The McGraw-Hill Companies

SCHAUM'S ouTlines

COMPUTER GRAPHICS

ZHIGANG XIANG | ROY A PLASTOCK

Second Edition

- Simplifies all aspects of creating digital graphics.
- Detailed discussions on Computer Animation, Graphic I/O Devices.
- Algorithms presented in 'C' language.
- Over 410 solved examples, problems and objective questions.

Adapted by: P S AVADHANI

Tata McGraw-Hill

Special Indian Edition 2006

Adapted in India by arrangement with The McGraw-Hill Companies, Inc., New York

Sales Territories: India, Pakistan, Nepal, Bangladesh, Sri Lanka and Bhutan

Schaums Outline of Theory and Problems of
COMPUTER GRAPHICS

Fourth reprint 2008
RBLCRRCXRZLZL

ISBN-13: 978-0-07-060165-9
ISBN-10: 0-07-060165-8

Published by the Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, typeset in Times at The Composers,
260 C.A. Apt., Paschim Vihar, New Delhi 110 063 and printed at
Gopaljee Enterprises, Delhi 110 053

Cover: SDR Printers

The McGraw-Hill Companies

Preface

We live in a world full of scientific and technological advances. In recent years it has become quite difficult not to notice the proliferation of something called computer graphics. Almost every computer system is set up to allow the user to interact with the system through a graphical user interface, where information on the display screen is conveyed in both textual and graphical forms. Movies and video games are popular showcases of the latest technology for people, both young and old. Watching the TV for a while, the likelihood is that you will see the magic touch of computer graphics in a commercial.

This book is both a self-contained text and a valuable study aid on the fundamental principles of computer graphics. It takes a goal-oriented approach to discuss the important concepts, the underlying mathematics, and the algorithmic aspects of the computerized image synthesis process. It contains hundreds of solved problems that help reinforce one's understanding of the field and exemplify effective problem-solving techniques.

Although the primary audience are college students taking a computer graphics course in a computer science or computer engineering program, any educated person with a desire to look into the inner workings of computer graphics should be able to learn from this concise introduction. The recommended prerequisites are some working knowledge of a computer system, the equivalent of one or two semesters of programming, a basic understanding of data structures and algorithms, and a basic knowledge of linear algebra and analytical geometry.

The field of computer graphics is characterized by rapid changes in how the technology is used in everyday applications and by constant evolution of graphics systems. The life span of graphics hardware seems to be getting shorter and shorter. An industry standard for computer graphics often becomes obsolete before it is finalized. A programming language that is a popular vehicle for graphics applications when a student begins his or her college study is likely to be on its way out by the time he or she graduates.

In this book we try to cover the key ingredients of computer graphics that tend to have a lasting value (only in relative terms, of course). Instead of compiling highly equipment-specific or computing environment-specific information, we strive to provide a good explanation of the fundamental concepts and the relationship between them. We discuss subject matters in the overall framework of computer graphics and emphasize mathematical and/or algorithmic solutions. Algorithms are presented in pseudo-code rather than a particular programming language. Examples are given with specifics to the extent that they can be easily made into working versions on a particular computer system.

We believe that this approach brings unique benefit to a diverse group of readers. First, the book can be read by itself as a general introduction to computer graphics for people who want technical substance but not the burden of implementational overhead. Second, it can be used by instructors and students as a resource book to supplement any comprehensive primary text. Third, it may serve as a stepping-stone for practitioners who want something that is more understandable than their graphics system's programmer's manuals.

The first edition of this book has served its audience well for over a decade. I would like to salute and thank my coauthors for their invaluable groundwork. The current version represents a significant revision to the original, with several chapters replaced to cover new topics, and the remaining material updated throughout the rest of the book. I hope that it can serve our future audience as well for years to come.

Thank you for choosing our book. May you find it stimulating and rewarding.

ZHIGANG XIANG

Contents

Chapter One

Introduction

Computer graphics is generally regarded as a branch of computer science that deals with the theory and technology for computerized image synthesis. A computer-generated image can depict a scene as simple as the outline of a triangle on a uniform background and as complex as a magnificent dinosaur in a tropical forest. But how do these things become part of the picture? What makes drawing on a computer different from sketching with a pen or photographing with a camera? In this chapter we will introduce some important concepts and outline the relationship among these concepts. The goal of such a mini-survey of the field of computer graphics is to enable us to appreciate the various answers to these questions that we will detail in the rest of the book not only in their own right but also in the context of the overall framework.

The task of composing an image on a computer is essentially a matter of setting pixel values. The collective effects of the pixels taking on different color attributes give us what we see as a picture. In the subsequent sections we first present an overview of image representation (Section 1.2) and introduce the basics of the most prevailing color specification method in computer graphics (Section 1.3). We then discuss the representation of images using direct coding of pixel colors (Section 1.4) versus using the lookup table approach (Section 1.5). Following a discussion of the working principles of two representative image presentation devices, the display monitor (Section 1.6) and the printer (Section 1.7), we examine image files as the primary means of image storage and transmission (Section 1.8). We then take a look at some of the most primitive graphics operations, which primarily deal with setting the color attributes of pixels (Section 1.9). Finally, to illustrate the construction of beautiful images directly in the discrete image space, we introduce the mathematical background and detail the algorithmic aspects of visualizing the Mandelbrot set (Section 1.10).

1.1 A MINI-SURVEY

First let's consider drawing the outline of a triangle (see Fig. 1.1). In real life this would begin with a decision in our mind regarding such geometric characteristics as the type and size of the triangle, followed by our action to move a pen across a piece of paper. In computer graphics terminology, what we have envisioned is called the object definition, which defines the triangle in an abstract space of our choosing. This space is continuous and is called the *object space*. Our action to draw maps the imaginary object into a triangle on paper, which constitutes a continuous display surface in another space called the *image space*. This mapping action is further influenced by our choice regarding such factors as the location and orientation of the triangle. In other words, we may place the triangle in the middle of the paper, or we may draw it near the upper left corner. We may have the sharp corner of the triangle pointing to the right, or we may have it pointing to the left.

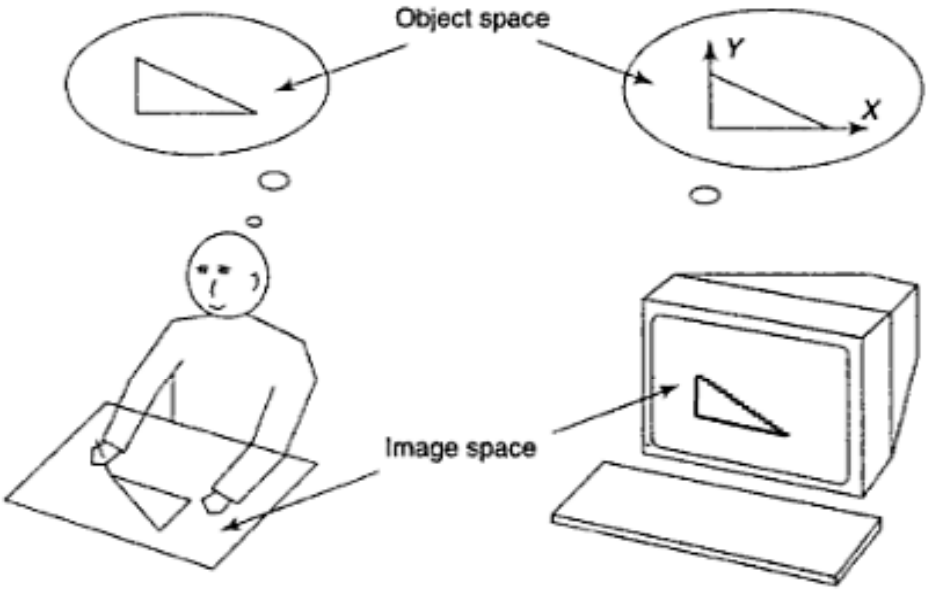


Fig. 1.1 *Drawing a Triangle*

A comparable process takes place when a computer is used to produce the picture. The major computational steps involved in the process give rise to several important areas of computer graphics. The area that attends to the need to define objects, such as the triangle, in an efficient and effective manner is called *geometric representation*. In our example we can place a two-dimensional Cartesian coordinate system into the object space. The triangle can then be represented by the x and y coordinates of its three vertices, with the understanding that the computer system will connect the first and second vertices with a line segment, the second and third vertices with another line segment, and the third and first with yet another line segment.

The next area of computer graphics that deals with the placement of the triangle is called *transformation*. Here we use matrices to realize the mapping of the triangle to its final destination in the image space. We can set up the transformation matrix to control the location and orientation of the displayed triangle. We can even enlarge or reduce its size. Furthermore, by using multiple

settings for the transformation matrix, we can instruct the computer to display several triangles of varying size and orientation at different locations, all from the same model in the object space.

At this point most readers may have already been wondering about the crucial difference between the triangle drawn on paper and the triangle displayed on the computer monitor (an exaggerated version of what you would see on a real monitor). The former has its vertices connected by smooth edges, whereas the latter is not exactly a line-drawing. The fundamental reason here is that the image space in computer graphics is, generally speaking, not continuous. It consists of a set of discrete pixels, i.e. picture elements, that are arranged in a row-and-column fashion. Hence a horizontal or vertical line segment becomes a group of adjacent pixels in a row or column, respectively, and a slanted line segment becomes something that resembles a staircase. The area of computer graphics that is responsible for converting a continuous figure, such as a line segment, into its discrete approximation is called *scan conversion*.

The distortion introduced by the conversion from continuous space to discrete space is referred to as the *aliasing effect* of the conversion. While reducing the size of individual pixels should make the distortion less noticeable, we do so at a significant cost in terms of computational resources. For instance, if we cut each pixel by half in both the horizontal and the vertical direction we would need four times the number of pixels in order to keep the physical dimension of the picture constant. This would translate into, among other things, four times the memory requirement for storing the image. Exploring other ways to alleviate the negative impact of the aliasing effect is the focus of another area of computer graphics called *anti-aliasing*.

Putting together what we have so far leads to a simplified graphics pipeline (see Fig. 1.2), which exemplifies the architecture of a typical graphics system. At the start of the pipeline, we have primitive objects represented in some application-dependent data structures. For example, the coordinates of the vertices of a triangle, viz. (x_1, y_1), (x_2, y_2), and (x_3, y_3), can be easily stored in a 3×2 array. The graphics system first performs transformation on the original data according to user-specified parameters, and then carries out scan conversion with or without anti-aliasing to put the picture on the screen. The coordinate system in the middle box in Figure 1.2 serves as an intermediary between the object coordinate system on the left and the image or device coordinate system on the right. It is called the *world coordinate system*, representing where we place transformed objects to compose the picture we want to draw. The example in the box shows two triangles: the one on the right is a scaled copy of the original that is moved up and to the right, the one on the left is another scaled copy of the original that is rotated 90° counterclockwise around the origin of the coordinate system and then moved up and to the right in the same way.

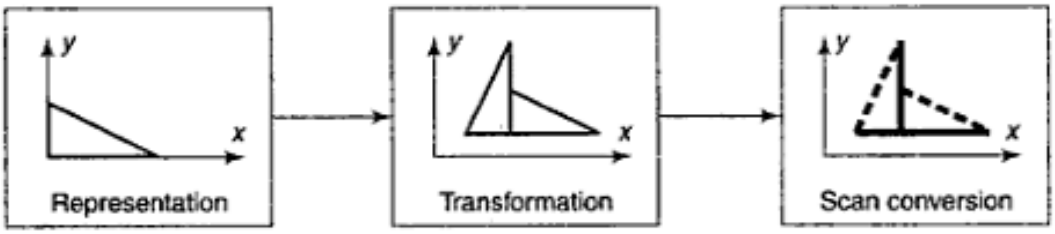


Fig. 1.2 *A Simple Graphics Pipeline*

In a typical implementation of the graphics pipeline we would write our application program in a host programming language and call library subroutines to perform graphics operations. Some subroutines are used to prescribe, among other things, transformation parameters. Others are used to draw, i.e. to feed original data into the pipeline so current system settings are automatically applied to shape the end product coming out of the pipeline, which is the picture on the screen.

Having looked at the key ingredients of what is called two-dimensional graphics, we now turn our attention to three-dimensional graphics. With the addition of a third dimension one should notice the profound distinction between an object and its picture. Figure 1.3 shows several possible ways to draw a cubic object, but none of the drawings even come close to being the object itself. The drawings simply represent projections of the three-dimensional object onto a two-dimensional display surface. This means that besides three-dimensional representation and transformation, we have an additional area of computer graphics that covers projection methods.

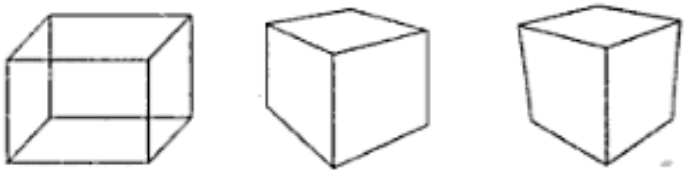


Fig. 1.3 *Several Ways to Depict a Cube*

Did you notice that each drawing in Figure 1.3 shows only three sides of the cubic object? Being a solid three-dimensional object the cube has six plane surfaces. However, we depict it as if we were looking at it in real life. We only draw the surfaces that are visible to us. Surfaces that are obscured from our eyesight are not shown. The area of computer graphics that deals with this computational task is called *hidden surface removal*. Adding projection and hidden surface removal to our simple graphics pipeline, right after transformation but before scan conversion, results in a prototype for three-dimensional graphics.

Now let's follow up on the idea that we want to produce a picture of an object in real-life fashion. This presents a great challenge for computer graphics, since there is an extremely effective way to produce such a picture: photography. In order to generate a picture that is photo-realistic, i.e. that looks as good as a photograph, we need to explore how a camera and nature work together to produce a snapshot.

When a camera is used to photograph a real-life object illuminated by a light source, light energy coming out of the light source gets reflected from the object surface through the camera lens onto the negative, forming an image of the object. Generally, the part of the object that is closer to the light source should appear brighter in the picture than the part that is further away, and the part of the object that is facing away from the light source should appear relatively dark. Figure 1.4 shows a computer-generated image that depicts two spherical objects illuminated by a light source that is located somewhere between the spheres and the "camera" at about the ten to eleven o'clock position. Although both spheres have gradual shadings, the bright spot on the large sphere looks like a reflection of the light source and hence suggests a difference in their reflectance property (the large sphere being shinier than the small one). The mathematical formulae that mimic this type of optical phenomenon are referred to as local illumination models, for the energy coming directly from the light source to a particular object surface is not a full account of the energy

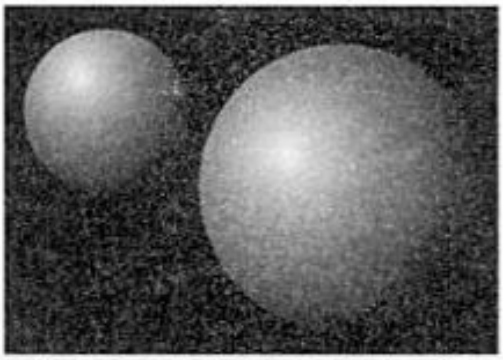


Fig. 1.4 *Two Shaded Spheres*

arriving at that surface. Light energy is also reflected from one object surface to another, and it can go through a transparent or translucent object and continue on to other places. Computational methods that strive to provide a more accurate account of light transport than local illumination models are referred to as *global illumination models*.

Now take a closer look at Fig. 1.4. The two objects seem to have super-smooth surfaces. What are they made of? How can they be so perfect? Do you see many physical objects around you that exhibit such surface characteristics? Furthermore, it looks like the small sphere is positioned between the light source and the large sphere. Shouldn't we see its shadow on the large sphere? In computer graphics the surface shading variations that distinguish a wood surface from a marble surface or other types of surface are referred to as *surface textures*. There are various techniques to add surface textures to objects to make them look more realistic. On the other hand, the computational task to include shadows in a picture is called *shadow generation*.

Before moving on for a closer look at each of the subject areas we have introduced in this mini-survey, we want to briefly discuss a couple of allied fields of computer science that also deal with graphical information.

Image Processing

The key element that distinguishes image processing (or digital image processing) from computer graphics is that image processing generally begins with images in the image space and performs pixel-based operations on them to produce new images that exhibit certain desired features. For example, we may reset each pixel in the image displayed on the monitor screen in Fig. 1.1 to its complementary color (e.g. black to white and white to black), turning a dark triangle on a white background to a white triangle on a dark background, or vice versa. While each of these two fields has its own focus and strength, they also overlap and complement each other. In fact, stunning visual effects are often achieved by using a combination of computer graphics and image processing techniques.

User Interface

While the main focus of computer graphics is the production of images, the field of user interface promotes effective communication between man and machine. The two fields join forces when it comes to such areas as graphical user interfaces. There are many kinds of physical devices that can

be attached to a computer for the purpose of interaction, starting with the keyboard and the mouse. Each physical device can often be programmed to deliver the function of various logical devices (e.g. Locator, Choice—see below). For example, a mouse can be used to specify locations in the image space (acting as a Locator device). In this case a cursor is often displayed as visual feedback to allow the user see the locations being specified. A mouse can also be used to select an item in a pull-down or pop-up manual (acting as a Choice device). In this case it is the identification of the selected manual item that counts and the item is often highlighted as a whole (the absolute location of the cursor is essentially irrelevant). From these we can see that a physical device may be used in different ways and information can be conveyed to the user in different graphical forms. The key challenge is to design interactive protocols that make effective use of devices and graphics in a way that is user-friendly—easy, intuitive, efficient, etc.

1.2 OVERVIEW OF IMAGE REPRESENTATION

A digital image, or image for short, is composed of discrete pixels or picture elements. These pixels are arranged in a row-and-column fashion to form a rectangular picture area, sometimes referred to as a raster. Clearly the total number of pixels in an image is a function of the size of the image and the number of pixels per unit length (e.g. inch) in the horizontal as well as the vertical direction. This number of pixels per unit length is referred to as the resolution of the image. Thus, a 3×2 inch image at a resolution of 300 pixels per inch would have a total of 540,000 pixels.

Frequently image size is given as the total number of pixels in the horizontal direction times the total number of pixels in the vertical direction (e.g. 512×512, 640×480, or 1024×768). Although this convention makes it relatively straightforward to gauge the total number of pixels in an image, it does not specify the size of the image or its resolution, as defined in the paragraph above. A 640×480 image would measure 62/3l inches by 5 inches when presented (e.g. displayed or printed) at 96 pixels per inch. On the other hand, it would measure 1.6 inches $\times$ 1.2 inches at 400 pixels per inch.

The ratio of an image's width to its height, measured in unit length or number of pixels, is referred to as its aspect ratio. Both a 2×2 inch image and a 512×512 image have an aspect ratio of 1/1, whereas both a $6 \times 41/2$ inch image and a 1024×768 image have an aspect ratio of 4/3.

Individual pixels in an image can be referenced by their coordinates. Typically the pixel at the lower left corner of an image is considered to be at the origin (0, 0) of a pixel coordinate system. Thus the pixel at the lower right corner of a 640×480 image would have coordinates (639, 0), the pixel at the upper left corner would have coordinates (0, 479), and the pixel at the upper right corner would have coordinates (639, 479).

1.3 THE RGB COLOR MODEL

Color is a complex, interdisciplinary subject spanning from physics to psychology. In this section we only introduce the basics of the most widely used color representation method in computer graphics. We will have detailed discussion on this topic later in another chapter.

Figure 1.5 shows a color coordinate system with three primary colors: R (red), G (green), and B (blue). Each primary color can take on an intensity value ranging from 0 (off—lowest) to 1

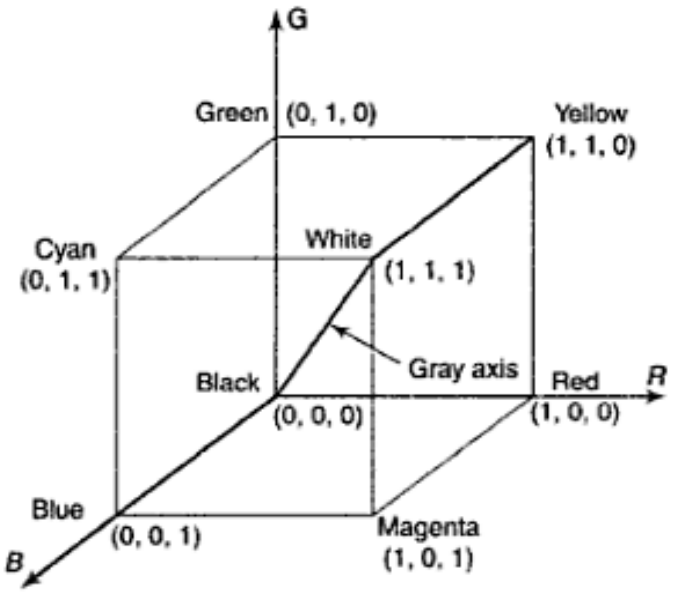


Fig. 1.5 *The RGB Color Space*

(on—highest). Mixing these three primary colors at different intensity levels produces a variety of colors. The collection of all the colors obtainable by such a linear combination of red, green, and blue forms the cube-shaped RGB color space. The corner of the RGB color cube that is at the origin of the coordinate system corresponds to black, whereas the corner of the cube that is diagonally opposite to the origin represents white. The diagonal line connecting black and white corresponds to all the gray colors between black and white. It is called the gray axis.

Given this RGB color model an arbitrary color within the cubic color space can be specified by its color coordinates: (r, g, b). For example, we have (0, 0, 0) for black, (1, 1, 1) for white, (1, 1, 0) for yellow, etc. A gray color at (0.7, 0.7, 0.7) has an intensity halfway between one at (0.9, 0.9, 0.9) and one at (0.5, 0.5, 0.5).

Color specification using the RGB model is an additive process. We begin with black and add on the appropriate primary components to yield a desired color. This closely matches the working principles of the display monitor (see Section 1.6). On the other hand, there is a complementary color model, called the CMY color model that defines colors using a subtractive process, which closely matches the working principles of the printer (see Section 1.7).

In the CMY model we begin with white and take away the appropriate primary components to yield a desired color. For example, if we subtract red from white, what remains consists of green and blue, which is cyan. Looking at this from another perspective, we can use the amount of cyan, the complementary color of red, to control the amount of red, which is equal to one minus the amount of cyan. Figure 1.6 shows a coordinate system using the three primaries' complementary colors: C (cyan), M (magenta), and Y (yellow). The corner of the CMY color cube that is at (0, 0, 0) corresponds to white, whereas the corner of the cube that is at (1, 1, 1) represents black (no red, no green, no blue). The following formulae summarize the conversion between the two color models:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix} \qquad \begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

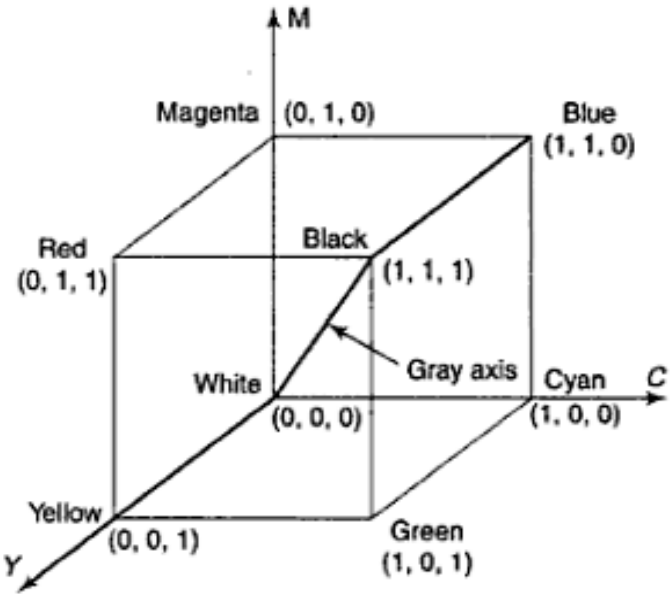


Fig. 1.6 *The CMY Color Space*

1.4 DIRECT CODING

Image representation is essentially the representation of pixel colors. Using direct coding we allocate a certain amount of storage space for each pixel to code its color. For example, we may allocate 3 bits for each pixel, with one bit for each primary color (see Table 1.1). This 3-bit representation allows each primary to vary independently between two intensity levels: 0 (off) or 1 (on). Hence each pixel can take on one of the eight colors that correspond to the corners of the RGB color cube.

Table 1.1 *Direct Coding of Colors using 3 bits*

bit 1 r	*bit 2 g*	*bit 3 b*	*color name*
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

A widely accepted industry standard uses 3 bytes, or 24 bits, per pixel, with one byte for each primary color. This way we allow each primary color to have 256 different intensity levels, corresponding to binary values from 00000000 to 11111111. Thus a pixel can take on a color from $256 \times 256 \times 256$ or 16.7 million possible choices. This 24-bit format is commonly referred to as

the true color representation, for the difference between two colors that differ by one intensity level in one or more of the primaries is virtually undetectable under normal viewing conditions. Hence a more precise representation involving more bits is of little use in terms of perceived color accuracy.

A notable special case of direct coding is the representation of black-and-white (bi-level) and gray-scale images, where the three primaries always have the same value and hence need not be coded separately. A black-and-white image requires only one bit per pixel, with bit value 0 representing black and 1 representing white. A gray-scale image is typically coded with 8 bits per pixel to allow a total of 256 intensity or gray levels.

Although this direct coding method features simplicity and has supported a variety of applications, we can see a relatively high demand for storage space when it comes to the 24-bit standard. For example, a 1000 × 1000 true color image would take up three million bytes. Furthermore, even if every pixel in that image had a different color, there would only be one million colors in the image. In many applications the number of colors that appear in any one particular image is much less. Therefore the 24-bit representation's ability to have 16.7 million different colors appear simultaneously in a single image seems to be somewhat overkill.

1.5 LOOKUP TABLE

Image representation using a lookup table can be viewed as a compromise between the desire to have a lower storage requirement and the need to support a reasonably sufficient number of simultaneous colors. In this approach pixel values do not code colors directly. Instead, they are addresses or indices into a table of color values. The color of a particular pixel is determined by the color value in the table entry that the value of the pixel references.

Figure 1.7 shows a lookup table with 256 entries. The entries have addresses 0 through 255. Each entry contains a 24-bit RGB color value. Pixel values are now 1-byte, or 8-bit, quantities. The color of a pixel whose value is i, where $0 < i < 255$, is determined by the color value in the

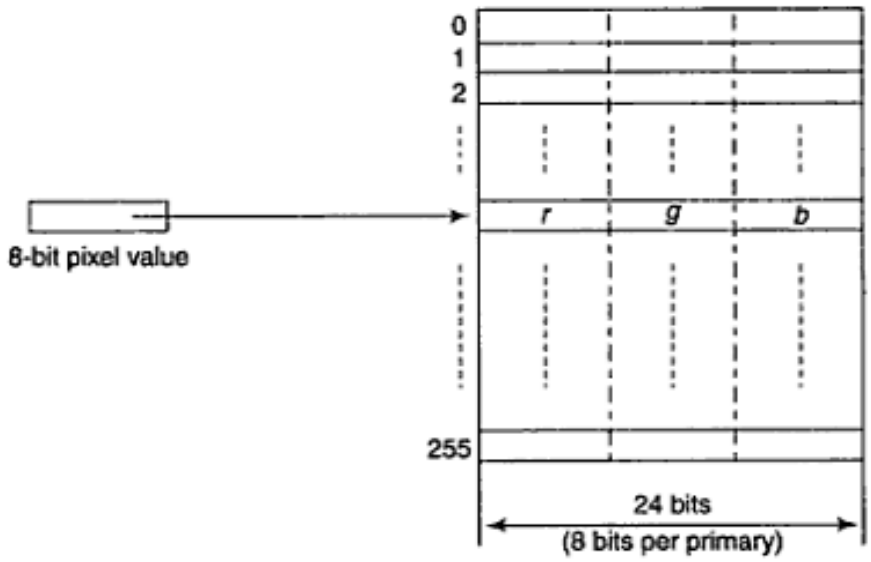


Fig. 1.7 *A 24-bit 256-entry Lookup Table*

table entry whose address is *i*. This 24-bit 256-entry lookup table representation is often referred to as the 8-bit format. It reduces the storage requirement of a 1000 × 1000 image to one million bytes plus 768 bytes for the color values in the lookup table. It allows 256 simultaneous colors that are chosen from 16.7 million possible colors.

It is important to remember that, using the lookup table representation, an image is defined not only by its pixel values but also by the color values in the corresponding lookup table. Those color values form a *color map* for the image.

1.6 DISPLAY MONITOR

Among the numerous types of image presentation or output devices that convert digitally represented images into visually perceivable pictures is the display or video monitor.

We first take a look at the working principle of a monochromatic display monitor, which consists mainly of a Cathode Ray Tube (CRT) along with related control circuits. The CRT is a vacuum glass tube with the display screen at one end and connectors to the control circuits at the other (see Fig. 1.8). Coated on the inside of the display screen is a special material, called phosphor, which emits light for a period of time when hit by a beam of electrons. The color of the light and the time period vary from one type of phosphor to another. The light given off by the phosphor during exposure to the electron beam is known as *fluorescence*, the continuing glow given off after the beam is removed is known as *phosphorescence*, and the duration of phosphorescence is known as the phosphor's *persistence*.

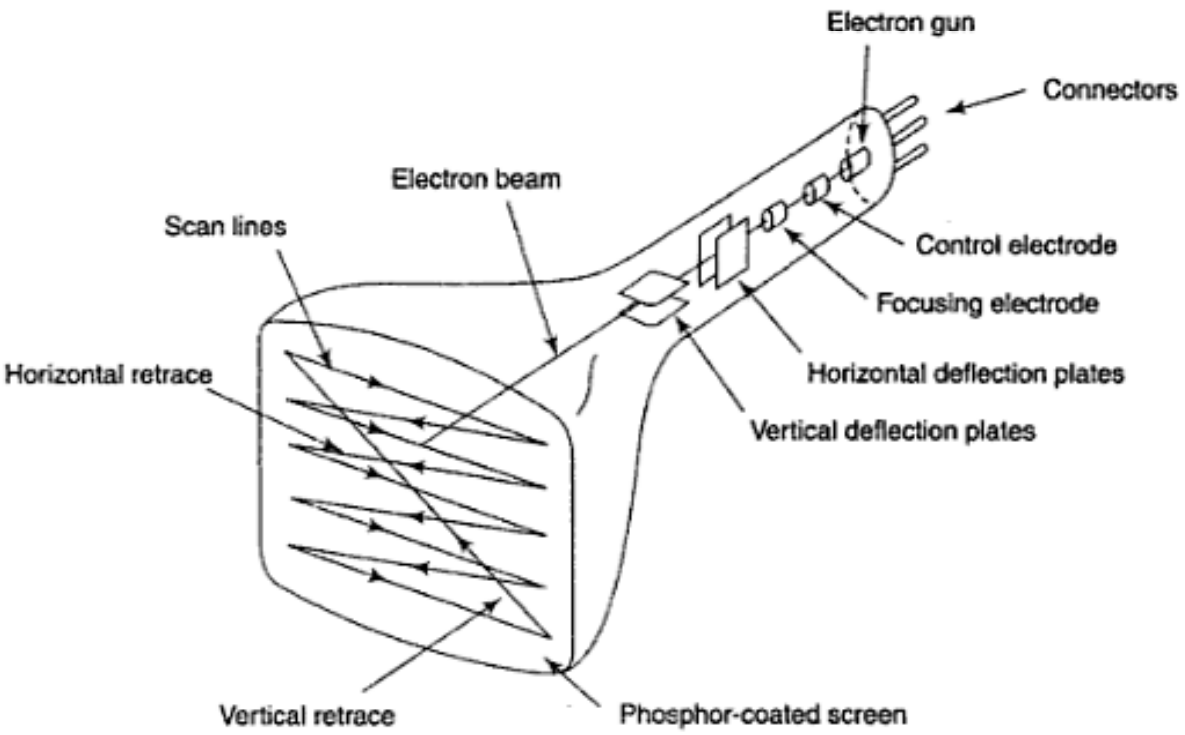


Fig. 1.8 *Anatomy of a Monochromatic CRT*

Opposite to the phosphor-coated screen is an electron gun that is heated to send out electrons. The electrons are regulated by the control electrode and forced by the focusing electrode into a narrow beam striking the phosphor coating at small spots. When this electron beam passes through the horizontal and vertical deflection plates, it is bent or deflected by the electric fields between the plates. The horizontal plates control the beam to scan from left to right and retrace from right to left. The vertical plates control the beam to go from the first scan line at the top to the last scan line at the bottom and retrace from the bottom back to the top. These actions are synchronized by the control circuits so that the electron beam strikes each and every pixel position in a scan line by scan line fashion. As an alternative to this electrostatic deflection method, some CRTs use magnetic deflection coils mounted on the outside of the glass envelope to bend the electron beam with magnetic fields.

The intensity of the light emitted by the phosphor coating is a function of the intensity of the electron beam. The control circuits shut off the electron beam during horizontal and vertical retraces. The intensity of the beam at a particular pixel position is determined by the intensity value of the corresponding pixel in the image being displayed.

The image being displayed is stored in a dedicated system memory area that is often referred to as the frame buffer or refresh buffer. The control circuits associated with the frame buffer generate proper video signals for the display monitor. The frequency at which the content of the frame buffer is sent to the display monitor is called the refreshing rate, which is typically 60 times or frames per second (60 Hz) or higher. A determining factor here is the need to avoid flicker, which occurs at lower refreshing rates when our visual system is unable to integrate the light impulses from the phosphor dots into a steady picture. The persistence of the monitor's phosphor, on the other hand, needs to be long enough for a frame to remain visible but short enough for it to fade before the next frame is displayed.

Some monitors use a technique called interlacing to "double" their refreshing rate. In this case only half of the scan lines in a frame is refreshed at a time, first the odd numbered lines, then the even numbered lines. Thus the screen is refreshed from top to bottom in half the time it would have taken to sweep across all the scan lines. Although this approach does not really increase the rate at which the entire screen is refreshed, it is quite effective in reducing flicker.

Color Display

Moving on to color displays there are now three electron guns instead of one inside the CRT (see Figure 1.9), with one electron gun for each primary color. The phosphor coating on the inside of the display screen consists of dot patterns of three different types of phosphors. These phosphors are capable of emitting red, green, and blue light, respectively. The distance between the center of the dot patterns is called the pitch of the color CRT. It places an upper limit on the number of addressable positions on the display area. A thin metal screen called a shadow mask is placed between the phosphor coating and the electron guns. The tiny holes on the shadow mask constrain each electron beam to hit its corresponding phosphor dots. When viewed at a certain distance, light emitted by the three types of phosphors blends together to give us a broad range of colors.

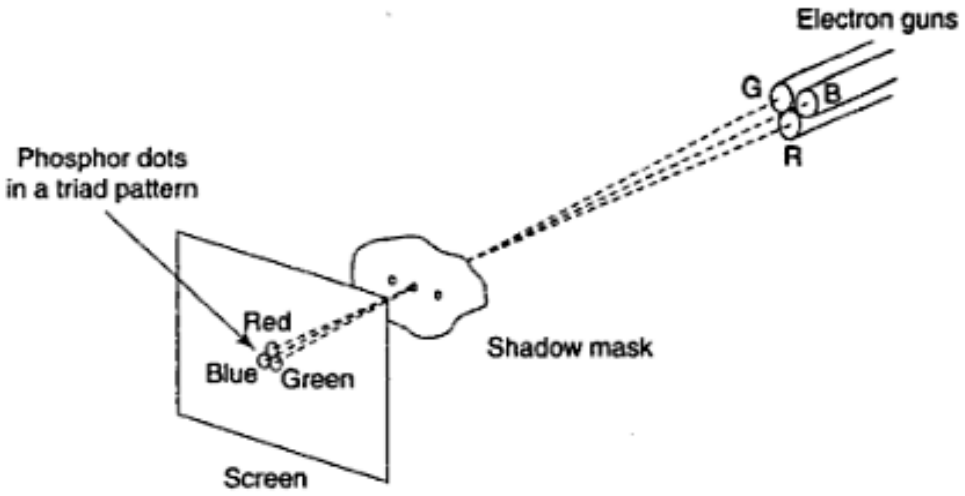


Fig. 1.9 *Color CRT using a Shadow Mask*

1.7 PRINTER

Another typical image presentation device is the printer. A printer deposits color pigments onto a print media, changing the light reflected from its surface and making it possible for us to see the print result.

Given the fact that the most commonly used print media is a piece of white paper, we can in principle utilize three types of pigments (cyan, magenta, and yellow) to regulate the amount of red, green, and blue light reflected to yield all RGB colors (see Section 1.3). However, in practice, an additional black pigment is often used due to the relatively high cost of color pigments and the technical difficulty associated with producing high-quality black from several color pigments.

While some printing methods allow color pigments to blend together, in many cases the various color pigments remain separate in the form of tiny dots on the print media. Furthermore, the pigments are often deposited with a limited number of intensity levels. There are various techniques to achieve the effect of multiple intensity levels beyond what the pigment deposits can offer. Most of these techniques can also be adapted by the display devices that we have just discussed in the previous section.

Halftoning

Let's first take a look at a traditional technique called halftoning from the printing industry for bilevel devices. This technique uses variably sized pigment dots that, when viewed from a certain distance, blend with the white background to give us the sensation of varying intensity levels. These dots are arranged in a pattern that forms a 45° screen angle with the horizon (see Fig. 1.10 where the dots are enlarged for illustration). The size of the dots is inversely proportional to the intended intensity level. When viewed at a far enough distance, the stripe in Fig. 1.10 exhibits a gradual shading from white (high intensity) on the left to black (low intensity) on the right. An image produced using this technique is called a halftone. In practice, newspaper halftones use 60 to 80 dots per inch (dpi), whereas book and magazine halftones use 120 to 200 dots per inch.

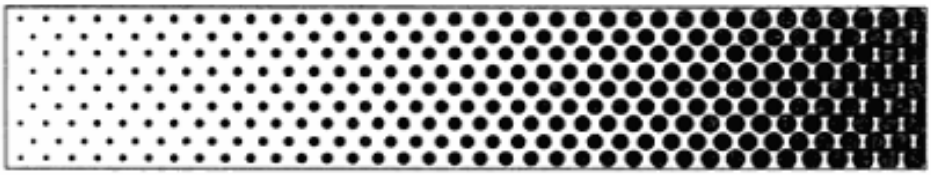


Fig. 1.10 *A Halftone Stripe*

Halftone Approximation

Instead of changing dot size we can approximate the halftone technique using pixel-grid patterns. For example, with a 2 × 2 bi-level pixel grid we can construct five grid patterns to produce five overall intensity levels (see Fig. 1.11). We can increase the number of overall intensity levels by increasing the size of the pixel grid (see the following paragraphs for an example). On the other hand, if the pixels can be set to multiple intensity levels, even a 2 × 2 grid can produce a relatively high number of overall intensity levels. For example, if the pixels can be intensified to four different levels, we can follow the pattern sequence in Fig. 1.11 to bring each pixel from one intensity level to the next to approximate a total of thirteen overall intensity levels (one for all pixels off and four for each of the three non-zero intensities, see Fig. 1.12).

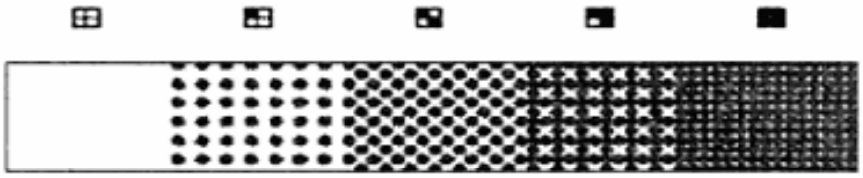


Fig. 1.11 *Halftone Approximation*

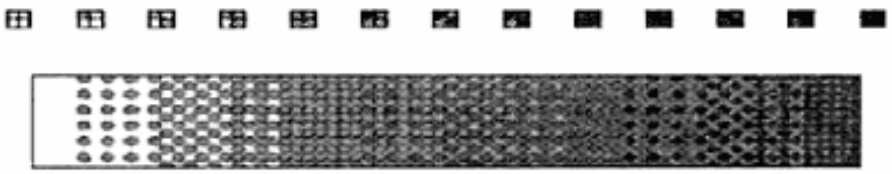


Fig. 1.12 *Halftone Approximation with 13 Intensity Levels*

These halftone grid patterns are sometimes referred to as dither patterns. There are several considerations in the design of dither patterns. First, the pixels should be intensified in a growth-from-the-grid-center fashion in order to mimic the growth of dot size. Second, a pixel that is intensified to a certain level to approximate a particular overall intensity should remain at least at that level for all subsequent overall intensity levels. In other words, the patterns should evolve from one to the next in order to minimize the differences in the patterns for successive overall intensity levels. Third, symmetry should be avoided in order to minimize visual artifacts such as streaks that would show up in image areas of uniform intensity. Fourth, isolated "on" pixels should be avoided since they are sometimes hard to reproduce.

We can use a dither matrix to represent a series of dither patterns. For example, the following 3×3 matrix:

$$\begin{pmatrix} 5 & 2 & 7 \\ 1 & 0 & 3 \\ 6 & 8 & 4 \end{pmatrix}$$

represents the order in which pixels in a 3×3 grid are to be intensified. For bi-level reproduction, this gives us ten intensity levels from level 0 to level 9, and intensity level I is achieved by turning on all pixels that correspond to values in the dither matrix that are less than I. If each pixel can be intensified to three different levels, we follow the order denned by the matrix to set the pixels to their middle intensity level and then to their high intensity level to approximate a total of nineteen overall intensity levels.

This halftone approximation technique is readily applicable to the reproduction of color images. All we need is to replace the dot at each pixel position with an RGB or CMY dot pattern (e.g. the triad pattern shown in Fig. 1.9). If we use a 2×2 pixel grid and each primary or its complement can take on two intensity levels, we achieve a total of $5 \times 5 \times 5 = 125$ color combinations.

At this point we can turn to the fact that halftone approximation is a technique that trades spatial resolution for more colors/intensity levels. For a device that is capable of producing images at a resolution of 400×400 pixels per inch, halftone approximation using 2×2 dither patterns would mean lowering its resolution effectively to 200×200 pixels per inch.

Dithering

A technique called dithering can be used to approximate halftones without reducing spatial resolution. In this approach the dither matrix is treated very much like a floor tile that can be repeatedly positioned one copy next to another to cover the entire floor, i.e. the image. A pixel at (x, y) is intensified if the intensity level of the image at that position is greater than the corresponding value in the dither matrix. Mathematically, if D_n stands for an $n \times n$ dither matrix, the element $D_n(i, j)$ that corresponds to pixel position (x, y) can be found by $i = x \bmod n$ and $j = y \bmod n$. For example, if we use the 3×3 matrix given earlier for a bilevel reproduction and the pixel of the image at position (2, 19) has intensity level 5, then the corresponding matrix element is $D_3(2, 1) = 3$, and hence a dot should be printed or displayed at that location.

It should be noted that, for image areas that have constant intensity, the results of dithering are exactly the same as the results of halftone approximation. Reproduction differences between these two methods occur only when intensity varies.

Error Diffusion

Another technique for continuous-tone reproduction without sacrificing spatial resolution is called the Floyd-Steinberg error diffusion. Here a pixel is printed using the closest intensity that the device can deliver. The error term, i.e. the difference between the exact pixel value and the

approximated value in the reproduction is then propagated to several yet-to-be-processed neighboring pixels for compensation. More specifically, let S be the source image that is processed in a left-to-right and top-to-bottom pixel order, $S(x, y)$ be the pixel value at location (x, y), and e be $S(x, y)$ minus the approximated value. We update the value of the pixel's four neighbors (one to its right and three in the next scan line) as follows:

$$S(x + 1, y) = S(x + 1, y) + ae$$
$$S(x - 1, y - 1) = S(x - 1, y - 1) + be$$
$$S(x, y - 1) = S(x, y - 1) + ce$$
$$S(x + 1, y - 1) = S(x + 1, y - 1) + de$$

where parameters a through d often take values 7/16, 3/16, 5/16, and 1/16 respectively. These modifications are for the purpose of using the neighboring pixels to offset the reproduction error at the current pixel location. They are not permanent changes made to the original image.

Consider, for example, the reproduction of a gray scale image (0: black, 255: white) on a bi-level device (level 0: black, level 1: white). If a pixel whose current value is 96 has just been mapped to level 0, we have $e = 96$ for this pixel location. The value of the pixel to its right is now increased by $96 \times 7/16 = 42$ in order to determine the appropriate reproduction level. This increment tends to cause such neighboring pixel to be reproduced at a higher intensity level, partially compensating the discrepancy brought on by mapping value 96 to level 0 (which is lower than the actual pixel value) at the current location. The other three neighboring pixels (one below and to the left, one immediately below, and one below and to the right) receive 18, 30, and 6 as their share of the reproduction error at the current location, respectively.

Results produced by this error diffusion algorithm are generally satisfactory, with occasional introduction of slight echoing of certain image parts. Improved performance can sometimes be obtained by alternating scanning direction between left-to-right and right-to-left (minor modifications need to be made to the above formulas).

1.8 IMAGE FILES

A digital image is often encoded in the form of a binary file for the purpose of storage and transmission. Among the numerous encoding formats are BMP (Windows Bitmap), JPEG (Joint Photographic Experts Group File Interchange Format), and TIFF (Tagged Image File Format). Although these formats differ in technical details, they share structural similarities.

Figure 1.13 shows the typical organization of information encoded in an image file. The file consists largely of two parts: header and image data. In the beginning of the file header a binary code or ASCII string identifies the format being used, possibly along with the version number. The width and height of the image are given in numbers of pixels. Common image types include black and white (1 bit per pixel), 8-bit gray scale (256 levels along the gray axis), 8-bit color (lookup table), and 24-bit color. Image data format specifies the order in which pixel values are stored in the image data section. A commonly used order is left to right and top to bottom. Another possible order is left to right and bottom to top. Image data format also specifies if the RGB values in the color map or in the image are interlaced. When the values are given in an interlaced fashion, the

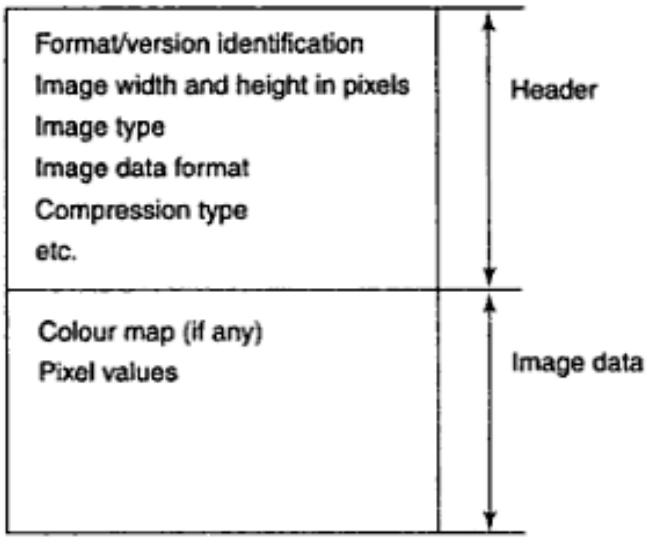


Fig. 1.13 *Typical Image File Format*

three primary color components for a particular lookup table entry or a particular pixel stay together consecutively, followed by the three color components for the next entry or pixel. Thus the color values in the image data section are a sequence of red, green, blue, red, green, blue, etc. When the values are given in a non-interlaced fashion, the values of one primary for all table entries or pixels appear first, then the values of another primary, followed by the values of the third primary. Thus the image data are in the form of red, red, ..., green, green, ..., blue, blue, ...

The values in the image data section may be compressed, using such compression algorithms as run-length encoding (RLE). The basic idea behind RLE can be illustrated with a character string "xxxxxxyyzzzz", which takes 12 bytes of storage. Now if we scan the string from left to right for segments of repeating characters and replace each segment by a 1-byte repeat count followed by the character being repeated, we convert or compress the given string to "6x2y4z", which takes only 6 bytes. This compressed version can be expanded or decompressed by repeating the character following each repeat count to recover the original string.

The length of the file header is often fixed, for otherwise it would be necessary to include length information in the header to indicate where image data starts (some formats include header length anyway). The length of each individual component in the image data section is, on the other hand, dependent on such factors as image type and compression type. Such information, along with additional format-specific information, can also be found in the header.

1.9 SETTING THE COLOR ATTRIBUTES OF PIXELS

Setting the color attributes of individual pixels is arguably the most primitive graphics operation. It is typically done by making system library calls to write the respective values into the frame buffer. An aggregate data structure, such as a three-element array, is often used to represent the three primary color components. Regardless of image type (direct coding versus lookup table), there are two possible protocols for the specification of pixel coordinates and color values.

In one protocol the application provides both coordinate information and color information simultaneously. Thus a call to set the pixel at location (x, y) in a 24-bit image to color (r, g, b) would look like

setPixel(x, y, rgb)

where rgb is a three-element array with $rgb[0] = r$, $rgb[1] = g$, and $rgb[2] = b$. On the other hand, if the image uses a lookup table then, assuming that the color is defined in the table, the call would look like

setPixel(x, y, i)

where i is the address of the entry containing (r, g, b).

Another protocol is based on the existence of a current color, which is maintained by the system and can be set by calls that look like

setColor(rgb)

for direct coding, or

setColor(i)

for the lookup table representation. Calls to set pixels now need only to provide coordinate information and would look like

setPixel(x, y)

for both image types. The graphics system will automatically use the most recently specified current color to carry out the operation.

Lookup table entries can be set from the application by a call that looks like

setEntry(i, rgb)

which puts color (r, g, b) in the entry whose address is i. Conversely, values in the lookup table can be read back to the application with a call that looks like

getEntry(i, rgb)

which returns the color value in entry i in array rgb.

There are sometimes two versions of the calls that specify RGB values. One takes RGB values as floating point numbers in the range of [0.0, 1.0], whereas the other takes them as integers in the range of [0, 255]. Although the floating point version is handy when the color values come from some continuous formula, the floating point values are mapped by the graphics system into integer values before being written into the frame buffer.

In order to provide basic support for pixel-based image-processing operations there are calls that look like

getPixel(x, y, rgb)

for direct coding

or

getPixel(x, y, i)

for the lookup table representation to return the color or index value of the pixel at (x, y) back to the application.

There are also calls that read and write rectangular blocks of pixels. A useful example would be a call to set all pixels to a certain background color. Assuming that the system uses a current color we would first set the current color to be the desired background color, and then make a call that looks like

clear()

to achieve the goal.

1.10 EXAMPLE: VISUALIZING THE MANDELBROT SET

An elegant and illustrative example showing the construction of beautiful images by setting the color attributes of individual pixels directly from the application is the visualization of the Mandelbrot set. This remarkable set is based on the following transformation:

$$x_{n+1} = x_n^2 + z$$

where both x and z represent complex numbers. For readers who are unfamiliar with complex numbers it suffices to know that a complex number is defined in the form of $a + bi$. Here both a and b are real numbers; a is called the real part of the complex number and b the imaginary part (identified by the special symbol i). The magnitude of $a + bi$, denoted by $|a + bi|$, is equal to the square root of $a^2 + b^2$. The sum of two complex numbers $a + bi$ and $c + di$ is defined to be $(a + c) + (b + d)i$. The product of $a + bi$ and $c + di$ is defined to be $(ac - bd) + (ad + bc)i$. Thus the square of $a + bi$ is equal to $(a^2 - b^2) + 2abi$. For example, the sum of $0.5 + 2.0i$ and $1.0 - 1.0i$ is $1.5 + 1.0i$. The product of the two is $2.5 + 1.5i$. The square of $0.5 + 2.0i$ is $-3.75 + 2.0i$ and the square of $1.0 - 1.0i$ is $0.0 - 2.0i$.

The Mandelbrot set is the set of complex numbers z that do not diverge under the above transformation with $x_0 = 0$ (both the real and imaginary parts of x_0 are 0). In other words, to determine if a particular complex number z is a member of the set, we begin with $x_0 = 0$, followed by $x_1 = x_0^2 + z$, $x_2 = x_1^2 + z$, ..., $x_{n+1} = x_n^2 + z$, ... If $|x|$ goes towards infinity when n increases, then z is not a member. Otherwise, z belongs to the Mandelbrot set.

