radius R is simply a scaling:

$$P_{18} = R \frac{\mathbf{w}_{18}}{|\mathbf{w}_{18}|} \quad (6.7)$$

where we write \mathbf{w}_{18} for the position vector associated with point W_{18} . The old and new vertices are connected by straight lines to produce the nine triangular faces for each face of the icosahedron. (Why isn’t this a “new” Platonic solid?) What are the values for E , F , and V for this polyhedron? Much more can be found on geodesic domes in the references, particularly [full75] and [kapp91].

| Practice Exercises.

6.3.11. Lists for the truncated icosahedron. Write the vertex, normal, and face lists for the truncated icosahedron described above.

6.3.12. Lists for a Bucky Ball. Create the vertex, normal, and face lists for a Bucky ball. As computing 60 vertices is tedious, it is perhaps easiest to write a small routine to form each new vertex using the vertex list of the icosahedron of Figure 6.29.

6.3.13. Build lists for the geodesic dome. Construct vertex, normal, and face lists for a frequency 3 geodesic dome as described above.

6.4. Extruded Shapes

A large class of shapes can be generated by **extruding** or **sweeping** a 2D shape through space. The prism shown in Figure 6.19 is an example of sweeping “linearly”, that is in a straight line. As we shall see, the tetrahedron and octahedron of Figure 6.22 are also examples of extruding a shape through space in a certain way. And surfaces of revolution can also be approximated by “extrusion” of a polygon, once we slightly broaden the definition of extrusion.

In this section we examine some ways to generate meshes by sweeping polygons in discrete steps. In Section 6.5 we develop similar tools for building meshes that attempt to approximate smoothly swept shapes.

6.4.1. Creating Prisms.

We begin with the prism, which is formed by sweeping a polygon in a straight line. Figure 6.38 shows a prism based on a polygon P lying in the xy -plane. P is swept through a distance H along the z -axis, forming the ARROW prism shown in Figure 6.38b. (More generally, the sweep could be along a vector \mathbf{d} , as in Figure 6.19.) As P is swept along, an edge in the z -direction is created for each vertex of P . This creates another 7 vertices, so the prism has 14 vertices in all. They appear in pairs: if $(x_i, y_i, 0)$ is one of the vertices of the base P , then the prism also contains the vertex (x_i, y_i, H) .

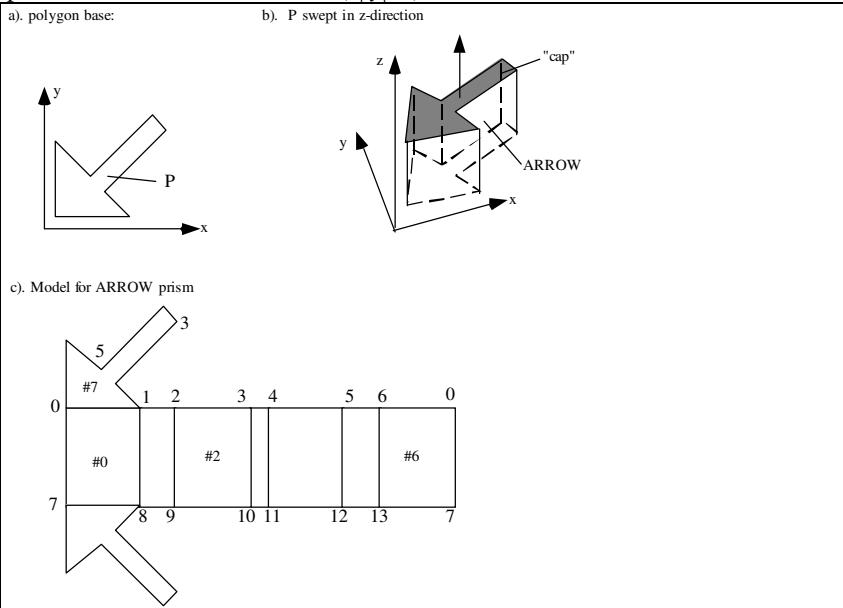


Figure 6.38. An example prism.

What face list describes ARROW? The prism is “unfolded” into the model shown in Figure 6.38c to expose its nine faces as seen from the outside. There are seven rectangular sides plus the bottom **base** P and the top **cap**. Face 2, for instance, is defined by vertices 2, 9, 10, and 3.

Because the prism has flat faces we associate the same normal vector with every vertex of a face: the normal vector to the face itself.

Building a mesh for the prism.

We want a tool to make a mesh for the prism based on an arbitrary polygon. Suppose the prism's base is a polygon with N vertices (x_i, y_i) . We number the vertices of the base $0, \dots, N-1$ and those of the cap $N, \dots, 2N-1$, so that an edge joins vertices i and $i + N$, as in the example. The vertex list is then easily constructed to contain the points $(x_i, y_i, 0)$ and (x_i, y_i, H) , for $i = 0, 1, \dots, N-1$.

The face list is also straightforward to construct. We first make the “side” faces or “walls”, and then add the cap and base. For the j -th wall ($j = 0, \dots, N-1$) we create a face with the four vertices having indices $j, j + N, next(j) + N$, and $next(j)$ where $next(j)$ is $j+1$ except if j equals $N-1$, whereupon it is 0. This takes care of the “wrap-around” from the $(N-1)$ st to the 0-th vertex. $next(j)$ is given by:

$$next(j) = (j+1) \text{ modulo } N \quad (6.8)$$

or in terms of program code: `next = (j < (N-1)) ? (j + 1) : 0.` Each face is inserted in the face list as it is created. The normal vector to each face is easily found using the Newell method described earlier. We then create the base and cap faces and insert them in the face list. Case Study 6.3 provides more details for building mesh models of prisms.

6.4.2. Arrays of Extruded Prisms - “brick laying”.

Some rendering tools, like OpenGL, can reliably draw only convex polygons. They might fail, for instance, to draw the arrow of Figure 6.38 correctly. If this is so the polygon can be decomposed (tesselated) into a set of convex polygons, and each one can be extruded. Figure 6.39 shows some examples, including a few extruded letters of the alphabet.

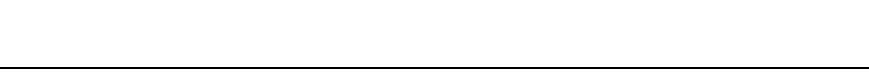


Figure 6.39. Extruded objects based on a collection of convex prisms.

Prisms like this are composed of an array of convex prisms. Some of the component prisms abut one another and therefore share all or parts of some walls. Because vertex positions are being computed at a high precision, the crease where two walls adjoin will usually be invisible.

For this family of shapes we need a method that builds a mesh out of an array of prisms, say

```
void Mesh:: makePrismArray(...)
```

which would take as its arguments a suitable list of (convex) base polygons (assumed to lie in the xy -plane), and perhaps a vector \mathbf{d} that describes the direction and amount of extrusion. The vertex list would contain the vertices of the cap and base polygons for each prism, and the individual walls, base, and cap of each prism would be stored in the face list. Drawing such a mesh would involve some wasted effort, since walls that abut would be drawn (twice), even though they are ultimately invisible.

Special case: Extruded Quad-Strips.

A simpler but very interesting family of such prisms can be built and manipulated more efficiently. These are prisms for which the base polygon can be represented by a “quad-strip”. The quad-strip is an array of quadrilaterals connected in a chain, like “laying bricks” in a row, such that neighboring faces coincide completely, as shown in Figure 6.40a. Recall from Figure 2.37 that the quad-strip is an OpenGL geometric primitive. A quad-strip is described by a sequence of vertices



Figure 6.40. Quad-strips and prisms built upon quad-strips.

$$\text{quad-strip} = \{p_0, p_1, p_2, \dots, p_{M-1}\} \quad (6.9)$$

The vertices are understood to be taken in pairs, with the odd ones forming one “edge” of the quad-strip, and the even ones forming the other edge. Not every polygon can be represented as a quad-strip. (Which of the polygons in Figure 6.39 are not quad-strips? What block letters of the alphabet can be drawn as quad-strips?)

When a mesh is formed as an extruded quad-strip only $2M$ vertices are placed in the vertex list, and only the “outside walls” are included in the face list. There are $2M - 2$ faces in all. (Why?) Thus no redundant walls are drawn when the mesh is rendered. A method for creating a mesh for an extruded quad-strip would take an array of 2D points and an extrusion vector as its parameters:

```
void Mesh:: makeExtrudedQuadStrip(Point2 p[], int numPts, Vector3 d);
```

Figure 6.41 shows some examples of interesting extruded quad-strips. Case Study 6.4 considers how to make such meshes in more detail.

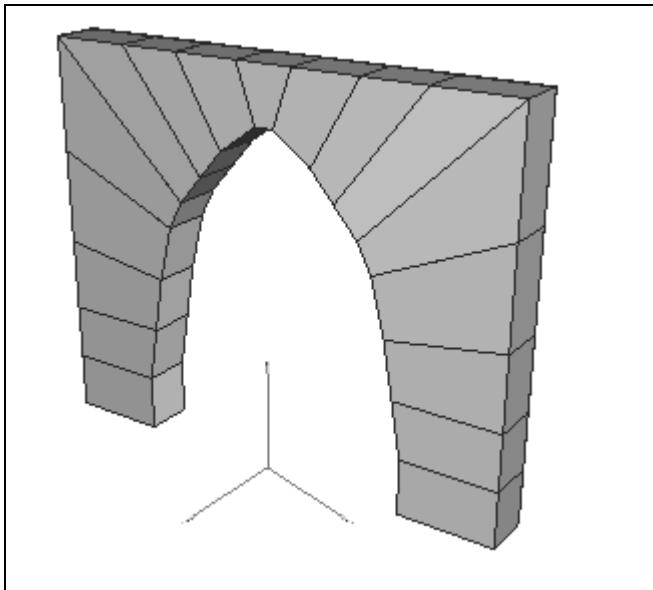


Figure 6.41. Extruded quad-strips - arches.

6.4.3. Extrusions with a “twist”.

So far an extrusion just shifts the base polygon to a new position to define the cap polygon. It is easy to generalize on this in a way that produces a much broader family of shapes; create the cap polygon as an enlarged or shrunk, and possibly rotated, version of the base polygon. Specifically, if the base polygon is P , with vertices $\{p_0, p_1, \dots, p_{N-1}\}$, the cap polygon has vertices

$$P' = \{ Mp_0, Mp_1, \dots, Mp_{N-1} \} \quad (6.10)$$

where M is some 4 by 4 matrix representing an affine transformation. Figure 6.42 shows some examples. Parts a and b show “pyramids”, or tapered cylinders” (also “truncated cones”), where the cap is a smaller version of the base. The transformation matrix for this is:

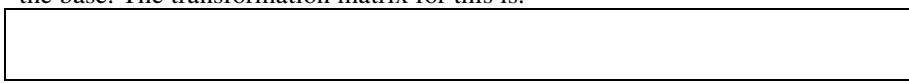


Figure 6.42. Pyramids and twisted prisms.

$$M = \begin{pmatrix} 0.7 & 0 & 0 & 0 \\ 0 & 0.7 & 0 & 0 \\ 0 & 0 & 1 & H \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

based simply on a scaling factor of 0.7 and a translation by H along z . Part c shows a cylinder where the cap has been rotated through an angle θ about the z -axis before translation, using the matrix:

$$M = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & H \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

And part d shows in cross section how cap P' can be rotated arbitrarily before it is translated to the desired position.

Prisms such as these are just as easy to create as those that use a simple translation for M : the face list is identical to the original; only the vertex positions (and the values for the normal vectors) are altered.

Practice exercises.

6.4.1. The tapered cylinder. Describe in detail how to make vertex, normal, and face lists for a tapered cylinder having regular pentagons for its base and cap, where the cap is one-half as large as the base.

6.4.2. The tetrahedron as a “tapered” cylinder. Describe how to model a tetrahedron as a tapered cylinder with a triangular base. Is this an efficient way to obtain a mesh for a tetrahedron?

6.4.3. An anti-prism. Discuss how to model the anti-prism shown in Figure 6.20b. Can it be modeled as a certain kind of extrusion?

6.4.4. Building Segmented Extrusions - Tubes and Snakes.

Another rich set of objects can be modeled by employing a sequence of extrusions, each with its own transformation, and laying them end-to-end to form a “tube”. Figure 6.43a shows a tube made by extruding a square P three times, in different directions with different tapers and twists. The first segment has end polygons M_0P and M_1P , where the initial matrix M_0 positions and orients the starting end of the tube. The second segment has end polygons M_1P and M_2P , etc. We shall call the various transformed squares the “waists” of the tube. In this example the vertex list of the mesh contains the 16 vertices $M_0p_0, M_0p_1, M_0p_2, M_0p_3, M_1p_0, M_1p_1, M_1p_2, M_1p_3, \dots, M_3p_0, M_3p_1, M_3p_2, M_3p_3$. Figure 6.43b shows a “snake”, so called because the matrices M_i cause the tube to grow and shrink to represent the body and head of a snake.



Figure 6.43*. A tube made from successive extrusions of a polygon.

Designing tubes based on 3D curves.

How do we design interesting and useful tubes and snakes? We could choose the individual matrices M_i by hand, but this is awkward at best. It is much easier to think of the tube as wrapped around a curve which we shall call the **spine** of the tube that undulates through space in some organized fashion⁵. We shall represent the curve parametrically as $C(t)$. For example, the helix (recall Section 3.8) shown in Figure 6.44a has the parametric representation

stereo pair of helix

Figure 6.44. a helix - stereo pair.

$$C(t) = (\cos(t), \sin(t), bt) \quad (6.11)$$

for some constant b .

To form the various waist polygons of the tube we sample $C(t)$ at a set of t -values, $\{t_0, t_1, \dots\}$, and build a transformed polygon in the plane perpendicular to the curve at each point $C(t_i)$, as suggested in Figure 6.45. It is convenient to think of erecting a local coordinate system at each chosen point along the spine: the local “z-axis” points along the curve, and the local “x and y-axes” point in directions normal to the z-axis (and normal to each other). The waist polygon is set to lie in the local xy-plane. All we need is a straightforward way to determine the vertices of each waist polygon.

Figure 6.45. Constructing local coordinate systems along the spine curve.

It is most convenient to let the curve $C(t)$ *itself* determine the local coordinate systems. A method well-known in differential geometry creates the **Frenet frame** at each point along the spine [gray 93]. At each value t_i of interest a vector $\mathbf{T}(t_i)$ that is tangent to the curve is computed. Then two vectors, $\mathbf{N}(t_i)$ and $\mathbf{B}(t_i)$, which are perpendicular to $\mathbf{T}(t_i)$ and to each other, are computed. These three vectors constitute the **Frenet frame** at t_i .

Once the Frenet frame is computed it is easy to find the transformation matrix M that transforms the base polygon of the tube to its position and orientation in this frame. It is the transformation that carries the world

⁵ The VRML 2.0 modeling language includes an “extrusion” node that works in a similar fashion, allowing the designer to define a “spine” along which the polygons are placed, each with its own transformation.

coordinate system into this new coordinate system. (The reasoning is very similar to that used in Exercise 5.6.1 on transforming the camera coordinate system into the world coordinate system.) The matrix M_i must carry \mathbf{i} , \mathbf{j} , and \mathbf{k} into $\mathbf{N}(t_i)$, $\mathbf{B}(t_i)$, $\mathbf{T}(t_i)$, respectively, and must carry the origin of the world into the spine point $C(t)$. Thus the matrix has columns consisting directly of $\mathbf{N}(t_i)$, $\mathbf{B}(t_i)$, $\mathbf{T}(t_i)$, and $C(t_i)$ expressed in homogeneous coordinates:

$$M_i = (\mathbf{N}(t_i) | \mathbf{B}(t_i) | \mathbf{T}(t_i) | C(t_i)) \quad (6.12)$$

Forming the Frenet frame.

The Frenet frame at each point along a curve depends on how the curve twists and undulates. It is derived from certain derivatives of $C(t)$, and so it is easy to form if these derivatives can be calculated.

Specifically, if the formula that we have for $C(t)$ is differentiable, we can take its derivative and form the tangent vector to the curve at each point, $\dot{\mathbf{C}}(t)$. (If $C(t)$ has components $C_x(t)$, $C_y(t)$, and $C_z(t)$ this derivative is simply $\dot{\mathbf{C}}(t) = (\dot{C}_x(t), \dot{C}_y(t), \dot{C}_z(t))$.) This vector points in the direction the curve “is headed” at each value of t , that is in the direction of the **tangent** to the curve. We normalize it to unit length to obtain the **unit tangent vector** at t . For example, the helix of Equation 6.11 has the unit tangent vector given by

$$\mathbf{T}(t) = \frac{1}{\sqrt{1+b^2}}(-\sin(t), \cos(t), b) \quad (6.13)$$

This tangent is shown for various values of t in Figure 6.46a.

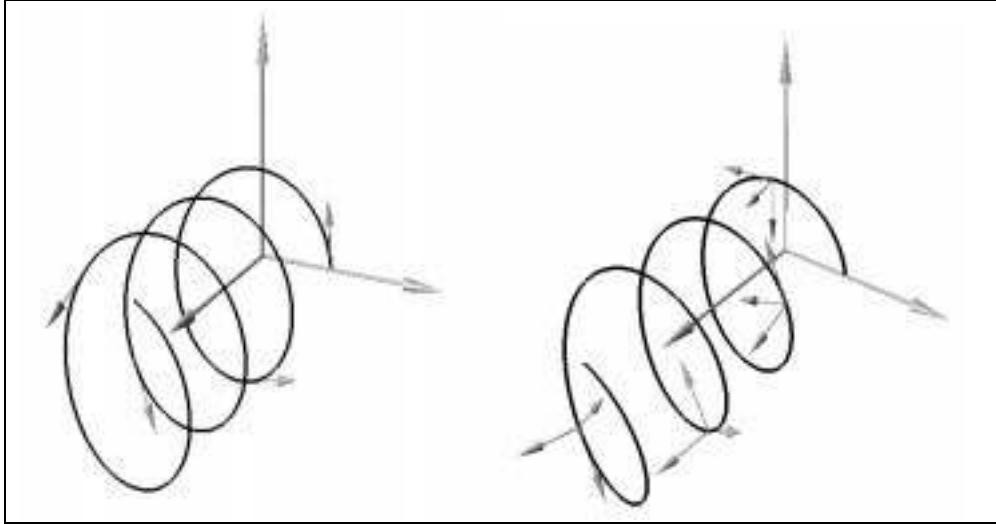


Figure 6.46*. a). Tangents to the helix. B). Frenet frame at various values of t , for the helix.

If we form the cross product of this with any non-collinear vector we must obtain a vector perpendicular to $\mathbf{T}(t)$ and therefore perpendicular to the spine of the curve. (Why?) A particularly good choice is the **acceleration**, based on the second derivative, $\ddot{\mathbf{C}}(t)$. So we form $\dot{\mathbf{C}}(t) \times \ddot{\mathbf{C}}(t)$, and since it will be used for an axis of the coordinate system, we normalize it, to obtain the “unit binormal” vector as:

$$\mathbf{B}(t) = \frac{\dot{\mathbf{C}}(t) \times \ddot{\mathbf{C}}(t)}{|\dot{\mathbf{C}}(t) \times \ddot{\mathbf{C}}(t)|} \quad (6.14)$$

We then obtain a vector perpendicular to both $\mathbf{T}(t)$ and $\mathbf{B}(t)$ by using the cross product again:

$$\mathbf{N}(t) = \mathbf{B}(t) \times \mathbf{T}(t) \quad (6.15)$$

Convince yourself that these three vectors are mutually perpendicular and have unit length, and thus constitute a local coordinate system at $\mathbf{C}(t)$ (known as a **Frenet frame**). For the helix example these vectors are given by:

$$\begin{aligned} \mathbf{B}(t) &= \frac{1}{\sqrt{1+b^2}}(b \sin(t), -b \cos(t), 1) \\ \mathbf{N}(t) &= (-\cos(t), -\sin(t), 0) \end{aligned} \quad (6.16)$$

Figure 6.46b shows the Frenet frame at various values of t along the helix.

Aside: Finding the Frenet frame Numerically.

If the formula for $\mathbf{C}(t)$ is complicated it may be awkward to form its successive derivatives in closed form, such that formulas for $\mathbf{T}(t)$, $\mathbf{B}(t)$, and $\mathbf{N}(t)$ can be hard-wired into a program. As an alternative, it is possible to approximate the derivatives numerically using:

$$\begin{aligned} \dot{\mathbf{C}}(t) &\doteq \frac{\mathbf{C}(t + \varepsilon) - \mathbf{C}(t - \varepsilon)}{2\varepsilon} \\ \ddot{\mathbf{C}}(t) &\doteq \frac{\mathbf{C}(t - \varepsilon) - 2\mathbf{C}(t) + \mathbf{C}(t + \varepsilon)}{\varepsilon^2} \end{aligned} \quad (6.17)$$

This computation will usually produce acceptable directions for $\mathbf{T}(t)$, $\mathbf{B}(t)$, and $\mathbf{N}(t)$, although the user should beware that numerical differentiation is an inherently unstable process [burden85].

Figure 4.47 shows the result of wrapping a decagon about the helix in this way. The helix was sampled at 30 points, a Frenet frame was constructed at each point, and the decagon was erected in the new frame.

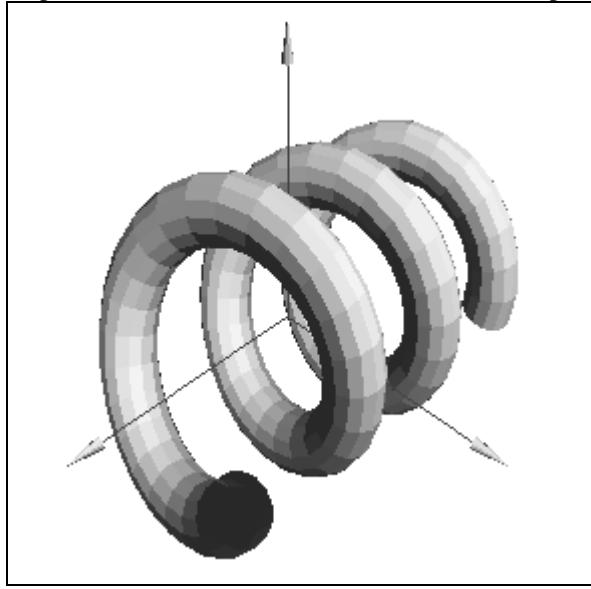


Figure 6.47. A tube wrapped along a helix.

Figure 4.48 shows other interesting examples, based on the toroidal spiral (which we first saw in Section 3.8.) [gray93, p.212]. (The edges of the individual faces are drawn to clarify how the tube twists as it proceeds. Drawing the edges of a mesh is considered in Case Study 6.7.) A toroidal spiral is formed when a spiral is wrapped about a torus (try to envision the underlying invisible torus here), and it is given by

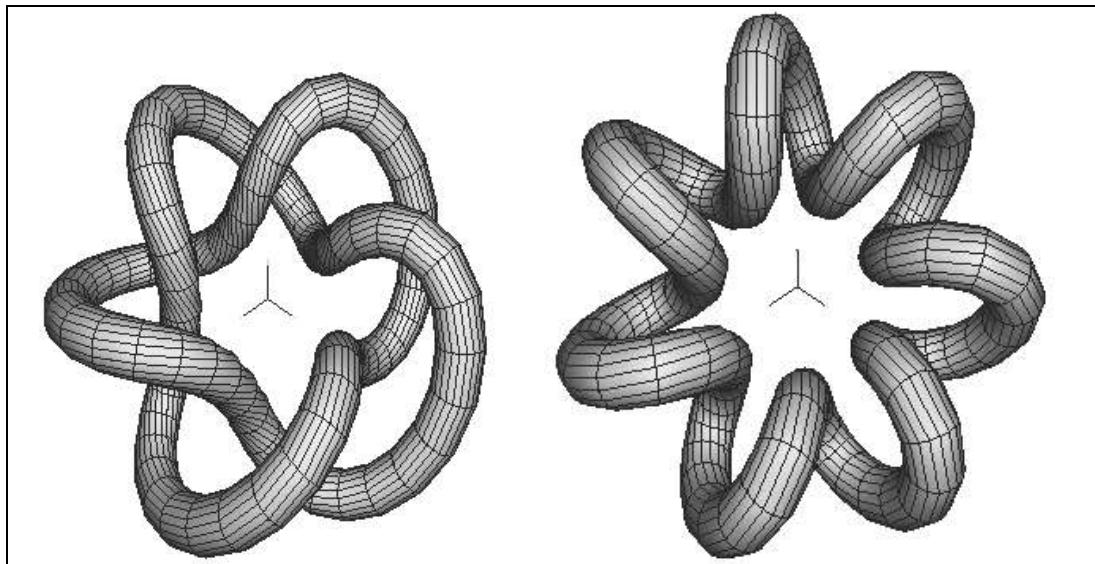


Figure 6.48. Tubes based on toroidal spirals. (file: torusKnot.bmp, file: torusKnot7.bmp)

$$C(t) = ((a + b \cos(qt)) \cos(pt), (a + b \cos(qt)) \sin(pt), c \sin(qt)) \quad (6.18)$$

for some choice of constants a , b , p , and q . For part a the parameters p and q were chosen as 2 and 5, and for part b they are chosen to be 1 and 7.

Figure 6.49 shows a “sea shell”, formed by wrapping a tube with a growing radius about a helix. To accomplish this, the matrix of Equation 6.12 was multiplied by a scaling matrix, where the scale factors also depend on t :

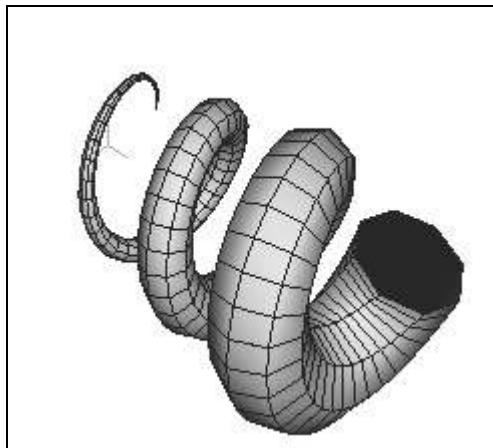


Figure 6.49. A “sea shell”.

$$M' = M \begin{pmatrix} g(t) & 0 & 0 & 0 \\ 0 & g(t) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Here $g(t) = t$. It is also possible to add a rotation to the matrix, so that the tube appears to twist more vigorously as one looks along the spine.

One of the problems with using Frenet frames for sweeping curves is that the local frame sometimes twists in such a way as to introduce undesired “knots” in the surface. Recent work, such as [wang97], finds alternatives to the Frenet frame that produce less twisting and therefore more graceful surfaces.

Practice Exercises.

6.4.4. What is $N(t)$? Show that $N(t)$ is parallel to $\ddot{\mathbf{C}}(t) - (\dot{\mathbf{C}}(t) \cdot \ddot{\mathbf{C}}(t))\dot{\mathbf{C}}(t) / |\dot{\mathbf{C}}(t)|^2$, so it points in the direction of the acceleration when the velocity and acceleration at t are perpendicular.

6.4.5. The Frame for the helix. Consider the circular helix treated in the example above. Show that the formulas above for the unit tangent, binormal, and normal vectors are correct. Also show that these are unit length and mutually perpendicular. Visualize how this local coordinate system orients itself as you move along the curve.

Figure 6.50 shows additional examples. Part (a) shows a hexagon wrapped about an elliptical spine to form a kind of elliptical torus, and part (b) shows segments arranged into a knot along a Lissajous figure given by:

$$\mathbf{C}(t) = (r \cos(Mt + \phi), 0, r \sin(Nt)) \quad (6.19)$$

with $M = 2, N = 3, \phi = 0$.

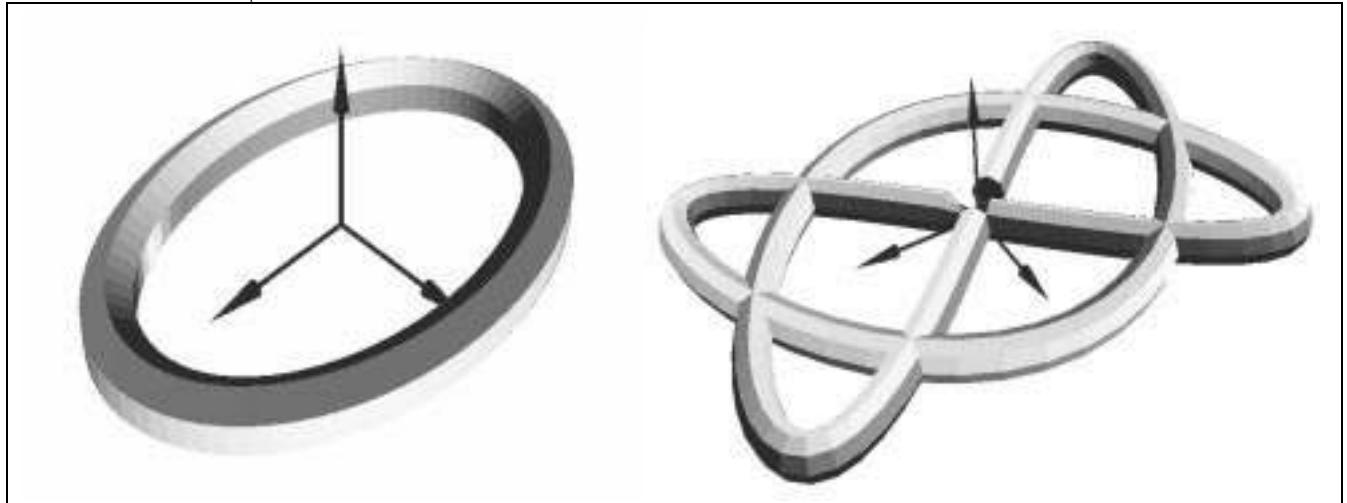


Figure 6.50. a). A hexagon wrapped about an elliptical torus. b). A 7-gon wrapped about a Lissajous figure.

Case Study 6.5 examines more details of forming meshes that model tubes based on a parametric curve.

6.4.5. “Discretely” Swept Surfaces of Revolution.

The tubes above use affine transformations to fashion a new coordinate system at each spine point. If we use pure rotations for the affine transformations, and place all spine points at the origin, a rich set of polyhedral shapes emerges. Figure 6.51 shows an example, where a base polygon - now called the **profile** - is initially positioned 3 units out along the x-axis, and then is successively rotated in steps about the y-axis to form an approximation of a torus.

Figure 6.51. Rotational sweeping in discrete steps.

This is equivalent to **circularly sweeping** a shape about an axis, and the resulting shape is often called a **surface of revolution**. We examine true surfaces of revolution in Section 6.5; here we are forming only a discrete approximation to them since we are sweeping in discrete steps.

Figure 6.52 shows an example that produces pictures of a martini glass. The profile here is not a closed polygon but a simple polyline based on points $P_j = (x_j, y_j, 0)$. If we choose to place this polyline at K equispaced angles about the y -axis, we set the transformations to have matrices:

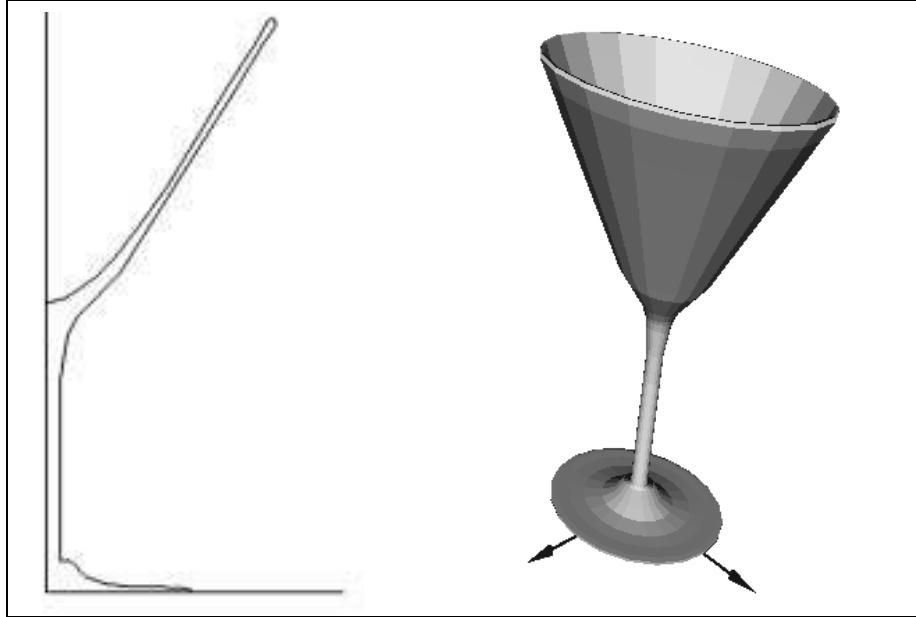


Figure 6.52. Approximating a martini glass with a discretely swept polyline. a). the profile, b). the swept surface.

$$\tilde{M}_i = \begin{pmatrix} \cos(\theta_i) & 0 & \sin(\theta_i) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_i) & 0 & \cos(\theta_i) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where $\theta_i = 2\pi i/K$, $i = 0, 1, \dots, K-1$. Note there is no translation involved. This transformation is simple enough that we can write the positions of the vertices directly. The rotation sets the points of the i -th “waist” polyline at:

$$(x_j \cos(\theta_i), y_j, x_j \sin(\theta_i)) \quad (6.20)$$

Building meshes that model surfaces of revolution is treated further in Case Study 6.6.

6.5. Mesh Approximations to Smooth objects.

So far we have built meshes to represent polyhedra, where each shape is a collection of flat polygonal faces. They tend to be “data-intensive”, specified by listing the vertices of each face individually. Now we want to build meshes that attempt to approximate inherently smooth shapes like a sphere or torus. These shapes are normally defined by formulas rather than data. We also want to arrange matters so that these meshes can be *smoothly shaded*: even though they are represented by a collection of flat faces as before, the proper algorithm (Gouraud shading) draws them with smooth gradations in shading, and the individual faces are invisible (recall Figure 6.1). All this requires is that we find the proper normal vector at each vertex of each face. Specifically,

we compute the normal vector to the underlying smooth surface. We discuss the Gouraud shading algorithm in Chapter 11.

The basic approach for each type of surface is to **polygonalize** (also called **tessellate**) it into a collection of flat faces. If the faces are small enough and there is a graceful change in direction from one face to the next, the resulting mesh will provide a good approximation to the underlying surface. The faces have vertices that are found by evaluating the surface's parametric representation at discrete points. A mesh is created by building a vertex list and face list in the usual way, except here the vertices are computed from formulas. The same is true for the vertex normal vectors: they are computed by evaluating formulas for the normal to the surface at discrete points.

6.5.1. Representations for Surfaces.

To set the stage, recall that in Section 4.5.5 we examined the planar **patch**, given parametrically by

$$P(u, v) = C + \mathbf{a}u + \mathbf{b}v \quad (6.21)$$

where C is a point, and \mathbf{a} and \mathbf{b} are vectors. The range of values for the parameters u and v is usually restricted to $[0, 1]$, in which case the patch is a parallelogram in 3D with corner vertices C , $C + \mathbf{a}$, $C + \mathbf{b}$, and $C + \mathbf{a} + \mathbf{b}$, (recall Figure 4.31).

Here we enlarge our interests to nonlinear forms, to represent more general surface shapes. We introduce three functions $X()$, $Y()$, and $Z()$ so that the surface has parametric representation in point form

$$P(u, v) = (X(u, v), Y(u, v), Z(u, v)) \quad (6.22)$$

with u and v restricted to suitable intervals. Different surfaces are characterized by different functions, X , Y , and Z . The notion is that the surface is “at” $(X(0, 0), Y(0, 0), Z(0, 0))$ when both u and v are zero, at $(X(1, 0), Y(1, 0), Z(1, 0))$ when $u = 1$ and $v = 0$, and so on. Keep in mind that two parameters are required when representing a surface, whereas a curve in 3D requires only one. Letting u vary while keeping v constant generates a curve called a **v -contour**. Similarly, letting v vary while holding u constant produces a **u -contour**. (Look ahead to Figure 6.58 to see examples u - and v -contours.)

The implicit form of a surface.

Although we are mainly concerned with parametric representations of different surfaces, it will prove useful to keep track of an alternative way to describe a surface, through its **implicit form**. Recall from Section 3.8 that a curve in 2D has an implicit form $F(x, y)$ which must evaluate to 0 for all points (x, y) that lie on the curve, and for only those. For surfaces in 3D a similar function $F(x, y, z)$ exists that evaluates to 0 if and only if the point (x, y, z) is on the surface. The surface therefore has an **implicit equation** given by

$$F(x, y, z) = 0 \quad (6.23)$$

that is satisfied for all points on the surface, and only those. The equation constrains the way that values of x , y , and z must be related to confine the point (x, y, z) to the surface in question. For example, recall (from Chapter 4) that the plane that passes through point B and has normal vector \mathbf{n} is described by the equation $n_x x + n_y y + n_z z = D$ (where $D = \mathbf{n} \cdot \mathbf{B}$), so the implicit form for this plane is $F(x, y, z) = n_x x + n_y y + n_z z - D$. Sometimes it is more convenient to think of F as a function of a point P , rather than a function of three variables x , y , and z , and we write $F(P) = 0$ to describe all points that lie on the surface. For the example of the plane here, we would define $F(P) = \mathbf{n} \cdot (P - B)$ and say that P lies in the plane if and only if $F(P) = \mathbf{n} \cdot (P - B)$ is zero. If we wish to work with coordinate frames (recall Section 4.5) so that \tilde{P} is the 4-tuple $\tilde{P} = (x, y, z, 1)^T$, the implicit form for a plane is even simpler: $F(\tilde{P}) = \tilde{\mathbf{n}} \cdot \tilde{P}$, where $\tilde{\mathbf{n}} = (n_x, n_y, n_z, -D)$ captures both the normal vector and the value $-D$.

It is not always easy to find the function $F(x, y, z)$ or $F(P)$ from a given parametric form (nor can you always find a parametric form when given $F(x, y, z)$). But if both a parametric form and an implicit form are available it is simple to determine whether they describe the same surface. Simply substitute $X(u, v)$, $Y(u, v)$, and $Z(u, v)$ for x , y , and z , respectively, in $F(x, y, z)$ and check that F is 0 for all values of u and v of interest.

For some surfaces like a sphere that enclose a portion of space it is meaningful to define an “inside” region and an “outside” region. Other surfaces like a plane clearly divide 3D space into two regions, but one must refer to the context of the application to tell which half-space is the inside and which the outside. There are also many surfaces, such as a strip of ribbon candy, for which it makes little sense to name an inside and an outside.

When it is meaningful to designate an inside and outside to a surface, the implicit form $F(x, y, z)$ of a surface is also called its **inside outside function**. We then say that a point (x, y, z) is

- inside the surface if: $F(x, y, z) < 0$
 - on the surface if: $F(x, y, z) = 0$
 - outside the surface if: $F(x, y, z) > 0$
- (6.24)

This provides a quick and simple test for the disposition of a given point (x', y', z') relative to the surface: Just evaluate $F(x', y', z')$ and test whether it is positive, negative, or zero. This is seen to be useful in hidden line and hidden surface removal algorithms in Chapter 14, and in Chapter 15 it is used in ray tracing algorithms. There has also been vigorous recent activity in rendering surfaces directly from their implicit forms: see [bloomenthal, 97].

6.5.2. The Normal Vector to a Surface.

As described earlier, we need to determine the direction of the normal vector to a surface at any desired point. Here we present one way based on the parametric expression, and one based on the implicit form, of the surface. As each surface type is examined later, we find the suitable expressions for its normal vector at any point.

The normal direction to a surface can be defined at a point, $P(u_0, v_0)$, on the surface by considering a very small region of the surface around $P(u_0, v_0)$. If the region is small enough and the surface varies “smoothly” in the vicinity, the region will be essentially flat. Thus it behaves locally like a tiny planar patch and has a well-defined normal direction. Figure 6.53 shows a surface patch with the normal vector drawn at various points. The direction of the normal vector is seen to be different at different points on the surface.

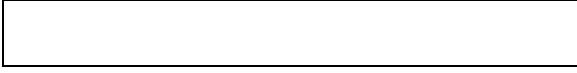


Figure 6.53*. The normal vector to a surface.

We use the name $\mathbf{n}(u, v)$ for the normal at (u, v) . We now examine how it can be calculated.

The Normal Vector for a Surface Given Parametrically.

Not surprisingly, $\mathbf{n}(u_0, v_0)$ takes the form of a cross product between two vectors that lie in the tiny planar patch near (u_0, v_0) . Being a cross product it is guaranteed to be perpendicular to both vectors. The two vectors in the plane (indicated as \mathbf{t}_u and \mathbf{t}_v in the figure) are certain tangent vectors. Calculus texts show that they are simply related to partial derivatives of $\mathbf{p}(u, v)$ (the vector from the origin to the surface point $P(u, v)$ ⁶), evaluated at the point in question [thomas53]. An expression for the normal vector is therefore

⁶ Since $\mathbf{p}(u, v)$ is simply the difference $P(u, v) - (0,0,0)$, the derivative of $\mathbf{p}()$ is the same as that of $P()$.

$$\mathbf{n}(u_0, v_0) = \left(\frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v} \right) \Big|_{u=u_0, v=v_0} \quad (6.25)$$

where the vertical bar | indicates that the derivatives are evaluated at $u = u_0$, $v = v_0$. Formed this way, $\mathbf{n}(u_0, v_0)$ is not automatically a unit length vector, but it can be normalized if desired.

Example 6.5.1: Does this work for a plane? Consider the plane given parametrically by $P(u, v) = C + \mathbf{a}u + \mathbf{b}v$. The partial derivative of this with respect to u is just \mathbf{a} , and that with respect to v is \mathbf{b} . Thus according to Equation 6.19, $\mathbf{n}(u, v) = \mathbf{a} \times \mathbf{b}$, which we recognize as the correct result.

More generally, the partial derivatives of $\mathbf{p}(u, v)$ exist whenever the surface is “smooth enough.” Most of the surfaces of interest to us in modeling scenes have the necessary smoothness and have simple enough mathematical expressions so that finding the required derivatives is not difficult. Because $\mathbf{p}(u, v) = X(u, v)\mathbf{i} + Y(u, v)\mathbf{j} + Z(u, v)\mathbf{k}$, the derivative of a vector is just the vector of the individual derivatives:

$$\frac{\partial \mathbf{p}(u, v)}{\partial u} = \left(\frac{\partial X(u, v)}{\partial u}, \frac{\partial Y(u, v)}{\partial u}, \frac{\partial Z(u, v)}{\partial u} \right) \quad (6.26)$$

We apply these formulas directly to each surface type we examine later.

The Normal Vector for a Surface given Implicitly.

An alternative expression is used if the surface is given by an implicit form, $F(x, y, z) = 0$. The normal direction at the surface point, (x, y, z) , is found using the **gradient**,

∇F , of F , which is given by [thomas 53]:

$$\mathbf{n}(x_0, y_0, z_0) = \nabla F \Big|_{x=x_0, y=y_0, z=z_0} = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right) \Big|_{x=x_0, y=y_0, z=z_0} \quad (6.27)$$

where each partial derivative is evaluated at the desired point, (x_0, y_0, z_0) . If the point (x_0, y_0, z_0) for the surface in question corresponds to the point $P(u_0, v_0)$ of the parametric form, then $\mathbf{n}(x_0, y_0, z_0)$ has the same direction as $\mathbf{n}(u_0, v_0)$ in Equation 6.19, but it may have a different length. Again, it can be normalized if desired.

Example 6.5.2. The plane again: Consider once again the plane with normal \mathbf{n} that passes through point A , given implicitly by $F(x, y, z) = \mathbf{n} \cdot ((x, y, z) - A) = 0$, or $n_x x + n_y y + n_z z - \mathbf{n} \cdot A = 0$. This has gradient $\nabla F = \mathbf{n}$ as expected.

Note that the gradient-based form gives the normal vector as a function of x , y , and z , rather than of u and v . Sometimes for a surface we know *both* the inside-outside function, $F(x, y, z)$, and the parametric form, $\mathbf{p}(u, v) = X(u, v)\mathbf{i} + Y(u, v)\mathbf{j} + Z(u, v)\mathbf{k}$. In such cases it may be easiest to find the parametric form, $\mathbf{n}(u, v)$, of the normal at (u, v) by a two-step method: (1) Use Equation 6.21 to get the normal at (x, y, z) in terms of x , y , and z , and then (2) substitute the known functions $X(u, v)$ for x , $Y(u, v)$ for y , and $Z(u, v)$ for z . Some of the later examples illustrate this method.

6.5.3. The Effect of an Affine Transformation.

We shall need on occasion to work with the implicit and parametric forms of a surface after the surface has been subjected to an affine transformation. We will also want to know how the normal to the surface is affected by the transformation.

Suppose the transformation is represented by 4 by 4 matrix M , and that the original surface has implicit form (in terms of points in homogeneous coordinates) $F(\tilde{P})$ and parametric form $\tilde{P}(u, v) = (X(u, v), Y(u, v), Z^*(u, v), 1)^T$. Then it is clear that the transformed surface has para-

metric form $M\tilde{P}(u, v)$ (why?). It is also easy to show (see the exercises) that the transformed surface has implicit form $F'(\tilde{P})$ given by:

$$F'(\tilde{P}) = F(M^{-1}\tilde{P})$$

Further, if the original surface has normal vector $\mathbf{n}(u, v)$ then the transformed surface has:

- normal vector: $M^{-T}\mathbf{n}(u, v)$

For example, suppose we transform the plane examined above, given by $F(\tilde{P}) = \tilde{n} \cdot \tilde{P}$ with $\tilde{n} = (n_x, n_y, n_z, -D)$. The transformed plane has implicit form $F'(\tilde{P}) = \tilde{n} \cdot (M^{-1}\tilde{P})$. This can be written (see the exercises) as $(M^{-T}\tilde{n}) \cdot \tilde{P}$, so the normal vector of the transformed plane involves the inverse transpose of the matrix, consistent with the claimed form for the normal to a general surface.

Practice Exercises.

6.5.1. The implicit form of a transformed surface. Suppose all points on a surface satisfy $F(P) = 0$, and that M transforms \tilde{P} into \tilde{Q} ; i.e. $\tilde{Q} = M\tilde{P}$. Then argue that any point \tilde{Q} on the transformed surface comes from a point $M^{-1}\tilde{Q}$, and those points all satisfy $F(M^{-1}\tilde{Q}) = 0$. Show that this proves that the implicit form for the transformed surface is $F'(\tilde{Q}) = F(M^{-1}\tilde{Q})$.

6.5.2. How are normal vectors affected? Let $\mathbf{n} = (n_x, n_y, n_z, 0)^T$ be the normal at P and let \mathbf{v} be any vector tangent to the surface at P . Then \mathbf{n} must be perpendicular to \mathbf{v} and we can write $\mathbf{n} \bullet \mathbf{v} = 0$.

- Show that the dot product can be written as a matrix product: $\mathbf{n}^T \mathbf{v} = 0$ (see Appendix 2).
- Show that this is still 0 when the matrix product $M^T M$ is inserted: $\mathbf{n}^T M^T M \mathbf{v} = 0$.
- Show that this can be rewritten as $(M^T \mathbf{n})(M \mathbf{v}) = 0$, so $M^T \mathbf{n}$ is perpendicular to $(M \mathbf{v})$.

Now since the tangent \mathbf{v} transforms to $M \mathbf{v}$, which is tangent to the transformed surface, show that this says $M^T \mathbf{n}$ must be normal to the transformed surface, which we wished to show.

d). The normal to a surface is also given by the gradient of the implicit form, so the normal to the transformed surface at point P must be the gradient of $F(M^T P)$. Show (by the chain rule of calculus) that the gradient of this function is M^T multiplied onto the gradient of $F()$.

