Preface

The field of computer graphics has experienced a number of hardware and software revolutions since the early 1970's when these notes began. The intentions of these notes now, however, are very much the same as then, to provide students with a basic knowledge of, and practical experience with, the fundamental mathematics, algorithms and representations that are needed to develop interactive computer graphics applications.

The chapters are arranged in an order that progresses from hardware to software, from two dimensions (2D) to three dimensions (3D), and from simple geometric forms to the more complex curves, surfaces and solids. At times, this causes some topics to be re-visited, such as clipping and transformations in 2D and 3D. This order was chosen to facilitate what the author considers to be a more natural topical progression for learning. Students accumulate knowledge and experience with somewhat simpler topics first, such as 2D programming techniques, and then are ready to combine these with the later topics that are more mathematically and computationally challenging.

Computer graphics and computer programming are inseparable. Typically in the course that uses these notes, four programming projects are assigned over sixteen weeks, each project requiring three to four weeks, coinciding as much as possible with the lecture topics. The goals of the projects are to give the student in-depth practice with the concepts while designing, coding and debugging a relevant and non-trivial application. Typically, the project topics are (1) basic 2D display with minor interaction, (2) fully interactive 2D drawing requiring dynamic data structures and color, (3) the wireframe 3D pipeline with projection, 3D clipping, and hierarchical transformations with little interaction, (4) 3D hidden surface removal and shading, such as scan-line rendering or ray tracing. The code for the first project is usually given to the students to demonstrate basic graphical programming techniques. This has proven to be effective in speeding the learning of programming details as well as various operating system and graphics package minutia, paving the way for more concentration on mathematical and algorithmic concepts.

Over the past two decades, there have been four generations of graphics devices used for the projects, starting with the Imlac™ minicomputer vector display system, progressing through Megatek™ 3D vector systems, and now raster graphics workstations and personal computers. Each new display technology spawned a new generation of graphics programming package. Such is the case for the GRAFIC package, the current version of which was developed during the first days of X Windows™ (version 10) to support the teaching of computer graphics. GRAFIC was designed to provide basic, multi-language (initially C, Pascal and Fortran; and now C++ as well), multi-platform access to window-based raster graphics. An important objective during the development of GRAFIC was that its use should require little training in the package itself. Now, versions of GRAFIC exist for X Windows™ for Unix™ workstations, IBM-compatible personal computers with Microsoft Windows™, and Apple Macintosh™ computers. GRAFIC is available as-is and free of charge from the author.

There are many individuals who have contributed to these notes over the years. Michael Bailey taught the course *Interactive Computer Graphics* with the author for several years, established some of the early topics, and developed some of the early versions of the written notes. Warren Waggenspack contributed significantly to the presentation of curves and surfaces while at Purdue, and has been a valued collaborator who has encouraged continued work on the notes by the author. Philip Cunningham helped with the presentation of the reflectance and shading models. Joseph Cychosz contributed to the presentation on color and some of the ray casting algorithms. Gregory Allgood implemented the first versions of X Windows™ GRAFIC. To these individuals, the author extends sincere appreciation for their invaluable help.

David C. Anderson
Purdue University
internet: http://www.cadlab.ecn.purdue.edu/~dave/

Bibliography

The following is a partial list of texts covering various topics in computer graphics. They are categorized by their primary emphasis.

General Computer Graphics Texts:

1. Berger, M., Computer Graphics with Pascal, The Benjamin/Cummings Publishing Company, Inc., 1986.
2. Dewey, B. R., Computer Graphics for Engineers, Harper & Row, 1988.
3. Foley, J.D. and Van Dam, A., Fundamentals of Interactive Computer Graphics, Addison-Wesley, 1982.
4. Giloi, W.K., Interactive Computer Graphics, Prentice-Hall, Inc, 1978.
5. Harrington, S., Computer Graphics - A Programming Approach, McGraw-Hill Book Company, 1987.
6. Hearn, D., Baker, M. P., Computer Graphics, Prentice-Hall, Inc., 1986.
7. Hill, F. S. Jr., Computer Graphics, Macmillan Publishing Company, 1990.
8. Newman, W.M. and Sproull, R.F., Principles of Interactive Computer Graphics, Second Edition, McGraw-Hill, 1979.
9. Plastok, R. A., Kalley, G., Theory and Problems of Computer Graphics, Shaum's Outline Series in Computers, McGraw-Hill Book Company, 1986.
10. Pokorny, C. K., Curtis, F. G., Computer Graphics: The Principles Behind the Art and Science, Franklin, Beedle & Associates, 1989.
11. Rogers, D.F., Procedural Elements for Computer Graphics, McGraw-Hill Book Company, New York, 1985.