Figure 1.14 shows how to produce a discrete snapshot of the Mandelbrot set. On the left hand side is the complex plane where the horizontal axis, Re, measures the real part of complex numbers and the vertical axis, Im, measures the imaginary part. Hence an arbitrary complex number z corresponds to a point in the complex plane. Our goal is to produce an image of width by height (in numbers of pixels) that depicts the z values in a rectangular area defined by (Re_min, Im_min) and (Re_max, Im_max). This rectangular area has the same aspect ratio as the image so as not to introduce geometric distortion. We subdivide the area to match the pixel grid in the image. The color of a pixel, shown as a little square in the pixel grid, is determined by the complex

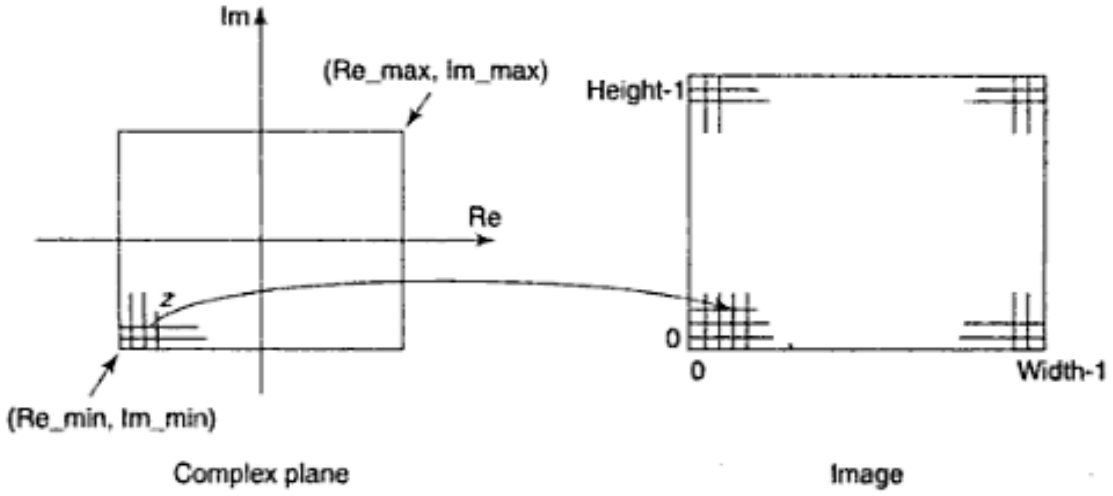


Fig. 1.14 *Visualizing the Mandelbrot Set*

number z that corresponds to the lower left corner of the little square. Although only width × height points in the complex plane are used to compute the image, this relatively straightforward approach to discrete sampling produces reasonably good approximations for the purpose of visualizing the set.

There are many ways to decide the color of a pixel based on the corresponding complex number z. What we do here is to produce a gray scale image where the gray level of a non-black pixel represents proportionally the number of iterations it takes for $|x|$ to be greater than 2. We use 2 as a threshold for divergence because x diverges quickly under the given transformation once $|x|$ becomes greater than 2. If $|x|$ remains less than or equal to 2 after a preset maximum number of iterations, we simply set the pixel value to 0 (black).

The following pseudo-code implements what we have discussed in the above paragraphs. We use TV to represent the maximum number of iterations, *z.real* the real part of z, and *z.imag* the imaginary part of z. We also assume a 256-entry gray scale lookup table where the color value in entry i is (i, i, i). The formula in the second call to setColor is to obtain a proportional mapping from $[0, N]$ to $[1, 255]$:

```
int i, j, count;
float delta = (Re_max - Re_min)/width;
for (i = 0, z.real = Re_min; i < width; i + + , z.real + = delta)
for (j = 0, z.imag = Im_min;y < height; j + + , z.imag + = delta) {
  count = 0;
  complex number x = 0;
  while (|x| <= 2.0 && count < N) {
        compute x = x^2 + z;
        count + + ;
  }
  if (|x| <= 2.0) setColor(0);
  else setColor(1 + 254*count/N);
  setPixel(i , j);
```

The image in Fig. 1.15 shows what is nicknamed the Mandelbrot bug. It visualizes an area where—2.0 < *z. real* < 0.5 and –1.25 < *z. imag* < 1.25 with $N = 64$. Most z values that are outside the area lead x to diverge quickly, whereas the z values in the black region belong to the Mandelbrot set. It is along the contour of the bug-like figure that we see the most dynamic alterations between divergence and non-divergence, together with the most significant variations in the number of iterations used in the divergence test. The brighter a pixel, the longer it takes to conclude divergence for the corresponding z. In principle the rectangular area can be reduced indefinitely to zoom in on any active region to show more intricate details.

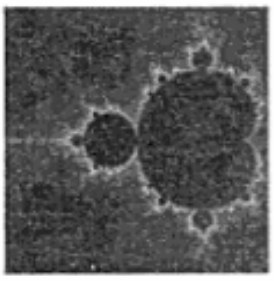


Fig. 1.15 *The Mandelbrot Set*

Julia Sets

Now if we set z to some fixed non-zero value and vary x_0 across the complex plane, we obtain a set of non-divergence numbers (values of x_0 that do not diverge under the given transformation) that

form a Julia set. Different z values lead to different Julia sets. The image in Fig. 1.16 is produced by making slight modifications to the pseudo-code for the Mandelbrot set. It shows the Julia set defined by $z = -0.74543 + 0.11301i$ with $-1.2 < x_o \cdot real < 1.2$, $-1.2 < x_o \cdot imag < 1.2$, and $N = 128$.

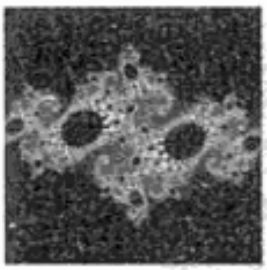


Fig. 1.16 *A Julia Set*

1.11 WHAT'S AHEAD

We hope that our brief flight over the landscape of the graphics kingdom has given you a good impression of some of the important landmarks and made you eager to further your exploration. The following chapters are dedicated to the various subject areas of computer graphics. Each chapter begins with the necessary background information (e.g. context and terminology) and a summary account of the material to be discussed in subsequent sections.

We strive to provide clear explanation and inter-subject continuity in our presentation. Illustrative examples are used to substantiate discussion on abstract concepts. While the primary mission of this book is to offer a relatively well-focused introduction to the fundamental theory and underlying technology, significant variations in such matters as basic definitions and implementation protocols are presented in order to have a reasonably broad coverage of the field. In addition, interesting applications are introduced as early as possible to highlight the usefulness of the graphics technology and to encourage those who are eager to engage in hands-on practice.

Algorithms and programming examples are given in pseudo-code that resembles the C programming language, which shares similar syntax and basic constructs with other widely used languages such as C + + and Java. We hope that the relative simplicity of the C-style code presents little grammatical difficulty and hence makes it easy for you to focus your attention on the technical substance of the code.

There are numerous solved problems at the end of each chapter to help reinforce the theoretical discussion. Some of the problems represent computation steps that are omitted in the text and are particularly valuable for those looking for further details and additional explanation. Other problems may provide new information that supplements the main discussion in the text.

SOLVED PROBLEMS

1.1 What is the resolution of an image?

The number of pixels (i.e. picture elements) per unit length (e.g. inch) in the horizontal as well as vertical direction.

1.2 Compute the size of a 640 × 480 image at 240 pixels per inch.

640/240 × 480/240 or $2\frac{2}{3} \times 2$ inches.

1.3 Compute the resolution of a 2 × 2 inch image that has 512 × 512 pixels.

512/2 or 256 pixels per inch.

1.4 What is an image's aspect ratio?

The **ratio of its width to its height, measured in unit length or number of pixels.**

1.5 If an image has a height of 2 inches and an aspect ratio of 1.5, what is its width?

width = 1.5 × **height = 1.5 × 2 = 3 inches.**

1.6 If we want to resize a 1024 × 768 image to one that is 640 pixels wide with the same aspect ratio, what would be the height of the resized image?

height = 640 × **768/1024 = 480.**

1.7 If we want to cut a 512 × 512 sub-image out from the center of an 800 × 600 image, what are the coordinates of the pixel in the large image that is at the lower left corner of the small image?

[(800 – 512)/2, (600 – 512)/2] = (144, 44).

1.8 Sometimes the pixel at the upper left corner of an image is considered to be at the origin of the pixel coordinate system (a left-handed system). How to convert the coordinates of a pixel at (x, y) in this coordinate system into its coordinates (x', y') in the lower-left-corner-as-origin coordinate system (a right-handed system)?

$(x', y') = (x, m - y - 1)$ **where m is the number of pixels in the vertical direction.**

1.9 Find the CMY coordinates of a color at (0.2, 1, 0.5) in the RGB space.

(1 – 0.2, 1 – 1, 1 – 0.5) = (0.8, 0, 0.5).

1.10 Find the RGB coordinates of a color at (0.15, 0.75, 0) in the CMY space.

(1 – 0.15, 1 – 0.75, 1 – 0) = (0.85, 0.25, 1).

1.11 If we use direct coding of RGB values with 2 bits per primary color, how many possible colors do we have for each pixel?

$2^2 \times 2^2 \times 2^2 = 4 \times 4 \times 4 = 64$.

1.12 If we use direct coding of RGB values with 10 bits per primary color, how many possible colors do we have for each pixel?

$2^{10} \times 2^{10} \times 2^{10} = 1024^3 = 1073{,}741{,}824 > 1$ billion.

1.13 The direct coding method is flexible in that it allows the allocation of a different number of bits to each primary color. If we use 5 bits each for red and blue and 6 bits for green for a total of 16 bits per pixel, how many possible simultaneous colors do we have?

$2^5 \times 2^5 \times 2^6 = 2^{16} = 65{,}536$.

1.14 If we use 12-bit pixel values in a lookup table representation, how many entries does the lookup table have?

$2^{12} = 4096$.

1.15 If we use 2-byte pixel values in a 24-bit lookup table representation, how many bytes does the lookup table occupy?

$2^{16} \times 24/8 = 65{,}536 \times 3 = 196{,}608$.

1.16 State whether the following statement is true or false: Fluorescence is the term used to describe the light given off by a phosphor after it has been exposed to an electron beam. Explain your answer.

False. Phosphorescence is the correct term. Fluorescence refers to the light given off by a phosphor while it is being exposed to an electron beam.

1.17 What is persistence?

The duration of phosphorescence exhibited by a phosphor.

1.18 What is the function of the control electrode in a CRT?

Regulate the intensity of the electron beam.

1.19 Name the two methods by which an electron beam can be bent?

Electrostatic deflection and magnetic deflection.

1.20 What do you call the path the electron beam takes when returning to the left side of the CRT screen?

Horizontal retrace.

1.21 What do you call the path the electron beam takes at the end of each refresh cycle?

Vertical retrace.

1.22 What is the pitch of a color CRT?

The distance between the center of the phosphor dot patterns on the inside of the display screen.

1.23 Why do many color printers use black pigment?

Color pigments (cyan, magenta, and yellow) are relatively more expensive and it is technically difficult to produce high-quality black using several color pigments.

1.24 Show that with an $n \times n$ pixel grid, where each pixel can take on m intensity levels, we can approximate $n \times n \times (m - 1) + 1$ overall intensity levels.

Since the $n \times n$ pixels can be set to a non-zero intensity value one after another to produce $n \times n$ overall intensity levels, and there are $m - 1$ non-zero intensity levels for the individual pixels, we can approximate a total of $n \times n \times (m - 1)$ non-zero overall intensity levels. Finally we need to add one more overall intensity level that corresponds to zero intensity (all pixels off).

1.25 Represent the grid patterns in Fig. 1.11 with a dither matrix.

$$\begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}$$

1.26 What are the error propagation formulas for a top-to-bottom and right-to-left scanning order in the Floyd-Steinberg error diffusion algorithm?

$$S(x + 1, y) = S(x + 1, y) + ae$$
$$S(x - 1, y - 1) = S(x - 1, y - 1) + be$$
$$S(x, y - 1) = S(x, y - 1) + ce$$
$$S(x + 1, y - 1) = S(x + 1, y - 1) + de$$

1.27 What is RLE?

RLE stands for Run-Length Encoding, a technique used for image data compression.

1.28 Follow the illustrative example in the text to reconstruct the string that has been compressed to "981435" using RLE.

"8888888884555"

1.29 If an 8-bit gray scale image is stored uncompressed in sequential memory or in an image file in left-to-right and bottom-to-top pixel order, what is the offset or displacement of the byte for the pixel at (x, y) from the beginning of the memory segment or the file's image data section?

offset = $y \times n + x$ where n is the number of pixels in the horizontal direction.

1.30 What if the image in Problem 1.29 is stored in left-to-right and top-to-bottom order?

offset = $(m - y - l)\, n + x$ where n and m are the number of pixels in the horizontal and vertical direction, respectively.

1.31 Develop a pseudo-code segment to initialize a 24-bit 256-entry lookup table with gray-scale values.

```
int i, rgb[3];
for (i = 0; i < 256; i + + ) {
   rgb[0] = rgb[i] = rgb[2] = i;
   setEntry(i, rgb);
```

1.32 Develop a pseudo-code segment to swap the red and green components of all colors in a 256-entry lookup table.

```
int i, x, rgb[3];
for (i = 0; i < 256; i + + ) {
   getEntry(i, rgb);
   x = rgb[0];
   rgb[0] = rgb[i];
   rgb[i]=x;
   setEntry(i, rgb);
```

1.33 Develop a pseudo-code segment to draw a rectangular area *of wxh* (in number of pixels) that starts at (x, y) using color *rgb*.

```
int i, j;
setColor(rgb);
for (j =y;j <y + h; j + + )
   for (i = x; i < x + w; i + + ) setPixel(i, j);
```

1.34 Develop a pseudo-code segment to draw a triangular area with the three vertices at (x, y), $(x, y + t)$, and $(x + t, y)$, where integer $t > 0$, using color *rgb*.

```
int i, j;
setColor(rgb);
for (j = y; j <=y + t;j + + )
   for (/ = x; i <= x + y + t – j; i + + ) setPixel(i, j);
```

1.35 Develop a pseudo-code segment to reset every pixel in an image that is in the 24-bit 256-entry lookup table representation to its complementary color.

```
int i, rgb[3];
for (i = 0; i < 256; i + + ) {
   getEntry(i, rgb);
   rgb[0] = 255 – rgb[0];
   rgb[1] = 255 – rgb[1],
   rgb[2] = 255 – rgb[2];
   setEntry(i, rgb);
```

1.36 What if the image in Problem 1.35 is in the 24-bit true color representation?

```
int i, j, rgb[3];
for (j = 0; j < height; j + + )
 for (i = 0;i < width; i + + ) {
   getPixel(i, j, rgb);
   rgb[0] = 255 – rgb[0];
   rgb[1] = 255 – rgb[1],
   rgb[2] = 255 – rgb[2];
   setPixel(i, j, rgb);
```

1.37 Calculate the sum and product of $0.5 + 2.0i$ and $1.0 - 1.0i$.

$$(0.5 + 1.0) + (2.0 + (-1.0))i = 1.5 + 1.0i$$
$$(0.5 \times 1.0 - 2.0 \times (-1.0)) + (0.5 \times (-1.0) + 2.0 \times 1.0)i = 2.5 + 1.5i$$

1.38 Calculate the square of the two complex numbers in Problem 1.37.

$$(0.5^2 - 2.0^2) + 2 \times 0.5 \times 2.0i = -3.75 + 2.0i$$
$$(1.0^2 - (-1.0)^2) + 2 \times 1.0 \times (-1.0)i = 0.0 - 2.0i$$

1.39 Show that $1 + 254 \times \text{count} / N$ provides a proportional mapping from count in $[0, N]$ to c in $[1, 255]$.

Proportional mapping means that we want

$$(c - 1)/(255 - 1) = (\text{count} - 0)/(N - 0)$$

Hence $c = 1 + 254 \times \text{count}/N$.

1.40 Modify the pseudo code for visualizing the Mandelbrot set to visualize the Julia sets.

```
int i, j, count;
float delta = (Re_max — Re_min)/width;
for (i = 0, x.real = Re_min; i < width; i + + , x.real + = delta)
   for (j = 0, x.imag = Im_min; j < height; j + + , x.imag + = delta) {
      count = 0;
      while (|x| < 2.0 && count < N) {
         compute x = x^2 + z;
         count + + ;
      }
      if (|x| < 2.0) setColor(0);
      else setColor(1 + (254 × count/N));
      setPixel(i, j);
```

1.41 How to avoid the calculation of square root in an actual implementation of the algorithms for visualizing the Mandelbrot and Julia sets?

Test for $|x|^2 < 4.0$ instead of $|x| < 2.0$.

SUPPLEMENTARY PROBLEMS

1.1 Can a $5 \times 3\frac{1}{2}$ inch image be presented at 6×4 inch without introducing geometric distortion?

1.2 Referring to Supplementary Problem 1.1, what if the original is $5\frac{1}{4} \times 3\frac{1}{2}$ inch?

1.3 Given the portrait image of a person, describe a simple way to make the person look more slender.

1.4 An RGB color image can be converted to a gray-scale image using the formula 0.299R + 0.587G + 0.114B for gray levels (see Chapter 11, Section 11.1 under "The NTSC YIQ Color Model"). Assuming that getPixel(*x*, *y*, *rgb*) now reads pixel values from a 24-bit input image and setPixel(*x*, *y*, *i*) assigns pixel values to an output image that uses a gray-scale lookup table, develop a pseudo-code segment to convert the input image to a gray-scale output image.

ANSWERS TO SUPPLEMENTARY PROBLEMS

1.1 No, since there is a change in aspect ratio ($5/3.5 \neq 6/4$).

1.2 Yes, since $5.25/3.5 = 6/4 = 1.5$.

1.3 Present the image at an aspect ratio that is lower than the original.

1.4
```
int i, j, c, rgb[3];
for (j = 0; j < height; j++)
   for (i = 0; i < width; i++) {
      getPixel(i, j, rgb);
      c = 0.299*rgb[0] + 0.587*rgb[1] +
0.144*rgb[2];
      setPixel(i, j, c);
   }
```

Chapter Two

Overview of Graphic I-O Devices

Computer Graphics is a study of generation of graphic images as diverse as line drawings and realistic rendering of natural objects. This requires both hardware and software that facilitate production of these images. There are many special purpose hardware which are basically used for generating and displaying graphic images. This chapter gives an overview of a graphic input and output devices and the logical functioning of these devices. Large majority of computer graphic systems utilize some type of CRT display and most of the fundamental display concepts are embodied in CRT display technology. The three most common types of CRT display technologies are direct view storage tube display, calligraphic refresh display and raster scan refresh display. The first two are line drawing displays while the third is a point-plotting device. An individual display may incorporate more than one technology. Cathode ray tube, which was discussed in the previous chapter, is the basis for all graphic devices. First we discuss about the types of displays that are used in computer graphics.

2.1 RANDOM SCAN DISPLAYS

The direct view storage tube display and calligraphic display are called random scan displays or line drawing displays. A line can be drawn directly from one addressable point to another.

Direct View Storage Tube Display

The direct view storage tube display is the simplest of all CRT display. It is a CRT with a long persistence phosphor. The line or character will remain visible nearly up to an hour unless it is

erased. The intensity of the electron beam of the CRT is sufficiently increased to draw a line on the display. This effects the phosphor to get its bright state. It can be erased in half a second by flooding the entire tube with a specific voltage causing the phosphor to get in to dark state. The main difficulty is that all lines and characters are erased because the entire tube has to be flooded. This is the limitation for animation using this display. This is also why it is not useful for interactive drawings. However, it is flicker free. It is capable of displaying any number of lines and is easier to program.

Calligraphic Refresh Display

This is also a random scan display or a line drawing display. However it uses very short persistence phosphor. Because of the short persistence, the picture has to be redrawn on the CRT a number of times per second. This redrawing on the CRT is called refreshing the CRT. Generally the rate of refresh should be at least 30 times per second otherwise a flickering image will be seen which is quite irritating to the viewer.

In addition to the CRT this display requires two more elements called the display buffer and the display controller. The display buffer is contiguous memory containing all the information required to draw the picture on the CRT. The display controller takes this information and gives it to the CRT for display at the refresh rate. The number of lines that can be displayed depend on size of the display buffer and the speed of the display controller. Another limitation is the speed at which the information of the picture can be processed.

2.2 RASTER REFRESH GRAPHICS DISPLAYS

The raster graphics display is a point-plotting device based on the cathode ray tube. This may be viewed as matrix of discrete cells, called the raster, each of which can be made bright. Lines will be drawn as a series of dots called pixels along the path of the line specified by a line drawing algorithm. As lines are represented by a series of pixels they will appear as a staircase. This is called *aliasing effect* [see Fig. 2.1(a)].

Special cases like horizontal, vertical or diagonal lines with 45° angles will appear as straight lines. [see Fig. 2.1(b)].

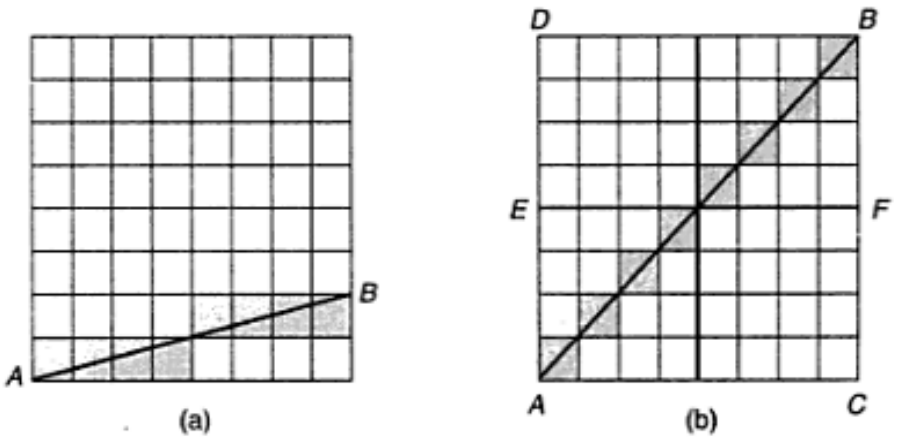


Fig. 2.1 *Rasterization (a) General Line (b) Special Cases*

Raster graphics display is implemented using a *frame buffer*. A frame buffer is a large, contiguous piece of computer memory. Each pixel is in the raster represented by at least one bit. This amount of memory of one bit per pixel is called a *bit plane*.

A 1024 × 1024 square raster requires 2^{20} bits in a single bit plane. The picture is built up in the frame buffer one bit at a time. A single bit plane yields a black-and-white or monochrome display, because a memory bit has only two states, namely, 0 or 1.

As the frame buffer is a digital device and the raster CRT is an analog device, we need a device to convert from a digital representation to an analog signal when information is read from the frame buffer and displayed on the raster graphics device. This is done by a *Digital-to-Analog Converter* (DAC). Figure 2.2 illustrates the raster graphics device with a single bit plane having a black and white frame buffer.

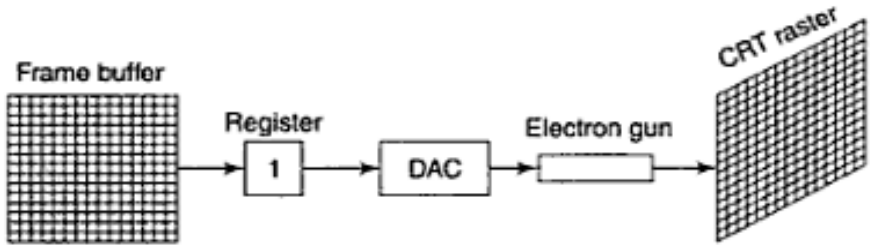


Fig. 2.2 *Raster Graphics Device with a Single Bit Plane having a Black and White Frame Buffer*

Using additional bit planes, color or gray levels can be incorporated into a frame buffer raster graphics device (see Fig. 2.3). In an N-bit plane frame buffer, the intensity of each pixel on the CRT is controlled by a corresponding pixel location in each of the N-bit planes. The bit from each of the N bit planes is loaded into corresponding positions in a register. The resulting binary number is interpreted as an intensity level between 0 and 2^N–1. The value 0 represents dark and

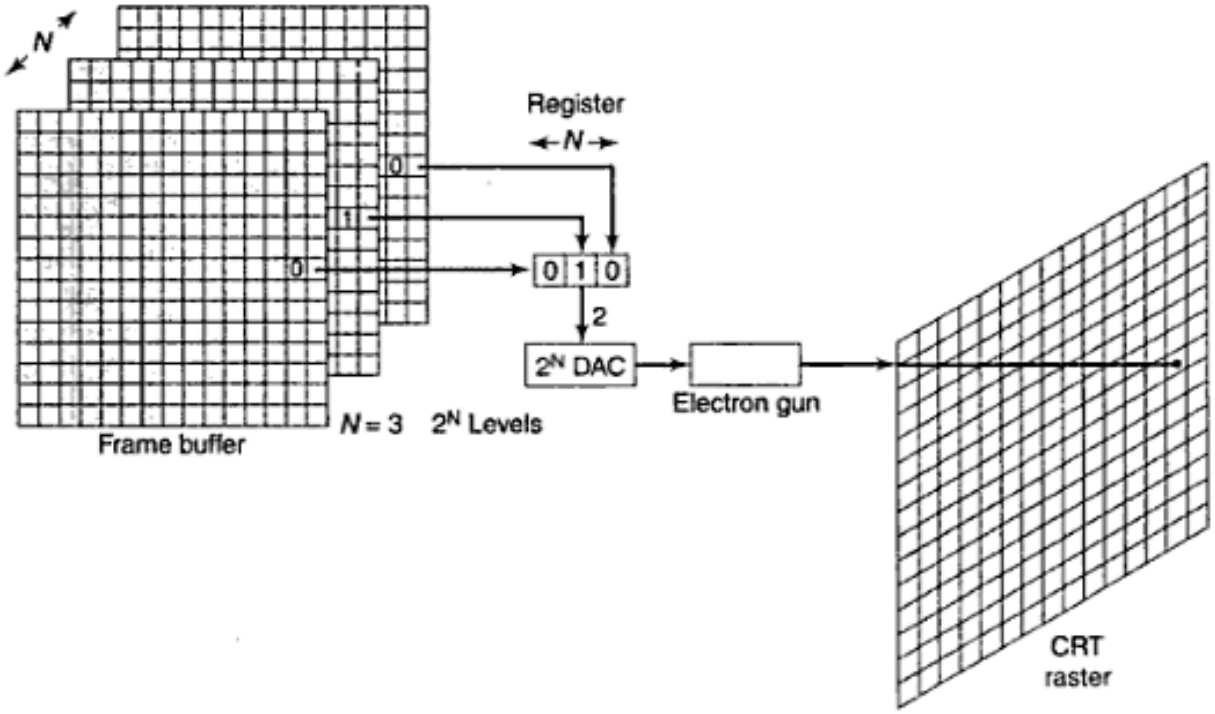


Fig. 2.3 *An N-bit Plane Gray Level Frame Buffer*

2^N-1 represents the full intensity level. This is converted into an analog voltage between 0 and the maximum voltage of the electron gun by the DAC, resulting in 2^N intensity levels or colors.

An increase in the number of intensity levels can be achieved by using a lookup table which is already discussed in chapter 1. After reading the bit planes in the frame buffer, the resulting number is used as an index into the lookup table. If there are N bit planes then the lookup table is a $2^N \times W$ sized table, where W is the number of columns. Thus 2^W colors are possible but only 2^N different intensities are available at one time (see Fig. 2.4).

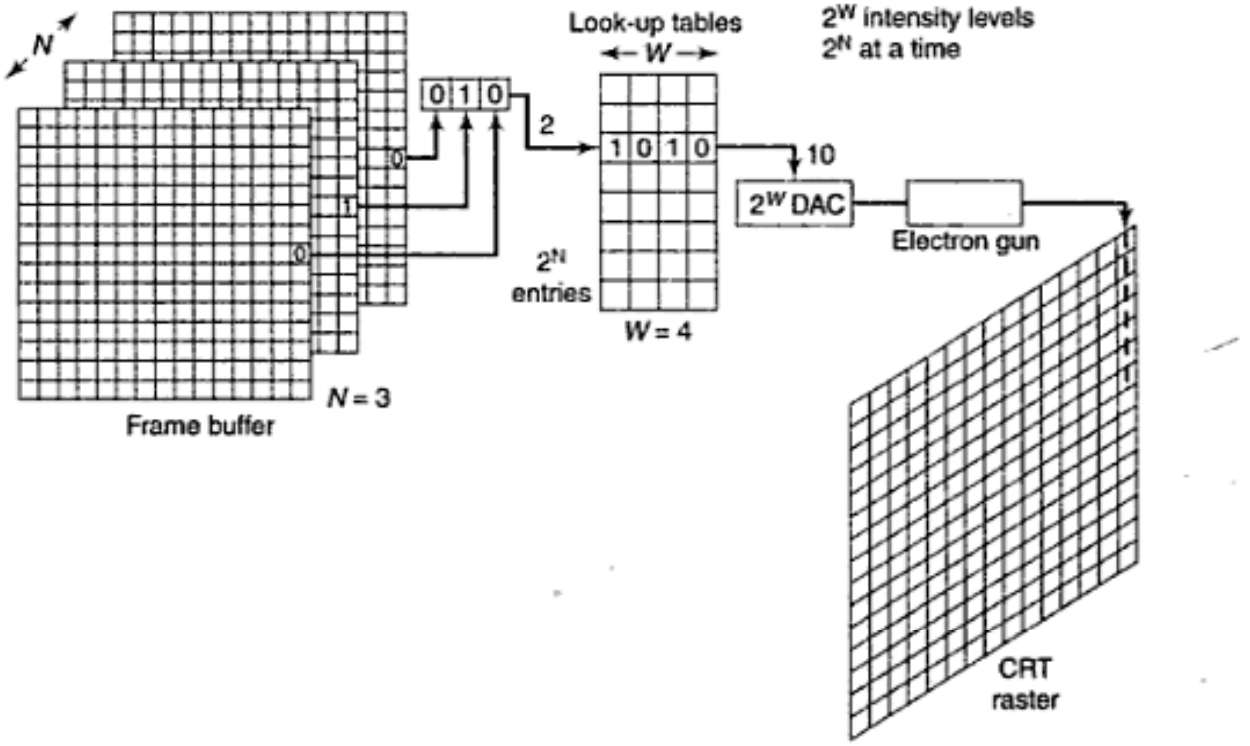


Fig. 2.4 *An N-bit-plane Gray Level Frame Buffer, with a W-bit-wide Lookup Table*

A simple color frame buffer is implemented with three bit planes because there are three primary colors as one bit plane for each of the three primary colors is used. Each bit plane drives one color gun for each primary color, thus yielding 8 colors.

Additional bit planes may be used for each of the three-color guns. A 24-bit plane frame buffer with 8 bit planes per color gives 2^{24} possible colors (see Fig. 2.5). 2^8 shades of red, green or blue are provided. The number of colors can further be extended by using the lookup table for each of the groups of bit planes of red, green and blue.

The raster refresh displays are further improved to some of the displays like flat panel display, liquid crystal display, plasma display and electroluminescent display.

2.3 INTERACTIVE DEVICES

Apart from the keyboard, there are many interactive devices. However, a few of them, namely, tablet, light pen, joystick, mouse, control dial, button, data glove, and touch screen are discussed. These physical devices are used to implement the logical interactive devices, which will be discussed in the next section.

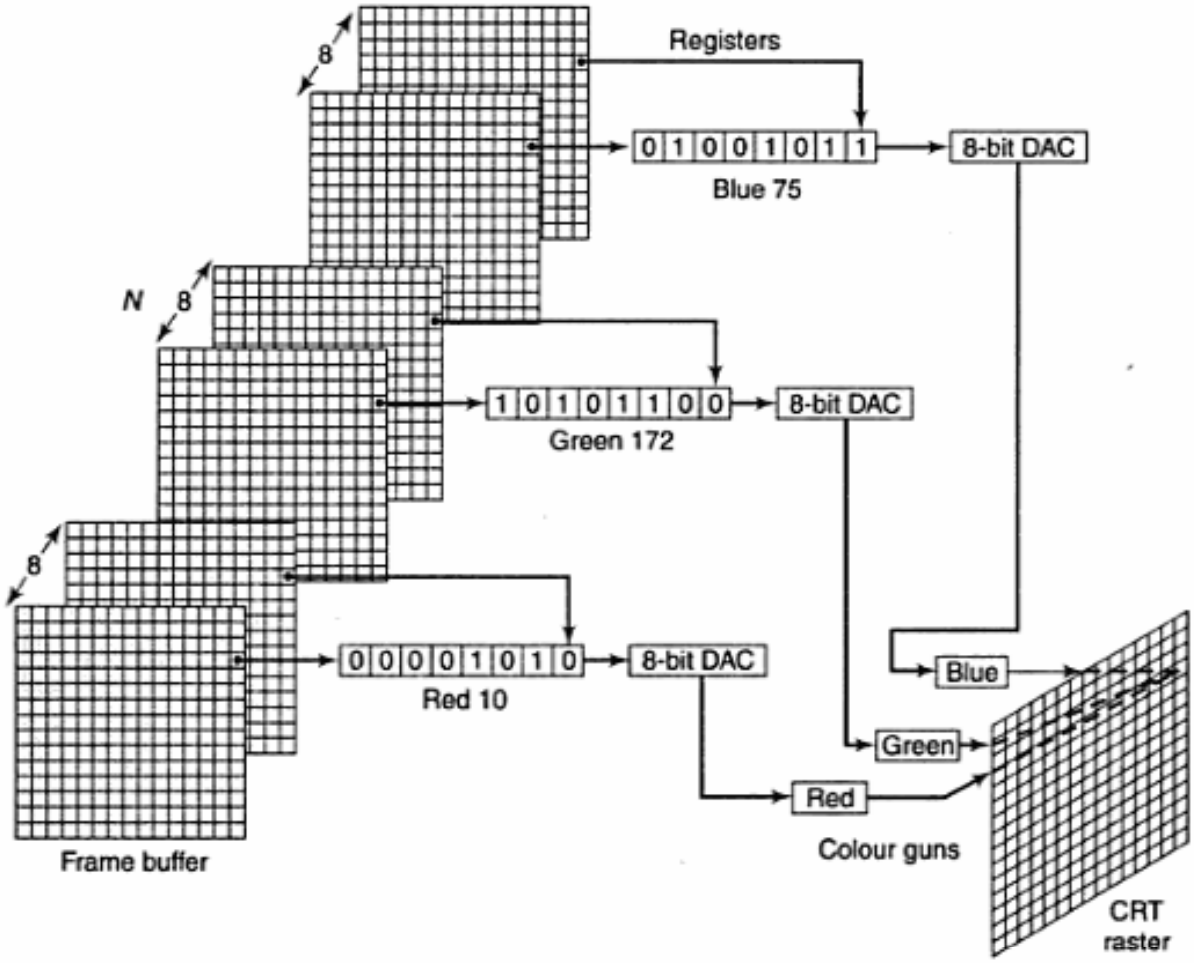


Fig. 2.5 *A 24-bit Plane Frame Buffer with 8 Bit Planes per Color gives 2^{24} Possible Colors*

Tablet

The tablet is the most common device used for locating (see Fig. 2.6). It may either be used in conjunction with a graphics display or standalone (called digitizer). It consists of a flat surface and a pen-like instrument called stylus. The stylus is used to indicate a location on the tablet surface.

When used in conjunction with graphics display, feedback from the display is provided by means of a small tracking symbol called a cursor, which follows the movement of the stylus on the tablet surface. Feedback is provided by digital readouts when used as standalone digitizer.

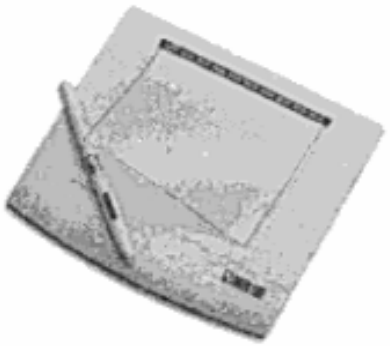


Fig. 2.6 *Tablet with Stylus*

Typically tablets provide two-dimensional coordinate information. The values returned are in tablet coordinates. Software is used to convert the tablet coordinates to world coordinates.

There are different principles used to implement tablets using wires, sound waves and using magnetic principles which are out of the scope of this book.

Joystick

Another locator device is joystick (see Fig. 2.7). A joystick is implemented with two valuators. The valuator controls the movement of the shaft. By using a third valuator to sense rotation of the shaft, a third dimension is incorporated into a joystick. A tracking symbol is normally used for feedback. Joysticks are implemented using forces. The strain in the joystick shaft is measured in two orthogonal directions and converted to cursor movement.

Fig. 2.7 *Joystick*

Trackball

The trackball is similar to the joystick (see Fig. 2.8). A spherical ball is mounted with a base with half of it projecting above the surface. The ball moves freely in any direction. Two valuators, either potentiometers or shaft encoders, are mounted in the base sense. The movement of the ball controls its relative position. Trackballs are frequently equipped with buttons in order that they can be substituted for a mouse.

Fig. 2.8 *Trackball*

Touch Panels

Touch panel is another locator device similar to a tablet. Light emitters are mounted on two adjacent edges, with companion light detectors mounted in the opposite edges. Anything interrupting the two orthogonal light beams yields an (x, y) coordinate pair with which a location is identified.

Mouse

Mouse consists of an upside-down trackball mounted in a small, lightweight box (see Fig. 2.9). As the mouse is moved across a surface, the ball rotates and drives the shafts of two valuators, either potentiometers or digital shaft encoders. The movement of the shafts provides (x, y) coordinates. The mouse can be picked up, moved and set back down in a different orientation. Optical mouse and mouse based on magnetic principles are also available. These use small light source and a small photo electric cell which produce light pulses which are counted and converted into (x, y) coordinates.

Fig. 2.9 *Mouse*

Light Pen

Light pen is a pick device. It contains a sensitive photoelectric cell. The basic information provided by the light pen is timing and hence it depends on the picture being repeatedly produced in a predictable manner.

There are many more devices that are useful for graphic applications which are not included in this book. Some of them are control dial, space ball, data glove, function switches, scanners and 3-D digitizers.

Fig. 2.10 *Light Pen*

2.4 LOGICAL FUNCTIONING OF GRAPHIC I-O DEVICES

We have already discussed some of the physical interactive devices. Now we discuss their logical functional capabilities. One physical device may have more than one logical functional capability. The functional capabilities are classified into four logical types. They are locator function, valuator function, choice function (button function) and pick function.

Locator Function

The locator function provides two or three dimensional coordinate information. Generally, the coordinates returned are in normalized coordinates and may be either relative or absolute. Examples include tablet, touch panel, joystick, mouse, track ball etc.

Valuator Function

The valuator function provides a single value as a real number. This may be bound or unbound. A bound valuator has mechanical or programmed stops within a fixed range while an unbounded valuator has an infinite range.

Button or Choice Function

The choice function selects and activates events or procedures which control the interactive flow or change the underlying task. It generally provides binary information. Keyboard is a specific example of collection of buttons or choice functions.

Pick Function

The pick function selects objects within the displayed picture. It picks up the selected objects.

2.5 OUTPUT DEVICES

Apart from the display devices there are many output devices. Some of them are hard copy devices like inkjet printers, laser printers, plotters etc., the features of some of them are discussed below.

Inkjet Printers

These are raster scan devices and useful for generating low cost color output. The basic idea of these printers is to sprinkle very small drops of ink onto the medium (normally a paper). There are many types of inkjet printers depending on their flow. The resolution of inkjet printers is determined by the size of the drops, i.e. the size of the nozzle of the drops. As the nozzle size is very small, nozzle clogging, air bubbles and dry up of ink are some of the general problems of these printers. Color inkjet printers have four nozzles one for each color and one for black. Each of the three colors is blended together before drying to get the required color.

Laser Printers

A laser printer is also a raster scan device. The print engine contains a drum coated with some material which is photoconductive. The drum is scanned by a semiconductor diode laser. As the drum rotates, the coating is electrically charged and remains charged until it is struck by light from the laser. The light discharges selecting on the drum to form a negative image. A toner, which is a black powder, is attracted to the charged areas of the drum. The toner is fused to the paper using heat and pressure which forms a permanent image.

There are other devices like plotters etc. that are not discussed here as they are outside the scope of this book.

SOLVED PROBLEMS

2.1 What is a frame buffer?

A frame buffer is a large contiguous piece of computer memory.

2.2 What is a bit plane?

The amount of memory with one memory bit per pixel is called a bit plane.

2.3 How many colors are possible with three bit plane frame buffer raster graphics device?

$2^3 = 8$

2.4 In a 24-bit plane color frame buffer with a 10 bit wide lookup tables for each of the three colors, how many colors are possible at a given instant? How many total colors are possible?

There are 24 bit planes in which 8 each are assigned for each of the primary colors. Thus 2^{24} = 16,777,216 colors are possible at any given instant. However the total colors are 2^{30} =1,073,741,824 because each of the 3 primary colors is assigned a 10 bit lookup table.

2.5 What is a locator?

Locator is a function of a physical interactive device that provides two or three dimensional coordinate information.

2.6 Give an example of a locator device.

Tablet

2.7 Give an example of a pick device.

Light pen

2.8 What is dpi?

Dots per inch

2.9 What is a Pixel?

The smallest addressable element of the screen is called a pixel (Picture element).

2.10 What is the minimum refresh rate of a raster scan display device?

30 frames/sec.

2.11 What is the job of a display controller?

The job of the display controller is to read the contents of the frame buffer and display it on the monitor at the required refresh rate (normally 30 times per second or more).

2.12 The frame buffer is a digital device and raster CRT is an analog device. How do we convert the digital information from the frame buffer on to the raster as a picture?

Using a Digital-to-Analog Converter (DAC)

2.13 In a 512 × 512 raster on a monochrome display with an average access rate of 200 nanoseconds per pixel, what is the refresh rate?

Access rate/pixel = 200 nanoseconds = 200×10^{-9} sec
Size of the raster = 512×512
No. of bit planes = 1 (monochrome display)
$\therefore$ Time required for accessing the raster = $200 \times 10^{-9} \times 512 \times 512 \times 1$ sec
$= 52428800 \times 10^{-9}$
$\therefore$ Refresh rate = No. of frames/sec $= 1/(52428800 \times 10^{-9})$
$= 19.07348633$
$\therefore$ Refresh rate = 19 frames/sec (approximately)

2.14 How many bits are required for a 512 × 512 raster with each pixel being represented by 3 bits?

$512 \times 512 \times 3 = 786342$ bits

2.15 What is the rate of a 1024 × 1024 frame buffer with an average access rate per pixel of 200 nanoseconds on a simple color display?

Access rate/pixel = 200 nanoseconds = 200×10^{-9} sec
Size of the raster = 1024×1024
No. of bit planes = 3 (simple color display)
$\therefore$ Time required for accessing the raster = $200 \times 10^{-9} \times 1024 \times 1024 \times 3$ sec
$= 629145600 \times 10^{-9}$

$\therefore$ Refresh rate = no. of frames/sec $= 1/(629145600 \times 10^{-9})$
$= 1.58945$
$\therefore$ Refresh rate = 1.6 frames/sec (approximately)

2.16 How many pixels are there in a 1024 × 1024 frame buffer?

$1024 \times 1024 = 2^{20}$ pixels

SUPPLEMENTARY PROBLEMS

2.1 How many memory bits are required for a 24-bit plane 1024 × 1024 element raster?

2.2 What are the bit representations of the RGB color combinations of a simple 3-bit plane frame buffer?

2.3 What is the access rate/pixel of a 4096 × 4096 raster having a refresh rate of 30 frames/sec?

ANSWERS TO SUPPLEMENTARY PROBLEMS

2.1 24 × 1024 × 1024 memory bits

2.2

	Red	Green	Blue
Black	0	0	0
Red	1	0	0
Green	0	1	0
Blue	0	0	0
Yellow	1	1	0
Cyan	0	1	1
Magenta	1	0	1
White	1	1	1

2.3 1/(30 × 4096 × 4096) seconds = 2 nanoseconds approximately

Chapter Three

Scan Conversion

Many pictures, from 2D drawings to projected views of 3D objects, consist of graphical primitives such as points, lines, circles, and filled polygons. These picture components are often defined in a continuous space at a higher level of abstraction than individual pixels in the discrete image space. For instance, a line is defined by its two endpoints and the line equation, whereas a circle is defined by its radius, center position, and the circle equation. It is the responsibility of the graphics system or the application program to convert each primitive from its geometric definition into a set of pixels that make up the primitive in the image space. This conversion task is generally referred to as scan conversion or rasterization.

The focus of this chapter is on the mathematical and algorithmic aspects of scan conversion. We discuss ways to handle several commonly encountered primitives including points, lines, circles, ellipses, characters, and filled regions in an efficient and effective manner. We also discuss techniques that help to "smooth out" the discrepancies between the original element and its discrete approximation. The implementation of these algorithms and mathematical solutions varies from one system to another and can be in the form of various combinations of hardware, firmware, and software.

3.1 SCAN-CONVERTING A POINT

A mathematical point (x, y) where x and y are real numbers within an image area, needs to be scan-converted to a pixel at location (x', y'). This may be done by making x' to be the integer part of x, and y' the integer part of y. In other words, $x' = \text{Floor}(x)$ and $y' = \text{Floor}(y)$, where function Floor returns the largest integer that is less than or equal to the argument. Doing so in essence places the

origin of a continuous coordinate system for (x, y) at the lower left corner of the pixel grid in the image space [see Fig. 3.1(a)]. All points that satisfy $x' \le x < x' + 1$ and $y' \le y < y' + 1$ are mapped to pixel (x', y'). For example, point $P_1(1.7, 0.8)$ is represented by pixel (1, 0). Points $P_2(2.2, 1.3)$ and $P_3(2.8, 1.9)$ are both represented by pixel (2, 1).

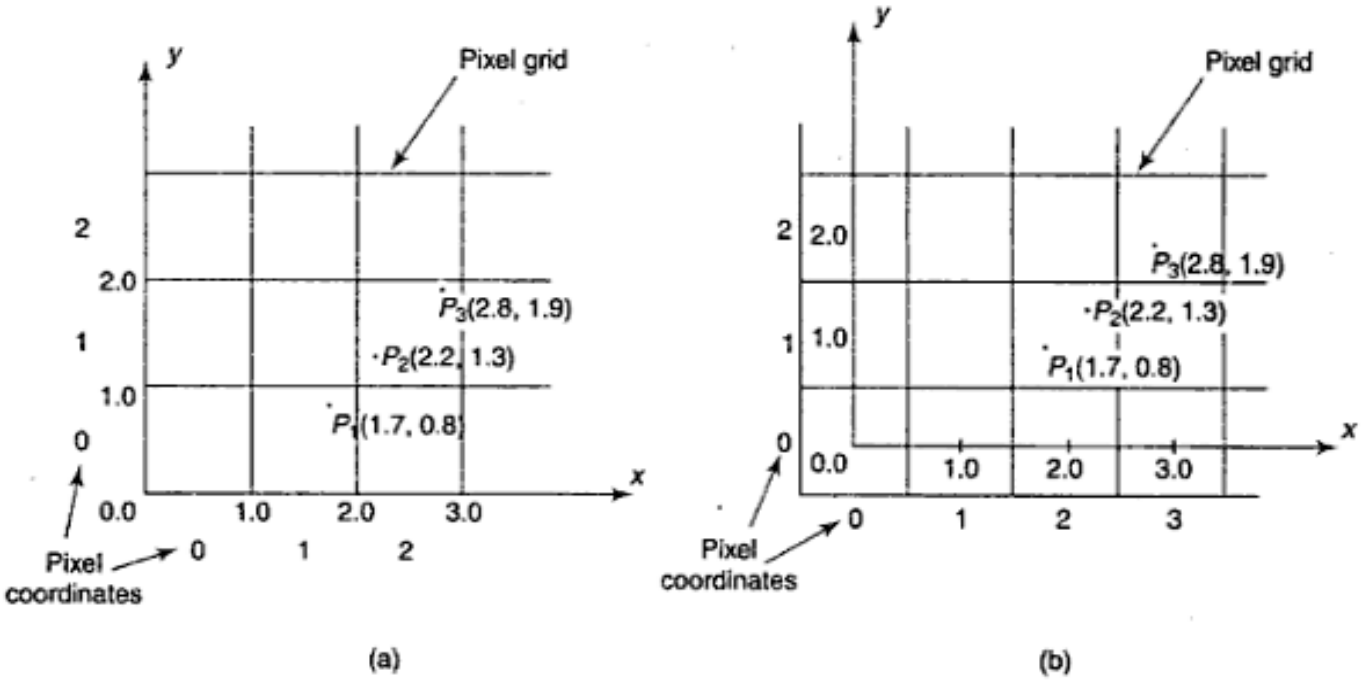


Fig. 3.1 *Scan-converting Points*

Another approach is to align the integer values in the coordinate system for (x, y) with the pixel coordinates [see Fig. 3.1(b)]. Here we scan convert (x, y) by making $x' = \text{Floor}(x + 0.5)$ and $y' = \text{Floor}(y + 0.5)$. This essentially places the origin of the coordinate system for (x, y) at the center of pixel (0, 0). All points that satisfy $x' - 0.5 \le x < x' + 0.5$ and $y' - 0.5 \le y < y' + 0.5$ are mapped to pixel (x', y'). This means that points P_1 and P_2 are now both represented by pixel (2, 1), whereas point P_3 is represented by pixel (3, 2).

We will assume, in the following sections, that this second approach to coordinate system alignment is used. Thus all pixels are centered at the integer values of a continuous coordinate system where abstract graphical primitives are defined.

3.2 SCAN-CONVERTING A LINE

A line in computer graphics typically refers to a line segment, which is a portion of a straight line that extends indefinitely in opposite directions. It is defined by its two endpoints and the line equation $y = mx + b$, where m is called the slope and b the y intercept of the line. In Fig. 3.2 the two endpoints are described by $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$. The line equation describes the coordinates of all the points that lie between the two endpoints.

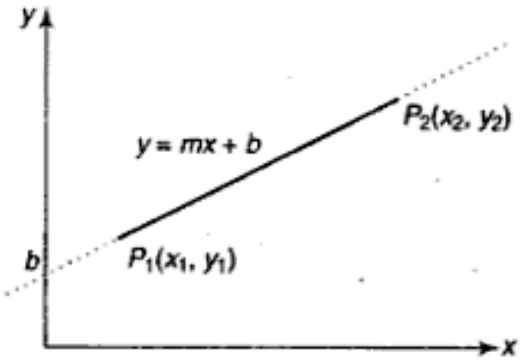


Fig. 3.2 *Defining a Line*

A note of caution: this slope-intercept equation is not suitable for vertical lines. Horizontal, vertical, and diagonal ($|m| = 1$) lines can, and often should, be handled as special cases without going through the following scan-conversion algorithms. These commonly used lines can be mapped to the image space in a straightforward fashion for high execution efficiency.

Direct Use of the Line Equation

A simple approach to scan-converting a line is to first scan-convert P_1 and P_2 to pixel coordinates (x_1', y_1') and (x_2', y_2') respectively; then set $m = (y_2' - y_1')/(x_2' - x_1')$ and $b = y_1' - mx_1'$, if $|m| \leq 1$, then for every integer value of x between and excluding x_1', x_2' calculate the corresponding value of y using the equation and scan-convert (x, y). If $|m| > 1$, then for every integer value of y between and excluding y_1' and y_2', calculate the corresponding value of x using the equation and scan-convert (x, y).

While this approach is mathematically sound, it involves floating-point computation (multiplication and addition) in every step that uses the line equation since m and b are generally real numbers. The challenge is to find a way to achieve the same goal as quickly as possible.