6.5.3. The tangent plane to a transformed surface. To find how normal vectors are transformed we can also find how the tangent plane to a surface is mapped to the tangent plane on the transformed surface. Suppose the tangent plane to the original surface at point P has parametric representation $P + a\mathbf{u} + b\mathbf{v}$, where \mathbf{a} and \mathbf{b} are two vectors lying in the plane. The normal to the surface is therefore $\mathbf{n} = \mathbf{a} \times \mathbf{b}$.

- Show that the parametric representation of the transformed plane is $M\mathbf{P} + Ma\mathbf{u} + Mb\mathbf{v}$, and that this plane has normal $\mathbf{n}' = (M\mathbf{a}) \times (M\mathbf{b})$.
- Referring to Appendix 2, show the following identity:

$$(M\mathbf{a}) \times (M\mathbf{b}) = (\det M) M^{-T} (\mathbf{a} \times \mathbf{b})$$

This relates the cross product of transformed vectors to the cross product of the vectors themselves. c). Show that \mathbf{n}' is therefore parallel to $M^T \mathbf{n}$.

6.5.4. Three “generic” shapes: the sphere, cylinder, and cone.

We begin with three classic objects, “generic” versions of the sphere, cylinder, and cone. We develop the implicit form and parametric form for each of these, and see how one might make meshes to approximate them. We also derive formulas for the normal direction at each point on these objects. Note that we have already used OpenGL functions in Chapter 5 to draw these shapes. Adding our own tools has several advantages, however: a). We have much more control over the detailed nature of the shape

being created; b). We have the object as an actual mesh that can be operated upon by methods of the Mesh class.

The Generic Sphere.

We call the sphere of unit radius centered at the origin the “generic sphere” (see Figure 6.54a). It forms the basis for all other sphere-like shapes we use. It has the familiar implicit form

$$F(x, y, z) = x^2 + y^2 + z^2 - 1 \quad (6.28)$$

In the alternate notation $F(P)$ we obtain the more elegant $F(P) = |P|^2 - 1$. (What would these forms be if the sphere has radius R ?)

A parametric description of this sphere comes immediately from the basic description of a point in spherical coordinates (see Appendix 2). We choose to let u correspond to azimuth and v correspond to latitude. Then any point $P = (x, y, z)$ on the sphere has the representation $(\cos(v) \cos(u), \cos(v) \sin(u), \sin(v))$ in spherical coordinates (see Figure 6.54b). We let u vary over $(0, 2\pi)$ and v vary over $(-\pi/2, \pi/2)$ to cover all such points. A parametric form for the sphere is therefore

- | | | |
|--------------------|--|---------------------|
| a). generic sphere | b). parametrized by azimuth, latitude. | c). par's & merid's |
|--------------------|--|---------------------|

Figure 6.54. a). The generic sphere, b) a parametric form. c). parallels and meridians

$$P(u, v) = (\cos(v) \cos(u), \cos(v) \sin(u), \sin(v)) \quad (6.29)$$

It's easy to check that this is consistent with the implicit form: substitute terms of Equation 6.29 into corresponding terms of Equation 6.28 and see that zero is obtained for *any* value of u and v .

(Question: What is the corresponding parametric form if the sphere instead has radius R and is centered at (a, b, c) ?)

For geographical reasons certain contours along a sphere are given common names. For this parametrization u -contours are called **meridians**, and v -contours are known as **parallels**, as suggested in Figure 6.54c. (Note that this classical definition of spherical coordinates, parallels and meridians causes the sphere to appear to lie on its side. This is simply a result of how we sketch 3D figures, with the y -axis pointing “up”.)

Different parametric forms are possible for a given shape. An alternative parametric form for the sphere is examined in the exercises.

What is the normal direction $\mathbf{n}(u, v)$ of the sphere's surface at the point specified by parameters (u, v) ? Intuitively the normal vector is always aimed “radially outward”, so it must be parallel to the vector from the origin to the point itself. This is confirmed by Equation 6.21: the gradient is simply $2(x, y, z)$, which is proportional to P . Working with the parametric form, Equation 6.19 yields $\mathbf{n}(u, v) = -\cos(v)\mathbf{p}(u, v)$, so $\mathbf{n}(u, v)$ is parallel to $\mathbf{p}(u, v)$ as expected. The scale factor $-\cos(v)$ will disappear when we normalize \mathbf{n} . We must make sure to use $\mathbf{p}(u, v)$ rather than $-\mathbf{p}(u, v)$ for the normal, so that it does indeed point radially outward.

The Generic Cylinder.

We adopt as the “generic” cylinder the cylinder whose axis coincides with the z -axis, has a circular cross section of radius 1, and extends in z from 0 to 1, as pictured in Figure 6.55a.

- | | |
|----------------------|----------------------|
| a). generic cylinder | b). tapered cylinder |
|----------------------|----------------------|

Figure 6.55. The generic cylinder and the tapered cylinder.

It is convenient to view this cylinder as one member of the large family of **tapered cylinders**, as we did in Chapter 5. Figure 6.55b shows the “generic” tapered cylinder, having a “small radius” of s when $z = 1$. The generic cylinder is simply a tapered cylinder with $s = 1$. Further, the generic cone to be examined next is simply a tapered cylinder with $s = 0$. We develop formulas for the tapered cylinder with an arbitrary value of s . These provide formulas for the generic cylinder and cone by setting s to 1 or 0, respectively.

If we consider the tapered cylinder to be a thin hollow “shell”, its **wall** is given by the implicit form

$$F(x, y, z) = x^2 + y^2 - (1 + (s - 1)z)^2 \quad \text{for } 0 < z < 1 \quad (6.30)$$

and by the parametric form:

$$P(u, v) = ((1 + (s - 1)v)\cos(u), (1 + (s - 1)v)\sin(u), v) \quad (6.31)$$

for appropriate ranges of u and v (which ones?). What are these expressions for the generic cylinder with $s = 1$?

When it is important to model the tapered cylinder as a solid object, we add two circular discs at its ends: a **base** and a **cap**. The cap is a circular portion of the plane $z = 1$, characterized by the inequality $x^2 + y^2 < s^2$, or given parametrically by $P(u, v) = (v \cos(u), v \sin(u), 1)$ for v in $[0, s]$. (What is the parametric representation of the base?)

The normal vector to the wall of the tapered cylinder is found using Equation 6.27. (Be sure to check this). It is

$$\mathbf{n}(x, y, z) = (x, y, -(s - 1)(1 + (s - 1)z)) \quad (6.32)$$

or in parametric form $\mathbf{n}(u, v) = (\cos(u), \sin(u), 1 - s)$. For the generic cylinder the normal is simply $(\cos(u), \sin(u), 0)$. This agrees with intuition: the normal is directed radially away from the axis of the cylinder. For the tapered cylinder it is also directed radially, but shifted by a constant z -component. (What are the normals to the cap and base?)

The Generic Cone.

We take as the “generic” cone the cone whose axis coincides with the z -axis, has a circular cross section of maximum radius 1, and extends in z from 0 to 1, as pictured in Figure 6.56. It is a tapered cylinder with small radius of $s = 0$. Thus its wall has implicit form

generic cone

Figure 6.56. The generic cone.

$$F(x, y, z) = x^2 + y^2 - (1 - z)^2 = 0 \quad \text{for } 0 < z < 1 \quad (6.33)$$

and parametric form $P(u, v) = ((1-v)\cos(u), (1-v)\sin(u), v)$ for azimuth u in $[0, 2\pi]$ and v in $[0, 1]$. Using the results for the tapered cylinder again, the normal vector to the wall of the cone is $(x, y, 1-z)$. What is it parametrically?

For easy reference Figure 6.57 shows the normal vector to the generic surfaces we have discussed.

surface	$\mathbf{n}(u, v)$ at $\mathbf{p}(u, v)$	$\nabla F(x, y, z)$
sphere	$\mathbf{p}(u, v)$	(x, y, z)
tapered cylinder	$(\cos(u), \sin(u), 1 - s)$	$(x, y, -(s - 1)(1 + (s - 1)z))$

cylinder	$(\cos(u), \sin(u), 0)$	$(x, y, 0)$
cone	$(\cos(u), \sin(u), 1)$	$(x, y, 1 - z)$

Figure 6.57. Normal vectors to the generic surfaces.

Practice Exercises.

6.5.4. Alternative representation for the generic sphere. We can associate different geometric quantities with the parameters u and v and obtain a different parametric form for the sphere. We again use parameter u for azimuth, but use v for the height of the point above the xy -plane. All points at height v lie on a circle of radius $\sqrt{1-v^2}$, so this parametric form is given by:

$$P_2(u, v) = (\sqrt{1-v^2} \cos(u), \sqrt{1-v^2} \sin(u), v) \quad (6.34)$$

for u in $[0, 2\pi]$ and v in $[-1, 1]$. Show that P_2 lies unit distance from the origin for all u and v .

6.5.5. What's the surface? Let A be a fixed point with position vector \mathbf{a} , and P be an arbitrary point with position vector \mathbf{p} . Describe in words and sketch the surface described by: a). $\mathbf{p} \bullet \mathbf{a} = 0$; b). $\mathbf{p} \bullet \mathbf{a} = |\mathbf{a}|$; c). $|\mathbf{p} \times \mathbf{a}| = |\mathbf{a}|$; d). $\mathbf{p} \bullet \mathbf{a} = \mathbf{p} \bullet \mathbf{p}$; e). $\mathbf{p} \bullet \mathbf{a} = |\mathbf{a}||\mathbf{p}|/2$.

6.5.6. Finding the normal vector to the generic cylinder and cone. Derive the normal vector for the generic tapered cylinder and the generic cone in two ways:

- a). Using the parametric representation;
- b). Using the implicit form, then expressing the result parametrically.

6.5.7. Transformed Spheres. Find the implicit form for a generic sphere that has been scaled in x by 2 and in y by 3, and then rotated 30° about the z -axis.

6.5.5. Forming a Polygonal Mesh for a Curved Surface.

Now we examine how to make a mesh object that approximates a smooth surface such as the sphere, cylinder, or cone. The process is called **polygonalization** or **tesselation**, and it involves replacing the surface by a collection of triangles and quadrilaterals. The vertices of these polygons lie in the surface itself, and they are joined by straight edges (which usually do not lie in the surface). One proceeds by choosing a number of values of u and v , and “sampling” the parametric form for the surface at these values to obtain a collection of vertices. These vertices are placed in a vertex list. A face list is then created: each face consists of three or four indices pointing to suitable vertices in the vertex list. Associated with each vertex in a face is the normal vector to the surface. This normal vector is the normal direction to the true underlying surface at each vertex location. (Note how this contrasts with the normal used when representing a flat-faced polyhedron: there the vertex of each face is associated with the normal to the face.)

Figure 6.58 shows how this works for the generic sphere. We think of slicing up the sphere along azimuth lines and latitude lines. Using OpenGL terminology of “slices” and “stacks” (see Section 5.6.3), we choose to slice the sphere into `nSlices` “slices” around the equator, and `nStacks` “stacks” from the south pole to the north pole. The figure shows the example of 12 slices and 8 stacks. The larger `nSlices` and `nStacks` are, the better the mesh approximates a true sphere.

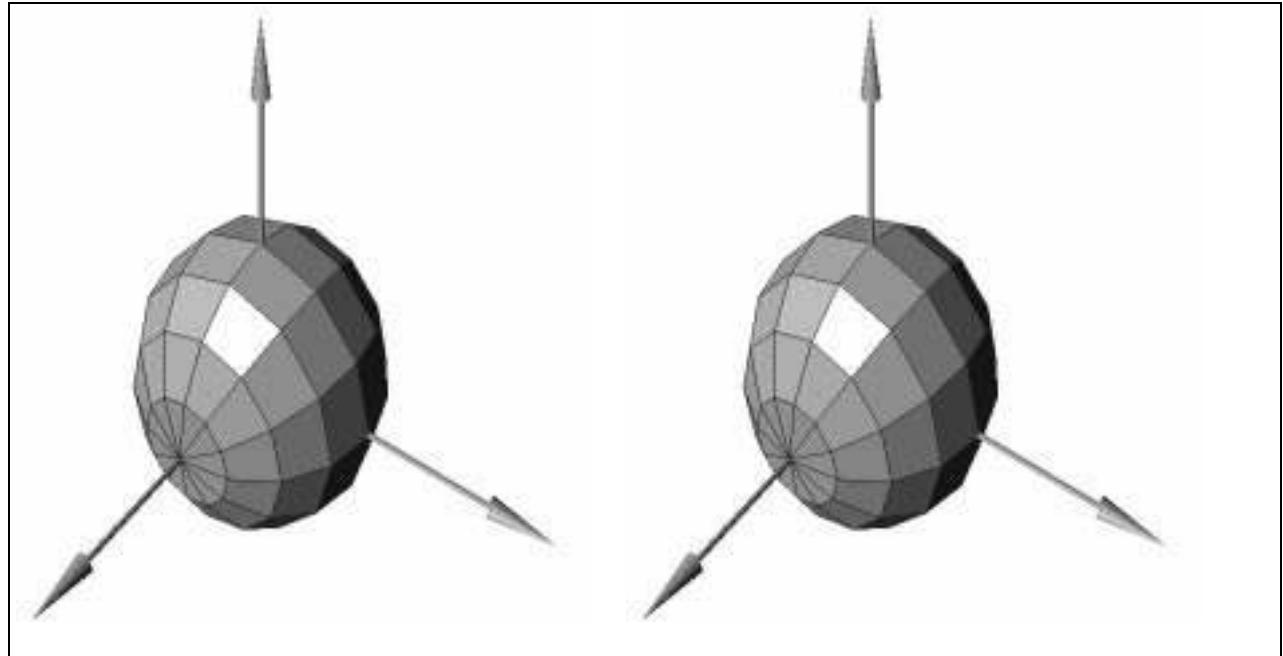


Figure 6.58*. a). A mesh approximation to the generic sphere. b). Numbering the vertices.

To make slices we need `nSlices` values of u between 0 and 2π . Usually these are chosen to be equi-spaced: $u_i = 2\pi i / nSlices$, $i = 0, 1, \dots, nSlices - 1$. As for stacks, we put half of them above the equator and half below. The top and bottom stacks will consist of triangles; all other faces will be quadrilaterals. This requires we define $(nStacks + 1)$ values of latitude: $v_j = \pi - \pi j / nStacks$, $j = 0, 1, \dots, nStacks$.

The vertex list can now be created. The figure shows how we might number the vertices (the ordering is a matter of convenience): We put the north pole in `pt[0]`, the bottom points of the top stack into the next 12 vertices, etc. With 12 slices and 8 stacks there will be a total of 98 points (why?).

The normal vector list is also easily created: `norm[k]` will hold the normal for the sphere at vertex `pt[k]`. `norm[k]` is computed by evaluating the parametric form of $\mathbf{n}(u, v)$ at the same (u, v) used for the points. For the sphere this is particularly easy since `norm[k]` is the same as `pt[k]`.

For this example the face list will have 96 faces, of which 24 are triangles. We can put the top triangles in the first 12 faces, the 12 quadrilaterals of the next stack down in the next 12 faces, etc. The first few faces will contain the data:

number of vertices:	3	3	3	...
vertex indices:	0 1 2	0 2 3	0 3 4	...
normal indices:	0 1 2	0 2 3	0 3 4	...

Note that for all meshes that try to represent smooth shapes the `normIndex` is always the same as the `vertIndex`, so the data structure holds redundant information. (Because of this, one could use a more streamlined data structure for such meshes. What would it be?) Polygonalization of the sphere in this way is straightforward, but for more complicated shapes it can be very tricky. See [refs] for further discussions.

Ultimately we need a method, such as `makeSurfaceMesh()`, that generates such meshes for a given surface $P(u, v)$. We discuss the implementation of such a function in Case Study 6.13.

Note that some graphics packages have routines that are highly optimized when they operate on triangles. To exploit these we might choose to polygonalize the sphere into a collection of triangles, subdividing each quadrilateral into two triangles.

A simple approach would use the same vertices as above, but alter the face list replacing each quadrilateral with two triangles. For instance, a face that uses vertices 2, 3, 15, 14 might be subdivided into two triangles, once using 2, 3, 15 and the other using 2, 15, 14.

The sphere is a special case of a surface of revolution, which we treat in Section 6.5.7. The tapered cylinder is also a surface of revolution. It is straightforward to develop a mesh model for the tapered cylinder. Figure 6.59 shows the tapered cylinder approximated with `nSlices = 10` and `nStacks = 1`. A decagon is used for its cap and base. (If you prefer to use only triangles in a mesh, the walls, the cap, and the base could be dissected into triangles. (How?))

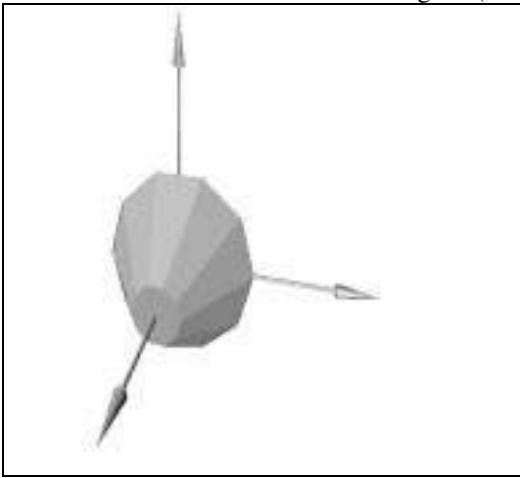


Figure 6.59. A mesh approximation to the tapered cylinder.

Practice Exercises.

6.5.8. The mesh for a given sphere. Write out the vertex, normal, and face lists for the sphere, when `nSlices = 6` and `nStacks = 4`, choosing a convenient numbering scheme.

6.5.9. Restricting the mesh to triangular faces. Adjust the lists in the previous exercise for the case where all faces are triangles.

6.5.10. The mesh for a cylinder and cone. Write out vertex, normal, and face lists for the generic tapered cylinder that uses `nSlices = 4` and `nStacks = 2`.

6.5.6. Ruled Surfaces.

We resume an exploration of curved surfaces with the family of **ruled surfaces**. This family is simple to describe yet provides a wide variety of useful and interesting shapes. We study how to describe them, polygonalize them and how to compute the normal vector to them at each point.

Ruled surfaces (also called "lofted surfaces") are swept out by moving a straight line along a particular trajectory. They are composed of a collection of straight lines in the following sense:

Definition: A surface is **ruled** if through every one of its points there passes at least one line that lies entirely on the surface.

Because ruled surfaces are based on a family of lines, it is not surprising to find buried in their parametric representations something akin to the familiar form for a line, $P(v) = (1 - v) P_0 + v P_1$, where P_0 and P_1

are points. But for ruled surfaces the points P_0 and P_1 become functions of another parameter u : P_0 becomes $P_0(u)$, and P_1 becomes $P_1(u)$. Thus the ruled surfaces that we examine have the parametric form

$$P(u, v) = (1 - v) P_0(u) + v P_1(u) \quad (6.35)$$

The functions $P_0(u)$ and $P_1(u)$ define curves lying in 3D space. Each is described by three component functions, as in $P_0(u) = (X_0(u), Y_0(u), Z_0(u))$. Both $P_0(u)$ and $P_1(u)$ are defined on the same interval in u (commonly from 0 to 1). The ruled surface consists of one straight line joining each pair of corresponding points, $P_0(u')$ and $P_1(u')$, for each u' in $(0, 1)$, as indicated in Figure 6.60. At $v = 0$ the surface is “at” $P_0(u')$, and at $v = 1$ it is at $P_1(u')$. The straight line at $u = u'$ is often called the **ruling** at u' .

1st Ed. Figure 9.3

Figure 6.60. A ruled surface as a family of straight lines.

For a particular fixed value, v' the **v' -contour** is some blend of the two curves $P_0(u)$ and $P_1(u)$. It is an affine combination of them, with the first weighted by $(1 - v')$ and the second by v' . When v' is close to 0, the shape of the v' -contour is determined mainly by $P_0(u)$, whereas when v' is close to 1, the curve $P_1(u)$ has the most influence.

If we restrict v to lie between 0 and 1, only the line segment between corresponding points on the curves will be part of the surface. On the other hand, if v is not restricted, each line will continue forever in both directions, and the surface will resemble an unbounded curved “sheet.” A **ruled patch** is formed by restricting the range of both u and v , to values between, say, 0 and 1.

A ruled surface is easily polygonalized in the usual fashion: choose a set of samples u_i and v_j , and compute the position $P(u_i, v_j)$ and normal $\mathbf{n}(u_i, v_j)$ at each. Then build the lists as we have done before.

Some special cases of ruled surfaces will reveal their nature as well as their versatility. We discuss three important families of ruled surfaces, the **cone**, the **cylinder**, and the **bilinear patch**.

Cones.

A cone is a ruled surface for which one of the curves, say, $P_0(u)$, is a *single* point $P_0(u) = P_0$, the “apex” of the cone, as suggested in Figure 6.61a. In Equation 6.35 this restriction produces:

Figure 6.61. A cone.

$$P(u, v) = (1 - v) P_0 + v P_1(u) \quad \{ \text{a general cone} \} \quad (6.36)$$

For this parameterization all lines pass through P_0 at $v = 0$, and through $P_1(u)$ at $v = 1$. Certain special cases are familiar. A circular cone results when $P_1(u)$ is a circle, and a right circular cone results when the circle lies in a plane that is perpendicular to the line joining the circle's center to P_0 . The specific example shown in Figure 6.61b uses $P_1(u) = (r(u) \cos u, r(u) \sin u, 1)$ where the “radius” curve $r(u)$ varies sinusoidally: $r(u) = 0.5 + 0.2 \cos(5u)$.

Cylinders.

A cylinder is a ruled surface for which $P_1(u)$ is simply a translated version of $P_0(u)$: $P_1(u) = P_0(u) + \mathbf{d}$, for some vector \mathbf{d} , as shown in Figure 6.62a. Sometimes one speaks of “sweeping” the line with endpoints $P_0(0)$ and $P_0(0) + \mathbf{d}$ (often called the “generator”) along the curve $P_0(u)$ (often called the “directrix”), without altering the direction of the line.

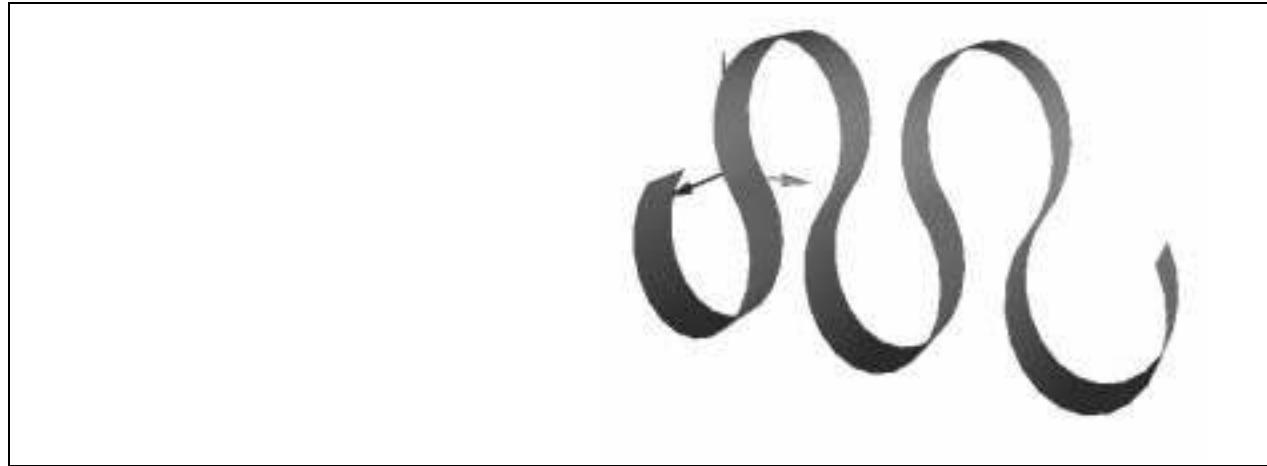


Figure 6.62*. a). A cylinder. b). ribbon candy cylinder.

The general cylinder therefore has the parametric form

$$P(u, v) = P_0(u) + \mathbf{d}v \quad (6.37)$$

To be a true cylinder, the curve $P_0(u)$ is confined to lie in a plane. If $P_0(u)$ is a circle the cylinder is a **circular cylinder**. The direction \mathbf{d} need not be perpendicular to this plane, but if it is, the surface is called a **right cylinder**. This is the case for the generic cylinder. Figure 6.55b shows a “ribbon candy cylinder” where $P_0(u)$ undulates back and forth like a piece of ribbon. The ribbon shape is explored in the exercises.

Bilinear Patches.

A bilinear patch is formed when both $P_0(u)$ and $P_1(u)$ are straight line segments defined over the same interval in u , say, 0 to 1. Suppose the endpoints of $P_0(u)$ are P_{00} and P_{01} , (so that $P_0(u)$ is given by $(1 - u)P_{00} + u P_{01}$), and the endpoints of $P_1(u)$ are P_{10} and P_{11} . Then using Equation 6.35 the patch is given parametrically by:

$$P(u, v) = (1 - v)(1 - u)P_{00} + (1 - v)u P_{01} + v(1 - u)P_{10} + uv P_{11} \quad (6.38)$$

This surface is called “bilinear” because its dependence is linear in u and linear in v . Bilinear patches need not be planar; in fact they are planar only if the lines $P_0(u)$ and $P_1(u)$ lie in the same plane (see the exercises). Otherwise there must be a “twist” in the surface as we move from one of the defining lines to the other.

An example of a nonplanar bilinear patch is shown in Figure 6.63. $P_0(u)$ is the line from $(2, -2, 2)$ to $(2, 2, -2)$, and $P_1(u)$ is the line from $(-2, -2, -2)$ to $(-2, 2, 2)$. These lines are not coplanar. Several u -contours are shown, and the twist in the patch is clearly visible.

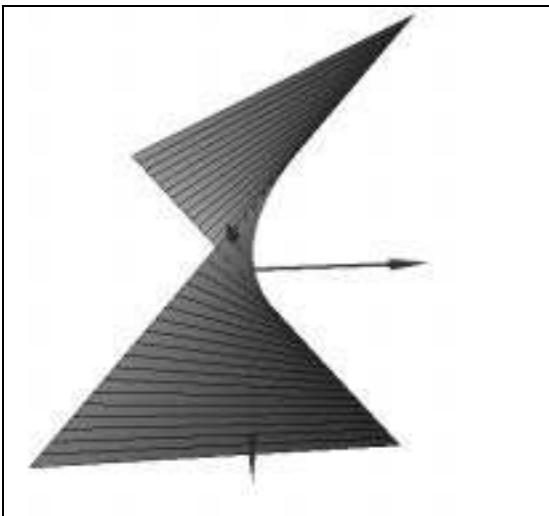


Figure 6.63. A bilinear patch.

The normal vector to a bilinear patch is easily found using Equation 6.26. If the patch is planar the direction of the normal is constant but its magnitude can vary with u and v . If the patch is nonplanar both the magnitude and direction of the normal vector vary with position.

Other ruled surfaces.

There are many other interesting ruled surfaces. Figure 6.64a shows a double helix formed when $P_0(u)$ and $P_1(u)$ are both helices that wind around each other. Part b shows the intriguing Möbius strip that has only one edge. The exercises explore the parametric representation of these surfaces. Part c shows a vaulted roof made up of four ruled surfaces. Case Study 6.8 examines modeling such vaulted domes of a cathedral.

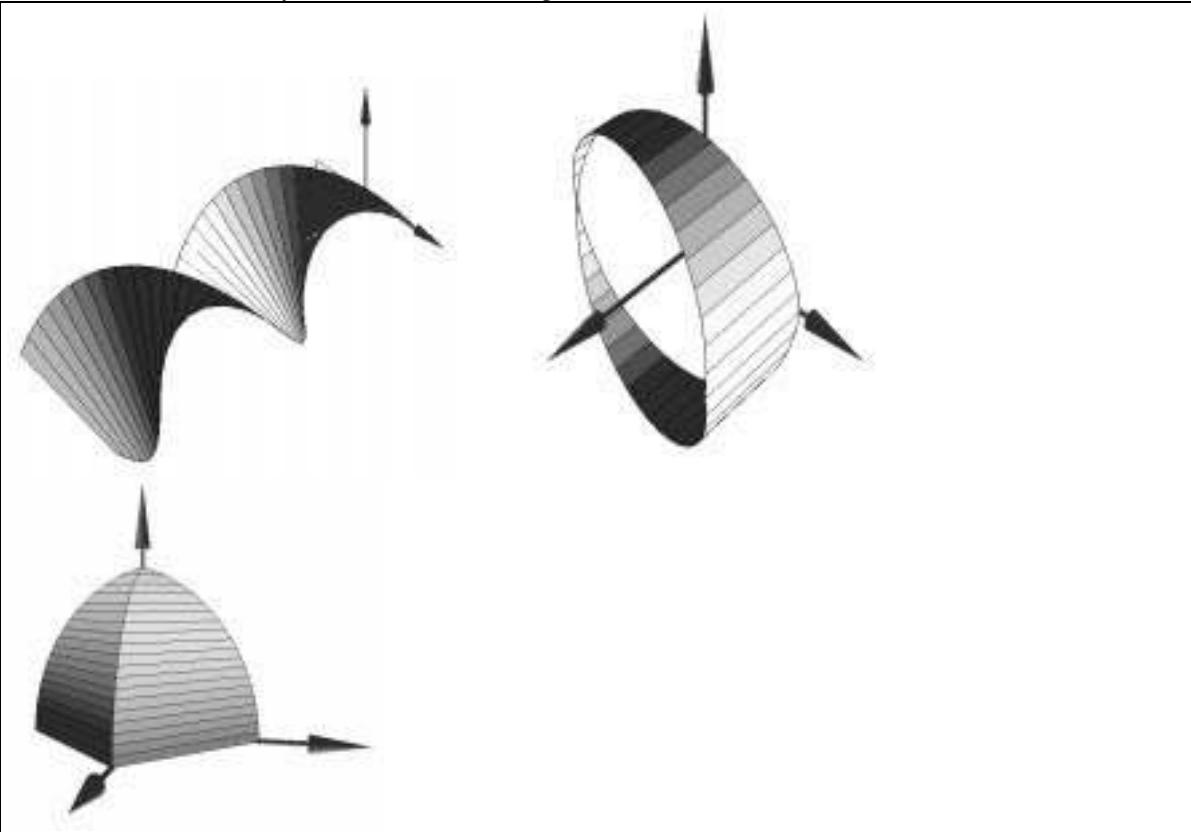


Figure 6.64. a). double helix b). Möbius strip, c).Vaulted roof.

Bilinearly blended Surfaces - Coons Patches.

An interesting and useful generalization of a ruled surface, which interpolates two boundary curves $P_0(u)$ and $P_1(u)$, is a bilinearly blended patch that interpolates to *four* boundary curves. This family was first developed by Steven Coons [coons...] and is sometimes called a Coons patch.

Figure 6.65 shows four adjoining boundary curves, named $p_{u0}(u)$, $p_{u1}(u)$, $p_{0v}(v)$, and $p_{1v}(v)$. These curves meet at the patch corners (where u and v are combinations of 0 and 1) but otherwise have arbitrary shapes. This therefore generalizes the bilinear patch for which the boundary curves are straight lines.

Figure 6.65. Four boundary curves determining a Coons patch.

We want a formula $P(u, v)$ that produces a smooth transition from each boundary curve to the other as u and v vary.

A natural first guess is to somehow combine a ruled patch built out of $p_{u0}(u)$ and $p_{u1}(u)$, with a ruled patch built out of $p_{0v}(v)$ and $p_{1v}(v)$. But simply adding such surfaces doesn't work: it fails to interpolate the four curves properly (and it's illegal: it results in a non-affine combination of points!). The trick is to add these surfaces and then subtract the bilinear patch formed from the four corners of these curves. Figure 6.66 shows visually how this works [heckbert94].

$$\text{surf 1} + \text{surf 2} - \text{surf 3} = \text{coons patch}$$

Figure 6.66. Combining patches to create the Coons patch.

The formula for the patch is therefore

$$\begin{aligned} P(u, v) &= [p_{0v}(v)(1-u) + p_{1v}(v)u] + [p_{u0}(u)(1-v) + p_{u1}(u)v] \\ &\quad - [(1-u)(1-v)p_{0v}(0) + u(1-v)p_{1v}(0)u + p_{0v}(1)v(1-u) + p_{1v}(1)uv] \end{aligned} \tag{6.39}$$

Note that at each (u, v) this is still an affine combination of points, as we insist. Check that at $(u, v)=(0, 0)$ this evaluates to $p_{u0}(0)$, and similarly that it coincides with the other three corners at the other extreme values of u and v . Figure 6.67 shows an example Coons patch bounded by curves that have a sinusoidal oscillation.

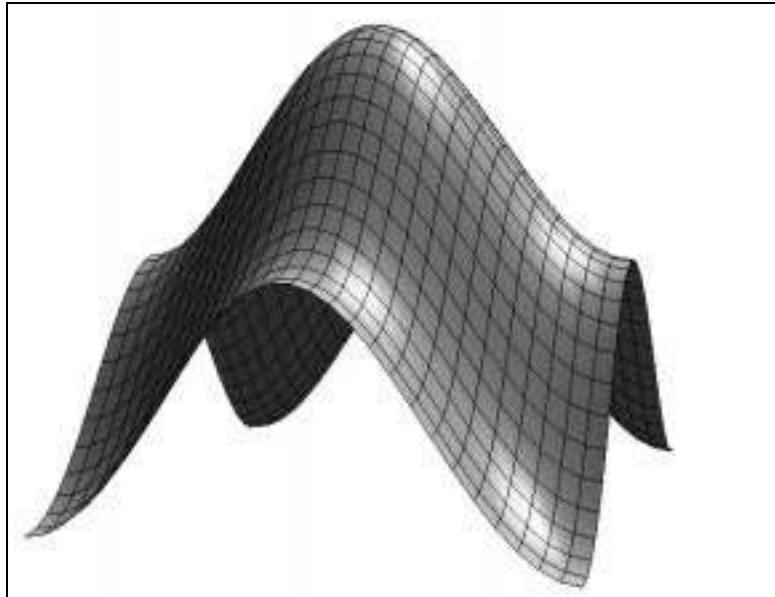


Figure 6.67. Example of a Coons patch.

Practice Exercises.

6.5.11. A Pyramid is a cone. What shape should $P_1(u)$ have to create a ruled surface that is a pyramid with a square base? Give specific expressions for the curve and the point P_0 so that the square base of the pyramid lies in the x, z -plane, centered at the origin, with sides of length 2. The pyramid should have height 1.5.

6.5.12. Ribbon candy cylinders. Find a parametric form for $P_0(u)$ that produces a good approximation to the ribbon candy cylinder shown in Figure 6.62b. Consider the ribbon as wrapped about a succession of abutting circular cylinders of radius 1. The center of the i -th cylinder lies at $(x, y_i) = (id, \pm r)$, where the + and - alternate, and $d^2 = 1 - r^2$. Choose some value of r between 0 and 1.

6.5.13. The double helix. The parametric form for a helix is $(\cos(t), \sin(t), t)$. Find expressions for two helices, $P_0(u)$ and $P_1(u)$, both of which wind around the z -axis yet are 180° out of phase so that they wind around each other. Write the parametric form for the ruled surface formed using these two curves.

6.5.14. The Möbius strip. Find a parametric form for the Möbius strip shown in Figure 6.64b.

Hint: revolve a line about the z -axis, but put in a twist as it goes around. Does the following attempt do the job: $P_0(u) = (\cos(2pu)), \sin(2pu), u)$, and $P_1(u) = (\cos(2pu)), \sin(2pu), 1-u)$?

6.5.15. Is it affine? Show that the Coons patch $P(u,v)$ of Equation 6.33 is composed of an affine combination of points.

6.5.16. Does it really interpolate? Check that $P(u, v)$ of Equation 6.33 interpolates each of the four boundary curves, and therefore interpolates each of the four corners.

6.5.7. Surfaces of Revolution.

As described earlier, a surface of revolution is formed by a **rotational sweep** of a profile curve, C , around an axis. Suppose we place the profile in the xz -plane and represent it parametrically by $C(v) = (X(v), Z(v))$. To generate the surface of revolution we sweep the profile about the z -axis under control of the u parameter, with u specifying the angle through which each point has been swept about the axis. As before, the different positions of the curve C around the axis are called **meridians**. When the point $(X(v), 0, Z(v))$ is rotated by u radians, it becomes $(X(v)\cos(u), X(v)\sin(u), Z(v))$. Sweeping it completely around generates a full circle, so contours of constant v are circles, called **parallels** of the surface⁷. The parallel at v has radius $X(v)$ and lies at height $Z(v)$ above the xy -plane. Thus the general point on the surface is

⁷ More formally, a **meridian** is the intersection of the surface with a plane that contains the axis of revolution., and a **parallel** is the intersection of the surface with a plane perpendicular to the axis.

$$P(u, v) = (X(v) \cos(u), X(v)\sin(u), Z(v)) \quad (6.40)$$

The generic sphere, tapered cylinder, and cone are all familiar special cases. (What are their profiles?)

The normal vector to a surface of revolution is easily found by direct application of Equation 6.34 to Equation 6.19 (see the exercises). This yields

$$\mathbf{n}(u, v) = X(v)(\dot{Z}(v) \cos(u), \dot{Z}(v) \sin(u), -\dot{X}(v)) \quad (6.41)$$

where the dot denotes the first derivative of the function. The scaling factor $X(v)$ disappears upon normalization of the vector. This result specializes to the forms we found above for the simple generic shapes (see the exercises).

For example, the **torus** is generated by sweeping a displaced circle about the z -axis, as shown in Figure 6.68. The circle has radius A and is displaced along the x -axis by D , so that its profile is $C(v) = (D + A \cos(v), A \sin(v))$. Therefore the torus (Figure 6.68b) has representation

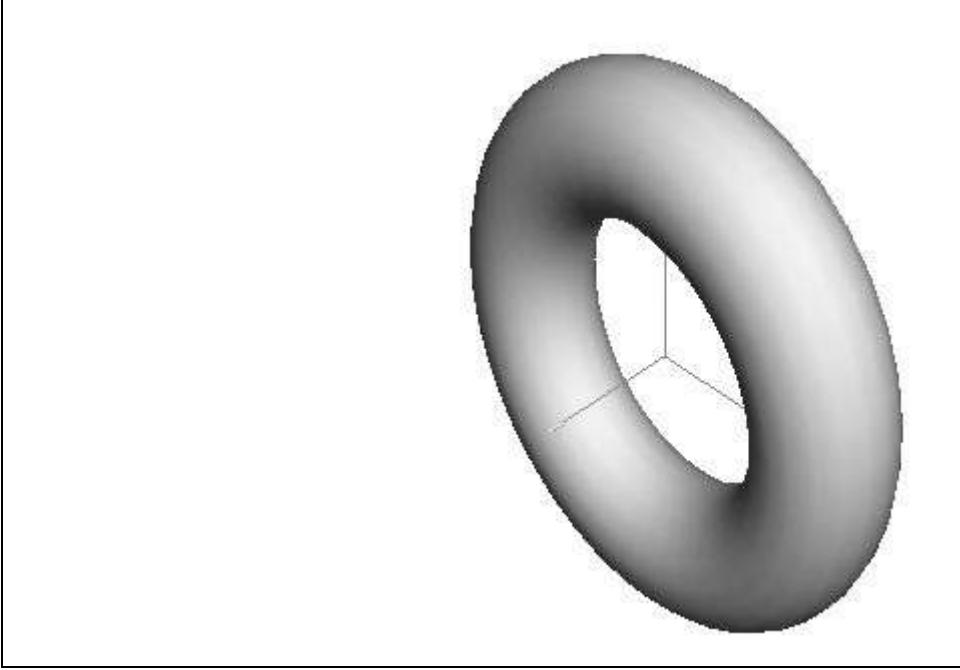


Figure 6.68. A torus.

$$P(u, v) = ((D + A \cos(v)) \cos(u), (D + A \cos(v)) \sin(u), A \sin(v)) \quad (6.42)$$

Its normal vector is developed in the exercises.

We usually sweep a curve lying in a plane about an axis that lies in that plane, but a surface of revolution can be formed by sweeping about any axis. Choosing different axes for a given profile can lead to interesting families of surfaces. The general form for $P(u, v)$ for the surface of revolution about an arbitrary axis is developed in the exercises.

A mesh for a surface of revolution is built in a program in the usual way (see Section 6.5.4). We choose a set of u and v values, $\{u_i\}$ and $\{v_j\}$, and compute a vertex at each from $P(u_i, v_j)$, and a normal direction from $\mathbf{n}(u_i, v_j)$. Polygonal faces are built by joining four adjacent vertices with straight lines. A method to do this is discussed in Case Study 6.13.

Figure 6.69 shows another example in which we try to model the dome of the exquisite Taj Mahal in Agra, India, shown in part a. Part b shows the profile curve in the xz -plane, and part c shows the resulting surface of revolution. Here we describe the profile by a collection of data points $C_i = (X_i, Z_i)$, since no suitable parametric formula is available. (We rectify this lack in Chapter 8 by using a B-spline curve to form a smooth parametric curve based on a set of data points.)

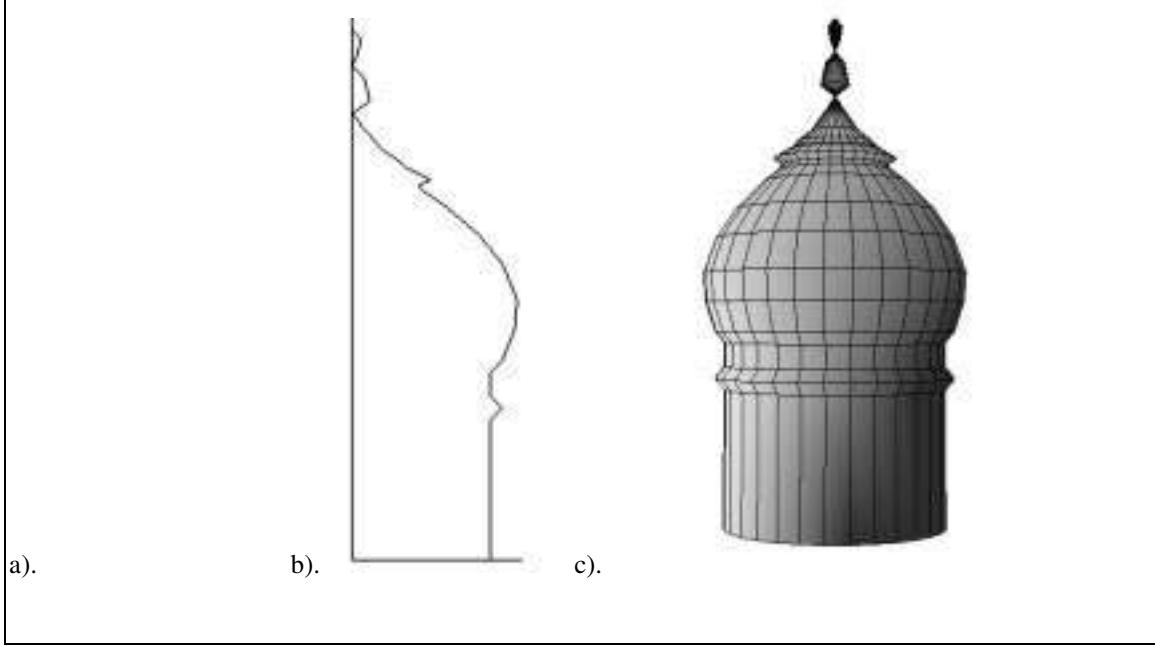


Figure 6.69*. A surface of revolution - the dome of the Taj Mahal.

To build a surface of revolution when the profile consists of discrete points, simply take as the ij -th vertex the slightly different form of Equation 6.41:

$$P_{ij} = (X_j \cos(u_i), X_j \sin(u_i), Z_j)$$

Practice Exercises.

6.5.17. The generic shapes as surfaces of revolution. Describe the profiles of the generic sphere, cylinder, and cone parametrically, and then express them as surfaces of revolution.

6.5.18. Rotation About Other Axes. Consider a profile curve $C(v) = (X(v), Z(v))$, lying in the xz -plane, and an arbitrary axis through the origin given by unit vector \mathbf{r} . We know from Equation 5.33 that the matrix $R_r(\theta)$ performs a rotation of a point through θ radians about the axis \mathbf{r} .

a). From this show that the surface of revolution formed by sweeping $C(v)$ about axis \mathbf{r} is

$$(X(u, v), Y(u, v), Z(u, v), 1) = R_r(u) \begin{pmatrix} X(v) \\ 0 \\ Z(v) \\ 1 \end{pmatrix}$$

b). Check this for the special case of rotation about the z -axis.

c). Repeat part b for rotations about the x -axis, and about the y -axis.

6.5.19. Finding normal vectors. a). Apply Equation 6.40 to Equation 6.25 to derive the form in Equation 6.41 for the normal vector to a surface of revolution. b). Use this result to find the normal to each of

the generic sphere, cylinder, and cone, and show the results agree with those found in Section 6.5.3.

Show that the normal vector to the torus has the form

$\mathbf{n}(u, v) = (\cos(v)\cos(u), \cos(v)\sin(u), \sin(v))(D + A \cos(v))$. Also, find the inside-outside function for the torus, and compute the normal using its gradient.

6.5.20. An Elliptical Torus. Find the parametric representation for the following two surfaces of revolution: a). The ellipse given by $(a \cos(v), b \sin(v))$ is first displaced R units along the x -axis and then revolved about the y -axis. b). The same ellipse is revolved about the x -axis.

6.5.21. A “Lissajous of Revolution.” Sketch what the surface would look like if the Lissajous figure of Equation 6.19 with $M = 2$, $N = 3$, and $\phi = 0$ is rotated about the y -axis.

6.5.22. Revolved n-gons. Sketch the surface generated when a square having vertices $(1, 0, 0)$, $(0, 1, 0)$, $(-1, 0, 0)$, $(0, -1, 0)$ is revolved about the y -axis. Repeat for a pentagon and a hexagon.

6.5.8. The Quadric Surfaces.

An important family of surfaces, the quadric surfaces, are the 3D analogs of conic sections (the ellipse, parabola, and hyperbola, which we examined in Chapter 3. Some of the quadric surfaces have beautiful shapes and can be put to good use in graphics.

The six quadric surfaces are illustrated in Figure 6.70.

1st Ed. Figure 9.12. the 6 quadric surfaces a),...,f).

Figure 6.70. The six quadric surfaces: a. Ellipsoid, b. Hyperboloid of one sheet, c. Hyperboloid of two sheets, d. Elliptic cone, e. Elliptic paraboloid, f. Hyperbolic paraboloid.

We need only characterize the “generic” versions of these shapes, since we can obtain all the variations of interest by scaling, rotating, and translating the generic shapes. For example, the ellipsoid is usually said to have the inside–outside function

$$F(x, y, z) = \left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 + \left(\frac{z}{c}\right)^2 - 1 \quad (6.43)$$

so that it extends in x from $-a$ to a , in y from $-b$ to b , and in z from $-c$ to c . This shape may be obtained from the form of the generic sphere by scaling it in x , y , and z by a , b , and c , respectively, as we describe below. We can obtain rotated versions of the ellipsoid in a similar manner.

Figure 6.71 provides descriptions of the six generic quadric surfaces, giving both their implicit and parametric forms. We discuss some interesting properties of each shape later.

name of quadric	Inside-outside function	parametric form	v-range,u-range
ellipsoid	$x^2 + y^2 + z^2 - 1$	$(\cos(v)\cos(u), \cos(v)\sin(u), \sin(v))$	$(-\pi/2, \pi/2), (-\pi, \pi)$
hyperboloid of one sheet	$x^2 + y^2 - z^2 - 1$	$(\sec(v)\cos(u), \sec(v)\sin(u), \tan(v))$	$(-\pi/2, \pi/2), (-\pi, \pi)$
hyperboloid of two sheets	$x^2 - y^2 - z^2 - 1$	$(\sec(v)\cos(u), \sec(v)\tan(u), \tan(v))$	$(-\pi/2, \pi/2)$ ⁸
elliptic cone	$x^2 + y^2 - z^2$	$(v\cos(u), v\sin(u), v)$	any real, $(-\pi, \pi)$

⁸ The v-range for sheet #1 is $(-\pi/2, \pi/2)$, and for sheet #2 is $(\pi/2, 3\pi/2)$.

elliptic paraboloid	$x^2 + y^2 - z$	$(v \cos(u), v \sin(u), v^2)$	$v \geq 0, (-\pi, \pi)$
hyperbolic paraboloid	$-x^2 + y^2 - z$	$(v \tan(u), v \sec(u), v^2)$	$v \geq 0, (-\pi, \pi)$

Figure 6.71. Characterization of the six “generic” quadric surfaces.