Geometric Modeling Emphasis:

12. Barnsley, M. F., Devaney, R. L., Mandelbrot, B. B., Peitgen, H.-O., Saupe, D., and Voss, R. F., The Science of Fractal Images, Springer-Verlag, 1988.
13. Ding, Q., Davies, B. J., Surface Engineering Geometry for Computer-Aided Design and Manufacture, John Wiley & Sons, 1987.
14. Farin, G., Curves and Surfaces for Computer Aided Geometric Design - A Practical Guide, Academic Press, Inc., Harcourt Brace Jovanovich Publishers, 1988.

15. Hoffmann, C. M., Geometric and Solid Modeling - An Introduction, Morgan Kaufmann Publishers, Inc. San Mateo, California, 1989.

16. Lyche, T., Schumaker, L. L., Mathematical Methods in Computer Aided Geometric Design, Academic Press, Inc., Harcourt Brace Jovanovich Publishers, 1989.

17. Mantyla, M., An Introduction to Solid Modeling, Computer Science Press, Inc., 1988.

18. Mortenson, M. E., Geometric Modeling, John Wiley & Sons, New York, 1980.

19. Mortenson, M. E., Computer Graphics - An Introduction to the Mathematics and Geometry, Industrial Press, Inc., 1989.

20. Rogers, D.F and Adams, J.A., Mathematical Elements for Computer Graphics, McGraw-Hill, 1976.

21. Yamaguchi, F., Curves and Surfaces in Computer Aided Geometric Design, Springer-Verlag, 1988.

Realistic Image Generation Emphasis:

22. Deken, Joseph, Computer Images, Stewart, Tabori, & Chang, New York, 1983.

23. Glassner, A. S. (ed.), An Introduction to Ray Tracing, Academic Press, Inc., Harcourt Brace Jovanovich Publishers, 1990.

24. Greenberg, D., Marcus, A., Schmidt, A., Gorter, V., The Computer Image: Applications of Computer Graphics, Addison-Wesley, 1982.

25. Hall, R., Illumination and Color in Computer Generated Imagery, Springer-Verlag, 1989.

26. Mandelbrot, Benoit, The Fractal Geometry of Nature, W.H. Freeman and Company, New York, 1983.

27. Overheim, R.D. and Wagner, D.L., Light and Color, John Wiley & Sons, 1982.

28. Schachter, B., Computer Image Generation, John Wiley & Sons, New York, 1983.

Contents

Chapter 1. Introduction

1.1 Historical Perspective

Computer graphics began in the late 1950's, developed primarily in the large aircraft and automobile industries. These were the only industries that could afford such expensive technology and the only ones whose massive operation could justify the equipment and development costs and effort.

In the early 1960's, Dr. Ivan Sutherland introduced the age of interactive computer graphics for engineering applications with his Ph.D. dissertation: "SKETCHPAD: A Man-Machine Graphical Communication System." His work demonstrated various ways that interactive graphics could revolutionize engineering and other application areas.

During the 1970's, advances in computer hardware caused the cost of computing to decline dramatically, making computer graphics applications more cost effective. In particular, computer-aided design (CAD) systems flourished in the 1970's with storage tube terminals and minicomputers, causing a dramatic increase in computer graphics users and applications.

The 1980's became the age of the workstation, which brought together computing, networking and interactive raster graphics technology. Computer graphics technology followed the way of computers, rapidly changing from add-on graphics devices to integral raster graphics workstations. Raster graphics systems and window-based systems became the standard.

In the mid-1980's, personal computers consumed the computing and graphics markets and quickly dominated the attention of nearly all graphics applications. The pervasiveness of the PC's made software developers flock to them. By the end of the 1980's, personal computer software was arguably the center of attention of the software developer community. Workstations also thrived as higher-end computing systems with strengths in networking and high-end computation. PC's, however, vastly outnumbered any other form of computer and became the mainstream graphics delivery system.