DDA Algorithm

The digital differential analyzer (DDA) algorithm is an incremental scan-conversion method. Such an approach is characterized by performing calculations at each step using results from the preceding step. Suppose at step i we have calculated (x_i, y_i) to be a point on the line. Since the next point (x_{i+1}, y_{i+1}) should satisfy $\Delta y/\Delta x = m$ where $\Delta y = y_{i+1} - y_i$ and $\Delta x = x_{i+1} - x_i$, we have

$$y_{i+1} = y_i + m\Delta x$$

or

$$x_{i+1} = x_i + \Delta y/m$$

These formulae are used in the DDA algorithm as follows. When $|m| \leq 1$, we start with $x = x_1'$ (assuming that $x_1' < x_2'$) and $y = y_1'$ and set $\Delta x = 1$ (i.e., unit increment in the x-direction). The y-coordinate of each successive point on the line is calculated using $y_{i+1} = y_i + m$. When $|m| > 1$, we start with $x = x_1'$ and $y = y_1'$ (assuming that $y_1' < y_2'$) and set $\Delta y = 1$ (i.e., unit increment in the y direction). The x coordinate of each successive point on the line is calculated using $x_{i+1} = x_i + 1/m$. This process continues until x reaches x_2' (for the $|m| \leq 1$ case) or y reaches y_2' (for the $|m| > 1$ case) and all points found are scan-converted to pixel coordinates.

The DDA algorithm is faster than the direct use of the line equation since it calculates points on the line without any floating-point multiplication. However, a floating-point addition is still needed to determine each successive point. Furthermore, cumulative error due to limited precision in the floating-point representation may cause calculated points to drift away from their true position when the line is relatively long.

Bresenham's Line Algorithm

Bresenham's line algorithm is a highly efficient incremental method for scan-converting lines. It produces mathematically accurate results using only integer addition, subtraction and multiplication by 2, which can be accomplished by a simple arithmetic shift operation.

The method works as follows. Assume that we want to scan-convert the line shown in Fig. 3.3 where $0 < m < 1$. We start with pixel $P_1'(x_1', y_1')$, then select subsequent pixels as we work our way to the right, one pixel position at a time in the horizontal direction towards $P_2'(x_2', y_2')$. Once a pixel is chosen at any step, the next pixel is either the one to its right (which constitutes a lower bound for the line) or the one to its right and up (which constitutes an upper bound for the line) due to the limit on m. The line is best approximated by those pixels that fall the least distance from its true path between P_1' and P_2'.

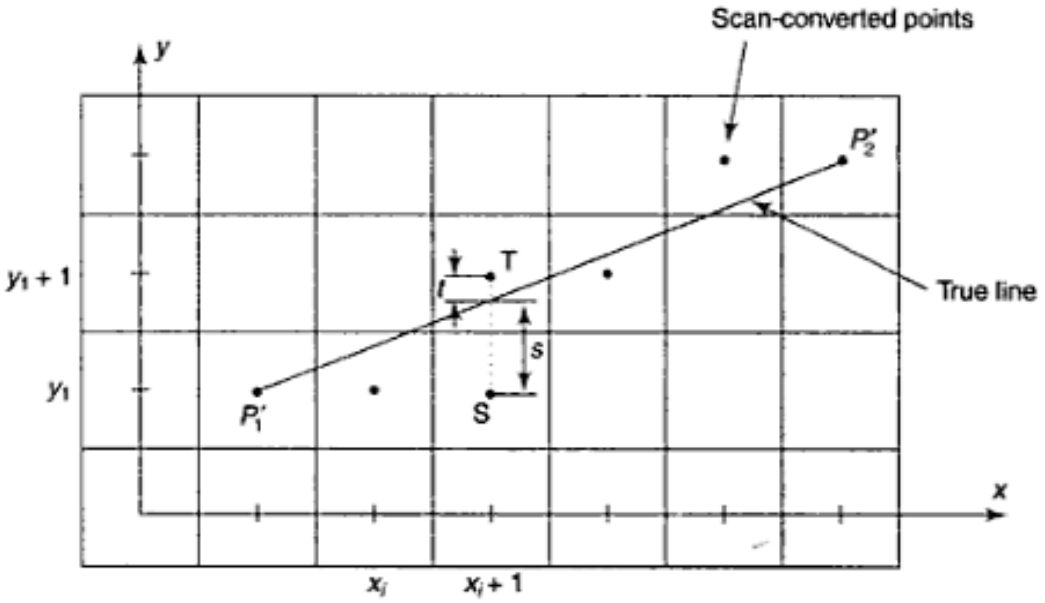


Fig. 3.3 *Scan-converting a Line*

Using the notation of Fig. 3.3, the coordinates of the last chosen pixel upon entering step i are (x_i, y_i). Our task is to choose the next one between the bottom pixel S and the top pixel T. If S is chosen, we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$. If T is chosen, we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i + 1$. The actual y coordinate of the line at $x = x_{i+1}$ is $y = mx_{i+1} + b = m(x_i + 1) + b$. The distance from S to the actual line in the y direction is $s = y - y_i$. The distance from T to the actual line in the y direction is $t = (y_i + 1) - y$.

Now consider the difference between these two distance values: $s - t$. When $s - t$ is less than zero, we have $s < t$ and the closest pixel is S. Conversely, when $s - t$ is greater than zero, we have $s > t$ and the closest pixel is T. We also choose T when $s - t$ is equal to zero. This difference is

$$\begin{aligned} s - t &= (y - y_i) - [(y_i + 1) - y] \\ &= 2y - 2y_i - 1 = 2m(x_i + 1) + 2b - 2y_i - 1 \end{aligned}$$

Substituting m by $\Delta y/\Delta x$ and introducing a decision variable $d_i = \Delta x(s - t)$, which has the same sign as $(s - t)$ since Δx is positive in our case, we have

$$d_i = 2\Delta y^* x_i - 2\Delta x^* y_i + C \qquad \text{where } C = 2\Delta y + \Delta x(2b - 1)$$

Similarly, we can write the decision variable d_{i+1} for the next step as

$$d_{i+1} = 2\Delta y^* x_{i+1} - 2\Delta x^* y_{i+1} + C$$

Then

$$d_{i+1} - d_i = 2\Delta y(x_{i+1} - x_i) - 2\Delta x(y_{i+1} - y_i)$$

Since $x_{i+1} = x_i + 1$, we have

$$d_{i+1} = d_i + 2\Delta y - 2\Delta x(y_{i+1} - y_i)$$

If the chosen pixel is the top pixel T (meaning that $d_i \geq 0$) then $y_{i+1} = y_i + 1$ and so

$$d_{i+1} = d_i + 2\Delta y - \Delta x$$

On the other hand, if the chosen pixel is the bottom pixel S (meaning that $d_i < 0$) then $y_{i+1} = y_i$ and so

$$d_{i+1} = d_i + 2\Delta y$$

Hence we have

$$d_{i+1} = \begin{cases} d_i + 2(\Delta y - \Delta x) & \text{if } d_i \geq 0 \\ d_i + 2\Delta y & \text{if } d_i < 0 \end{cases}$$

Finally, we calculate d_1, the base case value for this recursive formula, from the original definition of the decision variable d_i:

$$\begin{aligned} d_1 &= \Delta x\,[2m(x_1 + 1) + 2b - 2y_1 - 1] \\ &= \Delta x\,[2(mx_1 + b - y_1) + 2m - 1] \end{aligned}$$

Since $mx_1 + b - y_1 = 0$ and $m = \Delta y/\Delta x$, we have

$$d_1 = 2\Delta y - \Delta x$$

In summary Bresenham's algorithm for scan-converting a line from $P_1'(x_1', y_1')$ to $P_2'(x_2', y_2')$ with $x_1' < x_2'$ and $0 < m < 1$ can be stated as follows:

```
int x = x1', y = y1';
int dx = x2' - x1', dy = y2' - y1', dT = 2(dy - dx), dS = 2dy;
int d = 2dy - dx;
setPixel(x, y);
while (x < x2') {
   x ++ ;
   if (d < 0)
      d = d + dS;
   else {
      y ++ ;
      d = d + dT;
   }
   setPixel(x, y);
}
```

Here we first initialize decision variable d and set pixel P_1'. During each iteration of the while loop, we increment x to the next horizontal position, then use the current value of d to select the bottom or top (increment y) pixel and update d, and at the end set the chosen pixel.

As for lines that have other m values we can make use of the fact that they can be mirrored either horizontally, vertically, or diagonally into this 0° to 45° angle range. For example, a line from (x_1', y_1') to (x_2', y_2') with $-1 < m < 0$ has a horizontally mirrored counterpart, from $(x_1' - y_1')$ to $(x_2' - y_2')$ with $0 < m < 1$. We can simply use the algorithm to scan-convert this counterpart, but negate the y coordinate at the end of each iteration to set the right pixel for the line. For a line whose slope is in the 45° to 90° range, we can obtain its mirrored counterpart by exchanging the x and y coordinates of its endpoints. We can then scan-convert this counterpart but we must exchange x and y in the call to setPixel.

3.3 SCAN-CONVERTING A CIRCLE

A circle is a symmetrical figure. Any circle-generating algorithm can take advantage of the circle's symmetry to plot eight points for each value that the algorithm calculates. Eight-way symmetry is used by reflecting each calculated point around each 45° axis. For example, if point 1 in Fig. 3.4 were calculated with a circle algorithm, seven more points could be found by reflection. The reflection is accomplished by reversing the x, y coordinates as in point 2, reversing the x, y coordinates and reflecting about the y axis as in point 3, reflecting about the y axis as in point 4, switching the signs of x and y as in point 5, reversing the x, y coordinates, reflecting about the y axis and reflecting about the x axis as in point 6, reversing the x, y coordinates and reflecting about the y axis as in point 7, and reflecting about the x axis as in point 8.

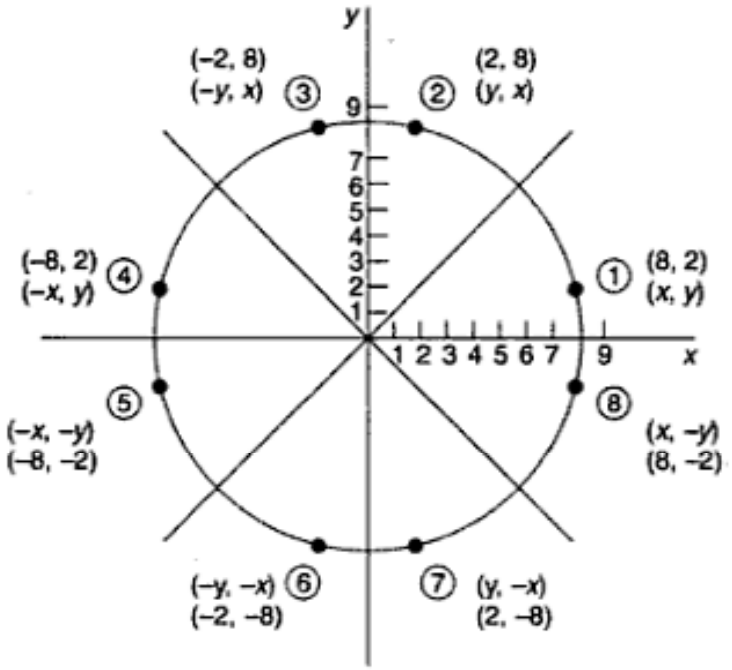


Fig. 3.4 *Eight-way Symmetry of a Circle*

To summarize:

$$P_1 = (x, y) \qquad P_5 = (-x, -y)$$
$$P_2 = (y, x) \qquad P_6 = (y, -x)$$
$$P_3 = (-y, x) \qquad P_7 = (y, -x)$$
$$P_4 = (-x, y) \qquad P_8 = (x, -y)$$

Defining a Circle

There are two standard methods of mathematically defining a circle centered at the origin. The first method defines a circle with the second-order polynomial equation (see Fig. 3.5)

$$y^2 = r^2 - x^2$$

where $x = x$ coordinate
$y = y$ coordinate
r = radius of the circle

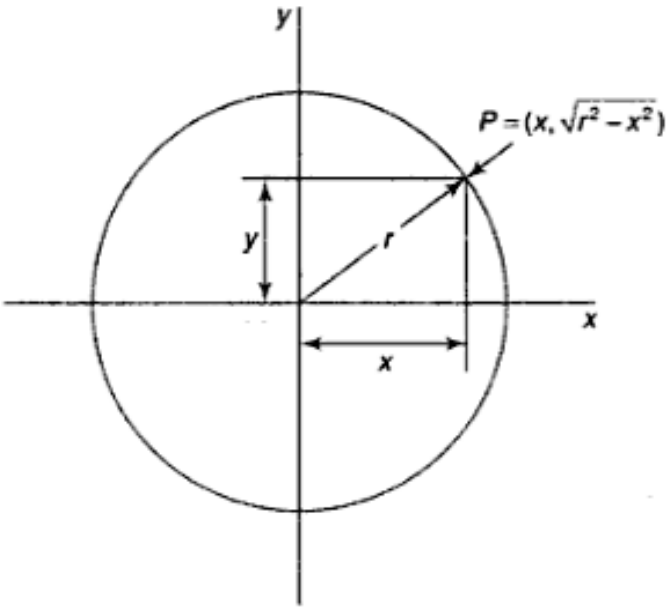


Fig. 3.5 *Circle defined with a Second-degree Polynomial Equation*

With this method, each x coordinate in the sector, from 90° to 45°, is found by stepping x from 0 to $r/\sqrt{2}$, and each y coordinate is found by evaluating $\sqrt{r^2 - x^2}$ for each step of x. This is a very inefficient method, however, because for each point both x and r must be squared and subtracted from each other; then the square root of the result must be found.

The second method of defining a circle makes use of trigonometric functions (see Fig. 3.6):

$$x = r \cos \theta \qquad y = r \sin \theta$$

where θ = current angle
r = radius of the circle
x = x coordinate
y = y coordinate

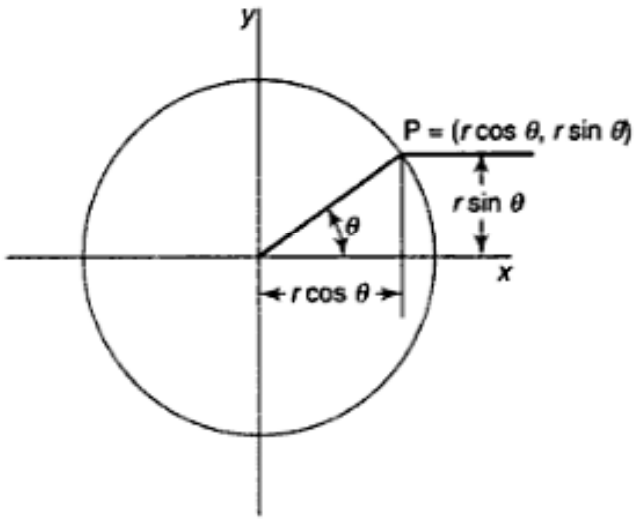


Fig. 3.6 *Circle defined with Trigonometric Functions*

By this method, θ is stepped from θ to $\pi/4$, and each value of x and y is calculated. However, computation of the values of sin θ and cos θ is even more time-consuming than the calculations required by the first method.

Bresenham's Circle Algorithm

If a circle is to be plotted efficiently, the use of trigonometric and power functions must be avoided. And, as with the generation of a straight line, it is also desirable to perform the calculations necessary to find the scan-converted points with only integer addition, subtraction, and multiplication by powers of 2. *Bresenham's circle algorithm* allows these goals to be met.

Scan-converting a circle using Bresenham's algorithm works as follows. If the eight-way symmetry of a circle is used to generate a circle, points will only have to be generated through a 45° angle. And, if points are generated from 90° to 45°, moves will be made only in the $+x$ and $-y$ directions (see Fig. 3.7).

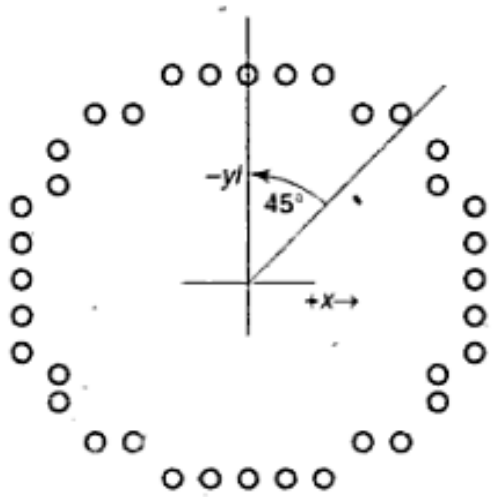


Fig. 3.7 *Circle Scan-converted with Bresenham's Algorithm*

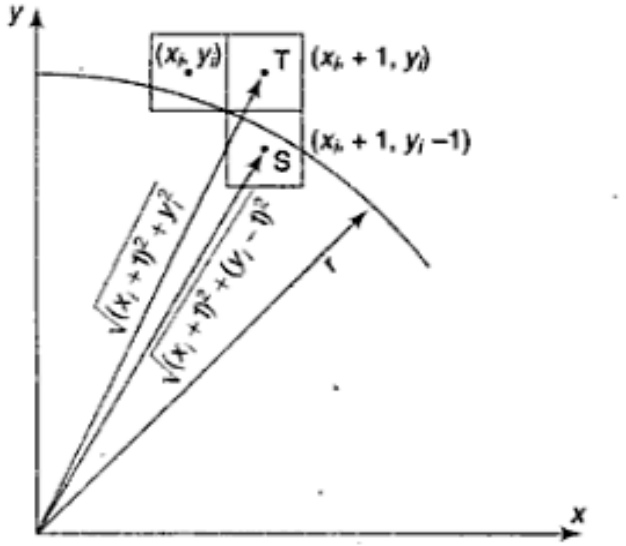


Fig. 3.8 *Choosing Pixels in Bresenham's Circle Algorithm*

The best approximation of the true circle will be described by those pixels in the raster that fall the least distance from the true circle. Examine Fig. 3.8. Notice that, if points are generated from 90° and 45°, each new point closest to the true circle can be found by taking either of two actions: (1) move in the x direction one unit or (2) move in the x direction one unit and move in the negative y direction one unit. Therefore, a method of selecting between these two choices is all that is necessary to find the points closest to the true circle.

Assume that (x_i, y_i) are the coordinates of the last scan-converted pixel upon entering step i (see Fig. 3.8). Let the distance from the origin to pixel T squared minus the distance to the true circle squared = $D(\mathrm{T})$. Then let the distance from the origin to pixel S squared minus the distance to the true circle squared = $D(\mathrm{S})$. As the coordinates of T are $(x_i + 1, y_i)$ and those of S are $(x_i + 1, y_i - 1)$, the following expressions can be developed:

$$D(\mathrm{T}) = (x_i + 1)^2 + y_i^2 - r^2 \qquad D(\mathrm{S}) = (x_i + 1)^2 + (y_i - 1)^2 - r^2$$

This function D provides a relative measurement of the distance from the center of a pixel to the true circle. Since $D(\mathrm{T})$ will always be positive (T is outside the true circle) and $D(\mathrm{S})$ will always be negative (S is inside the true circle), a decision variable d_i may be defined as follows:

$$d_i = D(\mathrm{T}) + D(\mathrm{S})$$

Therefore

$$d_i = 2(x_i + 1)^2 + y_i^2 + (y_i - 1)^2 - 2r^2$$

When $d_i < 0$, we have $|D(\mathrm{T})| < |D(\mathrm{S})|$ and pixel T is chosen. When $d_i \geq 0$, we have $|D(\mathrm{T})| \geq |D(\mathrm{S})|$ and pixel S is selected. We can also write the decision variable d_{i+1} for the next step:

$$d_{i+1} = (2x_{i+1} + 1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2r^2$$

Hence

$$d_{i+1} - d_i = 2(x_{i+1} + 1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2(x_i + 1)^2 - y_i^2 - (y_i - 1)^2$$

Since $x_{i+1} = x_i + 1$, we have

$$d_{i+1} = d_i + 4x_i + 2(y_{i+1}^2 - y_i^2) - 2(y_{i+1} - y_i) + 6$$

If T is the chosen pixel (meaning that $d_i < 0$) then $y_{i+1} = y_i$ and so

$$d_{i+1} = d_i + 4x_i + 6$$

On the other hand, if S is the chosen pixel (meaning that $d_i < 0$) then $y_{i+1} = y_i - 1$ and so

$$d_{i+1} = d_i + 4(x_i - y_i) + 10$$

Hence we have

$$d_{i+1} = \begin{cases} d_i + 4x_i + 6 & \text{if } d_i < 0 \\ d_i + 4(x_i - y_i) + 10 & \text{if } d_i \geq 0 \end{cases}$$

Finally, we set $(0, r)$ to be the starting pixel coordinates and compute the base case value d_1 for this recursive formula from the original definition of d_i:

$$d_1 = 2\,(0 + 1)^2 + r^2 + (r - 1)^2 - 2r^2 = 3 - 2r$$

We can now summarize the algorithm for generating all the pixel coordinates in the 90° to 45° octant that are needed when scan-converting a circle of radius r:

```
int x = 0, y = r, d = 3 - 2r;
while (x <= y) {
  setPixel(x, y);
  if (d < 0)
    d = d + 4x + 6;
  else {
    d = d + 4(x - y) + 10;
    y - -;
  }
  x + + ;
}
```

Note that during each iteration of the while loop we first set a pixel whose position has already been determined, starting with $(0, r)$. We then test the current value of decision variable d in order to update d and determine the proper y coordinate of the next pixel. Finally we increment x.

Midpoint Circle Algorithm

We present another incremental circle algorithm that is very similar to Bresenham's approach. It is based on the following function for testing the spatial relationship between an arbitrary point (x, y) and a circle of radius r centered at the origin:

$$f(x, y) = x^2 + y^2 - r^2 \begin{cases} < 0 & \text{for } (x, y) \text{ inside the circle} \\ = 0 & \text{for } (x, y) \text{ on the circle} \\ > 0 & \text{for } (x, y) \text{ outside the circle} \end{cases}$$

Now consider the coordinates of the point halfway between pixel T and pixel S in Fig. 3.8: $\left(x_1 + 1, y_i - \frac{1}{2}\right)$. This is called the midpoint and we use it to define a decision parameter:

$$p_i = f\left(x_1 + 1, y_i - \frac{1}{2}\right) = (x_i + 1)^2 + \left(y_i - \frac{1}{2}\right)^2 - r^2$$

If p_i is negative, the midpoint is inside the circle, and we choose pixel T. On the other hand, if p_i is positive (or equal to zero), the midpoint is outside them circle (or on the circle), and we choose pixel S. Similarly, the decision parameter for the next step is

$$p_{i+1} = (x_{i+1} + 1)^2 + \left(y_{i+1} - \frac{1}{2}\right)^2 - r^2$$

Since $x_{i+1} = x_i + 1$, we have

$$p_{i+1} - p_i = [(x_i + 1) + 1]^2 - (x_i + 1)^2 + \left(y_{i+1} - \frac{1}{2}\right)^2 - \left(y_i - \frac{1}{2}\right)^2$$

Hence

$$p_{i+1} = p_i + 2(x_i + 1) + 1 + (y_{i+1}^2 - y_i^2) - (y_{i+1} - y_i)$$

If pixel T is choosen (meaning $p_i < 0$), we have $y_{i+1} = y_i$. On the other hand, if pixel S is chosen (meaning $p_i \geq 0$), we have $y_{i+1} = y_i - 1$. Thus

$$p_{i+1} = \begin{cases} p_i + 2(x_i + 1) + 1 & \text{if } p_i < 0 \\ p_i + 2(x_i + 1) + 1 - 2(y_i - 1) & \text{if } p_i \geq 0 \end{cases}$$

We can continue to simplify this in terms of (x_i, y_i) and get

$$p_{i+1} = \begin{cases} p_i + 2x_i + 3 & \text{if } p_i < 0 \\ p_i + 2(x_i - y_i) + 5 & \text{if } p_i \geq 0 \end{cases}$$

Or we can write it in terms of (x_{i+1}, y_{i+1}) and have

$$p_{i+1} = \begin{cases} p_i + 2x_{i+1} + 1 & \text{if } p_i < 0 \\ p_i + 2(x_{i+1} - y_{i+1}) + 1 & \text{if } p_i \geq 0 \end{cases}$$

Finally, we compute the initial value for the decision parameter using the original definition of p_i and $(0, r)$:

$$p_i = (0 + 1)^2 + \left(r - \frac{1}{2}\right)^2 - r^2 = \frac{5}{4} - r$$

One can see that this is not really integer computation. However, when r is an integer we can simply set $p_1 = 1 - r$. The error of being $\frac{1}{4}$ less than the precise value does not prevent p_1 from getting the appropriate sign. It does not affect the rest of the scan-conversion process either, because the decision variable is only updated with integer increments in subsequent steps.

The following is a description of this midpoint circle algorithm that generates the pixel coordinates in the 90° to 45° octant:

```
int x = 0, y = r, p = 1 − r;
while (x <= y) {
  setPixel(x, y);
  if (p < 0)
    p = p + 2x + 3;
  else {
    p = p + 2(x − y) + 5;
    y − − ;
  }
  x ++ ;
}
```

Arbitrarily Centered Circles

In the above discussion of the circle algorithms we have assumed that a circle is centered at the origin. To scan-convert a circle centered at (x_c, y_c), we can simply replace the setPixel(x, y) statement in the algorithm description with setPixel($x + x_c$, $y + y_c$). The reason for this to work is that a circle centered at (x_c, y_c) can be viewed as a circle centered at the origin that is moved by x_c and y_c in the x and y direction, respectively. We can achieve the effect of scan-converting this arbitrarily centered circle by relocating scan-converted pixels in the same way as moving the circle's center from the origin.

3.4 SCAN-CONVERTING AN ELLIPSE

The ellipse, like the circle, shows symmetry. In the case of an ellipse, however, symmetry is four- rather than eight-way. There are two methods of mathematically defining an ellipse.

Polynomial Method of Defining an Ellipse

The polynomial method of defining an ellipse (Fig. 3.9) is given by the expression

$$\frac{(x-h)^2}{a^2} + \frac{(y-k)^2}{b^2} = 1$$

where (h, k) = ellipse center
a = length of major axis
b = length of minor axis

When the polynomial method is used to define an ellipse, the value of x is incremented from h to a. For each step of x, each value of y is found by evaluating the expression

$$y = b\sqrt{1 - \frac{(x-h)^2}{a^2}} + k$$

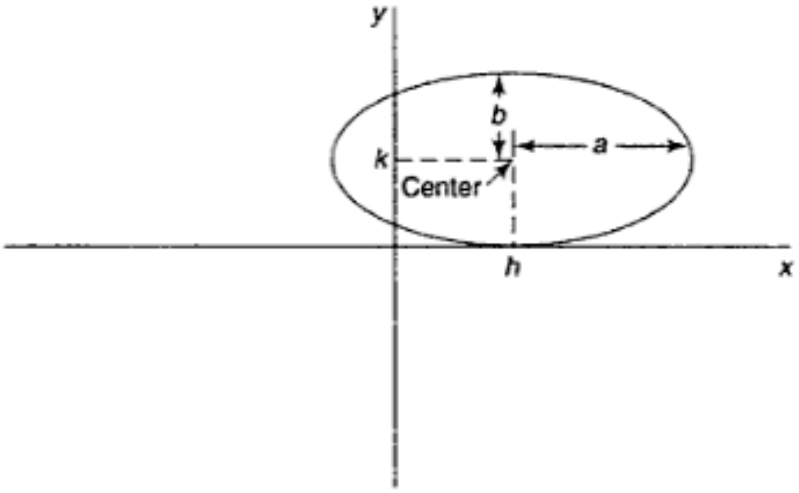


Fig. 3.9 *Polynomial Description of an Ellipse*

This method is very inefficient, however, because the squares of a and $(x - h)$ must be found; then floating-point division of $(x - h)^2$ by a^2 and floating-point multiplication of the square root of $[1 - (x - h)^2/a^2]$ by b must be performed (see Problem 3.20).

Routines have been found that will scan-convert general polynomial equations, including the ellipse. However, these routines are logic intensive and thus are very slow methods for scan-converting ellipses.

Trigonometric Method of Defining an Ellipse

A second method of defining an ellipse makes use of trigonometric relationships (see Fig. 3.10). The following equations define an ellipse trigonometrically:

$$x = a \cos \theta + h \qquad \text{and} \qquad y = b \sin \theta + k$$

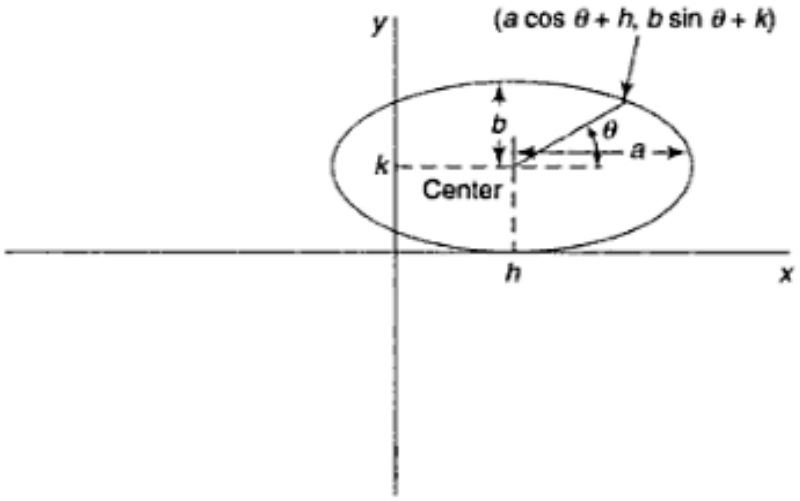


Fig. 3.10 *Trigonometric Description of an Ellipse*

where (x, y) = the current coordinates
a = length of major axis

b = length of minor axis
θ = current angle
(h, k) = center of the ellipse

For generation of an ellipse using the trigonometric method, the value of θ is varied from 0 to $\pi/2$ radians (rad). The remaining points are found by symmetry. While this method is also inefficient and thus generally too slow for interactive applications, a lookup table containing the values for sin θ and cos θ with θ ranging from 0 to $\pi/2$ rad can be used. This method would have been considered unacceptable at one time because of the relatively high cost of the computer memory used to store the values of θ. However, because the cost of computer memory has plummeted in recent years, this method is now quite acceptable.

Ellipse Axis Rotation

Since the ellipse shows four-way symmetry, it can easily be rotated by 90°. The new equation is found by trading a and b, the values which describe the major and minor axes. When the polynomial method is used, the equations used to describe the ellipse become

$$\frac{(x-h)^2}{b^2} + \frac{(y-k)^2}{a^2} = 1$$

where (h, k) = ellipse center
a = length of major axis
b = length of minor axis

When the trigonometric method is used, the equations used to describe the ellipse become

$$x = b \cos \theta + h \quad \text{and} \quad y = a \sin \theta + k$$

where (x, y) = current coordinates
a = length of major axis
b = length of minor axis
θ = current angle
(h, k) = ellipse center

Assume that you would like to rotate the ellipse through an angle other than 90°. It can be seen from Fig. 3.11 that. rotation of the ellipse may be accomplished by rotating the x and y axis by α degrees. When this is done, the equations describing the x, y coordinates of each scan-converted point become

$$x = a \cos \theta - b \sin (\theta + \alpha) + h$$
$$y = b \sin \theta + a \cos (\theta + \alpha) + k$$

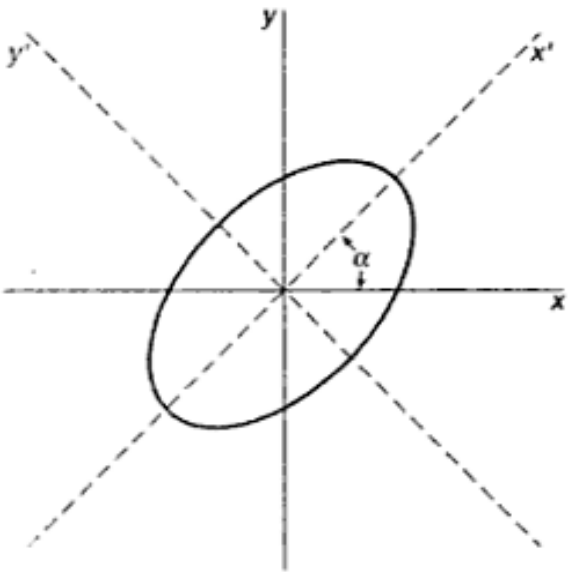


Fig. 3.11 *Rotation of an Ellipse*

Midpoint Ellipse Algorithm

This is an incremental method for scan-converting an ellipse that is centered at the origin in standard position (i.e., with its major and minor axes parallel to the coordinate system axes). It

works in a way that is very similar to the midpoint circle algorithm. However, because of the four-way symmetry property we need to consider the entire elliptical curve in the first quadrant (see Fig. 3.12).

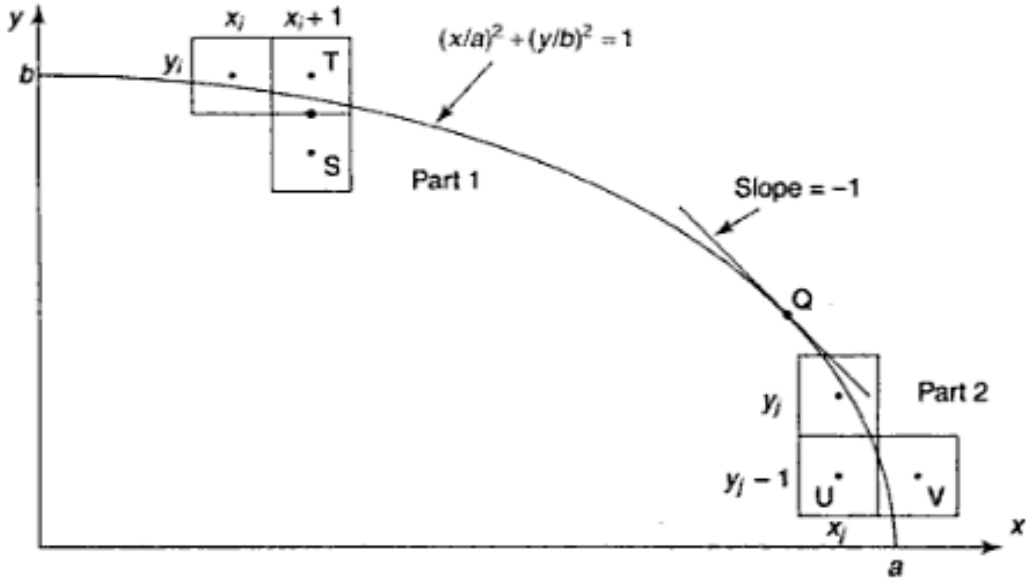


Fig. 3.12 *Scan-converting an Ellipse*

Let us first rewrite the ellipse equation and define function f that can be used to decide if the midpoint between two candidate pixels is inside or outside the ellipse:

$$f(x, y) = b^2x^2 + a^2y^2 - a^2b^2 \begin{cases} < 0 & (x, y) \text{ inside the ellipse} \\ = 0 & (x, y) \text{ on the ellipse} \\ > 0 & (x, y) \text{ outside the ellipse} \end{cases}$$

Now divide the elliptical curve from $(0, b)$ to $(a, 0)$ into two parts at point Q where the slope of the curve is -1. Recall that the slope of a curve defined by $f(x, y) = 0$ is $dy/dx = -fx/fy$, where fx and fy are partial derivatives of $f(x, y)$ with respect to x and y, respectively. We have $fx = 2b^2x$, $fy = 2a^2y$, and $dy/dx = -2b^2x/2a^2y$. This shows that the slope of the curve changes monotonically from one side of Q to the other. Hence we can monitor the slope value during the scan-conversion process to detect Q.

Our starting point is $(0, b)$. Suppose that the coordinates of the last scan-converted pixel upon entering step i are (x_i, y_i). We are to select either $T(x_i + 1, y_i)$ or $S(x_i + 1, y_i - 1)$ to be the next pixel. The midpoint of the vertical line connecting T and S is used to define the following decision parameter:

$$p_i = f\left(x_i + 1, y_i - \frac{1}{2}\right) = b^2(x_i + 1)^2 + a^2\left(y_i - \frac{1}{2}\right)^2 - a^2b^2$$

If $p_i < 0$, the midpoint is inside the curve, and we choose pixel T. On the other hand, if $p_i \geq 0$, the midpoint is outside or on the curve, and we choose pixel S. Similarly, we can write the decision parameter for the next step:

$$p_{i+1} = f\left(x_{i+1} + 1, y_{i+1} - \frac{1}{2}\right) = b^2(x_{i+1} + 1)^2 + a^2\left(y_{i+1} - \frac{1}{2}\right)^2 - a^2b^2$$

Since $x_{i+1} = x_i + 1$, we have

$$p_{i+1} - p_i = b^2[(x_{i+1} + 1)^2 + x_{i+1}^2] + a^2\left[\left(y_{i+1} - \frac{1}{2}\right)^2 - \left(y_i - \frac{1}{2}\right)^2\right]$$

Hence

$$p_{i+1} = p_i + 2b^2x_{i+1} + b^2 + a^2\left[\left(y_{i+1} - \frac{1}{2}\right)^2 - \left(y_i - \frac{1}{2}\right)^2\right]$$

If T is the chosen pixel (meaning $p_i < 0$), we have $y_{i+1} = y_i$. On the other hand, if pixel S is chosen (meaning $p_i > 0$), we have $y_{i+1} = y_i - 1$. Thus we can express p_{i+1} in terms of p_i and (x_{i+1}, y_{i+1}):

$$p_{i+1} = \begin{cases} p_i + 2b^2 x_{i+1} + b^2 & \text{if } p_i < 0 \\ p_i + 2b^2 x_{i+1} + b^2 - 2a^2 y_{i+1} & \text{if } p_i \geq 0 \end{cases}$$

The initial value for this recursive expression can be obtained by evaluating the original definition of p_i with $(0, b)$:

$$p_1 = b^2 + a^2\left(b - \frac{1}{2}\right)^2 - a^2b^2 = b^2 - a^2b + a^2/4$$

We now move on to derive a similar formula for part 2 of the curve. Suppose pixel (x_j, y_j) has just been scan-converted upon entering step j. The next pixel is either $U(x_j, y_j - 1)$ or $V(x_j + 1, y_j - 1)$. The midpoint of the horizontal line connecting U and V is used to define the decision parameter

$$q_j = f\left(x_j + \frac{1}{2}, y_j - 1\right) = b^2\left(x_j + \frac{1}{2}\right)^2 + a^2(y_i - 1)^2 - a^2b^2$$

If $q_j < 0$, the midpoint is inside the curve and V is chosen. On the other hand, if $q_j \geq 0$, it is outside or on the curve and U is chosen. We also have

$$q_{j+1} = f\left(x_{j+1} + \frac{1}{2}, y_{j+1} - 1\right) = b^2\left(x_{j+1} + \frac{1}{2}\right)^2 + a^2(y_{j+1} - 1)^2 - a^2b^2$$

Since $x_{j+1} = x_j - 1$, we have

$$q_{j+1} - q_j = b^2\left[\left(x_{j+1} + \frac{1}{2}\right)^2 - \left(x_j + \frac{1}{2}\right)^2\right] + a^2[(y_{j+1} - 1)^2 - y_{j+1}^2]$$

Hence

$$q_{j+1} = q_j + b^2\left[\left(x_{j+1} + \frac{1}{2}\right)^2 - \left(x_j + \frac{1}{2}\right)^2\right] + 2a^2y_{j+1} + a^2$$

If V is the chosen pixel (meaning $q_j < 0$), we have $x_{j+1} = x_j + 1$. On the other hand, if U is chosen (meaning $q_j \geq 0$), we have $x_{j+1} = x_j$. Thus we can express q_{j+1} in terms of q_j and (x_{j+1}, y_{j+1}):

$$q_{j+1} = \begin{cases} q_j + 2b^2 x_{j+1} - 2a^2 y_{j+1} + a^2 & \text{if } q_j < 0 \\ q_j - 2a^2 y_{j+1} + a^2 & \text{if } q_j \geq 0 \end{cases}$$

The initial value for this recursive expression is computed using the original definition of q_j and the coordinates (x_k, y_k) of the last pixel chosen for part 1 of the curve:

$$q_1 = f\left(x_k + \frac{1}{2}, y_k - 1\right) = b^2\left(x_k + \frac{1}{2}\right)^2 + a^2(y_k - 1)^2 - a^2b^2$$

We can now put together a pseudocode description of the midpoint algorithm for scan-converting the elliptical curve in the first quadrant. There are a couple of technical details that are worth noting. First, we keep track of the slope of the curve by evaluating the partial derivatives *fx* and *fy* at each chosen pixel position. This means that $fx = 2b^2x_i$ and $fy = 2a^2y_i$ for position (x_i, y_i). Since we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$ or $y_i - 1$ for part 1, and $x_{j+1} = x_j$ or $x_j + 1$ and $y_{j+1} = y_j - 1$ for part 2, the partial derivatives can be updated incrementally using $2b^2$ and/or $-2a^2$. For example, if $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i - 1$, the partial derivatives for position (x_{i+1}, y_{i+1}) are $2b^2x_{i+1} = 2b^2x_i + 2b^2$ and $2a^2y_{i+1} = 2a^2y_i - 2a^2$. Second, since $2b^2x_{i+1}$ and $2a^2y_{i+1}$ appear in the recursive expression for the decision parameters, we can use them to efficiently compute p_{i+1} as well as q_{j+1}:

```
int x = 0, y = b;  /* starting point */
int aa = a*a, bb = b*b, aa2 = aa*2, bb2 = bb*2;
int fx = 0, fy = aa2*b;  /* initial partial derivatives */
int p = bb - aa*b + 0.25*aa;  /* compute and round off p1 */
while (fx < fy) { /* |slope| < 1 */
   setPixel(x, y);
   x++;
   fx = fx + bb2;
   if (p < 0)
      p = p + fx + bb;
   else {
      y--;
      fy = fy - aa2;
      p = p + fx + bb - fy;
   }
}
setPixel(x, y);  /* set pixel at (xk, yk) */
p = bb(x + 0.5)(x + 0.5) + aa(y - 1)(y - 1) - aa*bb;  /* set q1 */
while (y > 0) {
   y--;
   fy = fy - aa2;
   if (p >= 0)
      p = p - fy + aa;
   else {
      x++;
      fx = fx + bb2;
      p = p + fx - fy + aa;
   }
   setPixel(x, y);
}
```

3.5 SCAN-CONVERTING ARCS AND SECTORS

Arcs

An arc [Fig. 3.13(a)] may be generated by using either the polynomial or the trigonometric method. When the trigonometric method is used, the starting value is set equal to θ_1 and the ending value is set equal to θ_2 [see Figs. 3.13(a) and 3.13(b)]. The rest of the steps are similar to those used when scan-converting a circle, except that symmetry is not used.

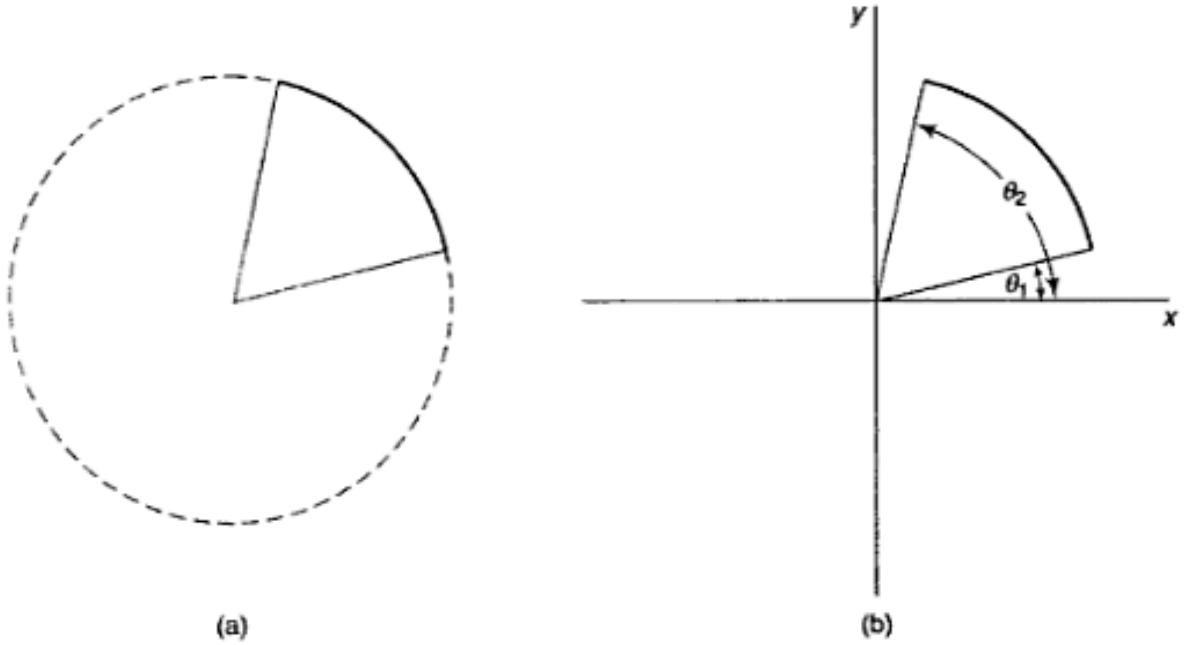


Fig. 3.13

When the polynomial method is used, the value of x is varied from x_1 to x_2 and the values of y are found by evaluating the expression $\sqrt{r^2 - x^2}$ (Fig. 3.14).

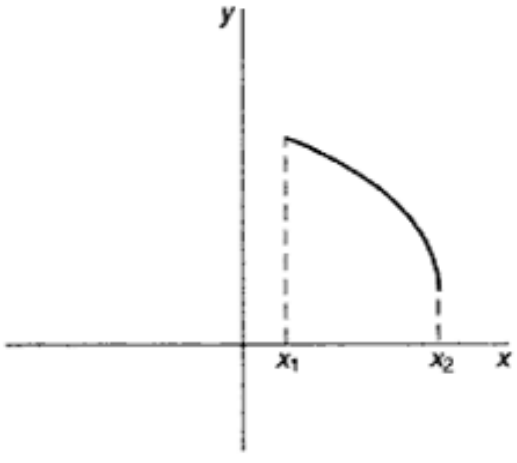


Fig. 3.14

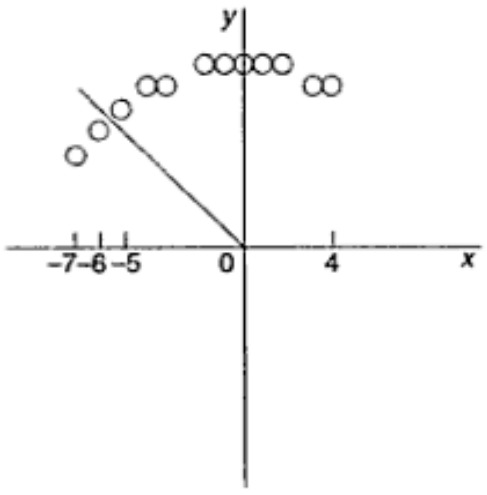


Fig. 3.15

From the graphics programmer's point of view, arcs would appear to be nothing more than portions of circles. However, problems occur if algorithms such as Bresenham's circle algorithm are used in drawing an arc. In the case of Bresenham's algorithm, the endpoints of an arc must be specified in terms of the x, y coordinates. The general formulation becomes inefficient when endpoints must be found (see Fig. 3.15). This occurs because the endpoints for each 45° increment of the arc must be found. Each of the eight points found by reflection must be tested to see if the point is between the specified endpoints of the arc. As a result, a routine to draw an arc based on Bresenham's algorithm must take the time to calculate and test every point on the circle's perimeter. There is always the danger that the endpoints will be missed when a method like this is used. If the endpoints are missed, the routine can become caught in an infinite loop.

Sectors

A sector is scan-converted by using any of the methods of scan-converting an arc and then scan-converting two lines from the center of the arc to the endpoints of the arc.

For example, assume that a sector whose center is at point (h, k) is to be scan-converted. First, scan-convert an arc from θ_1 to θ_2. Next, a line would be scan-converted from (h, k) to $(r \cos \theta_1 + h, r \sin \theta_1 + k)$. A second line would be scan-converted from (h, k) to $(r \cos \theta_2 + h, r \sin \theta_2 + k)$.

3.6 SCAN-CONVERTING A POLYGON

Generally closed contours are represented by a cluster of polygons. Thus, to fill or to draw a contour it is necessary to have methods for filling polygons. There are many algorithms for filling polygons. It may be ooserved that adjacent pixels are likely to have the same characteristics in polygons. This property is called *spatial coherence*. Adjacent pixels on a scan line are likely to have the same characteristics. This is called *scan line coherence*. These two properties are useful in scan converting polygons.

Parity Scan Conversion Algorithm

It may be observed that the scan lines intersect polygons in pairs (see Fig. 3.16). This can be used to set a flag called a *parity bit* to determine whether a particular pixel on a scan line is inside or outside of the polygon. Initially the parity bit is set to zero to indicate that the scan line is outside the polygon. When the scan line intersects the polygon the parity bit is set to 1. This indicates that the scan line is now inside the polygon. At the next intersection the parity bit is set to 0 indicating that the scan line is out of the polygon. When the parity bit is 1 the pixels are set to the polygon colour. When the parity bit is 0 the pixels are set to background colour. (see Fig. 3.17)

Ordered Edge List Algorithm for Polygon Filling

There are many algorithms for polygon filling and solid area scan converting the polygons. We mention here an ordered edge list algorithm which depends on sorting the intersecting points of the polygon edge with the scan line in the scan line order. Here we use half interval scan lines. The algorithm is as given on the next page.

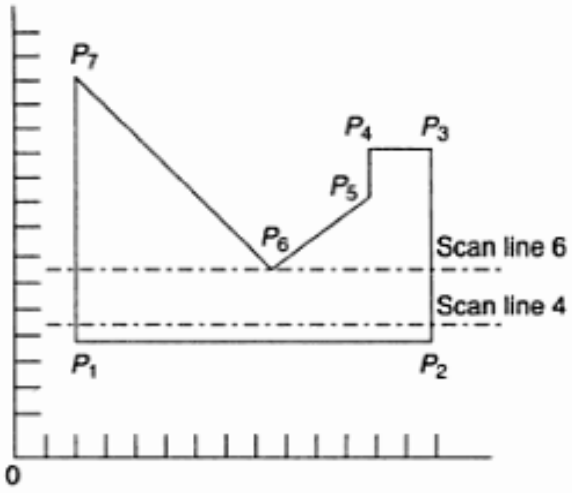


Fig. 3.16

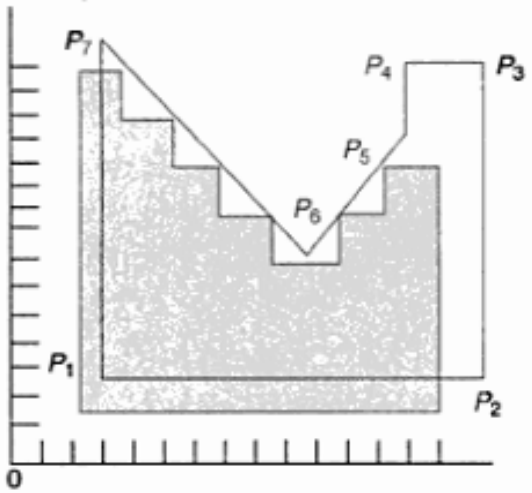


Fig. 3.17

Step 1: Find the intersections of the half interval scan lines with each polygon edge using any line drawing algorithm.

Step 2: Collect each intersection point $(x, y + 1/2)$ in a list.

Step 3: Sort the list by scan line and increasing order of x value on the scan line.

Step 4: Choose pairs of elements (x_1, y_1) and (x_2, y_2) from the sorted list.

Step 5: Activate pixels on the scan line Y for integer values of x.
Such that $x_1 \leq x + 1/2 \leq x_2$.