Figure 6.64 also reports the parametric form for each of the generic quadric surfaces. It is straightforward to check that each of these forms is consistent with its corresponding implicit form: substitute the parametric form for the x -, y -, and z - components, and use trigonometric identities to obtain 0.

Note that a change of sign in one term of the inside–outside function causes a $\cos()$ to be changed to $\sec()$ and a $\sin()$ to be changed to $\tan()$ in the parametric forms. Both $\sec()$ and $\tan()$ grow without bound as their arguments approach $\pi/2$, and so when drawing u -contours or v -contours the relevant parameter is restricted to a smaller range.

Some Notes on the Quadric Surfaces.

We shall summarize briefly some significant properties of each quadric surface. One such property of a quadric is the nature of its traces. A **trace** is the curve formed when the surface is “cut” by a plane. All traces of a quadric surface are conic sections - see the exercises. The **principal traces** are the curves generated when the cutting planes are aligned with the axes: the planes $z = k$, $y = k$, or $x = k$, where k is some constant.

In the discussion that follows we suppose that the generic surfaces have been scaled in x -, y -, and z - by values a , b , and c , respectively, to make it easier to talk about the dimensions of the surfaces, and to distinguish cases where the surface is a surface of revolution or not.

- **Ellipsoid.** Compare the implicit form and parametric form for the ellipse in Chapter 3 to see how they extend the 2D ellipse to this 3D ellipsoid. Parameters a , b , and c give the extent of the ellipsoid along each axis. When two of the parameters are equal, the ellipsoid is a surface of revolution. (If $a = b$, what is the axis of revolution?). When a , b , and c all are equal, the ellipsoid becomes a sphere. All traces of the ellipsoid are ellipses.

- **Hyperboloid of one sheet.** When $a = b$, the hyperboloid becomes a surface of revolution formed by rotating a hyperbola about an axis. The principal traces for the planes $z = k$ are ellipses, and those for the planes $x = k$ and $y = k$ are hyperbolas. The hyperboloid of one sheet is particularly interesting because it is a *ruled* surface, as suggested in Figure 6.72a. If a thread is woven between two parallel ellipses, as shown, this surface will be created. Formulas for the rulings are discussed in the exercises.

9.13 a). hyperboloid of one sheet	b). hyperbolic paraboloid.
-----------------------------------	----------------------------

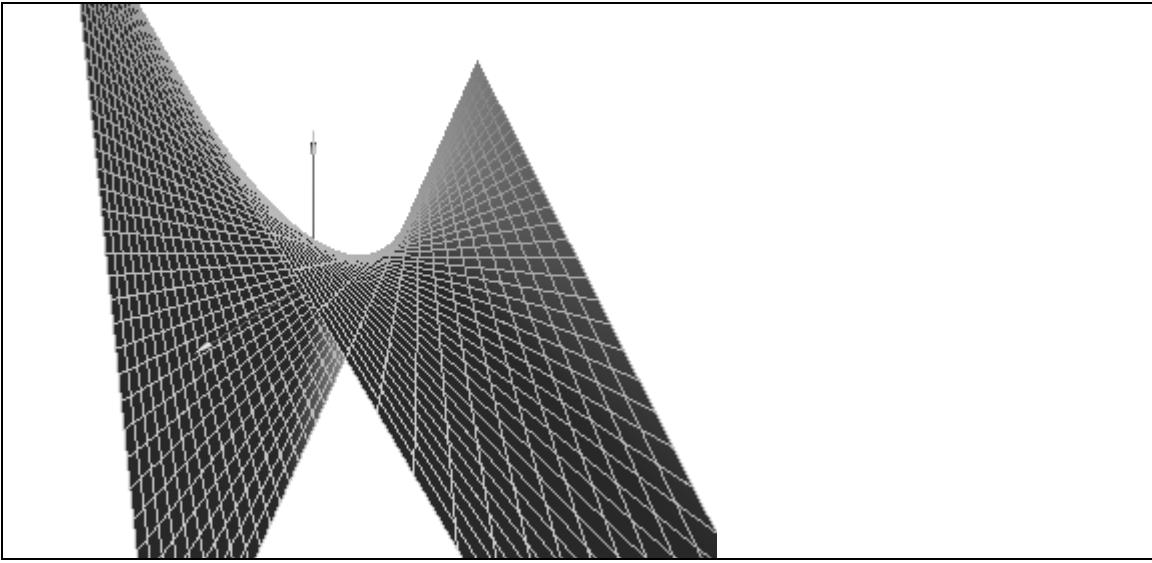


Figure 6.72. Two ruled quadric surfaces.

- **Hyperboloid of two sheets.** No part of the surface lies between $x = -a$ and $x = a$. (Why?). When $a = b$, it becomes a surface of revolution. The traces for planes $x = k$ when $|k| > a$ are ellipses, and the other principal traces are hyperbolas.

- **Elliptic cone.** The elliptic cone is a special case of the general cone treated earlier: Its generator lines trace an ellipse. This cone is, of course, a ruled surface, and the principal traces for planes $z = k$ are ellipses. What are traces for planes that contain the z -axis? When $a = b$, this quadric is a surface of revolution: It becomes a right circular cone.

- **Elliptic paraboloid.** The traces of an elliptic paraboloid for planes $z = k > 0$ are ellipses, and the other principal traces are parabolas. When $a = b$, it is a surface of revolution.

- **Hyperbolic paraboloid.** The hyperbolic paraboloid is sometimes called a “saddle-shaped” surface. The traces for planes $z = k$ (when $k \neq 0$) are hyperbolas, and for planes $x = k$ or $y = k$ they are parabolas. (What is the intersection of this surface with the plane $z = 0$?) This is also a ruled surface (see Figure 6.65b).

Normal Vectors to Quadric Surfaces.

Because the implicit form for each quadric surface is quadratic in x , y , and z , taking the gradient to find the normal presents no problem. Further, since each component of the gradient vector is linear in its own variable or just a constant, it is straightforward to write the gradient in parametric form: just substitute $X(u, v)$ for x , etc.

For instance, the gradient of $F(x, y, z)$ for the ellipsoid is

$$\nabla F = (2x, 2y, 2z)$$

and so the normal in parametric form is (after deleting the 2)

$$n(u, v) = (\cos(v) \cos(u), \cos(v) \sin(u), \sin(v)) \quad (6.44)$$

Normals for the other quadric surfaces follow just as easily, and so they need not be tabulated.

| Practice exercises.

6.5.23. The Hyperboloid is a Ruled Surface. Show that the implicit form of a hyperboloid of one sheet can be written $(x+z)(x-z) = (1-y)(1+y)$. Show that therefore two families of straight lines lie in the surface: the family $x-z = A(1-y)$ and the family $A(x+z) = 1+y$, where A is a constant. Sketch these families for various values of A . Examine similar rulings in the hyperbolic paraboloid.

6.5.24. The hyperboloid of one sheet. Show that an alternative parametric form for the hyperboloid of one sheet is $p(u,v) = (\cosh(v) \cos(u), \cosh(v) \sin(u), \sinh(v))$.

6.5.25. Traces of Quadrics are Conics. Consider any three (noncollinear) points lying on a quadric surface. They determine a plane that cuts through the quadric, forming the trace curve. Show that this curve is always a parabola, ellipse, or hyperbola.

6.5.26. Finding Normals to the Quadrics. Find the normal vector in parametric form for each of the six quadric surfaces.

6.5.27. The Hyperboloid As a Ruled Surface. Suppose $(x_0, y_0, 0)$ is a point on the hyperboloid of one sheet. Show that the vector $R(t) = (x_0 + y_0 t, y_0 - x_0 t, t)$

describes a straight line that lies everywhere on the hyperboloid and passes through $(x_0, y_0, 0)$. Is this sufficient to make the surface a ruled surface? Why or why not? [apostol p.329]

6.5.28. The Hyperbolic Paraboloid As a Ruled Surface. Show that the intersection of any plane parallel to the line $y = \pm x$ cuts the hyperbolic paraboloid along a straight line.

6.5.9. The Superquadrics.

Following the work of Alan Barr [barr81], we can extend the quadric surfaces to a vastly larger family, in much the way we extended the ellipse to the superellipse in Chapter 3. This provides additional interesting surface shapes we can use as models in applications.

Barr defines four superquadric solids: the superellipsoid, superhyperboloid of one sheet, and superhyperboloid of two sheets, which together extend the first three quadric surfaces, and the supertoroid which extends the torus. The extensions introduce two “bulge factors,” m and n , to which various terms are raised. These bulge factors affect the surfaces much as n does for the superellipse. When both factors equal 2, the first three superquadrics revert to the quadric surfaces catalogued previously. Example shapes for the four superquadrics are shown in Figure 6.73.

1st Ed. Figure 9.15

Figure 6.73. Examples of the four superquadrics. n_1 and n_2 are (left to right): 10, 2, 1.11, .77, and .514. (Courtesy of Jay Greco.)

The inside-outside functions and parametric forms⁹ are given in Figure 6.74.

name of quadric	Implicit form	parametric form	v-range, u-range
superellipsoid	$(x^n + y^n)^{m/n} + z^m - 1$	$(\cos^{2/m}(v) \cos^{2/n}(u), \cos^{2/m}(v) \sin^{2/n}(u), \sin^{2/m}(v))$	$[-\pi/2, \pi/2], [-\pi, \pi]$
superhyperboloid of one sheet	$(x^n + y^n)^{m/n} - z^m - 1$	$(\sec^{2/m}(v) \cos^{2/n}(u), \sec^{2/m}(v) \sin^{2/n}(u), \tan^{2/m}(v))$	$(-\pi/2, \pi/2), [-\pi, \pi]$
superhyperboloid of two sheets	$(x^n - y^n)^{m/n} - z^m - 1$	$(\sec^{2/m}(v) \sec^{2/n}(u), \sec^{2/m}(v) \tan^{2/n}(u), \tan^{2/m}(v))$	$(-\pi/2, \pi/2)$
supertoroid	$((x^n - y^n)^{1/n} - D)^m + z^m - 1$	$((d + \cos^{2/m}(v)) \cos^{2/n}(u), (d + \cos^{2/m}(v)) \sin^{2/n}(u), \sin^{2/m}(v))$	$[-\pi, \pi], [-\pi, \pi]$

Figure 6.74. Characterization of the four superquadric surfaces.

⁹ Keep in mind that it's illegal to raise a negative value to a fractional exponent. So expressions such as $\cos^{2/m}(v)$ should be evaluated as $\cos(v)|\cos(v)|^{2/m-1}$ or the equivalent.

These are “generic” superquadrics in the sense that they are centered at the origin, aligned with the coordinate axes, and have unit dimensions. Like other shapes they may be scaled, rotated, and translated as desired using the current transformation to prepare them properly for use in a scene.

The normal vector $\mathbf{n}(u, v)$ can be computed for each superquadric in the usual ways. We shall give only the results here.

Normals to the superellipsoid and supertoroid.

The normal vectors for the superellipsoid and the supertoroid are the same:

$$\mathbf{n}(u, v) = (\cos^{2-2/m}(v) \cos^{2-2/n}(u), \cos^{2-2/m}(v) \sin^{2-2/n}(u), \sin^{2-2/m}(v)) \quad (6.45)$$

(How can they be the same? The two surfaces surely don’t have the same shape.)

The Superhyperboloid of one sheet.

The normal vector to the superhyperboloid of one sheet is the same as that of the superellipsoid, except all occurrences of $\cos(v)$ are replaced with $\sec(v)$ and those of $\sin(v)$ are replaced with $\tan(v)$. Do not alter $\cos(u)$ or $\sin(u)$ or any other term.

The Superhyperboloid of two sheets.

For the superhyperboloid of two sheets, the trigonometric functions in both u and v are replaced. Replace all occurrences of $\cos(v)$ with $\sec(v)$, those of $\sin(v)$ with $\tan(v)$, those of $\cos(u)$ with $\sec(u)$, and those of $\sin(u)$ with $\tan(u)$.

Practice Exercises.

6.5.29. Extents of Superquadrics. What are the maximum x , y , and z values attainable for the superellipsoid and the supertoroid?

6.5.30. Surfaces of revolution. Determine the values of bulge factors m and n for which each of the superquadrics is a surface of revolution, and find the axis of revolution. Describe any other symmetries in the surfaces.

6.5.31. Deriving the Normal Vectors. Derive the formula for the normal vector to each superquadric. surface.

6.5.10. Tubes Based on 3D Curves.

In Section 6.4.4 we studied tubes that were based on a “spine” curve $C(t)$ meandering through 3D space. A polygon was stationed at each of a selection of spine points, and oriented according to the Frenet frame computed there. Then corresponding points on adjacent polygons were connected to form a flat-faced tube along the spine.

Here we do the same thing, except we compute the normal to the surface at each vertex, so that smooth shading can be performed. Figure 6.75 shows the example of a tube wrapped around a helical shape. Compare this with Figure 6.47.

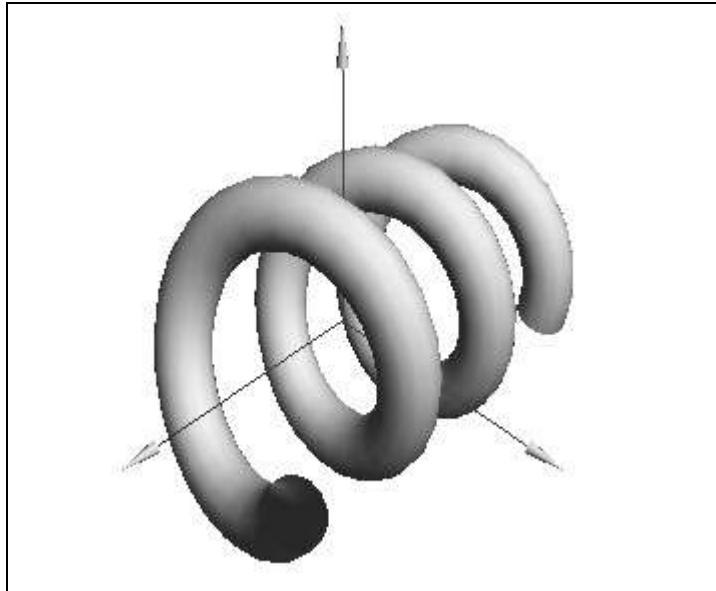


Figure 6.75. A helical tube undulating through space.

If we wish to wrap a circle $(\cos(u), \sin(u), 0)$ about the spine $C(t)$, the resulting surface has parametric representation

$$P(u, v) = C(v) + \cos(u)\mathbf{N}(v) + \sin(u)\mathbf{B}(v) \quad (6.46)$$

where the normal vector $\mathbf{N}(t)$ and binormal vector $\mathbf{B}(t)$ are those given in Equations 6.14 and 6.15. Now we can build a mesh for this tube in the usual way, by taking samples of $P(u, v)$, building the vertex, normal, and face lists, etc. (What would be altered if we wrapped a cycloid - recall Figure 3.77 - about the spine instead of a circle?)

6.5.11. Surfaces based on Explicit Functions of Two Variables.

Many surface shapes are **single-valued** in one dimension, so their position can be represented as an explicit function of two of the independent variables. For instance, there may be a single value of “height” of the surface above the xz -plane for each point (x, z) , as suggested in Figure 6.76. We can then say that the height of the surface at (x, z) is some $f(x, z)$. Such a function is sometimes called a **height field** [bloomenthal97]. A height field is often given by a formula such as

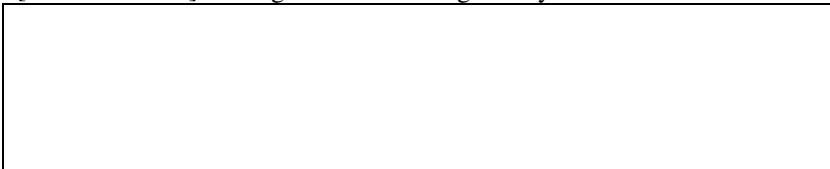


Figure 6.76. A single-valued height field above the xz -plane.

$$f(x, z) = e^{-ax^2 - bz^2} \quad (6.47)$$

(where a and b are given constants), or the circularly symmetric “sinc” function

$$f(x, z) = \frac{\sin(\sqrt{x^2 + z^2})}{\sqrt{x^2 + z^2}} \quad (6.48)$$

Contrast this with surfaces such as the sphere, for which more than one value of y is associated with each point (x, z) . Single valued functions permit a simple parametric form:

$$P(u, v) = (u, f(u, v), v) \quad (6.49)$$

and their normal vector is $\mathbf{n}(u, v) = (-\partial f / \partial u, 1, -\partial f / \partial v)$ (check this). That is, u and v can be used directly as the dependent variables for the function. Thus u -contours lie in planes of constant x , and v -contours lie in planes of constant z . Figure 6.77a shows a view of the example in Equation 6.47, and Figure 6.77b shows the function of Equation 6.48.

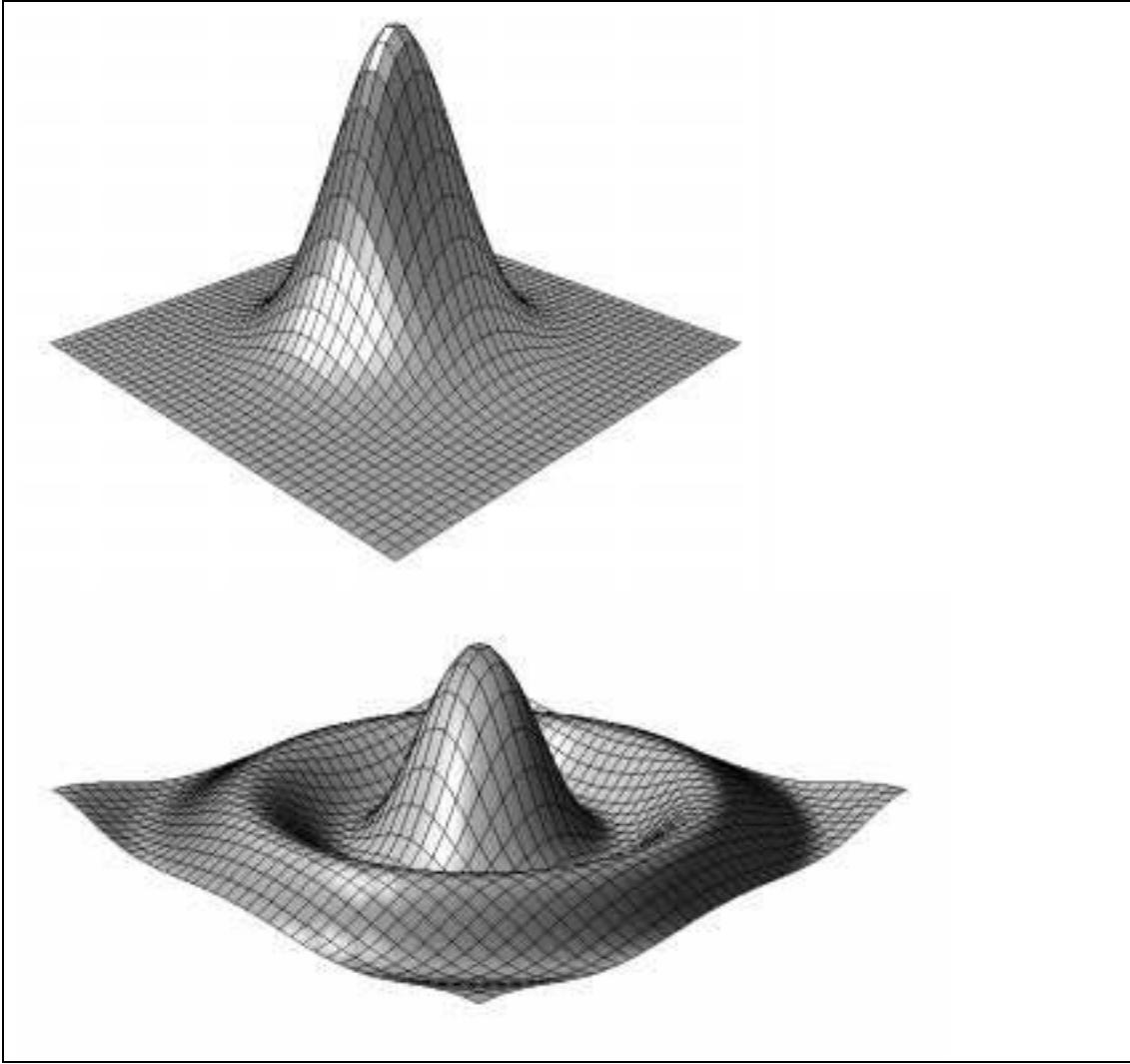


Figure 6.77. Two height fields. a) gaussian, b). *sinc* function. (files: fig6.77a.bmp, fig6.77b.bmp)

Each line is a trace of the surface cut by a plane, $x = k$ or $z = k$, for some value of k . Plots such as these can help illustrate the behavior of a mathematical function.

Practice exercise 6.5.32. The Quadrics as Explicit Functions. The elliptic paraboloid can be written as $z = f(x, y)$, so it has an alternate parametric form $(u, v, f(u, v))$. What is $f()$? In what ways is this alternate parametric form useful? What other quadrics can be represented this way? Can any superquadrics be represented explicitly this way?

6.6. Summary

This chapter is concerned with modeling and drawing a wide variety of surfaces of 3D objects. This involves finding suitable mathematical descriptions for surface shapes, and creating efficient data structures that hold sufficient detail about a surface to facilitate rendering of the surface. We developed the Mesh class, whose data fields include three lists: the vertex, normal vector, and face lists. This data structure can efficiently hold all relevant geometric data about a flat-faced object such as a polyhedron, and it can hold sufficient data to model a polygonal “skin” that approximates other smoothly curved surfaces.

We showed that once a mesh data structure has been built, it is straightforward to render it in an OpenGL environment. It is also easy to store a mesh in a file, and to read it back again into a program.

Modern shading algorithms use the normal vector at each vertex of each face to determine how light or dark the different points within a face should be drawn. If the face should be drawn flat the same normal vector - the normal vector to the face itself - is used for every vertex normal. If the mesh is designed to represent an underlying smoothly curved surface the normal vector at each vertex is set to the normal of the underlying surface at that point, and rendering algorithms use a form of interpolation to produce gracefully varying shades in the picture (as we discuss in Chapter 11). Thus the choice of what normal vectors to store in a mesh depends on how the designer wishes the object to appear.

A wide variety of polyhedral shapes that occur in popular applications were examined, and techniques were developed that build meshes for several polyhedral families. Here special care was taken to use the normal to the face in each of the face’s vertex normals. We also studied large families of smoothly varying objects, including the classical quadric surfaces, cylinders, and cones, and discussed how to compute the direction of the normal vector at each point by suitable derivatives of the parametric form for the surface.

The Case Studies in the next section elaborate on some of these ideas, and should not be skipped. Some of them probe further into theory. A derivation of the Newell method to compute a normal vector is outlined, and you are asked to fill in various details. The family of quadric surfaces is seen to have a unifying matrix form that reveals its underlying structure, and a congenial method for transforming quadrics is described. Other Case Studies ask that you develop methods or applications to create and draw meshes for the more interesting classes of shapes described.

6.7. Case Studies.

6.7.1. Case Study 6.1. Meshes stored in Files.

(Level of Effort: II.) We want the Mesh class to support writing of Mesh objects to a file, and reading files mesh objects back into a program. We choose a simple format for such files. The first line lists the number of vertices, number of normals, and number of faces in the mesh. Then each vertex in the mesh is listed as a triple of floating point values, (x_i, y_i, z_i) . Several vertices are listed on each line. Then each normal vector is listed, also as a triple of floating point numbers. Finally, each face is listed, in the format:

number of vertices in this face
the list of indices in the vertex list for the vertices in this face
the list of indices in the normal list for the vertices in this face

For example, the simple barn of Figure 6.5 would be stored as:

```

10 7 7
0 0 0   1 0 0   1 1 0   1 1.5 0   0 1 0
0 0 1   1 0 1   1 1 1   1 1.5 1   0 1 1
-1 0 0   -0.477 0.8944 0   0.447 0.8944 0
1 0 0   0 -1 0   0 0 1   0 0 -1
4 0 5 9 4   0 0 0 0
4 3 4 9 8   1 1 1 1
4 2 3 8 7   2 2 2 2
4 1 2 7 6   3 3 3 3
4 0 1 6 5   4 4 4 4
5 5 6 7 8 9   5 5 5 5 5
5 0 4 3 2 1   6 6 6 6 6

```

Here the first face is a quadrilateral based on the vertices numbered 0,5,9,4, and the last two faces are pentagons.

To read a mesh into a program from a file you might wish to use the code in Figure 6.78. Given a filename, it opens and reads the file into an existing mesh object, and returns 0 if it can do this successfully. It returns non-zero if an error occurs, such as when the named file cannot be found. (Additional testing should be done within the method to catch formatting errors, such as a floating point number when an integer is expected.)

```
int Mesh:: readmesh(char * fileName)
{
    fstream infile;
    infile.open(fileName, ios::in);
    if(infile.fail()) return -1; // error - can't open file
    if(infile.eof()) return -1; // error - empty file
    infile >> numVerts >> numNorms >> numFaces;
    pt = new Point3[numVerts];
    norm = new Vector3[numNorms];
    face = new Face[numFaces];
    //check that enough memory was found:
    if( !pt || !norm || !face) return -1; // out of memory
    for(int p = 0; p < numVerts; p++) // read the vertices
        infile >> pt[p].x >> pt[p].y >> pt[p].z;
    for(int n = 0; n < numNorms; n++) // read the normals
        infile >> norm[n].x >> norm[n].y >> norm[n].z;
    for(int f = 0; f < numFaces; f++)// read the faces
    {
        infile >> face[f].nverts;
        face[f].vert = new VertexId[face[f].nverts];
        for(int i = 0; i < face[f].nverts; i++)
            infile >> face[f].vert[i].vertIndex
                >> face[f].vert[i].normIndex;
    }
    return 0; // success
}
```

Figure 6.78. Reading a mesh into memory from a file.

Because no knowledge of the required mesh size is available before numVerts, numNorms, and numFaces is read, the arrays that hold the vertices, normals, and faces are allocated dynamically at runtime with the proper sizes.

A number of files in this format are available on the internet site for this book. (They have the suffix .3vn) Also, it is straightforward to convert IndexedFace objects in VRML2.0 files to this format.

It is equally straightforward to fashion the method `int Mesh:: writeMesh(char* fileName)` that writes a mesh object to a file.

Write an application that reads mesh objects from files and draws them, and also allows the user to write a mesh object to a file. As examples of simple files for getting started, arrange that the application also can create meshes for a tetrahedron and the simple barn.

6.7.2. Case Study 6.2. Derivation of the Newell Method.

(Level of Effort: II). This study develops the theory behind the **Newell method** for computing the normal to a polygon based on its vertices. The necessary mathematics are presented as needed as the discussion unfolds; you are asked to show several of the intermediate results.

In these discussions we work with the polygonal face P given by the N 3D vertices:

$$P = \{P_0, P_1, \dots, P_{N-1}\} \quad (6.50)$$

We want to show why the formulas in Equation 6.1 provide an exact computation of the normal vector $\mathbf{m} = (m_x, m_y, m_z)$ to P when P is planar, and a good direction to use as an “average” normal when P is nonplanar.

Derivation:

A). Figure 6.79 shows P projected (orthographically - along the principal axes) onto each of the principal planes: the $x = 0$, $y = 0$, and $z = 0$ planes. Each projection is a 2D polygon. We first show that the components of \mathbf{m} are proportional to the areas, A_x , A_y , and A_z , respectively, of these projected polygons.

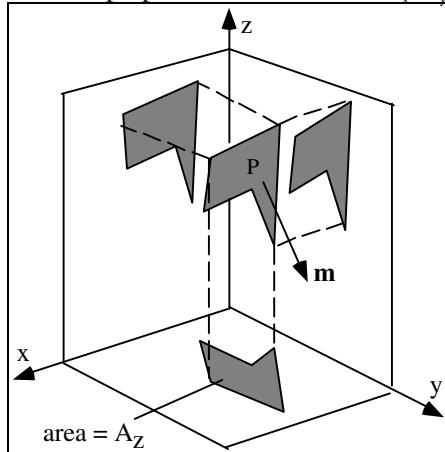


Figure 6.79. Using projected areas to find the normal vector.

For simplicity consider the case where P is a triangle, as shown in Figure 6.80. Call it T , and suppose its unit normal vector is some \mathbf{m} . Further, name as T' its projection onto the plane with unit normal \mathbf{n} . We show that the area of T' , denoted $\text{Area}(T')$, is a certain fraction of the area $\text{Area}(T)$ of T , and that the fraction is a simple dot product:

$$\text{Area}(T') = (\mathbf{m} \cdot \mathbf{n}) \text{Area}(T)$$

Suppose triangle T has edges defined by the vectors \mathbf{v} and \mathbf{w} as shown in the figure.

- a). Show that the area of T is $\text{Area}(T) = \frac{1}{2} |\mathbf{w} \times \mathbf{v}|$, and that $\mathbf{v} \times \mathbf{w} = 2\text{Area}(T)\mathbf{m}$.

We now explore the projection of T onto the plane with normal vector \mathbf{n} .

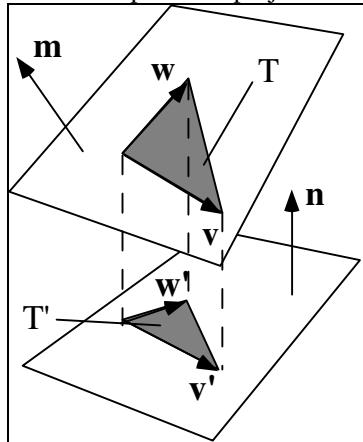


Figure 6.80. Effect of Orthographic projection on area.

This projection T' is defined by the projected vectors \mathbf{w}' and \mathbf{v}' : its area $Area(T') = \frac{1}{2}|\mathbf{w}' \times \mathbf{v}'|$, so $\mathbf{v}' \times \mathbf{w}' = 2Area(T')\mathbf{n}$. We now calculate \mathbf{w}' and \mathbf{v}' , and form $Area(T')$.

- b). Using ideas from Chapter 4, show that that \mathbf{v} projects to \mathbf{v}' given by $\mathbf{v}' = \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})\mathbf{n}$ and similarly that $\mathbf{w}' = \mathbf{w} - (\mathbf{w} \cdot \mathbf{n})\mathbf{n}$.

So we need only relate the sizes of the two cross products.

- c). Use the forms of \mathbf{u}' and \mathbf{v}' to show that: $\mathbf{v}' \times \mathbf{w}' = \mathbf{v} \times \mathbf{w} - (\mathbf{w} \cdot \mathbf{n})(\mathbf{v} \times \mathbf{n}) + (\mathbf{v} \cdot \mathbf{n})(\mathbf{w} \times \mathbf{n}) + (\mathbf{w} \cdot \mathbf{n})(\mathbf{v} \cdot \mathbf{n})(\mathbf{n} \times \mathbf{n})$

and explain why the last term vanishes. Thus we have

$$2 Area(T') \mathbf{n} = \mathbf{v} \times \mathbf{w} - (\mathbf{w} \cdot \mathbf{n})(\mathbf{v} \times \mathbf{n}) + (\mathbf{v} \cdot \mathbf{n})(\mathbf{w} \times \mathbf{n})$$

- d). Dot both sides of this with \mathbf{n} and show that the last two terms drop out, and that we have $2 Area(T') = \mathbf{v} \times \mathbf{w} \cdot \mathbf{n} = 2Area(T) \mathbf{m} \cdot \mathbf{n}$ as claimed.

- e). Show that this result generalizes to the areas of any planar polygon P and its projected image P' .

- f). Recalling that a dot product is proportional to the cosine of an angle, show that $Area(T') = Area(T) \cos \phi$ and state what the angle ϕ is.

- g). Show that the areas A_x, A_y, A_z defined above are simply Km_x, Km_y , and Km_z , respectively, where K is some constant. Hence the areas A_x, A_y , and A_z are in the same ratios as m_x, m_y , and m_z .

B). So to find \mathbf{m} we need only compute the vector (A_x, A_y, A_z) , and normalize it to unit length. We now show how to compute the area of the projection of P of Equation 6.50 onto the xy -plane directly from its vertices. The other two projected areas follow similarly.

Each 3D vertex $P_i = (x_i, y_i, z_i)$ projects onto the xy -plane as $V_i = (x_i, y_i)$. Figure 6.81 shows an example projected polygon P' .

Figure 6.81. Computing the area of a polygon.

Each edge of P' defines a trapezoidal region lying between it and the x -axis.

- h). Show that the area of such a trapezoid is the width of the base times the midpoint of the edge. For instance the area A_0 in the figure is $A_0 = 0.5(x_0 - x_1)(y_0 + y_1)$. This quantity is negative if x_0 lies to the left of x_1 , and is positive otherwise.

We use this same form for each edge. For the i -th edge define

$$A_i = \frac{1}{2}(x_i - x_{next(i)})(y_i + y_{next(i)})$$

where $next(i)$ is 0 if i is equal to $N-1$, and is $i+1$ otherwise.

- i). Show that if two adjacent edges of the polygon are collinear (which would make a cross product based on them zero), the areas contributed by these edges is still properly accounted for.

j). Show that the sum of the A_i properly adds the positive and negative contributions of area to form a resultant sum that is either the area of the polygon, or its negative.

Now we ask which of the two basic directions does \mathbf{m} point in? That is, if you circle the fingers of your right hand around the vertices of the polygon moving from P_0 to P_1 , to P_2 etc., the direction of the arrow in Figure 6.82, does \mathbf{m} point along your thumb or in the opposite direction?

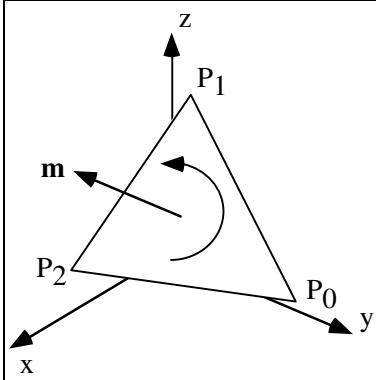


Figure 6.82. The direction of the normal found by the Newell method.

j). Show that \mathbf{m} *does* point as shown in Figure 6.82. Thus for a mesh that has a well-defined inside-outside, we can say that \mathbf{m} is the outward pointing normal if the vertices are labeled CCW as seen from the outside.

6.7.3. Case Study 6.3. The Prism.

(Level of Effort: III) Write an application that allows the user to specify the polygonal base of a prism using the mouse. It then creates the vertex, normal, and face lists for the prism, and displays it.

Figure 6.83a shows the user's "drawing area", a square presented on the screen. The user lays down a sequence of points in this square with the mouse, terminating the process with a right click.

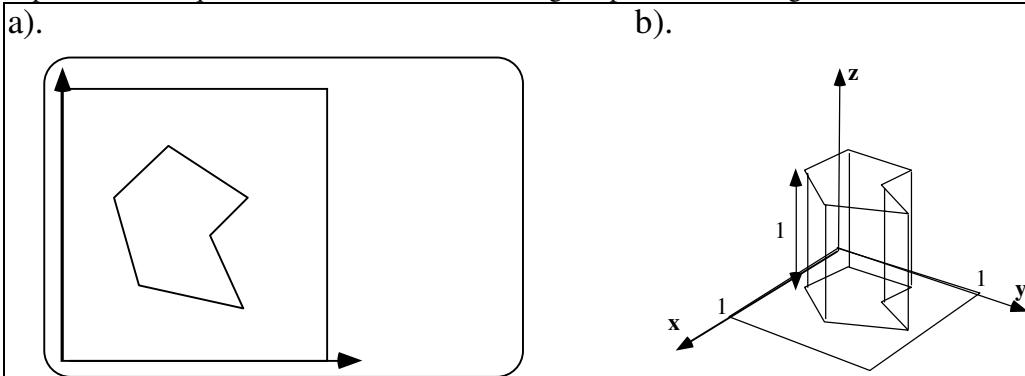


Figure 6.83. Designing and building a prism mesh.

In 3D space the corresponding square is considered to be a unit square lying in the xy -plane as suggested in Figure 6.83b, and the base of the prism lies within it. This establishes the size of the base in the 3D "world". The prism is considered to be the base polygon after it is swept one unit in the direction of the z -axis. Exercise the program on several prisms input by the user. See if your implementation of OpenGL properly draws non-convex base polygons.

6.7.4. Case Study 6.4. Prism Arrays and Extruded Quad-strips.

(Level of Effort: III) Write the two methods described in Section 6.4.2:

```
void Mesh:: makePrismArray(<... suitable arguments ..>);
void Mesh:: makeExtrudedQuadStrip(Point2 p[], int numPts, Vector3 d);
```

that create meshes for an array of prisms, and for an extruded quad-strip.

a). **Arrays of prisms:** Choose an appropriate data type to represent an array of prisms. Note that `makePrismArray()` is similar to the method that makes a mesh for a single prism. Exercise the first method on at least the two block letters with the shapes ‘K’ and ‘W’. (Try ‘D’ also if you wish.)

b). **Extruded quad-strips used to form tubes:** The process of building the vertex, normal, and face lists of a mesh is really a matter of keeping straight the many indices for these arrays. To assist in developing this method, consider a quad-strip base polygon described as in Equation 6.9 by the vertices

$$\text{quad-strip} = \{p_0, p_1, \dots, p_{M-1}\}$$

where $p_i = (x_i, y_i, 0)$ lies in the xy -plane, as shown in Figure 6.84a. When extruded, each successive pair of vertices forms a “waist” of the tube, as shown in Figure 6.84b. There are $\text{num} = M/2 - 1$ segments in the tube.

a). quad-strip in xy -plane b) the four extruded segments

Figure 6.84. Building a mesh from a quad-strip base polygon.

The 0-th waist consists of vertices $p_0, p_1, p_1 + \mathbf{d}$, and $p_0 + \mathbf{d}$, where \mathbf{d} is the extrusion vector. We add vertices to the vertex list as follows: $\text{pt}[4i] = p_{2i}$, $\text{pt}[4i + 1] = p_{2i+1}$, $\text{pt}[4i + 2] = p_{2i+1} + \mathbf{d}$, and $\text{pt}[4i+3] = p_{2i} + \mathbf{d}$, for $i = 0, \dots, \text{num}$ as suggested in Figure 6.84b.

Now for the face list. We first add all of the “outside walls” of each segment of the tube, and then append the “end walls” (i.e. the first end wall uses vertices of the first waist). Each of the num segments has four walls. For each wall we list the four vertices in CCW order as seen from the outside. There are patterns in the various indices encountered, but they are complicated. Check that the following vertex indices are correct for each of the four walls of the k -th segment: The j -th wall of the k -th segment has vertices with indices: i_0, i_1, i_2 , and i_3 , where:

$$i_0 = 4k + j$$

$$i_1 = i_0 + 4$$

$$i_3 = 4k + (j + 3) \% 4$$

$$i_2 = i_3 + 4;$$

for $k = 0, 1, \dots, \text{num}$ and $j = 0, 1, 2, 3$.

What are indices of the two end faces of the tube?

Each face has a normal vector determined by the Newell method, which is straightforward to calculate at the same time the vertex indices are placed in the face list. All vertex normals of a face use the same normal vector: `face[L].normIndex = {L, L, L, L}`, for each L .

Exercise the `makeExtrudedQuadStrip()` method by modeling and drawing some arches, such as the one shown in Figure 6.39, as well as some block letters that permit the use of quad-strips for their base polygon.

6.7.5. Case Study 6.5. Tubes and Snakes based on a Parametric Curve.

(Level of Effort III) Write and test a method

```
void Mesh::makeTube(Point2 P[], int numPts, float t[], int numTimes)
```

that builds a flat-faced mesh based on wrapping the polygon with vertices P_0, P_1, \dots, P_{N-1} about the spine curve $C(t)$. The waists of the tube are formed on the spine at the set of instants t_0, t_1, \dots, t_{M-1} , and a Frenet frame is constructed at each $C(t_i)$. The function $C(t)$ is “hard-wired” into the method as a formula, and its derivatives are formed numerically.

Experiment with the method by wrapping polygons taken from Example 3.6.3 that involve a line jumping back and forth between two concentric circles. Try at least the helix and a Lissajous figure as example spine curves.

6.7.6. Case Study 6.6. Building Discrete-Stepped Surfaces of Revolution.

(Level of Effort: III) Write an application that allows the user to specify the “profile” of an object with the mouse, as in Figure 6.85. It then creates the mesh for the surface of revolution, and displays it. The program also writes the mesh data to a file in the format described in Case Study 6.1.

Figure 6.85a shows the user's “drawing area”, a square presented on the screen. The user lays down a sequence of points in this square with the mouse.

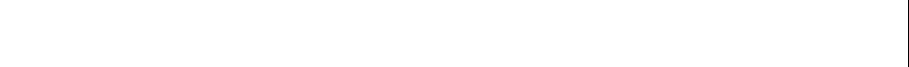


Figure 6.85. Designing a profile for a surface of revolution.

In 3D space the corresponding square is considered to be a unit square lying in the xy -plane. This establishes the size of the profile in the 3D “world”. The surface of revolution is formed by sweeping the profile about the z -axis, in a user-defined number of steps.

Exercise the program on several surfaces of revolution input by the user.

6.7.7. Case Study 6.7. On Edge Lists and Wireframe Models.

(Level of Effort: II) A wireframe version of a mesh can be drawn by drawing a line for each edge of the mesh. Write a routine `void Mesh:: drawEdges(void)` that does this for any given mesh. It simply traverses each face, connecting adjacent vertices with a line. This draws each line twice. (Why?)

In some time critical situations this inefficiency might be intolerable. In such a case an **edge list** can be built for the mesh which contains each edge of the mesh only once. An edge list is an array of index pairs, where the two indices indicate the two endpoints of each edge. Describe an algorithm that builds an edge list for any mesh. It traverses each face of the mesh, noting each edge as it is found, but adding it only if that edge is not already on the list.

Note that it is usually impossible to build a face list from an edge list and a vertex list. Figure 6.86 shows the classic example. From an edge list alone there is no way to tell where the faces are: Even a wireframe model for a cube could be a closed box or an open one. A face list has more information, therefore, than does an edge list.

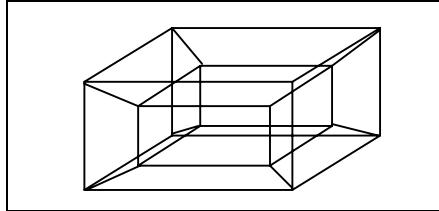


Figure 6.86. An ambiguous object.

6.7.8. Case Study 6.8. Vaulted Ceilings.

(Level of Effort: III) Many classic buildings have arched ceilings or roofs shaped as a **vault**. Figure 6.87a shows a “domical vault” [fleming66] built on a square base. Four “webs”, each a ruled surface, rise in a circular sweep to meet at the peak. The arch shape is based on an ogee arch, as described in Figure 3.88. Part b shows a domical vault built on an octagon, having eight webs. This arch shape is based on the pointed arch described in Figure 3.87. Write a function that creates a mesh model for a domical vault built on a cube, and on a octagon.

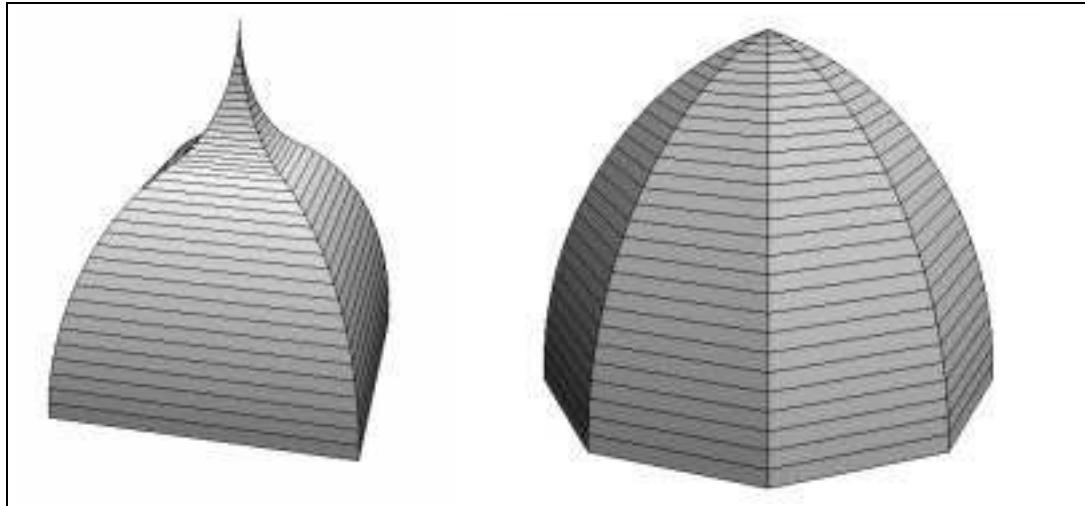


Figure 6.87. Examples of vaulted ceilings.

6.7.9. Case study 6.9. On Platonic Solids.

(Level of Effort: II) Create files (in the format described in Case Study 6.1) for each of the Platonic solids. Experiment by reading in each file into an application and drawing the associated object.

6.7.10. Case Study 6.10. On Archimedean solids

(Level of Effort: II) Pictures of the 13 Archimedean solids may be found in many sources (e.g. [kappraff91], [wenninger71]). Each can be formed by truncating one of the Platonic solids as described in Section 6.3.3.

Create files (in the format described in Case Study 6.1) for each of the following polyhedra:

- the Buckyball;
- the truncated cube;
- the cuboctahedron. (The cube is truncated: new vertices are formed from the midpoints of each of its edges).

Test your files by reading each object into an application and drawing it.

6.7.11. Case Study 6.11. Algebraic form for the Quadric Surfaces.

(Level of Effort: I) The implicit forms for the quadric surfaces share a compact and useful form based on a matrix. For example, the implicit form $F(x, y, z)$ of the generic sphere is given in Figure 6.64 as $x^2 + y^2 + z^2 - 1$. By inspection this can be written as the **quadratic form**:

$$F(x, y, z) = (x, y, z, 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad (6.51)$$

or more compactly, using the homogeneous representation of point (x, y, z) given by $P^T = (x, y, z, 1)$:

$$F(x, y, z) = P^T R_{sphere} P \quad (6.52)$$

where R_{sphere} is the 4 by 4 matrix displayed in Equation 6.51. The point (x, y, z) is on the ellipsoid whenever this function evaluates to 0. The implicit forms for the other quadric surfaces all have the same form; only the matrix R is different. For instance, the matrices for the hyperboloid of two sheets and the elliptic paraboloid are given by:

$$R_{hyperboloid2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}, R_{ellipticParab} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & -\frac{1}{2} & 0 \end{pmatrix}. \quad (6.53)$$

a). What are matrices for the remaining three shapes?

Transforming quadric Surfaces.

Recall from Section 6.5.3 that when an affine transformation with matrix M is applied to a surface with implicit function $F(P)$, the transformed surface has implicit function $F(M^{-1}P)$.

b). Show that when a quadric surface is transformed, its implicit function becomes:

$$G(P) = (M^{-1}P)^T R (M^{-1}P) \text{ which is easily manipulated (as in Appendix 2) into the form}$$

$$G(P) = P^T (M^{-T} R M^{-1}) P.$$

Thus the transformed surface *is also a quadric surface* with a different defining matrix. This matrix depends both on the original shape of the quadric and on the transformation.