Today, computer graphics equipment and software sales are a billions of dollars per year marketplace that undergoes rapid performance and capability improvements annually. Some of today's most popular applications are:

- Mechanical design and drafting ("CAD/CAM")

- Electronic design and circuit layout
- Natural resource exploration and production
- Simulation
- Chemical and molecular analysis
- Entertainment, such as animation, advertising, and communications
- Desktop publishing

1.2 Notes Overview

The objective for these notes is to study the principles of computer graphics from the points of view of both a developer and a user. Emphasis will be on understanding computer graphics devices, mathematics, algorithms, representations, methods, and software engineering for designing and implementing interactive computer graphics applications in two and three dimensions. Computer graphics is not an isolated field. Computer graphics algorithms lie at the foundation of many applications in science and engineering and the elements of computer graphics provide basic knowledge useful in other fields.

We will begin by studying several categories of computer graphics devices, beginning with static graphics without interactive capabilities, and ending with dynamic graphics with motion simulation capabilities. In each general category, we will discuss how it operates, typical configurations with a computer, fundamental graphics algorithms needed to produce images, and applications.

After gaining an understanding of basic graphics devices and capabilities, the notes move into higher-level topics that are more mathematical and algorithmic in nature, beginning with three-dimensional rendering. The last chapters address geometric modeling topics related to both computer graphics and computer-aided design.

1.3 Notation

Throughout the text, the following notation will be used.

a, b, c	scalar numbers (lower case plain text)

P, Q_i	position vectors (upper case plain text, letters P or Q, possibly subscripted)
V, Vi	direction vectors (upper case plain text, letter V, possibly subscripted)
M, $\mathbf{M}_i$	matrix (upper case bold)
[B]	tensor or matrix made from elements of B
θ, ϕ	angles (lower case Greek)
[x y z w]	
[x,y,z,w]	row or column vector with explicit coordinates (with or without commas)
`IF var`	Computer language statements and references in the text to computer variables and symbols.

Chapter 2. Static Graphics Hardware

The term “static graphics” describes a class of computer graphics hardware and software in which images are created but cannot be erased. Static graphics applications are not interactive. Generally, static graphics applications involve hardcopy output, such as plotting and printing.

Historically, these types of devices were among the first in computer graphics.

2.1 Plotters

Pen plotters are electromechanical devices that draw on paper and other materials. Some past and present manufacturers include companies like Calcomp, Houston Instruments, Hewlett Packard and Tektronix. Prices range from $2,000 to over $50,000 according to speed, accuracy, and local intelligence.

Local Intelligence means the plotter has capabilities for:

1. arcs and circles,
2. dashed lines,
3. multiple pens,
4. scaling, rotations, character generation, “filling” areas, etc.

Intelligence is typically accomplished with a microprocessor in the device.

As with computers, there are typical trade-offs among speed, cost, intelligence and accuracy.

2.1.1 Physical Arrangements

The *drum plotter* was one of the first computer graphics hardcopy devices (Figure 2-1). There are three independent actions: (1) raising or lowering the pen by activating or deactivating the solenoid and stepping the drum or carriage to produce relative motions between the paper and the pen along the (2) X and (3) Y directions. Commands sent from a computer connected to the plotter raise or lower the pen, and then *step* the pen the required number of steps in X and Y to produce the desired effect.

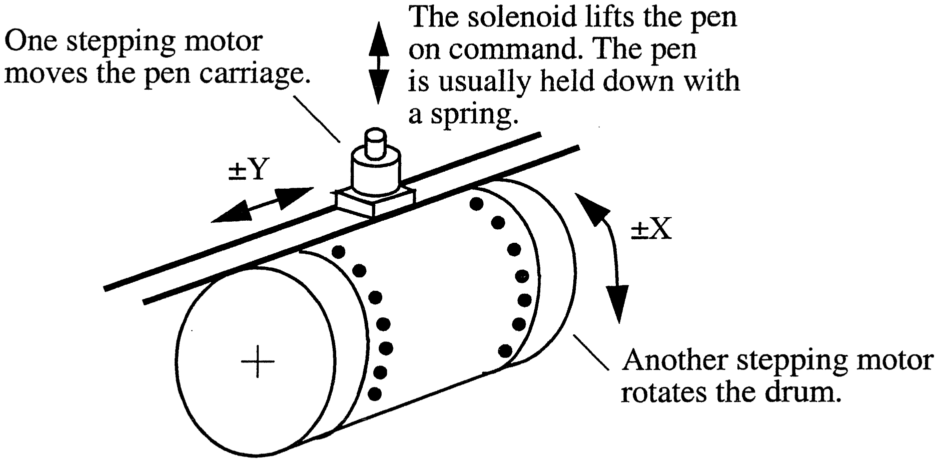

One stepping motor moves the pen carriage.
The solenoid lifts the pen on command. The pen is usually held down with a spring.
±Y
±X
Another stepping motor rotates the drum.