Example 3.1

Consider a polygon whose vertices are P_1 (2, 2), P_2 (8, 2), P_3 (8, 6), P_4 (5, 3), P_5 (1, 7). Intersections with the half interval scan lines are

Scan line 2.5: (8, 2.5), (1, 2.5)
Scan line 3.5: (8, 3.5), (1, 3.5), (5.5, 3.5), (4.5, 3.5)
Scan line 4.5: (8, 4.5), (1, 4.5), (6.5, 4.5), (3.5, 4.5)
Scan line 5.5: (8, 5.5), (1, 5.5), (7.5, 5.5), (2.5, 5.5)
Scan line 6.5: (1.5, 6.5), (1, 6.5)
Scan line 7.5: None

The complete list sorted in scan line order from the top to the bottom (i.e. from scan line 7.5 to 2.5) and then from left to right is

(1, 6.5), (1.5, 6.5), (1, 5.5), (2.5, 5.5), (7.5, 5.5), (8, 5.5), (1, 4.5), (3.5, 4.5)
(6.5, 4.5), (8, 4.5), (1, 3.5), (4.5, 3.5), (5.5, 3.5), (8, 3.5), (1, 2.5), (8, 2.5).

Extracting pair of intersections from the list and applying the algorithm give above yields the pixel activation list

(1, 6)
(1, 5), (2, 5), (7, 5)
(1, 4), (2, 4), (3, 4), (6, 4), (7, 4)
(1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3), (7, 3)
(1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (6, 2), (7, 2)

The results are shown in Fig. 3.18

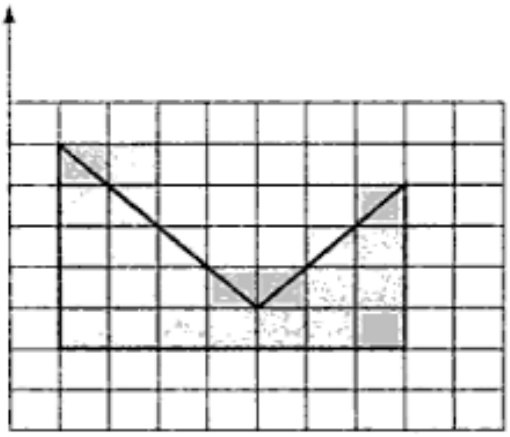


Fig. 3.18

Polygon Inside Test

Filling polygons can be done by setting the pixels inside the polygon as well as those on the boundary. This requires us to determine whether or not a given point is inside the polygon. There are two methods for doing this, namely, the even-odd method and the winding number method.

Even-odd Method

Construct a line segment between a point in the question and any point outside the polygon. Count the number of intersections of this line segment with each side of the polygon. If the number of intersections is even then the point is outside and if it is odd then the point lies inside (see Fig. 3.19).

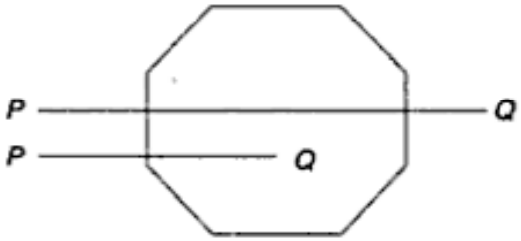


Fig. 3.19

However if the point of intersection is also a vertex point then counting the number of intersections requires some caution. The following rule may be followed in such cases. Count the number intersections as an even number if these two points lie on the same side of the constructed line otherwise count it as single intersection (see Fig. 3.20).

Now it is required to know how to find a point outside the polygon. Take any point Q(x, y) with x coordinate less than minimum of all the x-coordinates vertices of the polygon or greater than the maximum of all the x-coordinates of the vertices of the polygon or any point with y-coordinate with less than the minimum of all the y-coordinates of the vertices of the polygon or greater than the maximum of all the y-coordinates of all vertices of the polygon. It is easy to observe that (x, y) lies outside the polygon.

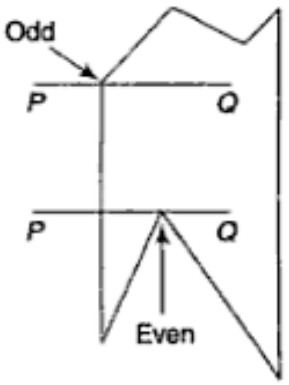


Fig. 3.20

Winding Number Method

This is an alternative method for finding the interior points of any polygon. Instead of just counting the number of intersections as in the case of even-odd method, we give each side of the polygon a number called *winding number*. The total of this winding number is called the net winding. If the net winding is zero then the point is outside otherwise it is inside.

Construct a line segment between the point in question and any point outside the polygon. Find all the sides which cross this line segment. There are two ways for this to happen, start below the line, cross it, and end above the line or start above the line, cross it, and end below the line (i.e. the first y-coordinate value is less than the second y-coordinate value or in the second case the first y-coordinate value is greater than the second y-coordinate value). In the first case the winding number of the side is given as -1 while in the second case the winding number is 1 as shown Fig. 3.21. Any other sides which do not cross this line have their winding numbers zero. The sum of the winding numbers of all the sides is the net winding. If the net winding is zero the point is outside, otherwise it is inside the polygen. These two methods give rise to different results in case of self intersecting polygons as shown in Fig. 3.22.

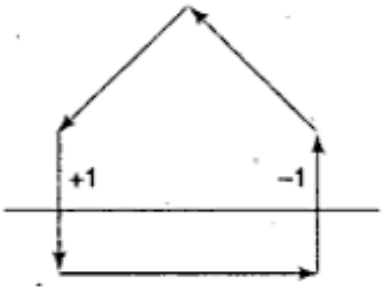


Fig. 3.21

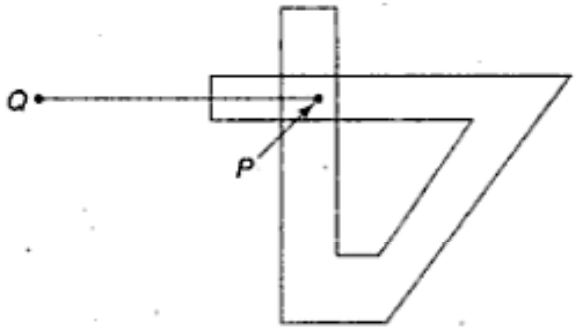


Fig. 3.22

Scan-converting a Rectangle

A rectangle whose sides are parallel to the coordinate axes may be constructed if the locations of two vertices are known [see Fig. 3.23(a)]. The remaining corner points are then derived [see Fig. 3.23(b)]. Once the vertices are known, the four sets of coordinates are sent to the line routine and the rectangle is scan-converted. In the case of the rectangle shown in Figs 3.23(a) and 3.23(b),

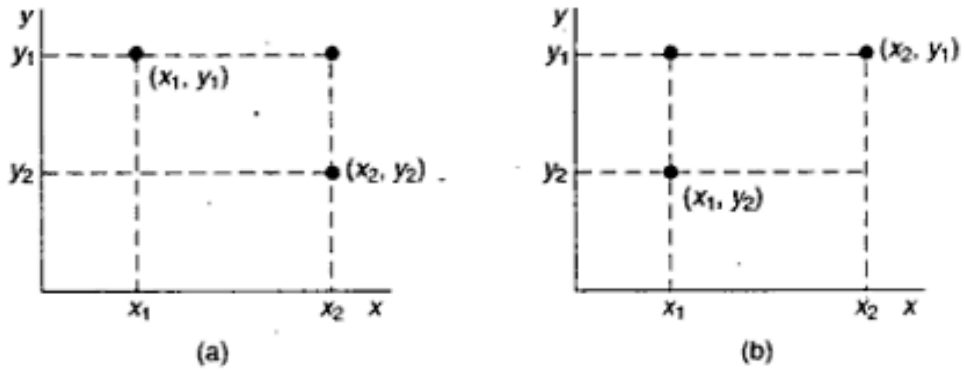


Fig. 3.23

lines would be drawn as follows: line (x_1, y_1) to (x_1, y_2); line (x_1, y_2) to (x_2, y_2); line (x_2, y_2) to (x_2, y_1); and line (x_2, y_1) to (x_1, y_1).

3.7 REGION FILLING

Region filling is the process of "coloring in" a definite image area or region. Regions may be defined at the pixel or geometric level. At the pixel level, we describe a region either in terms of the bounding pixels that outline it or as the totality of pixels that comprise it (see Fig. 3.24). In the first case the region is called boundary-defined and the collection of algorithms used for filling such a region are collectively called *boundary-fill algorithms*. The other type of region is called an interior-defined region and the accompanying algorithms are called *flood-fill algorithms*. At the geometric level a region is defined or enclosed by such abstract contouring elements as connected lines and curves. For example, a polygonal region, or a filled polygon, is defined by a closed polyline, which is a polyline (i.e., a series of sequentially connected lines) that has the end of the last line connected to the beginning of the first line.

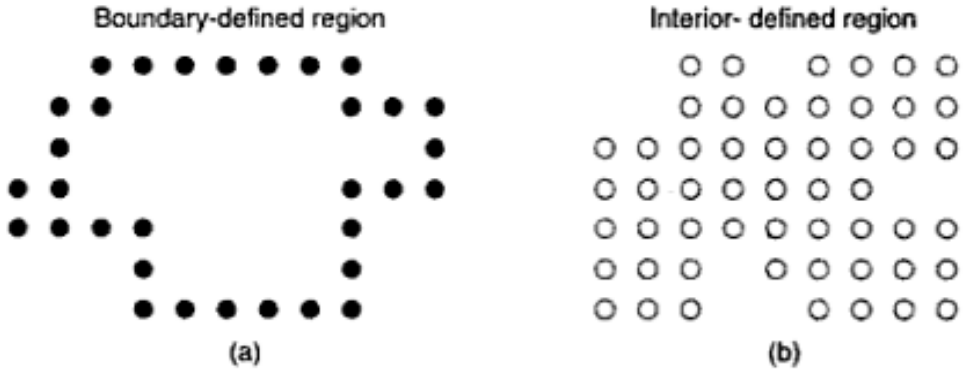


Fig. 3.24

4-Connected versus 8-Connected

An interesting point here is that, while a geometrically defined contour clearly separates the interior of a region from the exterior, ambiguity may arise when an outline consists of discrete pixels in the image space. There are two ways in which pixels are considered connected to each other to form a "continuous" boundary. One method is called 4-connected, where a pixel may have up to four neighbors [see Fig. 3.25(a)]; the other is called 8-connected, where a pixel may have up

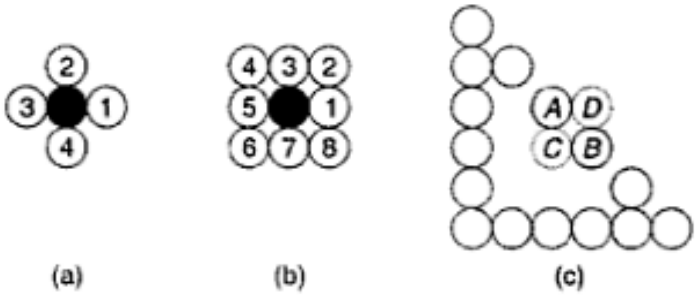


Fig. 3.25 *4-connected vs. 8-connected Pixels*

to eight neighbors [see Fig. 3.25(b)]. Using the 4-connected approach, the pixels in Fig. 3.25(c) do not define a region since several pixels such as A and B are not connected. However using the 8-connected definition we identify a triangular region.

We can further apply the concept of connected pixels to decide if a region is connected to another region. For example, using the 8-connected approach, we do not have an enclosed region in Fig. 3.25(c) since "interior" pixel C is connected to "exterior" pixel D. On the other hand, if we use the 4-connected definition we have a triangular region since no interior pixel is connected to the outside.

Note that it is not a mere coincidence that the figure in Fig. 3.25(c) is a boundary-defined region when we use the 8-connected definition for the boundary pixels and the 4-connected definition for the interior pixels. In fact, using the same definition for both boundary and interior pixels would simply result in contradiction. For example, if we use the 8-connected approach we would have pixel A connected to pixel B (continuous boundary) and at the same time pixel C connected to pixel D (discontinuous boundary). On the other hand, if we use the 4-connectd definition we would have pixel A disconnected from pixel B (discontinuous boundary) and at the same time pixel C disconnected from pixel D (continuous boundary).

Boundary-fill Algorithm

This is a recursive algorithm that begins with a starting pixel, called a seed, inside the region. The algorithm checks to see if this pixel is a boundary pixel or has already been filled. If the answer is no, it fills the pixel and makes a recursive call to itself using each and every neighboring pixel as a new seed. If the answer is yes, the algorithm simply returns to its caller.

This algorithm works elegantly on an arbitrarily shaped region by chasing and filling all non-boundary pixels that are connected to the seed, either directly or indirectly through a chain of neighboring relations. However, a straightforward implementation can take time and memory to execute due to the potentially high number of recursive calls, especially when the size of the region is relatively large.

Variations can be made to limit the number of recursive calls by structuring the order in which neighboring pixels are processed. For example, we can first fill pixels to the left and right of the seed on the same scan line until boundary pixels are hit (something that can be done using a loop control structure). We then inspect each pixel above and below the line just drawn (which can also be done with a loop) to see if it can be used as a new seed for the next horizontal line to fill. This way the number of recursive calls at any particular time is merely N when the current line is N scan lines away from the initial seed.

Flood-fill Algorithm

This algorithm also begins with a seed (starting pixel) inside the region. It checks to see if the pixel has the region's original color. If the answer is yes, it fills the pixel with a new color and uses each of the pixel's neighbors as a new seed in a recursive call. If the answer is no, it returns to the caller.

This method shares great similarities in its operating principle with the boundary-fill algorithm. It is particularly useful when the region to be filled has no uniformly colored boundary. On the

other hand, a region that has a well-defined boundary but is itself multiply colored would be better handled by the boundary-fill method.

The execution efficiency of this flood-fill algorithm can be improved in basically the same way as discussed above regarding the boundary-fill algorithm.

Scan-line Algorithm

In contrast to the boundary-fill and flood-fill algorithms that fill regions defined at the pixel level in the image space, this algorithm handles polygonal regions that are geometrically defined by the coordinates of their vertices (along with the edges that connect the vertices). Although such regions can be filled by first scan-converting the edges to get the boundary pixels and then applying a boundary-fill algorithm to finish the job, the following is a much more efficient approach that makes use of the information regarding edges that are available during scan conversion to facilitate the filling of interior pixels.

We represent a polygonal region in terms of a sequence of vertices V_1, V_2, V_3, ..., that are connected by edges E_1, E_2, E_3, ... ; (see Fig. 3.26). We assume that each vertex V_i has already been scan-converted to integer coordinates (x_i, y_i).

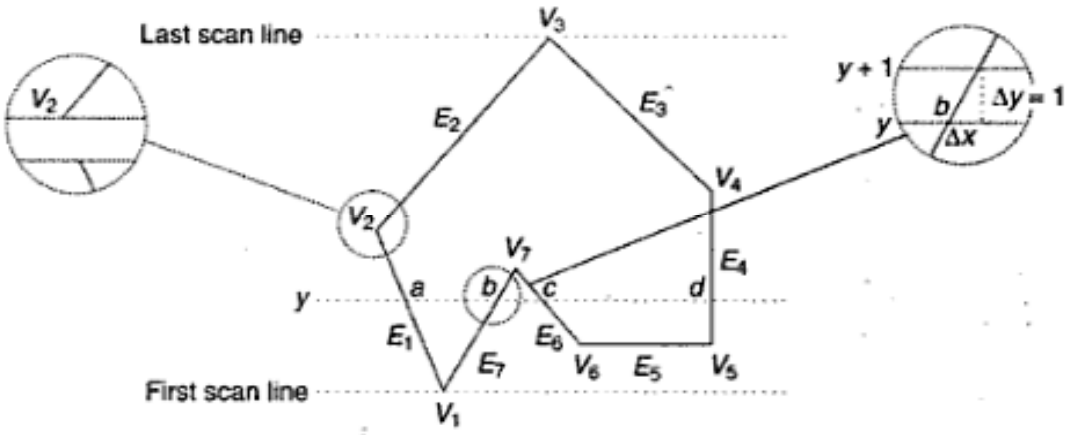


Fig. 3.26 *Scan Converting a Polygon Region*

The algorithm begins with the first scan line the polygon occupies and proceeds line by line towards the last scan line. For each scan line it finds all intersection points between the current scan line and the edges. For example, scan line y intersects edges E_1, E_7, E_6, and E_4 at points a, b, c, and d, respectively. The intersection points are sorted according to their x coordinates and grouped into pairs such as (a, b) and (c, d). A line is drawn from the first point to the second point in each pair.

Horizontal edges are ignored since the pixels that belong to them are automatically filled during scan conversion. For example, edge E_5 is drawn when the corresponding scan line is processed. The two intersection points between the scan line and edges E_6 and E_4 are connected by a line that equals exactly E_5.

Now we take a more careful look at cases when a scan line intersects a vertex. If the vertex is a local minimum or maximum such as V_1 and V_7, no special treatment is necessary. The two edges that join at the vertex will each yield an intersection point with the scan line. These two intersection

points can be treated just like any other intersection points to produce pairs of points for the filling operation. As for vertices V_5 and V_6, they are simply local minimums, each with one joining edge that produces a single intersection point. On the other hand, if a scan line intersects a vertex (e.g., V_4) that is joined by two monotonically increasing or decreasing edges, getting two intersection points at that vertex location would lead to incorrect results (e.g., a total of three intersection points on the scan line that intersects V_4). The solution to this problem is to record only one intersection point at the vertex.

In order to support an efficient implementation of this scan line algorithm we create a data structure called an edge list (see Table 3.1). Each non-horizontal edge occupies one row/record. Information stored in each row includes the y coordinate of the two endpoints of the edge, with y_{min} being the smaller value and y_{max} the larger value (may be decreased by 1 for reasons to be discussed below), the x coordinate of the endpoint whose y coordinate is y_{min}, and the inverse of the slope of the edge m. The rows are sorted according to y_{min}. Going back to our example in Fig. 3.19, since edges E_1 and E_7 both originate from the lowest vertex V_1 at (x_1, y_1), they appear on top of the edge list, with m_1 and m_7 being their slope value, respectively.

Table 3.1 *An Edge List*

Edge	y_{min}	y_{max}	*x coordinate of vertex with* $y = y_{min}$	$1 = m$
E_1	y_1	$y_2 - 1$	x_1	$1 = m_1$
E_7	y_1	y_7	x_1	$1 = m_7$
E_4	y_5	$y_4 - 1$	x_5	$1 = m_4$
E_6	y_6	y_7	x_6	$1 = m_6$
E_2	y_2	y_3	x_2	$1 = m_2$
E_3	y_4	y_3	x_4	$1 = m_3$

Edges in the edge list become active when the y coordinate of the current scan line matches their y_{min} value. Only active edges are involved in the calculation of intersection points. The first intersection point between an active edge and a scan line is always the lower endpoint of the edge, whose coordinates are already stored in the edge's record. For example, when the algorithm begins at the first scan line, edges E_1 and E_7 become active. They intersect the scan line at (x_1, y_1).

Additional intersection points between an edge and successive scan lines can be calculated incrementally. If the edge intersects the current scan line at (x, y), it intersects the next scan line at $(x + 1/m, y + 1)$. For example, edge E_7 intersects scan line y at point b, and so the next intersection point on scan line $y + 1$ can be calculated by $\Delta x = 1/m_7$ since $\Delta y = 1$. This new x value can simply be kept in the x field of the edge's record.

An edge is deactivated or may even be removed from the edge list once the scan line whose y coordinate matches its y_{max} value has been processed, since all subsequent scan lines stay clear from it. The need that was mentioned early to give special treatment to a vertex where two monotonically increasing or decreasing edges meet is elegantly addressed by subtracting one from the y_{max} value of the lower edge. This means that the lower edge is deactivated one line before the

scan line that intersects the vertex. Thus only the upper edge produces an intersection point with that scan line (see V_2 in Fig. 3.26). This explains why the initial y_{max} value for edges E_1 and E_4 has been decreased by one.

3.8 SCAN-CONVERTING A CHARACTER

Characters such as letters and digits are the building blocks of the textual contents of an image. They can be presented in a variety of styles and sizes. The overall design style of a set of characters is referred to as its typeface or font. Commonly used fonts include **Arial**, **Century Schoolbook**, `Courier`, and Times New Roman. In addition, fonts can vary in appearance: **bold**, *italic*, and ***bold and italic***. Character size is typically measured in height in inches, points (approximately $\frac{1}{72}$ inch, and picas (12 points).

Line Display

Generally frame buffer is first set to a background color or intensity. Then the line is rasterized using any line drawing algorithm and correspondingly the appropriate pixels are written to the frame buffer. When the frame buffer is completed then the display controller reads the frame buffer in scan line order and passes the result to the video monitor. Lines will be erased, if required, by setting the appropriate pixels to the background intensity or color. However this leads to a problem when we delete a line having intersection with another line. In this case erasing a line leads to some holes in the figure (see Fig. 3.27).

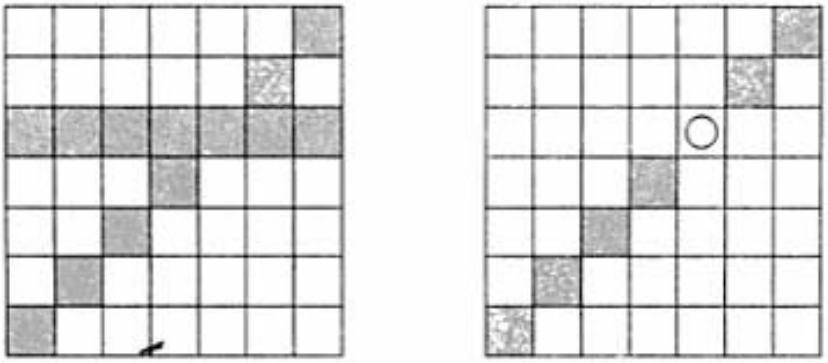


Fig. 3.27

Hence it is necessary to determine the intersections and fill them. This is done using a test called *minmax test* or *boxing test*. If we wish to find the intersections of the line segment AB, then form a rectangle from the minimum and the maximum values of the coordinates of A and B. If x_{min}, x_{max}, y_{min}, y_{max} denote the minimum and maximum values of the line segement with which we have to find the intersection and $xbox_{min}$, $xbox_{max}$, $ybox_{min}$ and $ybox_{max}$ denote the corresponding values of the box, then they will have intersection if $x_{max} < xbox_{min}$ or $x_{min} > xbox_{max}$ or $y_{max} < ybox_{min}$ or $y_{min} > ybox_{max}$. (see Fig. 3.27) Using this condition it can be found whether there is an intersection or not. If it exists, it can be found and filled.

Character Display

A mask is used to write characters to the frame buffer. A mask is a small raster containing the relative locations of the pixels that are used to represent the character. It contains binary values to represent the characters (see Fig. 3.28). For a black and white display 0 and 1 are used to represent whether or not a pixel is used in the mask. To provide multiple color shades for character displays additional bits are used. The character is inserted into the frame buffer by indicating the location in the frame buffer (x_0, y_0) of the origin of the mask. Then each pixel in the mask is displaced by (x_0, y_0) as shown in Fig. 3.28. A character is erased by changing the contents of the frame buffer to the background intensity or color. The mask can be modified to create other fonts and orientations.

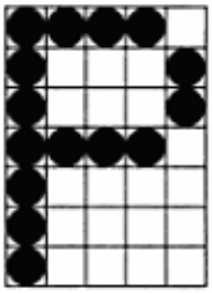


Fig. 3.28 *Bitmap Font*

Bitmap Font

There are two basic approaches to character representation. The first is called a raster or bitmap font, where each character is represented by the on pixels in a bilevel pixel grid pattern called a bitmap (see Fig. 3.28). This approach is simple and effective since characters are defined in already-scan-converted form. Putting a character into an image basically entails a direct mapping or copying of its bitmap to a specific location in the image space. On the other hand, a separate font consisting of scores of bitmaps for a set of characters is often needed for each combination of style, appearance, and size.

Although one might generate variations in appearance and size from one font, the overall results tends to be less than satisfactory. The example in Fig. 3.29 shows that we may overlay the bitmap in Fig. 3.28 onto itself with a horizontal offset of one pixel to produce (a) bold, and shift rows of pixels in Fig. 3.28 to produce (b) italic.

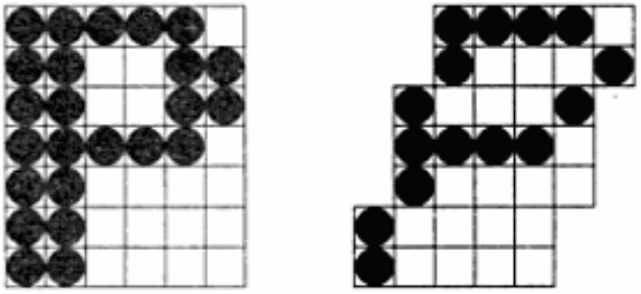


Fig. 3.29 *Generating Variations in Appearance*

Furthermore, the size of a bitmap font is dependent on image resolution. For example, a font using bitmaps that are 12 pixels high produces 12-point characters in an image with 72 pixels per inch. However, the same font will result in 9-point characters in an image with 96 pixels per inch, since 12 pixels now measure 0.125 inch, which is about 9 points. To get 12-point characters in the second image we would need a font with bitmaps that are 16 pixels high.

Outline Font

The second character representation method is called a vector or outline font, where graphical primitives such as lines and arcs are used to define the outline of each character (see Fig. 3.30). Although an outline definition tends to be less compact than a bitmap definition and requires relatively time-consuming scan-conversion operations, it can be used to produce characters of varying size, appearance, and even orientation. For example, the outline definition in Fig. 3.30 can be resized through a scaling transformation, made into italic through a shearing transformation, and turned around with respect to a reference point through a rotation transformation (see Chapter 4).

Fig. 3.30 *Outline Font*

These transformed primitives can be scan-converted directly into characters in the form of filled regions in the target image area. Or they can be used to create the equivalent bitmaps that are then used to produce characters. This alternative is particularly effective in limiting scan-conversion time when the characters have repetitive occurrences in the image.

3.9 ALIASING EFFECTS

Scan conversion is essentially a systematic approach to mapping objects that are defined in continuous space to their discrete approximation. The various forms of distortion that result from this operation are collectively referred to as the aliasing effects of scan conversion.

Staircase

A common example of aliasing effects is the staircase or jagged appearance we see when scan-converting a primitive such as a line or a circle. We also see the stair steps or "jaggies" along the border of a filled region.

Unequal Brightness

Another artifact that is less noticeable is the unequal brightness of lines of different orientation. A slanted line appears dimmer than a horizontal or vertical line, although all are presented at the same intensity level. The reason for this problem can be explained using Fig. 3.31, where the pixels on the horizontal line are placed one unit apart, whereas those on the diagonal line are approximately 1.414 units apart. This difference in density produces the perceived difference in brightness.

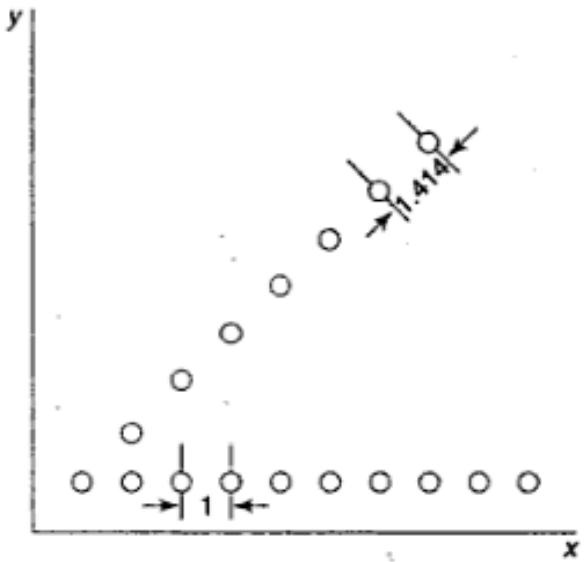


Fig. 3.31

The Picket Fence Problem

The picket fence problem occurs when an object is not aligned with, or does not fit into, the pixel grid properly.

Figure 3.32(a) shows a picket fence where the distance between two adjacent pickets is not a multiple of the unit distance between pixels. Scan-converting it normally into the image space will result in uneven distances between pickets since the endpoints of the pickets will have to be snapped to pixel coordinates [see Fig. 3.32(b)]. This is sometimes called *global aliasing*, as the overall length of the picket fence is approximately correct. On the other hand, an attempt to maintain equal spacing will greatly distort the overall length of the fence [see Fig. 3.32(c)]. This is sometimes called *local aliasing*, as the distances between pickets are kept close to their true distances.

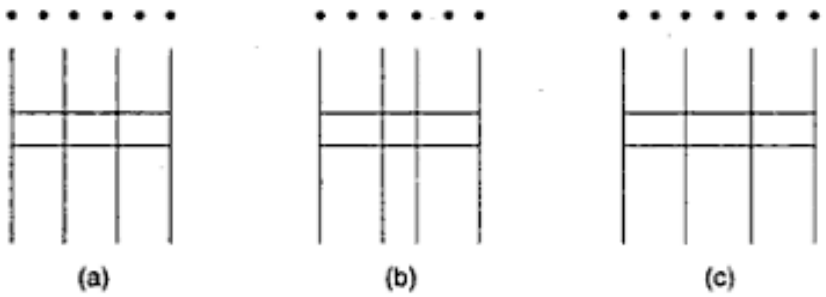


Fig. 3.32 *The Picket Fence Problem*

Another example of such a problem arises with the outline font. Suppose we want to scan-convert the uppercase character "E" in Fig. 3.33(a) from its outline description to a bitmap consisting of pixels inside the region defined by the outline. The result in Fig. 3.33(b) exhibits both asymmetry (the upper arm of the character is twice as thick as the other parts) and dropout (the middle arm is absent). A slight adjustment and/or realignment of the outline can lead to a reasonable outcome [see Fig. 3.33(c)].

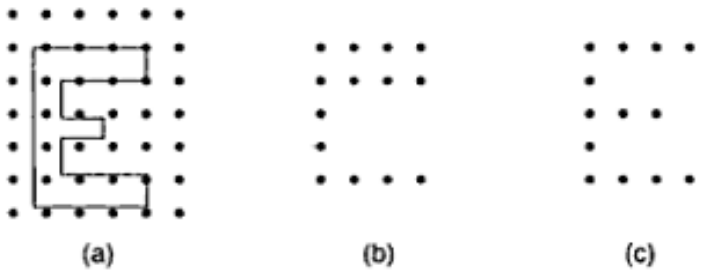


Fig. 3.33 *Scan-converting an Outline Font*

3.10 ANTI-ALIASING

Most aliasing artifacts, when appear in a static image at a moderate resolution, are often tolerable, and in many cases, negligible. However, they can have a significant impact on our viewing experience when left untreated in a series of images that animate moving objects. For example, a line being rotated around one of its endpoints becomes a rotating escalator with length-altering steps. A moving object with small parts or surface details may have some of those features intermittently change shape or even disappear.

Although increasing image resolution is a straightforward way to decrease the size of many aliasing artifacts and alleviate their negative impact, we pay a heavy price in terms of system resource (going from $W \times H$ to $2W \times 2H$ means quadrupling the number of pixels) and the results are not always satisfactory. On the other hand, there are techniques that can greatly reduce aliasing artifacts and improve the appearance of images without increasing their resolution. These techniques are collectively referred to as anti-aliasing techniques.

Some anti-aliasing techniques are designed to treat a particular type of artifact. For instance, an outline font can be associated with a set of rules or hints to guide the adjustment and realignment that is necessary for its conversion into bitmaps of relatively low resolution. An example of such approach is called the TrueType font.

Pre-filtering and Post-filtering

Pre-filtering and post-filtering are two types of general-purpose anti-aliasing techniques. The concept of filtering originates from the field of signal processing, where true intensity values are continuous signals that consist of elements of various frequencies. Constant intensity values that correspond to a uniform region are at the low end of the frequency range. Intensity values that change abruptly and correspond to a sharp edge are at the high end of the spectrum. In order to lessen the jagged appearance of lines and other contours in the image space, we seek to smooth out sudden intensity changes, or in signal-processing terms, to filter out the high frequency components. A pre-filtering technique works on the true signal in the continuous space to derive proper values for individual pixels (filtering before sampling), whereas a post-filtering technique takes discrete samples of the continuous signal and uses the samples to compute pixel values (sampling before filtering).

Area Sampling

Area sampling is a pre-filtering technique in which we superimpose a pixel grid pattern onto the continuous object definition. For each pixel area that intersects the object, we calculate the percentage of overlap by the object. This percentage determines the proportion of the overall intensity value of the corresponding pixel that is due to the object's contribution. In other words, the higher the percentage of overlap, the greater influence the object has on the pixel's overall intensity value.

In Fig. 3.34(a) a mathematical line shown in dotted form is represented by a rectangular region that is one pixel wide. The percentage of overlap between the rectangle and each intersecting pixel is calculated analytically. Assuming that the background is black and the line is white, the percentage values can be used directly to set the intensity of the pixels [see Fig. 3.34(b)]. On the other hand,

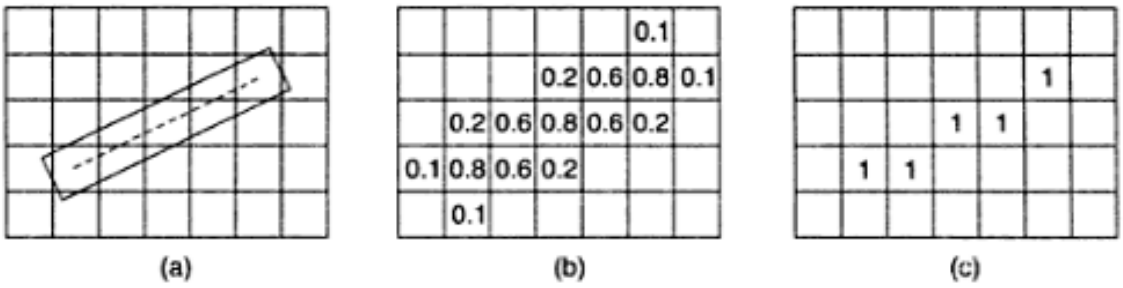


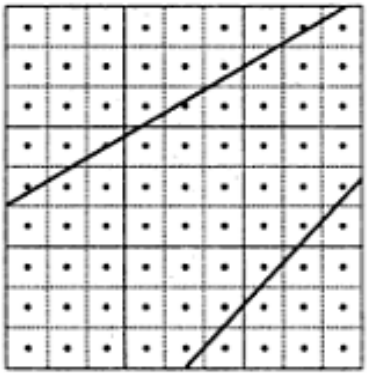
Fig. 3.34 *Area Sampling*

had the background been gray (0.5, 0.5, 0.5) and the line green (0, 1, 0), each blank pixel in the grid would have had the background gray value and each pixel filled with a fractional number f would have been assigned a value of $[0.5(1-f), 0.5(1-f)+f, 0.5(1-f)]$—a proportional blending of the background and object colors.

Although the resultant discrete approximation of the line in Fig. 3.34(b) takes on a blurry appearance, it no longer exhibits the sudden transition from an on pixel to an off pixel and vice versa, which is what we would get with an ordinary scan-conversion method [see Fig. 3.34(c)]. This tradeoff is characteristic of an anti-aliasing technique based on filtering.

Super Sampling

In this approach we subdivide each pixel into subpixels and check the position of each subpixel in relation to the object to be scan-converted. The object's contribution to a pixel's overall intensity value is proportional to the number of subpixels that are inside the area occupied by the object. Figure 3.35(a) shows an example where we have a white object that is bounded by two slanted lines on a black background. We subdivide each pixel into nine (3 × 3) subpixels. The scene is mapped to the pixel values in Fig. 3.35(b). The pixel at the upper right corner, for instance, is assigned $\frac{7}{9}$ since seven of its nine subpixels are inside the object area. Had the object been red (1, 0, 0) and the background light yellow (0.5, 0.5, 0), the pixel would have been assigned $\left(1\times\frac{7}{9}+0.5\times\frac{2}{9}, 0.5\times\frac{2}{9}, 0\right)$, which is $\left(\frac{8}{9}, \frac{1}{9}, 0\right)$.



(a)

0	2/9	7/9
6/9	1	8/9
1	8/9	1/9

(b)

Fig. 3.35 *Super Sampling*

Super sampling is often regarded as a post-filtering technique since discrete samples are first taken and then used to calculate pixel values. On the other hand, it can be viewed as an approximation to the area sampling method since we are simply using a finite number of values in each pixel area to approximate the accurate analytical result.

Low Pass Filtering

This is a post-filtering technique in which we reassign each pixel a new value that is a weighted

average of its original value and the original values of its neighbors. A low pass filter in the form of a $(2n + 1) \times (2n + 1)$ grid, where $n \geq 1$, holds the weights for the computation. All weight values in a filter should sum to one. An example of a 3×3 filter is given in Fig. 3.36(a).

0	1/8	0
1/8	1/2	1/8
0	1/8	0

(a)

0	0	0	0	0	1/8	0
0	0	0	1/8	1/4	1/2	1/8
0	1/8	1/4	5/8	5/8	1/4	0
1/8	5/8	5/8	1/4	1/8	0	0
0	1/8	1/8	0	0	0	0

(b)

Fig. 3.36 *Low Pass Filtering*

To compute a new value for a pixel, we align the filter with the pixel grid and center it at the pixel. The weighted average is simply the sum of products of each weight in the filter times the corresponding pixel's original value. The filter shown in Fig. 3.36(a) means that half of each pixel's original value is retained in its new value, while each of the pixel's four immediate neighbors contributes one eighth of its original value to the pixel's new value. The result of applying this filter to the pixel values in Fig. 3.6(a) is shown in Fig. 3.36(b).

A low pass filter with equal weights, sometimes referred to as a box filter, is said to be doing neighborhood averaging. On the other hand, a filter with its weight values conforming to a two-dimensional Gaussian distribution is called a Gaussian filter.

Pixel Phasing

Pixel phasing is a hardware-based anti-aliasing technique. The graphics system in this case is capable of shifting individual pixels from their normal positions in the pixel grid by a fraction (typically 1/4 and 1/2) of the unit distance between pixels. By moving pixels closer to the true line or other contour, this technique is very effective in smoothing out the stair steps without reducing the sharpness of the edges.

3.11 IMAGE COMPRESSION

Generally a digital image is encoded in the form of a binary file for storage purpose. There are many encoding formats, which typically require optimizing the space requirements. Image compression techniques are useful in this regard. Two major image compression techniques are explained in this section.

Run Length Encoding

In any image it may be observed that a large portion of the image will have same characteristics such as intensity, color, etc. Run length encoding is a technique based on this assumption. It specifies the number of successive pixels on a given scan line with the corresponding intensity. It

is considered in pairs of numbers, the first being the number of successive pixels on that scan line and the second one is the corresponding intensity level. Figure 3.37 illustrates this concept by showing a black and white image on a 20 × 20 rater.

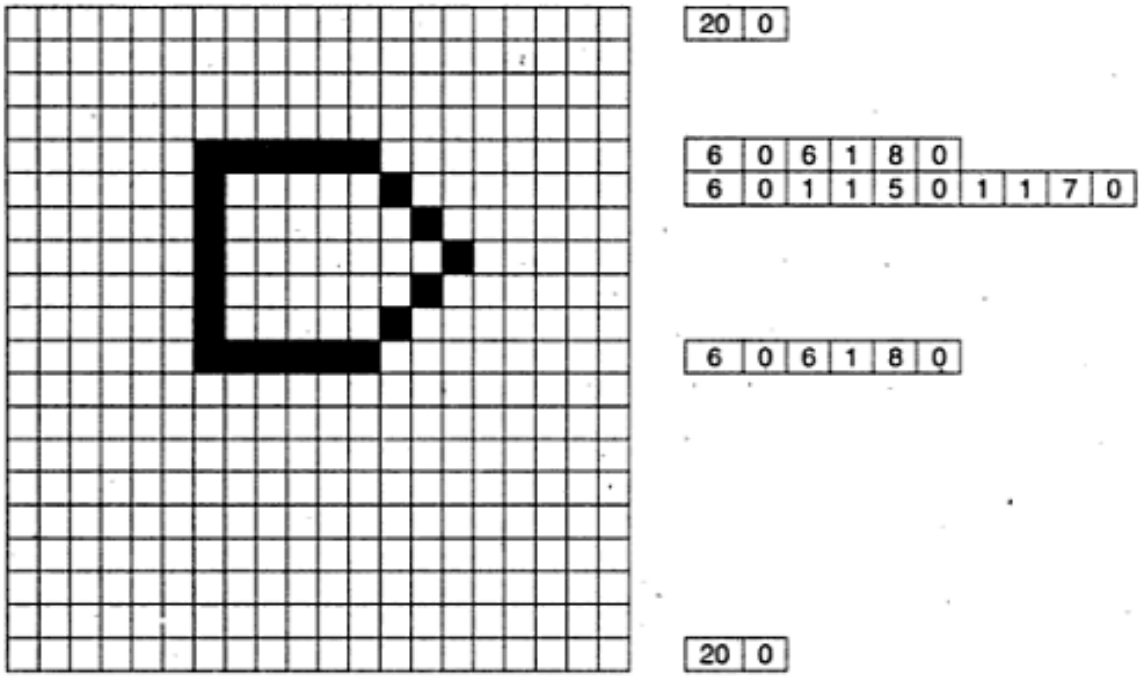


Fig. 3.37 *Black and White Image on a 20 × 20 Rater*

Assuming that 0 represents "on" intensity and 1 represents black, it can be seen that the scan line 1 has 20 pixels of 0 intensity. It is stored as

20	0

While scan lines 5 and 10 are stored as

6	0	6	1	8	0

the scan line 6 is represented as

6	0	1	1	5	0	1	1	7	0

If the intensity of each pixel is taken then, the image is encoded with 400 numbers while in the above scheme it takes only 88 numbers. The data compression in this case is 1 : 4.54. It can be observed that solid figures are easily handled with high ratio of compression.

The run length-encoding scheme can be extended to color images. In case of color images the intensities of red, green and blue colors are stored as:

Run length	Red intensity	Blue intensity	Green intensity

It has some disadvantages as well. The run lengths are stored sequentially and hence it is time-consuming to add or delete parts of the image. Moreover the storage may be twice (In case of black and white and even more otherwise) that of the pixel by pixel storage for images in which alternative columns have different intensities.

Example 3.2

A picture has a resolution of 1024 × 1280 with each of the three colors being represented by 8 bit plans each. What is the storage requirement for a 10 second animation of the above picture with 30 frames per second? If the compression ratio is 5:1 what is the storage requirement?

Since there are 8 bits per each of the colors, the total number of bits to represent is 1024 × 1280 × 3 × 8. A 20 second animated film with 30 frames per second requires 1024 × 1280 × 3 × 8 × 20 × 30 bits = 2250 mega bytes. The storage requirement for a 5:1 compression is 2250/5 = 450 mega bytes.

3.12 RECURSIVELY DEFINED DRAWINGS

In this section we use two common graphical primitives to produce some interesting drawings. Each of these drawings is defined by applying a modification rule to a line or a filled triangle, breaking it into smaller pieces so the rule can be used to recursively modify each piece in the same manner. As the pieces become smaller and smaller, an intriguing picture emerges.

C Curve

A line by itself is a first-order C curve, denoted by C_0 (see Fig. 3.38). The modification rule for constructing successive generations of the C curve is to replace a line by two shorter, equal-length lines joining each other at a 90° angle, with the original line and the two new lines forming a right-angled triangle. See Fig. 3.38 for C_1, C_2, C_3, C_4, and C_5.

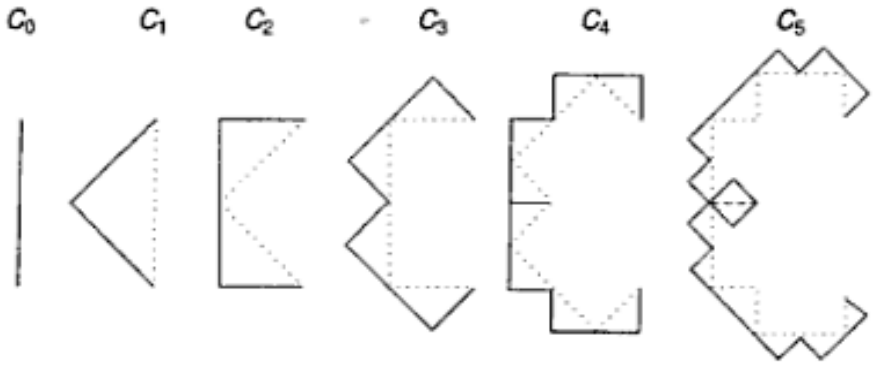


Fig. 3.38 *Successive Generations of the C Curve*

Presume that the following call to the graphics library causes a line to be drawn from (x_1, y_1) to (x_2, y_2) using the system's current color:

line(x_1, y_1, x_2, y_2)

We can describe a pseudo-code procedure that generates C_n:

C-curve (float x, y, len, alpha; int n)

```
{
  if (n > 0) {
    len = len/sqrt(2.0);
    C-curve(x, y, len, alpha + 45, n – 1);
    x = x + len*cos(alpha + 45);
    y = y + len*sin(alpha + 45);
    C-curve(x, y, len, alpha – 45, n – 1);
  } else
    line(x, y, x + len*cos(alpha), y + len*sin(alpha);
}
```

where x and y are the coordinates of the starting point of a line, len the length of the line, alpha the angle (in degrees) between the line and the x-axis, and n the number of recursive applications of the modification rule that is necessary to produce C_n. If $n = 0$, no modification is done and a straight line is drawn. Otherwise, two properly shortened lines, with one rotated counter-clockwise by 45° and the other clockwise by 45° from the current line position, are generated (representing one application of the modification rule), each of which is the basis of the remaining $n - 1$ steps of recursive construction.

Koch Curve

As in the case of the C curve, a line by itself is a first-order Koch curve, denoted by K_0 (see Fig. 3.39). The modification rule for constructing successive generations of the Koch curve is to divide a line into three equal segments and replace the middle segment with two lines of the same length (the replaced segment and the two added lines form an equilateral triangle). See Fig. 3.39 for K_1, K_2, and K_3.

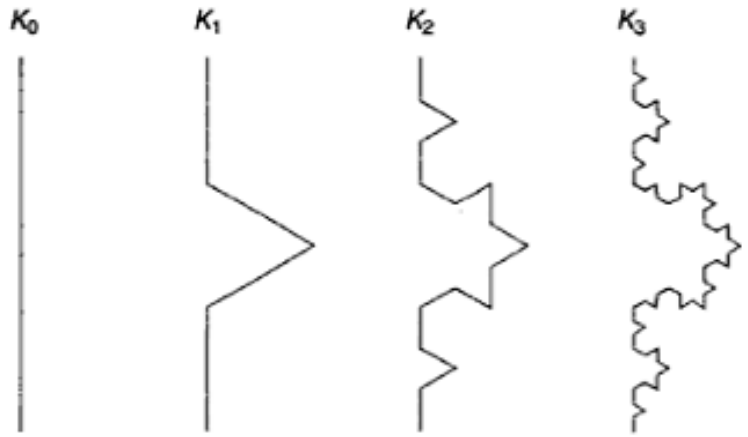


Fig. 3.39 *Successive Generations of the Koch Curve*

Sierpinski Gasket

This time our graphical primitive is a filled triangle, denoted by S_0 (see Fig. 3.40). The modification rule for constructing successive generations of the Sierpinski gasket is to take out the area defined by the lines connecting the midpoint of the edges of a filled triangle, resulting in three smaller ones that are similar to the original. See Fig. 3.40 for S_1, S_2, and S_3.

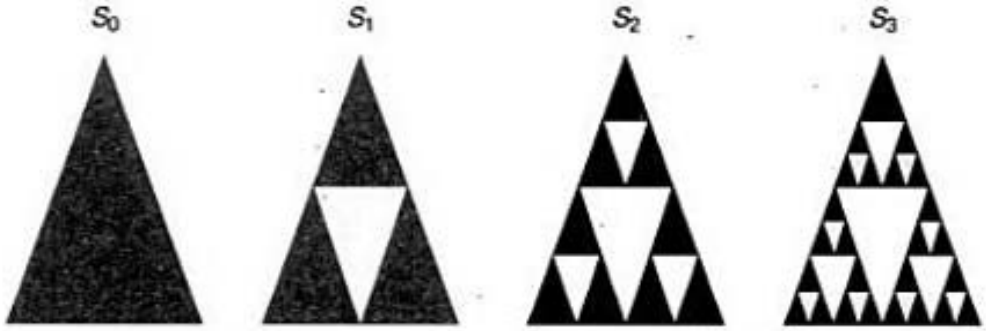


Fig. 3.40 *Successive Generations of the Sierpinski Gasket*

SOLVED PROBLEMS

3.1 The endpoints of a given line are (0, 0) and (6, 18). Compute each value of y as x steps from 0 to 6 and plot the results.

An equation for the line was not given. Therefore, the equation of the line must be found before proceeding. The equation of the line ($y = mx + b$) is found as follows.

First the slope m is found:

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{18 - 0}{6 - 0} = \frac{18}{6} = 3$$

Next, the y intercept b is found by plugging y_1 and x_1 into the equation $y = 3x + b$: $0 = 3(0) + b$. Therefore, $b = 0$. Hence the equation for the line is $y = 3x$ (see Fig. 3.41).

3.2 Write the steps required to plot a line whose slope is between 0° and 45° using the slope–intercept equation.

1. Compute dx: $dx = x_2 - x_1$.
2. Compute dy: $dy = y_2 - y_1$.
3. Compute m: $m = dy/dx$.
4. Compute b: $b = y_1 - m \times x_1$.
5. Set (x, y) equal to the lower left-hand endpoint and set x_{end} equal to the largest value of x. If $dx < 0$, then $x = x_2$, $y = y_2$, and $x_{end} = x_1$. If $dx > 0$, then $x = x_1$, $y = y_1$, and $x_{end} = x_2$.
6. Test to determine whether the entire line has been drawn. If $x > x_{end}$, stop.
7. Plot a point at the current (x, y) coordinates.
8. Increment x: $x = x + 1$.
9. Compute the next value of y from the equation $y = mx + b$.
10. Go to step 6.

3.3 Use pseudo-code to describe the steps that are required to plot a line whose slope is between 45° and −45° (i.e. $|m| > 1$) using the slope-intercept equation.

$y = 3x + 0$	x
0	0
3	1
6	2
9	3
12	4
15	5
18	6

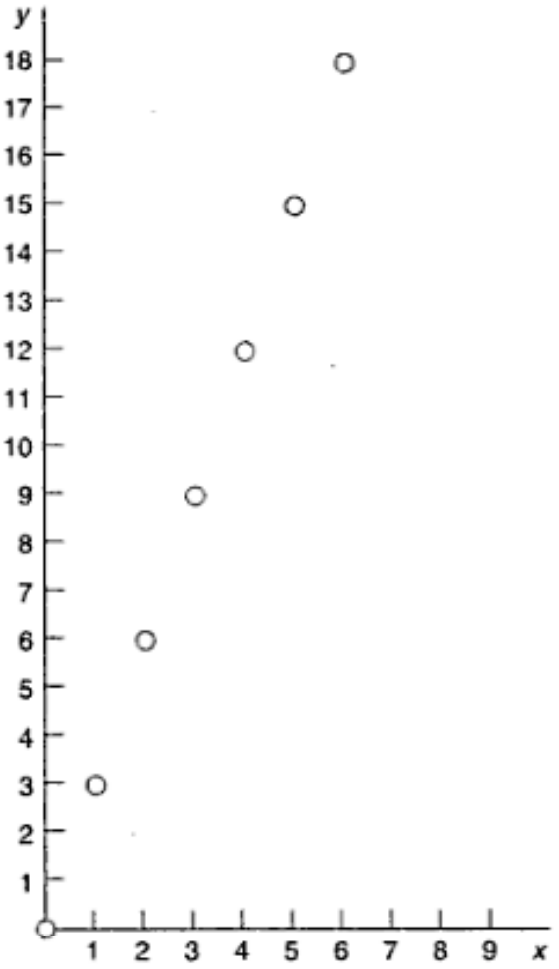


Fig. 3.41

Presume $y_1 < y_2$ for the two endpoints (x_1, y_1) and (x_2, y_2):

```
int x = x1, y = y1;
float xf, m = (y2 - y1)/(x2 - x1), b = y1 - mx1;
setPixel(x, y);
while (y < y2) {
   y ++ ;
   xf = (y - b)/m;
   x = Floor(xf + 0.5);
   setPixel(x, y);
}
```

3.4 Use pseudo-code to describe the DDA algorithm for scan-converting a line whose slope is between $-45°$ and $45°$ (i.e., $|m| \leq 1$).