For example, to convert the generic sphere into an ellipsoid that extends from $-a$ to a in x , $-b$ to b in y , and $-c$ to c in z , use the scaling matrix:

$$M = \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

c). Find M^{-1} and show that the matrix for the ellipsoid is:

$$M^{-T} R M^{-1} = \begin{pmatrix} \frac{1}{a^2} & 0 & 0 & 0 \\ 0 & \frac{1}{b^2} & 0 & 0 \\ 0 & 0 & \frac{1}{c^2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and write the ellipsoid's implicit form.

d). Find the defining matrix for an elliptic cone that has been scaled by 2 in the x -direction, by 3 in the y -direction, and then rotated through 30° about the y -axis.

e). Show that the matrix R in the implicit function $F(P)=P^T R P$ for a quadric surface can always be taken to be symmetric. (Hint: write R as the sum of a symmetric and an antisymmetric part, and show that the antisymmetric part has no effect on the surface shape.)

6.7.12. Case Study 6.12. Superquadric Scenes.

(Level of Effort: III) Write an application that can create generic superquadrics with user-selected bulge factors, and which place several of them in the scene at different sizes and orientations.

6.7.13. Case Study 6.13. Drawing Smooth Parametric Surfaces.

(Level of Effort III) Develop a function that creates a mesh model of any well-behaved smooth surface given by $P(u, v) = (X(u, v), Y(u, v), Z(u, v))$. It “samples” the surface at numValuesU uniformly spaced values in u between u_{Min} and u_{Max} , and at numValuesV values in v between v_{Min} and v_{Max} . The functions $X()$, $Y()$, and $Z()$, as well as the normal component functions $nx()$, $ny()$, and $nz()$, are “hard-wired into the routine. It

builds the vertex list and normal list based on these sample values, and creates a face list consisting of quadrilaterals. The only difficult part is keeping straight the indices of the vertices for each face in the face list. The suggested skeleton shown in Figure 6.86 may prove helpful.

Apply this function to building an interesting surface of revolution, and a height field.

```
void Mesh:: makeSurfaceMesh()
{
    int i, j, numValsU = numValsV = 40; // set these
    double u, v, uMin = vMin = -10.0, uMax = vMax = 10.0;
    double delU = (uMax - uMin)/(numValsU - 1);
    double delV = (vMax - vMin)/(numValsV - 1);

    numVerts = numValsU * numValsV + 1; // total # of vertices
    numFaces = (numValsU - 1) * (numValsV - 1); // # of faces
    numNorms = numVerts; // for smooth shading - one normal per vertex
    pt = new Point3[numVerts]; assert(pt != NULL); // make space
    face = new Face[numFaces]; assert(face != NULL);
    norm = new Vector3[numNorms]; assert(norm != NULL);

    for(i = 0, u = uMin; i < numValsU; i++, u += delU)
        for(j = 0, v = vMin; j < numValsV; j++, v += delV)
    {
        int whichVert = i * numValsV + j; // index of the vertex and normal
        // set this vertex: use functions X, Y, and Z
        pt[whichVert].set(X(u, v), Y(u, v), Z(u, v));
        // set the normal at this vertex: use functions nx, ny, nz
        norm[whichVert].set(nx(u, v), ny(u, v), nz(u, v));
        normalize(norm[whichVert]);
        // make quadrilateral
        if(i > 0 && j > 0) // when to compute next face
        {
            int whichFace = (i - 1) * (numValsV - 1) + (j - 1);
            face[whichFace].vert = new VertexID[4];
            assert(face[whichFace].vert != NULL);
            face[whichFace].nVerts = 4;
            face[whichFace].vert[0].vertIndex = // same as norm index
            face[whichFace].vert[0].normIndex = whichVert;
            face[whichFace].vert[1].vertIndex =
            face[whichFace].vert[1].normIndex = whichVert - 1;
            face[whichFace].vert[2].vertIndex =
            face[whichFace].vert[2].normIndex = whichVert - numValsV - 1;
            face[whichFace].vert[3].vertIndex =
            face[whichFace].vert[3].normIndex = whichVert - numValsV;
        }
    }
}
```

Figure 6.88. Skeleton of a mesh creation routine for a smooth surface.

6.7.14. Case Study 11.14. Taper, Twist, Bend, and Squash it.

(Level of Effort III) It is useful to have a method for **deforming** a 3D object in a controlled way. For instance, in an animation a bouncing rubber ball is seen to deform into a squished version as it hits the floor, then to regain its spherical shape as it bounces up again. Or a mound of jello bends and wiggles, or a flag waves in the breeze. In cases like these it is important to have the deformations look natural, following rules of physics that take into account conservation of mass, elasticity, and the like. **Physically-based modeling**, that attempts to mirror how actual objects behave under various forces is a large and fascinating subject, described in many books and articles as in [watt92,bloomenthal97].

You can also produce purely geometrical deformations [barr84], chosen by the designer for their visual effect. For instance, it is straightforward to **taper** an object along an axis, as suggested in Figure 6.89.

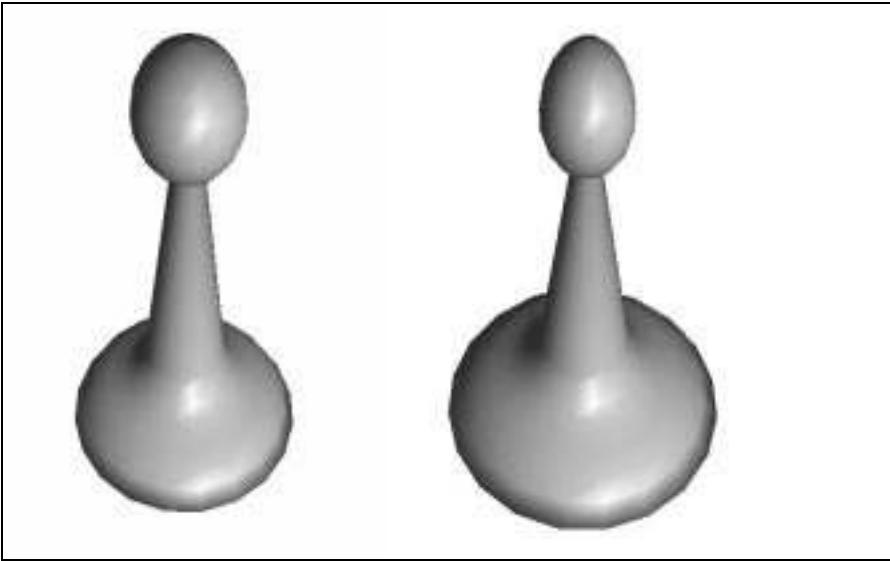


Figure 6.89. The pawn before and after tapering.

This is achieved by scaling all points in the x and y dimensions by amounts that vary with z , according to some profile function, say $g(z)$. This defines a (non-affine) transformation that can be written as a scaling matrix

$$M = \begin{pmatrix} g(z) & 0 & 0 \\ 0 & g(z) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (6.54)$$

If the undeformed surface has parametric representation $P(u, v) = (X(u, v), Y(u, v), Z(u, v))$ then this deformation converts it to

$$P'(u, v) = (X(u, v)g(Z(u, v)), Y(u, v)g(Z(u, v)), Z(u, v)) \quad (6.55)$$

For Figure 6.89, the mesh for the pawn was first created, and then each mesh vertex (x, y, z) was altered to (xF, yF, z) , where F is $1 - 0.04 * (z + 6)$ (note: the pawn extends from 0 to -12 in z).

Another useful deformation is **twisting**. To twist about the z -axis, for instance, rotate all points on the object about the z -axis by an angle that depends on z , using the matrix

$$M = \begin{pmatrix} \cos(g(z)) & \sin(g(z)) & 0 \\ -\sin(g(z)) & \cos(g(z)) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (6.56)$$

Figure 6.90 shows the pawn after a linearly increasing twist is applied. The pawn is a surface of revolution about the z -axis, so it doesn't make much sense to twist it about the z -axis. Instead the twist here is about the y -axis, with $g(z) = 0.02\pi|z + 6|$.



Figure 6.90. The pawn after twisting.

Apply these deformations to several mesh models, including the torus. (Note that you cannot use OpenGL's modelview matrix to perform this deformation, since the transformation here is not affine. The vertices in the actual vertex list must be transformed. Bending is another deformation treated by Barr. Refer to his paper [barr84], and experiment with the bending deformation as well.)

6.8. For Further Reading

A number of books are available that treat the definition and generation of surfaces and solids. Rogers and Adams' MATHEMATICAL ELEMENTS FOR COMPUTER GRAPHICS [rogers90] provides a clear introduction to curves and surfaces, as does Faux and Pratt's COMPUTATIONAL GEOMETRY FOR DESIGN AND MANUFACTURE [Faux79]. Gray's MODERN DIFFERENTIAL GEOMETRY OF CURVES AND SURFACES WITH MATHEMATICA [gray93] offers a rigorous mathematical treatment of curve and surface shapes, and provides code in Mathematica for drawing them. Mortenson's GEOMETRIC MODELING [mortenson85] also provides an excellent discussion of solid modeling used in the CAD industry.

(for ECE660, Fall, 1999)

CHAPTER 7 Three-Dimensional Viewing

I am a camera with its shutter open, quite passive, recording, not thinking.
Christopher Isherwood, A Berlin Diary

Goals of the Chapter

- To develop tools for creating and manipulating a “camera” that produces pictures of a 3D scene.
- To see how to “fly” a camera through a scene interactively, and to make animations.
- To learn the mathematics that describe various kinds of projections.
- To see how each operation in the OpenGL graphics pipeline operates, and why it is used.
- To build a powerful clipping algorithm for 3D objects.
- To devise a means for producing stereo views of objects.

Preview

Section 7.1 provides an overview of the additional tools that are needed to build an application that lets a camera “fly” through a scene. Section 7.2 defines a camera that produces perspective views, and shows how to make such a camera using OpenGL. It introduces aviation terminology that helps to describe ways to manipulate a camera. It develops some of the mathematics needed to describe a camera’s orientation through a matrix. Section 7.3 develops the Camera class to encapsulate information about a camera, and develops methods that create and adjust a camera in an application.

Section 7.4 examines the geometric nature of perspective projections, and develops mathematical tools to describe perspective. It shows how to incorporate perspective projections in the graphics pipeline, and describes how OpenGL does it. An additional property of homogeneous coordinates is introduced to facilitate this. The section also develops a powerful clipping algorithm that operates in homogeneous coordinate space, and shows how its efficiency is a result of proper transformations applied to points before clipping begins. Code for the clipper is given for those programmers who wish to develop their own graphics pipeline.

Section 7.5 shows how to produce stereo views of a scene in order to make them more intelligible. Section 7.6 develops a taxonomy of the many kinds of projections used in art, architecture, and engineering, and shows how to produce each kind of projection in a program. The chapter closes with a number of Case Studies for developing applications that test the techniques discussed.

7.1 Introduction.

We are already in a position to create pictures of elaborate 3D objects. As we saw in Chapter 5, OpenGL provides tools for establishing a camera in the scene, for projecting the scene onto the camera’s viewplane, and for rendering the projection in the viewport. So far our camera only produces parallel projections. In Chapter 6 we described several classes of interesting 3D shapes that can be used to model the objects we want in a scene, and through the `Mesh` class we have ways of drawing any of them with appropriate shading.

So what’s left to do? For greater realism we want to create and control a camera that produces perspective projections. We also need ways to take more control of the camera’s position and orientation, so that the user can “fly” the camera through the scene in an animation. This requires developing more controls than OpenGL provides. We also need to achieve precise control over the camera’s view volume, which is determined in the perspective case as it was when forming parallel projections: by a certain matrix. This requires a deeper use of homogeneous coordinates than we have used so far, so we develop the mathematics of perspective projections from the beginning, and see how they are incorporated in the OpenGL graphics pipeline. We also describe how clipping is done against the camera’s view volume, which again requires some detailed working with homogeneous coordinates. So we finally see how it is all done, from start to finish! This also provides the underlying theory for those programmers who must develop 3D graphics software without the benefit of OpenGL.

7.2. The Camera Revisited.

It adds a precious seeing to the eye.

In Chapter 5 we used a camera that produces parallel projections. Its view volume is a parallelepiped bounded by six walls, including a near plane and a far plane. OpenGL also supports a camera that creates perspective views of 3D scenes. It is similar in many ways to the camera used before, except that its view volume has a different shape.

Figure 7.1 shows its general form. It has an **eye** positioned at some point in space, and its **view volume** is a portion of a rectangular pyramid, whose apex is at the eye. The opening of the pyramid is set by the **viewangle** θ (see part b of the figure). Two planes are defined perpendicular to the axis of the pyramid: the **near plane** and the **far plane**. Where these planes intersect the pyramid they form rectangular windows. The windows have a certain **aspect ratio**, which can be set in a program. OpenGL clips off any parts of the scene that lie outside the view volume. Points lying inside the view volume are projected onto the **view plane** to a corresponding point P' as suggested in part c. (We shall see that it doesn't matter which plane one uses as the view plane, so for now take it to be the near plane.) With a perspective projection the point P' is determined by finding where a line from the eye to P intersects the view plane. (Contrast this with how a parallel projection operates.)

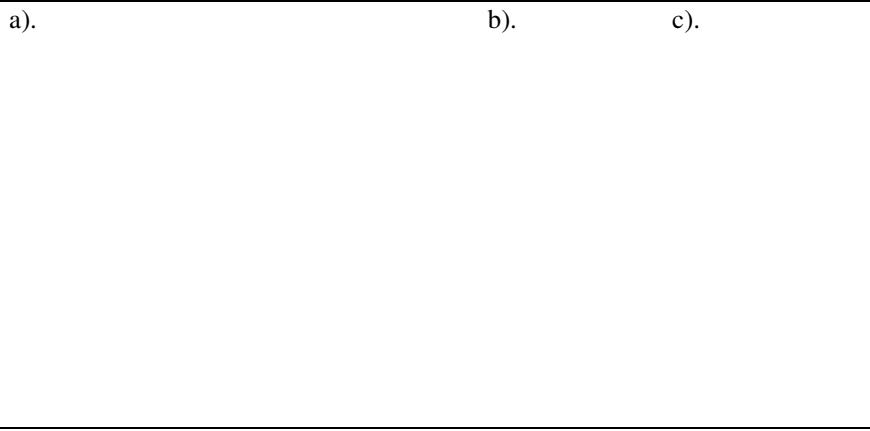


Figure 7.1. A camera to produce perspective views of a scene.

Finally, the image formed on the view plane is mapped into the viewport as shown in part c, and becomes visible on the display device.

7.2.1. Setting the View Volume.

Figure 7.2 shows the camera in its default position, with the eye at the origin and the axis of the pyramid aligned with the z -axis. The eye is “looking” down the negative z -axis.

Figure 7.2. the camera in its default position.

OpenGL provides a simple way to set the view volume in a program. Recall that the shape of the camera's view volume is encoded in the **projection matrix** that appears in the graphics pipeline. The projection matrix is set up using the function `gluPerspective()` with four parameters. The sequence to use is:

```
glMatrixMode(GL_PROJECTION); // make the projection matrix current
glLoadIdentity();           // start with a unit matrix
gluPerspective(viewAngle, aspectRatio, N, F); // load the appropriate
values
```

The parameter `viewAngle`, shown as θ in the figure, is given in degrees, and sets the angle between the top and bottom walls of the pyramid. `aspectRatio` sets the aspect ratio of any window parallel to the xy -plane. The value `N` is the distance from the eye to the near plane, and `F` is the distance from the eye to the far plane. `N` and `F` should be positive. For example, `gluPerspective(60.0, 1.5, 0.3, 50.0)` establishes the view volume to have a vertical opening of 60° , with a window that has an aspect

ratio of 1.5. The near plane lies at $z = -0.3$ and the far plane lies at $z = -50.0$. We see later exactly what values this function places in the projection matrix.

7.2.2. Positioning and pointing the camera.

In order to obtain the desired view of a scene, we move the camera away from its default position shown in Figure 7.2, and aim it in a particular direction. We do this by performing a rotation and a translation, and these transformations become part of the **modelview matrix**, as we discussed in Section 5.6.

We set up the camera's position and orientation in *exactly* the same way as we did for the parallel-projection camera. (The only difference between a parallel- and perspective-projection camera resides in the projection matrix, which determines the *shape* of the view volume.) The simplest function to use is again `gluLookAt()`, using the sequence

```
glMatrixMode(GL_MODELVIEW); // make the modelview matrix current  
glLoadIdentity(); // start with a unit matrix  
gluLookAt(eye.x, eye.y, eye.z, look.x, look.y, look.z, up.x, up.y,  
up.z);
```

As before this moves the camera so its eye resides at point `eye`, and it looks towards the point of interest `look`. The “upward” direction is generally suggested by the vector `up`, which is most often set simply to $(0, 1, 0)$. We took these parameters and the whole process of setting the camera pretty much for granted in Chapter 5. In this chapter we will probe deeper, both to see how it is done and to take finer control over setting the camera. We also develop tools to make *relative* changes to the camera's direction, such as rotating it slightly to the left, tilting it up, or sliding it forward.

The General camera with arbitrary orientation and position.

A camera can have any position in the scene, and any orientation. Imagine a transformation that picks up the camera of Figure 7.2 and moves it somewhere in space, then rotates it around to aim it as desired. We need a way to describe this precisely, and to determine what the resulting modelview matrix will be.

It will serve us well to attach an explicit coordinate system to the camera, as suggested by Figure 7.3. This coordinate system has its origin at the eye, and has three axes, usually called the *u*-, *v*-, and *n*- axes, that define its orientation. The axes are pointed in directions given by the vectors **u**, **v**, and **n** as shown in the figure. Because the camera by default looks down the negative *z*-axis, we say in general that the camera looks down the negative *n*-axis, in the direction **-n**. The direction **u** points off “to the right of” the camera, and direction **v** points “upward”. Think of the *u*-, *v*-, and *n*-axes as “clones” of the *x*-, *y*-, and *z*-axes of Figure 7.2, that are moved and rotated as we move the camera into position.

Figure 7.3. Attaching a coordinate system to the camera.

Position is easy to describe, but orientation is difficult. It helps to specify orientation using the flying terms **pitch**, **heading**, **yaw**, and **roll**, as suggested in Figure 7.4. The *pitch* of a plane is the angle that its longitudinal axis (running from tail to nose and having direction **-n**) makes with the horizontal plane. A plane *rolls* by rotating about this longitudinal axis; its *roll* is the amount of this rotation relative to the horizontal. A plane's *heading* is the direction in which it is headed. (Other terms are *azimuth* and *bearing*.) To find the heading and pitch given **n**, simply express **-n** in spherical coordinates, as shown in Figure 7.5. (See Appendix 2 for a review of spherical coordinates.) The vector **-n** has longitude and latitude given by angles θ and ϕ , respectively. The heading of a plane is given by the longitude of **-n**, and the pitch is given by the latitude of **-n**. Formulas for roll, pitch, and heading in terms of the vectors **u** and **n** are developed in the exercises.

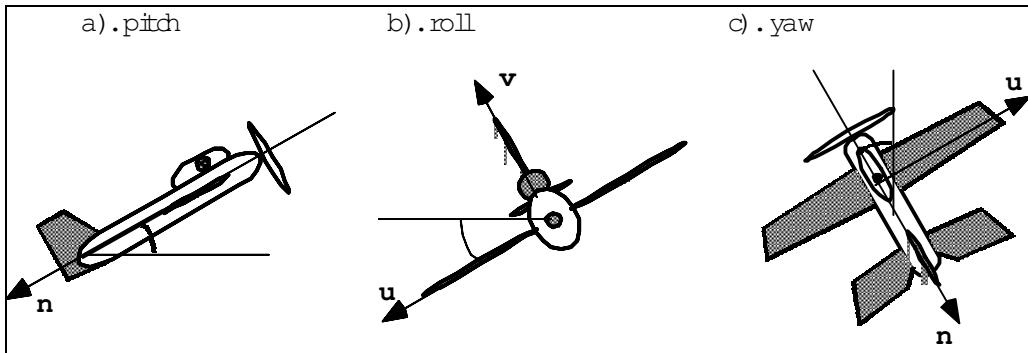


Figure 7.4. A plane's orientation relative to the “world”.

Figure 7.5. The heading and pitch of a plane.

Pitch and *roll* are both nouns and verbs: when used as verbs they describe a change in the plane’s orientation. You can say a plane “pitches up” when it increases its pitch (rotates about its *u*-axis), and that it “rolls” when it rotates about its *n*-axis. The common term for changing heading is *yaw*: to yaw left or right it rotates about its *v*-axis.

These terms can be used with a camera as well. Figure 7.6a shows a camera with the same coordinate system attached: it has *u*, *v*, and *n*- axes, and its origin is at position *eye*. The camera in part b has some non-zero roll, whereas the one in part c has zero roll. We most often set a camera to have zero roll, and call it a “no-roll” camera. The *u*-axis of a no-roll camera is horizontal: that is, perpendicular to the *y*-axis of the world. Note that a no-roll camera can still have an arbitrary *n* direction, so it can have any pitch or heading.

How do we control the roll, pitch, and heading of a camera? *gluLookAt()* is handy for setting up an initial camera, since we usually have a good idea of how to choose *eye* and *look*. But it’s harder to visualize how to choose **up** to obtain a certain roll, and it’s hard to make later relative adjustments to the camera using only *gluLookAt()*. (*gluLookAt()* works with Cartesian coordinates, whereas orientation deals with angles and rotations about axes.) OpenGL doesn’t give direct access to the **u**, **v**, and **n** directions, so we’ll maintain them ourselves in a program. This will make it much easier to describe and adjust the camera.

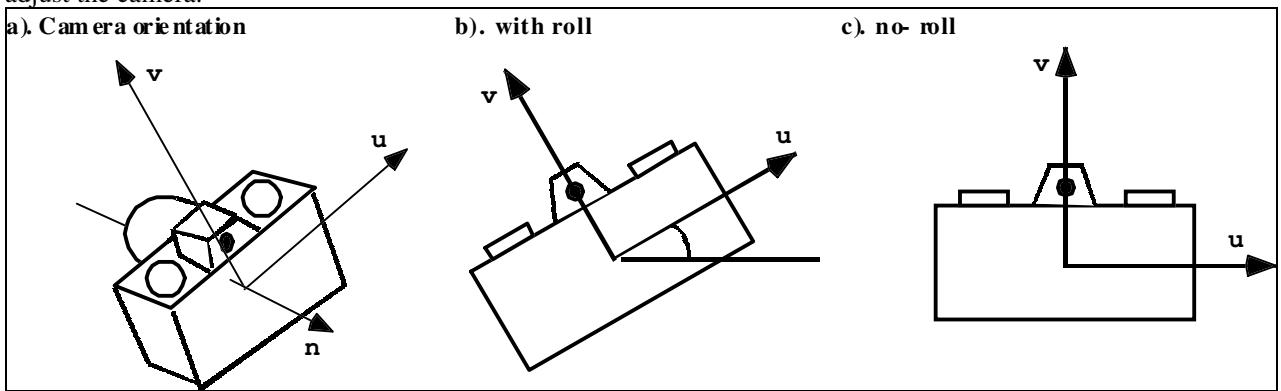


Figure 7.6. Various camera orientations.

What *gluLookAt()* does: some mathematical underpinnings.

What then are the directions **u**, **v**, and **n** when we execute *gluLookAt()* with given values for *eye*, *look*, and **up**? Let’s see exactly what *gluLookAt()* does, and why it does it.

As shown in Figure 7.7a, we are given the locations of *eye* and *look*, and the **up** direction. We immediately know that **n** must be parallel to the vector *eye* - *look*, as shown in Figure 7.7b, so we set **n** = *eye* - *look*. (We’ll normalize this and the other vectors later as necessary.)



Figure 7.7. Building the vectors \mathbf{u} , \mathbf{v} , and \mathbf{n} .

We now need to find \mathbf{u} and \mathbf{v} that are perpendicular to \mathbf{n} and to each other. The \mathbf{u} direction points “off to the side” of a camera, so it is (fairly) natural to make it perpendicular to \mathbf{up} , which the user has said is the “upward” direction. This is the assumption `gluLookAt()` makes in any case, and so the direction \mathbf{u} is made perpendicular to \mathbf{n} and \mathbf{up} . An excellent way to build a vector that is perpendicular to two given vectors is to form their cross product, so we set $\mathbf{u} = \mathbf{up} \times \mathbf{n}$. (The user should not choose an \mathbf{up} direction that is parallel to \mathbf{n} , as then \mathbf{u} would have zero length - why?) We choose $\mathbf{u} = \mathbf{up} \times \mathbf{n}$ rather than $\mathbf{n} \times \mathbf{up}$ so that \mathbf{u} will point “to the right” as we look along $-\mathbf{n}$.

With \mathbf{u} and \mathbf{n} in hand it is easy to form \mathbf{v} : it must be perpendicular to both \mathbf{u} and \mathbf{v} so use a cross product again: $\mathbf{v} = \mathbf{n} \times \mathbf{u}$. Notice that \mathbf{v} will usually not be aligned with \mathbf{up} : \mathbf{v} must be aimed perpendicular to \mathbf{n} , whereas the user provides \mathbf{up} as a suggestion of “upwardness”, and the only property of it that is used is its cross product with \mathbf{n} .

Summarizing: given eye , $look$, and \mathbf{up} , we form

$$\begin{aligned}\mathbf{n} &= eye - look \\ \mathbf{u} &= \mathbf{up} \times \mathbf{n} \\ \mathbf{v} &= \mathbf{n} \times \mathbf{u}\end{aligned}\tag{7.1}$$

and then normalize all three to unit length.

Note how this plays out for the common case where $\mathbf{up} = (0, 1, 0)$. Convince yourself that in this case $\mathbf{u} = (n_z, 0, -n_x)$ and $\mathbf{v} = (-n_x n_y, n_x^2 + n_z^2, -n_z n_y)$. Notice that \mathbf{u} does indeed have a y -component of 0, so it is “horizontal”. Further, \mathbf{v} has a positive y -component, so it is pointed more or less “upward”.

Example 7.2.1. Find the camera coordinate system. Consider a camera with $eye = (4, 4, 4)$ that “looks down” on a look-at point $look = (0, 1, 0)$. Further suppose that \mathbf{up} is initially set to $(0, 1, 0)$. Find \mathbf{u} , \mathbf{v} , and \mathbf{n} . Repeat for $up = (2, 1, 0)$.

Solution: From Equation 7.1 we find: $\mathbf{u} = (4, 0, -4)$, $\mathbf{v} = (-12, 32, -12)$, $\mathbf{n} = (4, 3, 4)$, which are easily normalized to unit length. (Sketch this situation.) Note that \mathbf{u} is indeed horizontal. Check that these are mutually perpendicular. For the case of $up = (2, 1, 0)$ (try to visualize this camera before working out the arithmetic), $\mathbf{u} = (4, -8, 2)$, $\mathbf{v} = (38, 8, -44)$, and $\mathbf{n} = (4, 3, 4)$. Sketch this situation. Check that these vectors are mutually perpendicular.

Example 7.2.2. Building intuition with cameras. To assist in developing geometric intuition when setting up a camera, Figure 7.8 shows two example cameras - each depicted as a coordinate system with a view volume - positioned above the world coordinate system, which is made more visible by grids drawn in the xz -plane. The view volume of both cameras has an aspect ratio of 2. One camera is set with $eye = (-2, 2, 0)$, $look = (0, 0, 0)$, and $\mathbf{up} = (0, 1, 0)$. For this camera we find from Equation 7.1 that $\mathbf{n} = (-2, 2, 0)$, $\mathbf{u} = (0, 0, 2)$, and $\mathbf{v} = (4, 4, 0)$. The figure shows these vectors (\mathbf{v} is drawn the darkest) as well as the \mathbf{up} vector. The second camera uses $eye = (2, 2, 0)$, $look = (0, 0, 0)$, and $\mathbf{up} = (0, 0, 1)$. In this case $\mathbf{u} = (-2, -2, 0)$ and $\mathbf{v} = (0, 0, 8)$. The direction \mathbf{v} is parallel to \mathbf{up} here. Note that this camera appears to be “on its side”: (Check that all of these vectors appear drawn in the proper directions.)

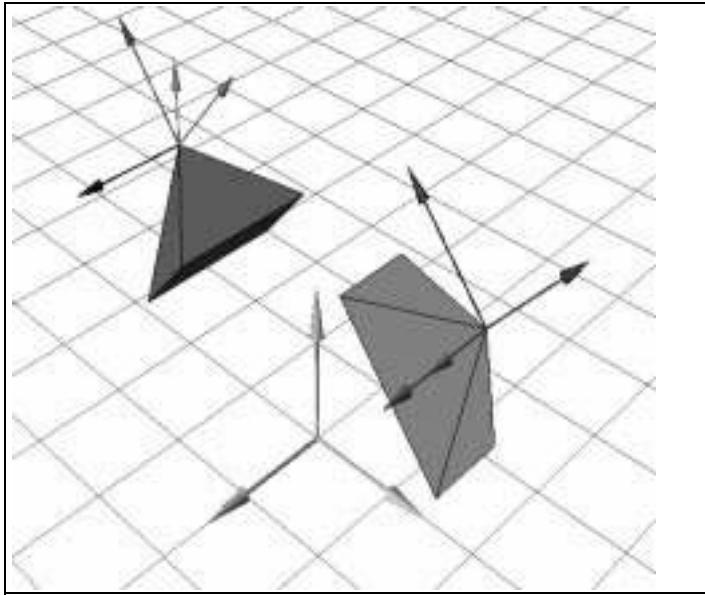


Figure 7.8. Two example settings of the camera.

Finally, we want to see what values `gluLookAt()` places in the modelview matrix. From Chapter 5 we know that the modelview matrix is the product of two matrices, the matrix V that accounts for the transformation of world points into camera coordinates, and the matrix M that embodies all of the modeling transformations applied to points. `gluLookAt()` builds the V matrix and postmultiplies the current matrix by it. Because the job of the V matrix is to convert world coordinates to camera coordinates, it must transform the camera's coordinate system into the generic position for the camera as shown in Figure 7.9. This means it must transform eye into the origin, \mathbf{u} into the vector \mathbf{i} , \mathbf{v} into \mathbf{j} , and \mathbf{n} into \mathbf{k} . There are several ways to derive what V must be, but it's easiest to check that the following matrix does the trick:

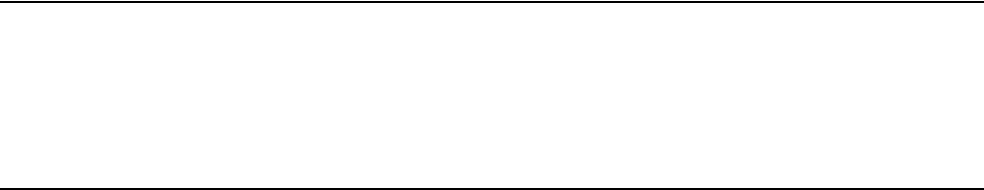


Figure 7.9. The transformation which `gluLookAt()` sets up.

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (7.2)$$

where $(d_x, d_y, d_z) = (-eye \cdot \mathbf{u}, -eye \cdot \mathbf{v}, -eye \cdot \mathbf{n})^T$ ¹. Check that in fact

$$V \begin{pmatrix} eye_x \\ eye_y \\ eye_z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

¹ A technicality: since it's not legal to dot a point and a vector, eye should be replaced here by the vector $(eye - (0,0,0))$.

as desired, where we have extended point *eye* to homogeneous coordinates. Also check that

$$V \begin{pmatrix} u_x \\ u_y \\ u_z \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

and that *V* maps *v* into $(0,1,0,0)^T$ and maps *n* into $(0,0,1,0)^T$. The matrix *V* is created by `gluLookAt()` and is postmultiplied with the current matrix. We will have occasion to do this same operation later when we maintain our own camera in a program.

Practice Exercises.

7.2.1. Finding roll, pitch, and heading given vectors *u*, *v*, and *n*. Suppose a camera is based on a coordinate system with axes in the directions *u*, *v*, and *n*, all unit vectors. The heading and pitch of the camera is found by representing *-n* in spherical coordinates. Using Appendix 2, show that

$$\text{heading} = \arctan(-n_z, -n_x)$$

$$\text{pitch} = \sin^{-1}(-n_y)$$

Further, the roll of the camera is the angle its *u*-axis makes with the horizontal. To find it, construct a vector *b* that is horizontal and lies in the *uv*-plane. Show that $\mathbf{b} = \mathbf{j} \times \mathbf{n}$ has these properties. Show that the angle between *b* and *u* is given by

$$\text{roll} = \cos^{-1} \left(\frac{u_x n_z - u_z n_x}{\sqrt{n_x^2 + n_z^2}} \right)$$

7.2.2. Using up sets v to a “best approximation” to up. Show that using *up* as in Equation 7.1 to set *u* and *v* is equivalent to making *v* the closest vector to *up* that is perpendicular to vector *n*. Use these steps:

- Show that $\mathbf{v} = \mathbf{n} \times (\mathbf{up} \times \mathbf{n})$;
- Use a property of the “triple vector product”, that says $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \cdot \mathbf{c})\mathbf{b} - (\mathbf{a} \cdot \mathbf{b})\mathbf{c}$.
- Show that *v* is therefore the projection of *up* onto the plane with normal *n* (see Chapter 4), and therefore is the closest vector in this plane to *up*.

7.3 Building a Camera in a Program.

It is as interesting and as difficult to say a thing well as to paint it.
Vincent van Gogh

In order to have fine control over camera movements, we create and manipulate our own camera in a program. After each change to this camera is made, the camera “tells” OpenGL what the new camera is.

We create a `Camera` class that knows how to do all the things a camera does. It’s very simple and the payoff is high. In a program we create a `Camera` object called, say, `cam`, and adjust it with functions such as:

```
cam.set(eye, look, up); // initialize the camera - similar to
gluLookAt()
cam.slide(-1,0,-2); // slide the camera forward and to the left
cam.roll(30); // roll it through 30°
cam.yaw(20); // yaw it through 20°
etc.
```

Figure 7.10 shows the basic definition of the `Camera` class. It contains fields for the *eye* and the directions *u*, *v*, and *n*. (`Point3` and `Vector3` are the basic data types defined in Appendix 3.) It also has fields that describe the shape of the view volume: `viewAngle`, `aspect`, `nearDist`, and `farDist`.

```
class Camera{
```

```

private:
    Point3 eye;
    Vector3 u,v,n;
    double viewAngle, aspect, nearDist, farDist; // view volume shape
    void setModelviewMatrix(); // tell OpenGL where the camera is

public:
    Camera(); // default constructor
    void set(Point3 eye, Point3 look, Vector3 up); // like gluLookAt()
    void roll(float angle); // roll it
    void pitch(float angle); // increase pitch
    void yaw(float angle); // yaw it
    void slide(float delU, float delV, float delN); // slide it
    void setShape(float vAng, float asp, float nearD, float farD);
};

```

Figure 7.10. The Camera class definition.

The utility routine `setModelviewMatrix()` communicates the modelview matrix to OpenGL. It is used only by member functions of the class, and needs to be called after each change is made to the camera's position or orientation. Figure 7.11 shows a possible implementation. It computes the matrix of Equation 7.2 based on current values of `eye`, `u`, `v`, and `n`, and loads the matrix directly into the modelview matrix using `glLoadMatrixf()`.

The method `set()` acts just like `gluLookAt()`: it uses the values of `eye`, `look`, and `up` to compute `u`, `v`, and `n` according to Equation 7.1. It places this information in the camera's fields and communicates it to OpenGL. Figure 7.11 shows a possible implementation.

```

void Camera :: setModelViewMatrix(void)
{ // load modelview matrix with existing camera values
    float m[16];
    Vector3 eVec(eye.x, eye.y, eye.z); // a vector version of eye
    m[0] = u.x; m[4] = u.y; m[8] = u.z; m[12] = -eVec.dot(u);
    m[1] = v.x; m[5] = v.y; m[9] = v.z; m[13] = -eVec.dot(v);
    m[2] = n.x; m[6] = n.y; m[10] = n.z; m[14] = -eVec.dot(n);
    m[3] = 0; m[7] = 0; m[11] = 0; m[15] = 1.0;
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(m); // load OpenGL's modelview matrix
}
void Camera:: set(Point3 Eye, Point3 look, Vector3 up)
{ // create a modelview matrix and send it to OpenGL
    eye.set(Eye); // store the given eye position
    n.set(eye.x - look.x, eye.y - look.y, eye.z - look.z); // make n
    u.set(up.cross(n)); // make u = up X n
    n.normalize(); u.normalize(); // make them unit length
    v.set(n.cross(u)); // make v = n X u
    setModelViewMatrix(); // tell OpenGL
}

```

Figure 7.11. The utility routines `set()` and `setModelViewMatrix()`.

The routine `setShape()` is even simpler: It puts the four argument values into the appropriate camera fields, and then calls `gluPerspective(viewangle,aspect,nearDist,farDist)` (along with `glMatrixMode(GL_PROJECTION)` and `glLoadIdentity()`) to set the projection matrix.

The central camera methods are `slide()`, `roll()`, `yaw()`, and `pitch()`, which make *relative* changes to the camera's position and orientation. (The whole reason for maintaining the `eye`, `u`, `v`, and `n` fields in our Camera data structure is so that we have a record of the “current” camera, and can therefore alter it.) We examine how the camera methods operate next.

7.3.1. “Flying” the Camera.

The user flies the camera through a scene interactively by pressing keys or clicking the mouse. For instance, pressing ‘u’ might slide the camera “up” some amount, pressing ‘y’ might yaw it to the left, and pressing ‘f’ might slide it forward. The user can see how the scene looks from one point of view, then

change the camera to a better viewing spot and direction and produce another picture. Or the user can fly around a scene taking different snapshots. If the snapshots are stored and then played back rapidly, an animation is produced of the camera flying around the scene.

There are six degrees of freedom for adjusting a camera: it can be “slid” in three dimensions, and it can be rotated about any of three coordinate axes. We first develop the `slide()` function.

Sliding the camera.

Sliding a camera means to move it along one its *own* axes, that is, in the **u**, **v**, or **n** direction, without rotating it. Since the camera is looking along the negative **n**-axis, movement along **n** is “forward” and “back”. Similarly, movement along **u** is “left” or “right”, and along **v** is “up” or “down”.

It is simple to move the camera along one of its axes. To move it distance D along its **u**-axis, set *eye* to *eye* + D **u**. For convenience we can combine the three possible slides in a single function. `slide(delU, delV, delN)` slides the camera amount `delU` along **u**, `delV` along **v**, and `delN` along **n**:

```
void Camera:: slide(float delU, float delV, float delN)
{
    eye.x += delU * u.x + delV * v.x + delN * n.x;
    eye.y += delU * u.y + delV * v.y + delN * n.y;
    eye.z += delU * u.z + delV * v.z + delN * n.z;
    setModelViewMatrix();
}
```

Rotating the Camera.

We want to roll, pitch, or yaw the camera. This involves a rotation of the camera about one of its own axes. We look at rolling in detail; the other two types of rotation are similar.

To roll the camera is to rotate it about its own **n** axis. This means that both the directions **u** and **v** must be rotated, as shown in Figure 7.12. We form two new axes **u'** and **v'** that lie in the same plane as **u** and **v** yet have been rotated through the angle α degrees.

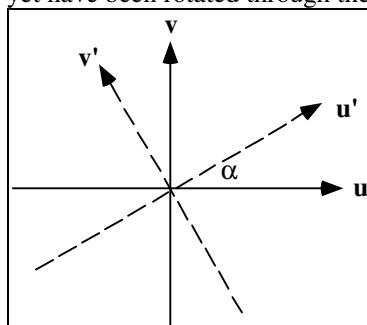


Figure 7.12. Rolling the camera.

So we need only form **u'** as the appropriate linear combination of **u** and **v**, and do similarly for **v'**:

$$\begin{aligned} \mathbf{u}' &= \cos(\alpha) \mathbf{u} + \sin(\alpha) \mathbf{v} \\ \mathbf{v}' &= -\sin(\alpha) \mathbf{u} + \cos(\alpha) \mathbf{v} \end{aligned} \tag{7.3}$$

The new axes **u'** and **v'** then replace **u** and **v** in the camera. This is straightforward to implement.

```
void Camera :: roll(float angle)
{ // roll the camera through angle degrees
    float cs = cos(3.14159265/180 * angle);
    float sn = sin(3.14159265/180 * angle);
    Vector3 t(u); // remember old u
    u.set(cs*t.x - sn*v.x, cs*t.y - sn*v.y, cs*t.z - sn*v.z);
    v.set(sn*t.x + cs*v.x, sn*t.y + cs*v.y, sn*t.z + cs*v.z);
    setModelViewMatrix();
}
```

The functions `pitch()` and `yaw()` are implemented in a similar fashion. See the exercises.

Putting it all together.

We show in Figure 7.13 how the Camera class can be used with OpenGL to fly a camera through a scene. The scene consists of the lone teapot here. The camera is a global object, and is set up in `main()` with a good starting view and shape. When a key is pressed `myKeyboard()` is called, and the camera is slid or rotated depending on which key was pressed. For instance, if ‘P’ is pressed the camera is pitched up by 1 degree. If CTRL F is pressed² (hold down the control key and press ‘f’), the camera is pitched down by 1 degree. After the keystroke has been processed `glutPostRedisplay()` causes `myDisplay()` to be called again to draw the new picture.

Figure 7.13. Application to fly a camera around the teapot.

² On most keyboards pressing CTRL and a letter key returns an ASCII value that is 64 less than the ASCII value returned by the letter itself.

Notice the call to `glutSwapBuffers()`³. This application uses **double-buffering** to produce a rapid and smooth transition between one picture and the next. There are two memory buffers used to store the generated pictures. The display switches from showing one buffer to showing the other under the control of `glutSwapBuffers()`. Each new picture is drawn in the invisible buffer, and when the drawing is complete the display switches to it. Thus the viewer doesn't see the screen erased and the new picture slowly emerge line-by-line, which is visually annoying. Instead the "old" picture is displayed steadily while the picture is being composed off-screen, and then the display switches very rapidly to the newly completed picture.

Drawing SDL scenes using a camera.

It is just as easy to incorporate a camera in an application that reads SDL files, as described in Chapter 5. There are then two global objects:

```
Camera cam;
Scene scn;
```

and in `main()` an SDL file is read and parsed using `scn.read("myScene.dat")`. Finally, in `myDisplay(void)`, simply replace `glutWireTeapot(1.0)` with `scn.drawSceneOpenGL()`;

Practice Exercises.

7.3.1. Implementing `pitch()` and `yaw()`. Write the functions `void Camera:: pitch(float angle)` and `void Camera :: yaw(float angle)` that respectively pitch and yaw the camera. Arrange matters so that a positive yaw yaws the camera to the "left" and a positive pitch pitches the camera "up".

7.3.2. Building a universal `rotate()` method. Write the functions `void Camera:: rotate(Vector3 axis, float angle)` that rotates the camera through angle degrees about axis. It rotates all three axes **u**, **v**, and **n** about the eye.

7.4. Perspective Projections of 3D Objects

*Treat them in terms of the cylinder, the sphere, the cone, all in perspective.
Ashanti proverb*

With the Camera class in hand we can navigate around 3D scenes and readily create pictures. Using OpenGL each picture is created by passing vertices of objects (such as a mesh representing a teapot or chess piece) down the graphics pipeline, as we described in Chapter 5. Figure 7.14 shows the graphics pipeline again, with one new element.

has perspective division too

Figure 7.14. The graphics pipeline revisited.

Recall that each vertex v is multiplied by the modelview matrix (VM). The modeling part (M) embodies all of the modeling transformations for the object; the viewing part (V) accounts for the transformation set by the camera's position and orientation. When a vertex emerges from this matrix it is in **eye coordinates**, that is, in the coordinate system of the eye. Figure 7.15 shows this system, for which the eye is at the origin, and the near plane is perpendicular to the z -axis residing at $z = -N$. A vertex located at P in eye coordinates is passed through the next stages of the pipeline where it is (somehow) projected to a certain point (x^*, y^*) on the near plane, clipping is carried out, and finally the surviving vertices are mapped to the viewport on the display.

Figure 7.15. Perspective projection of vertices expressed in eye coordinates.

³ `glutInitDisplayMode()` must have an argument of `GLUT_DOUBLE` to enable double-buffering.

At this point we must look more deeply into the process of forming perspective projections. We need answers to a number of questions. What operations constitute forming a perspective projection, and how does the pipeline do these operations? What's the relationship between perspective projections and matrices. How does the projection map the view volume into a "canonical view volume" for clipping? How is clipping done? How do homogeneous coordinates come into play in the process? How is the "depth" of a point from the eye retained so that proper hidden surface removal can be done? And what is that "perspective division" step?

We start by examining the nature of perspective projection, independent of specific processing steps in the pipeline. Then we see how the steps in the pipeline are carefully crafted to produce the numerical values required for a perspective projection.

7.4.1. Perspective Projection of a Point.

The fundamental operation is projecting a 3D point to a 2D point on a plane. Figure 7.16 elaborates on Figure 7.15 to show point $P = (P_x, P_y, P_z)$ projecting onto the near plane of the camera to a point (x^*, y^*) . We erect a local coordinate system on the near plane, with its origin on the camera's z -axis. Then it is meaningful to talk about the point x^* units over to the right of the origin, and y^* units above the origin.

Figure 7.16. Finding the projection of a point P in eye coordinates.

The first question is then, what are x^* and y^* ? It's simplest to use similar triangles, and say x^* is in the same ratio to P_x as the distance N is to the distance $|P_z|$. Or since P_z is negative, we can say

$$\frac{x^*}{P_x} = \frac{N}{-P_z}$$

or $x^* = NP_x/(-P_z)$. Similarly $y^* = NP_y/(-P_z)$. So we have that P projects to the point on the viewplane:

$$(x^*, y^*) = \left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z} \right) \quad (\text{the projection of } P) \quad (7.4)$$

An alternate (analytical) method for arriving at this result is given in the exercises.

Example 7.4.1: Where on the viewplane does $P = (1, 0.5, -1.5)$ lie for the camera having a near plane at $N = 1$? **Solution:** Direct use of Equation 7.4 yields $(x^*, y^*) = (0.666, 0.333)$.

We can make some preliminary observations about how points are projected.

- 1). Note the denominator term $-P_z$. It is larger for more remote points (those farther along the negative z -axis), which reduces the values of x^* and y^* . This introduces **perspective foreshortening**, and makes remote parts of an object appear smaller than nearer parts.
- 2). Denominators have a nasty way of evaluating to zero, and P_z becomes 0 when P lies in the "same plane" as the eye: the $z = 0$ plane. Normally we use clipping to remove such offending points before trying to project them.
- 3). If P lies "behind the eye" there is a reversal in sign of P_z , which causes further trouble, as we see later. These points, too, are usually removed by clipping.
- 4). The effect of the near plane distance N is simply to *scale* the picture (both x^* and y^* are proportional to N). So if we choose some other plane (still parallel to the near plane) as the view plane onto which to project pictures, the projection will differ only in size with the projection onto the near plane. Since we ultimately map this projection to a viewport of a fixed size on the display, the size of the projected image makes no difference. This shows that any viewplane parallel to the near plane would work just as well, so we might as well use the near plane itself.

5). Straight lines project to straight lines. Figure 7.17 provides the simplest proof. Consider the line in 3D space between points A and B . A projects to A' and B projects to B' . But do points between A and B project to points directly between A' and B' ? Yes: just consider the plane formed by A , B and the origin. Since any two planes intersect in a straight line, this plane intersects the near plane in a straight line. Thus line segment AB projects to *line segment $A'B'$* .

Figure 7.17. Proof that a straight line projects to a straight line.

Example 7.4.2. Three projections of the barn.

A lot of intuition can be acquired by seeing how a simple object is viewed by different cameras. Here we examine how the edges of the barn defined in Chapter 6 and repeated in Figure 7.18 are projected onto three cameras. The barn has 10 vertices, 15 edges and seven faces.

Figure 7.18. The basic barn revisited.