Presume $x_1 < x_2$ for the two endpoints (x_1, y_1) and (x_2, y_2):

```
int x = x1, y;
float yf = y1, m = (y2 - y1)/(x2 - x1);
while (x <= x2) {
   y = Floor(yf + 0.5);
```

```
    setPixel(x, y);
    x++;
    y_f = y_f + m;
}
```

3.5 Use pseudo-code to describe the DDA algorithm for scan-converting a line whose slope is between 45° and −45° (i.e. $|m| > 1$).

Presume $y_1 < y_2$ for the two endpoints (x_1, y_1) and (x_2, y_2):

```
int x, y = y_1;
float x_f = x_1, m_inv = (x_2 - x_1)/(y_2 - y_1);
while (y <= y_2) {
    x = Floor(x_f + 0.5);
    setPixel(x, y);
    x_f = x_f + m_inv;
    y++;
}
```

3.6 What steps are required to plot a line whose slope is between 0° and 45° using Bresenham's method?

1. Compute the initial values:

$$dx = x_2 - x_1 \qquad Inc_2 = 2(dy - dx)$$
$$dy = y_2 - y_1 \qquad d = Inc_1 - dx$$
$$Inc_1 = 2dy$$

2. Set (x, y) equal to the lower left-hand endpoint and x_{end} equal to the largest value of x. If $dx < 0$, then $x = x_2$, $y = y_2$, $x_{end} = x_1$. If $dx > 0$, then $x = x_1$, $y = y_1$, $x_{end} = x_2$.
3. Plot a point at the current (x, y) coordinates.
4. Test to see whether the entire line has been drawn. If $x = x_{end}$, stop.
5. Compute the location of the next pixel. If $d < 0$, then $d = d + Inc_1$. If $d \geq 0$, then $d = d + Inc_2$, and then $y = y + 1$.
6. Increment x: $x = x + 1$.
7. Plot a point at the current (x, y) coordinates.
8. Go to step 4.

3.7 Indicate which raster locations would be chosen by Bresenham's algorithm when scan-converting a line from pixel coordinate (1, 1) to pixel coordinate (8, 5).

First, the starting values must be found. In this case

$$dx = x_2 - x_1 = 8 - 1 = 7 \qquad dy = y_2 - y_1 = 5 - 1 = 4$$

Therefore

$$Inc_1 = 2dy = 2 \times 4 = 8$$
$$Inc_2 = 2(dy - dx) = 2 \times (4 - 7) = -6$$

$$d = Inc_1 - dx = 8 - 7 = 1$$

The following table indicates the values computed by the algorithm (see also Fig. 3.42).

d	x	y
1	1	1
$1 + Inc_2 = -5$	2	2
$-5 + Inc_1 = 3$	3	2
$3 + Inc_2 = -3$	4	3
$-3 + Inc_1 = 5$	5	3
$5 + Inc_2 = -1$	6	4
$-1 + Inc_1 = 7$	7	4
$7 + Inc_2 = 1$	8	5

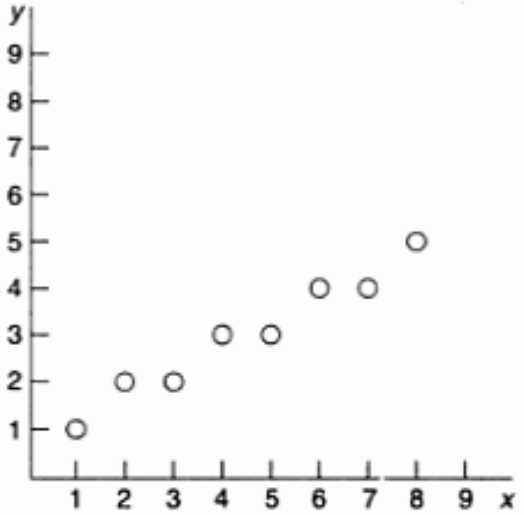


Fig. 3.42

3.8 In the derivation of Bresenham's line algorithm we have used s and t to measure the closeness of pixels S and T to the true line. However, s and t are only distances in the y direction. They are not really distances between a point to a line as defined in geometry. Can we be sure that, when $s = t$, the two pixels S and T are truly equidistant from the true line (hence we can choose either one to approximate the line)?

As we can see in Fig. 3.43, when $s = t$ the true line intersects the vertical line connecting S and T at midpoint. The true distance from S to the line is dS and that from T to the line is dT. Since the two right-angled triangles are totally equal (they have equal angles and one pair of equal edges s and t), we get dS = dT.

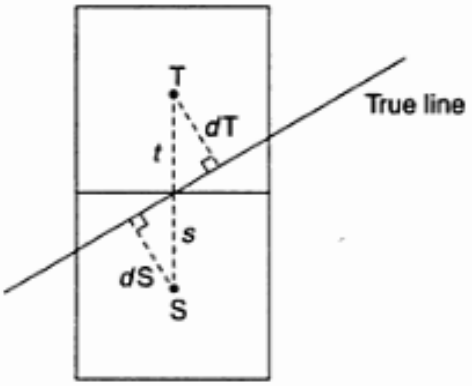


Fig. 3.43

3.9 Modify the description of Bresenham's line algorithm in the text to set all pixels from inside the loop structure.

Method 1

```
int x = x1', y = y1';
int dx = x2' - x1', dy = y2' - y1', dT = 2(dy - dx),dS = 2dy;
int d = 2dy - dx;
while (x <= x2') {
   setPixel(x, y);
   x++;
   if (d < 0)
      d = d + dS;
   else {
```

```
        y++;
        d = d + dT;
      }
    }
```

Method 2

```
  int x = x1' - 1, y = y1';
  int dx = x2' - x1', dy = y2' - y1', dT = 2(dy - dx), dS = 2dy;
  int d = -dx;
  while (x < x2') {
    x++;
    if (d < 0)
      d = d + dS;
    else {
      y++;
      d = d + dT;
    }
    setPixel(x, y);
  }
```

3.10 Write the steps required to generate a circle using the polynomial method.

1. Set the initial variables: r = circle radius; (h, k) = coordinates of the circle center; $x = 0$; i = step size; $x_{end} = r/\sqrt{2}$.
2. Test to determine whether the entire circle has been scan-converted. If $x > x_{end}$, stop.
3. Compute the value of the y coordinate, where $y = \sqrt{r^2 - x^2}$.
4. Plot the eight points, found by symmetry with respect to the center (h, k), at the current (x, y) coordinates:

Plot$(x + h, y + k)$	Plot$(x + h, -y + k)$
Plot$(y + h, x + k)$	Plot$(y + h, -x + k)$
Plot$(y + h, x + k)$	Plot$(y + h, -x + k)$
Plot$(x + h, y + k)$	Plot$(x + h, -y + k)$

5. Increment x: $x = x + i$.
6. Go to step 2.

3.11 Write the steps required to scan-convert a circle using the trigonometric method.

1. Set the initial variables: r = circle radius; (h, k) = coordinates of the circle center; i = step size; $\theta_{end} = \pi/4$ radians = 45°; $\theta = 0$.
2. Test to determine whether the entire circle has been scan-converted. If $\theta > \theta_{end}$, stop.
3. Compute the value of the x and y coordinates:

$$x = r \cos \theta \qquad y = r \sin \theta$$

4. Plot the eight points, found by symmetry with respect to the center (h, k), at the current (x, y) coordinates:

Plot($x + h$, $y + k$)	Plot($-x + h$, $-y + k$)
Plot($y + h$, $x + k$)	Plot($-y + h$, $-x + k$)
Plot($-y + h$, $x + k$)	Plot($y + h$, $-x + k$)
Plot($-x + h$, $y + k$)	Plot($x + h$, $-y + k$)

5. Increment θ: $\theta = \theta + i$.
6. Go to step 2.

3.12 Write the steps required to scan-convert a circle using Bresenham's algorithm.

1. Set the initial values of the variables: (h, k) = coordinates of circle center; $x = 0$; y = circle radius r; and $d = 3 - 2r$.
2. Test to determine whether the entire circle has been scan-converted. If $x > y$, stop.
3. Plot the eight points, found by symmetry with respect to the center (h, k), at the current (x, y) coordinates:

Plot($x + h$, $y + k$)	Plot($-x + h$, $-y + k$)
Plot($y + h$, $x + k$)	Plot($-y + h$, $-x + k$)
Plot($-y + h$, $x + k$)	Plot($y + h$, $-x + k$)
Plot($-x + h$, $y + k$)	Plot($x + h$, $-y + k$)

4. Compute the location of the next pixel. If $d < 0$, then $d = d + 4x + 6$ and $x = x + 1$. If $d \geq 0$, then $d = d + 4(x - y) + 10$, $x = x + 1$, and $y = y - 1$.
5. Go to step 2.

3.13 When eight-way symmetry is used to obtain a full circle from pixel coordinates generated for the 0° to 45° or the 90° to 45° octant, certain pixels are set or plotted twice. This phenomenon is sometimes referred to as overstrike. Identify the locations where overstrike occurs.

At locations resulted from the initial coordinates $(r, 0)$ or $(0, r)$ since $(0, r) = (-0, r)$, $(0, -r) = (-0, -r)$, $(r, 0) = (r, -0)$, and $(-r, 0) = (-r, -0)$.

In addition, if the last generated pixel is on the diagonal line at $(\alpha r, \alpha r)$ where α approximates $1/\sqrt{(2.0)} = 0.7071$, then overstrike also occurs at $(\alpha r, \alpha r)$, $(-\alpha r, \alpha r)$, $(\alpha r, -\alpha r)$ and $(-\alpha r, -\alpha r)$.

3.14 Is overstrike harmful besides wasting time?

It is often harmless since resetting a pixel with the same value does not really change the image in the frame buffer. However, if pixel values are sent out directly, for example, to control the exposure of a photographic medium, such as a slide or a negative, then overstrike amounts to double exposure at locations where it occurred.

Furthermore, if we set pixels using their complementary colors, then overstrike would leave them unchanged, since complementing a color twice simply yields the color itself.

Chapter Four

Two-Dimensional Transformations

Fundamental to all computer graphics systems is the ability to simulate the manipulation of objects in space. This simulated spatial manipulation is referred to as *transformation*. The need for transformation arises when several objects, each of which is independently defined in its own coordinate system, need to be properly positioned into a common scene in a master coordinate system. Transformation is also useful in other areas of the image synthesis process e.g. viewing transformation (Chapter 5).

There are two complementary points of view for describing object transformation. The first is that the object itself is transformed relative to a stationary coordinate system or background. The mathematical statement of this viewpoint is described by *geometric transformations* applied to each point of the object. The second point of view holds that the object is held stationary while the coordinate system is transformed relative to the object. This effect is attained through the application of *coordinate transformations*. An example that helps to distinguish these two viewpoints involves the movement of an automobile against a scenic background. We can simulate this by moving the automobile while keeping the backdrop fixed (a geometric transformation), or we can keep the car fixed while moving the backdrop scenery (a coordinate transformation).

This chapter covers transformations in the plane, i.e. the two-dimensional (2D) space. We detail three basic transformations: translation, rotation and scaling, along with other transformations that can be accomplished in terms of a sequence of basic transformations. We describe these operations in mathematical form suitable for computer processing and show how they are used to achieve the ends of object manipulation.

4.1 GEOMETRIC TRANSFORMATIONS

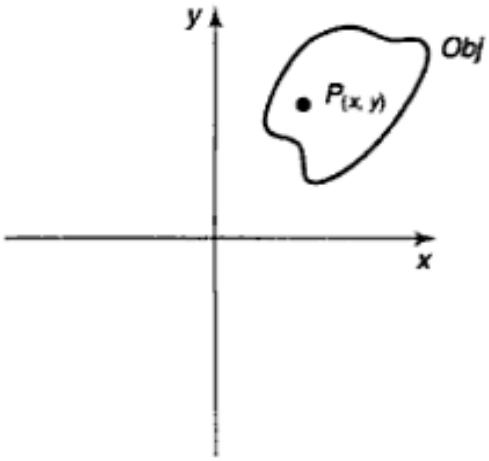


Fig. 4.1

Let us impose a coordinate system on a plane. An object *Obj* in the plane can be considered as a set of points. Every object point P has coordinates (x, y), and so the object is the sum total of all its coordinate points (Fig. 4.1). If the object is moved to a new position, it can be regarded as a new object *Obj'*, all of whose coordinate points P' can be obtained from the original points P by the application of geometric transformation.

Translation

In *translation*, an object is displaced a given distance and direction from its original position. If the displacement is given by the vector $\mathbf{v} = t_x\mathbf{I} + t_y\mathbf{J}$, the new object point $P'(x', y')$ can be found by applying the transformation $T_\mathbf{v}$ to $P(x, y)$ (see Fig. 4.2).

$$P' = T_\mathbf{v}(P)$$

where $x' = x + t_x$ and $y' = y + t_y$.

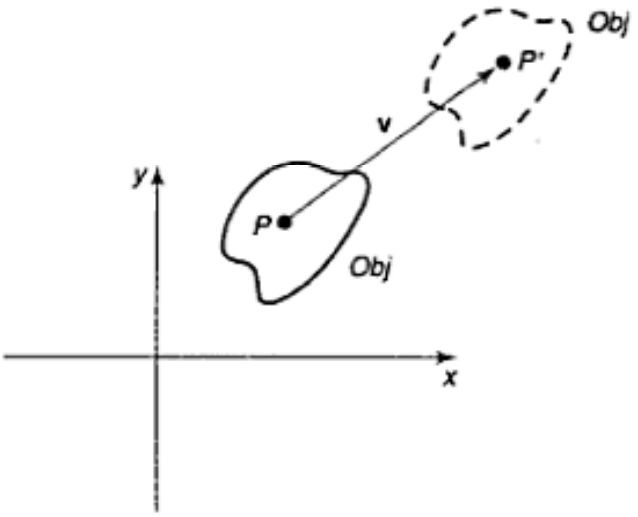


Fig. 4.2

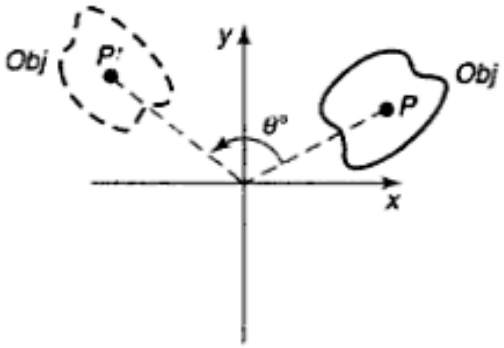


Fig. 4.3

Rotation about the Origin

In *rotation*, the object is rotated $\theta°$ about the origin. The convention is that the direction of rotation is counterclockwise if θ is a positive angle, and clockwise if θ is a negative angle (see Fig. 4.3). The transformation of rotation R_θ is

$$P' = R_\theta(P)$$

where $x' = x\cos(\theta) - y\sin(\theta)$ and $y' = x\sin(\theta) + y\cos(\theta)$.

Scaling with Respect to the Origin

Scaling is the process of expanding or compressing the dimensions of an object. Positive scaling constants s_x and s_y are used to describe changes in length with respect to the x direction and y direction, respectively. A scaling constant greater than one indicates an expansion of length, and less than one, compression of length. The scaling transformation S_{s_x,s_y} is given by $P' = S_{s_x,s_y}(P)$ where $x' = s_x x$ and $y' = s_y y$. Notice that, after a scaling transformation is performed, the new object is located at a different position relative to the origin. In fact, in a scaling transformation the only point that remains fixed is the origin. Figure 4.4 shows scaling transformation with scaling factors $s_x = 2$ and $s_y = \frac{1}{2}$

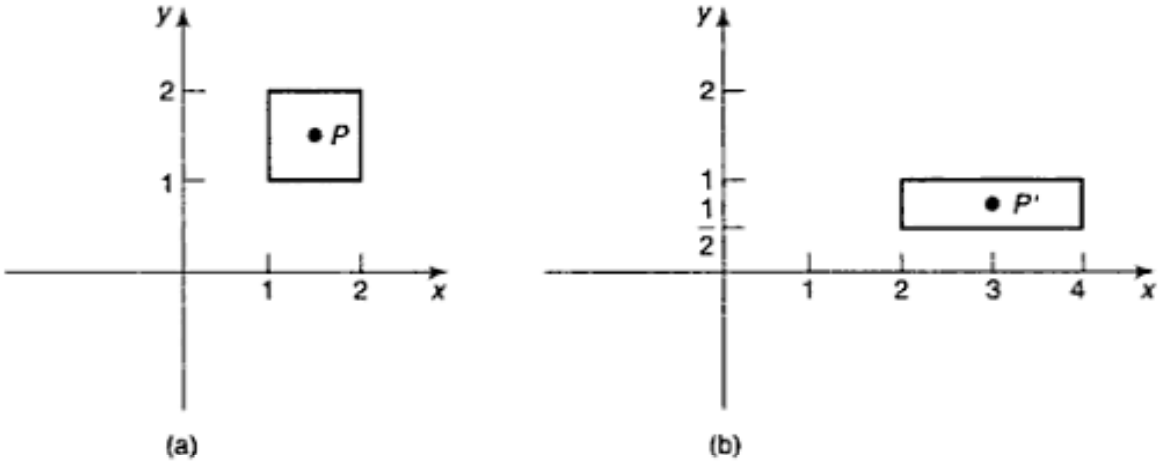


Fig. 4.4

If both scaling constants have the same value s, the scaling transformation is said to be *homogeneous* or *uniform*. Furthermore, if $s > 1$, it is a *magnification* and for $s < 1$, a *reduction*.

Mirror Reflection about an Axis

If either the x or y axis is treated as a mirror, the object has a mirror image or reflection. Since the reflection P' of an object point P is located the same distance from the mirror as P (Fig. 4.5), the mirror reflection transformation M_x about the x axis is given by

$$P' = M_x(P)$$

where $x' = x$ and $y' = -y$.

Similarly, the mirror reflection about the y axis is

$$P' = M_y(P)$$

where $x' = -x$ and $y' = y$.

Note that $M_x = S_{1,-1}$ and $M_{xy} = S_{-1,-1}$. The two reflection transformations are simply special cases of scaling.

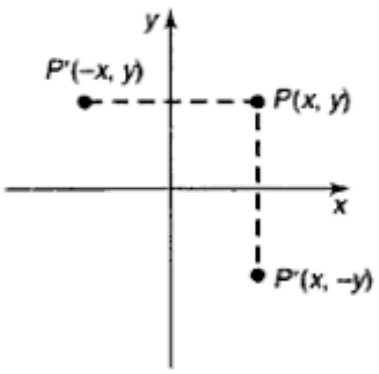


Fig. 4.5

Inverse Geometric Transformations

Each geometric transformation has an inverse (see Appendix 1) which is described by the opposite operation performed by the transformation:

Translation: $T_{\mathbf{v}}^{-1} = T_{-\mathbf{v}}$ or translation in the opposite direction

Rotation: $R_{\theta}^{-1} = R_{-\theta}$, or rotation in the opposite direction

Scaling: $S_{s_x,s_y}^{-1} = S_{1/s_x,1/s_y}$

Mirror reflection: $M_x^{-1} = M_x$ and $M_y^{-1} = M_y$.

4.2 COORDINATE TRANSFORMATIONS

Suppose that we have two coordinate systems in the plane. The first system is located at origin O and has coordinates axes xy. The second coordinate system is located at origin O' and has coordinate axes $x'y'$ (Fig. 4.6). Now each point in the plane has two coordinate descriptions: (x, y) or (x', y'), depending on which coordinate system is used. If we think of the second system $x'y'$ as arising from a transformation applied to the first system xy, we say that a *coordinate transformation* has been applied. We can describe this transformation by determining how the (x', y') coordinates of a point P are related to the (x, y) coordinates of the same point.

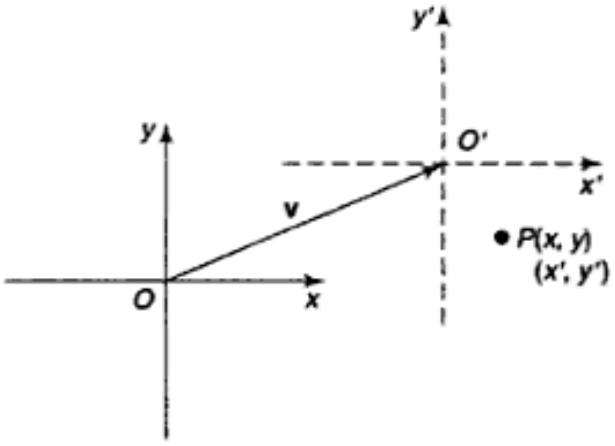


Fig. 4.6

Translation

If the xy coordinate system is displaced to a new position, where the direction and distance of the displacement is given by the vector $\mathbf{v} = t_x\mathbf{I} + t_y\mathbf{J}$, the coordinates of a point in both systems are related by the translation transformation $\bar{T}_{\mathbf{v}}$:

$$(x', y') = \bar{T}_{\mathbf{v}}(x, y)$$

where $x' = x - t_x$ and $y' = y - t_y$.

Rotation about the Origin

The xy system is rotated $\theta°$ about the origin (see Fig. 4.7). Then the coordinates of a point in both systems are related by the rotation transformation $\bar{R}_{\theta}$:

$$(x', y') = \bar{R}_{\theta}(x, y)$$

where $x' = x\cos(\theta) + y\sin(\theta)$ and $y' = -x\sin(\theta) + y\cos(\theta)$.

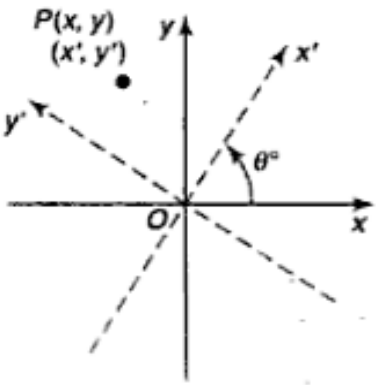


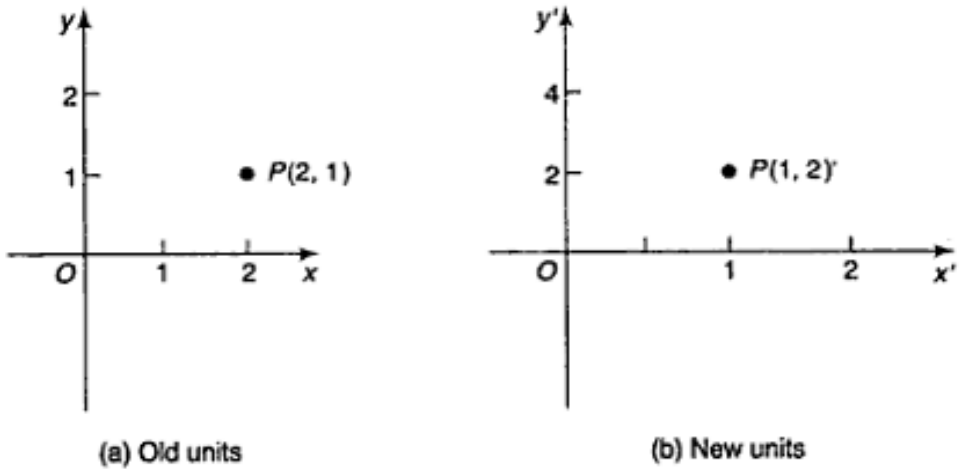
Fig.4.7

Scaling with Respect to the Origin

Suppose that a new coordinate system is formed by leaving the origin and coordinate axes unchanged, but introducing different units of measurement along the x and y axes. If the new units are obtained from the old units by a scaling of s_x along the x axis and s_y along the y axis, the coordinates in the new system are related to coordinates in the old system through the scaling transformation $\bar{S}_{s_x, s_y}$:

$$(x', y') = \bar{S}_{s_x, s_y}(x, y)$$

where $x' = (1/s_x)x$ and $y' = (1/s_y)y$. Figure 4.8 shows coordinate scaling transformation using scaling factors $s_x = 2$ and $s_y = \frac{1}{2}$.



(a) Old units (b) New units

Fig. 4.8

Mirror Reflection about an Axis

If the new coordinate system is obtained by reflecting the old system about either x or y axis, the relationship between coordinates is given by the coordinate transformations $\bar{M}_x$ and $\bar{M}_y$. For reflection about the x axis [Fig. 4.9(a)]

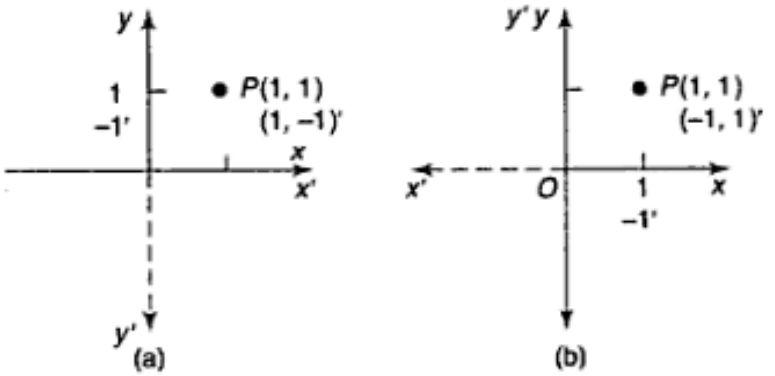


Fig. 4.9

$$(x', y') = \bar{M}_x(x, y)$$

where $x' = x$ and $y' = -y$. For reflection about the y axis (Fig. 4.9)

$$(x', y') = \bar{M}_y(x, y)$$

where $x' = -x$ and $y' = y$.

Notice that the reflected coordinate system is left-handed; thus reflection changes the orientation of the coordinate system. Also note that $\bar{M}_x = \bar{S}_{1,-1}$ and $\bar{M}_y = \bar{S}_{-1,1}$.

Inverse Coordinate Transformations

Each coordinate transformation has an inverse (see Appendix 1) which can be found by applying the opposite transformation:

Translation: $\bar{T}_{\mathbf{v}}^{-1} = \bar{T}_{-\mathbf{v}}$, translation in the opposite direction

Rotation: $\bar{R}_{\theta}^{-1} = \bar{R}_{-\theta}$, rotation in the opposite direction

Scaling: $\bar{S}_{s_x,s_y}^{-1} = \bar{S}_{1/s_x,1/s_y}$

Mirror reflection: $\bar{M}_x^{-1} = \bar{M}_x$ and $\bar{M}_y^{-1} = \bar{M}_y$

4.3 COMPOSITE TRANSFORMATIONS

More complex geometric and coordinate transformations can be built from the basic transformations described above by using the process of *composition of functions* (see Appendix 1). For example, such operations as rotation about a point other than the origin or reflection about lines other than the axes can be constructed from the basic transformations.

Example 4.1

Magnification of an object while keeping its center fixed (see Fig. 4.10). Let the geometric center be located at $C(h, k)$ [Fig. 4.10(a)]. Choosing a magnification factor $s > 1$, we construct the transformation by performing the following sequence of basic transformations: (1) translate the object so that its center coincides with the origin [Fig. 4.10(b)], (2) scale the object with respect to the origin [Fig. 4.10(c)], and (3) translate the scaled object back to the original position [Fig. 4.10(d)].

The required transformation $S_{s,C}$ can be formed by compositions $S_{s,C} = T_{\mathbf{v}}^{-1} \cdot S_{s,s} \cdot T_{\mathbf{v}}$ where $\mathbf{v} = h\mathbf{I} + k\mathbf{J}$. By using composition, we can build more general and reflection transformations. For these transformations, we shall use the following notations: (1) $S_{s_x,s_y,P}$—scaling with respect to a fixed point P, (2) $R_{\theta,P}$—rotation about a point P; and (3) M_L—reflection about a line L.

The matrix description of these transformations can be found in Problems. 4.4, 4.7, and 4.10.

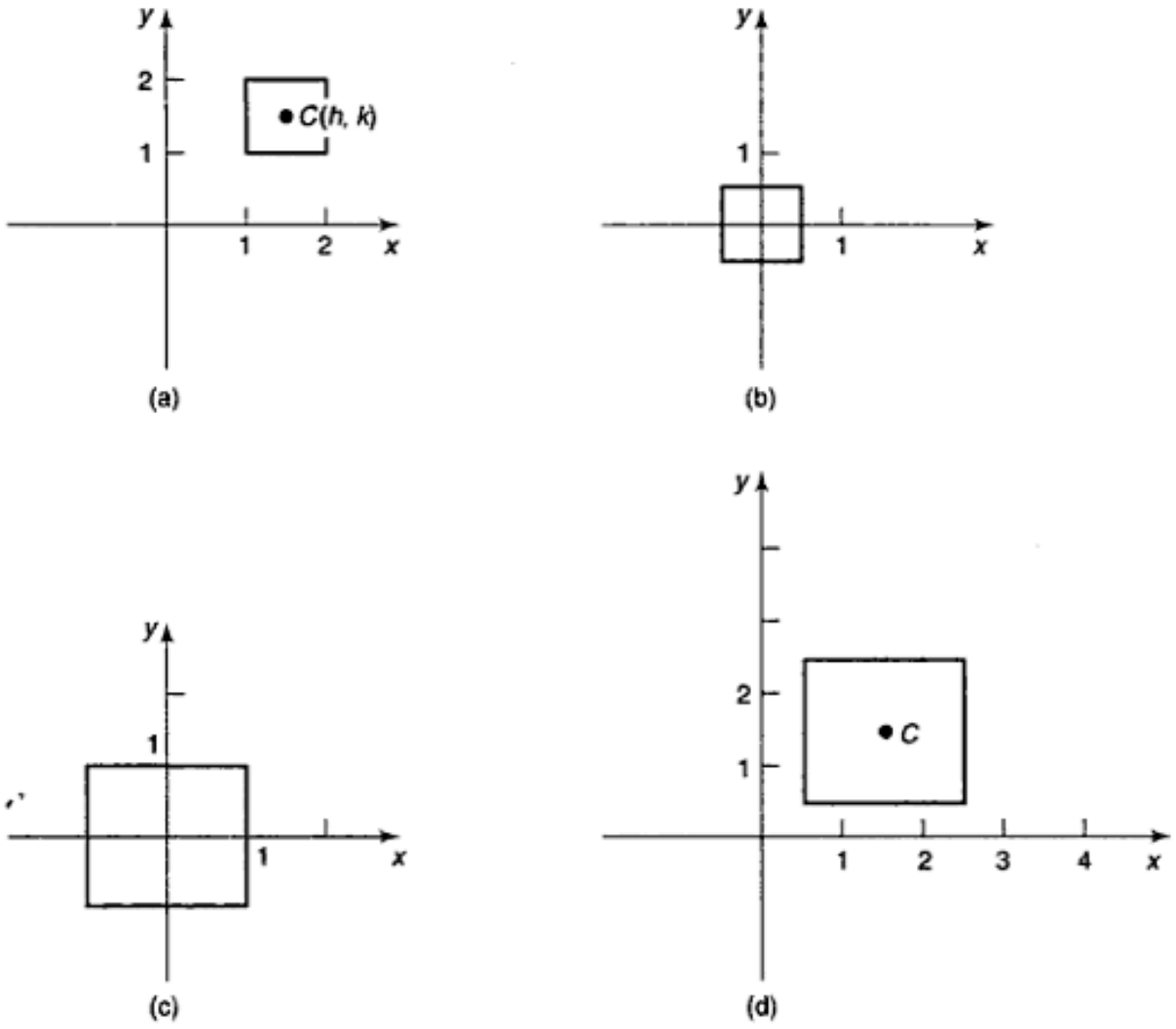


Fig. 4.10

Matrix Description of the Basic Transformations

The transformations of rotation, scaling, and reflection can be represented as matrix functions:

Geometric Transformations	Coordinate Transformations
$R_\theta = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$	$\bar{R}_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$
$S_{s_x,s_y} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$	$\bar{S}_{s_x,s_y} = \begin{pmatrix} \frac{1}{s_x} & 0 \\ 0 & \frac{1}{s_y} \end{pmatrix}$
$M_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\bar{M}_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

$$M_y = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \bar{M}_y = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

The translation transformation cannot be expressed as a 2×2 matrix function. However, using a methodology called *homogeneous coordinates* the transformations can be introduced as 3×3 matrix functions.

Homogeneous Coordinates

Homogeneous coordinates is a technique based on projective geometry. Any point (a, b) in two dimensional Cartesian system is represented as $(a, b, 1)$ in homogeneous coordinate system. Further any point (a, b, w) when $w \neq 0$ in homogeneous coordinates corresponds the point $(a/w, b/w)$ in two dimensional Cartesian system. This correspondence is logical because the point in (a, b) in two dimensional is equivalent to $(a, b, 0)$ in three dimensional on $z = 0$ plane which is projected to $z = 1$ plane to be equivalent to $(a, b, 1)$. Similarly a point (a, b, c) in 3D Cartesian system is modelled as $(a, b, c, 1)$ in homogeneous coordinates and a point (a, b, c, w) in homogeneous coordinates is mapped to $(a/w, b/w, c/w)$ when $w \neq 0$.
The advantages of the representation in homogeneous coordinates are as follows.

1. The two dimensional transformations can be represented as 3×3 matrices so that usage of the operations becomes relatively easy.
2. The floating point arithmetic can be avoided sometimes by transforming them into integer arithmetic. For example, a point (1/2, 1/3) in 2D can be represented as (3, 2, 6) in homogeneous coordinates

We represent the coordinate pair (x, y) of a point P by the triple $(x, y, 1)$ in homogeneous coordinates. Then translation in the direction $\mathbf{v} = t_x\mathbf{I} + t_y\mathbf{J}$ can be expressed by the matrix function

$$T_v = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{pmatrix}$$

Then

$$(x \quad y \quad 1)\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{pmatrix} = (x+t_x \quad y+t_y \quad 1)$$

From this we extract the coordinate pair $(x + t_x, y + t_y)$.

The Rotation matrix after introducing homogeneous coordinates will be $\mathrm{R}_\theta = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$

$$\text{as } (x \quad y \quad 1)\begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} = (x\cos\theta - y\sin\theta \quad x\sin\theta + y\cos\theta \quad 1)$$

The scaling transformation in homogeneous coordinates becomes, $S_{s_x s_y} = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$

$$\text{as } (x \quad y \quad 1)\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} = (xS_x \quad yS_y \quad 1)$$

Concatenation of Matrices

The advantage of introducing a matrix form for translation is that we can now build complex transformations by multiplying the basic matrix transformations. This process is sometimes called *concatenation of matrices* and the resulting matrix is often referred to as the *composite transformation matrix* (CTM). Here, we are using the fact that the composition of matrix functions is equivalent to matrix multiplication (Appendix 1). We must be able to represent the basic transformations as 3×3 *homogeneous coordinate matrices* so as to be compatible (from the point of view of matrix multiplication) with the matrix of translation. This is accomplished by augmenting the 2×2 matrices with a third column $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ and a third row $(0 \quad 0 \quad 1)$. The homogeneous matrix therefore becomes

$$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Example 4.2

Express as a composite transformation matrix (i.e. CTM) the transformation which magnifies an object about its center $C(h, k)$. From Example 4.1, the required transformation $S_{s,C}$ can be written as

$$S_{s,C} = T_v^{-1} \cdot S_{s,s} \cdot T_v$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -h & -k & 1 \end{pmatrix}\begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ h & k & 1 \end{pmatrix} = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ -sh+h & -sk+k & 1 \end{pmatrix}$$

Shear Transformation

There are two more transformations which are useful in graphics applications, namely the X-shear and Y-shear, called the *shear transformations*. The shear transformations cause the image to slant. X-shear maintains the y-coordinates but changes the x values which cause the vertical lines to tilt left or right. The Y-shear preserves all the x coordinate values but shifts the y coordinate. This causes horizontal lines to transform into lines which slope up or down.

It is possible to form the shear transformations using the transformations of rotation and scaling. However it is easy to use the matrix form directly. Similarly it is also possible to built rotation and scaling transformations out of shear transformations.

The X-shear transformation in the homogeneous matrix form is given by

$$Sh_x = \begin{pmatrix} 1 & 0 & 0 \\ a & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ where } a \text{ is the X-shear factor.}$$

When b is the Y-shear factor, the homogeneous matrix transformation of Y-shear is

$$Sh_y = \begin{pmatrix} 1 & b & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The effects of these are shown in Fig. 4.11.

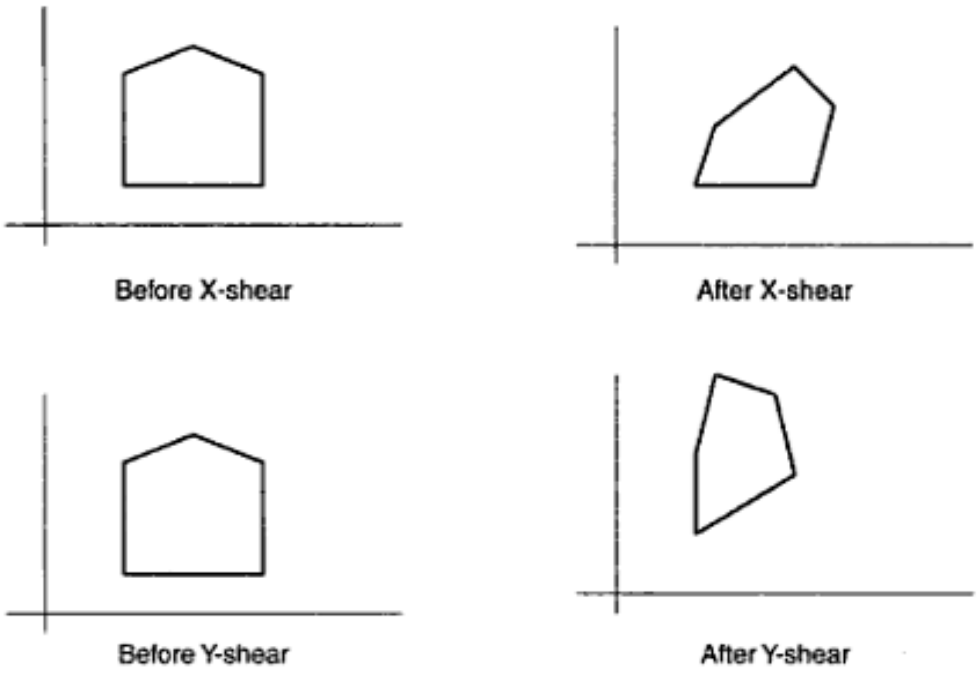


Fig. 4.11

4.4 INSTANCE TRANSFORMATIONS

Quite often a picture or design is composed of many objects used several times each. In turn, these objects may also be composed of other symbols and objects. We suppose that each object is defined, independently of the picture, in its own coordinate system. We wish to place these objects together to form the picture or at least part of the picture, called a *subpicture*. We can accomplish this by defining a transformation of coordinates, called an *instance transformation*, which converts object coordinates to picture coordinates so as to place or create an *instance* of the object in the picture coordinate system.

The *instance transformation* $N_{\text{picture, object}}$ is formed as a composition or concatenation of scaling, rotation, and translation operations, usually performed in this order (although any order can be used)

$$N_{\text{picture, object}} = S_{a,b,p} \cdot R_{\theta, P} \cdot T_v$$

With the use of different instance transformations, the same object can be placed in different positions, sizes, and orientations within a subpicture. For instance, Fig. 4.12(a) is placed in the picture coordinate system of Fig. 4.12(b) by using the instance transformations $N_{\text{picture, object}}$.

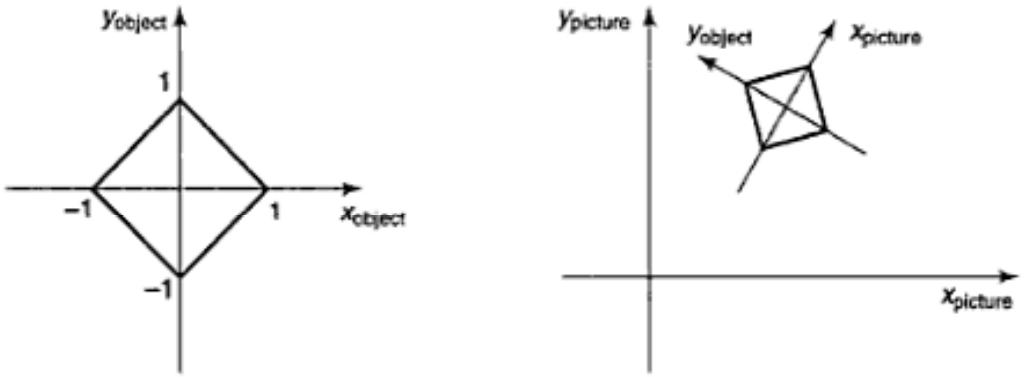


Fig. 4.12

Nested Instances and Multilevel Structures

A subpicture or picture may exhibit a multilevel or nested structure by being composed of objects which are, in turn, composed of still other objects, and so on. Separate instance transformations must then be applied, in principle, at each level of the picture structure for each picture component.

Example 4.3

A picture of an apple tree contains branches, and an apple hangs on each branch. Suppose that each branch and apple is described in its own coordinate system [Figs 4.13 (a) and 4.13 (b)]. Then a sub-picture function call to place an instance of this branch in the picture of the tree requires an additional sub-picture function call to place an instance of the apple into the branch coordinate system.

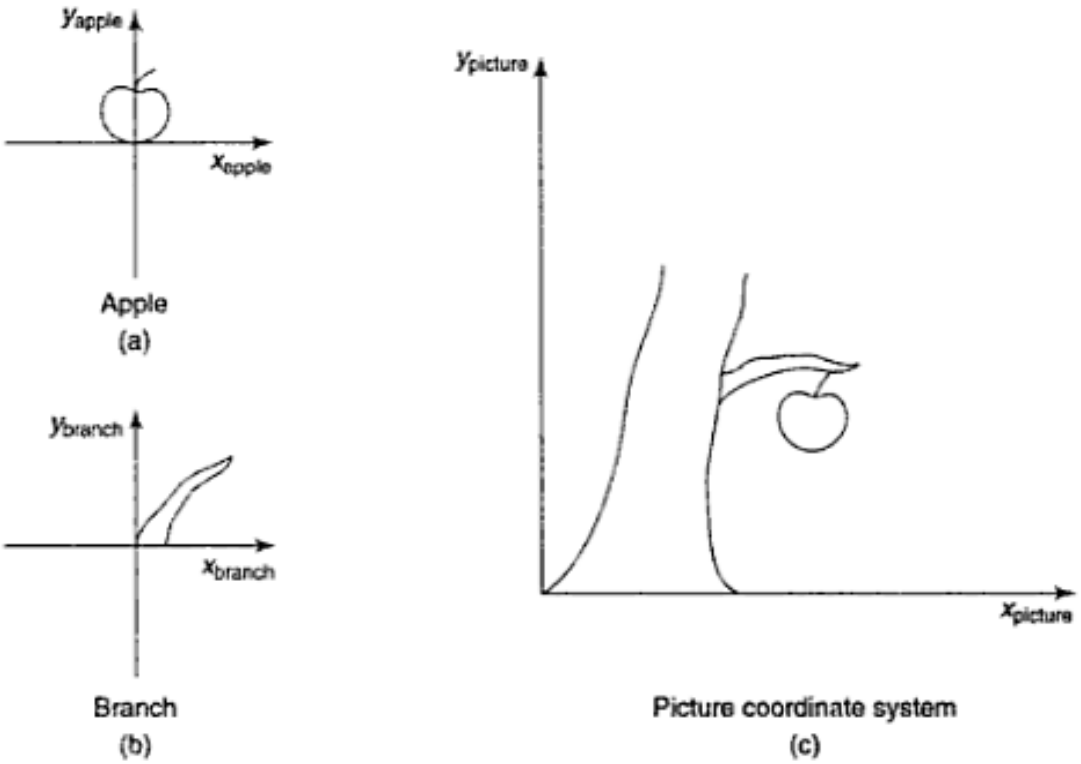


Fig. 4.13

We can perform each instance transformation separately, i.e., instance the apple in the branch coordinate system and then instance both branch and apple from the branch coordinate system to the picture coordinate system. However, it is much more efficient to transform the apple directly into picture coordinates [see Fig. 4.13(c)]. This is accomplished by defining the composite transformation matrix $C_{picture,object}$ to be the composition of the nested instance transformations from apple coordinates to branch coordinates and then from branch coordinates to picture coordinates:

$$C_{picture,apple} = N_{branch,apple} \cdot N_{picture,branch}$$

Since the branch subpicture is only one level below the picture

$$C_{picture,branch} = N_{picture,branch}$$

SOLVED PROBLEMS

4.1 Derive the transformation that rotates an object point $\theta°$ about the origin. Write the matrix representation for this rotation.

Refer to Fig. 4.14. Definition of the trigonometric functions sin and cos yields

$$x' = r\cos(\theta + \phi) \qquad y' = r\sin(\phi + \theta) \text{ and}$$
$$x = r\cos\theta \qquad y = r\sin\phi$$

Using trigonometric identities, we obtain

$$r\cos(\theta + \phi) = r(\cos\theta\cos\phi - \sin\phi\sin\theta) = x\cos\theta - y\sin\theta$$

and

$$r \sin(\theta + \phi) = r(\sin\theta\cos\phi + \cos\theta\sin\phi)$$
$$= x\sin\theta - y\cos\theta$$

or

$$x' = x\cos\theta - y\sin\theta \quad y' = x\sin\theta + y\cos\theta$$

Writing $P' = (x', y')$, $P = (x, y)$, and

$$R_\theta = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

We can now write $P' = P \cdot R_\theta$

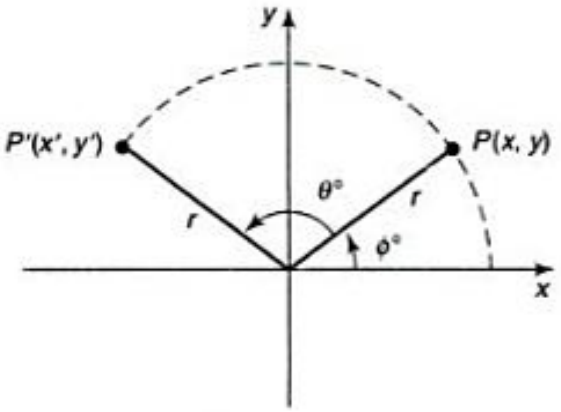


Fig. 4.14

4.2 (a) Find the matrix that represents rotation of an object by 30° about the origin.
(b) What are the new coordinates of the point $P(2, -4)$ after the rotation?

(a) From Problem 4.1:

$$R_{30^\circ} = \begin{pmatrix} \cos 30^\circ & \sin 30^\circ \\ -\sin 30^\circ & \cos 30^\circ \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix}$$

(b) The new coordinates can be found by multiplying:

$$(2 \quad -4)\begin{pmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix} = \left(\sqrt{3}+2 \quad 1-2\sqrt{3}\right)$$

4.3 Describe the transformation that rotates an object point, $Q(x, y)$, θ° about a fixed center of rotation $P(h, k)$ (Fig. 4.15).

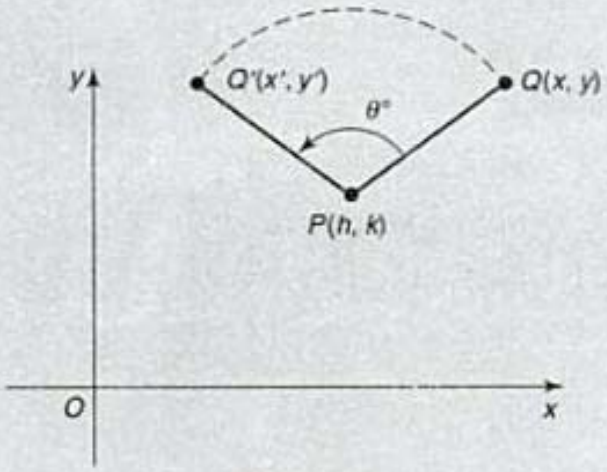


Fig. 4.15

We determine the transformation $R_{\theta,P}$ in three steps:

1. Translate so that the center of rotation P is at the origin,
2. Perform a rotation of θ degrees about the origin, and
3. Translate P back to (h, k).

Using $\mathbf{v} = h\mathbf{I} + k\mathbf{J}$ as the translation vector, we build $R_{\theta,P}$ by composition of transformations:

$$R_{\theta,0} = T_{-v} \cdot R_{\theta} \cdot T_{v}$$

4.4 Write the general form of the matrix for rotation about a point $P(h, k)$.

Following Solved Problem. 4.3, we write $R_{\theta,P} = T_{-v} \cdot R_{\theta} \cdot T_{v}$, where $\mathbf{V} = h\mathbf{I} + k\mathbf{J}$. Using the 3×3 homogeneous coordinate form for the rotation and translation matrices, we have

$$R_{\theta,P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -h & -k & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ h & k & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ -h\cos\theta + k\sin\theta + h & -h\sin\theta - k\cos\theta + k & 1 \end{pmatrix}$$

4.5 Perform a 45° rotation of triangle $A(0,0)$, $B(1, 1)$, $C(5, 2)$

(a) about the origin and
(b) about $P(-1, -1)$.

We represent the triangle by a matrix formed from the homogeneous coordinates of the vertices:

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 5 & 2 & 1 \end{pmatrix}$$

(a) The matrix of rotation is

$$R_{45^\circ} = \begin{pmatrix} \cos 45^\circ & \sin 45^\circ & 0 \\ -\sin 45^\circ & \cos 45^\circ & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ \frac{-\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

So the coordinates $A'B'C'$ of the rotated triangle ABC can be found as

$$\begin{bmatrix} A' \\ B' \\ C' \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot R_{45^\circ} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 5 & 2 & 1 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ \frac{-\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{matrix} A' \\ B' \\ C' \end{matrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & \sqrt{2} & 1 \\ \frac{3\sqrt{2}}{2} & \frac{7\sqrt{2}}{2} & 1 \end{pmatrix}$$

Thus $A' = (0, 0)$, $B' = (0, \sqrt{2})$, and $C' = \left(\frac{3}{2}\sqrt{2}, \frac{7}{2}\sqrt{2}\right)$

(b) From Problem 4.4, the rotation matrix is given by $R_{45^\circ, p} = T_{-v} \cdot R_{45^\circ} \cdot T_v$, where $\mathbf{V} = -\mathbf{I} - \mathbf{J}$. So

$$R_{45^\circ, p} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ \frac{-\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ \frac{-\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ -1 & (\sqrt{2}-1) & 1 \end{pmatrix}$$

Now

$$\begin{bmatrix} A' \\ B' \\ C' \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot R_{45^\circ, p} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 5 & 2 & 1 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ \frac{-\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ -1 & (\sqrt{2}-1) & 1 \end{pmatrix}$$

$$= \begin{pmatrix} -1 & (\sqrt{2}-1) & 1 \\ -1 & (2\sqrt{2}-1) & 1 \\ \left(\frac{3}{2}\sqrt{2}-1\right) & \left(\frac{9}{2}\sqrt{2}-1\right) & 1 \end{pmatrix}$$

So $A' = (-1, \sqrt{2} - 1)$, $B' = (-1, 2\sqrt{2} - 1)$, and $C' = \left(\frac{3}{2}\sqrt{2}-1, \frac{9}{2}\sqrt{2}-1\right)$

4.6 Find the transformation that scales (with respect to the origin) by

(a) a units in the X-direction,
(b) b units in the Y-direction, and
(c) simultaneously a units in the X-direction and b units in the Y-direction.

(a) The scaling transformation applied to a point $P(x, y)$ produces the point (ax, y). We can write this in matrix form as $P \cdot S_{a,1}$ or

$$(x \quad y)\begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix} = (ax \quad y)$$

(b) As in part (a), the required transformation can be written in matrix form as $P \cdot S_{1,b}$. So

$$(x \quad y)\begin{pmatrix} 1 & 0 \\ 0 & b \end{pmatrix} = (x \quad by)$$

(c) Scaling in both directions is described by the transformation $x' = ax$ and $y' = by$. Writing this in matrix form as $P \cdot S_{a,b}$, we have

$$(x \quad y)\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} = (ax \quad by)$$

4.7 Write the general form of a scaling matrix with respect to a fixed point $P(h, k)$.

Following the same general procedure as in Problems 4.3 and 4.4, we write the required transformation with $\mathbf{v} = h\mathbf{I} + k\mathbf{J}$ as

$$S_{a,b,P} = T_{-v} \cdot S_{a,b} \cdot T_v$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -h & -k & 1 \end{pmatrix}\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ h & k & 1 \end{pmatrix}$$

$$= \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ -ah + h & -bk + k & 1 \end{pmatrix}$$

4.8 Magnify the triangle with vertices $A(0, 0)$, $B(1, 1)$, and $C(5, 2)$ to twice its size while keeping $C(5, 2)$ fixed.

From Problem. 4.7, we can write the required transformation with $\mathbf{v} = 5\mathbf{I} + 2\mathbf{J}$ as

$$S_{2,2,C} = T_{-v} \cdot S_{2,2} \cdot T_v$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -5 & -2 & 1 \end{pmatrix}\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 2 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 5 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ -5 & -2 & 1 \end{pmatrix}$$

Representing a point P with coordinates (x, y) by the row vector $(x \quad y \quad 1)$, we have

$$A \cdot S_{2,2,C} = (0 \quad 0 \quad 1)\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ -5 & -2 & 1 \end{pmatrix} = (-5 \quad -2 \quad 1)$$

$$B \cdot S_{2,2,C} = (1 \quad 1 \quad 1)\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ -5 & -2 & 1 \end{pmatrix} = (-3 \quad 0 \quad 1)$$

$$C \cdot S_{2,2,C} = (5 \quad 2 \quad 1)\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ -5 & -2 & 1 \end{pmatrix} = (5 \quad 2 \quad 1)$$

So $\bar{A}' = (-5, -2)$, $\bar{B}' = (-3, 0)$, and $\bar{C}' = (5, 2)$. Note that, since the triangle ABC is completely determined by its vertices, we could have saved much writing by representing the vertices using a 3×3 matrix

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 5 & 2 & 1 \end{pmatrix}$$

and applying $S_{2,2,C}$ this. So

$$\begin{bmatrix} A \\ B \\ C \end{bmatrix} S_{2,2,C} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 5 & 2 & 1 \end{pmatrix}\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ -5 & -2 & 1 \end{pmatrix} = \begin{pmatrix} -5 & -2 & 1 \\ -3 & 0 & 1 \\ 5 & 2 & 1 \end{pmatrix} = \begin{bmatrix} A' \\ B' \\ C' \end{bmatrix}$$

4.9 Describe the transformation M_L which reflects an object about a line L.

Let line L in Fig. 4.16 have ay intercept $(0, b)$ and an angle of inclination $\theta°$ (with respect to the x axis). We reduce the description to known transformations:

1. Translate the intersection point B to the origin.
2. Rotate by $-\theta°$ so that line L aligns with the x axis.
3. Mirror-reflect about the x axis.
4. Rotate back by $\theta°$.
5. Translate B back to $(0, b)$.

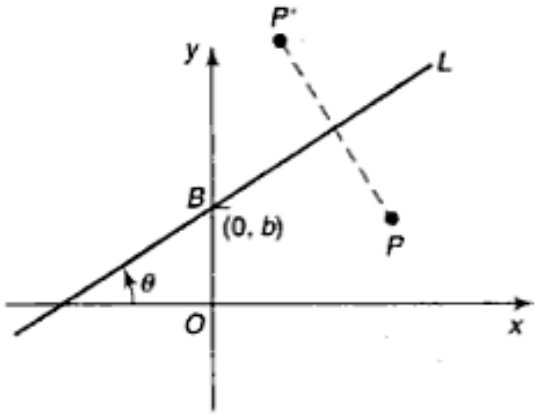


Fig. 4.16

In transformation notation, we have

$$M_L = T_{-v} R_{-\theta} M_x R_\theta T_v$$

where $\mathbf{v} = b\mathbf{J}$

4.10 Find the form of the matrix for reflection about a line L with slope m and y intercept $(0, b)$.

Following Solved Problem 4.9 and applying the fact that the angle of inclination of a line is related to its slope m by the equation $\tan\theta = m$, we have with $\mathbf{v} = b\mathbf{J}$,

$$M_L = T_{-v} R_{-\theta} M_x R_\theta T_v$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -b & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & b & 1 \end{pmatrix}$$

Now if $\tan\theta = m$, standard trigonometry yields $\sin\theta = m/\sqrt{m^2+1}$ and $\cos\theta = 1/\sqrt{m^2+1}$. Substituting these values for $\sin\theta$ and $\cos\theta$ after matrix multiplication, we have

$$M_L = \begin{pmatrix} \dfrac{1-m^2}{m^2+1} & \dfrac{2m}{m^2+1} & 0 \\ \dfrac{2m}{m^2+1} & \dfrac{m^2-1}{m^2+1} & 0 \\ \dfrac{-2bm}{m^2+1} & \dfrac{2b}{m^2+1} & 1 \end{pmatrix}$$

4.11 Reflect the diamond-shaped polygon whose vertices are $A(-1, 0)$, $B(0, -2)$, $C(1, 0)$, and $D(0, 2)$ about (a) the horizontal line $y = 2$, (b) the vertical line $x = 2$, and (c) the line $y = x + 2$.

We represent the vertices of the polygon by the homogeneous coordinate matrix

$$V = \begin{pmatrix} -1 & 0 & 1 \\ 0 & -2 & 1 \\ 1 & 0 & 1 \\ 0 & 2 & 1 \end{pmatrix}$$

From Solved Problem 4.9, the reflection matrix can be written as

$$M_L = T_{-v} R_{-\theta} M_x R_\theta T_V$$

(a) The line $y = 2$ has y intercept $(0, 2)$ and makes an angle of $0°$ with the x axis. So with $\theta = 0$ and $\mathbf{v} = 2\mathbf{J}$, the transformation matrix is

$$M_L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 4 & 1 \end{pmatrix}$$

This same matrix could have been obtained directly by using the results of Problem 4.10 with slope $m = 0$ and y intercept $b = 2$. To reflect the polygon, we set

$$V \cdot M_L = \begin{pmatrix} -1 & 0 & 1 \\ 0 & -2 & 1 \\ 1 & 0 & 1 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 4 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 4 & 1 \\ 0 & 6 & 1 \\ 1 & 4 & 1 \\ 0 & 2 & 1 \end{pmatrix}$$

Converting from homogeneous coordinates, $A' = (-1, 4)$, $B' = (0, 6)$, $C' = (1, 4)$, and $D' = (0, 2)$.

(b) The vertical line $x = 2$ has no y intercept and an infinite slope. We can use M_y, reflection about the y axis, to write the desired reflection by (1) translating the given line two units over to the y axis, (2) reflect about the y axis, and (3) translate back two units. So with $\mathbf{v} = 2\mathbf{I}$,

Finally

$$M_L = T_{-V} M_y T_v$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix} \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 4 & 0 & 1 \end{pmatrix}$$

Finally

$$V \cdot M_L = \begin{pmatrix} -1 & 0 & 1 \\ 0 & -2 & 1 \\ 1 & 0 & 1 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 4 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 5 & 0 & 1 \\ 4 & -2 & 1 \\ 3 & 0 & 1 \\ 4 & 2 & 1 \end{pmatrix}$$

or $A' = (5, 0)$, $B' = (4, -2)$, $C' = (3, 0)$, and $D' = (4, 2)$.

(c) The line $y = x + 2$ has slope 1 and a y intercept $(0, 2)$. From Solved Problem 4.10, with $m = 1$ and $b = 2$, we find

$$M_L = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ -2 & 2 & 1 \end{pmatrix}$$

The required coordinates A', B', C', and D' can now be found.

$$V \cdot M_L = \begin{pmatrix} -1 & 0 & 1 \\ 0 & -2 & 1 \\ 1 & 0 & 1 \\ 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ -2 & 2 & 1 \end{pmatrix} = \begin{pmatrix} -2 & 1 & 1 \\ -4 & 2 & 1 \\ -2 & 3 & 1 \\ 0 & 2 & 1 \end{pmatrix}$$

So $A' = (-2, 1)$, $B' = (-4, 2)$, $C = (-2, 3)$, and $D' = (0, 2)$.

4.12 The matrix $\begin{pmatrix} 1 & b \\ a & 1 \end{pmatrix}$ defines a transformation called a *simultaneous shearing* or *shearing* for short.

The special case when $b = 0$ is called *shearing in the x direction*. When $a = 0$, we have *shearing in the y direction*. Illustrate the effect of these shearing transformations on the square $A(0, 0)$, $B(1, 0)$, $C(1, 1)$, and $D(0, 1)$ when $a = 2$ and $b = 3$.

Figure 4.17(a) shows the original square, Fig. 4.17(b) shows shearing in the x direction, Fig. 4.17(c) shows shearing in the y direction, and Fig. 4.17(d) shows shearing in both directions.

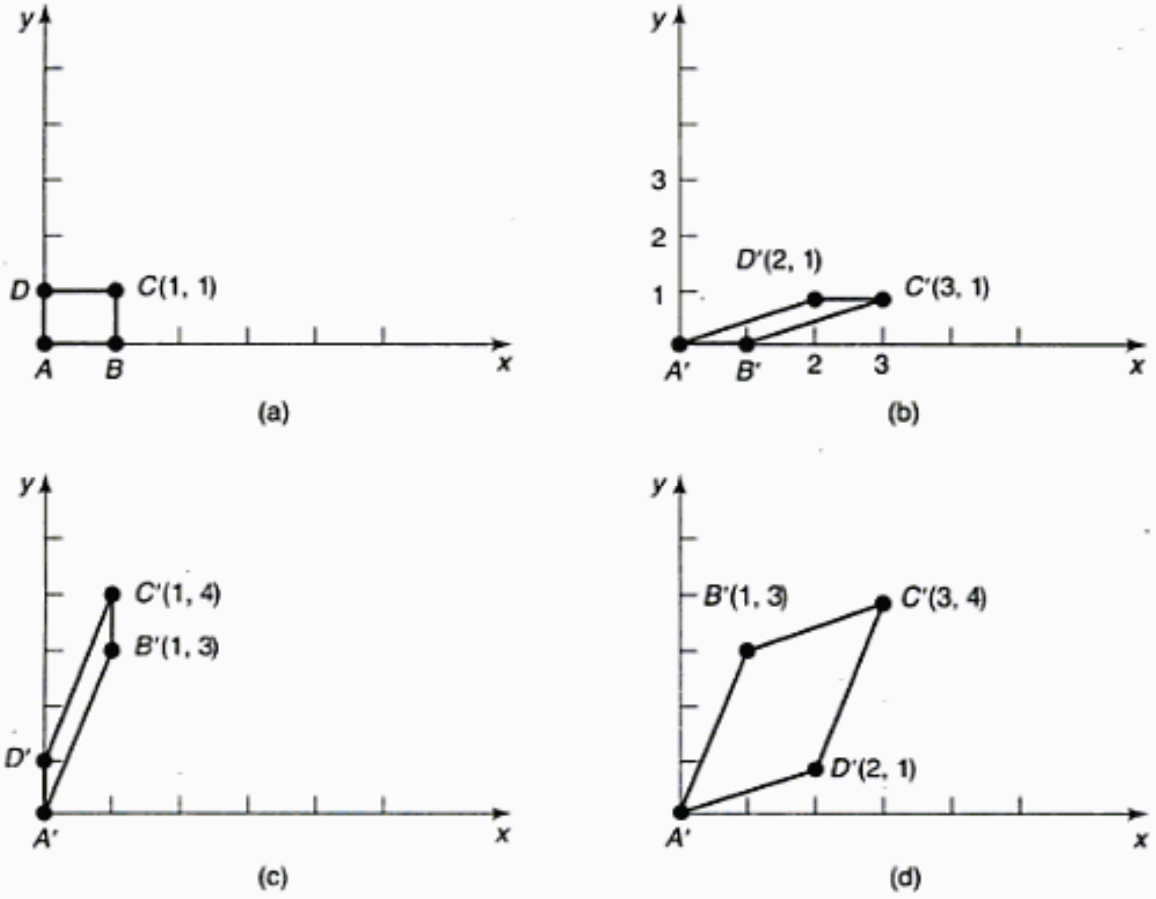


Fig. 4.17

4.13 An observer standing at the origin sees a point $P(1, 1)$. If the point is translated one unit in the direction $\mathbf{v} = \mathbf{I}$, its new coordinate position is $P'(2, 1)$. Suppose instead that the observer stepped back one unit along the x axis. What would be the apparent coordinates of P with respect to the observer?

The problem can be set up as a transformation of coordinate systems. If we translate the origin O in the direction $\mathbf{v} = -\mathbf{I}$ (to a new position at C) the coordinates of P in this system can be found by the translation $\overline{T}_v$

$$P \cdot \bar{T}_v = (1 \quad 1 \quad 1)\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} = (2 \quad 1 \quad 1)$$

So the new coordinates are $(2, 1)'$. This has the following interpretation: a displacement of one unit in a given direction can be achieved by either moving the object forward or stepping back from it.

4.14 An object is defined with respect to a coordinate system whose units are measured in feet. If an observer's coordinate system uses inches as the basic unit, what is the coordinate transformation used to describe object coordinates in the observer's coordinate system?

Since there are 12 inches to a foot, the required transformation can be described by a coordinate scaling transformation with $s = \frac{1}{12}$ or

$$\bar{S}_{1/12} = \begin{pmatrix} \frac{1}{12} & 0 \\ 0 & \frac{1}{12} \end{pmatrix} = \begin{pmatrix} 12 & 0 \\ 0 & 12 \end{pmatrix}$$

and so

$$(x \quad y) \cdot \bar{S}_{1/12} = (x \quad y)\begin{pmatrix} 12 & 0 \\ 0 & 12 \end{pmatrix} = (12x \quad 12y)$$

4.15 Find the equation of the circle $(x')^2 + (y')^2 = 1$ in terms of xy coordinates, assuming that the $x'y'$ coordinate system results from a scaling of a units in the x direction and b units in the y direction.

From the equations for a coordinate scaling transformation, we find

$$x' = \frac{1}{a}x \qquad y' = \frac{1}{b}y$$

Substituting, we have

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1$$

Notice that as a result of scaling, the equation of the circle is transformed to the equation of an ellipse in the xy coordinate system.

4.16 Find the equation of the line $y' = mx' + b$ in xy coordinates if the $x'y'$ coordinate system results from a 90° rotation of the xy coordinate system.

The rotation coordinate transformation equations can be written as

$$x' = x \cos 90° + y \sin 90° = y \qquad y' = -x \sin 90° + y \cos 90° = -x$$

Substituting, we find $-x = my + b$. Solving for y, we have $y = (-1/m)x - b/m$.

4.17 Find the instance transformation which places a half-size copy of the square $A(0, 0)$, $B(1, 0)$, $C(1, 1)$, $D(0, 1)$ [Fig. 4.18(a)] into a master picture coordinate system so that the center of the square is at $(-1, -1)$ [Fig. 4.18(b)].

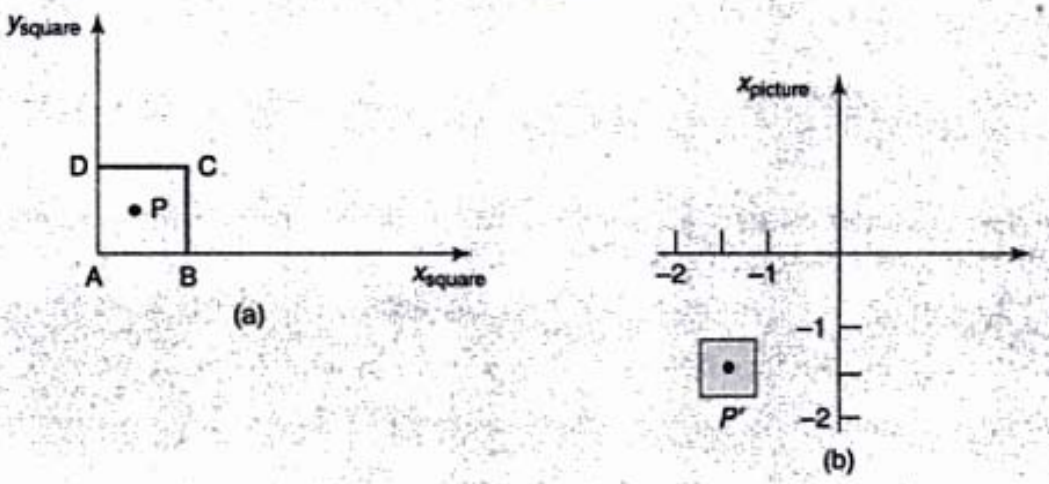


Fig. 4.18

The center of the square $ABCD$ is at $P\left(\frac{1}{2}, \frac{1}{2}\right)$. We shall first apply a scaling transformation while keeping P fixed (see Problem 4.7). Then we shall apply a translation that moves the center P to $P'(-1, -1)$. Taking $t_x = (-1) - \frac{1}{2} = -(3/2)$ and similarly $t_y = -(3/2)$ (so $\mathbf{v} = -3/2\ \mathbf{I} - 3/2\ \mathbf{J}$), we obtain

$$N_{\text{picture,object}} = S_{1/2,1/2,\,P} \cdot T_v = \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ \frac{1}{4} & \frac{1}{4} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{-3}{2} & \frac{-3}{2} & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ \frac{-5}{4} & \frac{-5}{4} & 1 \end{pmatrix}$$

4.18 Write the composite transformation that creates the design in Fig. 4.20 from the symbols in Fig. 4.19.

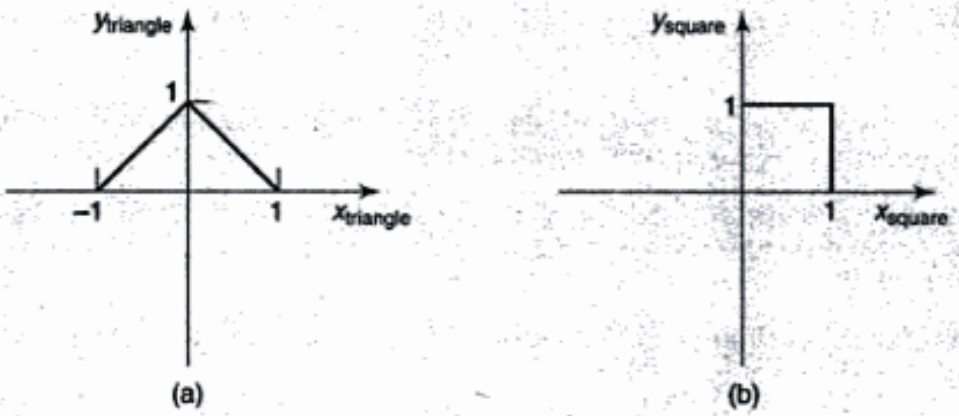


Fig. 4.19

First we create an instance of the triangle [Fig. 4.19(a)] in the square [Fig. 4.19(b)]. Since the base of the triangle must be halved while keeping the height fixed at one unit, the appropriate instance transformation is $N_{\text{square, triangle}} = T_{1/2,1} \cdot S_{1/2,1}$.

The instance transformation needed to place the square at the desired position in the picture coordinate system (Fig. 4.19) is a translation in the direction $\mathbf{v} = \mathbf{I} + \mathbf{J}$:

$$N_{\text{picture, square}} = T_{\mathbf{v}}$$

Then the composite transformation for placing the triangle into the picture is

$$C_{\text{picture, triangle}} = N_{\text{square, triangle}} \cdot N_{\text{picture, square}}$$

and the composite transformation to place the square into the picture is

$$C_{\text{picture, square}} = N_{\text{picture, square}}$$

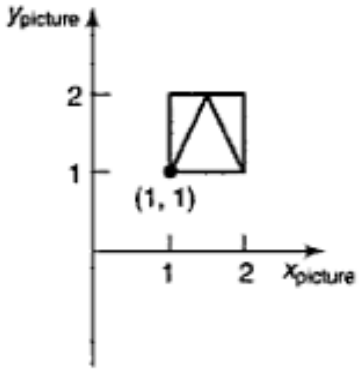


Fig. 4.20

4.19 Write an algorithm for even odd method for polygon inside test.

Suppose the polygon vertices are $P_1(x_1, y_1)$, $P_2(x_1, y_1)$, ... $P_n(x_n, y_n)$ and the point in question is $P(x, y)$

1. Choose a point Q (x_0, y_0) outside the polygon. This can be done by choosing $x_0 < \min(x_1, x_2, \ldots, x_n)$ or $x_0 > \max(x_1, x_2, \ldots x_n)$ or $y_0 < \min(y_1, y_2, \ldots y_n)$ or $y_0 > \max(y_1, y_2, \ldots y_n)$.
2. Draw a line joining P and Q.
3. Count the number of intersections as follows
   - **3.1** If the intersection point is a vertex point then check whether the points P_{i+1} and P_{i-1} are on the same side of the line PQ, if so count the number of intersections as two.
     else
   - **3.2** Count the number of intersections as one
4. Find the total number of intersections in step 3 above.
5. If the total number is even then the point P is outside the polygon otherwise it is inside.

4.20 Write an algorithm for drawing an ellipse using 2D transformations and any circle generating algorithm from chapter 3.

Step 1: Input a, b, x_0, y_0.

Step 2: If $a > b$ then

- 2.1 Draw the circle $(x - x_0)^2 + (y - y_0)^2 = a^2$
- 2.2 Scale the circle with scaling factor b/a along y-axis

  else

Step 3: Draw the circle with $(x - x_0)^2 + (y - y_0)^2 = b^2$

- 3.1 Scale the circle with scaling factor b/a along x-axis

SUPPLEMENTARY PROBLEMS

4.1 Derive the X-shear transformation from the rotation and scaling transformations.

4.2 Derive the rotation transformation from the shear and other transformations.

4.3 What is the relationship between the rotations R_θ, $R_{-\theta}$, and R_θ^1?

4.4 Describe the transformations used in magnification and reduction with respect to the origin. Find the new coordinates of the triangle $A(0, 0)$, $B(1, 1)$, $C(5, 2)$ after it has been (a) magnified to twice its size and (b) reduced to half its size.

4.5 Show that reflection about the line $y = x$ is attained by reversing coordinates. That is,

$$M_L(x, y) = (y, x)$$

4.6 Show that the order in which transformations are performed is important by the transformation of triangle $A(1, 0)$, $B(0, 1)$, $C(1, 1)$, by (a) rotating 45° about the origin and then translating in the direction of vector I, and (b) translating and then rotating.

4.7 An object point $P(x, y)$ is translated in the direction $\mathbf{v} = a\mathbf{I} + b\mathbf{J}$ and simultaneously an observer moves in the direction $\mathbf{v}$. Show that there is no apparent motion (from the point of view of the observer) of the object point.

4.8 Assuming that we have a mathematical equation defining a curve in $x'y'$ coordinates, and the $x'y'$ coordinate system is the result of a coordinate transformation from the xy coordinate system, write the equation in terms of xy coordinates.

4.9 Show that $T_{v_1} \cdot T_{v_2} = T_{v_2} \cdot T_{v_1} = T_{v_1+v_2}$

4.10 Show that $S_{a,b} \cdot S_{c,d} = S_{c,d} \cdot S_{a,b} = S_{ac,\,bd}$.

4.11 Show that $R_\alpha \cdot R_\beta = R_\beta \cdot R_\alpha = R_{\alpha+\beta}$.

4.12 Find the condition under which we have $S_{s_x,s_y} \cdot R_\theta = R_\theta \cdot S_{s_x,s_y}$

4.13 Is simultaneous shearing the same as shearing in one direction followed by a shearing in another direction? Why?

4.14 Find the condition under which we can switch the order of a rotation and a simultaneous shearing and still get the same result.

4.15 Express a simultaneous shearing in terms of rotation and scaling transformations.

4.16 Express R_θ in terms of shearing and scaling transformations.

4.17 Express R_θ in terms of shearing transformation.

4.18 Prove that the 2D composite transformation matrix is always in the following form:

$$\begin{pmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{pmatrix}$$

4.19 Consider a line from P_1 to P_2 and an arbitrary point P on the line. Prove that for any given composite transformation, the transformed P is on the line between the transformations of P_1 and P_2.

4.20 Check whether the point $(-1, 2)$ is inside the polygon with vertices $(2, 3)$, $(4, 5)$, $(1, 9)$, $(6, 3)$ and $(0, 0)$ or not.

4.21 Write an algorithm for winding number method of polygon inside test.

ANSWERS TO SUPPLEMENTARY PROBLEMS

4.1 Put $b = 0$ in the solution of the 4.15 of the supplementary problems.

4.2 Put $a = \tan\theta$, $b = -\tan\theta$, $S_x = \sec\theta$, $S_y = \cos\theta$ the rotation matrix

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \begin{pmatrix} 1 & 0 \\ b & 1 \end{pmatrix}$$

4.3 $R_\theta = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$ and $R_{-\theta} = \begin{pmatrix} \cos(-\theta) & \sin(-\theta) \\ -\sin(-\theta) & \cos(-\theta) \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$

Also

$$R_{-\theta} \cdot R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

$$= \begin{pmatrix} (\cos^2\theta + \sin^2\theta) & (\cos\theta\sin\theta - \sin\theta\cos\theta) \\ (\sin\theta\cos\theta - \cos\theta\sin\theta) & (\sin^2\theta + \cos^2\theta) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Therefore, R_θ and $R_{-\theta}$ are inverses, so $R_{-\theta} = R_\theta^{-1}$. In other words, the inverse of a rotation by θ degrees, is a rotation in the opposite direction.

4.4 Magnification and reduction can be achieved by a uniform scaling of s units in both the X and Y directions. If $s > 1$, the scaling produces magnification. If $s < 1$, the result is a reduction. The transformation can be written as

$$(x, y) \mapsto (sx, sy)$$

In matrix form, this becomes

$$(x, y)\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} = (sx, sy)$$

(a) Choosing $s = 2$ and applying the transformation to the coordinates of the points A, B, C yields the new coordinates $A'(0, 0)$, $B'(2, 2)$, $C'(10, 4)$.

(b) Here, $s = \frac{1}{2}$ and the new coordinates are

$$A''(0, 0),\ B''\left(\frac{1}{2}, \frac{1}{2}\right),\ C''\left(\frac{5}{2}, 1\right).$$

4.5 The line $y = x$ has slope 1 and y intercept $(0, 0)$. If point P has coordinates (x, y), then following Problem 4.10 we have

$$P \cdot M_L = (x, y)\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = (y, x)$$

or $\quad M_L(x, y) = (y, x)$

4.6 The rotation matrix is

$$R_{45^\circ} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The translation matrix is

$$T_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

The matrix of vertices $\begin{bmatrix} A \\ B \\ C \end{bmatrix}$ is

Since (x, y) satisfies

$$\frac{y_2 - y_1}{x_2 - x_1} = \frac{y_2 - y}{x_2 - x}$$

we have established the equality that show P' being on the line between P_1' and P_2'.

4.20 We casn see that the x-cordinate of (–1, 2) is less than the minimum of the x-cordinates of (2, 3), (4, 5), (1, 9), (6, 3) and (0, 0). Hence the point (–1, 2) is outside the given polygon.

4.21 Suppose the polygon vertices are $P_1(x_1, y_1)$, $P_2(x_2, y_2)$, ..., $P_n(x_n, y_n)$ and the point in question is $P(x, y)$.

1. Choose a point $Q(x_0, y_0)$ outside the polygon. This can be done by choosing $x_0 < \min(x_1, x_2 \ldots, x_n)$ or $y_0 > \min(y_1, y_2, \ldots, y_n)$ or $y_0 > \max(y_1, y_2 \ldots y_n)$.
2. Draw a line joining P and Q
3. If $P_{i-1}\,P_i$ is the edge intersecting with the line segment PQ then the winding number of the edge $P_{i-1}P_i$ is determined as follows.
   If the y-coordinate of P_{i-1} is less than that of P_i then the winding number of the edge $P_{i+1}P_i$ is –1 else it is +1.
4. Find the net winding as the sum of the winding numbers of all edges of step 3 above.
5. If the net winding is zero then the point P is outside the polygon otherwise it is inside.

Chapter Five

Two-Dimensional Viewing and Clipping

Much like what we see in real life through a small window on the wall or the viewfinder of a camera, a computer-generated image often depicts a partial view of a large scene. Objects are placed into the scene by modelling transformations to a master coordinate system, commonly referred to as the *world coordinate system* (WCS). A rectangular area with its edges parallel to the axes of the WCS is used to select the portion of the scene for which an image is to be generated (see Fig. 5.1). This rectangular area is called a window. Sometimes an additional coordinate system called the *viewing coordinate system* is introduced to simulate the effect of moving and/or tilting the camera.

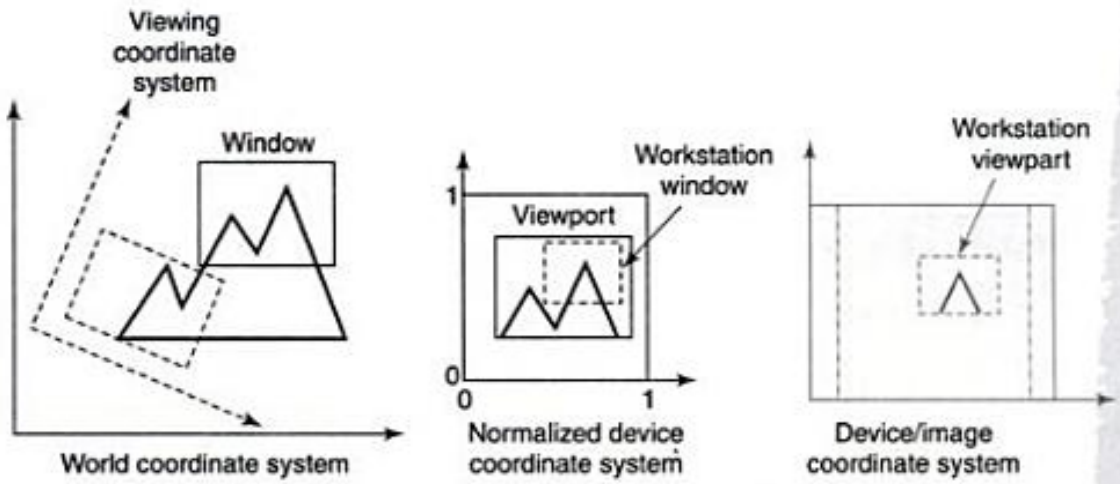


Fig. 5.1 *Viewing Transformation*

On the other hand, an image representing a view often becomes part of a larger image, like a photo on an album page, which models a computer monitor's display area. Since the monitor sizes differ from one system to another, we want to introduce a device-independent tool to describe the display area. This tool is called the *normalized device coordinate system* (NDCS) in which a unit (1×1) square whose lower left corner is at the origin of the coordinate system defines the display area of a virtual display device. A rectangular area with its edges parallel to the axes of the NDCS is used to specify a sub-region of the display area that embodies the image. This rectangular area of NDCS is called a viewpart.

The process that converts object coordinates in WCS to normalized device coordinates is called *window-to-viewport mapping* or *normalization transformation*, which is the subject of Section 5.1. The process that maps normalized device coordinates to discrete device/image coordinates is called *workstation transformation*, which is essentially a second window-to-viewport mapping, with a workstation window in the normalized device coordinate system and a workstation viewport in the device coordinate system. Collectively, these two coordinate-mapping operations are referred to as *viewing transformation*.

Workstation transformation is dependent on the resolution of the display device/frame buffer. When the whole display area of the virtual device is mapped to a physical device that does not have a 1/1 aspect ratio, it may be mapped to a square sub-region (see Fig. 5.1) so as to avoid introducing unwanted geometric distortion.

Along with the convenience and flexibility of using a window to specify a localized view comes the need for *clipping*, since objects in the scene may be completely inside the window, completely outside the window, or partially visible through the window (e.g. the mountain-like polygon in Fig. 5.1). The clipping operation eliminates objects or portions of objects that are not visible through the window to ensure the proper construction of the corresponding image.

Note that clipping may occur in the world coordinate or viewing coordinate space, where the window is used to clip the objects; it may also occur in the normalized device coordinate space, where the viewport/workstation window is used to clip. In either case we refer to the window or the viewport/workstation window as the *clipping window*. We discuss point clipping, line clipping, and polygon clipping in Sections 5.2, 5.3 and 5.4 respectively.

5.1 WINDOW-TO-VIEWPORT MAPPING

A window is specified by four world coordinates: wx_{min}, wx_{max}, wy_{min}, and wy_{max} (see Fig. 5.2). Similarly, a viewport is described by four normalized device coordinates: vx_{min}, vx_{max}, vy_{min}, and vy_{max}. The objective of window-to-viewport mapping is to convert the world coordinates (wx, wy) of an arbitrary point to its corresponding normalized device coordinates (vx, vy). In order to maintain the same relative placement of the point in the viewport as in the window, we require:

$$\frac{wx - wx_{min}}{wx_{max} - wx_{min}} = \frac{vx - vx_{min}}{vx_{max} - vx_{min}} \quad \text{and} \quad \frac{wy - wy_{min}}{wy_{max} - wy_{min}} = \frac{vy - vy_{min}}{vy_{max} - vy_{min}}$$

Thus

$$\begin{cases} vx = \dfrac{vx_{max} - vx_{min}}{wx_{max} - wx_{min}}(wx - wx_{min}) + vx_{min} \\ vy = \dfrac{vy_{max} - vy_{min}}{wy_{max} - wy_{min}}(wy - wy_{min}) + vy_{min} \end{cases}$$

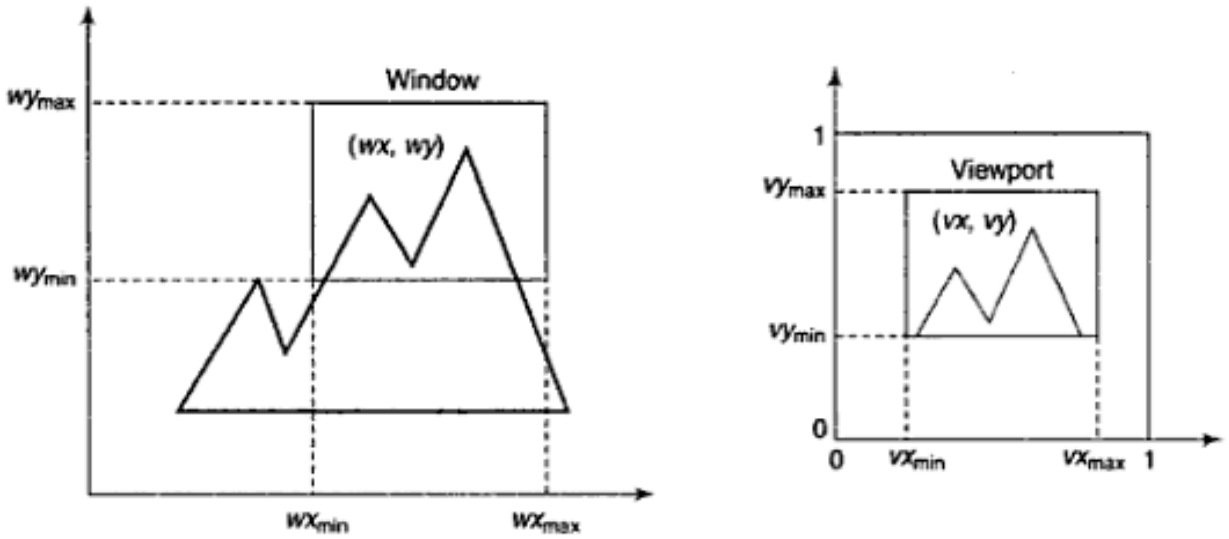


Fig. 5.2 *Window-to-viewport Mapping*

Since the eight coordinate values that define the window and the viewport are just constants, we can express these two formulas for computing (vx, vy) from (wx, wy) in terms of a translate–scale–translate transformation N

$$(vx, vy\ 1) = (wx, wy\ 1) \cdot N$$

where

$$N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -wx_{\min} & -wy_{\min} & 1 \end{pmatrix} \cdot \begin{bmatrix} \dfrac{vx_{\max} - vx_{\min}}{wx_{\max} - wx_{\min}} & 0 & 0 \\ 0 & \dfrac{vx_{\max} - vy_{\min}}{wx_{\max} - wy_{\min}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ wx_{\min} & wy_{\min} & 1 \end{pmatrix}$$

Note that geometric distortions occur (e.g. squares in the window become rectangles in the viewport) whenever the two scaling constants differ.

5.2 POINT CLIPPING

Point clipping is essentially the evaluation of the following inequalities:

$$x_{\min} \le x \le x_{\max} \quad \text{and} \quad y_{\min} \le y \le y_{\max}$$

where $x_{\min}$, $x_{\max}$, $y_{\min}$ and $y_{\max}$ define the clipping window. A point (x, y) is considered inside the window when the inequalities all evaluate to true.

5.3 LINE CLIPPING

Lines that do not intersect the clipping window are either completely inside the window or completely outside the window. On the other hand, a line that intersects the clipping window is divided by the intersection point(s) into segments that are either inside or outside the window. The following

algorithms provide efficient ways to decide the spatial relationship between an arbitrary line and the clipping window and to find intersection point(s).

The Cohen-Sutherland Algorithm

In this algorithm we divide the line clipping process into two phases: (1) identify those lines which intersect the clipping window and so need to be clipped and (2) perform the clipping.

All lines fall into one of the following *clipping categories*:

1. *Visible*—both endpoints of the line lie within the window.
2. *Not visible*—the line definitely lies outside the window. This will occur if the line from (x_1, y_1) to (x_2, y_2) satisfies any one of the following four inequalities:

$$x_1, x_2 > x_{max} \qquad y_1, y_2 > y_{max}$$
$$x_1, x_2 < x_{min} \qquad y_1, y_2 < y_{min}$$

3. *Clipping candidate*—the line is in neither category 1 nor 2.

In Fig. 5.3, line AB is in category 1 (visible); lines CD and EF are in category 2 (not visible); and lines GH, IJ, and KL are in category 3 (clipping candidate).

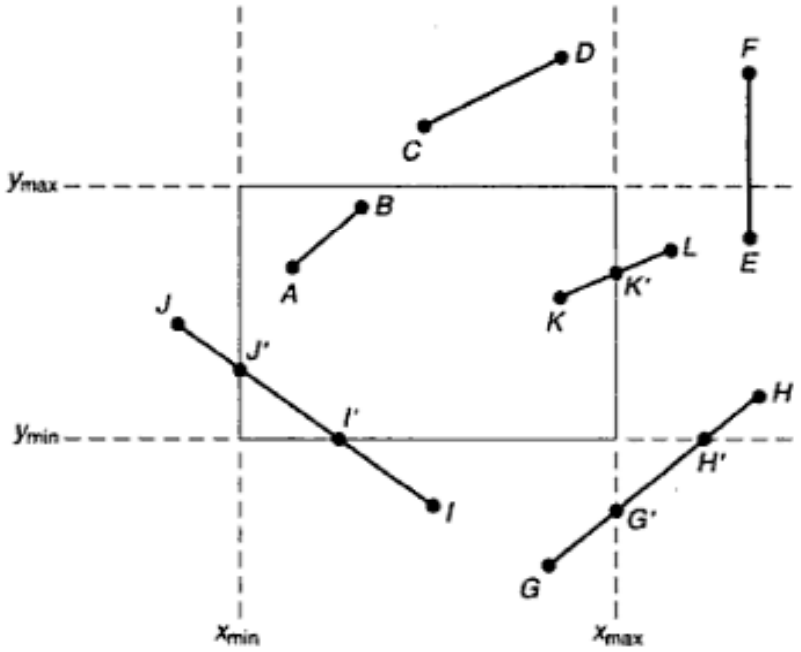


Fig. 5.3

The algorithm employs an efficient procedure for finding the category of a line. It proceeds in two steps:

1. Assign a 4-bit region code to each endpoint of the line. The code is determined according to which of the following nine regions of the plane the endpoint lies in

	1001	1000	1010
y_{max}	0001	0000	0010
y_{min}	0101	0100	0110
	x_{min}	x_{max}	

Starting from the leftmost bit, each bit of the code is set to true (1) or false (0) according to the scheme

Bit 1 ≡ endpoint is above the window = sign $(y - y_{max})$
Bit 2 ≡ endpoint is below the window = sign $(y_{min} - y)$
Bit 3 ≡ endpoint is to the right of the window = sign $(x - x_{max})$
Bit 4 ≡ endpoint is to the left of the window = sign $(x_{min} - x)$

We use the convention that sign(a) = 1 if a is positive, 0 otherwise. Of course, a point with code 0000 is inside the window.

2. The line is visible if both region codes are 0000, not visible if the bitwise logical AND of the codes is not 0000, and a candidate for clipping if the bitwise logical AND of the region codes is 0000 (see Problem 5.8).

For a line in category 3 we proceed to find the intersection point of the line with one of the boundaries of the clipping window, or to be exact, with the infinite extension of one of the boundaries (see Fig. 5.4). We choose an endpoint of the line, say (x_1, y_1), that is outside the window, i.e. whose region code is not 0000. We then select an extended boundary line by observing that those boundary lines that are candidates for intersection are the ones for which the chosen endpoint must be "pushed across" so as to change a "1" in its code to a "0" (see Fig. 5.4). This means:

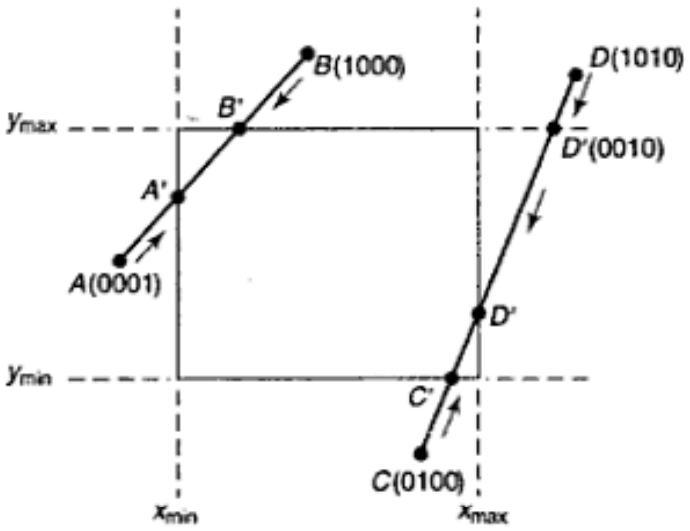


Fig. 5.4

If bit 1 is 1, intersect with line $y = y_{max}$.
If bit 2 is 1, intersect with line $y = y_{min}$.
If bit 3 is 1, intersect with line $x = x_{max}$.
If bit 4 is 1, intersect with line $x = x_{min}$.

Consider line CD in Fig. 5.4. If endpoint C is chosen, then the bottom boundary line $y = y_{min}$ is selected for computing intersection. On the other hand, if endpoint D is chosen, then either the top boundary line $y = y_{max}$ or the right boundary line $x = x_{max}$ is used. The coordinates of the intersection point are

$$\begin{cases} x_i = x_{min} \text{ or } x_{max} \\ y_i = y_1 + m\,(x_i - x_1) \end{cases} \quad \text{if the boundary line is vertical}$$

or

$$\begin{cases} x_i = x_1 + (y_i - y_1)/m \\ y_i = y_{min} \text{ or } y_{max} \end{cases} \quad \text{if the boundary line is horizontal}$$

where $m = (y_2 - y_1)/(x_2 - x_1)$ is the slope of the line.

Now we replace endpoint (x_1, y_1) with the intersection point (x_i, y_i), effectively eliminating the portion of the original line that is on the outside of the selected window boundary. The new endpoint is then assigned an updated region code and the clipped line re-categorized and handled

in the same way. This iterative process terminates when we finally reach a clipped line that belongs to either category 1 (visible) or category 2 (not visible).

Midpoint Subdivision

An alternative way to process a line in category 3 is based on binary search. The line is divided at its midpoint into two shorter line segments. The clipping categories of the two new line segments are then determined by their region codes. Each segment in category 3 is divided again into shorter segments and categorized. This bisection and categorization process continues until each line segment that spans across a window boundary (hence encompasses an intersection point) reaches a threshold for line size and all other segments are either in category 1 (visible) or in category 2 (invisible). The midpoint coordinates (x_m, y_m) of a line joining (x_1, y_1) and (x_2, y_2) are given by

$$x_m = \frac{x_1 + x_2}{2} \quad y_m = \frac{y_1 + y_2}{2}$$

The example in Fig. 5.5 illustrates how midpoint subdivision is used to zoom in onto the two intersection points I_1 and I_2 with 10 bisections. The process continues until we reach two line segments that are, say, pixel-sized, i.e. mapped to one single pixel each in the image space. If the maximum number of pixels in a line is M, this method will yield a pixel-sized line segment in N subdivisions, where $2^N = M$ or $N = \log_2 M$. For instance, when $M = 1024$ we need at most $N = \log_2 1024 = 10$ subdivisions.

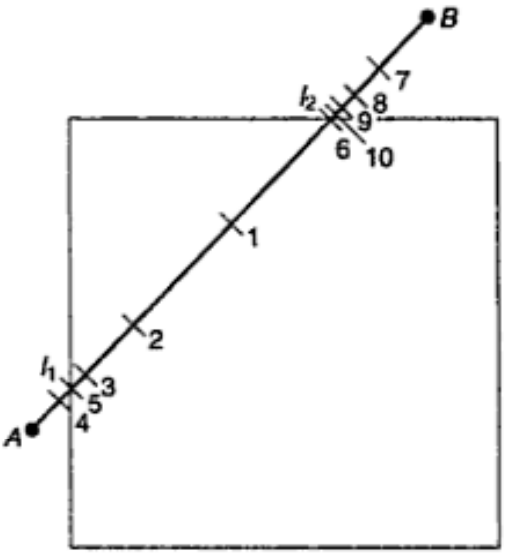


Fig. 5.5

The Liang–Barsky Algorithm

The following parametric equations represent a line from (x_1, y_1) to (x_2, y_2) along with its infinite extension:

$$\begin{cases} x = x_1 + \Delta x \cdot u \\ y = y_1 + \Delta y \cdot u \end{cases}$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. The line itself corresponds to $0 \le u \le 1$ (see Fig. 5.6). Notice that when we traverse along the extended line with u increasing from $-\infty$ to $+\infty$, we first move from the outside to the inside of the clipping window's two boundary lines (bottom and left), and then move from the inside to the outside of the other two boundary lines (top and right). If we use u_1 and u_2, where $u_1 \le u_2$, to represent the beginning and end of the visible portion of the line, we have $u_1 = \text{maximum}(0, u_l, u_b)$ and $u_2 = \text{minimum}(1, u_t, u_r)$, where u_l u_b, u_t, and u_r correspond to the intersection point of the extended line with the window's left, bottom, top, and right boundary, respectively.

Now consider the tools we need to turn this basic idea into an efficient algorithm. For point (x, y) inside the clipping window, we have

$$x_{\min} \le x_1 + \Delta x \cdot u \le x_{\max}$$
$$y_{\min} \le y_1 + \Delta y \cdot u \le y_{\max}$$

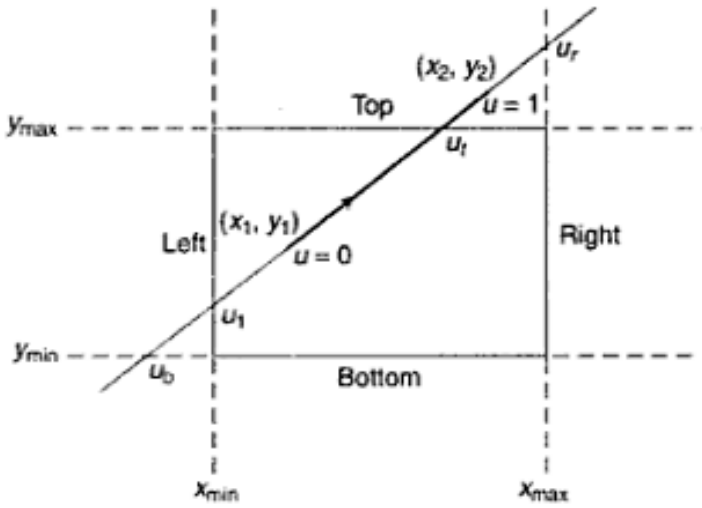


Fig. 5.6

Rewrite the four inequalities as

$$p_k\, u \le q_k,\ k = 1, 2, 3, 4$$

where

$p_1 = -\Delta x$	$q_1 = x_1 - x_{min}$	(left)
$p_2 = \Delta x$	$q_2 = x_{max} - x_1$	(right)
$p_3 = -\Delta y$	$q_3 = y_1 - y_{min}$	(bottom)
$p_4 = \Delta y$	$q_4 = y_{max} - y_1$	(top)

Observe the following facts:

- if $p_k = 0$, the line is parallel to the corresponding boundary and
  - if $q_k < 0$, the line is completely outside the boundary and can be eliminated
  - if $q_k \ge 0$, the line is inside the boundary and needs further consideration,
- if $p_k < 0$, the extended line proceeds from the outside to the inside of the corresponding boundary line,
- if $p_k > 0$, the extended line proceeds from the inside to the outside of the corresponding boundary line,
- when $p_k \ne 0$, the value of u that corresponds to the intersection point is q_k/p_k.

The Liang-Barsky algorithm for finding the visible portion of the line, if any, can be stated as a four-step process:

1. If $p_k = 0$ and $q_k < 0$ for any k, eliminate the line and stop. Otherwise proceed to the next step.
2. For all k such that $p_k < 0$, calculate $r_k = q_k/p_k$. Let u_1 be the maximum of the set containing 0 and the calculated r values.
3. For all k such that $p_k > 0$, calculate $r_k = q_k/p_k$. Let u_2 be the minimum of the set containing 1 and the calculated r values.
4. If $u_1 > u_2$, eliminate the line since it is completely outside the clipping window. Otherwise, use u_1 and u_2 to calculate the endpoints of the clipped line.

5.4 POLYGON CLIPPING

In this section we consider the case of using a polygonal clipping window to clip a polygon.

Convex Polygonal Clipping Windows

A polygon is called *convex* if the line joining any two interior points of the polygon lies completely inside the polygon (see Fig. 5.7). A non-convex polygon is said to be *concave*.

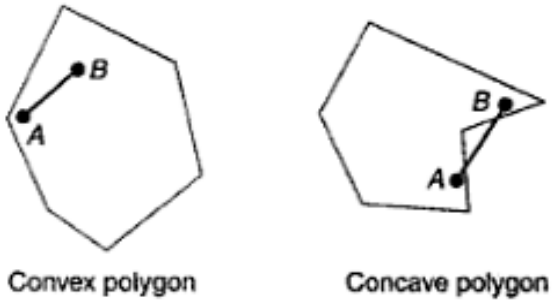


Fig. 5.7

By convention, a polygon with vertices $P_1,\ldots, P_N$ (and edges P_{i-1}, P_i and P_NP_1) is said to be *positively oriented* if a tour of the vertices in the given order produces a counterclockwise circuit.

Equivalently, the left hand of a person standing along any directed edge $\overline{P_{i-1}P_i}$ or $\overline{P_NP_1}$ would be pointing inside the polygon [see orientations in Figs. 5.8(a) and 5.8(b)].

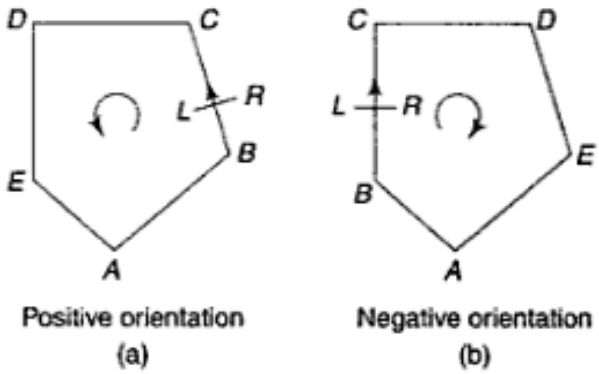


Fig. 5.8

Let $A(x_1, y_1)$ and $B(x_2, y_2)$ be the endpoints of a directed line segment. A point $P(x, y)$ will be to the *left* of the line segment if the expression $C = (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)$ is positive (see Problem 5.13). We say that the point is to the *right* of the line segment if this quantity is negative. If a point P is to the right of *any one* edge of a positively oriented, convex polygon, it is outside the polygon. If it is to the left of *every* edge of the polygon, it is inside the polygon.