- **View #1:** We first set the camera's *eye* at $(0, 0, 3)$ and have it look down the negative z -axis, with $\mathbf{u} = (1, 0, 0)$ and $\mathbf{n} = (-1, 0, 0)$. We will set the near plane at distance 1 from the eye. (The near plane happens therefore to coincide with the front of the barn.) In terms of camera coordinates all points on the front wall of the barn have $P_z = -1$ and those on the back wall have $P_z = -2$. So from Equation 7.4 any point (P_x, P_y, P_z) on the front wall projects to:

$$P' = (P_x, P_y) \quad \{ \text{projection of a point on the front wall} \}$$

and any point on the back wall projects to

$$P' = (P_x / 2, P_y / 2). \quad \{ \text{projection of a point on the back wall} \}$$

The foreshortening factor is two for those points on the back wall. Figure 7.19a shows the projection of the barn for this view. Note that edges on the rear wall project at half their true length. Also note that edges of the barn that are actually parallel in 3D *need not* project as parallel. (We shall see that parallel edges that are parallel to the viewplane do project as parallel, but parallel edges that are not parallel to the viewplane are not parallel: they recede to a "vanishing point".)

Figure 7.19. Projections of the barn for views #1 and #2.

View #2: Here the camera has been slid over so that $\text{eye} = (0.5, 0, 2)$, but \mathbf{u} , and \mathbf{n} are the same as in view #1. Figure 7.19b shows the projection.

View #3 Here we use the camera with $\text{Eye} = (2, 5, 2)$ and $\text{look} = (0, 0, 0)$, resulting in Figure 7.20. The world axes have been added as a guide. This shows the barn from an informative point of view. From a wireframe view it is difficult to discern which faces are where.

Figure 7.20. A third view of the barn.

Practice Exercises.

7.4.1. Sketch a Cube in Perspective. Draw (by hand) the perspective view of a **cube** C (axis-aligned and centered at the origin, with sides of length 2) when the eye is at $E = 5$ on the z -axis. Repeat when C is shifted so that its center is at $(1, 1, 1)$.

7.4.2. Where does the ray hit the viewplane? (Don't skip this one.)

We want to derive Equation 7.4 by finding where the ray from the origin to P intersects the near plane.

- a). Show that if this ray is at the origin at $t = 0$ and at P at time $t = 1$, then it has parametric representation $r(t) = Pt$.
- b). Show that it hits the near plane at $t = N/(-P_z)$;
- c). Show that the “hit point” is $(x^*, y^*) = (NP_x/(-P_z), NP_y/(-P_z))$.

7.4.2. Perspective Projection of a Line.

We develop here some interesting properties of perspective projections by examining how straight lines project.

- 1). Lines that are parallel in 3D project to lines, but these lines aren’t necessary parallel. If not parallel, they meet at some “vanishing point.”
- 2). Lines that pass behind the eye of the camera cause a catastrophic “passage through infinity”. (Such lines should be clipped off.)
- 3). Perspective projections usually produce geometrically realistic pictures. But realism is strained for very long lines parallel to the viewplane.

1). Projecting Parallel Lines.

We suppose the line in 3D passes (using camera coordinates) through point $A = (A_x, A_y, A_z)$ and has direction vector $\mathbf{c} = (c_x, c_y, c_z)$. It therefore has parametric form $P(t) = A + \mathbf{c} t$. Substituting this form in Equation 7.4 yields the parametric form for the projection of this line:

$$p(t) = \left(N \frac{A_x + c_x t}{-A_z - c_z t}, N \frac{A_y + c_y t}{-A_z - c_z t} \right) \quad (7.5)$$

(This may not look like the parametric form for a straight line, but it is. See the exercises.) Thus the point A in 3D projects to the point $p(0)$, and as t varies the projected point $p(t)$ moves across the screen (in a straight line). We can discern several important properties directly from this formula.

Suppose the line $A + \mathbf{c}t$ is parallel to the viewplane. Then $c_z = 0$ and the projected line is given by:

$$p(t) = \frac{N}{-A_z} (A_x + c_x t, A_y + c_y t)$$

This is the parametric form for a line with slope c_y/c_x . This slope does not depend on the position of the line, only its direction \mathbf{c} . Thus all lines in 3D with direction \mathbf{c} will project with this slope, so their projections are parallel. We conclude:

If two lines in 3D are parallel to each other *and* to the viewplane, they project to two parallel lines.

Now consider the case where the direction \mathbf{c} is not parallel to the viewplane. For convenience suppose $c_z < 0$, so that as t increases the line recedes further and further from the eye. At very large values of t , Equation 7.5 becomes:

$$p(\infty) = \left(N \frac{c_x}{-c_z}, N \frac{c_y}{-c_z} \right) \quad (\text{the vanishing point for the line}) \quad (7.6)$$

This is called the “**vanishing point**” for this line: it’s the point towards which the projected line moves as t gets larger and larger. Notice that it depends only on the direction \mathbf{c} of the line and not its position (which is embodied in A). Thus all parallel lines share the same vanishing point.

In particular, these lines project to lines that are *not* parallel.

Figure 7.21a makes this more vivid for the example of a cube. Several edges of the cube are parallel: there are those that are horizontal, those that are vertical, and those that recede from the eye. This picture

was made with the camera oriented so that its near plane was parallel to the front face of the cube. Thus in camera coordinates the z -component of \mathbf{c} for the horizontal and vertical edges is 0. The horizontal edges therefore project to parallel lines, and so do the vertical edges. The receding edges, however, are not parallel to the view plane, and hence converge onto a vanishing point (VP). Artists often set up drawings of objects this way, choosing the vanishing point and sketching parallel lines as pointing at the VP. We shall see more on vanishing points as we proceed.

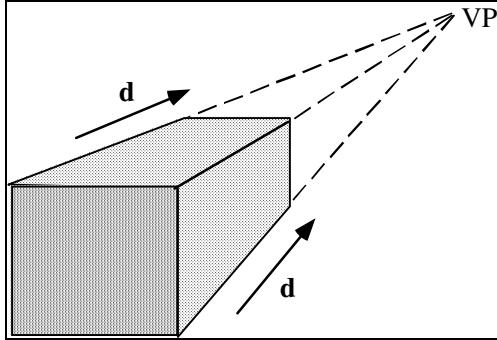


Figure 7.21. The vanishing point for parallel lines.

Figure 7.22 suggests what a vanishing point is geometrically. Looking down onto the camera's xz -plane from above, we see the eye viewing various points on the line AB . A projects to A' , B projects to B' , etc. Very remote points on the line project to VP as shown. The point VP is situated so that the line from the eye through VP is *parallel* to AB (why?).



Figure 7.22. The geometry of a vanishing point.

2). Lines that Pass Behind the Eye.

We saw earlier that trying to project a point that lies in the plane of the eye ($z = 0$ in eye coordinates) results in a denominator of 0, and would surely spell trouble if we try to project it. We now examine the projection of a line segment where one endpoint lies in front of the eye, and one endpoint lies behind.

Figure 7.23 again looks down on the camera from above. Point A lies in front of the eye and projects to A' in a very reasonable manner. Point B , on the other hand, lies behind the eye, and projects to B' , which seems to end up on the wrong side of the viewplane! Consider a point C that moves from A to B , and sketch how its projection moves. As C moves back towards the plane of the eye, its projection slides further and further along the viewplane to the right. As C approaches the plane of the eye its projection spurts off to infinity, and as C moves behind the eye its projection reappears from far off to the left on the viewplane! You might say that the projection has “wrapped around infinity” and come back from the opposite direction [blinn96]. If we tried to draw such a line there would most likely be chaos. Normally all parts of the line closer to the eye than the near plane are clipped off before the projection is attempted.



Figure 7.23. Projecting the line segment AB , with B “behind the eye.”

Example 7.4.3. The Classic Horizontal Plane in Perspective.

A good way to gain insight into vanishing points is to view a grid of lines in perspective, as in Figure 7.24. Grid lines here lie in the xz -plane, and are spaced 1 unit apart. The eye is perched 1 unit above the xz -plane, at $(0, 1, 0)$, and looks along $-\mathbf{n}$, where $\mathbf{n} = (0, 0, 1)$. As usual we take $\mathbf{up} = (0, 1, 0)$. N is chosen to be 1.

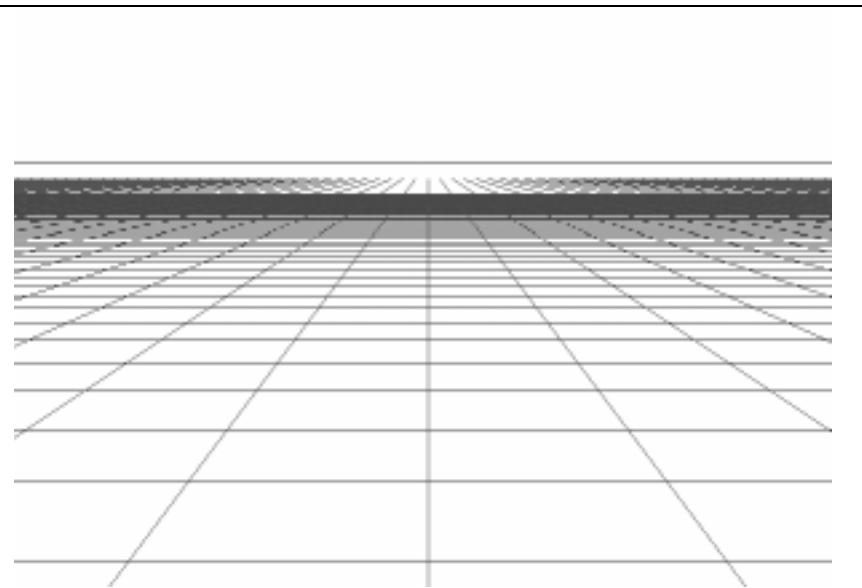


Figure 7.24. Viewing a horizontal grid on the xz -plane.

The grid lines of constant x have parametric form in eye coordinates of $(i, 0, t)$, where $i = \dots, -2, -1, 0, 1, 2, \dots$ and t varies from 0 to ∞ . By Equation 7.4 the i -th line projects to $(-i/t, 2/t)$, which is a line through the vanishing point $(0, 0)$, so all of these lines converge on the same vanishing point, as expected.

The grid lines of constant z are given by $(t, 0, -i)$, where $i = 1, 2, \dots, N$ for some N , and t varies from $-\infty$ to ∞ . These project to $(-t/i, -2/i)$, which appear as horizontal straight lines (check this). Their projections are parallel since the gridlines themselves are parallel to the viewplane. The more remote ones (larger values of i) lie closer together, providing a vivid example of perspective foreshortening. Many of the remote contours are not drawn here, as they become so crowded they cannot be drawn clearly. The **horizon** consists of all the contours where z is very large and negative; it is positioned at $y = 0$.

3). The anomaly of viewing long parallel lines.

Perspective projections seem to be a reasonable model for the way we see. But there are some anomalies, mainly because our eyes do not have planar “view screens”. The problem occurs for very long objects. Consider an experiment, for example, where you look up at a pair of parallel telephone wires, as suggested in Figure 7.25a.

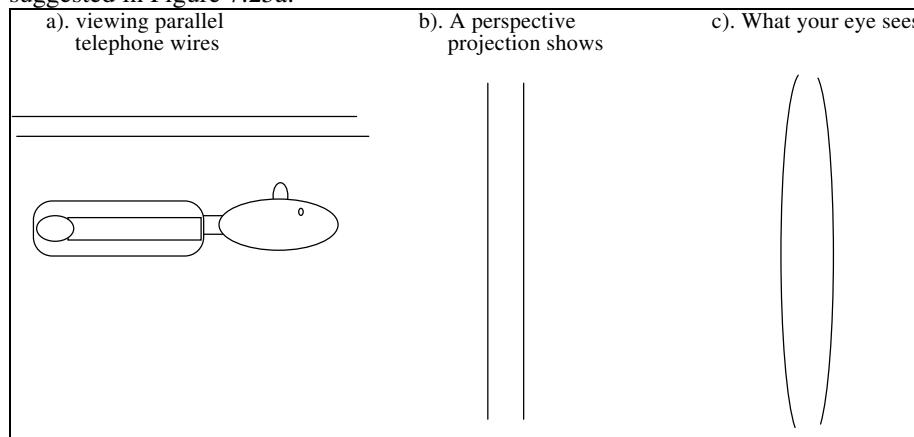


Figure 7.25. Viewing very long parallel wires. (use old 12.14).

For the perspective view, if we orient the viewplane to be parallel to the wires, we know the image will show two straight and parallel lines (part b). But what you see is quite different. The wires appear curved as they converge to “vanishing points” in both directions (part c)! In Practice this anomaly is barely

visible because the window or your eye limits the field of view to a reasonable region. (To see different parts of the wires you have to roll your eyes up and down, which of course rotates your “view planes”.)

Practice Exercises.

7.4.3. Straight lines project as straight lines: the parametric form. Show that the parametric form in Equation 7.5 is that of a straight line. Hint: For the x -component divide the denominator into the numerator to get $-A_x N/A_z + R g(t)$ where R depends on the x -components of A and \mathbf{c} , but not the y -components, and $g(t)$ is some function of t that depends on neither the x nor y -components. Repeat for the y -component, obtaining $-A_y N/A_z + Sg(t)$ with similar properties. Argue why this is the parametric representation of a straight line, (albeit one for which the point does not move with constant speed as t varies).

7.4.4. Derive results for horizontal grids. Derive the parametric forms for the projected grid lines in Example 7.4.3.

7.4.3. Incorporating Perspective in the Graphics Pipeline.

Only a fool tests the depth of the river with both feet.
Paul Cezanne, 1925

We want the graphics pipeline to project vertices of 3D objects onto the near plane, then map them to the viewport. After passing through the modelview matrix, the vertices are represented in the camera’s coordinate system, and Equation 7.4 shows the values we need to compute for the proper projection. We need to do clipping, and then map what survives to the viewport. But we need a little more as well.

Adding Pseudodepth.

Taking a projection discards depth information; that is, how far the point is from the eye. But we mustn’t discard this information completely, or it will be impossible to do hidden surface removal later.

The actual distance of a point P from the eye in camera coordinates is $\sqrt{P_x^2 + P_y^2 + P_z^2}$, which would be cumbersome and slow to compute for each point of interest. All we really need is some measure of distance that tells when two points project to the *same* point on the near plane, which is the closer. Figure 7.26 shows points P_1 and P_2 that both lie on a line from the eye, and therefore project to the same point. We must be able to test whether P_1 obscures P_2 or vice versa. So for each point P that we project we compute a value called the **pseudodepth** that provides an adequate measure of depth for P . We then say that P projects to (x^*, y^*, z^*) , where (x^*, y^*) is the value provided in Equation 7.4 and z^* is its pseudodepth.

Figure 7.26. Is P_1 closer than P_2 or farther away?

What is a good choice for the pseudodepth function? Notice that if two points project to the same point the farther one always has a more negative value of P_z , so we might use $-P_z$ itself for pseudodepth. But it will turn out to be very harmonious and efficient to choose a function with the *same denominator* ($-P_z$) as occurs with x^* and y^* . So we try a function that has this denominator, and a numerator that is linear in P_z , and say that P “projects to”

$$(x^*, y^*, z^*) = \left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right) \quad (7.7)$$

for some choice of the constants a and b . Although many different choices for a and b will do, we choose them so that the pseudodepth varies between -1 and 1 (we see later why these are good choices). Since depth increases as a point moves further down the negative z -axis, we decide that the pseudodepth is -1 when $P_z = -N$, and is +1 when $P_z = -F$. With these two conditions we can easily solve for a and b , obtaining:

$$a = -\frac{F + N}{F - N}, b = \frac{-2FN}{F - N} \quad (7.8)$$

Figure 7.27 plots pseudodepth versus $(-P_z)$. As we insisted it grows from -1 for a point on the near plane up to +1 for a point on the far plane. As P_z approaches 0 (so that the point is just in front of the eye) pseudodepth plummets to $-\infty$. For a point just behind the eye, the pseudodepth is huge and positive. But we will clip off points that lie closer than the near plane, so this catastrophic behavior will never be encountered.

Figure 7.27. Pseudodepth grows as P_z becomes more negative.

Also notice that pseudodepth values bunch together as $(-P_z)$ gets closer to F . Given the finite precision arithmetic of a computer, this can cause a problem when you must distinguish the pseudodepth of two points during hidden surface removal: the points have different true depths from the eye, but their pseudodepths come out with the same value!

Note that defining pseudodepth as in Equation 7.7 causes it to become more positive as P_z becomes more negative. This seems reasonable since depth from the eye increases as P_z moves further along the negative z -axis.

Example 7.4.4. Pseudodepth varies slowly as $(-P_z)$ approaches F .

Suppose $N = 1$ and $F = 100$. This produces $a = -101/99$ and $b = -200/99$, so pseudodepth is given by

$$\text{pseudodepth} \Big|_{N=1, F=100} = \frac{101P_z + 200}{99P_z}$$

This maps appropriately to -1 at $P_z = -N$, and 1 at $P_z = -F$. But close to $-F$ it varies quite slowly with $(-P_z)$. For $(-P_z)$ values of 97, 98, and 99, for instance, this evaluates to 1.041028, 1.040816, and 1.040608.

A little algebra (see the exercises) shows that when N is much smaller than F as it normally will be, pseudodepth can be approximated by

$$\text{pseudodepth} \approx 1 + \frac{2N}{P_z} \quad (7.9)$$

Again you see that it varies more and more slowly as $(-P_z)$ approaches F . But its variation is increased by using large values of N . N should be set as large as possible (but of course not so large that objects nearest to the camera are clipped off!).

Using Homogeneous Coordinates.

Why was there consideration given to having the same denominator for each term in Equation 7.7? As we now show, this makes it possible to represent all of the steps so far in the graphics pipeline as matrix multiplications, offering both processing efficiency and uniformity. (Chips on some graphics cards can multiply a point by a matrix in hardware, making this operation extremely fast!) Doing it this way will also allow us to set things up for a highly efficient and reliable clipping step.

The new approach requires that we represent points in homogeneous coordinates. We have been doing that anyway, since this makes it easier to transform a vertex by the modelview matrix. But we are going to expand the notion of the homogeneous coordinate representation beyond what we have needed before now, and therein find new power. In particular, a matrix will not only be able to perform an affine transformation, it will be able to perform a “perspective transformation.”

Up to now we have said that a point $P = (P_x, P_y, P_z)$ has the representation $(P_x, P_y, P_z, 1)$ in homogeneous coordinates, and that a vector $\mathbf{v} = (v_x, v_y, v_z)$ has the representation $(v_x, v_y, v_z, 0)$. We have simply appended a 1 or 0. This made it possible to use coordinate frames as a basis for representing the points and vectors of interest, and it allowed us to represent an affine transformation by a matrix.

Now we extend the idea, and say that a point $P = (P_x, P_y, P_z)$ has a whole family of homogeneous representations (wP_x, wP_y, wP_z, w) for *any* value of w except 0. For example, the point $(1, 2, 3)$ has the representations $(1, 2, 3, 1), (2, 4, 6, 2), (0.003, 0.006, 0.009, 0.001), (-1, -2, -3, -1)$, and so forth. If

someone hands you a point in this form, say $(3, 6, 2, 3)$ and asks what point is it, just divide through by the last component to get $(1, 2, 2/3, 1)$, then discard the last component: the point in “ordinary” coordinates is $(1, 2, 2/3)$. Thus:

- To convert a point from *ordinary coordinates* to *homogeneous coordinates*, append a 1⁴;
- To convert a point from *homogeneous coordinates* to *ordinary coordinates*, divide all components by the last component, and discard the fourth component.

The additional property of being able to scale all the components of a point without changing the point is really the basis for the name “homogeneous”. Up until now we have always been working with the special case where the final component is 1.

We examine homogeneous coordinates further in the exercises, but now focus on how they operate when transforming points. Affine transformations work fine when homogeneous coordinates are used. Recall that the matrix for an affine transformation always has $(0,0,0,1)$ in its fourth row. Therefore if we multiply a point P in homogeneous representation by such a matrix M , to form $MP = Q$ (recall Equation 5.24), as in the example

$$\begin{pmatrix} 2 & -1 & 3 & 1 \\ 6 & .5 & 1 & 4 \\ 0 & 4 & 2 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix} = \begin{pmatrix} wQ_x \\ wQ_y \\ wQ_z \\ w \end{pmatrix}$$

the final component of Q will always be unaltered: it is still w . Therefore we can convert the Q back to ordinary coordinates in the usual fashion.

But something new happens if we deviate from a fourth row of $(0,0,0,1)$. Consider the important example that has a fourth row of $(0, 0, -1, 0)$, (which is close to what we shall later call the “projection matrix”):

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (\text{the projection matrix - version 1}) \quad (7.10)$$

for any choices of N , a , and b . Multiply this by a point represented in homogeneous coordinates with an arbitrary w :

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix} = \begin{pmatrix} wNP_x \\ wNP_y \\ w(aP_z + b) \\ -wP_z \end{pmatrix}$$

This corresponds to an ordinary point, but which one? Divide through by the fourth component and discard it, to obtain

$$\left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right)$$

⁴ and, if you wish, multiply all four components by any nonzero value.

which is precisely what we need according to Equation 7.7. Thus using homogeneous coordinates allows us to capture perspective using a matrix multiplication! To make it work we must always divide through by the fourth component, a step which is called **perspective division**.

A matrix that has values other than (0,0,0,1) for its fourth row does not perform an affine transformation. It performs a more general class of transformation called a **perspective transformation**. It is a transformation not a projection. A projection reduces the dimensionality of a point, to a 3-tuple or a 2-tuple, whereas a perspective transformation takes a 4-tuple and produces a 4-tuple.

Consider the algebraic effect of putting nonzero values in the fourth row of the matrix, such as (A,B,C,D) . When you multiply the matrix by $(P_x, P_y, P_z, 1)$ (or any multiple thereof) the fourth term in the resulting point becomes $AP_x + BP_y + CP_z + D$, making it linearly dependent on each of the components of P . After perspective division this term appears in the denominator of the point. Such a denominator is exactly what is needed to produce the geometric effect of perspective projection onto a general plane, as we show in the exercises.

The perspective transformation therefore carries a 3D point P into another 3D point P' , according to:

$$(P_x, P_y, P_z) \rightarrow \left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right) \quad \text{"the perspective transformation"} \quad (7.11)$$

Where does the projection part come into play? Further along the pipeline the first two components of this point are used for drawing: to locate in screen coordinates the position of the point to be drawn. The third component is “peeled off” to be used for depth testing. As far as locating the point on the screen is concerned, ignoring the third component is equivalent to replacing it by 0, as in:

$$\left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right) \rightarrow \left(N \frac{P_x}{-P_z}, N \frac{P_y}{-P_z}, 0 \right) \quad \text{"the projection"} \quad (7.12)$$

This is just what we did in Chapter 5 to project a point “orthographically” (meaning perpendicularly to the viewplane) when setting up a camera for our first efforts at viewing a 3D scene. We will study orthographic projections in full detail later. For now we can conclude:

$$(\text{perspective projection}) = (\text{perspective transformation}) + (\text{orthographic projection})$$

This decomposition of a perspective projection into a specific transformation followed by a (trivial) projection will prove very useful, both algorithmically and for understanding better what each point actually experiences as it passes through the graphics pipeline. OpenGL does the transformation step separately from the projection step. In fact it inserts clipping, perspective division, and one additional mapping between them. We next look deeper into the transformation part of the process.

The Geometric Nature of the Perspective Transformation.

The perspective transformation alters 3D point P into another 3D point according to Equation 7.11, to “prepare it” for projection. It is useful to think of it as causing a “warping” of 3D space, and to see how it warps one shape into another. Very importantly, it preserves straightness and “flatness”, so lines transform into lines, planes into planes, and polygonal faces into other polygonal faces. It also preserves “in-between-ness”, so if point a is inside an object, the transformed point will also be inside the transformed object. (Our choice of a suitable pseudodepth function was guided by the need to preserve these properties.) The proof of these properties is developed in the exercises.

Of particular interest is how it transforms the camera’s view volume, because if we are going to do clipping in the warped space, we will be clipping against the warped view volume. The perspective transformation shines in this regard: the warped view volume is a perfect shape for simple and efficient clipping! Figure 7.28 suggests how the view volume and other shapes are transformed. The near plane W at $z = -N$ maps into the plane W' at $z = -1$, and the far plane maps to the plane at $z = +1$. The top wall T is “tilted” into the horizontal plane T' so that is parallel to the z -axis. The bottom wall S becomes the

horizontal S' , and the two side walls become parallel to the z -axis. The camera's view volume is transformed into a parallelepiped!

Figure 7.28. The view volume warped by the perspective transformation.

It's easy to prove how these planes map, because they all involve lines that are either parallel to the near plane or that pass through the eye. Check these carefully:

Fact: Lines through the eye map into lines parallel to the z -axis. **Proof:** All points of such a line project to a single point, say (x^*, y^*) , on the viewplane. So all of the points along the line transform to all of the points (x, y, z) with $x = x^*$, $y = y^*$, and z taking on all pseudodepth values between -1 and 1.

Fact: Lines perpendicular to the z -axis map to lines perpendicular to the z -axis. **Proof:** All points along such a line have the same z -coordinate, so they all map to points with the same pseudodepth value.

Using these facts it is straightforward to derive the exact shape and dimensions of the warped view volume.

The transformation also "warps" objects, like the blocks shown, into new shapes. Figure 7.28b shows a block being projected onto the near plane. Suppose the top edge of its front face projects to $y = 2$ and the top edge of its back face projects to $y = 1$. When this block is transformed, it becomes a truncated pyramid: the top edge of its front face *lies at* $y = 2$, and the top edge of its back face lies at $y = 1$. Things closer to the eye than the near plane become bigger, and things beyond the near plane become smaller. The transformed object is smaller at the back than the front because the original object projects that way. The x and y -coordinates of the transformed object are the x - and y -coordinates of the *projection* of the original object. These are the coordinates you would encounter upon making an orthographic projection of the transformed object. In a nutshell:

The perspective transformation "warps" objects so that, when viewed with an orthographic projection, they appear the same as the original objects do when viewed with a perspective projection.

So all objects are warped into properly foreshortened shapes according to the rules of perspective projection. Thereafter they can be viewed with an orthographic projection, and the correct picture is produced.

We look more closely at the specific shape and dimensions of the transformed view volume.

Details of the Transformed View Volume, and mapping into the Canonical View Volume.

We want to put some numbers on the dimensions of the view volume before and after it is warped. Consider the top plane, and suppose it passes through the point (*left*, *top*, $-N$) at $z = -N$ as shown in Figure 7.29. Because it is composed of lines that pass through the eye and through points in the near plane all of which have a y -coordinate of *top*, it must transform to the plane $y = \text{top}$. Similarly,

Figure 7.29. Details of the perspective transformation.

- the bottom plane transforms to the $y = \text{bott}$ plane;
- the left plane transforms to the $x = \text{left}$ plane;
- the right plane transforms to the $x = \text{right}$.

We now know the transformed view volume precisely: a parallelepiped with dimensions that are related to the camera's properties in a very simple way. This is a splendid shape to clip against as we shall see, because its walls are parallel to the coordinate planes. But it would be an even better shape for clipping if its dimensions didn't depend on the particular camera being used. OpenGL composes the perspective transformation with another mapping that scales and shifts this parallelepiped into the **canonical view volume**, a cube that extends from -1 to 1 in each dimension. Because this scales things differently in the

x- and *y*- dimensions as it “squashes” the scene into a fixed volume it introduces some distortion, but the distortion will be eliminated in the final viewport transformation.

The transformed view volume already extends from -1 to 1 in *z*, so it only needs to be scaled in the other two dimensions. We therefore include a scaling and shift in both *x* and *y* to map the parallelepiped into the canonical view volume. We first shift by $-(right + left)/2$ in *x* and by $-(top + bott)/2$ in *y*. Then we scale by $2/(right - left)$ in *x* and by $2/(top - bott)$ in *y*. When the matrix multiplications are done (see the exercises) we obtain the final matrix:

$$R = \begin{pmatrix} \frac{2N}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2N}{top - bott} & \frac{top + bott}{top - bott} & 0 \\ 0 & 0 & \frac{-(F + N)}{F - N} & \frac{-2FN}{F - N} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (\text{the projection matrix}) \quad (7.13)$$

This is known as the **projection matrix**, and it performs the perspective transformation plus a scaling and shifting to transform the camera’s view volume into the canonical view volume. It is precisely the matrix that OpenGL creates (and by which it multiplies the current matrix) when `glFrustum(left, right, bott, top, N, F)` is executed. Recall that `gluPerspective(viewAngle, aspect, N, F)` is usually used instead, as its parameters are more intuitive. This sets up the same matrix, after computing values for *top*, *bott*, etc. using

$$top = N \tan\left(\frac{\pi}{180} \text{viewAngle} / 2\right)$$

bott = *-top*, *right* = *top * aspect*, and *left* = *-right*.

Clipping Faces against the View Volume.

Recall from Figure 7.14 that clipping is performed after vertices have passed through the projection matrix. It is done in this warped space because the canonical view volume is particularly well suited for efficient clipping. Here we show how to exploit this, and we develop the details of the clipping algorithm.

Clipping in the warped space works because a point lies inside the camera’s view volume if and only if its transformed version lies inside the canonical view volume. Figure 7.30a shows an example of clipping in action. A triangle has vertices v_1 , v_2 , and v_3 . Vertex v_3 lies outside the canonical view volume, *CVV*. The clipper works on edges: it first clips edge v_1v_2 , and finds that the entire edge lies inside *CVV*. Then it clips edge v_2v_3 , and records the new vertex a formed where the edge exits from the *CVV*. Finally it clips edge v_3v_1 and records the new vertex where the edge enters the *CVV*. At the end of the process the original triangle has become a quadrilateral with vertices v_1, v_2, a, b . (We will see later that in addition to identifying the locations of the new vertices, the pipeline also computes new color and texture parameters at these new vertices.)

a). clip a triangle

b). clip an edge

Figure 7.30. Clipping against the canonical view volume.

The clipping problem is basically the problem of clipping a line segment against the *CVV*. We examined such an algorithm, the Cyrus-Beck clipper, in Section 4.8.3. The clipper we develop here is similar to that one, but of course it works in 3D rather than 2D.

Actually, it works in 4D. We will clip in the 4D homogeneous coordinate space called “clip coordinates” in Figure 7.14. This is easier than it might seem, and it will nicely distinguish between points in front of, and behind, the eye.

Suppose we want to clip the line segment *AC* shown in Figure 7.30b against the *CVV*. This means we are given two points in homogeneous coordinates, $A = (a_x, a_y, a_z, a_w)$ and $C = (c_x, c_y, c_z, c_w)$, and we want to

determine which part of the segment lies inside the *CVV*. If the segment intersects the boundary of the *CVV* we will need to compute the intersection point $I = (I_x, I_y, I_z, I_w)$.

As with the Cyrus-Beck algorithm we view the *CVV* as six infinite planes, and consider where the given edge lies relative to each plane in turn. We can represent the edge parametrically as $A + (C-A)t$. It lies at A when t is 0, and at C when t is 1. For each wall of the *CVV* we first test whether A and C lie on the same side of a wall: if they do there is no need to compute the intersection of the edge with the wall. If they lie on opposite sides we locate the intersection point and clip off the part of the edge that lies outside.

So we must be able to test whether a point is on the “outside” or “inside” of a plane. Take the plane $x = -1$, for instance, which is one of the walls of the *CVV*. The point A lies to the right of it (on the “inside”) if

$$\frac{a_x}{a_w} > -1 \quad \text{or} \quad a_x > -a_w \quad \text{or} \quad (a_w + a_x) > 0. \quad (7.14)$$

(When you multiply both sides of an inequality by a negative term you must reverse the direction of the inequality. But we are ultimately dealing with only positive values of a_w here - see the exercises.) Similarly A is inside the plane $x = 1$ if

$$\frac{a_x}{a_w} > 1 \quad \text{or} \quad (a_w - a_x) > 0$$

Blinn [blinn96] calls these quantities the “boundary coordinates” of point A , and he lists the six such quantities that we work with as in Figure 7.31:

<i>boundary coordinate</i>	<i>homogeneous value</i>	<i>clip plane</i>
BC_0	$w+x$	$X=-1$
BC_1	$w-x$	$X=1$
BC_2	$w+y$	$Y=-1$
BC_3	$w-y$	$Y=1$
BC_4	$w+z$	$Z=-1$
BC_5	$w-z$	$Z=1$

Figure 7.31. The boundary codes computed for each end point of an edge.

We form these six quantities for A and again for C . If all six are positive the point lies inside the *CVV*. If any are negative the point lies outside. If both points lie inside we have the same kind of “trivial accept” we had in the Cohen Sutherland clipper of Section 3.3. If A and C lie outside on the same side (corresponding BC ’s are negative) the edge must lie wholly outside the *CVV*.

Trivial accept: both endpoints lie inside the CVV (all 12 BC’s are positive)

Trivial reject: both endpoints lie outside the same plane of the CVV.

If neither condition prevails we must clip segment AC against each plane individually. Just as with the Cyrus-Beck clipper, we keep track of a **candidate interval** (*CI*) (see Figure 4.45), an interval of time during which the edge might still be inside the *CVV*. Basically we know the converse: if t is outside the *CI* we know for sure the edge is *not* inside the *CVV*. The *CI* extends from $t = t_{in}$ to t_{out} .

We test the edge against each wall in turn. If the corresponding boundary codes have opposite signs we know the edge hits the plane at some t_{hit} , which we then compute. If the edge “is entering” (is moving into the “inside” of the plane at t increases) we update $t_{in} = \max(\text{old } t_{in}, t_{hit})$, since it could not possibly be entering at an earlier time than t_{hit} . Similarly, if the edge is exiting, we update $t_{out} = \min(\text{old } t_{out}, t_{hit})$. If at any time the *CI* is reduced to the empty interval (t_{out} becomes $> t_{in}$) we know the entire edge is clipped off and we have an “early out”, which saves unnecessary computation.

It is straightforward to calculate the hit time of an edge with a plane. Write the edge parametrically in homogeneous coordinates:

$$\text{edge}(t) = (a_x + (c_x - a_x)t, a_y + (c_y - a_y)t, a_z + (c_z - a_z)t, a_w + (c_w - a_w)t)$$

If it's the $X = 1$ plane, for instance, when the x -coordinate of $A + (C-A)t$ is 1:

$$\frac{a_x + (c_x - a_x)t}{a_w + (c_w - a_w)t} = 1$$

This is easily solved for t , yielding

$$t = \frac{a_w - a_x}{(a_w - a_x) - (c_w - c_x)} \quad (7.15)$$

Note that t_{hit} depends on only two boundary coordinates. Intersection with other planes yield similar formulas.

This is easily put into code, as shown in Figure 7.32. This is basically the Liang Barsky algorithm [liang84], with some refinements suggested by Blinn [blinn96]. The routine `clipEdge` (`Point4 A, Point4 C`) takes two points in homogeneous coordinates (having fields x, y, z , and w) and returns 0 if no part of AC lies in the CVV , and 1 otherwise. It also alters A and C so that when the routine is finished A and C are the endpoints of the clipped edge.

The routine finds the six boundary coordinates for each endpoint and stores them in `aBC[]` and `cBC[]`. For efficiency it also builds an **outcode** for each point, which holds the *signs* of the six boundary codes for that point. Bit i of A 's outcode holds a 0 if $aBC[i] > 0$ (A is inside the i -th wall) and a 1 otherwise. Using these, a trivial accept occurs when both `aOutcode` and `cOutcode` are 0. A trivial reject occurs when the bit-wise AND of the two outcodes is nonzero.

```
int clipEdge(Point4& A, Point4& C)
{
    double tIn = 0.0, tOut = 1.0, tHit;
    double aBC[6], cBC[6];
    int aOutcode = 0, cOutcode = 0;
    <.. find BC's for A and C ..>
    <.. form outcodes for A and C ..>

    if((aOutcode & cOutcode) != 0) // trivial reject
        return 0;
    if((aOutcode | cOutcode) == 0) // trivial accept
        return 1;

    for(int i = 0; i < 6; i++) // clip against each plane
    {
        if(cBC[i] < 0) // exits: C is outside
        {
            tHit = aBC[i]/(aBC[i] - cBC[i]);
            tOut = MIN(tOut, tHit);
        }
        else if(aBC[i] < 0) // enters: A is outside
        {
            tHit = aBC[i]/(aBC[i] - cBC[i]);
            tIn = MAX(tIn, tHit);
        }
        if(tIn > tOut) return 0; // CI is empty early out
    }
    // update the end points as necessary
    Point4 tmp;
    if(aOutcode != 0) // A is out: tIn has changed
    { // find updated A, (but don't change it yet)
        tmp.x = A.x + tIn * (C.x - A.x);
        tmp.y = A.y + tIn * (C.y - A.y);
        tmp.z = A.z + tIn * (C.z - A.z);
        tmp.w = A.w + tIn * (C.w - A.w);
    }
}
```

```

    }
    if(cOutcode != 0) // C is out: tOut has changed
    {
        // update C (using original value of A)
        C.x = A.x + tOut * (C.x - A.x);
        C.y = A.y + tOut * (C.y - A.y);
        C.z = A.z + tOut * (C.z - A.z);
        C.w = A.w + tOut * (C.w - A.w);
    }
    A = tmp; // now update A
    return 1; // some of the edge lies inside the CVV
}

```

Figure 7.32. The edge clipper (as refined by Blinn).

In the loop that tests the edge against each plane, at most one of the BC's can be negative. (Why?) If A has negative BC the edge must be entering at the hit point; if C has a negative BC the edge must be exiting at the hit point. (Why?) (Blinn uses a slightly faster test by incorporating a mask that tests one bit of an outcode.) Each time tIn or $tOut$ are updated an early out is taken if tIn has become greater than $tOut$.

When all planes have been tested, one or both of tIn and $tOut$ have been altered (why?). A is updated to $A + (i - A)tIn$ if tIn has changed, and C is updated to $A + (C - A)tOut$ if $tOut$ has changed.

Blinn suggests pre-computing the BC's and outcode for every point to be processed. The eliminates the need to re-compute these quantities when a vertex is an endpoint of more than one edge, as is often the case.

Why did we clip against the canonical view volume?

Now that we have seen how easy it is to do clipping against the canonical view volume, we can see the value of having transformed all objects of interest into it prior to clipping. There are two important features of the *CVV*:

1. It is parameter-free: the algorithm needs no extra information to describe the clipping volume. It uses only the values -1 and 1. So the code itself can be highly tuned for maximum efficiency.
2. Its planes are aligned with the coordinate axes (after the perspective transformation is performed). This means that we can determine which side of a plane a point lies on using a single coordinate, as in $a_x > -1$. If the planes were not aligned, an expensive dot product would be needed.

Why did we clip in homogeneous coordinates, rather than after the perspective division step?

This isn't completely necessary, but it makes the clipping algorithm clean, fast, and simple. Doing the perspective divide step destroys information: if you have the values a_x and a_w explicitly you know of course the signs of both of them. But given only the ratio a_x/a_w you can tell only whether a_x and a_w have the same or opposite signs. Keeping values in homogeneous coordinates and clipping points closer to the eye than the near plane automatically removes points that lie behind the eye, such as B in Figure 7.23.

Some "perverse" situations that necessitate clipping in homogeneous coordinates are described in [blinn96, foley90]. They involve peculiar transformations of objects, or construction of certain surfaces, where the original point (a_x, a_y, a_z, a_w) has a negative fourth term, even though the point is in front of the eye. None of the objects we discuss modeling here involve such cases. We conclude that clipping in homogeneous coordinates, although usually not critical, makes the algorithm fast and simple, and brings it almost no cost.

Following the clipping operation **perspective division** is finally done, (as in Figure 7.14), and the 3-tuple (x, y, z) is passed through the viewport transformation. As we discuss next, this transformation sizes and shifts the x - and y - values so they are placed properly in the viewport, and makes minor adjustments on the z - component (pseudodepth) to make it more suitable for depth testing.

The Viewport Transformation.

As we have seen the perspective transformation squashes the scene into the canonical cube, as suggested in Figure 7.33. If the aspect ratio of the camera's view volume (that is, the aspect ratio of the window on the near plane) is 1.5, there is obvious distortion introduced when the perspective transformation scales objects into a window with aspect ratio 1. But the viewport transformation can undo this distortion by mapping a square into a viewport of aspect ratio 1.5. We normally set the aspect ratio of the viewport to be the same as that of the view volume.

Figure 7.33. The viewport transformation restores aspect ratio.

We have encountered the OpenGL function `glViewport(x, y, wid, ht)` often before. It specifies that the viewport will have lower left corner (x, y) in screen coordinates, and will be wid pixels wide and ht pixels high. It thus specifies a viewport with aspect ratio wid/ht . The viewport transformation also maps pseudodepth from the range -1 to 1 into the range 0 to 1.

Review Figure 7.14, which reveals the entire graphics pipeline. Each point P (which is usually one vertex of a polygon) is passed through the following steps:

- P is **extended** to a homogeneous 4-tuple by appending a 1;
- This 4-tuple is multiplied by the **modelview matrix**, producing a 4-tuple giving the position in eye coordinates;
- The point is then multiplied by the **projection matrix**, producing a 4-tuple in clip coordinates;
- The edge having this point as an endpoint is **clipped**;
- **Perspective division** is performed, returning a 3-tuple;
- The **viewport transformation** multiplies the 3-tuple by a matrix: the result (sx, sy, dz) is used for drawing and depth calculations. (sx, sy) is the point in screen coordinates to be displayed; dz is a measure of the depth of the original point from the eye of the camera.

Practice Exercises.

7.4.5. P projects where? Suppose the viewplane is given in camera coordinates by the equation $Ax + By + Cz = D$. Show that any point P projects onto this plane at the point given in homogeneous coordinates

$$P' = (DP_x, DP_y, DP_z, AP_x + BP_y + CP_z)$$

Do it using these steps.

- Show that the projected point is the point at which the ray between the eye and P hits the given plane.
- Show that the ray is given by P_t , and it hits the plane at $t^* = D/(AP_x + BP_y + CP_z)$.
- Show that the projected point - the hit point - is therefore given properly above.
- Show that for the near plane we use earlier, we obtain (x^*, y^*) as given by Equation 7.4.

7.4.6. A revealing approximate form for pseudodepth. Show that pseudodepth $a + b/(-P_z)$, where a and b are given in Equation 7.8, is well approximated by Equation 7.9 when N is much smaller than F .

7.4.7. Points at infinity in homogeneous coordinates. Consider the nature of the homogeneous coordinate point (x, y, z, w) as w becomes smaller and smaller. For $w = .01$ it is $(100x, 100y, 100z)$, for $w = 0.0001$ it is $(10000x, 10000y, 10000z)$, etc. It progresses out" toward infinity" in the direction (x, y, z) . The point with representation $(x, y, z, 0)$ is in fact called a "point at infinity". It is one of the advantages of homogeneous coordinates that such an idealized point has a perfectly "finite" representation: in some mathematical derivations this removes many awkward special cases. For instance, two lines will always intersect, even if they are parallel [ayers67, semple52]. But other things don't work as well. What is the difference of two points in homogeneous coordinates?

7.4.8. How does the perspective transformation affect lines and planes?

We must show that the perspective transformation preserves flatness and in-between-ness.

- Argue why this is proven if we can show that a point P lying on the line between two points A and B transforms to a point P' that lies between the transformed versions of A and B .
- Show that the perspective transformation does indeed produce a point P' with the property just stated.
- Show that each plane that passes through the eye maps to a plane that is parallel to the z -axis.
- Show that each plane that is parallel to the z -axis maps to a plane parallel to the z -axis.
- Show that relative depth is preserved;

7.4.9. The details of the transformed view volume. Show that the warped view volume has the dimensions given in the five points above. You can use the facts developed in the preceding exercise.

7.4.10. Show the final form of the projection matrix. The projection matrix is basically that of Equation 7.10, followed by the shift and scaling described. If the matrix of Equation 7.10 is denoted as M , and T represents the shifting matrix, and S the scaling matrix, show that the matrix product STM is that given in Equation 7.13.

7.4.11. What Becomes of Points Behind the Eye? If the perspective transformation moves the eye off to $-\infty$, what happens to points that lie behind the eye? Consider a line, $P(t)$, that begins at a point in front of the eye at $t = 0$ and moves to one behind the eye at $t = 1$.

- a). Find its parametric form in homogeneous coordinates;
- b). Find the parametric representation after it undergoes the perspective transformation;
- c). Interpret it geometrically. Specifically state what the fourth homogeneous coordinate is geometrically.
A valuable discussion of this phenomenon is given in [blinn78].

Chapter 8 Rendering Faces for Visual Realism.

Goals of the Chapter

- To add realism to drawings of 3D scenes.
- To examine ways to determine how light reflects off of surfaces.
- To render polygonal meshes that are bathed in light.
- To see how to make a polygonal mesh object appear smooth.
- To develop a simple hidden surface removal using a depth-buffer.
- To develop methods for adding textures to the surfaces of objects.
- To add shadows of objects to a scene.

Preview.

Section 8.1 motivates the need for enhancing the realism of pictures of 3D objects. Section 8.2 introduces various shading models used in computer graphics, and develops tools for computing the ambient, diffuse, and specular light contributions to an object's color. It also describes how to set up light sources in OpenGL, how to describe the material properties of surfaces, and how the OpenGL graphics pipeline operates when rendering polygonal meshes.

Section 8.3 focuses on rendering objects modeled as polygon meshes. Flat shading, as well as Gouraud and Phong shading, are described. Section 8.4 develops a simple hidden surface removal technique based on a depth buffer. Proper hidden surface removal greatly improves the realism of pictures.

Section 8.5 develops methods for "painting" texture onto the surface of an object, to make it appear to be made of a real material such as brick or wood, or to wrap a label or picture of a friend around it. Procedural textures, which create texture through a routine, are also described. The thorny issue of proper interpolation of texture is developed in detail. Section 8.5.4 presents a complete program that uses OpenGL to add texture to objects. The next sections discuss mapping texture onto curved surfaces, bump mapping, and environment mapping, providing more tools for making a 3D scene appear real.

Section 8.6 describes two techniques for adding shadows to pictures. The chapter finishes with a number of Case Studies that delve deeper into some of these topics, and urges the reader to experiment with them.

8.1. Introduction.

In previous chapters we have developed tools for modeling mesh objects, and for manipulating a camera to view them and make pictures. Now we want to add tools to make these objects and others look visually interesting, or realistic, or both. Some examples in Chapter 5 invoked a number of OpenGL functions to produce shiny teapots and spheres apparently bathed in light, but none of the underlying theory of how this is done was examined. Here we rectify this, and develop the lore of **rendering** a picture of the objects of interest. This is the business of *computing* how each pixel of a picture should look. Much of it is based on a **shading model**, which attempts to model how light that emanates from light sources would interact with objects in a scene. Due to practical limitations one usually doesn't try to simulate all of the physical principles of light scattering and reflection; this is very complicated and would lead to very slow algorithms. But a number of approximate models have been invented that do a good job and produce various levels of realism.

We start by describing a hierarchy of techniques that provide increasing levels of realism, in order to show the basic issues involved. Then we examine how to incorporate each technique in an application, and also how to use OpenGL to do much of the hard work for us.

At the bottom of the hierarchy, offering the lowest level of realism, is a **wire-frame** rendering. Figure 8.1 shows a flurry of 540 cubes as wire-frames. Only the edges of each object are drawn, and you can see right through an object. It can be difficult to see what's what. (A stereo view would help a little.)