This observation forms the basis for clipping any polygon, convex or concave, against a convex polygonal clipping window.

The Sutherland-Hodgeman Algorithm

Let $P_1,\ldots, P_N$ be the vertex list of the polygon to be clipped. Let edge E, determined by endpoints A and B, be any edge of the positively oriented, convex clipping polygon. We clip each edge of the polygon in turn against the edge E of the clipping polygon, forming a new polygon whose vertices are determined as follows.

Consider the edge $\overline{P_{i-1}P_i}$:

1. If both P_{i-1} and P_i, are to the left of the edge, vertex P_i is placed on the *vertex output list* of the clipped polygon [Fig. 5.9(a)].
2. If both P_{i-1} and P_i, are to the right of the edge, nothing is placed on the vertex output list [Fig. 5.9(b)].
3. If P_{i-1} is to the left and P_i, is to the right of the edge E, the intersection point I of line segment $\overline{P_{i-1}, P_i}$ with the extended edge E is calculated and placed on the vertex output list [Fig. 5.9(c)].

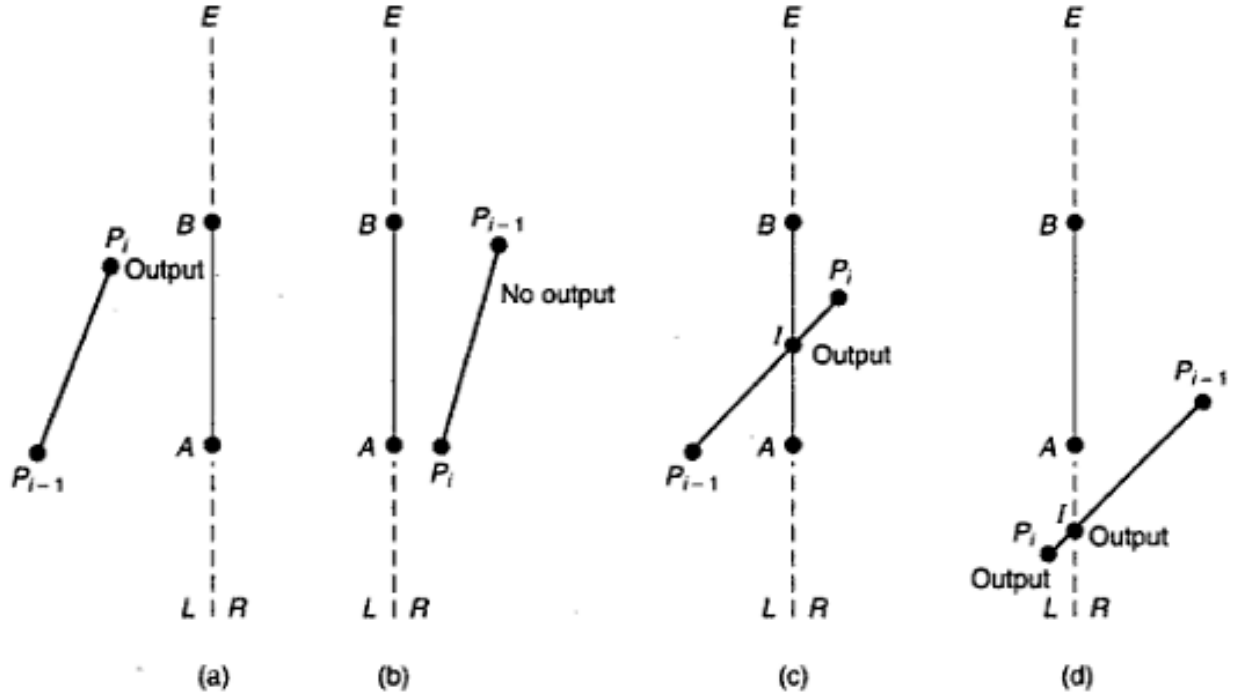


Fig. 5.9

4. If P_{i-1} is to the right and P_i is to the left of edge E, the intersection point I of the line segment $\overline{P_{i-1}, P_i}$ with the extended edge E is calculated. Both I and P_i are placed on the vertex output list [Fig. 5.9(d)].

The algorithm proceeds in stages by passing each clipped polygon to the next edge of the window and clipping. See Problems 5.14 and 5.15.

Special attention is necessary in using the Sutherland–Hodgman algorithm in order to avoid unwanted effects. Consider the example in Fig. 5.10(a). The correct result should consist of two disconnected parts, a square in the lower left corner of the clipping window and a triangle at the top [see Fig. 5.10(b)]. However, the algorithm produces a list of vertices (see Problem 5.16) that forms a figure with the two parts connected by extra edges [see Fig. 5.10(c)]. The fact that these edges are drawn twice in opposite direction can be used to devise a post-processing step to eliminate them.

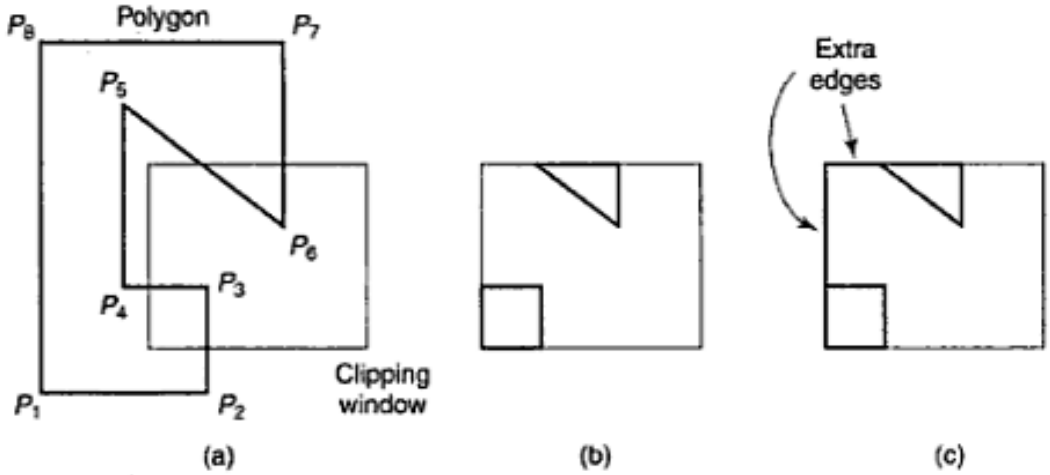


Fig. 5.10

The Weiler–Atherton Algorithm

Let the clipping window be initially called the clip polygon, and the polygon to be clipped the subject polygon [see Fig. 5.11 (a)]. We start with an arbitrary vertex of the subject polygon and trace around its border in the clockwise direction until an intersection with the clip polygon is encountered:

- If the edge enters the clip polygon, record the intersection point and continue to trace the subject polygon.
- If the edge leaves the clip polygon, record the intersection point and make a right turn to follow the clip polygon in the same manner (i.e. treat the clip polygon as subject polygon and the subject polygon as clip polygon and proceed as before).

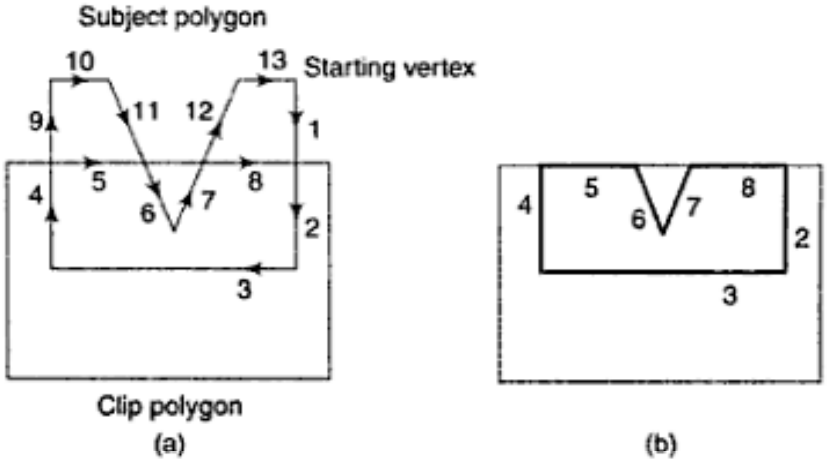


Fig. 5.11

Whenever our path of traversal forms a sub-polygon we output the sub-polygon as part of the overall result. We then continue to trace the rest of the original subject polygon from a recorded intersection point that marks the beginning of a not-yet-traced edge or portion of an edge. The algorithm terminates when the entire border of the original subject polygon has been traced exactly once.

For example, the numbers in Fig. 5.11(a) indicate the order in which the edges and portions of edges are traced. We begin at the starting vertex and continue along the same edge (from 1 to 2) of the subject polygon as it enters the clip polygon. As we move along the edge that is leaving the clip polygon we make a right turn (from 4 to 5) onto the clip polygon, which is now considered the subject polygon. Following the same logic leads to the next right turn (from 5 to 6) onto the current clip polygon, which is really the original subject polygon. With the next step done (from 7 to 8) in the same way we have a sub-polygon for output [see Fig. 5.11(b)]. We then resume our traversal of the original subject polygon from the recorded intersection point where we first changed our course. Going from 9 to 10 to 11 produces no output. After skipping the already-traversed 6 and 7, we continue with 12 and 13 and come to an end. The figure in Fig. 5.11 (b) is the final result.

Applying the Weiler-Atherton algorithm to clip the polygon in Fig. 5.10(a) produces correct result [see Fig. 5.12(a) and (b)].

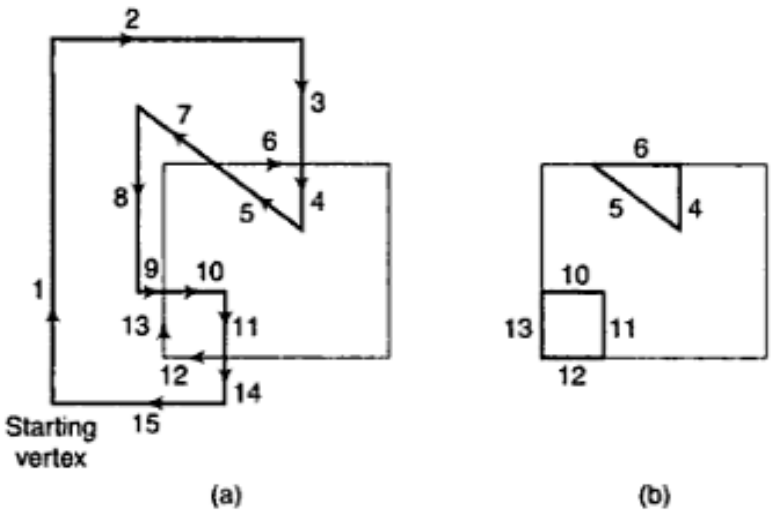


Fig. 5.12

5.5 EXAMPLE: A 2D GRAPHICS PIPELINE

Shared by many graphics systems is the overall system architecture called the graphics pipeline. The operational organization of a 2D graphics pipeline is shown in Fig. 5.13. Although 2D graphics is typically treated as a special case ($z = 0$) of three-dimensional graphics, it demonstrates the common working principle and basic application of these pipelined systems.

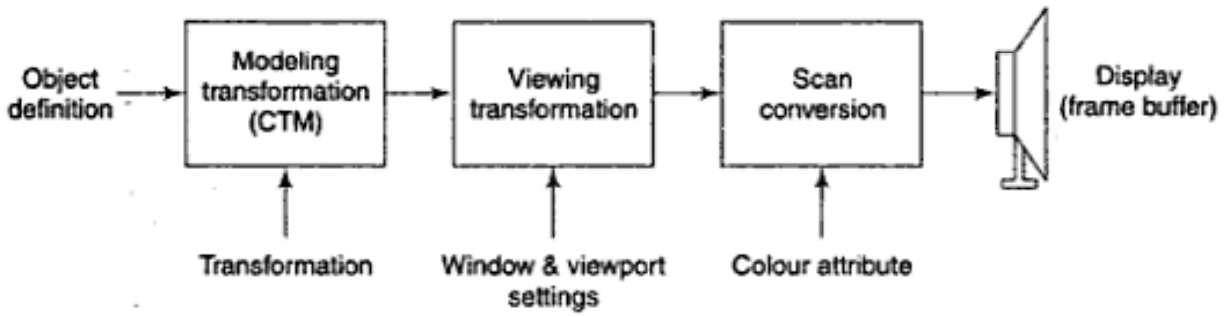


Fig. 5.13 *A 2D Graphics Pipeline*

At the beginning of the pipeline we have object data (e.g. vertex coordinates for lines and polygons that make up individual objects) stored in application-specific data structures. A graphics application uses system subroutines to initialize and to change, among other things, the transformations that are to be applied to the original data, window and viewport settings, and the color attributes of the objects. Whenever a drawing subroutine is called to render a pre-defined object, the graphics system first applies the specified modeling transformation to the original data, then carries out viewing transformation using the current window and viewport settings, and finally performs scan conversion to set the proper pixels in the frame buffer with the specified color attributes.

Suppose that we have an object centered in its own coordinate system [see Fig. 5.14(a)], and we are to construct a sequence of images that animates the object rotating around its center and moving along a circular path in a square display area [see Fig. 5.14(b)]. We generate each image as

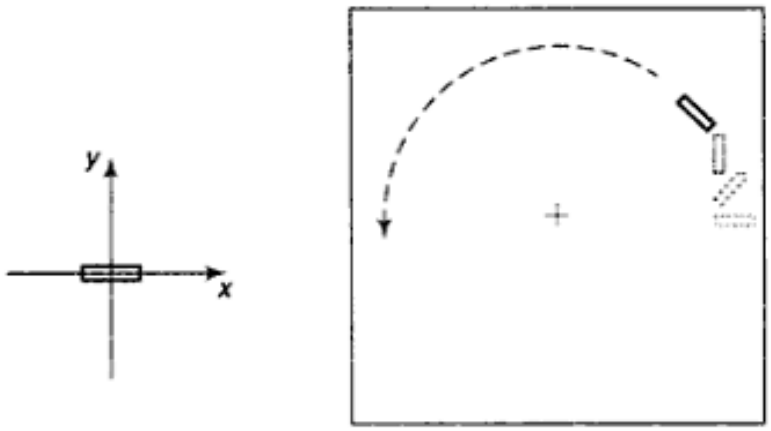


Fig. 5.14

follows: first rotate the object around its center by angle α, then translate the rotated object by offset I to position its center on the positive x axis of the WCS, and rotate it with respect to the origin of the WCS by angle β. We control the amount of the first rotation from one image to the next by $\Delta\alpha$, and that of the second rotation by $\Delta\beta$.

```
window(-winsize/2, winsize/2, -winsize/2, winsize/2);
α = 0;
while (1) {
   setColor(background);
   clear();
   setColor(color);
   pushCTM();
   translate(offset, 0);
   rotate(α);
   drawObject();
   popCTM();
   α = α + Δα;
   rotate (Δβ);
}
```

We first set the size of the window by winsize to be sufficiently large and centered at the origin of the WCS to cover the entire scene. The system's default viewport coincides with the unit display area in the NDCS. The default workstation window is the same as the viewport and the default workstation viewport corresponds to the whole square display area.

The graphics system maintains a stack of composite transformation matrices. The CTM on top of the stack, called the current CTM, is initially an identity matrix and is automatically used in modeling transformation. Each call to translate, scale, and rotate causes the system to generate a corresponding transformation matrix and to reset the current CTM to take into account the generated matrix. The order of transformation is maintained in such a way that the most recently specified transformation is applied first. When pushCTM() is called, the system makes a copy of the current CTM and pushes it onto the stack (now we have two copies of the current CTM on the stack).

When popCTM () is called, the system simply removes the CTM on top of the stack (now we have restored the CTM that was second to the removed CTM to be the current CTM).

Panning and Zooming

Two simple camera effects can be achieved by changing the position or size of the window. When the position of the window is, for example, moved to the left, an object in the scene that is visible through the window would appear moved to the right, much like what we see in the viewfinder when we move or pan a camera. On the other hand, if we fix the window on an object but reduce or increase its size, the object would appear bigger (zoom in) or smaller (zoom out), respectively.

Double Buffering

Producing an animation sequence by clearing the display screen and constructing the next frame of image often leads to flicker, since an image is erased almost as soon as it is completed. An effective solution to this problem is to have two frame buffers: one holds the image on display while the system draws a new image into the other. Once the new image is drawn, a call that looks like swapBuffer() would cause the two buffers to switch their roles.

Lookup Table Animation

We can sometimes animate a displayed image in the lookup table representation by changing or cycling the color values in the lookup table. For example, we may draw the monochromatic object in Fig. 5.14(a) into the frame buffer in several pre-determined locations, using consecutive lookup table entries for the color attribute in each location (see Fig. 5.15). We initialize lookup table entry

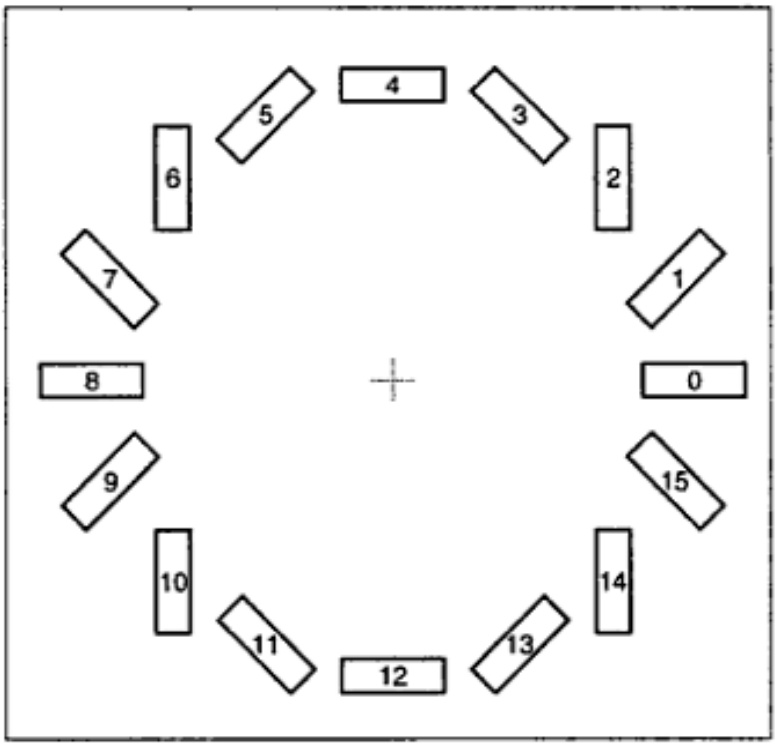


Fig. 5.15

0 with the color of the object, and all other entries with the background color. This means that in the beginning the object is visible only in its first position (labeled 0). Now if we simply reset entry 0 with the background color and entry 1 with the object color, we would have the object "moved" to its second position (labeled 1) without redrawing the image. The object's circular motion could hence be produced by cycling the object color through all relevant lookup table entries.

SOLVED PROBLEMS

5.1 Let

$$s_x = \frac{vx_{max} - vx_{min}}{wx_{max} - wx_{min}} \quad \text{and} \quad s_y = \frac{vy_{max} - vy_{min}}{wy_{max} - wy_{min}}$$

Express window-to-viewport mapping in the form of a composite transformation matrix.

$$N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -wx_{min} & -wy_{min} & 1 \end{pmatrix} \cdot \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ vx_{min} & vy_{min} & 1 \end{pmatrix}$$

$$= \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ -s_x\, wx_{min} + vx_{min} & -s_y\, wy_{min} + vy_{min} & 1 \end{pmatrix}$$

5.2 Find the normalization transformation that maps a window whose lower left corner is at (1, 1) and upper right corner is at (3, 5) onto

(a) a viewport that is the entire normalized device screen, and

(b) a viewport that has lower left corner at (0, 0) and upper right corner $\left(\frac{1}{2}, \frac{1}{2}\right)$.

From Solved Problem 5.1, we need only identify the appropriate parameters.

(a) The window parameters are $wx_{min} = 1$, $wx_{max} = 3$, $wy_{min} = 1$, and $wy_{max} = 5$. The viewport parameters are $vx_{min} = 0$, $vx_{max} = 1$, $vy_{min} = 0$, and $vy_{max} = 1$. Then $s_x = \frac{1}{2}$, $s_y = \frac{1}{4}$, and

$$N = \begin{pmatrix} 1/2 & 0 & 0 \\ 0 & 1/4 & 0 \\ -1/2 & -1/4 & 1 \end{pmatrix}$$

(b) The window parameters are the same as in (a). The viewport parameters are now $vx_{min} = 0$, $vx_{max} = \frac{1}{2}$, $vy_{min} = 0$, $vy_{max} = \frac{1}{2}$. Then $s_x = \frac{1}{4}$, $s_y = \frac{1}{8}$, and

$$N = \begin{pmatrix} 1/4 & 0 & 0 \\ 0 & 1/8 & 0 \\ -1/4 & -1/8 & 1 \end{pmatrix}$$

5.3 Find the complete viewing transformation that maps a window in world coordinates with x extent 1 to 10 and y extent 1 to 10 onto a viewport with x extent $\frac{1}{4}$ to $\frac{3}{4}$ and y extent 0 to $\frac{1}{2}$ in normalized device space, and then maps a workstation window with x extent $\frac{1}{4}$ to $\frac{1}{2}$ and y extent $\frac{1}{4}$ to $\frac{1}{2}$ in the normalized device space into a workstation viewport with x extent 1 to 10 and y extent 1 to 10 on the physical display device.

From Solved Problem 5.1, the parameters for the normalization transformation are $wx_{\min} = 1$, $wx_{\max} = 10$, $wy_{\min} = 1$, $wy_{\max} = 10$, and $vx_{\min} = \frac{1}{4}$, $vx_{\max} = \frac{3}{4}$, $vy_{\min} = 0$, and $vy_{\max} = \frac{1}{2}$. Then

$$s_x = \frac{1/2}{9} = \frac{1}{18} \qquad s_y = \frac{1/2}{9} = \frac{1}{18}$$

and

$$N = \begin{pmatrix} \frac{1}{18} & 0 & 0 \\ 0 & \frac{1}{18} & 0 \\ \frac{7}{36} & \frac{-1}{18} & 1 \end{pmatrix}$$

The parameters for the workstation transformation are $wx_{\min} = \frac{1}{4}$, $wx_{\max} = \frac{1}{2}$, $wy_{\min} = \frac{1}{4}$, $wy_{\max} = \frac{1}{2}$ and $vx_{\min} = 1$, $vx_{\max} = 10$, $vy_{\min} = 1$, and $vy_{\max} = 10$. Then

$$s_x = \frac{9}{1/4} = 36 \qquad s_y = \frac{9}{1/4} = 36$$

and

$$W = \begin{pmatrix} 36 & 0 & 0 \\ 0 & 36 & 0 \\ -8 & -8 & 1 \end{pmatrix}$$

The complete viewing transformation V is

$$V = N \cdot W = \begin{pmatrix} \frac{1}{18} & 0 & 0 \\ 0 & \frac{1}{18} & 0 \\ \frac{7}{36} & \frac{-1}{18} & 1 \end{pmatrix} \cdot \begin{pmatrix} 36 & 0 & 0 \\ 0 & 36 & 0 \\ -8 & -8 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ -1 & -10 & 1 \end{pmatrix}$$

5.4 Find a normalization transformation from the window whose lower left corner is at (0, 0) and upper right corner is at (4, 3) onto the normalized device screen so that aspect ratios are preserved.

The window aspect ratio is $a_w = \frac{4}{3}$. Unless otherwise indicated, we shall choose a viewport that is as large as possible with respect to the normalized device screen. To this end, we choose the x extent from 0 to 1 and the y extent from 0 to $\frac{3}{4}$. So

$$a_v = \frac{1}{3/4} = \frac{4}{3}$$

As in Solved Problem 5.2, with parameters $wx_{\min} = 0$, $wx_{\max} = 4$, $wy_{\min} = 0$, $wy_{\max} = 3$ and $vx_{\min} = 0$, $vx_{\max} = 1$, $vy_{\min} = 0$, $vy_{\max} = \frac{3}{4}$,

$$N = \begin{pmatrix} \frac{1}{4} & 0 & 0 \\ 0 & \frac{1}{4} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

5.5 Find the normalization transformation N which uses the rectangle $A(1, 1)$, $B(5, 3)$, $C(4, 5)$, $D(0, 3)$ as a window [Fig. 5.16(a)] and the normalized device screen as a viewport [Fig. 5.16(b)].

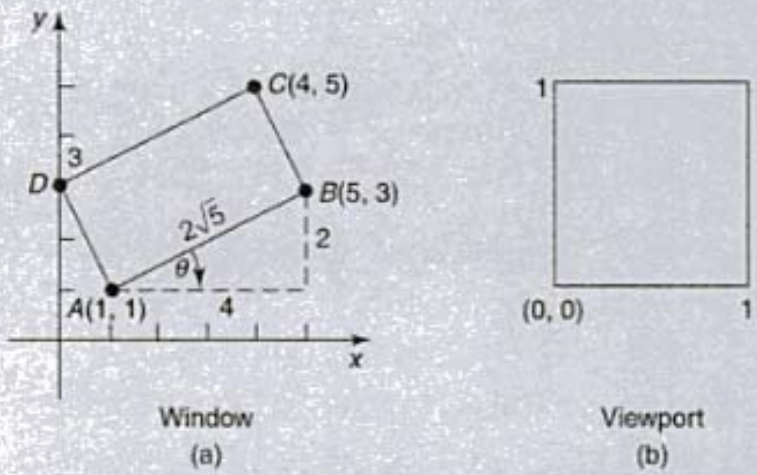


Fig. 5.16

We will first rotate the rectangle about A so that it is aligned with the coordinate axes. Next, as in Solved Problem 5.1, we calculate s_x and s_y and finally we compose the rotation and the transformation N (from Problem 5.1) to find the required normalization transformation N_R.

The slope of the line segment $\overline{AB}$ is

$$m = \frac{3-1}{5-1} = \frac{1}{2}$$

Looking at Fig. 5.11, we see that $-\theta$ will be the direction of the rotation. The angle θ is determined from the slope of a line (Appendix 1) by the equation $\tan\theta = \frac{1}{2}$. Then

$$\sin\theta = \frac{1}{\sqrt{5}} \text{ and so } \sin(-\theta) = -\frac{1}{\sqrt{5}},\ \cos\theta = \frac{2}{\sqrt{5}},\ \cos(-\theta) = -\frac{2}{\sqrt{5}}$$

The rotation matrix about $A(1, 1)$ is then (Chapter 4, Problem 4.4):

$$R_{-\theta, A} = \begin{pmatrix} \frac{2}{\sqrt{5}} & \frac{-1}{\sqrt{5}} & 0 \\ \frac{1}{\sqrt{5}} & \frac{2}{\sqrt{5}} & 0 \\ \left(1-\frac{3}{\sqrt{5}}\right) & \left(1-\frac{1}{\sqrt{5}}\right) & 1 \end{pmatrix}$$

The x extent of the rotated window is the length of $\overline{AB}$. Similarly, the y extent is the length of $\overline{AD}$. Using the distance formula (Appendix 1) to calculate these lengths yields

$$d(A, B) = \sqrt{2^2 + 4^2} = \sqrt{20} = 2\sqrt{5}\ \ d(A,D) = \sqrt{1^2 + 2^2} = \sqrt{5}$$

Also, the x extent of the normalized device screen is 1, as is the y extent. Calculating s_x and s_y,

$$s_x = \frac{\text{viewport } x \text{ extent}}{\text{window } x \text{ extent}} = \frac{1}{2\sqrt{5}} \qquad s_y = \frac{\text{viewport } y \text{ extent}}{\text{window } y \text{ extent}} = \frac{1}{\sqrt{5}}$$

so

$$N = \begin{pmatrix} \frac{1}{2\sqrt{5}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{5}} & 0 \\ \frac{-1}{2\sqrt{5}} & \frac{-1}{\sqrt{5}} & 1 \end{pmatrix}$$

The normalization transformation is then

$$N_R = R_{-\theta A} \cdot N = \begin{pmatrix} \frac{1}{5} & \frac{-1}{5} & 0 \\ \frac{1}{10} & \frac{2}{5} & 0 \\ \frac{-3}{10} & \frac{-1}{5} & 1 \end{pmatrix}$$

5.6 Let R be the rectangular window whose lower left-hand corner is at $L(-3, 1)$ and upper right-hand corner is at $R(2, 6)$. Find the region codes for the endpoints in Fig. 5.17.

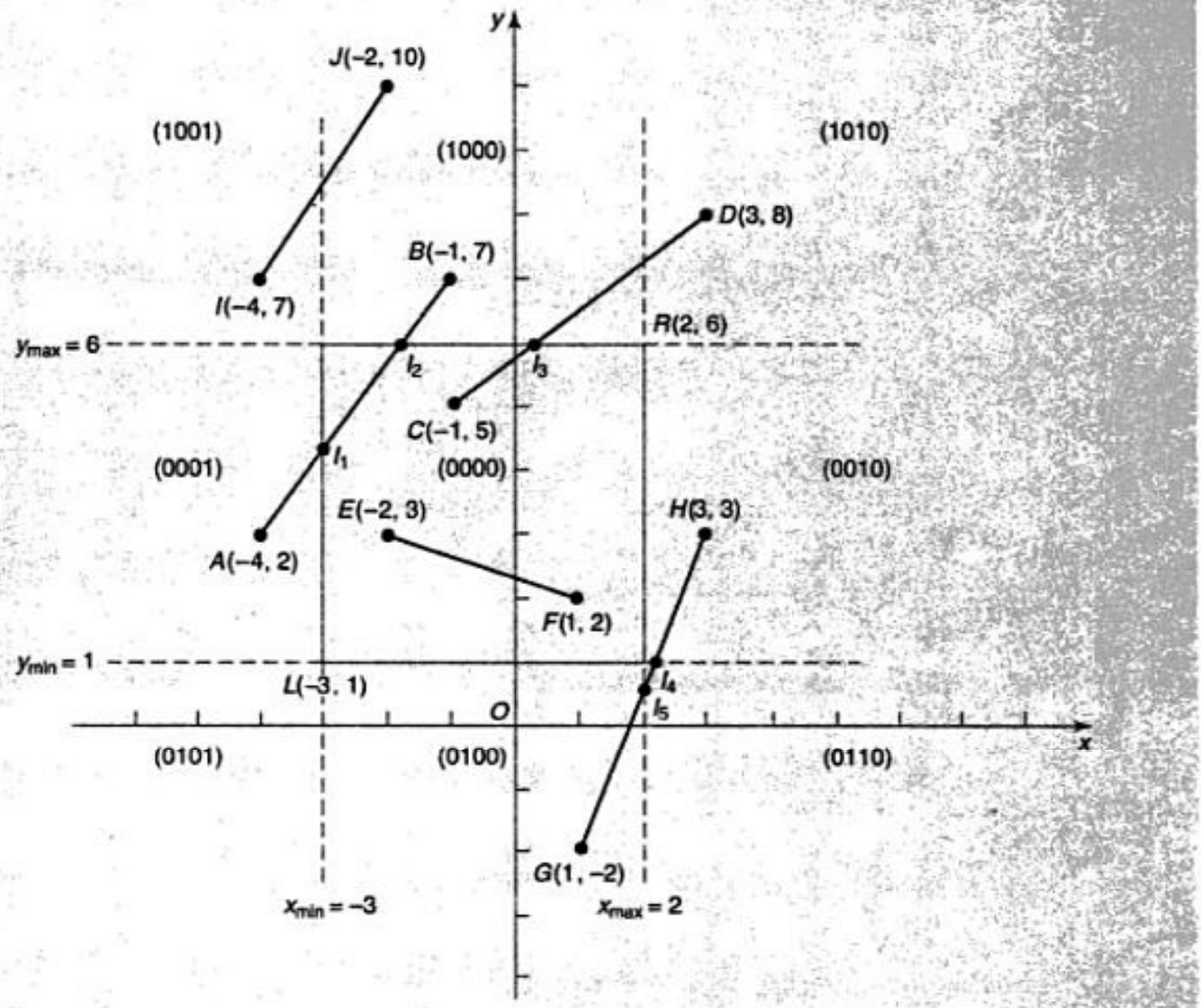


Fig. 5.17

The region code for point (x, y) is set according to the scheme

Bit 1 = $\text{sign}(y - y_{max}) = \text{sign}(y - 6)$ Bit 3 = $\text{sign}(x - x_{max}) = \text{sign}(x - 2)$

Bit 2 = $\text{sign}(y_{min} - y) = \text{sign}(1 - y)$ Bit 4 = $\text{sign}(x_{min} - x) = \text{sign}(-3 - x)$

Here

$$\text{sign}(a) = \begin{cases} 1 & \text{if } a \text{ is positive} \\ 0 & \text{otherwise} \end{cases}$$

So

$A(-4, 2) \rightarrow 0001$ $F(1, 2) \rightarrow 0000$

$B(-1, 7) \rightarrow 1000$ $G(1, -2) \rightarrow 0100$

$C(-1, 5) \rightarrow 0000$ $H(3,3) \rightarrow 0010$

$D(3, 8) \rightarrow 1010$ $I(-4, 7) \rightarrow 1001$
$E(-2, 3) \rightarrow 0000$ $J(-2, 10) \rightarrow 1000$

5.7 Clipping against rectangular windows whose sides are aligned with the x and y axes involves computing intersections with vertical and horizontal lines. Find the intersection of a line segment $\overline{P_1P_2}$ [joining $P_1(x_1, y_1)$ to $P_2(x_2, y_2)$] with

(a) the vertical line $x = a$ and
(b) the horizontal line $y = b$.

We write the equation of $\overline{P_1P_2}$ in parametric form (Appendix 1, Problem A1.23):

$$\begin{cases} x = x_1 + t(x_2 - x_1) & (5.1) \\ y = y_1 + t(y_2 - y_1), & (5.2) \end{cases} \qquad 0 \le t \le 1$$

(a) Since $x = a$, we substitute this into equation (5.1) and find $t = (a - x_1)/(x_2 - x_1)$. Then, substituting this value into equation (5.2), we find that the intersection point is $x_I = a$ and

$$y_I = y_1 + \left(\frac{a - x_1}{x_2 - x_1}\right)(y_2 - y_1)$$

(b) Substituting $y = b$ into equation (5.2), we find $t = (b - y_1)/(y_2 - y_1)$. When this is placed into equation (5.1), the intersection point is $y_1 = b$ and

$$x_I = x_1 + \left(\frac{b - y_1}{y_2 - y_1}\right)(x_2 - x_1)$$

5.8 Find the clipping categories for the line segments in Solved Problem 5.6 (see Fig. 5.17).

We place the line segments in their appropriate categories by testing the region codes found in Problem 5.6.

Category 1 (visible): $\overline{EF}$ since the region code for both endpoints is 0000.
Category 2 (not visible): $\overline{IJ}$ since (1001) AND (1000) = 1000 (which is not 0000).
Category 3 (candidates for clipping): $\overline{AB}$ since (0001) AND (1000) = 0000, $\overline{CD}$ since (0000) AND(1010) = 0000, and $\overline{GH}$ since (0100) AND (0010) = 0000.

5.9 Use the Cohen-Sutherland algorithm to clip the line segments in Problem 5.6 (see Fig. 5.17).

From Solved Problem 5.8, the candidates for clipping are $\overline{AB}$, $\overline{CD}$ and $\overline{GH}$.

In *clipping* $\overline{AB}$, the code for A is 0001. To push the 1 to 0, we clip against the boundary line $x_{min} = -3$. The resulting intersection point is $I_1\left(-3, 3\frac{2}{3}\right)$. We clip (do not display) $\overline{AI_1}$ and work on $\overline{I_1B}$. The code for I_1 is 0000. The clipping category for $\overline{I_1B}$ is 3 since

(0000) AND (1000) is (0000). Now B is outside the window (i.e. its code is 1000), so we push the 1 to a 0 by clipping against the line $y_{max} = 6$. The resulting intersection is $I_2\left(-1\frac{3}{5}, 6\right)$. Thus $\overline{I_2B}$ is clipped. The code for I_2 is 0000. The remaining segment $\overline{I_1I_2}$ is displayed since both endpoints lie in the window (i.e. their codes are 0000).

For *clipping* $\overline{CD}$, we start with D since it is outside the window. Its code is 1010. We push the first 1 to a 0 by clipping against the line $y_{max} = 6$. The resulting intersection I_3 is $\left(\frac{1}{3}, 6\right)$ and its code is 0000. Thus $\overline{I_3D}$ is clipped and the remaining segment $\overline{CI_3}$ has both endpoints coded 0000 and so it is displayed.

For *clipping* $\overline{GH}$, we can start with either G or H since both are outside the window. The code for G is 0100, and we push the 1 to a 0 by clipping against the line $y_{min} = 1$. The resulting intersection point is $I_4\left(2\frac{1}{5}, 1\right)$, and its code is 0010. We clip $\overline{GI_4}$ and work on $\overline{I_4H}$. Segment $\overline{I_4H}$ is not displayed since (0010) AND (0010) = 0010.

5.10 **Clip line segment $\overline{CD}$ of Solved Problem 5.6 by using the midpoint subdivision process.**

The midpoint subdivision process is based on repeated bisections. To avoid continuing indefinitely, we agree to say that a point (x_1, y_1) lies on any of the boundary lines of the rectangle, say, boundary line $x = x_{max}$, for example, if $-\text{TOL} \le x_1 - x_{max} \le \text{TOL}$. Here TOL is a prescribed tolerance, some small number that is set before the process begins.

To clip $\overline{CD}$, we determine that it is in category 3. For this problem we arbitrarily choose TOL = 0.1. We find the midpoint of $\overline{CD}$ to be $M_1(1, 6.5)$. Its code is 1000.

So $\overline{M_1D}$ is not displayed since (1000) AND (1010) = 1000. We further subdivide $\overline{CM_1}$ since (0000) AND (1000) = 0000. The midpoint of $\overline{CM_1}$ is $M_2(0, 5.75)$; the code for M_2 is 0000. Thus $\overline{CM_2}$ is displayed since both endpoints are 0000 and $\overline{M_2M_1}$ is a candidate for clipping. The midpoint of $\overline{M_2M_1}$ is $M_3(0.5, 6.125)$, and its code is 1000. Thus $\overline{M_3M_1}$ is chipped and $\overline{M_2M_3}$ is subdivided. The midpoint of $\overline{M_2M_3}$ is $M_4(0.25, 5.9375)$, whose code is 0000. However, since $y_1 = 5.9375$ lies within the tolerance 0.1 of the boundary line $y_{max} = 6$ – that is, $6 - 5.9375 = 0.0625 < 0.1$, we agree that M_4 lies on the boundary line $y_{max} = 6$. Thus $\overline{M_2M_4}$ is displayed and $\overline{M_4M_3}$ is not displayed. So the original line segment $\overline{CD}$ is clipped at M_4 and the process stops.

5.11 Suppose that in an implementation of the Cohen–Sutherland algorithm we choose boundary lines in the top–bottom–right–left order to clip a line in category 3, draw a picture to show a worst-case scenario, i.e. one that involves the highest number of iterations.

See Fig. 5.18.

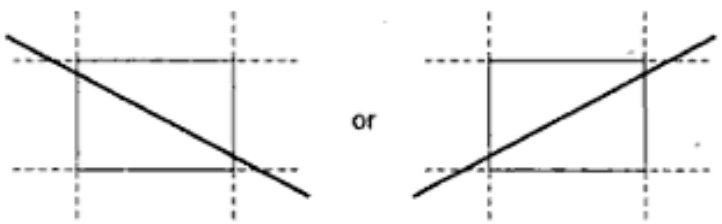


Fig. 5.18

5.12 Use the Liang–Barsky algorithm to clip the lines in Fig. 5.19.

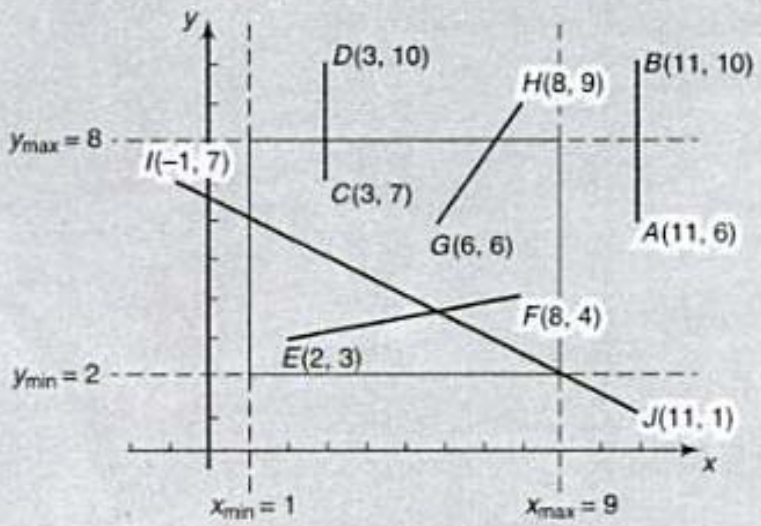


Fig. 5.19

For line AB, we have

$$p_1 = 0 \qquad q_1 = 10$$
$$p_2 = 0 \qquad q_2 = -2$$
$$p_3 = -4 \qquad q_3 = 4$$
$$p_4 = 4 \qquad q_4 = 2$$

Since $p_2 = 0$ and $q_2 < -2$, AB is completely outside the right boundary.

For line CD, we have

$$p_1 = 0 \qquad q_1 = 2$$
$$p_2 = 0 \qquad q_2 = 6$$
$$p_3 = -3 \qquad q_3 = 5 \qquad r_3 = -\frac{5}{3}$$
$$p_4 = 3 \qquad q_4 = 1 \qquad r_4 = \frac{1}{3}$$

Thus $u_1 = \max\left(0, -\frac{5}{3}\right) = 0$ and $u_2 = \min\left(1, \frac{1}{3}\right) = \frac{1}{3}$. Since $u_1 < u_2$, the two endpoints of the clipped line are (3, 7) and $\left(3, 7 + 3\left(\frac{1}{3}\right)\right) = 3, 8$.

For line EF, we have

$$p_1 = -6 \qquad q_1 = 1 \qquad r_1 = -\frac{1}{6}$$
$$p_2 = 6 \qquad q_2 = 7 \qquad r_2 = \frac{7}{6}$$
$$p_3 = -1 \qquad q_3 = 1 \qquad r_3 = -\frac{1}{1}$$
$$p_4 = 1 \qquad q_4 = 5 \qquad r_4 = \frac{5}{1}$$

Thus $u_1 = \max\left(0, -\frac{1}{6}, -1\right) = 0$ and $u_2 = \min\left(1, \frac{7}{6}, 5\right) = 1$. Since $u_1 = 0$ and $u_2 = 1$, line EF is completely inside the clipping window.

For line GH, we have

$$p_1 = -2 \qquad q_1 = 5 \qquad r_1 = -\frac{5}{2}$$

$$p_2 = 2 \qquad q_2 = 3 \qquad r_2 = \frac{3}{2}$$

$$p_3 = -3 \qquad q_3 = 4 \qquad r_3 = -\frac{4}{3}$$

$$p_4 = 3 \qquad q_4 = 2 \qquad r_4 = \frac{2}{3}$$

Thus $u_1 = \max\left(0, -\frac{5}{2}, -\frac{4}{3}\right) = 0$ and $u_2 = \min\left(1, \frac{3}{2}, \frac{2}{3}\right) = \frac{2}{3}$. Since $u_1 < u_2$, the two end-points of the clipped line are (6, 6) and $\left(6 + 2\left(\frac{2}{3}\right), 6 + 3\left(\frac{2}{3}\right)\right) = \left(7\frac{1}{3}, 8\right)$

For line IJ, we have

$$p_1 = -12 \qquad q_1 = -2 \qquad r_1 = \frac{1}{6}$$

$$p_2 = 12 \qquad q_2 = 10 \qquad r_2 = \frac{5}{6}$$

$$p_3 = 6 \qquad q_3 = 5 \qquad r_3 = \frac{5}{6}$$

$$p_4 = -6 \qquad q_4 = 1 \qquad r_4 = -\frac{1}{6}$$

Thus $u_1 = \max\left(0, \frac{1}{6}, -\frac{1}{6}\right) = \frac{1}{6}$ and $u_2 = \min\left(1, \frac{5}{6}, \frac{5}{6}\right) = \frac{5}{6}$. Since $u_1 < u_2$, the two end-points of the clipped line are $\left(-1 + 12\left(\frac{1}{6}\right), 7 + (-6)\left(\frac{1}{6}\right)\right)$ and $\left(-1 + 12\left(\frac{5}{6}\right), 7 + (-6)\left(\frac{5}{6}\right)\right)$ = (9, 2)

5.13 How can we determine whether a point $P(x, y)$ lies to the left or to the right of a line segment joining the points $A(x_1, y_1)$ and $B(x_2, y_2)$?

Refer to Fig. 5.20. Form the vectors **AB** and **AP**. If the point P is to the left of **AB**, then by the definition of the cross product of two vectors (Appendix 2) the vector **AB** × **AP**

points in the direction of the vector **K** perpendicular to the xy plane (see Fig. 5.20). If it lies to the right, the cross product points in the direction $-\mathbf{K}$. Now

$$\mathbf{AB} = (x_2 - x_1)\mathbf{I} + (y_2 - y_1)\mathbf{J}$$

$$\mathbf{AP} = (x - x_1)\mathbf{I} + (y - y_1)\mathbf{J}$$

So

$$\mathbf{AB} \times \mathbf{AP} = [(x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)]\mathbf{K}$$

Then the direction of this cross product is determined by the number

$$\bar{C} = (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)$$

If $\bar{C}$ is positive, P lies to the left of **AB**. If $\bar{C}$ is negative, then P lies to the right of **AB**.

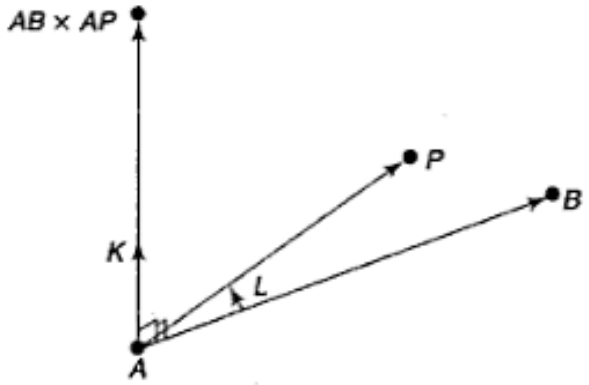


Fig. 5.20

5.14 Draw a flowchart illustrating the logic of the Sutherland—Hodgman algorithm.

The algorithm inputs the vertices of a polygon one at a time. For each input vertex, either zero, one, or two output vertices will be generated depending on the relationship of the input vertices to the clipping edge E.

We denote by P the input vertex, S the previous input vertex, and F the first arriving input vertex. The vertex or vertices to be output are determined according to the logic illustrated in the flowchart in Fig. 5.21. Recall that a polygon with n vertices $P_1, P_2, \ldots, P_n$ has n edges $\overline{P_1P_2}, \ldots, \overline{P_{n-1}P_n}$ and the edge $\overline{P_nP_1}$ closing the polygon. In order to avoid the need to duplicate the input of P_1 as the final input vertex (and a corresponding mechanism to duplicate the final output vertex to close the polygon), the closing logic shown in the flowchart in Fig. 5.22 is called after processing the final input vertex P_n.

5.15 Clip the polygon $P_1, \ldots, P_9$ in Fig. 5.23 against the window $ABCD$ using the Sutherland—Hodgman algorithm.

At each stage the new output polygon, whose vertices are determined by applying the Sutherland—Hodgman algorithm (Problem 5.14), is passed on to the next clipping edge of the window $ABCD$. The results are illustrated in Figs. 5.24 through 5.27.

5.16 Clip the polygon $P_1, \ldots, P_8$ in Fig. 5.10 against the rectangular clipping window using the Sutherland-Hodgman algorithm.

We first clip against the top boundary line, then the left, and finally the bottom. The right boundary is omitted since it does not affect any vertex list. The intermediate and final results are in Fig. 5.28.