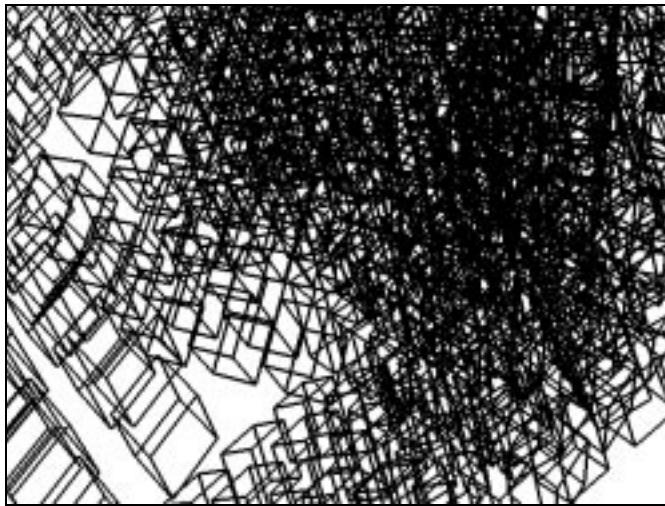


Figure 8.1. A wire-frame rendering of a scene.

Figure 8.2 makes a significant improvement by not drawing any edges that lie behind a face. We can call this a “wire-frame with hidden surface removal” rendering. Even though only edges are drawn the objects now look solid and it is easy to tell where one stops and the next begins. Notice that some edges simply end abruptly as they slip behind a face.

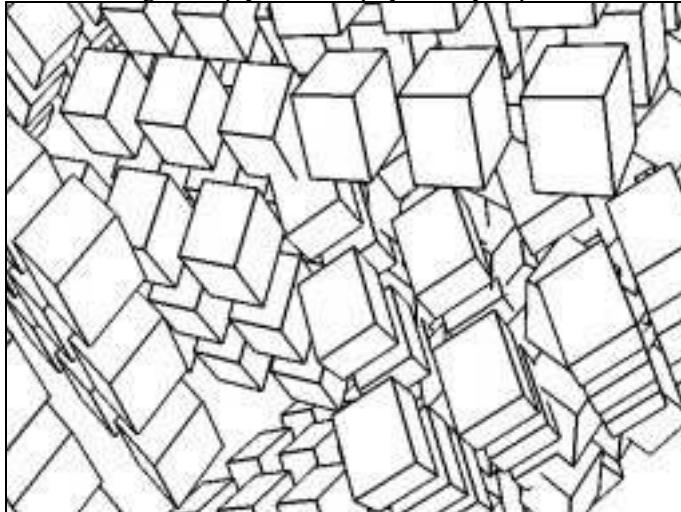


Figure 8.2. Wire-frame view with hidden surfaces removed.

(For the curious: This picture was made using OpenGL with its depth buffer enabled. For each mesh object the faces were drawn in white using `drawMesh()`, and then the edges were drawn in black using `drawEdges()`. Both routines were discussed in Chapter 6.)

The next step in the hierarchy produces pictures where objects appear to be “in a scene”, illuminated by some light sources. Different parts of the object reflect different amounts of light depending on the properties of the surfaces involved, and on the positions of the sources and the camera’s eye. This requires computing the brightness or color of each fragment rather than having the user choose it. The computation requires the use of some shading model that determines the proper amount of light that is reflected from each fragment.

Figure 8.3 shows a scene modeled with polygonal meshes: a buckyball rests atop two cylinders, and the column rests on a floor. Part a shows the wire-frame version, and part b shows a shaded version (with hidden surfaces removed). Those faces aimed toward the light source appear brighter than those aimed away from the source. This picture shows **flat shading**: a calculation of how much light is scattered from each face is computed at a single point, so all points on a face are rendered with the same gray level.

a).wire-frame b). flat shaded

Figure 8.3. A mesh approximation shaded with a shading model. a). wire-frame view b). flat shading,

The next step up is of course to use color. Plate 22 shows the same scene where the objects are given different colors.

In Chapter 6 we discussed building a mesh approximation to a smoothly curved object. A picture of such an object ought to reflect this smoothness, showing the smooth “underlying surface” rather than the individual polygons. Figure 8.4 show the scene rendered using **smooth shading**. (Plate 23 shows the colored version.) Here different points of a face are drawn with different gray levels found through an interpolation scheme known as **Gouraud shading**. The variation in gray levels is much smoother, and the edges of polygons disappear, giving the impression of a smooth, rather than a faceted, surface. We examine Gouraud shading in Section 8.3.

Figure 8.4. The scene rendered with smooth shading.

Highlights can be added to make objects look shiny. Figure 8.5 show the scene with **specular light** components added. (Plate 24 shows the colored version.) The shinier an object is, the more localized are its specular highlights, which often make an object appear to be made from plastic.

Figure 8.5. Adding specular highlights.

Another effect that improves the realism of a picture is shadowing. Figure 8.6 show the scene above with shadows properly rendered, where one object casts a shadow onto a neighboring object. We discuss how to do this in Section 8.6.

Figure 8.6. The scene rendered with shadows.

Adding texture to an object can produce a big step in realism. Figure 8.7 (and Plate 25) show the scene with different textures “painted” on each surface. These textures can make the various surfaces appear to made of some material such as wood, marble, or copper. And images can be “wrapped around” an object like a decal.

Figure 8.7. Mapping textures onto surfaces.

There are additional techniques that improve realism. In Chapter 14 we study ray tracing in depth. Although raytracing is a computationally expensive approach, it is easy to program, and produces pictures that show proper shadows, mirror-like reflections, and the passage of light through transparent objects.

In this chapter we describe a number of methods for rendering scenes with greater realism. We first look at the classical lighting models used in computer graphics that make an object appear bathed in light from some light sources, and see how to draw a polygonal mesh so that it appears to have a smoothly curved surface. We then examine a particular hidden surface removal method - the one that OpenGL uses - and see how it is incorporated into the rendering process. (Chapter 13 examines a number of other hidden surface removal methods.) We then examine techniques for drawing shadows that one object casts upon another, and for adding texture to each surface to make it appear to be made of some particular material, or to have some image painted on it. We also examine chrome mapping and environment mapping to see how to make a local scene appear to be imbedded in a more global scene.

8.2. Introduction to Shading Models.

The mechanism of light reflection from an actual surface is very complicated, and it depends on many factors. Some of these are geometric, such as the relative directions of the light source, the

observer's eye, and the normal to the surface. Others are related to the characteristics of the surface, such as its roughness, and color of the surface.

A shading model dictates how light is scattered or reflected from a surface. We shall examine some simple shading models here, focusing on achromatic light. **Achromatic** light has brightness but no color; it is only a shade of gray. Hence it is described by a single value: intensity. We shall see how to calculate the intensity of the light reaching the eye of the camera from each portion of the object. We then extend the ideas to include colored lights and colored objects. The computations are almost identical to those for achromatic light, except that separate intensities of red, green, and blue components are calculated.

A shading model frequently used in graphics supposes that two types of light sources illuminate the objects in a scene: point light sources and **ambient** light. These light sources "shine" on the various surfaces of the objects, and the incident light interacts with the surface in three different ways:

- Some is absorbed by the surface and is converted to heat;
- Some is reflected from the surface;
- Some is transmitted into the interior of the object, as in the case of a piece of glass.

If all incident light is absorbed, the object appears black and is known as a **black body**. If all is transmitted, the object is visible only through the effects of refraction, which we shall discuss in Chapter 14.

Here we focus on the part of the light that is reflected or scattered from the surface. Some amount of this reflected light travels in just the right direction to reach the eye, causing the object to be seen. The fraction that travels to the eye is highly dependent on the geometry of the situation. We assume that there are two types of reflection of incident light: diffuse scattering and specular reflection.

- **Diffuse scattering**, occurs when some of the incident light slightly penetrates the surface and is re-radiated uniformly in all directions. Scattered light interacts strongly with the surface, and so its color is usually affected by the nature of the surface material.
- **Specular reflections** are more mirror-like and are highly directional: Incident light does not penetrate the object but instead is reflected directly from its outer surface. This gives rise to highlights and makes the surface look shiny. In the simplest model for specular light the reflected light has the same color as the incident light. This tends to make the material look like plastic. In a more complex model the color of the specular light varies over the highlight, providing a better approximation to the shininess of metal surfaces. We discuss both models for specular reflections.

Most surfaces produce some combination of the two types of reflection, depending on surface characteristics such as roughness and type of material. We say that the total light reflected from the surface in a certain direction is the sum of the diffuse component and the specular component. For each surface point of interest we compute the size of each component that reaches the eye. Algorithms are developed next that accomplish this.

8.2.1. Geometric Ingredients for Finding Reflected Light.

*On the outside grows the furside, on the inside grows the skinside;
So the furside is the outside, and the skinside is the inside.
Herbert George Ponting, The Sleeping Bag*

We need to find three vectors in order to compute the diffuse and specular components. Figure 8.8 shows the three principal vectors required to find the amount of light that reaches the eye from a point P .



Figure 8.8. Important directions in computing reflected light.

1. The normal vector, \mathbf{m} , to the surface at P .

2. The vector \mathbf{v} from P to the viewer's eye.
3. The vector \mathbf{s} from P to the light source.

The angles between these three vectors form the basis for computing light intensities. These angles are normally calculated using world coordinates, because some transformations (such as the perspective transformation) do not preserve angles.

Each face of a mesh object has two sides. If the object is solid one is usually the “inside” and one is the “outside”. The eye can then see only the outside (unless the eye is inside the object!), and it is this side for which we must compute light contributions. But for some objects, such as the open box of Figure 8.9, the eye might be able to see the inside of the lid. It depends on the angle between the normal to that side, \mathbf{m}_2 , and the vector to the eye, \mathbf{v} . If the angle is less than 90° this side is visible. Since the cosine of that angle is proportional to the dot product $\mathbf{v} \cdot \mathbf{m}_2$, the eye can see this side only if $\mathbf{v} \cdot \mathbf{m}_2 > 0$.

Figure 8.9. Light computations are made for one side of each face.

We shall develop the shading model for a given side of a face. If that side of the face is “turned away” from the eye there is normally no light contribution. In an actual application the rendering algorithm must be told whether to compute light contributions from one side or both sides of a given face. We shall see that OpenGL supports this.

8.2.2. Computing the Diffuse Component.

Suppose that light falls from a point source onto one side of a facet (a small piece of a surface). A fraction of it is re-radiated diffusely in all directions from this side. Some fraction of the re-radiated part reaches the eye, with intensity w denoted by I_d . How does I_d depend on the directions \mathbf{m} , \mathbf{v} , and \mathbf{s} ?

Because the scattering is uniform in all directions, the orientation of the facet, F , relative to the eye is not significant. Therefore, I_d is independent of the angle between \mathbf{m} and \mathbf{v} (unless $\mathbf{v} \cdot \mathbf{m} < 0$, whereupon I_d is zero.) On the other hand, the amount of light that illuminates the facet *does* depend on the orientation of the facet relative to the point source: It is proportional to the area of the facet that it sees, that is, the area subtended by a facet.

Figure 8.10a shows in cross section a point source illuminating a facet S when \mathbf{m} is aligned with \mathbf{s} . In Figure 8.10b the facet is turned partially away from the light source through angle θ . The area subtended is now only $\cos(\theta)$ as much as before, so that the brightness of S is reduced by this same factor. This relationship between brightness and surface orientation is often called **Lambert's law**. Notice that for θ near 0, brightness varies only slightly with angle, because the cosine changes slowly there. As θ approaches 90° , however, the brightness falls rapidly to 0.

Figure 8.10. The brightness depends on the area subtended.

Now we know that $\cos(\theta)$ is the dot product between normalized versions of \mathbf{s} and \mathbf{m} . We can therefore adopt as the strength of the diffuse component:

$$I_d = I_s \rho_d \frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}| |\mathbf{m}|}$$

In this equation, I_s is the intensity of the light source, and ρ_d is the **diffuse reflection coefficient**. Note that if the facet is aimed away from the eye this dot product is negative and we want I_d to evaluate to 0. So a more precise computation of the diffuse component is:

$$I_d = I_s \rho_d \max\left(\frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}| |\mathbf{m}|}, 0\right) \quad (8.1)$$

This `max` term might be implemented in code (using the `Vector3` methods `dot()` and `length()` - see Appendix 3) by:

```
double tmp = s.dot(m); // form the dot product
double value = (tmp<0) ? 0 : tmp/(s.length() * m.length());
```

Figure 8.11 shows how a sphere appears when it reflects diffuse light, for six reflection coefficients: 0, 0.2, 0.4, 0.6, 0.8, and 1. In each case the source intensity is 1.0 and the background intensity is set to 0.4. Note that the sphere is totally black when ρ_d is 0.0, and the shadow in its bottom half (where the dot product above is negative) is also black.

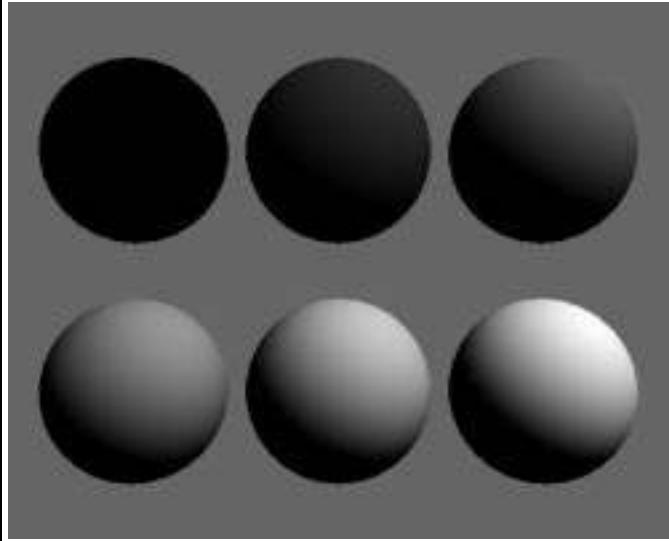


Figure 8.11. Spheres with various reflection coefficients shaded with diffuse light.
(file: fig8.11.bmp)

In reality the mechanism behind diffuse reflection is much more complex than the simple model we have adopted here. The reflection coefficient ρ_d depends on the wavelength (color) of the incident light, the angle θ , and various physical properties of the surface. But for simplicity and to reduce computation time, these effects are usually suppressed when rendering images. A “reasonable” value for ρ_d is chosen for each surface, sometimes by trial and error according to the realism observed in the resulting image.

In some shading models the effect of distance is also included, although it is somewhat controversial. The light intensity falling on facet S in Figure 8.10 from the point source is known to fall off as the inverse square of the distance between S and the source. But experiments have shown that using this law yields pictures with exaggerated depth effects. (What is more, it is sometimes convenient to model light sources as if they lie “at infinity”. Using an inverse square law in such a case would quench the light entirely!) The problem is thought to be in the model: We model light sources as point sources for simplicity, but most scenes are actually illuminated by additional reflections from the surroundings, which are difficult to model. (These effects are lumped together into an ambient light component.) It is not surprising, therefore, that strict adherence to a physical law based on an unrealistic model can lead to unrealistic results.

The realism of most pictures is enhanced rather little by the introduction of a distance term. Some approaches force the intensity to be inversely proportional to the distance between the eye and the object, but this is not based on physical principles. It is interesting to experiment with such effects, and OpenGL provides some control over this effect, as we see in Section 8.2.9, but we don't include a distance term in the following development.

8.2.3. Specular Reflection.

Real objects do not scatter light uniformly in all directions, and so a specular component is added to the shading model. Specular reflection causes highlights, which can add significantly to the realism of a picture when objects are shiny. In this section we discuss a simple model for the behavior of specular light due to Phong [Phong 1975]. It is easy to apply and OpenGL supports a good approximation to it. Highlights generated by Phong specular light give an object a “plastic-like”

appearance, so the Phong model is good when you intend the object to be made of shiny plastic or glass. The Phong model is less successful with objects that are supposed to have a shiny metallic surface, although you can roughly approximate them with OpenGL by careful choices of certain color parameters, as we shall see. More advanced models of specular light have been developed that do a better job of modeling shiny metals. These are not supported directly by OpenGL's rendering process, so we defer a detailed discussion of them to Chapter 14 on ray tracing.

Figure 8.12a shows a situation where light from a source impinges on a surface and is reflected in different directions. In the **Phong model** we discuss here, the amount of light reflected is greatest in the direction of perfect mirror reflection, \mathbf{r} , where the angle of incidence θ equals the angle of reflection. This is the direction in which all light would travel if the surface were a perfect mirror. At other near-by angles the amount of light reflected diminishes rapidly, as indicated by the relative lengths of the reflected vectors. Part b shows this in terms of a “beam pattern” familiar in radar circles. The distance from P to the beam envelope shows the relative strength of the light scattered in that direction.

a). b). c).

Figure 8.12. Specular reflection from a shiny surface.

Part c shows how to quantify this beam pattern effect. We know from Chapter 5 that the direction \mathbf{r} of perfect reflection depends on both \mathbf{s} and the normal vector \mathbf{m} to the surface, according to:

$$\mathbf{r} = -\mathbf{s} + 2 \frac{(\mathbf{s} \cdot \mathbf{m})}{|\mathbf{m}|^2} \mathbf{m} \quad (\text{the mirror-reflection direction}) \quad (8.2)$$

For surfaces that are shiny but not true mirrors, the amount of light reflected falls off as the angle ϕ between \mathbf{r} and \mathbf{v} increases. The actual amount of falloff is a complicated function of ϕ , but in the Phong model it is said to vary as some power f of the cosine of ϕ , that is, according to $(\cos(\phi))^f$, in which f is chosen experimentally and usually lies between 1 and 200.

Figure 8.13 shows how this intensity function varies with ϕ for different values of f . As f increases, the reflection becomes more mirror-like and is more highly concentrated along the direction \mathbf{r} . A perfect mirror could be modeled using $f = \infty$, but pure reflections are usually handled in a different manner, as described in Chapter 14.

similar to old15.14

Figure 8.13. Falloff of specular light with angle.

Using the equivalence of $\cos(\phi)$ and the dot product between \mathbf{r} and \mathbf{v} (after they are normalized), the contribution I_{sp} due to specular reflection is modeled by

$$I_{sp} = I_s \rho_s \left(\frac{\mathbf{r}}{|\mathbf{r}|} \cdot \frac{\mathbf{v}}{|\mathbf{v}|} \right)^f \quad (8.3)$$

where the new term ρ_s is the **specular reflection coefficient**. Like most other coefficients in the shading model, it is usually determined experimentally. (As with the diffuse term, if the dot product $\mathbf{r} \bullet \mathbf{v}$ is found to be negative, I_{sp} is set to zero.)

A boost in efficiency using the “halfway vector”. It can be expensive to compute the specular term in Equation 8.3, since it requires first finding vector \mathbf{r} and normalizing it. In practice an alternate term, apparently first described by Blinn [blinn77], is used to speed up computation. Instead of using the cosine of the angle between \mathbf{r} and \mathbf{v} , one finds a vector halfway between \mathbf{s} and \mathbf{v} , that is, $\mathbf{h} = \mathbf{s} + \mathbf{v}$, as suggested in Figure 8.14. If the normal to the surface were oriented along \mathbf{h} the viewer would see the brightest specular highlight. Therefore the angle β between \mathbf{m} and \mathbf{h} can be used to measure the falloff of specular intensity that the viewer sees. The angle β is not the same as ϕ (in fact β is twice ϕ if the various vectors are coplanar - see the exercises), but this difference can be compensated for by using a different value of the exponent f . (The specular term is not based on physical principles anyway, so it is at least plausible that our adjustment to it yields acceptable

results.) Thus it is common practice to base the specular term on $\cos(\beta)$ using the dot product of \mathbf{h} and \mathbf{m} :

Figure 8.14. The halfway vector.

$$I_{sp} = I_s \rho_s \max(0, \left(\frac{\mathbf{h}}{|\mathbf{h}|} \cdot \frac{\mathbf{m}}{|\mathbf{m}|} \right)^f) \quad \{ \text{adjusted specular term} \} \quad (8.4)$$

Note that with this adjustment the reflection vector \mathbf{r} need not be found, saving computation time. In addition, if both the light source and viewer are very remote then \mathbf{s} and \mathbf{v} are constant over the different faces of an object, so \mathbf{b} need only be computed once.

Figure 8.15 shows a sphere reflecting different amounts of specular light. The reflection coefficient ρ_s varies from top to bottom with values 0.25, 0.5, and 0.75, and the exponent f varies from left to right with values 3, 6, 9, 25, and 200. (The ambient and diffuse reflection coefficients are 0.1 and 0.4 for all spheres.)

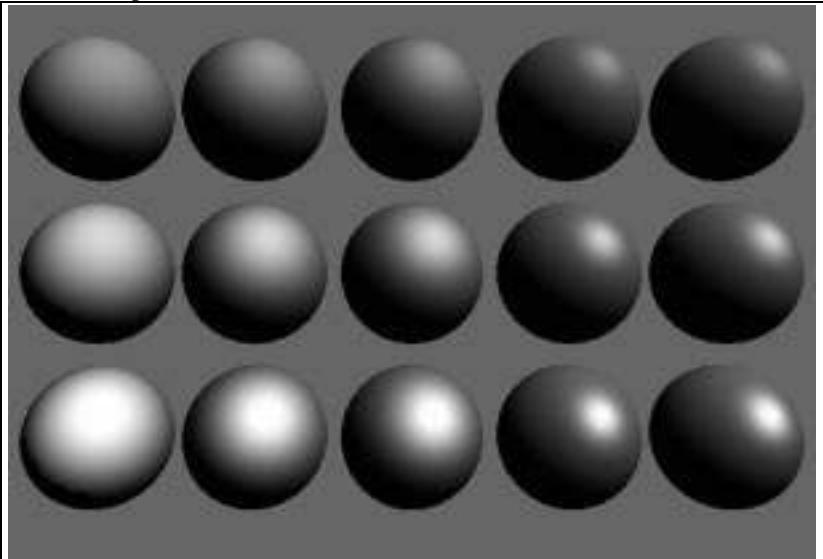


Figure 8.15. Specular reflection from a shiny surface.

The physical mechanism for specularly reflected light is actually much more complicated than the Phong model suggests. A more realistic model makes the specular reflection coefficient dependent on both the wavelength λ (i.e. the color) of the incident light and the angle of incidence θ_i , (the angle between vectors \mathbf{s} and \mathbf{m} in Figure 8.10) and couples it to a “Fresnel term” that describes the physical characteristics of how light reflects off certain classes of surface materials. As mentioned, OpenGL is not organized to include these effects, so we defer further discussion of them until Chapter 14 on ray tracing, where we compute colors on a point by point basis, applying a shading model directly.

Practice Exercises.

8.2.1. Drawing Beam Patterns. Draw beam patterns similar to that in Figure 8.12 for the cases $f=1$, $f=10$, and $f=100$.

8.2.2. On the halfway vector. By examining the geometry displayed in Figure 8.14, show that $\beta = 2\phi$ if the vectors involved are coplanar. Show that this is not so if the vectors are non coplanar. See also [fisher94]

8.2.3. A specular speed up. Schlick [schlick94] has suggested an alternative to the exponentiation required when computing the specular term. Let D denote the dot product $\mathbf{r} \cdot \mathbf{v} / |\mathbf{r}| |\mathbf{v}|$ in Equation 8.3. Schlick suggests replacing D^f with $\frac{D}{f-D+1}$, which is faster to compute. Plot these two functions for values of D in $[0,1]$ for various values of f and compare them. Pay particular attention to values of D near 1, since this is where specular highlights are brightest.

8.2.4. The Role of Ambient Light.

The diffuse and specular components of reflected light are found by simplifying the “rules” by which physical light reflects from physical surfaces. The dependence of these components on the relative positions of the eye, model, and light sources greatly improves the realism of a picture over renderings that simply fill a wireframe with a shade.

But our desire for a simple reflection model leaves us with far from perfect renderings of a scene. As an example, shadows are seen to be unrealistically deep and harsh. To soften these shadows, we can add a third light component called “ambient light.”

With only diffuse and specular reflections, any parts of a surface that are shadowed from the point source receive no light and so are drawn black! But this is not our everyday experience. The scenes we observe around us always seem to be bathed in some soft non-directional light. This light arrives by multiple reflections from various objects in the surroundings and from light sources that populate the environment, such as light coming through a window, fluorescent lamps, and the like. But it would be computationally very expensive to model this kind of light precisely.

Ambient Sources and Ambient Reflections

To overcome the problem of totally dark shadows, we imagine that a uniform “background glow” called **ambient light** exists in the environment. This ambient light source is not situated at any particular place, and it spreads in all directions uniformly. The source is assigned an intensity, I_a . Each face in the model is assigned a value for its **ambient reflection coefficient**, ρ_a (often this is the same as the diffuse reflection coefficient, ρ_d), and the term $I_a \rho_a$ is simply added to whatever diffuse and specular light is reaching the eye from each point P on that face. I_a and ρ_a are usually arrived at experimentally, by trying various values and seeing what looks best. Too little ambient light makes shadows appear too deep and harsh; too much makes the picture look washed out and bland.

Figure 8.16 shows the effect of adding various amounts of ambient light to the diffuse light reflected by a sphere. In each case both the diffuse and ambient sources have intensity 1.0, and the diffuse reflection coefficient is 0.4. Moving from left to right the ambient reflection coefficient takes on values 0.0, 0.1, 0.3, 0.5, and 0.7. With only a modest amount of ambient light the harsh shadows on the underside of the sphere are softened and look more realistic. Too much ambient reflection, on the other hand, suppresses the shadows excessively.

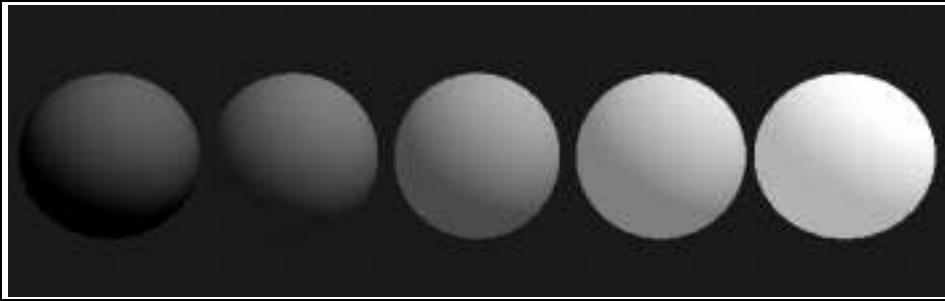


Figure 8.16. On the effect of ambient light.

8.2.5. Combining Light Contributions.

We can now sum the three light contributions - diffuse, specular, and ambient - to form the total amount of light I that reaches the eye from point P :

$$I = I_a \rho_a + I_d \rho_d \times lambert + I_{sp} \rho_s \times phong^f \quad (8.5)$$

where we define the values

$$lambert = \max(0, \frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}| |\mathbf{m}|}), \text{ and } phong = \max(0, \frac{\mathbf{h} \cdot \mathbf{m}}{|\mathbf{h}| |\mathbf{m}|}) \quad (8.6)$$

I depends on the various source intensities and reflection coefficients, as well as on the relative positions of the point P , the eye, and the point light source. Here we have given different names, I_d

and I_{sp} , to the intensities of the diffuse and specular components of the light source, because OpenGL allows you to set them individually, as we see later. In practice they usually have the same value.

To gain some insight into the variation of I with the position of P , consider again Figure 8.10. I is computed for different points P on the facet shown. The ambient component shows no variation over the facet; \mathbf{m} is the same for all P on the facet, but the directions of both \mathbf{s} and \mathbf{v} depend on P . (For instance, $\mathbf{s} = S - P$ where S is the location of the light source. How does \mathbf{v} depend on P and the eye?) If the light source is fairly far away (the typical case), \mathbf{s} will change only slightly as P changes, so that the diffuse component will change only slightly for different points P . This is especially true when \mathbf{s} and \mathbf{m} are nearly aligned, as the value of $\cos()$ changes slowly for small angles. For remote light sources, the variation in the direction of the halfway vector \mathbf{h} is also slight as P varies. On the other hand, if the light source is close to the facet, there can be substantial changes in \mathbf{s} and \mathbf{h} as P varies. Then the specular term can change significantly over the facet, and the bright highlight can be confined to a small portion of the facet. This effect is increased when the eye is also close to the facet -causing large changes in the direction of \mathbf{v} - and when the exponent f is very large.

Practice Exercise 8.2.4. Effect of the Eye Distance. Describe how much the various light contributions change as P varies over a facet when a). the eye is far away from the facet and b). when the eye is near the facet.

8.2.6. Adding Color.

It is straightforward to extend this shading model to the case of colored light reflecting from colored surfaces. Again it is an approximation born from simplicity, but it offers reasonable results and is serviceable.

Chapter 12 provides more detail and background on the nature of color, but as we have seen previously colored light can be constructed by adding certain amounts of red, green, and blue light. When dealing with colored sources and surfaces we calculate each color component individually, and simply add them to form the final color of reflected light. So Equation 8.5 is applied three times:

$$\begin{aligned} I_r &= I_{ar}\rho_{ar} + I_{dr}\rho_{dr} \times \text{lambert} + I_{spr}\rho_{sr} \times \text{phong}^f \\ I_g &= I_{ag}\rho_{ag} + I_{dg}\rho_{dg} \times \text{lambert} + I_{spg}\rho_{sg} \times \text{phong}^f \\ I_b &= I_{ab}\rho_{ab} + I_{db}\rho_{db} \times \text{lambert} + I_{spb}\rho_{sb} \times \text{phong}^f \end{aligned} \quad (8.7)$$

(where *lambert* and *phong* are given in Equation 8.6) to compute the red, green, and blue components of reflected light. Note that we say the light sources have three “types” of color: ambient = (I_{ar}, I_{ag}, I_{ab}) , diffuse = (I_{dr}, I_{dg}, I_{db}) , and specular = $(I_{spr}, I_{spg}, I_{spb})$. Usually the diffuse and specular light colors are the same. Note also that the *lambert* and *phong* terms do not depend on which color component is being computed, so they need only be computed once. To pursue this approach we need to define nine reflection coefficients:

ambient reflection coefficients: ρ_{ar} , ρ_{ag} , and ρ_{ab} ,
 diffuse reflection coefficients: , ρ_{dr} , ρ_{dg} , and ρ_{db}
 specular reflection coefficients: , ρ_{sr} , ρ_{sg} , and ρ_{sb}

The ambient and diffuse reflection coefficients are based on the color of the surface itself. By “color” of a surface we mean the color that is reflected from it when the illumination is *white* light: a surface is red if it appears red when bathed in white light. If bathed in some other color it can exhibit an entirely different color. The following examples illustrate this.

Example 8.2.1. The color of an object. If we say that the color of a sphere is 30% red, 45% green, and 25% blue, it makes sense to set its ambient and diffuse reflection coefficients to $(0.3K, 0.45K, 0.25K)$, where K is some scaling value that determines the overall fraction of incident light that is reflected from the sphere. Now if it is bathed in white light having equal amounts of red, green, and blue ($I_r = I_g = I_b = I$) the individual diffuse components have intensities $I_r = 0.3 K I$, $I_g = 0.45 K I$, $I_b = 0.25 K I$, so as expected we see a color that is 30% red, 45% green, and 25% blue.

Example 8.2.2. A reddish object bathed in greenish light. Suppose a sphere has ambient and diffuse reflection coefficients (0.8, 0.2, 0.1), so it appears mostly red when bathed in white light. We illuminate it with a greenish light $\mathbf{I}_s = (0.15, 0.7, 0.15)$. The reflected light is then given by (0.12, 0.14, 0.015), which is a fairly even mix of red and green, and would appear yellowish (as we discuss further in Chapter 12).

The color of specular light. Because specular light is mirror-like, the color of the specular component is often the same as that of the light source. For instance, it is a matter of experience that the specular highlight seen on a glossy red apple when illuminated by a yellow light is yellow rather than red. This is also observed for shiny objects made of plastic-like material. To create specular highlights for a plastic surface the specular reflection coefficients, ρ_{sr} , ρ_{sg} , and ρ_{sh} used in Equation 8.7 are set to the same value, say ρ_s , so that the reflection coefficients are ‘gray’ in nature and do not alter the color of the incident light. The designer might choose $\rho_s = 0.5$ for a slightly shiny plastic surface, or $\rho_s = 0.9$ for a highly shiny surface.

Objects made of different materials.

A careful selection of reflection coefficients can make an object appear to be made of a specific material such as copper, gold, or pewter, at least approximately. McReynolds and Blythe [mcReynolds97] have suggested using the reflection coefficients given in Figure 8.17. Plate ??? shows several spheres modelled using these coefficients. The spheres do appear to be made of different materials. Note that the specular reflection coefficients have different red, green, and blue components, so the color of specular light is not simply that of the incident light. But McReynolds and Blythe caution users that, because OpenGL’s shading algorithm incorporates a Phong specular component, the visual effects are not completely realistic. We shall revisit the issue in Chapter 14 and describe the more realistic Cook-Torrance shading approach..

Material	ambient: $\rho_{ar}, \rho_{ag}, \rho_{ah}$	diffuse: $\rho_{dr}, \rho_{dg}, \rho_{dh}$	specular: $\rho_{sr}, \rho_{sg}, \rho_{sh}$	exponent: f
Black	0.0	0.01	0.50	
Plastic	0.0 0.0 0.0	0.01 0.01 0.01	0.50 0.50 0.50	32
Brass	0.329412 0.223529 0.027451	0.780392 0.568627 0.113725	0.992157 0.941176 0.807843	27.8974
Bronze	0.2125 0.1275 0.054	0.714 0.4284 0.18144	0.393548 0.271906 0.166721	25.6
Chrome	0.25 0.25 0.25	0.4 0.4 0.4	0.774597 0.774597 0.774597	76.8
Copper	0.19125 0.0735 0.0225	0.7038 0.27048 0.0828	0.256777 0.137622 0.086014	12.8
Gold	0.24725 0.1995 0.0745	0.75164 0.60648 0.22648	0.628281 0.555802 0.366065	51.2
Pewter	0.10588 0.058824 0.113725	0.427451 0.470588 0.541176	0.3333 0.3333 0.521569	9.84615
Silver	0.19225 0.19225 0.19225	0.50754 0.50754 0.50754	0.508273 0.508273 0.508273	51.2
Polished Silver	0.23125 0.23125 0.23125	0.2775 0.2775 0.2775	0.773911 0.773911 0.773911	89.6

Figure 8.17. Parameters for common materials [mcReynolds97].

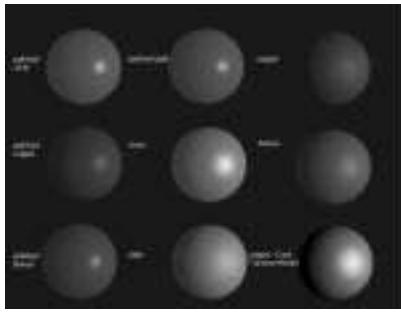


Plate 26. Shiny spheres made of different materials.

8.2.7. Shading and the Graphics pipeline.

At which step in the graphics pipeline is shading performed? And how is it done? Figure 8.18 shows the pipeline again. The key idea is that the vertices of a mesh are sent down the pipeline along with their associated vertex normals, and all shading calculations are done on *vertices*. (Recall that the `draw()` method in the `Mesh` class sends a vertex normal along with each vertex, as in Figure 6.15.)

Figure 8.18. The graphics pipeline revisited.

The figure shows a triangle with vertices v_0 , v_1 , and v_2 being rendered. Vertex v_i has the normal vector \mathbf{m}_i associated with it. These quantities are sent down the pipeline with calls such as:

```
glBegin(GL_POLYGON);
    for(int i = 0; i < 3; i++)
    {
        glNormal3f(norm[i].x, norm[i].y, norm[i].z);
        glVertex3f(pt[i].x, pt[i].y, pt[i].z);
    }
glEnd();
```

The call to `glNormal3f()` sets the “current normal vector”, which is applied to all vertices subsequently sent using `glVertex3f()`. It remains current until changed with another call to `glNormal3f()`. For this code example a new normal is associated with each vertex.

The vertices are transformed by the modelview matrix, M , effectively expressing them in camera (eye) coordinates. The normal vectors are also transformed, but vectors transform differently from points. As shown in Section 6.5.3, transforming points of a surface by a matrix M causes the normal \mathbf{m} at any point to become the normal $M^T \mathbf{m}$ on the transformed surface, where M^T is the transpose of the inverse of M . OpenGL automatically performs this calculation on normal vectors.

As we discuss in the next section OpenGL allows you to specify various light sources and their locations. Lights are objects too, and the light source positions are also transformed by the modelview matrix.

So all quantities end up after the modelview transformation being expressed in camera coordinates. At this point the shading model of Equation 8.7 is applied, and a color is “attached” to each vertex. The computation of this color requires knowledge of vectors \mathbf{m} , \mathbf{s} , and \mathbf{v} , but these are all available at this point in the pipeline. (Convince yourself of this).

Progressing farther down the pipeline, the pseudodepth term is created and the vertices are passed through the perspective transformation. The color information tags along with each vertex. The clipping step is performed in homogeneous coordinates as described earlier. This may alter some of the vertices. Figure 8.19 shows the case where vertex v_1 of the triangle is clipped off, and two new vertices, a and b , are created. The triangle becomes a quadrilateral. The color at each of the new vertices must be computed, since it is needed in the actual rendering step.

Figure 8.19. Clipping a polygon against the (warped) view volume.

The color at each new vertex is usually found by interpolation. For instance, suppose that the color at v_0 is (r_0, g_0, b_0) and the color at v_1 is (r_1, g_1, b_1) . If the point a is 40% of the way from v_0 to v_1 , the color associated with a is a blend of 60% of (r_0, g_0, b_0) and 40% of (r_1, g_1, b_1) . This is expressed as

$$\text{color at point } a = (\text{lerp}(r_0, r_1, 0.4), \text{lerp}(g_0, g_1, 0.4), \text{lerp}(b_0, b_1, 0.4)) \quad (8.8)$$

where we use the convenient function *lerp()* (short for “linear interpolation” - recall “tweening” in Section 4.5.3) defined by:

$$\text{lerp}(G, H, f) = G + (H - G)f \quad (8.9)$$

Its value lies at fraction f of the way from G to H .¹

The vertices are finally passed through the viewport transformation where they are mapped into screen coordinates (along with pseudodepth which now varies between 0 and 1). The quadrilateral is then rendered (with hidden surface removal), as suggested in Figure 8.19. We shall say much more about the actual rendering step.

8.2.8. Using Light Sources in OpenGL.

OpenGL provides a number of functions for setting up and using light sources, as well as for specifying the surface properties of materials. It can be daunting to absorb all of the many possible variations and details, so we describe the basics here. In this section we discuss how to establish different kinds of light sources in a scene. In the next section we look at ways to characterize the reflective properties of the surfaces of an object.

Creating a light source.

OpenGL allows you to define up to eight sources, which are referred to through names `GL_LIGHT0`, `GL_LIGHT1`, etc. Each source is invested with various properties, and must be enabled. Each property has a default value. For example, to create a source located at $(3, 6, 5)$ in world coordinates, use²:

```
GLfloat myLightPosition[] = {3.0, 6.0, 5.0, 1.0};
glLightfv(GL_LIGHT0, GL_POSITION, myLightPosition);
 glEnable(GL_LIGHTING); // enable
 glEnable(GL_LIGHT0); // enable this particular source
```

The array `myLightPosition[]` (use any name you wish for this array) specifies the location of the light source, and is passed to `glLightfv()` along with the name `GL_LIGHT0` to attach it to the particular source `GL_LIGHT0`.

Some sources, such as a desk lamp, are “in the scene”, whereas others, like the sun, are infinitely remote. OpenGL allows you to create both types by using homogeneous coordinates to specify light position:

- $(x, y, z, 1)$: a local light source at the position (x, y, z)
- $(x, y, z, 0)$: a vector to an infinitely remote light source in the direction (x, y, z)

Figure 8.20 shows a local source positioned at $(0, 3, 3, 1)$ and a remote source “located” along vector $(3, 3, 0, 0)$. Infinitely remote light sources are often called “**directional**”. There are computational advantages to using directional light sources, since the direction s in the calculations of diffuse and specular reflections is *constant* for all vertices in the scene. But directional light sources are not always the correct choice: some visual effects are properly achieved only when a light source is close to an object.

¹ In Section 8.5 we discuss replacing linear interpolation by “hyperbolic interpolation” as a more accurate way to form the colors at the new vertices formed by clipping.

² Here and elsewhere the type `float` would most likely serve as well as `GLfloat`. But using `GLfloat` makes your code more portable to other OpenGL environments.

Figure 8.20. A local source and an infinitely remote source.

You can also spell out different colors for a light source. OpenGL allows you to assign a different color to three “types of light” that a source emits: ambient, diffuse, and specular. It may seem strange to say that a source emits ambient light. It is still treated as in Equation 8.7: a global omnidirectional light that bathes the entire scene. The advantage of attaching it to a light source is that it can be turned on and off as an application proceeds. (OpenGL also offers a truly ambient light, not associated with any source, as we discuss later in connection with “lighting models”.)

Arrays are defined to hold the colors emitted by light sources, and they are passed to `glLightfv()`.

```
GLfloat amb0[] = {0.2, 0.4, 0.6, 1.0}; // define some colors
GLfloat diff0[] = {0.8, 0.9, 0.5, 1.0};
GLfloat spec0[] = {1.0, 0.8, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, amb0); // attach them to LIGHT0
glLightfv(GL_LIGHT0, GL_DIFFUSE, diff0);
glLightfv(GL_LIGHT0, GL_SPECULAR, spec0);
```

Colors are specified in so-called **RGBA** format, meaning red, green, blue, and “alpha”. The alpha value is sometimes used for blending two colors on the screen. We discuss it in Chapter 10. For our purposes here it is normally 1.0.

Light sources have various default values. For all sources:

default ambient = (0, 0, 0, 1); ← dimmest possible: black

For light source `LIGHT0`:

default diffuse = (1, 1, 1, 1); ← brightest possible: white
default specular = (1, 1, 1, 1); ← brightest possible: white

whereas for the other sources the diffuse and specular values have defaults of black.

Spotlights.

Light sources are *point sources* by default, meaning that they emit light uniformly in all directions. But OpenGL allows you to make them into spotlights, so they emit light in a restricted set of directions. Figure 8.21 shows a spotlight aimed in direction \mathbf{d} , with a “cutoff angle” of α . No light is seen at points lying outside the cutoff cone. For vertices such as P that lie inside the cone, the amount of light reaching P is attenuated by the factor $\cos^\epsilon(\beta)$ where β is the angle between \mathbf{d} and a line from the source to P , and ϵ is an exponent chosen by the user to give the desired fall-off of light with angle.

Figure 8.21. Properties of a spotlight.

The parameters for a spotlight are set using `glLightf()` to set a single value, and `glLightfv()` to set a vector:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0); // a cutoff angle of 45°
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 4.0); // ε = 4.0
GLfloat dir[] = {2.0, 1.0, -4.0}; // the spotlight's direction
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, dir);
```

The default values for these parameters are $\mathbf{d} = (0,0,-1)$, $\alpha = 180^\circ$, and $\epsilon = 0$, which makes a source an omni-directional point source.

Attenuation of light with distance.

OpenGL also allows you to specify how rapidly light diminishes with distance from a source. Although we have downplayed the importance of this dependence, it can be interesting to

experiment with different fall-off rates, and to fine tune a picture. OpenGL attenuates the strength of a positional³ light source by the following attenuation factor:

$$atten = \frac{1}{k_c + k_l D + k_q D^2} \quad (8.11)$$

where k_c , k_l , and k_q are coefficients and D is the distance between the light's position and the vertex in question. This expression is rich enough to allow you to model any combination of constant, linear, and quadratic (inverse square law) dependence on distance from a source. These parameters are controlled by function calls:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
```

and similarly for `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`. The default values are $k_c = 1$, $k_l = 0$, and $k_q = 0$, which eliminate any attenuation.

Lighting model.

OpenGL allows three parameters to be set that specify general rules for applying the shading model. These parameters are passed to variations of the function `glLightModel`.

- a). **The color of global ambient light.** You can establish a global ambient light source in a scene that is independent of any particular source. To create this light, specify its color using:

```
GLfloat amb[] = {0.2, 0.3, 0.1, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
```

This sets the ambient source to the color (0.2, 0.3, 0.1). The default value is (0.2, 0.2, 0.2, 1.0), so this ambient light is always present unless you purposely alter it. This makes objects in a scene visible even if you have not invoked any of the lighting functions.

- b). **Is the viewpoint local or remote?** OpenGL computes specular reflections using the “halfway vector” $\mathbf{h} = \mathbf{s} + \mathbf{v}$ described in Section 8.2.3. The true directions \mathbf{s} and \mathbf{v} are normally different at each vertex in a mesh (visualize this). If the light source is directional then \mathbf{s} is constant, but \mathbf{v} still varies from vertex to vertex. Rendering speed is increased if \mathbf{v} is made constant for all vertices. This is the default: OpenGL uses $\mathbf{v} = (0, 0, 1)$, which points along the positive z-axis in camera coordinates. You can force the pipeline to compute the true value of \mathbf{v} for each vertex by executing:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

- c). **Are both sides of a polygon shaded properly?** Each polygonal face in a model has two sides. When modeling we tend to think of them as the “inside” and “outside” surfaces. The convention is to list the vertices of a face in counter-clockwise (CCW) order as seen from outside the object. Most mesh objects represent solids that enclose space, so there is a well defined inside and outside. For such objects the camera can only see the outside surface of each face (assuming the camera is not inside the object!). With proper hidden surface removal the inside surface of each face is hidden from the eye by some closer face.

OpenGL has no notion of “inside” and “outside.” It can only distinguish between “front faces” and “back faces”. A face is a **front face** if its vertices are listed in counter-clockwise (CCW) order as seen by the eye⁴. Figure 8.22a shows the eye viewing a cube, which we presume was modeled using the CCW ordering convention. Arrows indicate the order in which the vertices of each face are passed to OpenGL (in a `glBegin(GL_POLYGON);...; glEnd()` block). For a space-enclosing object all faces that are visible to the eye are therefore front faces, and OpenGL draws them

³ This attenuation factor is disabled for directional light sources, since they are infinitely remote.

⁴ You can reverse this sense with `glFrontFace(GL_CW)`, which decrees that a face is a front face only if its vertices are listed in clock-wise order. The default is `glFrontFace(GL_CCW)`.

properly with the correct shading. OpenGL also draws the back faces⁵, but they are ultimately hidden by closer front faces.

a). b).

Figure 8.22. OpenGL's definition of a front face.

Things are different in part b, which shows a box with a face removed. Again arrows indicate the order in which vertices of a face are sent down the pipeline. Now three of the visible faces are back faces. By default OpenGL does not shade these properly. To coerce OpenGL to do proper shading of back faces, use:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

Then OpenGL reverses the normal vectors of any back-face so that they point toward the viewer, and it performs shading computations properly. Replace GL_TRUE with GL_FALSE (the default) to turn off this facility.

Note: Faces drawn by OpenGL do not cast shadows, so the back faces receive the same light from a source even though there may be some other face between them and the source.

Moving light sources.

Recall that light sources pass through the modelview matrix just as vertices do. Therefore lights can be repositioned by suitable uses of glRotated() and glTranslated(). The array position specified using glLightfv(GL_LIGHT0, GL_POSITION, position) is modified by the modelview matrix in effect at the time glLightfv() is called. So to modify the light position with transformations, and independently move the camera, imbed the light positioning command in a push/pop pair, as in:

```
void display()
{
    GLfloat position[] = {2, 1, 3, 1}; //initial light position

    <.. clear color and depth buffers ..>
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glPushMatrix();
        glRotated(...); // move the light
        glTranslated(...);
        glLightfv(GL_LIGHT0, GL_POSITION, position);
    glPopMatrix();

    gluLookAt(...); // set the camera position
    <.. draw the object ..>
    glutSwapBuffers();
}
```

On the other hand, to have the light move with the camera, use:

```
GLfloat pos[] = {0,0,0,1};
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0, GL_POSITION, position); // light at (0,0,0)
gluLookAt(...); // move the light and the camera
<.. draw the object ..>
```

This establishes the light to be positioned at the eye (like a minor's lamp), and the light moves with the camera.