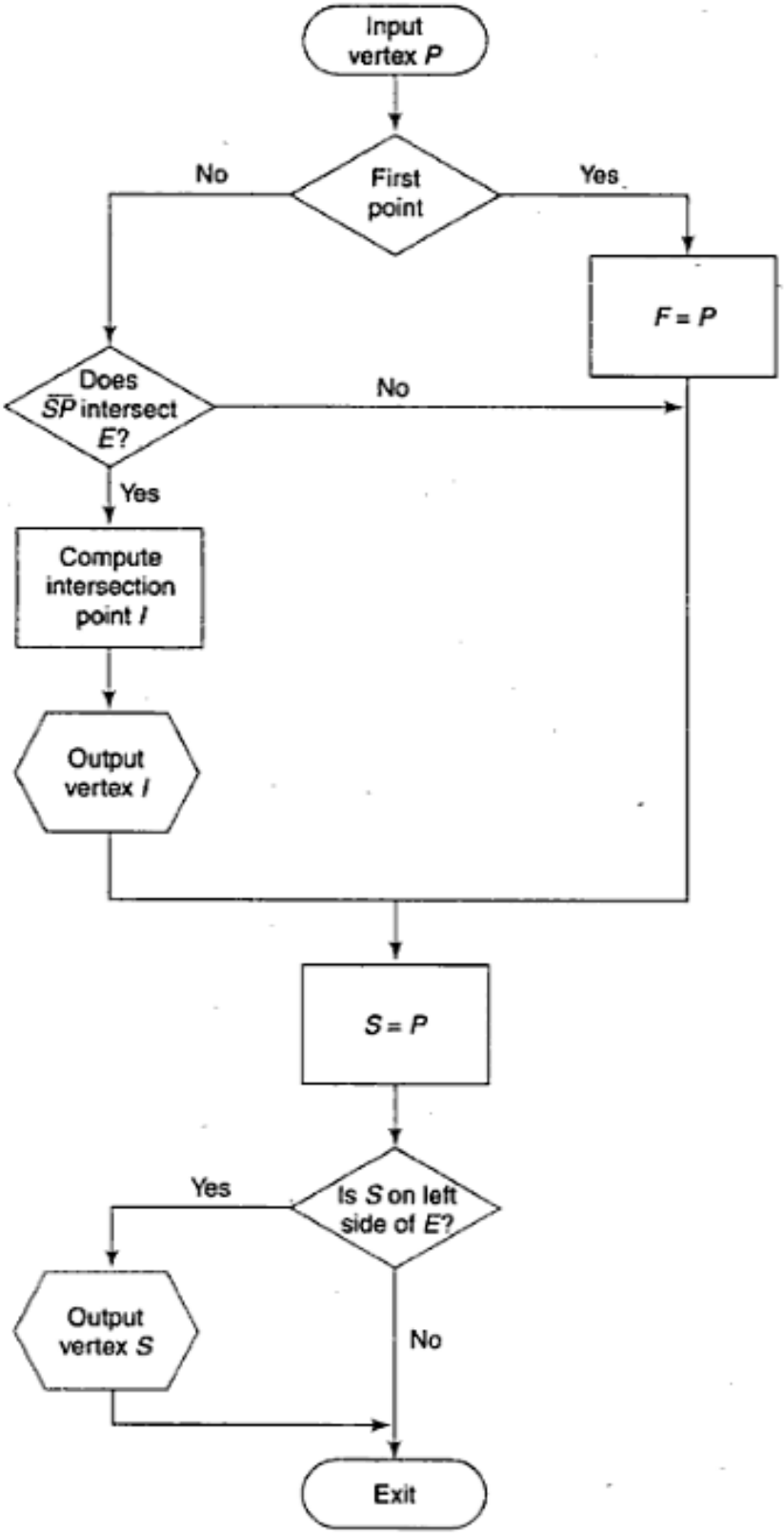


Fig. 5.21

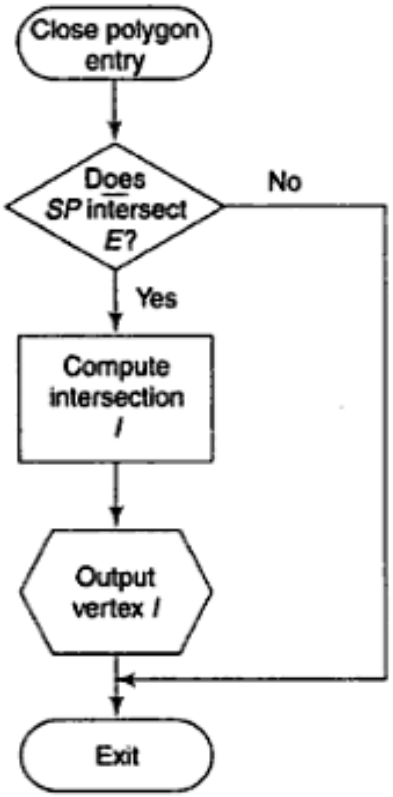


Fig. 5.22

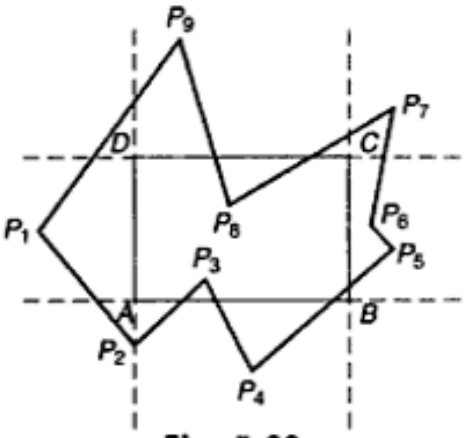


Fig. 5.23

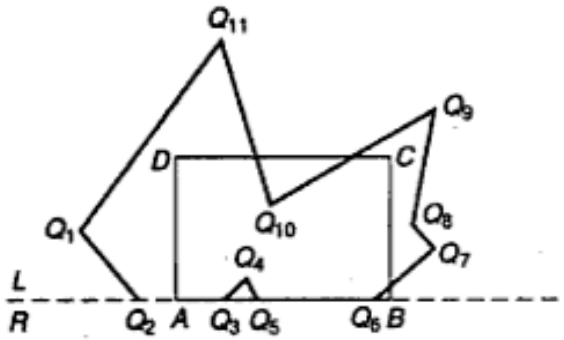


Fig. 5.24 *Clip against* $\overline{AB}$

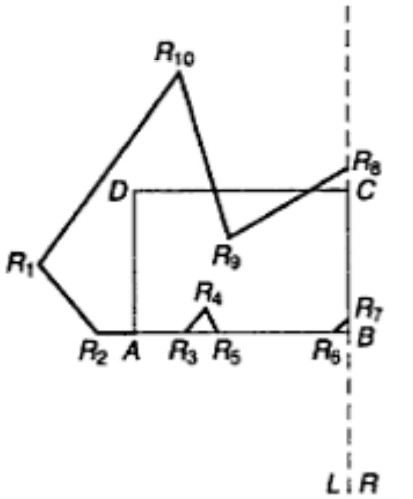


Fig. 5.25 *Clip against* $\overline{BC}$

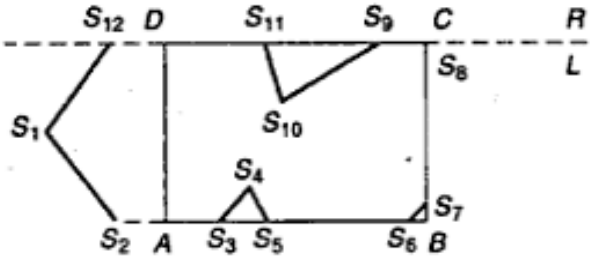


Fig. 5.26 *Clip against* $\overline{CD}$

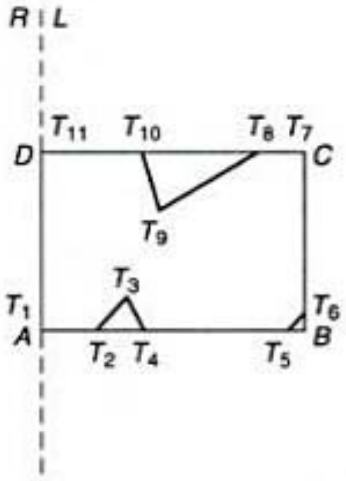


Fig. 5.27 *Clip against* $\overline{DA}$

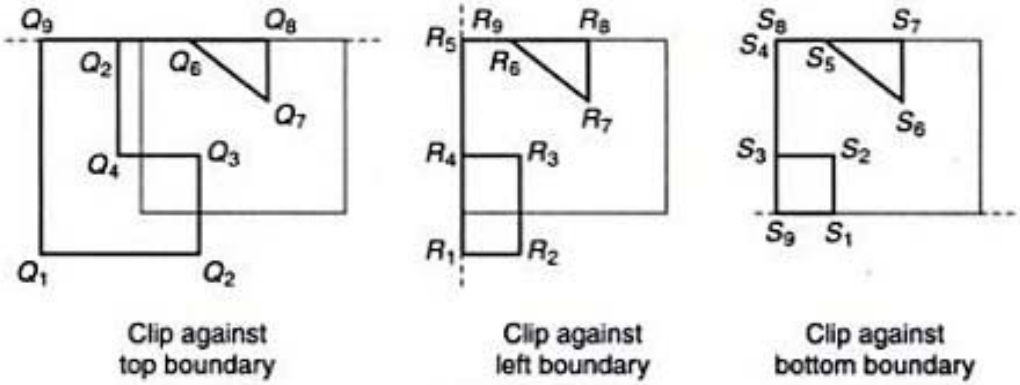


Fig. 5.28

5.17 Use the Weiler–Atherton algorithm to clip the polygon in Fig. 5.29(a).

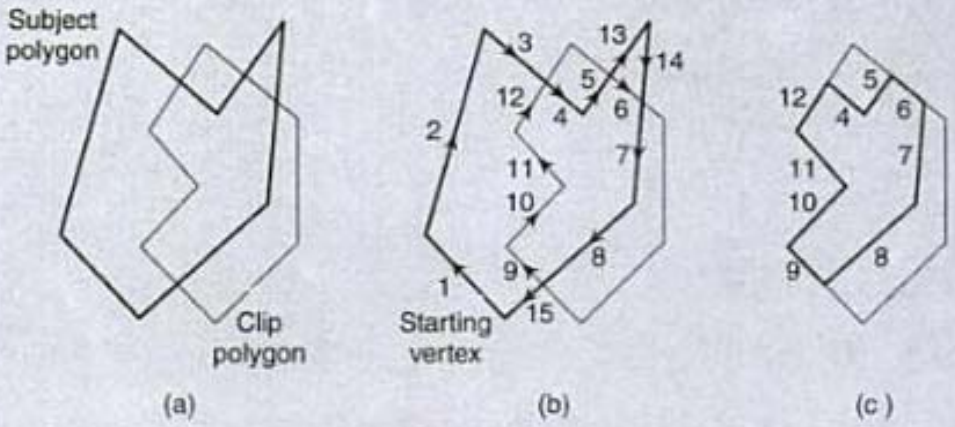


Fig. 5.29

See Figs 5.29(b) and (c).

5.18 Consider the example in Section 5.5, where the object would appear turning slowly around its center even if we set $\Delta\alpha = 0$. How can the orientation of the object be kept constant while making it rotate around the center of the display area?

$\Delta\alpha = -\Delta\beta$, i.e. $\alpha = -\beta$

5.19 How can the flag in Fig. 5.30(a) be animated that may be in two different positions using lookup table animation?

See Fig. 5.30(b). The area where position 1 overlaps position 2 is assigned entry 0 that has the colour of the flag. The rest of position 1 is assigned entry 1 and that of position 2 entry 2. Now we only need to alternate entries 1 and 2 between the flag colour and the background colour.

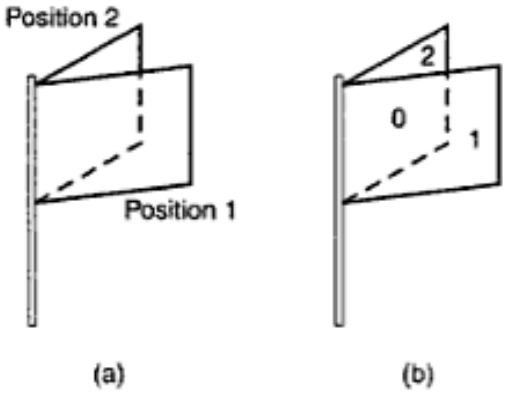


Fig. 5.30

SUPPLEMENTARY PROBLEMS

5.1 Find the workstation transformation that maps the normalized device screen onto a physical device whose x extent is 0 to 199 and y extent is 0 to 639 where the origin is located at the (a) lower left corner and (b) upper left corner of the device.

5.2 Show that for a viewing transformation, $s_x = s_y$ if and only if $a_w = a_v$, where a_w is the aspect ratio of the window and a_v the aspect ratio of the viewport.

5.3 Find the normalization transformation which uses a circle of radius five units and center (1, 1) as a window and a circle of radius $\frac{1}{2}$ and center $\left(\frac{1}{2}, \frac{1}{2}\right)$ as a viewport.

5.4 Describe how clipping a line against a circular window (or viewport) might proceed. Refer to Fig. 5.31.

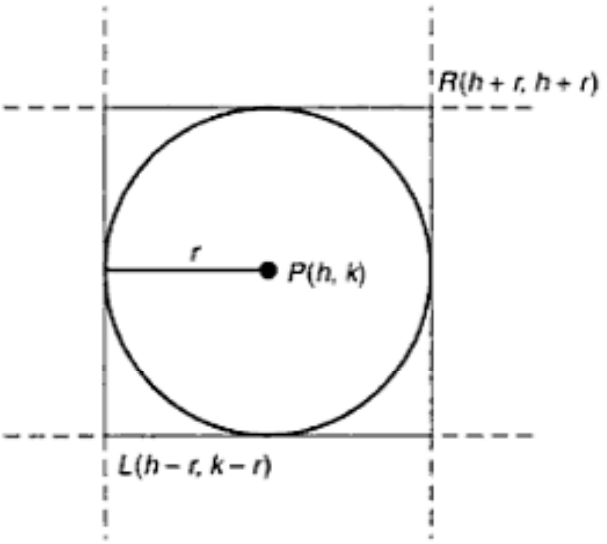


Fig. 5.31

5.5 Use the Sutherland–Hodgman algorithm to clip the line segment joining $P_1(-1, 2)$ to $P_2(6,4)$ against the rotated window in Problem 5.5.

ANSWERS TO SUPPLEMENTARY PROBLEMS

5.1 From Problem 5.1 we need only identify the appropriate parameters.

(a) The window parameters are $wx_{\min} = 0$, $wx_{\max} = 1$, $wy_{\min} = 0$ and $wy_{\max} = 1$. The viewport parameters are $vx_{\min} = 0$, $vx_{\max} = 199$, $vy_{\min} = 0$, and $vy_{\max} = 639$. Then $s_x = 199$, $s_y = 639$, and

$$W = \begin{pmatrix} 199 & 0 & 0 \\ 0 & 639 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(b) The parameters are the same, but the device y coordinate is now $639 - y$ (see Problem 2.8) instead of the y value computed by W in (a)

$$W = \begin{pmatrix} 199 & 0 & 0 \\ 0 & -639 & 0 \\ 0 & 639 & 1 \end{pmatrix}$$

5.2 If $s_x = s_y$, then

$$\frac{vx_{max} - vx_{min}}{wx_{max} - wx_{min}} = \frac{vy_{max} - vy_{min}}{wy_{max} - wy_{min}}$$

or $$\frac{wy_{max} - wy_{min}}{wx_{max} - wx_{min}} = \frac{vy_{max} - vy_{min}}{vx_{max} - vx_{min}}$$

Inverting, we have $a_w = a_v$.

A similar argument shows that the aspect ratios are equal, $a_w = a_v$, the scale factors are equal, $s_x = s_y$.

5.3 We from N by composing (1) a translation mapping the center (1, 1) to the center $\left(\frac{1}{2}, \frac{1}{2}\right)$ and (2) a scaling about $C\left(\frac{1}{2}, \frac{1}{2}\right)$ with uniform scaling factor $s = \frac{1}{10}$, so

$N = T_{\mathbf{v}} \cdot S_{1/10,\ 1/10}$

where $\mathbf{v} = -\frac{1}{2}\mathbf{I} - \frac{1}{2}\mathbf{J}$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & -\frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{10} & 0 & 0 \\ 0 & \frac{1}{10} & 0 \\ \frac{9}{20} & \frac{9}{20} & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{10} & 0 & 0 \\ 0 & \frac{1}{10} & 0 \\ \frac{2}{5} & \frac{2}{5} & 1 \end{pmatrix}$$

5.4 Let the clipping region be a circle with center at $O(h, k)$ and radius r. We reduce the number of candidates for clipping by assigning region codes as in the Cohen–Sutherland algorithm. To do this, we use the circumscribed square with lower left corner at $(h - r, k - r)$ and upper right corner at $(h + r, k + r)$ to preprocess the line segments. However, we now have only two clipping categories—not displayed and candidates for clipping. Next, we decide which line segments are to be displayed. Since the (non-parametric) equation of the circle is $(x - h)^2 + (y - k)^2 = r^2$, the quantity $K(x, y) = (x - h)^2 + (y - k)^2 - r^2$ determines whether a point $P(x, y)$ is inside, on, or outside the circle. So, if $K \leq 0$ for both endpoints P_1 and P_2 of a line segment, both points are inside or on the circle and so the line segment is displayed. If $K > 0$ for either P_1 or P_2 or both, we calculate the intersection(s) of the line segment and the circle. Using parametric representa-tions, we find (Appendix 1, Problem A1.24) that the intersection parameter is

$$t_I = \frac{-S \pm \sqrt{S^2 - L^2 C}}{L^2}$$

where

$L^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$

$S = (x_1 - h)(x_2 - x_1) + (y_1 - k)(y_2 - y_1)$

$C = (x_1 - h)^2 + (y_1 - k)^2 - r^2$

If $0 \leq t_I \leq 1$, the actual intersection point(s) $I(\bar{x}, \bar{y})$ is (are)

$\bar{x} = x_1 + t_I(x_2 - x_1)$ $\bar{y} = y_1 - t_I(y_2 - y_1)$

So, if $K > 0$ for either P_1 or P_2 (or both) we first relabel the endpoints so that P_1 satisfies $K > 0$. Next we calculate t_I. The following situations arise:

1. $S^2 - L^2C < 0$. Then t_I is undefined and no intersection takes place. The line segment is not displayed.
2. $S^2 - L^2C = 0$. There is exactly one intersection. If $t_I > 1$ or $t_I < 0$, the intersection point is on the extended line, and so there is no actual intersection. The line is not displayed. If $0 < t_I \leq 1$, $\overline{P_1P_2}$ is tangent to the circle at point I, so only I is displayed.
3. $S^2 - L^2C > 0$. Then there are two values for t_I, t_I^+, and t_I^-. If $0 < t_I^+$, $t_I^- \leq 1$, the line segment $\overline{I^- I^+}$ is displayed and the segments (assuming $t_I^+ > t_I^-$) $\overline{P_1 I^-}$ and $\overline{I^+ P_2}$ are clipped. If only one value, say t_I^+, satisfies $0 < t_I^+ \leq 1$, there is one actual intersection and one apparent intersection. Since in this case P_2 is either point I^+ or inside the circle $\overline{P_1 I^+}$ is clipped and $\overline{I^+ P_2}$ is displayed. If $t_I^+, t_I^- > 1$ or $t_I^+, t_I^- < 0$, then $\overline{P_1P_2}$ is not displayed.

5.5 Following the logic of the Sutherland–Hodgman algorithm as described in Problem 5.14, we first clip the "polygon P_1P_2 against edge $\overline{AB}$ of the window:

1. $\overline{AB}$. We first determine which side of $\overline{AB}$ the points P_1 and P_2 lie. Calculating the quantity (see Problem 5.13), we have

$$\bar{C} = (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)$$

With point $A = (x_1, y_1)$ and point $B = (x_2, y_2)$ we find $\bar{C} = 8$ for point P_1 and $\bar{C} = 2$ for point P_2. So both points lie on the left of $\overline{AB}$. Consequently, the algorithm will output both P_1 and P_2.

2. $\overline{BC}$. Setting point $B = (x_1, y_1)$ and $\bar{C} = (x_2, y_2)$, we calculate $\bar{C} = 13$ for point P_1 of $\bar{C} = -3$ for point P_2. Thus P_1 is to the left of $\overline{BC}$ and P_2 is to right of $\overline{BC}$. We now find the intersection point I_1 of $\overline{P_1P_2}$ with the extended line $\overline{BC}$. From Problem A2.7 in Appendix 2, we have $I_1 = \left(4\frac{11}{16}, 3\frac{5}{8}\right)$. Following the algorithm, points P_1 and I_1 are passed on to be clipped.
3. $\overline{CD}$. Proceeding as before, we find that $\bar{C} = 2$ for point P_1 and $\bar{C} = 6\frac{7}{8}$ for point I_1. So both points lie to the left of $\overline{CD}$ and consequently are passed on.
4. $\overline{DA}$. Setting point $D = (x_1, y_1)$ and $A = (x_2, y_2)$, we find $\bar{C} = -5$ for P_1 and $\bar{C} = 10$ for I_1. Then P_1 lies to the right of $\overline{DA}$ and I_1 to the left. The intersection point of $\overline{P_1I_1}$ with the extended edge $\overline{DA}$ is $I_2 = \left(\frac{5}{16}, 2\frac{3}{8}\right)$. The clipped line is the segment $\overline{I_1I_2}$.

Chapter Six

Three-Dimensional Transformations

Manipulation, viewing, and construction of three-dimensional graphic images requires the use of three-dimensional geometric and coordinate transformations. These transformations are formed by composing the basic transformations of translation, scaling, and rotation. Each of these transformations can be represented as a matrix transformation. This permits more complex transformations to be built up by use of matrix multiplication or concatenation.

As with two-dimensional transformations, two complementary points of view are adopted: either the object is manipulated directly through the use of geometric transformations, or the object remains stationary and the viewer's coordinate system is changed by using coordinate transformations. In addition, the construction of complex objects and scenes is facilitated by the use of instance transformations. The concepts and transformations introduced here are direct generalizations of those introduced in Chapter 4 for two-dimensional transformations.

6.1 GEOMETRIC TRANSFORMATIONS

With respect to some three-dimensional coordinate system, an object *Obj* is considered as a set of points:

$$Obj = \{P(x, y, z)\}$$

If the object is moved to a new position, we can regard it as a new object *Obj'*, all of whose coordinate points $P'(x', y', z')$ can be obtained from the original coordinate points $P(x, y, z)$ of *Obj* through the application of a geometric transformation.

Translation

An object is displaced a given distance and direction from its original position. The direction and displacement of the translation is prescribed by a vector

$$\mathbf{V} = a\mathbf{I} + b\mathbf{J} + c\mathbf{K}$$

The new coordinates of a translated point can be calculated by using the transformation

$$T_{\mathbf{v}}: \begin{cases} x' = x + a \\ y' = y + b \\ z' = z + c \end{cases}$$

(see Fig. 6.1). In order to represent this transformation as a matrix transformation, we need to use homogeneous coordinates (Appendix 2). The required homogeneous matrix transformation can then be expressed as

$$(x' \quad y' \quad z' \quad 1) = (x \quad y \quad z \quad 1)\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{pmatrix}$$

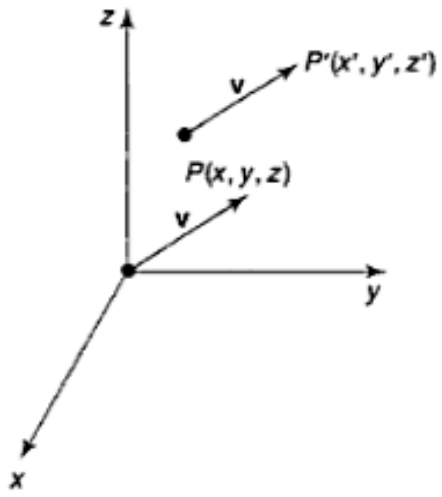


Fig. 6.1

Scaling

The process of scaling changes the dimensions of an object. The scale factor s determines whether the scaling is a magnification, $s > 1$, or a reduction, $s < 1$.

Scaling with respect to the origin, where the origin remains fixed, is effected by the transformation

$$S_{s_x,s_y,s_z}: \begin{cases} x' = s_x \cdot x \\ y' = s_y \cdot y \\ z' = s_z \cdot z \end{cases}$$

In matrix form this is

$$S_{s_x,s_y,s_z} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}$$

Rotation

Rotation in three dimensions is considerably more complex than rotation in two dimensions. In two dimensions, a rotation is prescribed by an angle of rotation θ and a center of rotation P. Three-dimensional rotations require the prescription of an angle of rotation and an axis of rotation. The

canonical rotations are defined when one of the positive x, y, or z coordinate axes is chosen as the axis of rotation. Then the construction of the rotation transformation proceeds just like that of a rotation in two dimensions about the origin (see Fig. 6.2).

Rotation about the z axis

From Chapter 4 we know that

$$R_{\theta,\mathbf{K}}:\begin{cases} x' = x\cos\theta - y\sin\theta \\ y' = x\sin\theta + y\cos\theta \\ z' = z \end{cases}$$

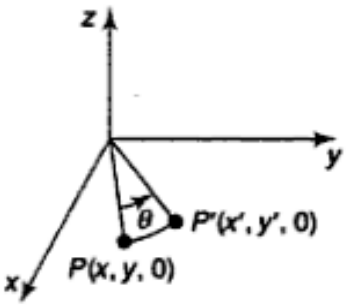


Fig. 6.2

Rotation about the y axis

An analogous derivation leads to

$$R_{\theta,\mathbf{J}}:\begin{cases} x' = x\cos\theta + z\sin\theta \\ y' = y \\ z' = -x\sin\theta + z\cos\theta \end{cases}$$

Rotation about the x axis

Similarly:

$$R_{\theta,\mathbf{I}}:\begin{cases} x' = x \\ y' = y\cos\theta - z\sin\theta \\ z' = y\sin\theta + z\cos\theta \end{cases}$$

Note that the direction of a positive angle of rotation is chosen in accordance to the right-hand rule with respect to the axis of rotation (Appendix 2).

The corresponding matrix transformations are

$$R_{\theta,\mathbf{K}} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R_{\theta,\mathbf{J}} = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix}$$

$$R_{\theta,\mathbf{I}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix}$$

The general use of rotation about an axis L can be built up from these canonical rotations using matrix multiplication (Problem 6.3).

6.2 COORDINATE TRANSFORMATIONS

We can also achieve the effects of translation, scaling, and rotation by moving the observer who views the object and by keeping the object stationary. This type of transformation is called a *coordinate transformation*.

We first attach a coordinate system to the observer and then move the observer and the attached coordinate system. Next, we recalculate the coordinates of the observed object with respect to this new observer coordinate system. The new coordinate values will be exactly the same as if the observer had remained stationary and the object had moved, corresponding to a geometric transformation (see Fig. 6.3).

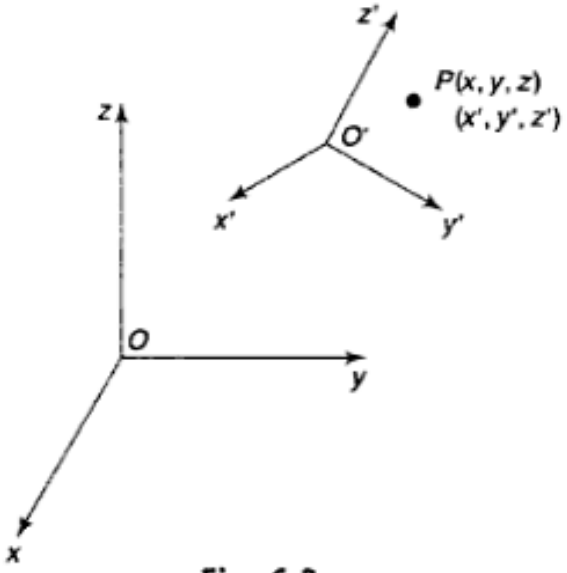


Fig. 6.3

If the displacement of the observer coordinate system to a new position is prescribed by a vector $\mathbf{V} = a\mathbf{I} + b\mathbf{J} + c\mathbf{K}$, a point $P(x, y, z)$ in the original coordinate system has coordinates $P(x', y', z')$ in the new coordinate system, and

$$\bar{T}_{\mathbf{V}}: \begin{cases} x' = x - a \\ y' = y - b \\ z' = z - c \end{cases}$$

The derivation of this transformation is completely analogous to that of the two-dimensional transformation (see Chapter 4).

Similar derivations hold for coordinate scaling and coordinate rotation transformations.

As in the two-dimensional case, we summarize the relationships between the matrix forms of the coordinate transformations and the geometric transformations:

	Coordinate Transformations	Geometric Transformations
Translation	$\bar{T}_{\mathbf{V}}$	$T_{\mathbf{V}}$
Rotation	$\bar{R}_{\theta}$	$R_{-\theta}$
Scaling	$\bar{S}_{s_x, s_y, s_z}$	$\bar{S}_{1/s_x, 1/s_y, 1/s_z}$

Inverse geometric and coordinate transformations are constructed by performing the reverse operation. Thus, for coordinate transformations (and similarly for geometric transformations):

$$\bar{T}_{\mathbf{V}}^{-1} = \bar{T}_{-\mathbf{V}} \qquad \bar{R}_{\theta}^{-1} = \bar{R}_{-\theta} \qquad \bar{S}_{s_x, s_y, s_z}^{-1} = \bar{S}_{1/s_x, 1/s_y, 1/s_z}$$

6.3 COMPOSITE TRANSFORMATIONS

More complex geometric and coordinate transformations are formed through the process of *composition of functions*. For matrix functions, however, the process of composition is equivalent to matrix multiplication or concatenation. In Problems 6.2, 6.3, 6.5, and 6.13, the following transformations are constructed:

1. $A_{\mathbf{V},\mathbf{N}}$ = aligning a vector **V** with a vector **N**.
2. $R_{\theta,L}$ = rotation about an axis L. This axis is prescribed by giving a direction vector **V** and a point P through which the axis passes.
3. $S_{s_x,s_y,s_z,P}$ = scaling with respect to an arbitrary point P.

In order to build these more complex transformations through matrix concatenation, we must be able to multiply translation matrices with rotation and scaling matrices. This necessitates the use of homogeneous coordinates and 4×4 matrices (Appendix 2). The standard 3×3 matrices of rotation and scaling can be represented as 4×4 homogeneous matrices by adjoining an extra row and column as follows:

$$\begin{pmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

These transformations are then applied to points $P(x, y, z)$ having the homogeneous form:

$$(x \quad y \quad z \quad 1)$$

Example 6.1

The matrix of rotation about the y axis has the homogeneous 4×4 form:

$$R_{\theta,J} = \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

6.4 SHEARING TRANSFORMATIONS

Similar to the two-dimensional shearing transformations, we have three-dimensional shearing transformations also. These transformations cause the image to slant in a given direction of x, y or z. The x-shear maintains the y and z coordinates but changes the values of x coordinates causing it to tilt left or right depending on the x-shear value so that the image get slanted towards x direction. Similarly y-shear and z-shear transformations yield slanting towards the y and z directions. The general form of the three-dimensional shearing transformation is given by

$$\begin{pmatrix} 1 & a & b & c \\ c & 1 & d & 0 \\ e & f & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

6.5 INSTANCE TRANSFORMATIONS

If an object is created and described in coordinates with respect to its own object coordinate space, we can place an instance or copy of it within a larger scene that is described in an independent coordinate space by the use of three-dimensional coordinate transformations. In this case, the transformations are referred to as *instance transformations*. The concepts and construction of three-dimensional instance transformations and the composite transformation matrix are completely analogous to the two-dimensional cases described in Chapter 4.

SOLVED PROBLEMS

6.1 Define *tilting* as a rotation about the x axis followed by a rotation about the y axis: (a) find the tilting matrix; (b) does the order of performing the rotation matter?

(a) We can find the required transformation T by composing (concatenating) two rotation matrices:

$$T = R_{\theta_x,\mathbf{I}} \cdot R_{\theta_y,\mathbf{J}}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x & 0 \\ 0 & -\sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_y & 0 & -\sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos\theta_y & 0 & -\sin\theta_y & 0 \\ \sin\theta_y \sin\theta_x & \cos\theta_x & \cos\theta_y \sin\theta_x & 0 \\ \sin\theta_y \cos\theta_x & -\sin\theta_x & \cos\theta_y \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) We multiply $R_{\theta_x,\mathbf{J}} \cdot R_{\theta_y,\mathbf{I}}$ to obtain the matrix

$$= \begin{pmatrix} \cos\theta_y & \sin\theta_x \sin\theta_y & -\cos\theta_x \sin\theta_y & 0 \\ 0 & \cos\theta_x & \sin\theta_x & 0 \\ \sin\theta_y & -\sin\theta_x \cos\theta_y & \cos\theta_x \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This is not the same matrix as in part (a); thus the order of rotation matters.

6.2 Find a transformation $A_{\mathbf{V}}$ which aligns a given vector $\mathbf{V}$ with the vector $\mathbf{K}$ along the positive z axis.

See Fig. 6.4(a). Let $\mathbf{V} = a\mathbf{I} + b\mathbf{J} + c\mathbf{K}$. We perform the alignment through the following sequence of transformations [Figs. 6.4(b) and 6.4(c)]:

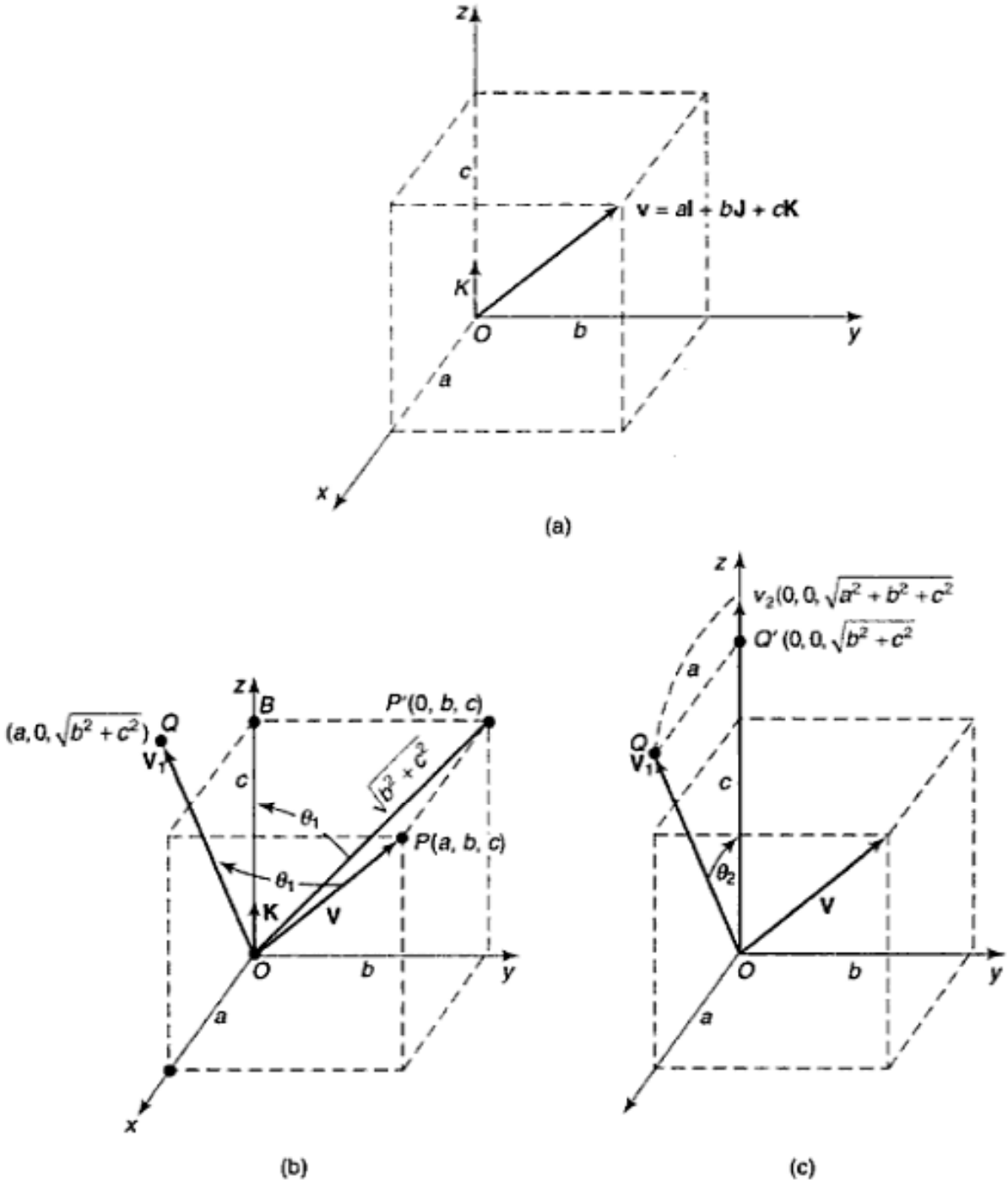


Fig. 6.4

1. Rotate about the x axis by an angle θ_1 so that $\mathbf{V}$ rotates into the upper half of the xz plane (as the vector $\mathbf{V}_1$).
2. Rotate the vector $\mathbf{V}_1$ about the y axis by an angle $-\theta_2$ so that $\mathbf{V}_1$ rotates to the positive z axis (as the vector $\mathbf{V}_2$).

Implementing step 1 from Fig. 6.4(b), we observe that the required angle of rotation θ_1 can be found by looking at the projection of $\mathbf{V}$ onto the yz plane. (We assume that b and c are not both zero.) From triangle $OP'B$:

$$\sin\theta_1 = \frac{b}{\sqrt{b^2+c^2}} \qquad \cos\theta_1 = \frac{c}{\sqrt{b^2+c^2}}$$

The required rotation is

$$R_{\theta_1,\mathbf{I}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \dfrac{c}{\sqrt{b^2+c^2}} & \dfrac{b}{\sqrt{b^2+c^2}} & 0 \\ 0 & -\dfrac{b}{\sqrt{b^2+c^2}} & \dfrac{c}{\sqrt{b^2+c^2}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Applying this rotation to the vector $\mathbf{V}$ produces the vector $\mathbf{V}_1$ with the components $(a, 0, \sqrt{b^2+c^2})$.

Implementing step 2 from Fig. 6.4(c), we see that a rotation of θ_2 degrees is required, and so from triangle OQQ':

$$\sin(-\theta_2) = -\sin\theta_2 = -\frac{a}{\sqrt{a^2+b^2+c^2}} \quad \text{and} \quad \cos(-\theta_2) = \cos\theta_2 = \frac{\sqrt{b^2+c^2}}{\sqrt{a^2+b^2+c^2}}$$

Then

$$R_{-\theta_2,\mathbf{J}} = \begin{pmatrix} \dfrac{\sqrt{b^2+c^2}}{\sqrt{a^2+b^2+c^2}} & 0 & \dfrac{a}{\sqrt{a^2+b^2+c^2}} & 0 \\ 0 & 1 & 0 & 0 \\ -\dfrac{a}{\sqrt{a^2+b^2+c^2}} & 0 & \dfrac{\sqrt{b^2+c^2}}{\sqrt{a^2+b^2+c^2}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Since $|\mathbf{V}| = \sqrt{a^2+b^2+c^2}$, and introducing the notation $\lambda = \sqrt{b^2+c^2}$, we find

$$A_{\mathbf{V}} = R_{\theta_1,\mathbf{I}} \cdot R_{-\theta_2,\mathbf{J}}$$

$$= \begin{pmatrix} \dfrac{\lambda}{|\mathbf{V}|} & 0 & \dfrac{a}{|\mathbf{V}|} & 0 \\ \dfrac{-ab}{\lambda|\mathbf{V}|} & \dfrac{c}{\lambda} & \dfrac{b}{|\mathbf{V}|} & 0 \\ \dfrac{-ac}{\lambda|\mathbf{V}|} & \dfrac{-b}{\lambda} & \dfrac{c}{|\mathbf{V}|} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

If both b and c are zero, then $\mathbf{V} = a\mathbf{I}$, and so $\lambda = 0$. In this case, only a $\pm 90°$ rotation about the y axis is required. So if $\lambda = 0$, it follows that

$$A_{\mathbf{V}} = R_{-\theta_2,\mathbf{J}} = \begin{pmatrix} 0 & 0 & \frac{a}{|a|} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{-a}{|a|} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In the same manner we calculate the inverse transformation that aligns the vector $\mathbf{K}$ with the vector $\mathbf{V}$:

$$A_{\mathbf{V}}^{-1} = (R_{\theta_1,\mathbf{I}} \cdot R_{-\theta_2,\mathbf{J}})^{-1} = R_{-\theta_2,\mathbf{j}}^{-1} \cdot R_{\theta_1,\mathbf{I}}^{-1} = R_{\theta_2,\mathbf{j}} \cdot R_{-\theta_1,\mathbf{I}}$$

$$= \begin{pmatrix} \frac{\lambda}{|\mathbf{V}|} & \frac{-ab}{\lambda|\mathbf{V}|} & \frac{-ac}{\lambda|\mathbf{V}|} & 0 \\ 0 & \frac{c}{\lambda} & \frac{-b}{|\mathbf{V}|} & 0 \\ \frac{a}{|\mathbf{V}|} & \frac{b}{|\mathbf{V}|} & \frac{c}{|\mathbf{V}|} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

6.3 Let an axis of rotation L be specified by a direction vector $\mathbf{V}$ and a location point P. Find the transformation for a rotation of $\theta°$ about L. Refer to Fig. 6.5.

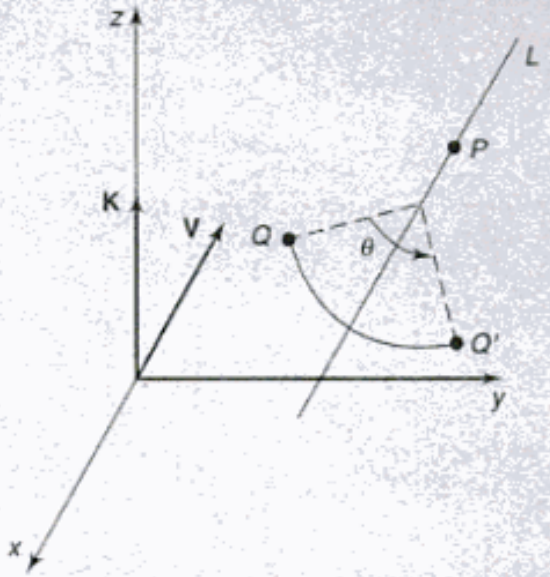


Fig. 6.5

We can find the required transformation by the following steps:

1. Translate P to the origin.
2. Align $\mathbf{V}$ with the vector $\mathbf{K}$.
3. Rotate by $\theta°$ about $\mathbf{K}$.
4. Reverse steps 2 and 1.

So

$$R_{\theta,L} = T_{-P} \cdot A_{\mathbf{V}} \cdot R_{\theta,\mathbf{K}} \cdot A_{\mathbf{V}}^{-1} \cdot T_{-P}^{-1}$$

Here, $A_{\mathbf{V}}$ is the transformation described in Problem 6.2.

6.4 The pyramid defined by the coordinates $A(0, 0, 0)$, $B(1, 0, 0)$, $C(0, 1, 0)$, and $D(0, 0, 1)$ is rotated 45° about the line L that has the direction $\mathbf{V} = \mathbf{J} + \mathbf{K}$ and passing through point $C(0, 1, 0)$ (Fig. 6.6). Find the coordinates of the rotated figure.

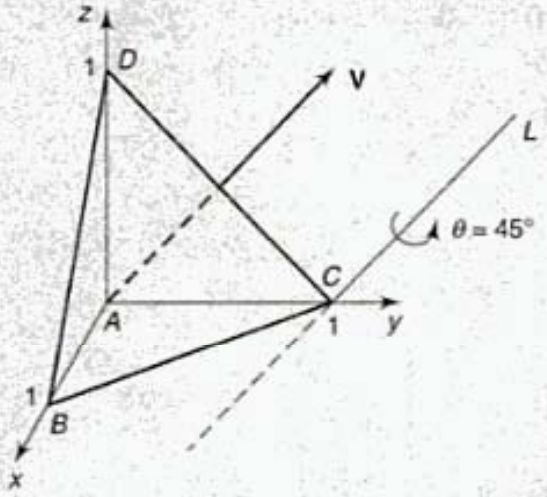


Fig. 6.6

From Solved Problem 6.3, the rotation matrix $R_{\theta,L}$ can be found by concatenating the matrices

$$R_{\theta,L} = T_{-P} \cdot A_{\mathbf{V}} \cdot R_{\theta,\mathbf{K}} \cdot A_{\mathbf{V}}^{-1} \cdot T_{-P}^{-1}$$

With $P = (0, 1, 0)$, then

$$\mathrm{T}_{-P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix}$$

Now $\mathbf{V} = \mathbf{J} + \mathbf{K}$. So from Problem 6.2, with $a = 0$, $b = 1$, $c = 1$, we find $\lambda = \sqrt{2}$, $|\mathbf{V}| = \sqrt{2}$, and

Also
$$A_V = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad A_V^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_{45°,K} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad T_{-P}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

Then
$$R_{\theta,L} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{1}{2} & \frac{-1}{2} & 0 \\ \frac{-1}{2} & \frac{2+\sqrt{2}}{4} & \frac{2-\sqrt{2}}{4} & 0 \\ \frac{1}{2} & \frac{2-\sqrt{2}}{4} & \frac{2+\sqrt{2}}{4} & 0 \\ \frac{1}{2} & \frac{2-\sqrt{2}}{4} & \frac{\sqrt{2}-2}{4} & 1 \end{pmatrix}$$

To find the coordinates of the rotated figure, we apply the rotation matrix $R_{\theta,L}$ to the matrix of homogeneous coordinates of the vertices A, B, C, and D:

$$C = \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

So
$$C \cdot R_{\theta,L} = \begin{pmatrix} \frac{1}{2} & \frac{2-\sqrt{2}}{4} & \frac{\sqrt{2}-2}{4} & 1 \\ \frac{1+\sqrt{2}}{2} & \frac{4-\sqrt{2}}{4} & \frac{\sqrt{2}-4}{4} & 1 \\ 0 & -1 & 0 & 1 \\ 1 & \frac{2-\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 1 \end{pmatrix}$$

The rotated coordinates are (Fig. 6.7).

$$A' = \left(\frac{1}{2}, \frac{2-\sqrt{2}}{4}, \frac{\sqrt{2}-2}{4}\right)$$

$$B' = \left(\frac{1+\sqrt{2}}{2}, \frac{4-\sqrt{2}}{4}, \frac{\sqrt{2}-4}{4}\right)$$

$$C' = (0, 0, 1)$$

$$D' = \left(1, \frac{2-\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)$$

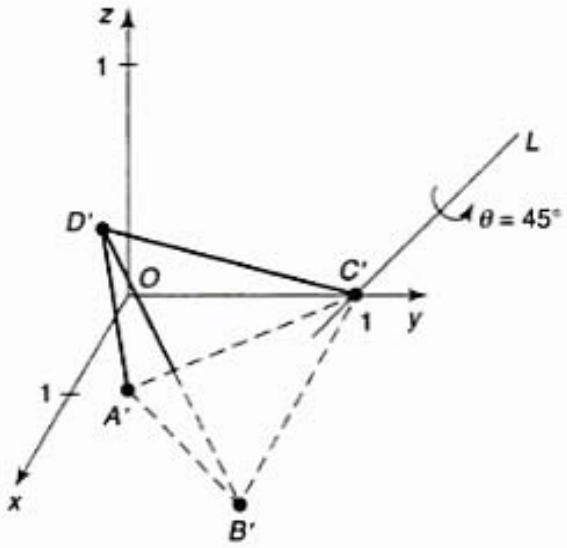


Fig. 6.7

6.5 Find a transformation $A_{\mathbf{V},\mathbf{N}}$ which aligns a vector **V** with a vector **N**.

We form the transformation in two steps. First, align **V** with vector **K**, and second, align vector **K** with vector **N**. So from Problem 6.2,

$$A_{\mathbf{V},\mathbf{N}} = A_{\mathbf{V}} \cdot A_{\mathbf{N}}^{-1}$$

Referring to Solved Problem 6.12, we could also get $A_{\mathbf{V},\mathbf{N}}$ by rotating **V** towards **N** about the axis $\mathbf{V} \times \mathbf{N}$.

6.6 Find the transformation for mirror reflection with respect to the *xy* plane.

From Fig. 6.8, it is easy to see that the reflection of $P(x, y, z)$ is $P'(x, y, z)$. The transformation that performs this reflection is

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

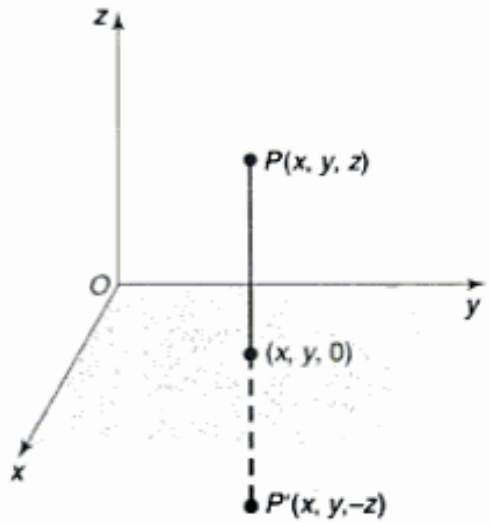


Fig. 6.8

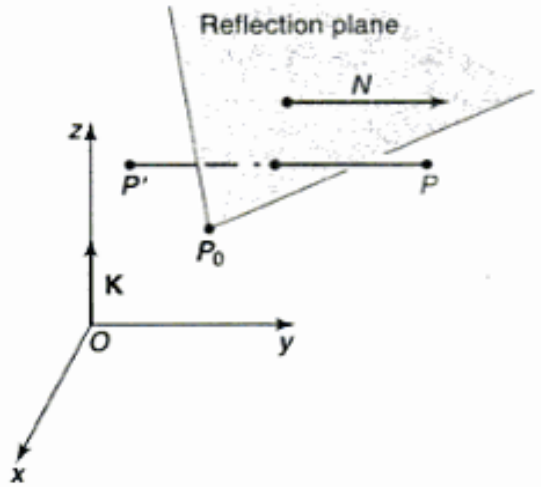


Fig. 6.9

6.7 Find the transformation for mirror reflection with respect to a given plane. Refer to Fig. 6.9.

Let the plane of reflection be specified by a normal vector **N** and a reference point $P_0(x_0, y_0, z_0)$. To reduce the reflection to a mirror reflection with respect to the xy plane:

1. Translate P_0 to the origin.
2. Align the normal vector **N** with the vector **K** normal to the xy plane.
3. Perform the mirror reflection in the xy plane (Problem 6.6).
4. Reverse steps 1 and 2.

So, with translation vector $\mathbf{V} = -x_0\mathbf{I} - y_0\mathbf{J} - z_0\mathbf{K}$

$$M_{N,P_0} = T_{\mathbf{V}} \cdot A_{\mathbf{N}} \cdot M \cdot A_{\mathbf{N}}^{-1} \cdot T_{\mathbf{V}}^{-1}$$

Here, $A_{\mathbf{N}}$ is the alignment matrix defined in Problem 6.2. So if the vector $\mathbf{N} = n_1\mathbf{I} + n_2\mathbf{J} + n_3\mathbf{K}$, then from Solved Problem 6.2, with $|\mathbf{N}| = \sqrt{n_1^2 + n_2^2 + n_3^2}$ and $\lambda = \sqrt{n_2^2 + n_3^2}$, we find

$$A_{\mathbf{N}} = \begin{pmatrix} \frac{\lambda}{|\mathbf{N}|} & 0 & \frac{n_1}{|\mathbf{N}|} & 0 \\ \frac{-n_1 n_2}{\lambda|\mathbf{N}|} & \frac{n_3}{\lambda} & \frac{n_2}{|\mathbf{N}|} & 0 \\ \frac{-n_1 n_3}{\lambda|\mathbf{N}|} & \frac{-n_2}{\lambda} & \frac{n_3}{|\mathbf{N}|} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad A_{\mathbf{N}}^{-1} = \begin{pmatrix} \frac{\lambda}{|\mathbf{N}|} & \frac{-n_1 n_2}{\lambda|\mathbf{N}|} & \frac{-n_1 n_3}{\lambda|\mathbf{N}|} & 0 \\ 0 & \frac{n_3}{\lambda} & \frac{-n_2}{\lambda} & 0 \\ \frac{n_1}{|\mathbf{N}|} & \frac{n_2}{|\mathbf{N}|} & \frac{n_3}{|\mathbf{N}|} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In addition

$$T_{\mathbf{V}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -x_0 & -y_0 & -z_0 & 1 \end{pmatrix} \quad \text{and} \quad T_{\mathbf{V}}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_0 & y_0 & z_0 & 1 \end{pmatrix}$$

Finally, from Problem 6.6, the homogeneous form of M is

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

6.8 Find the matrix for mirror reflection with respect to the plane passing through the origin and having a normal vector whose direction is $\mathbf{N} = \mathbf{I} + \mathbf{J} + \mathbf{K}$.

From Problem 6.7, with $P_0(0, 0, 0)$ and $\mathbf{N} = \mathbf{I} + \mathbf{J} + \mathbf{K}$, we find $|\mathbf{N}| = \sqrt{3}$ and $\lambda = \sqrt{2}$. Then

$$T_{\mathbf{V}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (\mathbf{V} = 0\mathbf{I} + 0\mathbf{J} + 0\mathbf{K}) \qquad T_{\mathbf{V}}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_{\mathbf{N}} = \begin{pmatrix} \frac{\sqrt{2}}{\sqrt{3}} & 0 & \frac{1}{\sqrt{3}} & 0 \\ \frac{-1}{\sqrt{2}\sqrt{3}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} & 0 \\ \frac{-1}{\sqrt{2}\sqrt{3}} & \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{3}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad A_{\mathbf{N}}^{-1} = \begin{pmatrix} \frac{\sqrt{2}}{\sqrt{3}} & \frac{-1}{\sqrt{2}\sqrt{3}} & \frac{-1}{\sqrt{2}\sqrt{3}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{-1}{\sqrt{3}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The reflection matrix is

$$M_{\mathbf{N},O} = T_{\mathbf{V}} \cdot A_{\mathbf{N}} \cdot M \cdot A_{\mathbf{N}}^{-1} \cdot T_{\mathbf{V}}^{-1}$$

$$= \begin{pmatrix} \frac{1}{3} & -\frac{2}{3} & -\frac{2}{3} & 0 \\ -\frac{2}{3} & \frac{1}{3} & -\frac{2}{3} & 0 \\ -\frac{2}{3} & -\frac{2}{3} & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

SUPPLEMENTARY PROBLEMS

6.1 Align the vector $\mathbf{V} = \mathbf{I} + \mathbf{J} + \mathbf{K}$ with the vector $\mathbf{K}$.

6.2 Find a transformation which aligns the vector $\mathbf{V} = \mathbf{I} + \mathbf{J} + \mathbf{K}$ with the vector $\mathbf{N} = 2\mathbf{I} - \mathbf{J} - \mathbf{K}$.

The McGraw-Hill Companies

SCHAUM'S ouTlines

The High Performance Study Guide

Schaum's helps you

- Hone problem-solving skills
- Find answers fast
- Cut study time
- Brush up before tests
- Achieve high performance

Related titles in Schaum's Outlines

Computing

- Programming with C++, 2e
- Programming with C, 2e
- Data Structures
- Data Structures with C++
- Data Structures with Java
- Operating Systems
- Computer Architecture
- Computer Networking
- Fundamentals of Relational Databases
- Discrete Mathematics, 2e
- Fundamentals of SQL Programming
- XML

Visit us at : www.tatamcgrawhill.com

ISBN-13:978-0-07-060165-9
ISBN-10: 0-07-060165-8

9 780070 601659

Tata McGraw-Hill