8.2.9. Working with Material Properties in OpenGL.

⁵ You can improve performance by instructing OpenGL to skip rendering of back faces, with glCullFace(GL_BACK); glEnable(GL_CULL_FACE);

You can see the effect of a light source only when light reflects off an object's surface. OpenGL provides ways to specify the various reflection coefficients that appear in Equation 8.7. They are set with variations of the function `glMaterial`, and they can be specified individually for front faces and back faces (see the discussion concerning Figure 8.22). For instance,

```
GLfloat myDiffuse[] = {0.8, 0.2, 0.0, 1.0};
glMaterialfv(GL_FRONT, GL_DIFFUSE, myDiffuse);
```

sets the diffuse reflection coefficient (ρ_{dr} , ρ_{dg} , ρ_{db}) = (0.8, 0.2, 0.0) for all subsequently specified front faces. Reflection coefficients are specified as a 4-tuple in RBGA format, just like a color. The first parameter of `glMaterialfv()` can take on values:

`GL_FRONT`: set the reflection coefficient for front faces
`GL_BACK`: set it for back faces
`GL_FRONT_AND_BACK`: set it for both front and back faces

The second parameter can take on values:

`GL_AMBIENT`: set the ambient reflection coefficients
`GL_DIFFUSE`: set the diffuse reflection coefficients
`GL_SPECULAR`: set the specular reflection coefficients
`GL_AMBIENT_AND_DIFFUSE`: set both the ambient and diffuse reflection coefficients to the same values. This is for convenience, since the ambient and diffuse coefficients are so often chosen to be the same.
`GL_EMISSION`: set the emissive color of the surface.

The last choice sets the **emissive color** of a face, causing it to "glow" in the specified color, independent of any light source.

Putting it all together.

We now extend Equation 8.7 to include the additional contributions that OpenGL actually calculates. The total red component is given by:

$$I_r = e_r + I_{mr} \rho_{ar} + \sum_i atten_i \times spot_i \times (I^i_{ar} \rho_{ar} + I^i_{dr} \rho_{dr} \times lambert_i + I^i_{sr} \rho_{sr} \times phong_i^f) \quad (8.12)$$

Expressions for the green and blue components are similar. The emissive light is e_r , and I_{mr} is the global ambient light introduced in the lighting model. The summation denotes that the ambient, diffuse, and specular contributions of all light sources are summed. For the i -th source $atten_i$ is the attenuation factor as in Equation 8.10, $spot_i$ is the spotlight factor (see Figure 8.21), and $lambert_i$ and $phong$, are the familiar diffuse and specular dot products. All of these terms must be recalculated for each source.

Note: If I_r turns out to have a value larger than 1.0, OpenGL clamps it to 1.0: the brightest any light component can be is 1.0.

8.2.10. Shading of Scenes Specified by SDL.

The scene description language SDL introduced in Chapter 5 supports the loading of material properties into objects, so that they can be shaded properly. For instance,

```
light 3 4 5 .8 .8 .8 ! bright white light at (3, 4, 5)
background 1 1 1 ! white background
globalAmbient .2 .2 .2 ! a dark gray global ambient light
ambient .2 .6 0
diffuse .8 .2. 1 ! red material
specular 1 1 1 ! bright specular spots - the color of the source
exponent 20 !set the Phong exponent
scale 4 4 4 sphere
```

describes a scene containing a sphere with material properties (see Equation 8.7):

- ambient reflection coefficients: $(\rho_{ar}, \rho_{ag}, \rho_{ab}) = (.2, 0.6, 0)$,
- diffuse reflection coefficients: $(\rho_{dr}, \rho_{dg}, \rho_{db}) = (0.8, 0.2, 1.0)$,
- specular reflection coefficients: $(\rho_{sr}, \rho_{sg}, \rho_{sb}) = (1.0, 1.0, 1.0)$
- and Phong exponent $f = 20$.

The light source is given a color of (0.8, 0.8, 0.8) for both its diffuse and specular components. There is a global ambient term $(I_{ar}, I_{ag}, I_{ab}) = (0.2, 0.2, 0.2)$.

The current material properties are loaded into each object's `mtrl` field at the time it is created (see the end of `Scene :: getObject()` in `Shape.cpp` of Appendix 4). When an object draws itself using its `drawOpenGL()` method, it first passes its material properties to OpenGL (see `Shape::tellMaterialsGL()`), so that at the moment it is actually drawn OpenGL has these properties in its current state.

In Chapter 14 when raytracing we shall use each object's material field in a similar way to acquire the material properties and do proper shading.

8.3. Flat Shading and Smooth Shading.

Different objects require different shading effects. In Chapter 6 we modeled a variety of shapes using polygonal meshes. For some, like the barn or buckyball, we want to see the individual faces in a picture, but for others, like the sphere or chess pawn, we want to see the “underlying” surface that the faces approximate.

In the modeling process we attached a normal vector to each vertex of each face. If a certain face is to appear as a distinct polygon we attach the *same* normal vector to all of its vertices; the normal vector chosen is the normal direction to the plane of the face. On the other hand, if the face is supposed to approximate an underlying surface we attach to each vertex the normal to the underlying surface at that point.

We examine now how the normal vector information at each vertex is used to perform different kinds of shading. The main distinction is between a shading method that accentuates the individual polygons (flat shading) and a method that blends the faces to de-emphasize the edges between them (smooth shading). There are two kinds of smooth shading, called Gouraud and Phong shading, and we shall discuss both.

For both kinds of shading the vertices are passed down the graphics pipeline, shading calculations are performed to attach a color to each vertex, and ultimately the vertices of the face are converted to screen coordinates and the face is “painted” pixel by pixel with the appropriate color.

Painting a Face.

The face is colored using a polygon-fill routine. Filling a polygon is very simple, although fine tuning the fill algorithm for highest efficiency can get complex. (See Chapter 10.) Here we look at the basics, focusing on how the color of each pixel is set.

A polygon-fill routine is sometimes called a **tiler**, because it moves over the polygon pixel by pixel, coloring each pixel as appropriate, as one would lay down tiles on a parquet floor. Specifically, the pixels in a polygon are visited in a regular order, usually scan-line by scan-line from the bottom to the top of the polygon, and across each scan-line from left to right.

We assume here that the polygons of interest are *convex*. A tiler designed to fill only convex polygons can be made highly efficient, since at each scan-line there is a single unbroken “run” of pixels that lie inside the polygon. Most implementations of OpenGL exploit this and always fill convex polygons correctly, but do not guarantee to fill non-convex polygons properly. See the exercises for more thoughts on convexity.

Figure 8.23 shows an example where the face is a convex quadrilateral. The screen coordinates of each vertex are noted. The lowest and highest points on the face are y_{bot} and y_{top} , respectively. The tiler first fills in the row at $y = y_{bot}$ (in this case a single pixel), then the one at $y_{bot} + 1$, etc. At each scan-line, say y_s in the figure, there is a leftmost pixel, x_{left} , and a rightmost, x_{right} . The tiler moves from x_{left} to x_{right} , placing the desired color in each pixel. So the tiler is implemented as a simple double loop:

Figure 8.23. Filling a polygonal face with color.

```
for (int y = ybottom; y <= ytop; y++)           // for each scan-line
{
    <.. find xleft and xright ..>
    for (int x = xleft; x <= xright; x++) // fill across the scan-line
    {
        <.. find the color c for this pixel ..>
        <.. put c into the pixel at (x, y) ..>
    }
}
```

(We shall see later how hidden surface removal is easily accomplished within this double loop as well.) The principal difference between flat and smooth shading is the manner in which the color c is determined at each pixel.

8.3.1. Flat Shading.

When a face is flat (like the roof of a barn) and the light sources are quite distant the diffuse light component varies little over different points on the roof (the *lambert* term in Equation 8.6 is nearly the same at each vertex of the face). In such cases it is reasonable to use the same color for every pixel “covered” by the face. OpenGL offers a rendering mode in which the entire face is drawn with the same color. Although a color is passed down the pipeline as part of each vertex of the face, the painting algorithm uses only one of them (usually that of the first vertex in the face). So the command above, <find the color c for this pixel>, is not inside the loops but instead appears just prior to the loops , setting c to the color of one of the vertices. (Using the same color for every pixel tends to make flat shading quite fast.)

Flat shading is established in OpenGL using:

```
glShadeModel(GL_FLAT);
```

Figure 8.24 shows a buckyball and a sphere rendered using flat shading. The individual faces are clearly visible on both objects. The sphere is modeled as a smooth object, but no smoothing is taking place in the rendering, since the color of an entire face is set to that of only one vertex.

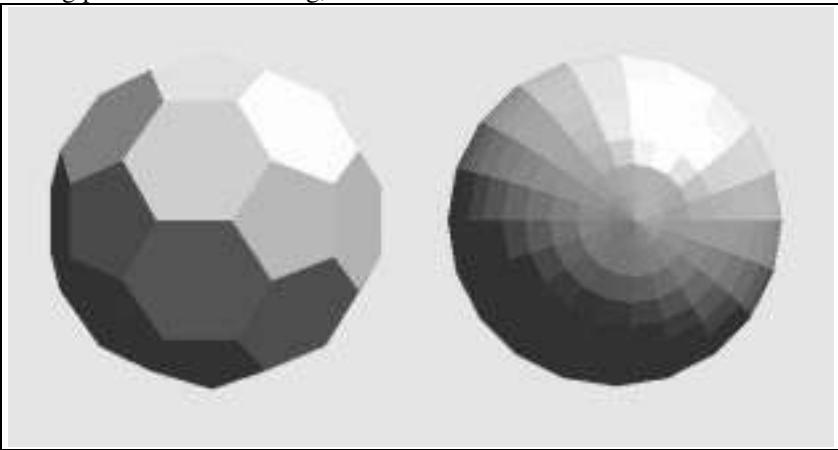


Figure 8.24. Two meshes rendered using flat shading.

Edges between faces actually appear more pronounced than they “are”, due to a phenomenon in the eye known as **lateral inhibition**, first described by Ernst Mach⁶. When there is a discontinuity in intensity across an object the eye manufactures a **Mach band** at the discontinuity, and a vivid edge

⁶ Ernst Mach (1838-1916), an Austrian physicist, whose early work strongly influenced the theory of relativity.

is seen (as discussed further in the exercises). This exaggerates the polygonal “look” of mesh objects rendered with flat shading.

Specular highlights are rendered poorly with flat shading, again because an entire face is filled with a color that was computed at only one vertex. If there happens to be a large specular component at the representative vertex, that brightness is drawn uniformly over the entire face. If a specular highlight doesn’t fall on the representative point, it is missed entirely. For this reason, there is little incentive for including the specular reflection component in the shading computation.

8.3.2. Smooth Shading.

Smooth shading attempts to de-emphasize edges between faces by computing colors at more points on each face. There are two principal types of smooth shading, called Gouraud shading and Phong shading [gouraud71, phong75]. OpenGL does only Gouraud shading, but we describe both of them.

Gouraud shading computes a different value of c for each pixel. For the scanline at y_s (in figure 8.23) it finds the color at the leftmost pixel, $color_{left}$ by linear interpolation of the colors at the top and bottom of the left edge⁷. For the scan-line at y_s the color at the top is $color_4$ and that at the bottom is $color_1$, so $color_{left}$ would be calculated as (recall Equation 8.9):

$$color_{left} = lerp(color_1, color_4, f) \quad (8.13)$$

where fraction f , given by

$$f = \frac{y_s - y_{bott}}{y_4 - y_{bott}}$$

varies between 0 and 1 as y_s varies from y_{bott} to y_4 . Note that Equation 8.13 involves three calculations since each color quantity has a red, green, and blue component.

Similarly $color_{right}$ is found by interpolating the colors at the top and bottom of the right edge. The tiler then fills across the scanline, linearly interpolating between $color_{left}$ and $color_{right}$ to obtain the color at pixel x :

$$c(x) = lerp(color_{left}, color_{right}, \frac{x - x_{left}}{x_{right} - x_{left}}) \quad (8.14)$$

To increase efficiency this color is computed incrementally at each pixel. That is, there is a constant difference between $c(x+1)$ and $c(x)$, so

$$c(x+1) = c(x) + \frac{color_{right} - color_{left}}{x_{right} - x_{left}} \quad (8.15)$$

The increment is calculated only once outside of the innermost loop. In terms of code this looks like:

```
for (int y = ybott; y <= ytop; y++) // for each scan-line
{
    <.. find xleft and xright ..>
    <.. find colorleft and colorright ..>
    colorinc = (colorright - colorleft) / (xright - xleft);
    for (int x = xleft, c = colorleft; x <= xright; x++, c+=colorinc)
        <.. put c into the pixel at (x, y) ..>
}
```

⁷ We shall see later that, although colors are usually interpolated *linearly* as we do here, better results can be obtained by using so-called *hyperbolic interpolation*. For Gouraud shading the distinction is minor; for texture mapping it is crucial.

Gouraud shading is modestly more expensive computationally than flat shading. Gouraud shading is established in OpenGL using:

```
glShadeModel(GL_SMOOTH);
```

Figure 8.25 shows a buckyball and a sphere rendered using Gouraud shading. The buckyball looks the same as when it was flat shaded in Figure 8.24, because the same color is associated with each vertex of a face, so interpolation changes nothing. But the sphere looks much smoother. There are no abrupt jumps in color between neighboring faces. The edges of the faces (and the Mach bands) are gone, replaced by a smoothly varying color across the object. Along the silhouette, however, you can still see the bounding edges of individual faces.

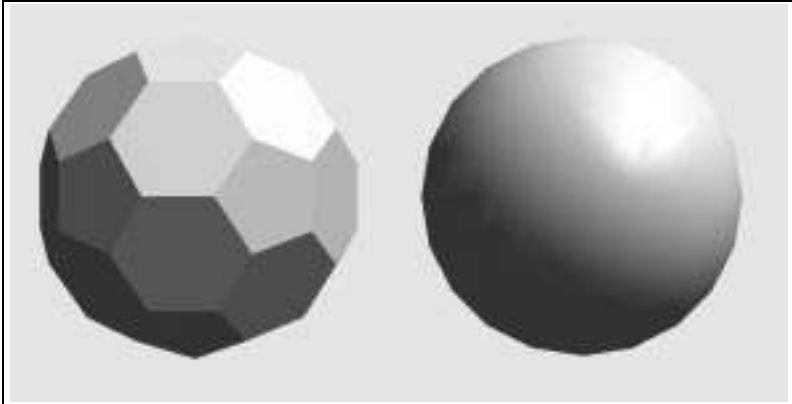


Figure 8.25. Two meshes rendered using smooth shading. (file: fig8.25.bmp)

Why do the edges disappear with this technique? Figure 8.26a shows two faces, F and F' , that share an edge. When rendering F the colors c_L and c_R are used, and when rendering F' the colors c'_L and c'_R are used. But since c_R equals c'_L there is an abrupt change in color at the edge along the scanline.

a). two faces abutting b). cross section: can see underlying surface

Figure 8.26. Continuity of color across a polygon edge.

Figure 8.26b suggests how this technique reveals the “underlying” surface approximated by the mesh. The polygonal surface is shown in cross section, with vertices V_1 , V_2 , etc. marked. The imaginary smooth surface that the mesh supposedly represents is suggested as well. Properly computed vertex normals \mathbf{m}_1 , \mathbf{m}_2 , etc. point perpendicularly to this imaginary surface, so the normal for “correct” shading is being used at each vertex, and the color thereby found is correct. The color is then made to vary smoothly between vertices, not following any physical law but rather a simple mathematical one.

Because colors are formed by interpolating rather than computing colors at every pixel, Gouraud shading does not picture highlights well. Therefore, when Gouraud shading is used, one normally suppresses the specular component of intensity in Equation 8.12. Highlights are better reproduced using Phong shading, discussed next.

Phong Shading.

Greater realism can be achieved - particularly with regard to highlights on shiny objects - by a better approximation of the normal vector to the face at each pixel. This type of shading is called **Phong shading**, after its inventor Phong Bui-tuong [phong75].

When computing Phong shading we find the normal vector *at each point* on the face and we apply the shading model there to find the color. We compute the normal vector at each pixel by interpolating the normal vectors at the vertices of the polygon.

Figure 8.27 shows a projected face, with the normal vectors \mathbf{m}_1 , \mathbf{m}_2 , \mathbf{m}_3 , and \mathbf{m}_4 indicated at the four vertices. For the scan-line y_s as shown the vectors \mathbf{m}_{left} and $\mathbf{m}_{\text{right}}$ are found by linear interpolation. For instance, \mathbf{m}_{left} is found as

Figure 8.27. Interpolating normals.

$$\mathbf{m}_{\text{left}} = \text{lerp}(\mathbf{m}_4, \mathbf{m}_3, \frac{y_s - y_4}{y_3 - y_4})$$

This interpolated vector must be normalized to unit length before its use in the shading formula. Once \mathbf{m}_{left} and $\mathbf{m}_{\text{right}}$ are known, they are interpolated to form a normal vector at each x along the scan-line. This vector, once normalized, is used in the shading calculation to form the color at that pixel.

Figure 8.28 shows an object rendered using Gouraud shading and Phong shading. Because the direction of the normal vector varies smoothly from point to point and more closely approximates that of an underlying smooth surface, the production of specular highlights is much more faithful than with Gouraud shading, and more realistic renderings are produced.

1st Ed. Figure 15.25

Figure 8.28. Comparison of Gouraud and Phong shading (Courtesy of Bishop and Weimar 1986).

The principal drawback of Phong shading is its speed: a great deal more computation is required per pixel, so that Phong shading can take 6 to 8 times longer than Gouraud shading to perform. A number of approaches have been taken to speed up the process [bishop86, claussen90].

OpenGL is not set up to do Phong shading, since it applies the shading model once per vertex right after the modelview transformation, and normal vector information is not passed to the rendering stage following the perspective transformation and perspective divide. We will see in Section 8.5, however, that an approximation to Phong shading can be created by mapping a “highlight” texture onto an object using the environment mapping technique.

Practice Exercises.

8.3.1. Filling your face. Fill in details of how the polygon fill algorithm operates for the polygon with vertices $(x, y) = (23, 137), (120, 204), (200, 100), (100, 25)$, for scan lines $y = 136, y = 137$, and $y = 138$. Specifically write the values of x_{left} and x_{right} in each case.

8.3.2. Clipped convex polygons are still convex. Develop a proof that if a convex polygon is clipped against the camera’s view volume, the clipped polygon is still convex.

8.3.3. Retaining edges with Gouraud Shading. In some cases we may want to show specific creases and edges in the model. Discuss how this can be controlled by the choice of the vertex normal vectors. For instance, to retain the edge between faces F and F' in Figure 8.26, what should the vertex normals be? Other tricks and issues can be found in the references [e.g. Rogers85].

8.3.4. Faster Phong shading with fence shading. To increase the speed of Phong shading Behrens [behrens94] suggests interpolating normal vectors between vertices to get \mathbf{m}_L and \mathbf{m}_R in the usual way at each scan line, but then computing colors only at these left and right pixels, interpolating them along a scan line as in Gouraud shading. This so-called “fence shading” speeds up rendering dramatically, but does less well in rendering highlights than true Phong shading. Describe general directions for the vertex normals $\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3$, and \mathbf{m}_4 in Figure 8.27 such that

- a). Fence shading produces the same highlights as Phong shading;
- b). Fence shading produces very different highlights than does Phong shading.

8.3.5. The Phong shading algorithm. Make the necessary changes to the tiling code to incorporate Phong shading. Assume the vertex normal vectors are available for each face. Also discuss how Phong shading can be approximated by OpenGL’s smooth shading algorithm. Hint: increase the number of faces in the model.

8.4. Adding Hidden Surface Removal

It is very simple to incorporate hidden surface removal in the rendering process above if enough memory is available to have a “depth buffer” (also called a “z-buffer”). Because it fits so easily into the rendering mechanisms we are discussing, we include it here. Other (more efficient and less memory-hungry) hidden surface removal algorithms are described in Chapter 13.

8.4.1. The Depth Buffer Approach.

The depth buffer (or z-buffer) algorithm is one of the simplest and most easily implemented hidden surface removal methods. Its principal limitations are that it requires a large amount of memory,

and that it often renders an object that is later obscured by a nearer object (so time spent rendering the first object is wasted).

Figure 8.29 shows a depth buffer associated with the frame buffer. For every pixel $p[i][j]$ on the display the depth buffer stores a b bit quantity $d[i][j]$. The value of b is usually in the range of 12 to 30 bits.



Figure 8.29. Conceptual view of the depth buffer.

During the rendering process the depth buffer value $d[i][j]$ contains the pseudodepth of the closest object encountered (so far) at that pixel. As the tiler proceeds pixel by pixel across a scan-line filling the current face, it tests whether the pseudodepth of the current face is less than the depth $d[i][j]$ stored in the depth buffer at that point. If so the color of the closer surface replaces the color $p[i][j]$ and this smaller pseudodepth replaces the old value in $d[i][j]$. Faces can be drawn in any order. If a remote face is drawn first some of the pixels that show the face will later be replaced by the colors of a nearer face. The time spent rendering the more remote face is therefore wasted. Note that this algorithm works for objects of any shape including curved surfaces, because it finds the closest surface based on a point-by-point test.

The array $d[][]$ is initially loaded with value 1.0, the greatest pseudodepth value possible. The frame buffer is initially loaded with the background color.

Finding the pseudodepth at each pixel.

We need a rapid way to compute the pseudodepth at each pixel. Recall that each vertex $P = (P_x, P_y, P_z)$ of a face is sent down the graphics pipeline, and passes through various transformations. The information available for each vertex after the viewport transformation is the 3-tuple that is a scaled and shifted version of (see Equation 7.2)

$$(x, y, z) = \left(\frac{P_x}{-P_z}, \frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z} \right)$$

The third component is pseudodepth. Constants a and b have been chosen so that the third component equals 0 if P lies in the near plane, and 1 if P lies in the far plane. For highest efficiency we would like to compute it at each pixel incrementally, which implies using linear interpolation as we did for color in Equation 8.15.

Figure 8.30 shows a face being filled along scanline y . The pseudodepth values at various points are marked. The pseudodepths d_1, d_2, d_3 , and d_4 at the vertices are known. We want to calculate d_{left} at scan-line y_s as $\text{lerp}(d_1, d_4, f)$ for fraction $f = (y_s - y_1)/(y_4 - y_1)$, and similarly d_{right} as $\text{lerp}(d_2, d_3, h)$ for the appropriate h . And we want to find the pseudodepth d at each pixel (x, y) along the scan-line as $\text{lerp}(d_{\text{left}}, d_{\text{right}}, k)$ for the appropriate k . (What are the values of h and k ?) The question is whether this calculation produces the “true” pseudodepth of the corresponding point on the 3D face.



Figure 8.30. Incremental computation of pseudodepth.

The answer is that it works correctly. We prove this later after developing some additional algebraic artillery, but the key idea is that the original 3D face is flat, and perspective projection preserves flatness, so pseudodepth varies linearly with the projected x and y coordinates. (See Exercise 8.5.2.)

Figure 8.31 shows the nearly trivial additions to the Gouraud shading tiling algorithm that accomplish hidden surface removal. Values of d_{left} and d_{right} are found (incrementally) for each scan-line, along with d_{inc} which is used in the innermost loop. For each pixel d is found, a single comparison is made, and an update of $d[i][j]$ is made if the current face is found to be closest.

```
for (int y = Ybott; y <= Ytop; y++) // for each scan-line
{
    <.. find xleft and xright ..>
    <.. find dleft and dright, and dinc ..>
    <.. find colorleft and colorright, and colorinc ..>
```

```

for (int x = xleft, c = colorleft, d = dleft; x <= xright; x++, c+=colorinc, d+= dinc)
{
    if(d < d[x][y])
    {
        <.. put c into the pixel at (x, y) ..
        d[x][y] = d; // update the closest depth
    }
}

```

Figure 8.31. Doing depth computations incrementally.

Depth compression at greater distances.

Recall from Example 7.4.4 that the pseudodepth of a point does not vary linearly with actual depth from the eye, but instead approaches an asymptote. This means that small changes in true depth map into extremely small changes in pseudodepth when the depth is large. Since only a limited number of bits are used to represent pseudodepth, two nearby values can easily map into the same value, which can lead to errors in the comparison $d < d[x][y]$. Using a larger number of bits to represent pseudodepth helps, but this requires more memory. It helps a little to place the near plane as far away from the eye as possible.

OpenGL supports a depth buffer, and uses the algorithm described above to do hidden surface removal. You must instruct OpenGL to create a depth buffer when it initializes the display mode:

```
glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB);
```

and enable depth testing with

```
 glEnable(GL_DEPTH_TEST);
```

Then each time a new picture is to be created the depth buffer must be initialized using:

```
 glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT); // clear screen
```

Practice Exercises.

8.4.1. The increments. Fill in details of how d_{left} , d_{right} , and d are found from the pseudodepth values known at the polygon vertices.

8.4.2. Coding depth values. Suppose b bits are allocated for each element in the depth buffer.

These b bits must record values of pseudodepth between 0 and 1. A value between 0 and 1 can be expressed in binary in the form $.d_1d_2d_3\dots d_b$ where d_i is 0 or 1. For instance, a pseudodepth of 0.75 would be coded as .1100000000... Is this a good use of the b bits? Discuss alternatives.

8.4.3. Reducing the Size of the Depth Buffer. If there is not enough memory to implement a full depth buffer, one can generate the picture in pieces. A depth buffer is established for only a fraction of the scan lines, and the algorithm is repeated for each fraction. For instance, in a 512-by-512 display, one can allocate memory for a depth buffer of only 64 scan lines and do the algorithm eight times. Each time the entire face list is scanned, depths are computed for faces covering the scan lines involved, and comparisons are made with the reigning depths so far. Having to scan the face list eight times, of course, makes the algorithm operate more slowly. Suppose that a scene involves F faces, and each face covers on the average L scanlines. Estimate how much more time it takes to use the depth buffer method when memory is allocated for only $nRows/N$ scanlines.

8.4.4. A single scanline depth buffer. The fragmentation of the frame buffer of the previous exercise can be taken to the extreme where the depth buffer records depths for only *one* scan line. It appears to require more computation, as each face is “brought in fresh” to the process many times, once for each scan line. Discuss how the algorithm is modified for this case, and estimate how much longer it takes to perform than when a full-screen depth buffer is used.

8.5. Adding Texture to Faces.

I found Rome a city of bricks and left it a city of marble.
Augustus Caesar, from SUETONIUS

The realism of an image is greatly enhanced by adding surface texture to the various faces of a mesh object. Figure 8.32 shows some examples. In part a) images have been “pasted onto” each of the faces of a box. In part b) a label has been wrapped around a cylindrical can, and the wall behind the can appears to be made of bricks. In part c) a table has a wood-grain surface, and the

floor is tiled with decorative tiles. The picture on the wall contains an image pasted inside the frame.

a). box b). beer can c). wood table – screen shots

Figure 8.32. Examples of texture mapped onto surfaces.

The basic technique begins with some texture function in “**texture space**” such as that shown in Figure 8.33a. Texture space is traditionally marked off by parameters named s and t . The texture is a function $\text{texture}(s, t)$ which produces a color or intensity value for each value of s and t between 0 and 1.

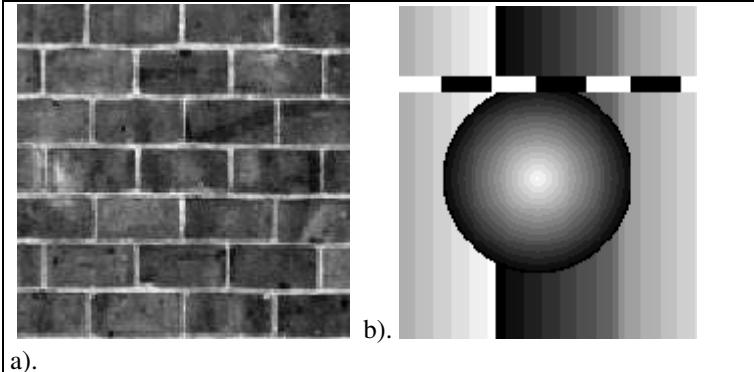


Figure 8.33. Examples of textures. a). image texture, b). procedural texture.

There are numerous sources of textures. The most common are bitmaps and computed functions.

- **Bitmap textures.**

Textures are often formed from bitmap representations of images (such as a digitized photo, clip art, or an image computed previously in some program). Such a texture consists of an array, say $\text{txtr}[c][r]$, of color values (often called **texels**). If the array has C columns and R rows, the indices c and r vary from 0 to $C-1$ and $R-1$, respectively. In the simplest case the function $\text{texture}(s, t)$ provides “samples” into this array as in

```
Color3 texture(float s, float t)
{
    return txtr[(int)(s * C)][(int)(t * R)];
```

where `Color3` holds an RGB triple. For example, if $R = 400$ and $C = 600$, then $\text{texture}(0.261, 0.783)$ evaluates to $\text{txtr}[156][313]$. Note that a variation of s from 0 to 1 encompasses 600 pixels, whereas the same variation in t encompasses 400 pixels. To avoid distortion during rendering this texture must be mapped onto a rectangle with aspect ratio 6/4.

- **Procedural textures.**

Alternatively we can define a texture by a mathematical function or procedure. For instance, the “sphere” shape that appears in Figure 8.33b could be generated by the function

```
float fakeSphere(float s, float t)
{
    float r = sqrt((s-0.5)*(s-0.5)+(t-0.5)*(t-0.5));
    if(r < 0.3) return 1 - r/0.3; // sphere intensity
    else return 0.2; // dark background
}
```

that varies from 1 (white) at the center to 0 (black) at the edges of the apparent sphere. Another example that mimics a checkerboard is examined in the exercises. Anything that can be computed can provide a texture: smooth blends and swirls of color, the Mandelbrot set, wireframe drawings of solids, etc.

We see later that the value $texture(s, t)$ can be used in a variety of ways: it can be used as the color of the face itself as if the face is “glowing”; it can be used as a reflection coefficient to “modulate” the amount of light reflected from the face; it can be used to alter the normal vector to the surface to give it a “bumpy” appearance.

Practice Exercise 8.5.1. The classic checkerboard texture. Figure 8.34 shows a checkerboard consisting of 4 by 5 squares with brightness levels that alternate between 0 (for black) and 1 (for white).

- Write the function `float texture(float s, float t)` for this texture. (See also Exercise 2.3.1.)
- Write `texture()` for the case where there are M rows and N columns in the checkerboard.
- Repeat part b for the case where the checkerboard is rotated 40° relative to the s and t axes.

Figure 8.34. A classic checkerboard pattern.

With a texture function in hand, the next step is to map it properly onto the desired surface, and then to view it with a camera. Figure 8.35 shows an example that illustrates the overall problem. Here a single example of texture is mapped onto three different objects: a planar polygon, a cylinder, and a sphere. For each object there is some transformation, say T_{tw} (for “texture to world”) that maps texture (s, t) values to points (x, y, z) on the object’s surface. The camera takes a snapshot of the scene from some angle, producing the view shown. We call the transformation from points in 3D to points on the screen T_{ws} (“from world to screen”), so a point (x, y, z) on a surface is “seen” at pixel location $(sx, sy) = T_{ws}(x, y, z)$. So overall, the value (s^*, t^*) on the texture finally arrives at pixel $(sx, sy) = T_{ws}(T_{tw}(s^*, t^*))$.

Figure 8.35. Drawing texture on several object shapes.

The rendering process actually goes the other way: for each pixel at (sx, sy) there is a sequence of questions:

- What is the closest surface “seen” at (sx, sy) ? This determines which texture is relevant.
- To what point (x, y, z) on this surface does (sx, sy) correspond?
- To which texture coordinate pair (s, t) does this point (x, y, z) correspond?

So we need the inverse transformation, something like $(s, t) = T_{tw}^{-1}(T_{ws}^{-1}(sx, sy))$, that reports (s, t) coordinates given pixel coordinates. This inverse transformation can be hard to obtain or easy to obtain, depending on the surface shapes.

8.5.1. Pasting the Texture onto a Flat Surface.

We first examine the most important case: mapping texture onto a flat surface. This is a modeling task. In Section 8.5.2 we tackle the viewing task to see how the texture is actually rendered. We then discuss mapping textures onto more complicated surface shapes.

Pasting Texture onto a Flat Face.

Since texture space itself is flat, it is simplest to paste texture onto a flat surface. Figure 8.36 shows a texture image mapped to a portion of a planar polygon F . We must specify how to associate points on the texture with points on F . In OpenGL we associate a point in texture space $P_i = (s_i, t_i)$ with each vertex V_i of the face using the function `glTexCoord2f()`. The function `glTexCoord2f(s, t)` sets the “current texture coordinates” to (s, t) , and they are attached to subsequently defined vertices. Normally each call to `glVertex3f()` is preceded by a call to `glTexCoord2f()`, so each vertex “gets” a new pair of texture coordinates. For example, to define a quadrilateral face and to “position” a texture on it, we send OpenGL four texture coordinates and the four 3D points, as in:

Figure 8.36. Mapping texture onto a planar polygon.

```
glBegin(GL_QUADS); // define a quadrilateral face
```

```

glTexCoord2f(0.0, 0.0); glVertex3f(1.0, 2.5, 1.5);
glTexCoord2f(0.0, 0.6); glVertex3f(1.0, 3.7, 1.5);
glTexCoord2f(0.8, 0.6); glVertex3f(2.0, 3.7, 1.5);
glTexCoord2f(0.8, 0.0); glVertex3f(2.0, 2.5, 1.5);
glEnd();

```

Attaching a P_i to each V_i is equivalent to prescribing a polygon P in texture space that has the same number of vertices as F . Usually P has the same shape as F as well: then the portion of the texture that lies inside P is pasted without distortion onto the whole of F . When P and F have the same shape the mapping is clearly affine: it is a scaling, possibly accompanied by a rotation and a translation.

Figure 8.37 shows the very common case where the four corners of the texture square are associated with the four corners of a rectangle. (The texture coordinates (s, t) associated with each corner are noted on the 3D face.) In this example the texture is a 640 by 480 pixel bitmap, and it is pasted onto a rectangle with aspect ratio 640/480, so it appears without distortion. (Note that the texture coordinates s and t still vary from 0 to 1.) Figure 8.38 shows the use of texture coordinates that “tile” the texture, making it repeat. To do this some texture coordinates that lie outside of the interval $[0,1]$ are used. When the renderer encounters a value of s and t outside of the unit square such as $s = 2.67$ it ignores the integral part and uses only the fractional part 0.67. Thus the point on a face that requires $(s, t) = (2.6, 3.77)$ is textured with $texture(0.6, 0.77)$. By default OpenGL tiles texture this way. It may be set to “clamp” texture values instead, if desired; see the exercises.

Figure 8.37. Mapping a square to a rectangle.

Figure 8.38. Producing repeated textures.

Thus a coordinate pair (s, t) is sent down the pipeline along with each vertex of the face. As we describe in the next section, the notion is that points inside F will be filled with texture values lying inside P , by finding the internal coordinate values (s, t) using interpolation. This interpolation process is described in the next section.

Adding texture coordinates to Mesh objects.

Recall from Figure 6.13 that a mesh object has three lists: the vertex, normal vector, and face lists. We must add to this a “texture coordinate” list, that stores the coordinates (s_i, t_i) to be associated with various vertices. We can add an array of elements of the type:

```
class TxtrCoord{public: float s, t;};
```

to hold all of the coordinate pairs of interest for the mesh. There are several different ways to treat texture for an object, and each has implications for how texture information is organized in the model. The two most important are:

1. The mesh object consists of a small number of flat faces, and a different texture is to be applied to each. Here each face has only a single normal vector but its own list of texture coordinates. So the data associated with each face would be:
 - the number of vertices in the face;
 - the index of the normal vector to the face;
 - a list of indices of the vertices;
 - a list of indices of the texture coordinates;
2. The mesh represents a smooth underlying object, and a single texture is to be “wrapped” around it (or a portion of it). Here each vertex has associated with it a specific normal vector and a particular texture coordinate pair. A single index into the vertex/normals/texture lists is used for each vertex. The data associated with each face would then be:
 - the number of vertices in the face;

- a list of indices of the vertices;

The exercises take a further look at the required data structures for these types of meshes.

8.5.2. Rendering the Texture.

Rendering texture in a face F is similar to Gouraud shading: the renderer moves across the face pixel by pixel. For each pixel it must determine the corresponding texture coordinates (s, t) , access the texture, and set the pixel to the proper texture color. We shall see that finding the coordinates (s, t) must be done very carefully.

Figure 8.39 shows the camera taking a snapshot of face F with texture pasted onto it, and the rendering in progress. Scanline y is being filled from x_{left} to x_{right} . For each x along this scanline we must compute the correct position (shown as $P(x, y)$) on the face, and from this obtain the correct position (s^*, t^*) within the texture.



Figure 8.39. Rendering the face in a camera snapshot.

Having set up the texture to object mapping, we know the texture coordinates at each of the vertices of F , as suggested in Figure 8.40. The natural thing is to compute $(s_{\text{left}}, t_{\text{left}})$ and $(s_{\text{right}}, t_{\text{right}})$ for each scanline in a rapid incremental fashion and to interpolate between these values moving across the scanline. But we must be careful: simple increments from s_{left} to s_{right} as we march across scanline y from x_{left} to x_{right} won't work, since equal steps across a projected face do *not* correspond to equal steps across the 3D face.



Figure 8.40. Incremental calculation of texture coordinates.

Figure 8.41 illustrates the problem. Part a shows face F viewed so that its left edge is closer to the viewer than its right edge. Part b shows the projection F' of this face on the screen. At scan-line $y = 170$ we mark points equally spaced across F' , suggesting the positions of successive pixels on the face. The corresponding positions of these marks on the actual face are shown in part a. They are seen to be more closely spaced at the farther end of F . This is simply the effect of perspective foreshortening.



Figure 8.41. Spacing of samples with linear interpolation.

If we use simple linear interpolation and take equally spaced steps in s and t to compute texture coordinates, we "sample" into the texture at the wrong spots, and a distorted image results. Figure 8.42 shows what happens with a simple checkerboard texture mapped onto a rectangle. Linear interpolation is used in part a, producing palpable distortion in the texture. This distortion is particularly disturbing in an animation where the polygon is rotating, as the texture appears to warp and stretch dynamically. Correct interpolation is used in part b, and the checkerboard looks as it should. In an animation this texture would appear to be firmly attached to the moving or rotating face.

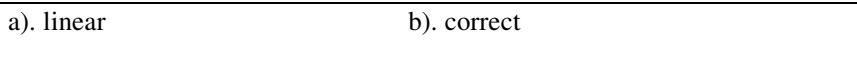


Figure 8.42. Images formed using linear interpolation and correct interpolation.

Several approaches have appeared in the literature that develop the proper interpolation method. Heckbert and Moreton [heckbert91] and Blinn [blinn92] describe an elegant development based on the general nature of affine and projective mappings. Segal et al [segal92] arrive at the same result using a more algebraic derivation based on the parametric representation for a line segment. We follow the latter approach here.

Figure 8.43 shows the situation to be analyzed. We know that affine and projective transformations preserve straightness, so line L_e in eye space projects to line L_s in screen space, and similarly the texels we wish to draw on line L_s lie along the line L_t in texture space that maps

to L_c . The key question is this: if we move in equal steps across L_s on the screen how should we step across texels along L_t in texture space?

Figure 8.43. Lines in one space map to lines in another.

We develop a general result next that summarizes how interpolation works: it all has to do with the effect of perspective division. Then we relate the general result to the transformations performed in the graphics pipeline, and see precisely where extra steps must be taken to do proper mapping of texture.

Figure 8.44 shows the line AB in 3D being transformed into the line ab in 3D by matrix M . (M might represent an affine transformation, or a more general perspective transformation.) A maps to a , B maps to b . Consider the point $R(g)$ that lies fraction g of the way between A and B . It maps to some point $r(f)$ that lies fraction f of the way from a to b . The fractions f and g are *not* the same as we shall see. The question is, as f varies from 0 to 1 how exactly does g vary? That is, how does motion along ab correspond to motion along AB ?

Figure 8.44. How does motion along corresponding lines operate?

Deriving how g and f are related.

We denote the homogeneous coordinate version of a by \tilde{a} , and name its components $\tilde{a} = (a_1, a_2, a_3, a_4)$. (We use subscripts 1, 2, 3, and 4 instead of x , y , etc. to prevent ambiguity, since there are so many different “ x , y , z ” spaces.) So point a is found from \tilde{a} by perspective division: $a = \left(\frac{a_1}{a_4}, \frac{a_2}{a_4}, \frac{a_3}{a_4} \right)$. Since M maps $A = (A_1, A_2, A_3)$ to a we know $\tilde{a} = M(A, 1)^T$ where $(A, 1)^T$ is the column vector with components A_1, A_2, A_3 , and 1. Similarly, $\tilde{b} = M(B, 1)^T$. (Check each of these relations carefully.) Now using *lerp()* notation to keep things succinct, we have defined $R(g) = \text{lerp}(A, B, g)$, which maps to $M(\text{lerp}(A, B, g), 1)^T = \text{lerp}(\tilde{a}, \tilde{b}, g)$ $= (\text{lerp}(a_1, b_1, g), \text{lerp}(a_2, b_2, g), \text{lerp}(a_3, b_3, g), \text{lerp}(a_4, b_4, g))$. (Check these, too.) This is the homogeneous coordinate version $\tilde{r}(f)$ of the point $r(f)$. We recover the actual components of $r(f)$ by perspective division. For simplicity write just the first component $r_1(f)$, which is:

$$r_1(f) = \frac{\text{lerp}(a_1, b_1, g)}{\text{lerp}(a_4, b_4, g)} \quad (8.16)$$

But since by definition $r(f) = \text{lerp}(a, b, f)$ we have another expression for the first component $r_1(f)$:

$$r_1(f) = \text{lerp}\left(\frac{a_1}{a_4}, \frac{b_1}{b_4}, f\right) \quad (8.17)$$

Expressions (what are they?) for $r_2(f)$ and $r_3(f)$ follow similarly. Equate these two versions of $r_1(f)$ and do a little algebra to obtain the desired relationship between f and g :

$$g = \frac{f}{\text{lerp}\left(\frac{b_4}{a_4}, 1, f\right)} \quad (8.18)$$

Therefore the point $R(g)$ maps to $r(f)$, but g and f aren't the same fraction. g matches at $f=0$ and at $f=1$, but its growth with f is tempered by a denominator that depends on the ratio b_4/a_4 . If a_4 equals b_4 then g is identical to f (check this). Figure 8.45 shows how g varies with f , for different values of b_4/a_4 .

g vs f

Figure 8.45. How g depends on f .

We can go the final step and show where the point $R(g)$ is on the 3D face that maps into $r(f)$. Simply use Equation 8.17 in $R(g) = A(1-g)+Bg$ and simplify algebraically (check this out) to obtain for the first component:

$$R_1 = \frac{\text{lerp}\left(\frac{A_1}{a_4}, \frac{B_1}{b_4}, f\right)}{\text{lerp}\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)} \quad (8.19)$$

with similar expressions resulting for the components R_2 and R_3 (which have the *same* denominator as R_1). This is a key result. It tells which 3D point (R_1, R_2, R_3) corresponds (in eye coordinates) to a given point that lies (fraction f of the way) between two given points a and b in screen coordinates. So any quantity (such as texture) that is “attached” to vertices of the 3D face and varies linearly between them will behave the same way.

The two cases of interest for the transformation with matrix M are:

- The transformation is affine;
- The transformation is the perspective transformation.

a). When the transformation is affine then a_4 and b_4 are both 1 (why?), so the formulas above simplify immediately. The fractions f and g become identical, and R_1 above becomes $\text{lerp}(A_1, B_1, f)$. We can summarize this as:

Fact: If M is *affine*, equal steps along the line ab do correspond to equal steps along the line AB .

b). When M represents the perspective transformation from eye coordinates to clip coordinates the fourth components a_4 and b_4 are no longer 1. We developed the matrix M in Chapter 7. Its basic form, given in Equation 7.10, is:

$$M = \begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where c and d are constants that make pseudodepth work properly. What is $M(A, 1)^T$ for this matrix? It's $\tilde{a} = (NA_1, NA_2, CA_3 + d, -A_3)$, the crucial part being that $a_4 = -A_3$. This is the position of the point along the z -axis in camera coordinates, that is the depth of the point in front of the eye.

So the relative sizes of a_4 and b_4 lie at the heart of perspective foreshortening of a line segment: they report the “depths” of A and B , respectively, along the camera’s viewplane normal. If A and B have the same depth (i.e. they lie in a plane parallel to the camera’s viewplane), there is no perspective distortion along the segment, so g and f are indeed the same. Figure 8.46 shows in cross section how rays from the eye through evenly spaced spots (those with equal increments in f) on the viewplane correspond to unevenly spaced spots on the original face in 3D. For the case shown A is closer than B , causing $a_4 < b_4$, so the g -increments grow in size moving across the face from A to B .

Figure 8.46. The values of a_4 and b_4 are related to the depths of points.

Rendering incrementally.

We now put these ingredients together and find the proper texture coordinates (s, t) at each point on the face being rendered. Figure 8.47 shows a face of the barn being rendered. The left edge of the face has endpoints a and b . The face extends from x_{left} to x_{right} across scan-line y . We need to find appropriate texture coordinates $(s_{\text{left}}, t_{\text{left}})$ and $(s_{\text{right}}, t_{\text{right}})$ to attach to x_{left} and x_{right} , respectively, which we can then interpolate across the scan-line. Consider finding $s_{\text{left}}(y)$, the value of s_{left} at scan-line y .

We know that texture coordinate s_A is attached to point a , and s_B is attached to point b , since these values have been passed down the pipeline along with the vertices A and B . If the scan-line at y is fraction f of the way between y_{bott} and y_{top} (so that $f = (y - y_{\text{bott}})/(y_{\text{top}} - y_{\text{bott}})$), then we know from Equation 8.19 that the proper texture coordinate to use is:

Figure 8.47. Rendering the texture on a face.

$$s_{\text{left}}(y) = \frac{\text{lerp}\left(\frac{s_A}{a_4}, \frac{s_B}{b_4}, f\right)}{\text{lerp}\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)} \quad (8.20)$$

and similarly for t_{left} . Notice that s_{left} and t_{left} have the same denominator: a linear interpolation between values $1/a_4$ and $1/b_4$. The numerator terms are linear interpolations of texture coordinates which have been divided by a_4 and b_4 . This is sometimes called “rational linear” rendering [heckbert91] or “hyperbolic interpolation” [blinn92]. To calculate (s, t) efficiently as f advances we need to store values of s_A/a_4 , s_B/b_4 , t_A/a_4 , t_B/b_4 , $1/a_4$, and $1/b_4$, as these don’t change from pixel to pixel. Both the numerator and denominator terms can be found incrementally for each y , just as we did for Gouraud shading (see Equation 8.15). But to find s_{left} and t_{left} we must still perform an explicit division at each value of y .

The pair $(s_{\text{right}}, t_{\text{right}})$ is calculated in a similar fashion. They have denominators that are based on values of a_4' and b_4' that arise from the projected points a' and b' .

Once $(s_{\text{left}}, t_{\text{left}})$ and $(s_{\text{right}}, t_{\text{right}})$ have been found the scan-line can be filled. For each x from x_{left} to x_{right} the values s and t are found, again by hyperbolic interpolation. (what is the expression for s at x ?)

Implications for the graphics pipeline.

What are the implications of having to use hyperbolic interpolation to render texture properly? And does the clipping step need any refinement? As we shall see, we must send certain additional information down the pipeline, and calculate slightly different quantities than supposed so far.

Figure 8.48 shows a refinement of the pipeline. Various points are labeled with the information that is available at that point. Each vertex V is associated with a texture pair (s, t) as well as a vertex normal. The vertex is transformed by the modelview matrix (and the normal is multiplied by the inverse transpose of this matrix), producing vertex $A = (A_1, A_2, A_3)$ and a normal \mathbf{n}' in eye coordinates. Shading calculations are done using this normal, producing the color $\mathbf{c} = (c_r, c_g, c_b)$. The texture coordinates (s_A, t_A) (which are the same as (s, t)) are still attached to A . Vertex A then undergoes the perspective transformation, producing $\tilde{a} = (a_1, a_2, a_3, a_4)$. The texture coordinates and color \mathbf{c} are not altered.

Figure 8.48. Refinement of the graphics pipeline to include hyperbolic interpolation.

Now clipping against the view volume is done, as discussed in Chapter 7. As the figure suggests, this can cause some vertices to disappear and others to be formed. When a vertex such as D is created we must determine its position (d_1, d_2, d_3, d_4) and attach to it the appropriate color and texture point. By the nature of the clipping algorithm the position components d_i are formed by linear interpolation: $d_i = \text{lerp}(a_i, b_i, t)$, for $i = 1, \dots, 4$, for some t . Notice that the fourth component d_4 is also formed this way. It is natural to use linear interpolation here also to form both the color components and the texture coordinates. (The rationale for this is discussed in the exercises.) Therefore after clipping the face still consists of a number of vertices, and to each is attached a color and a texture point. For point A the information is stored in the array $(a_1, a_2, a_3, a_4, s_A, t_A, \mathbf{c}, 1)$. A final term of 1 has been appended: we will use it in the next step.

Now perspective division is done. Since for hyperbolic interpolation we need terms such as s_A/a_4 and $1/a_4$ (see Equation 8.20) we divide *every* item in the array that we wish to interpolate hyperbolically by a_4 to obtain the array $(x, y, z, 1, s_A/a_4, t_A/a_4, \mathbf{c}, 1/a_4)$. (We could also divide the color components in order to obtain slightly more realistic Gouraud shading. See the exercises.) The

first three, $(x, y, z) = (a_1/a_4, a_2/a_4, a_3/a_4)$ report the position of the point in normalized device coordinates. The third component is pseudodepth. The first two components are scaled and shifted by the viewport transformation. To simplify notation we shall continue to call the screen coordinate point (x, y, z) .

So finally the renderer receives the array $(x, y, z, 1, s_A/a_4, t_A/a_4, \mathbf{c}, 1/a_4)$ for each vertex of the face to be rendered. Now it is simple to render texture using hyperbolic interpolation as in Equation 8.20: the required values s_A/a_4 and $1/a_4$ are available for each vertex.

Practice exercises.

8.5.1. Data structures for mesh models with textures. Discuss the specific data types needed to represent mesh objects in the two cases:

- a). a different texture is to be applied to each face;
- b). a single texture is to be “wrapped” around the entire mesh.

Draw templates for the two data types required, and for each show example data in the various arrays when the mesh holds a cube.

8.5.2. Pseudodepth calculations are correct. Show that it is correct, as claimed in Section 8.4, to use linear (rather than hyperbolic) interpolation when finding pseudodepth. Assume point A projects to a , and B projects to b . With linear interpolation we compute pseudodepth at the projected point $\text{lerp}(a, b, f)$ as the third component of this point. This is the correct thing to do only if the resulting value equals the true pseudodepth of the point that $\text{lerp}(A, B, g)$ (for the appropriate g) projects to. Show that it is in fact correct. Hint: Apply Equations 8.16 and 8.17 to the third component of the point being projected.

8.5.3. Wrapping and clamping textures in OpenGL. To make the pattern “wrap” or “tile” in the s direction use: `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)`. Similarly use `GL_TEXTURE_WRAP_T` for wrapping in the t -direction. This is actually the default, so you needn’t do this explicitly. To turn off tiling replace `GL_REPEAT` with `GL_CLAMP`. Refer to the OpenGL documentation for more details, and experiment with different OpenGL settings to see their effect.

8.5.4. Rationale for linear interpolation of texture during clipping. New vertices are often created when a face is clipped against the view volume. We must assign texture coordinates to each vertex. Suppose a new vertex V is formed that is fraction f of the way from vertex A to vertex B on a face. Further suppose that A is assigned texture coordinates (s_A, t_A) , and similarly for B . Argue why, if a texture is considered as “pasted” onto a flat face, it makes sense to assign texture coordinates $(\text{lerp}(s_A, s_B, f), \text{lerp}(t_A, t_B, f))$ to V .

8.5.5. Computational burden of hyperbolic interpolation. Compare the amount of computation required to perform hyperbolic interpolation versus linear interpolation of texture coordinates.

Assume multiplication and division each require 10 times as much time as addition and subtraction.

8.5.3. What does the texture modulate?

How are the values in a texture map “applied” in the rendering calculation? We examine three common ways to use such values in order to achieve different visual effects. We do it for the simple case of the gray scale intensity calculation of Equation 8.5. For full color the same calculations are applied individually for the red, green, and blue components.

1). Create a glowing object.

This is the simplest method computationally. The visible intensity I is set equal to the texture value at each spot:

$$I = \text{texture}(s, t)$$

(or to some constant multiple of it). So the object appears to emit light or glow: lower texture values emit less light and higher texture values emit more light. No additional lighting calculations need be done.

(For colored light the red, green, and blue components are set separately: for instance, the red component is $I_r = \text{texture}_r(s, t)$.)

To cause OpenGL to do this type of texturing, specify:

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE8);
```

2). Paint the texture by modulating the reflection coefficient.

We noted earlier that the color of an object is the color of its diffuse light component (when bathed in white light). Therefore we can make the texture appear to be painted onto the surface by varying the diffuse reflection coefficient, and perhaps the ambient reflection coefficient as well. We say that the texture function “modulates” the value of the reflection coefficient from point to point. Thus we replace Equation 8.5 with:

$$I = \text{texture}(s, t)[I_a \rho_a + I_d \rho_d \times \text{lambert}] + I_{sp} \rho_s \times \text{phong}^f$$

for appropriate values of s and t . Since Phong specular reflections are the color of the source rather than the object, highlights do not depend on the texture.

To cause OpenGL to do this type of texturing, specify:

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

3). Simulate roughness by Bump Mapping.

Bump mapping is a technique developed by Blinn [blinn78] to give a surface a wrinkled (like a raisin) or dimpled (like an orange) appearance without struggling to model each dimple itself. Here the texture function is used to perturb the surface normal vector, which causes perturbations in the amount of diffuse and specular light. Figure 8.49 shows one example, and Plate ??? shows another. One problem associated with bump mapping is that since the model itself does not contain the dimples, the object’s silhouette doesn’t show dimples either, but is perfectly smooth along each face. In addition, the corner between two adjacent faces is also visible in the silhouette. This can be seen in the example.

screen shot – bump mapping on a buckyball, showing some (smooth) edges in silhouette

Figure 8.49. An apparently dimpled surface, achieved by bump mapping.

The goal is to make a scalar function $\text{texture}(s, t)$ perturb the normal vector at each spot in a controlled fashion. In addition, the perturbation should depend only on the surface shape and the texture itself, and not on the orientation of the object or position of the eye. If it depended on orientation, the dimples would change as the object moved in an animation, contrary to the desired effect.

Figure 8.50 shows in cross section how bump mapping works. Suppose the surface is represented parametrically by the function $P(u, v)$, and has unit normal vector $\mathbf{m}(u, v)$. Suppose further that the 3D point at (u^*, v^*) corresponds to the texture at (u^*, v^*) . Blinn’s method simulates perturbing the position of the true surface in the direction of the normal vector by an amount proportional to $\text{texture}(u^*, v^*)$:

a). b).

Figure 8.50. On the nature of bump mapping.

$$P'(u^*, v^*) = P(u^*, v^*) + \text{texture}(u^*, v^*) \mathbf{m}(u^*, v^*) \quad (8.21)$$

as shown in Figure 8.50a, which adds undulations and wrinkles in the surface. This perturbed surface has a new normal vector $\mathbf{m}'(u^*, v^*)$ at each point. The idea is to use this perturbed normal as if it were “attached” to the original unperturbed surface at each point, as shown in Figure 8.50b. Blinn shows that a good approximation to the $\mathbf{m}'(u^*, v^*)$ (before normalization) is given by:

$$\mathbf{m}'(u^*, v^*) = \mathbf{m}(u^*, v^*) + \mathbf{d}(u^*, v^*) \quad (8.22)$$

where the perturbation vector \mathbf{d} is given by

$$(u^*, v^*) = (\mathbf{m} \times P_v) \text{texture}_u - (\mathbf{m} \times P_u) \text{texture}_v$$

⁸ Use either GL_REPLACE or GL_DECAL.

where texture_u and texture_v are partial derivatives of the texture function with respect to u and v respectively. Further, P_u and P_v are partial derivative of $P(u, v)$ with respect to u and v , respectively. All functions are evaluated at (u^*, v^*) . Derivations of this result may also be found in [watt2, miller98]. Note that the perturbation function depends only on the partial derivatives of $\text{texture}()$, not on $\text{texture}()$ itself.

If a mathematical expression is available for $\text{texture}()$ you can form its partial derivatives analytically. For example, $\text{texture}()$ might undulate in two directions by combining sinewaves, as in: $\text{texture}(u, v) = \sin(au)\sin(bv)$ for some constants a and b . If the texture comes instead from an image array, linear interpolation can be used to evaluate it at (u^*, v^*) , and finite differences can be used to approximate the partial derivatives.

8.5.4. A Texturing Example using OpenGL.

To illustrate how to invoke the texturing tools that OpenGL provides, we show an application that displays a rotating cube having different images painted on its six sides. Figure 8.51 shows a snapshot from the animation created by this program.

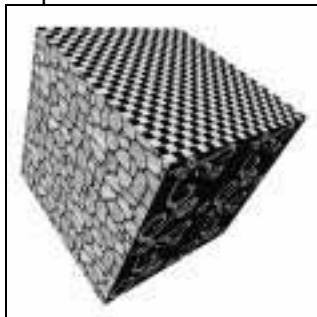


Figure 8.51. The textured cube generated by the example code.

The code for the application is shown in Figure 8.52. It uses a number of OpenGL functions to establish the six textures and to attach them to the walls of the cube. There are many variations of the parameters shown here that one could use to map textures. The version shown works well, but careful adjustment of some parameters (using the OpenGL documentation as a guide) can improve the images or increase performance. We discuss only the basics of the key routines.

One of the first tasks when adding texture to pictures is to create a **pixmap** of the texture in memory. OpenGL uses textures that are stored in “pixel maps”, or pixmaps for short. These are discussed in depth in Chapter 10, and the class **RGB pixmap** is developed that provides tools for creating and manipulating pixmaps. Here we view a pixmap as a simple array of pixel values, each pixel value being a triple of bytes to hold the red, green, and blue color values:

```
class RGB{ // holds a color triple - each with 256 possible intensities
    public: unsigned char r,g,b;
};
```

The **RGB pixmap** class stores the number of rows and columns in the pixmap, as well as the address of the first pixel in memory:

```
class RGBPixmap{
public:
    int nRows, nCols; // dimensions of the pixmap
    RGB* pixel; // array of pixels
    int readBMPFile(char * fname); // read BMP file into this pixmap
    void makeCheckerboard();
    void setTexture(GLuint textureName);
};
```

We show it here as having only three methods that we need for mapping textures. Other methods and details are discussed in Chapter 10. The method `readBMPFile()` reads a BMP file⁹ and stores the pixel values in its `pixmap` object; it is detailed in Appendix 3. The other two methods are discussed next.

Our example OpenGL application will use six textures. To create them we first make an `RGB pixmap` object for each:

```
RGB pixmap pix[6]; // create six (empty) pixmaps
```

and then load the desired texture image into each one. Finally each one is passed to OpenGL to define a texture.

1). Making a procedural texture.

We first create a checkerboard texture using the method `makeCheckerboard()`. The checkerboard pattern is familiar and easy to create, and its geometric regularity makes it a good texture for testing correctness. The application generates a checkerboard pixmap in `pix[0]` using:
`pix[0].makeCheckerboard()`.

The method itself follows:

```
void RGB pixmap:: makeCheckerboard()
{ // make checkerboard pattern
    nRows = nCols = 64;
    pixel = new RGB[3 * nRows * nCols];
    if(!pixel){cout << "out of memory!";return;}
    long count = 0;
    for(int i = 0; i < nRows; i++)
        for(int j = 0; j < nCols; j++)
    {
        int c = (((i/8) + (j/8)) %2) * 255; 10
        pixel[count].r = c; // red
        pixel[count].g = c; // green
        pixel[count++].b = 0; // blue
    }
}
```

It creates a 64 by 64 pixel array, where each pixel is an RGB triple. OpenGL requires that texture pixel maps have a width and height that are both some power of two. The pixel map is laid out in memory as one long array of bytes: row by row from bottom to top, left to right across a row. Here each pixel is loaded with the value $(c, c, 0)$, where c jumps back and forth between 0 and 255 every 8 pixels. (We used a similar “jumping” method in Exercise 2.3.1.) The two colors of the checkerboard are black: (0,0,0), and yellow: (255,255,0). The function returns the address of the first pixel of the pixmap, which is later passed to `glTexImage2D()` to create the actual texture for OpenGL.

Once the pixel map has been formed, we must bind it to a unique integer “name” so that it can be referred to in OpenGL without ambiguity. We arbitrarily assign the names 2001, 2002, ..., 2006 to our six textures in this example¹¹. The texture is created by making certain calls to OpenGL, which we encapsulate in the method:

```
void RGB pixmap :: setTexture(GLuint textureName)
{
    glBindTexture(GL_TEXTURE_2D, textureName);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

⁹ This is a standard device-independent image file format from Microsoft. Many images are available on the internet in BMP format, and tools are readily available on the internet to convert other image formats to BMP files.

¹⁰ A faster way that uses C++’s bit manipulation operators is $c = ((i\&8)\wedge(j\&8)) * 255;$

¹¹ To avoid overlap in (integer) names in an application that uses many textures, it is better to let OpenGL supply unique names for textures using `glGenTextures()`. If we need six unique names we can build an array to hold them: `GLuint name[6]`; and then call `glGenTextures(6, name)`. OpenGL places six heretofore unused integers in `name[0], ..., name[5]`, and we subsequently refer to the i -th texture using `name[i]`.

```
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,  
                GL_UNSIGNED_BYTE, pixel);  
}
```

The call to `glBindTexture()` binds the given name to the texture being formed. When this call is made at a later time, it will make this texture the “active” texture, as we shall see.

The calls to `glTexParameteri()` specify that a pixel should be filled with the texel whose coordinates are nearest the center of the pixel, both when the texture needs to be magnified or reduced in size. This is fast but can lead to aliasing effects. We discuss filtering of images and antialiasing further in Chapter 10. Finally, the call to `glTexImage2D()` associates the pixmap with this current texture. This call describes the texture as 2D consisting of RGB byte-triples, gives its width, height, and the address in memory (`pixel`) of the first byte of the bitmap.

2. Making a texture from a stored image.

OpenGL offers no support for reading an image file and creating the pixel map in memory. The method `readBMPfile()`, given in Appendix 3, provides a simple way to read a BMP image into a pixmap. For instance,

```
pix[1].readBMPFile("mandrill.bmp");
```

reads the file `mandrill.bmp` and creates the pixmap in `pix[1]`.

Once the pixel map has been created, `pix[1].setTexture()` is used to pass the pixmap to OpenGL to make a texture.

Texture mapping must also be enabled with `glEnable(GL_TEXTURE_2D)`. In addition, the routine `glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)` is used to request that OpenGL render the texture properly (using hyperbolic interpolation), so that it appears correctly attached to faces even when a face rotates relative to the viewer in an animation.

Figure 8.52. An application of a rotating textured cube.

The texture creation, enabling, and hinting needs to be done only once, in an initialization routine. Then each time through the display routine the texture is actually applied. In `display()` the cube is rotated through angles `xAngle`, and `yAngle`, and the six faces are drawn. This requires simply that the appropriate texture be bound to the face and that within a `glBegin()/glEnd()` pair the texture coordinates and 3D positions of the face's vertices be specified, as shown in the code.

Once the rendering (off screen) of the cube is complete `glutSwapBuffers()` is called to make the new frame visible. The animation is controlled by using the callback function `spinner()` as the “idle function”. Whenever the system is idle – not responding to user input – `spinner` is called automatically. It alters the rotation angles of the cube slightly, and calls `display()` once again. The effect is an ongoing animation showing the cube rotating, so that its various faces come into view and rotate out of view again and again.

8.5.5. Wrapping texture on curved surfaces.

We have seen how to paste a texture onto a flat surface. Now we examine how to wrap texture onto a curved surface, such as a beer can or a chess piece. We assume as before that the object is modeled by a mesh, so it consists of a large number of small flat faces. As discussed at the end of Section 8.5.1 each vertex of the mesh has an associated texture coordinate pair (s_i, t_i) . The main question is finding the proper texture coordinate (s, t) for each vertex of the mesh.

We present examples of mapping textures onto “cylinder-like” objects and “sphere-like” objects, and see how a modeler might deal with each one.

| Example 8.5.1. Wrapping a label around a can.

Suppose that we want to wrap a label about a circular cylinder, as suggested in Figure 8.53a. It's natural to think in terms of cylindrical coordinates. The label is to extend from θ_a to θ_b in azimuth and from z_a to z_b along the z -axis. The cylinder is modeled as a polygonal mesh, so its walls are rectangular strips as shown in part b. For vertex V_i of each face we must find suitable texture coordinates (s_i, t_i) , so that the correct "slice" of the texture is mapped onto the face.

a). b).

Figure 8.53. Wrapping a label around a cylinder.

The geometry is simple enough here that a solution is straightforward. There is a direct linear relationship between (s, t) and the azimuth and height (θ, z) of a point on the cylinder's surface:

$$s = \frac{\theta - \theta_a}{\theta_b - \theta_a}, \quad t = \frac{z - z_a}{z_b - z_a} \quad (8.23)$$

So if there are N faces around the cylinder, the i -th face has left edge at azimuth $\theta_i = 2\pi i/N$, and its upper left vertex has texture coordinates $(s_i, t_i) = ((2\pi i/N - \theta_a)/(\theta_b - \theta_a), 1)$. Texture coordinates for the other three vertices follow in a similar fashion. This association between (s, t) and the vertices of each face is easily put in a loop in the modeling routine (see the exercises).

Things get more complicated when the object isn't a simple cylinder. We see next how to map texture onto a more general surface of revolution.

Example 8.5.2. "Shrink wrapping" a label onto a Surface of Revolution.

Recall from Chapter 6 that a surface of revolution is defined by a profile curve $(x(v), z(v))^{12}$ as shown in Figure 8.54a, and the resulting surface - here a vase - is given parametrically by $P(u, v) = (x(v) \cos u, x(v) \sin u, z(v))$. The shape is modeled as a collection of faces with sides along contours of constant u and v (see Figure 8.54b). So a given face F_i has four vertices $P(u_i, v_i)$, $P(u_{i+1}, v_i)$, $P(u_i, v_{i+1})$, and $P(u_{i+1}, v_{i+1})$. We need to find the appropriate (s, t) coordinates for each of these vertices.

a). a vase profile b). a face on the vase - four corners

Figure 8.54. Wrapping a label around a vase.

One natural approach is to proceed as above and to make s and t vary linearly with u and v in the manner of Equation 8.23. This is equivalent to wrapping the texture about an imaginary rubber cylinder that encloses the vase (see Figure 8.55a), and then letting the cylinder collapse, so that each texture point slides radially (and horizontally) until it hits the surface of the vase. This method is called "shrink wrapping" by Bier and Sloane [bier86], who discuss several possible ways to map texture onto different classes of shapes. They view shrink wrapping in terms of the imaginary cylinder's normal vector (see Figure 8.55b): texture point P_i is associated with the object point V_i that lies along the normal from P_i .

Figure 8.55. Shrink wrapping texture onto the vase.

Shrink wrapping works well for cylinder-like objects, although the texture pattern will be distorted if the profile curve has a complicated shape.

Bier and Sloane suggest some alternate ways to associate texture points on the imaginary cylinder with vertices on the object. Figure 8.56 shows two other possibilities.

a). centroid b). object normal

Figure 8.56. Alternative mappings from the imaginary cylinder to the object.

¹² We revert to calling the parameters u and v in the parametric representation of the shape, since we are using s and t for the texture coordinates.

In part a) a line is drawn from the object's centroid C , through the vertex V_i , to its intersection with the cylinder P_i . And in part b) the normal vector to the object's surface at V_i is used: P_i is at the intersection of this normal from V_i with the cylinder. Notice that these three ways to associate texture points with object points can lead to very different results depending on the shape of the object (see the exercises). The designer must choose the most suitable method based on the object's shape and the nature of the texture image being mapped. (What would be appropriate for a chess pawn?)

Example 8.5.3. Mapping texture onto a sphere.

It was easy to wrap a texture rectangle around a cylinder: topologically a cylinder can be sliced open and laid flat without distortion. A sphere is a different matter. As all map makers know, there is no way to show accurate details of the entire globe on a flat piece of paper: if you slice open a sphere and lay it flat some parts always suffer serious stretching. (Try to imagine a checkerboard mapped over an entire sphere!)

It's not hard to paste a rectangular texture image onto a *portion* of a sphere, however. To map the texture square to the portion lying between azimuth θ_a to θ_b and latitude ϕ_a to ϕ_b just map linearly as in Equation 8.23: if vertex V_i lies at (θ_i, ϕ_i) associate it with texture coordinates $(s_i, t_i) = ((\theta_i - \theta_a)/(\theta_b - \theta_a), (\phi_i - \phi_a)/(\phi_b - \phi_a))$. Figure 8.57 shows an image pasted onto a band around a sphere. Only a small amount of distortion is seen.

a). texture on portion of sphere b). 8 maps onto 8 octants

Figure 8.57. Mapping texture onto a sphere.

Figure 8.57b shows how one might cover an entire sphere with texture: map eight triangular texture maps onto the eight octants of the sphere.

Example 8.5.4. Mapping texture to sphere-like objects.

We discussed adding texture to cylinder-like objects above. But some objects are more sphere-like than cylinder-like. Figure 8.58a shows the buckyball, whose faces are pentagons and hexagons. One could devise a number of pentagonal and hexagonal textures and manually paste one of each face, but for some scenes it may be desirable to wrap the whole buckyball in a single texture.

a). buckyball b). three mapping methods

Figure 8.58. Sphere-like objects.

It is natural to surround a sphere-like object with an imaginary sphere (rather than a cylinder) that has texture pasted to it, and use one of the association methods discussed above. Figure 8.58b shows the buckyball surrounded by such a sphere in cross section. The three ways of associating texture points P_i with object vertices V_i are sketched:

object-centroid: P_i is on a line from the centroid C through vertex V_i ;
 object-normal: P_i is the intersection of a ray from V_i in the direction of the face normal;
 sphere-normal: V_i is the intersection of a ray from P_i in the direction of the normal to the sphere at P_i .

(Question: Are the object-centroid and sphere-normal methods the same if the centroid of the object coincides with the center of the sphere?) The object centroid method is most likely the best, and it is easy to implement. As Bier and Sloane argue, the other two methods usually produce unacceptable final renderings.

Bier and Sloane also discuss using an imaginary box rather than a sphere to surround the object in question. Figure 8.59a shows the six faces of a cube spread out over a texture image, and part b) shows the texture wrapped about the cube, which in turn encloses an object. Vertices on the object can be associated with texture points in the three ways discussed above: the object-centroid and cube-normal are probably the best choices.

a). texture on 6 faces of box b). wrapping texture onto

Figure 8.59. Using an enclosing box.

Practice exercises.

8.5.7. How to associate P_i and V_i . Surface of revolution S shown in Figure 8.60 consists of a sphere resting on a cylinder. The object is surrounded by an imaginary cylinder having a checkerboard texture pasted on it. Sketch how the texture will look for each of the following methods of associating texture points to vertices:

- a). shrink wrapping;
- b). object centroid;
- c). object normal;

Figure 8.60. A surface of revolution surrounded by an imaginary cylinder.

8.5.8. Wrap a texture onto a torus. A torus can be viewed as a cylinder that “bends” around and closes on itself. The torus shown in Figure 8.61 has the parametric representation given by $P(u, v) = ((D + A \cos(v)) \cos(u), (D + A \cos(v)) \sin(u), A \sin(v))$. Suppose you decide to polygonalize the torus by taking vertices based on the samples $u_i = 2\pi i/N$ and $v_j = 2\pi j/M$, and you wish to wrap some texture from the unit texture space around this torus. Write code that generates, for each of the faces, each vertex and its associated texture coordinates (s, t) .

Figure 8.61. Wrapping texture about a torus.

8.5.6. Reflection mapping.

The class of techniques known as “reflection mapping” can significantly improve the realism of pictures, particularly in animations. The basic idea is to see reflections in an object that suggest the “world” surrounding that object.

The two main types of reflection mapping are called “chrome mapping” and “environment mapping.” In the case of **chrome mapping** a rough and usually blurry image that suggests the surrounding environment is reflected in the object, as you would see in a surface coated with chrome. Television commercials abound with animations of shiny letters and logos flying around in space, where the chrome map includes occasional spotlights for dramatic effect. Figure 8.62 offers an example. Part a) shows the chrome texture, and part b) shows it reflecting in the shiny object. The reflection provides a rough suggestion of the world surrounding the object.

- | | |
|----------------|---|
| a). chrome map | b). scene with chrome mapping
(screen shots) |
|----------------|---|

Figure 8.62. Example of chrome mapping.

In the case of **environment mapping** (first introduced by Blinn and Newell [blinn 76]) a recognizable image of the surrounding environment is seen reflected in the object. We get valuable visual cues from such reflections, particularly when the object moves about. Everyone has seen the classic photographs of an astronaut walking on the moon with the moonscape reflected in his face mask. And in the movies you sometimes see close-ups of a character's reflective dark glasses, in which the world about her is reflected. Figure 8.63 shows two examples where a cafeteria texture is wrapped about a large sphere that surrounds the object, so that the texture coordinates (s, t) correspond to azimuth and latitude about the enclosing sphere.



Figure 8.63. Example of environment mapping (courtesy of Haeblerli and Segal).

Figure 8.64 shows the use of a surrounding cube rather than a sphere. Part a) shows the map, consisting of six images of various views of the interior walls, floor, and ceiling of a room. Part b) shows a shiny object reflecting different parts of the room. The use of an enclosing cube was introduced by Greene [greene 86], and generally produces less distorted reflections than are seen with an enclosing sphere. The six maps can be generated by rendering six separate images from the point of view of the object (with the object itself removed, of course). For each image a synthetic camera is set up and the appropriate window is set. Alternatively, the textures can be digitized from photos taken by a real camera that looks in the six principal directions inside an actual room or scene.

a). six images make the map (screen shots)	b). environment mapping
---	-------------------------

Figure 8.64. Environment mapping based on a surrounding cube.

Chrome and environment mapping differ most dramatically from normal texture mapping in an animation when the shiny object is moving. The reflected image will “flow” over the moving object, whereas a normal texture map will be attached to the object and move with it. And if a shiny sphere rotates about a fixed spot a normal texture map spins with the sphere, but a reflection map stays fixed.

How is environment mapping done? What you see at point P on the shiny object is what has arrived at P from the environment in just the right direction to reflect into your eye. To find that direction trace a ray from the eye to P , and determine the direction of the reflected ray. Trace this ray to find where it hits the texture (on the enclosing cube or sphere). Figure 8.65 shows a ray emanating from the eye to point P . If the direction of this ray is \mathbf{u} and the unit normal at P is \mathbf{m} , we know from Equation 8.2 that the reflected ray has direction $\mathbf{r} = \mathbf{u} - 2(\mathbf{u} \bullet \mathbf{m})\mathbf{m}$. The reflected ray moves in direction \mathbf{r} until it hits the hypothetical surface with its attached texture. It is easiest computationally to suppose that the shiny object is centered in, and much smaller than, the enclosing cube or sphere. Then the reflected ray emanates approximately from the object’s center, and its direction \mathbf{r} can be used directly to index into the texture.

Figure 8.65. Finding the direction of the reflected ray.

OpenGL provides a tool to perform approximate environment mapping for the case where the texture is wrapped about a large enclosing sphere. It is invoked by setting a mapping mode for both s and t using:

```
glTexGenf(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGenf(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
 glEnable(GL_TEXTURE_GEN_S);
 glEnable(GL_TEXTURE_GEN_T);
```

Now when a vertex P with its unit normal \mathbf{m} is sent down the pipeline, OpenGL calculates a texture coordinate pair (s, t) suitable for indexing into the texture attached to the surrounding sphere. This is done for each vertex of the face on the object, and the face is drawn as always using interpolated texture coordinates (s, t) for points in between the vertices.

How does OpenGL rapidly compute a suitable coordinate pair (s, t) ? As shown in Figure 8.66a it first finds (in eye coordinates) the reflected direction \mathbf{r} (using the formula above), where \mathbf{u} is the unit vector (in eye coordinates) from the eye to the vertex V on the object, and \mathbf{m} is the normal at V .

a).	b).
-----	-----

Figure 8.66. OpenGL’s computation of the texture coordinates.

It then simply uses the expression:

$$(s, t) = \left(\frac{1}{2} \left(\frac{r_x}{p} + 1 \right), \frac{1}{2} \left(\frac{r_y}{p} + 1 \right) \right) \quad (8.24)$$

where p is a mysterious scaling factor $p = \sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$. The derivation of this term is developed in the exercises. We must precompute a texture that shows what you would see of the environment in a perfectly reflecting sphere, from an eye position far removed from the sphere [haeberli93]. This maps the part of the environment that lies in the hemisphere behind the eye into a circle in the middle of the texture, and the part of the environment in the hemisphere in front of the eye into an annulus around this circle (visualize this). This texture must be recomputed if the eye changes position. The pictures in Figure 8.63 were made using this method.

Simulating Highlights using Environment mapping.

Reflection mapping can be used in OpenGL to produce specular highlights on a surface. A texture map is created that has an intense concentrated bright spot. Reflection mapping “paints” this highlight onto the surface, making it appear to be an actual light source situated in the environment. The highlight created can be more concentrated and detailed than those created using the Phong specular term with Gouraud shading. Recall that the Phong term is computed only at the vertices of a face, and it is easy to “miss” a specular highlight that falls between two vertices. With reflection mapping the coordinates (s, t) into the texture are formed at each vertex, and then interpolated in between. So if the coordinates indexed by the vertices happen to surround the bright spot, the spot will be properly rendered inside the face.

Practice Exercise 8.5.9. OpenGL’s computation of texture coordinates for environment mapping.

Derive the result in Equation 8.24. Figure 8.66b shows in cross-sectional view the vectors involved (in eye coordinates). The eye is looking from a remote location in the direction $(0,0,1)$. A sphere of radius 1 is positioned on the negative z-axis. Suppose light comes in from direction \mathbf{r} , hitting the sphere at the point (x, y, z) . The normal to the sphere at this point is (x, y, z) , which also must be just right so that light coming along \mathbf{r} is reflected into the direction $(0, 0, 1)$. This means the normal must be half-way between \mathbf{r} and $(0, 0, 1)$, or must be proportional to their sum, so $(x, y, z) = K(r_x, r_y, r_z + 1)$ for some K .

- Show that the normal vector has unit length if K is $1/p$, where p is given as in Equation 8.24.
- Show that therefore $(x, y) = (r_x/p, r_y/p)$.
- Suppose for the moment that the texture image extends from -1 to 1 in x and from -1 to 1 in y . Argue why what we want to see reflected at the point (x, y, z) is the value of the texture image at (x, y) .
- Show that if instead the texture uses coordinates from 0 to 1 – as is true with OpenGL – that we want to see at (x, y) the value of the texture image at (s, t) given by Equation 8.24.

8.6. Adding Shadows of Objects.

Shadows make an image much more realistic. From everyday experience the way one object casts a shadow on another object gives important visual cues as to how they are positioned. Figure 8.67 shows two images involving a cube and a sphere suspended above a plane. Shadows are absent in part a, and it is impossible to see how far above the plane the cube and sphere are floating. By contrast, the shadows seen in part b give useful hints as to the positions of the objects. A shadow conveys a lot of information; it’s as if you are getting a second look at the object (from the viewpoint of the light source).

- | | |
|---------------------|------------------|
| a). with no shadows | b). with shadows |
| (screen shots) | |

Figure 8.67. The effect on shadows.

In this section we examine two methods for computing shadows: one is based on “painting” shadows as if they were texture, and the other is an adaptation of the depth buffer approach for hidden surface removal. In Chapter 14 we see that a third method arises naturally when raytracing. There are many other techniques, well surveyed in [watt92, crow77, woo90, bergeron86].

8.6.1. Shadows as Texture.

This technique displays shadows that are cast onto a flat surface by a point light source. The problem is to compute the shape of the shadow that is cast. Figure 8.68a shows a box casting a shadow onto the floor. The shape of the shadow is determined by the projections of each of the faces of the box onto the plane of the floor, using the source as the center of projection. In fact the shadow is the union¹³ of the projections of the six faces. Figure 8.68b shows the superposed

¹³ the set theoretic union: A point is in the shadow if it is in one or more of the projections.

projections of two of the faces: the top face projects to *top*' and the front face to *front*'. (Sketch the projections of the other four faces, and see that their union is the required shadow¹⁴.)

a). b).

Figure 8.68. Computing the shape of a shadow.

This is the key to drawing the shadow. After drawing the plane using ambient, diffuse, and specular light contributions, draw the six projections of the box's faces on the plane using only ambient light. This will draw the shadow in the right shape and color. Finally draw the box. (If the box is near the plane parts of it might obscure portions of the shadow.)

Building the “projected” face:

To make the new face F' produced by F , project each of its vertices onto the plane in question. We need a way to calculate these vertex positions on the plane. Suppose, as in Figure 8.68a, that the plane passes through point A and has normal vector \mathbf{n} . Consider projecting vertex V , producing point V' . The mathematics here are familiar: Point V' is the point where the ray from the source at S through V hits the plane. As developed in the exercises, this point is:

$$V' = S + (V - S) \frac{\mathbf{n} \cdot (A - S)}{\mathbf{n} \cdot (V - S)} \quad (8.25)$$

The exercises show how this can be written in homogeneous coordinates as V times a matrix, which is handy for rendering engines, like OpenGL, that support convenient matrix multiplication.

Practice Exercises.

8.6.1. Shadow shapes. Suppose a cube is floating above a plane. What is the shape of the cube's shadow if the point source lies a) directly above the top face? b). along a main diagonal of the cube (as in an isometric view)? Sketch shadows for a sphere and for a cylinder floating above a plane for various source positions.

8.6.2. Making the “shadow” face. a). Show that the ray from the source point S through vertex V hits the plane $\mathbf{n} \cdot (P - A) = 0$ at $t^* = \mathbf{n} \cdot (A - S) / \mathbf{n} \cdot (V - S)$; b). Show that this defines the hit point V' as given in Equation 8.25.

8.6.3. It's equivalent to a matrix multiplication. a). Show that the expression for V' in Equation 8.25 can be written as a matrix multiplication: $V' = M(V_x, V_y, V_z, 1)^T$, where M is a 4 by 4 matrix
b). Express the terms of M in terms of A , S , and \mathbf{n} .

8.6.2. Shadows using a shadow buffer.

A rather different method for drawing shadows uses a variant of the depth buffer that performs hidden surface removal. It uses an auxiliary second depth buffer, called a **shadow buffer**, for each light source. This requires a lot of memory, but this approach is not restricted to casting shadows onto planar surfaces.

The method is based on the principle that any points in the scene that are “hidden” from the light source must be in shadow. On the other hand, if no object lies between a point and the light source the point is not in shadow. The shadow buffer contains a “depth picture” of the scene from the point of view of the light source: each of its elements records the distance from the source to the *closest* object in the associated direction.

Rendering is done in two stages:

1). **Shadow buffer loading.** The shadow buffer is first initialized with 1.0 in each element, the largest pseudodepth possible. Then, using a camera positioned at the light source, each of the faces in the scene is scan converted, but only the pseudodepth of the point on the face is tested. Each element of the shadow buffer keeps track of the smallest pseudodepth seen so far.

To be more specific, Figure 8.69 shows a scene being viewed by the usual “eye camera” as well as a “source camera” located at the light source. Suppose point P is on the ray from the source through

¹⁴ You need to form the union of the projections of only the three “front” faces: those facing toward the light source. (Why?)

shadow buffer “pixel” $d[i][j]$, and that point B on the pyramid is also on this ray. If the pyramid is present $d[i][j]$ contains the pseudodepth to B ; if it happens to be absent $d[i][j]$ contains the pseudodepth to P .

Figure 8.69. Using the shadow buffer.

Note that the shadow buffer calculation is independent of the eye position, so in an animation where only the eye moves the shadow buffer is loaded only once. The shadow buffer must be recalculated, however, whenever the objects move relative to the light source.

2). Render the scene. Each face in the scene is rendered using the eye camera as usual. Suppose the eye camera “sees” point P through pixel $p[c][r]$. When rendering $p[c][r]$ we must find¹⁵:

- the pseudodepth D from the source to P ;
- the index location $[i][j]$ in the shadow buffer that is to be tested;
- the value $d[i][j]$ stored in the shadow buffer.

If $d[i][j]$ is less than D the point P is in shadow, and $p[c][r]$ is set using only ambient light. Otherwise P is not in shadow and $p[c][r]$ is set using ambient, diffuse, and specular light.

How are these steps done? As described in the exercises, to each point on the eye camera viewplane there corresponds a point on the source camera viewplane¹⁶. For each screen pixel this correspondence is invoked to find the pseudodepth from the source to P as well as the index $[i][j]$ that yields the minimum pseudodepth stored in the shadow buffer.

Practice Exercises.

8.6.4. Finding pseudodepth from the source. Suppose the matrices M_c and M_s map the point P in the scene to the appropriate (3D) spots on the eye camera’s viewplane and the source camera’s viewplane, respectively. a). Describe how to establish a “source camera” and how to find the resulting matrix M_s . b). Find the transformation that, given position (x, y) on the eye camera’s viewplane produces the position (i, j) and pseudodepth on the source camera’s viewplane. c). Once (i, j) are known, how is the index $[i][j]$ and the pseudodepth of P on the source camera determined?

8.6.5. Extended Light sources. We have considered only point light sources in this chapter.

Greater realism is provided by modeling extended light sources. As suggested in Figure 8.70a such sources cast more complicated shadows, having an **umbra** within which no light from the source is seen, and a lighter **penumbra** within which a part of the source is visible. In part b) a glowing sphere of radius 2 shines light on a unit cube, thereby casting a shadow on the wall W . Make an accurate sketch of the umbra and penumbra that is observed on the wall. As you might expect, algorithms for rendering shadows due to extended light sources are complex. See [watt92] for a thorough treatment.

a). umbra and penumbra b). example to sketch

Figure 8.70. Umbra and penumbra for extended light sources.

8.7. Summary

Since the beginning of computer graphics there has been a relentless quest for greater realism when rendering 3D scenes. Wireframe views of objects can be drawn very rapidly but are difficult to interpret, particularly if several objects in a scene overlap. Realism is greatly enhanced when the faces are filled with some color and surfaces that should be hidden are removed, but pictures rendered this way still do not give the impression of objects residing in a scene, illuminated by light sources.

¹⁵ Of course, this test is made only if P is closer to the eye than the value stored in the normal depth buffer of the eye camera.

¹⁶ Keep in mind these are 3D points: 2 position coordinates on the viewplane, and pseudodepth.

What is needed is a shading model, that describes how light reflects off a surface depending on the nature of the surface and its orientation to both light sources and the camera's eye. The physics of light reflection is very complex, so programmers have developed a number of approximations and tricks that do an acceptable job most of the time, and are reasonably efficient computationally. The model for the diffuse component is the one most closely based on reality, and becomes extremely complex as more and more ingredients are considered. Specular reflections are not modeled on physical principles at all, but can do an adequate job of recreating highlights on shiny objects. And ambient light is purely an abstraction, a shortcut that avoids dealing with multiple reflections from object to object, and prevents shadows from being too deep.

Even simple shading models involve several parameters such as reflection coefficients, descriptions of a surface's roughness, and the color of light sources. OpenGL provides ways to set many of these parameters. There is little guidance for the designer in choosing the values of these parameters; they are often determined by trial and error until the final rendered picture looks right.

In this chapter we focused on rendering of polygonal mesh models, so the basic task was to render a polygon. Polygonal faces are particularly simple and are described by a modest amount of data, such as vertex positions, vertex normals, surface colors and material. In addition there are highly efficient algorithms for filling a polygonal face with calculated colors, especially if it is known to be convex. And algorithms can capitalize on the flatness of a polygon to interpolate depth in an incremental fashion, making the depth buffer hidden surface removal algorithm simple and efficient.

When a mesh model is supposed to approximate an underlying smooth surface the appearance of a face's edges can be objectionable. Gouraud and Phong shading provide ways to draw a smoothed version of the surface (except along silhouettes). Gouraud shading is very fast but does not reproduce highlights very faithfully; Phong shading produces more realistic renderings but is computationally quite expensive.

The realism of a rendered scene is greatly enhanced by the appearance of texturing on object surfaces. Texturing can make an object appear to be made of some material such as brick or wood, and labels or other figures can be pasted onto surfaces. Texture maps can be used to modulate the amount of light that reflects from an object, or as "bump maps" that give a surface a bumpy appearance. Environment mapping shows the viewer an impression of the environment that surrounds a shiny object, and this can make scenes more realistic, particularly in animations. Texture mapping must be done with care, however, using proper interpolation and antialiasing (as we discuss in Chapter 10).

The chapter closed with a description of some simple methods for producing shadows of objects. This is a complex subject, and many techniques have been developed. The two algorithms described provide simple but partial solutions to the problem.

Greater realism can be attained with more elaborate techniques such as ray tracing and radiosity. Chapter 14 develops the key ideas of these techniques.

8.8. Case Studies.

8.8.1. Case Study 8.1. Creating shaded objects using OpenGL

(Level of Effort: II beyond that of Case Study 7.1). Extend Case Study 7.1 that flies a camera through space looking at various polygonal mesh objects. Extend it by establishing a point light source in the scene, and assigning various material properties to the meshes. Include ambient, diffuse, and specular light components. Provide a keystroke that switches between flat and smooth shading

8.8.2. Case Study 8.2. The Do-it-yourself graphics pipeline.

(Level of Effort: III) Write an application that reads a polygonal mesh model from a file as described in Chapter 6, defines a camera and a point light source, and renders the mesh object using flat shading with ambient and diffuse light contributions. Only gray scale intensities need be computed. For this project do *not* use OpenGL's pipeline; instead create your own. Define modelview, perspective, and viewport matrices. Arrange that vertices can be passed through the

first two matrices, have the shading model applied, followed by perspective division (no clipping need be done) and by the viewport transformation. Each vertex emerges as the array $\{x, y, z, b\}$ where x and y are screen coordinates, z is pseudodepth, and b is the grayscale brightness of the vertex. Use a tool that draws filled polygons to do the actual rendering: if you use OpenGL, use only its 2D drawing (and depth buffer) components. Experiment with different mesh models, camera positions, and light sources to insure that lighting is done properly.

8.8.3. Case Study 8.3. Add Polygon Fill and Depth Buffer HSR.

(Level of Effort: III beyond that needed for Case Study 8.2.) Implement your own depth buffer, and use it in the application of Case Study 8.2. This requires the development of a polygon fill routine as well - see Chapter 10.

8.8.4. Case Study 8.4. Texture Rendering.

(Level of Effort: II beyond that of Case Study 8.1). Enhance the program of Case Study 8.1 so that textures can be painted on the faces of the mesh objects. Assemble a routine that can read a BMP image file and attach it to an OpenGL texture object. Experiment by putting five different image textures and one procedural texture on the sides of a cube, and arranging to have the cube rotate in an animation. Provide a keystroke that lets the user switch between linear interpolation and correct interpolation for rendering textures.

8.8.5. Case Study 8.5. Applying Procedural 3D textures.

(Level of Effort: III) An interesting effect is achieved by making an object appear to be carved out of some solid material, such as wood or marble. Plate ??? shows a (raytraced) vase carved out of marble, and Plate ??? shows a box apparently made of wood. 3D textures are discussed in detail in Chapter 14 in connection with ray tracing, but it is also possible to map “slices” of a 3D texture onto the surfaces of an object, to achieve a convincing effect.

Suppose you have a texture function $B(x, y, z)$ which attaches different intensities or colors to different points in 3D space. For instance $B(x, y, z)$ might represent how “inky” the sea is at position (x, y, z) . As you swim around you encounter a varying inkiness right before your eyes. If you freeze a block of this water and carve some shape out of the block, the surface of the shape will exhibit a varying inkiness. $B()$ can be vector-valued as well: providing three values at each (x, y, z) , which might represent the diffuse reflection coefficients for red, green, and blue light of the material at each point in space. It’s not hard to construct interesting functions $B()$:

a). A 3D black and white checkerboard with 125 blocks is formed using:

$$B(x, y, z) = ((int)(5x) + (int)(5y) + (int)(5z)) \% 2 \text{ as } x, y, z \text{ vary from 0 to 1.}$$

b). A “color cube” has six different colors at its vertices, with a continuously varying color at points in between. Just use $B(x, y, z) = (x, y, z)$ where x, y , and z vary from 0 to 1. The vertex at $(0, 0, 0)$ is black, that at $(1, 0, 0)$ is red, etc.

c). All of space can be filled with such cubes stacked upon one another using $B(x, y, z) = (fract(x), fract(y), fract(z))$ where $fract(x)$ is the fractional part of the value x .

Methods for creating wood grain and turbulent marble are discussed in Chapter 14. They can be used here as well.

In the present context we wish to paste such texture onto surfaces. To do this a bitmap is computed using $B()$ for each surface of the object. If the object is a cube, for instance, six different bitmaps are computed, one for each face of the cube. Suppose a certain face of the cube is characterized by the planar surface $P + \mathbf{a}t + \mathbf{b}s$ for s, t in 0 to 1. Then use as texture $B(P_x + a_xt + b_xs, P_y + a_yt + b_ys, P_z + a_zt + b_zs)$. Notice that if there is any coherence to the pattern $B()$ (so nearby points enjoy somewhat the same inkiness or color), then nearby points on adjacent faces of the cube will also have nearly the same color. This makes the object truly look like it is carved out of a single solid material.

Extend Case Study 8.4 to include pasting texture like this onto the faces of a cube and an icosahedron. Use a checkerboard texture, a color cube texture, and a wood grain texture (as described in Chapter 14).

Form a sequence of images of a textured cube, where the cube moves slightly through the material from frame to frame. The object will appear to “slide through” the texture in which it is imbedded. This gives a very different effect from an object moving with its texture attached. Experiment with such animations.

8.8.6. Case Study 8.6. Drawing Shadows.

(Level of Effort:III) Extend the program of Case Study 8.1 to produce shadows. Make one of the objects in the scene a flat planar surface, on which is seen shadows of other objects. Experiment with the “projected faces” approach. If time permits, develop as well the shadow buffer approach.

8.8.7. Case Study 8.7. Extending SDL to Include Texturing.

(Level of Effort:III) The SDL scene description language does not yet include a means to specify the texture that one wants applied to each face of an object. The keyword `texture` is currently in SDL, but does nothing when encountered in a file. Do a careful study of the code in the `Scene` and `Shape` classes, available on the book’s internet site, and design an approach that permits a syntax such as

```
texture giraffe.bmp p1 p2 p3 p4
```

to create a texture from a stored image (here `giraffe.bmp`) and paste it onto certain faces of subsequently defined objects. Determine how many parameters `texture` should require, and how they should be used. Extend `drawOpenGL()` for two or three shapes so that it properly pastes such texture onto the objects in question.

8.9. For Further Reading

Jim Blinn’s two JIM BLINN’S CORNER books: A TRIP DOWN THE GRAPHICS PIPELINE [blinn96] and DIRTY PIXELS [blinn98] offer several articles that lucidly explain the issues of drawing shadows and the hyperbolic interpolation used in rendering texture. Heckbert’s “Survey of Texture Mapping” [heckbert86] gives many interesting insights into this difficult topic. The papers “Fast Shadows and Lighting Effects Using Texture Mapping” by Segal et al [segal92] and “Texture Mapping as a Fundamental Drawing Primitive” by Haeberli and Segal [haeberli93] (also available on-line: <http://www.sgi.com/grafica/texmap/>) provide excellent background and context.