Figure 2-1. Elements of a Drum Plotter.

A variation of the drum plotter that facilitates drawing on thicker materials and materials that cannot bend is the *flatbed plotter* (Figure 2-2). The basic operation of the pen carriage and two stepping motors is like the drum plotter.

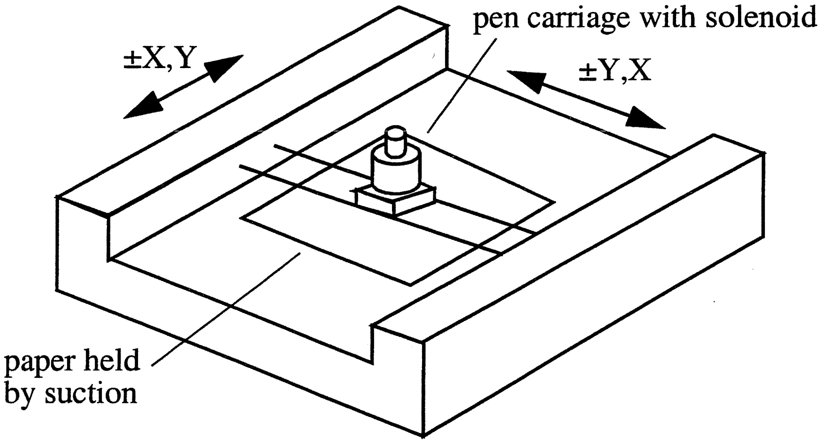


Figure 2-2. Elements of a Flatbed Plotter.

Plotters are capable of drawing lines of different colors using multiple pen carriages (Figure 2-3). The pen carriage is rotated or translated to place the desired pen in the writing position. A more elaborate version of a multiple pen carriage changes pens by moving the pen carriage to the corner of the pen bed, depositing the current pen in a second pen carriage that holds extra pens, picking up a new pen from the extra carriage, and returning to the writing position to continue drawing.

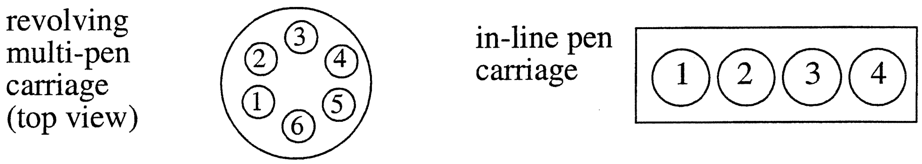


Figure 2-3. Multiple Pen Carriages.

2.1.2 Basic Operation

The pen is held in a chamber and is pressed against the paper surface by a spring. The chamber is held inside a solenoid coil. When activated by the plotter electronics, the solenoid lifts the pen above the plotting surface. X and Y motions are controlled by *stepping motors*. The solenoid and motors are controlled by a computer. Only one step is taken at a time.

The *resolution* of a plotter is a measure of its accuracy, or *step-size*. Typical resolutions are 0.002 and 0.005 inch per step, or, as it is sometimes given, 500 or 200 steps per inch.

There are several critical factors in the design of plotters:

speed — The speed of a plotter is typically given in "steps per second." The time required to complete a given drawing is a function of the distance travelled by the pen, and the resolution and speed of the plotter. For example, consider drawing an 8" by 10" box around a page using a plotter with 500 steps per inch and a speed of 500 steps per second. The time to complete the drawing is computed as follows:

time (sec) = distance (inch) / { resolution (inch/step) * speed (step/sec) }

or,

time = (8+8+10+10) / { 0.002 * 500 }

time = 36 sec

(Note the plotter travels at 1 inch/second.)

accuracy — Generally, increasing accuracy requires slower speeds and more massive structures to maintain mechanical rigidity.

ink flow — High speed plotters must force ink from the pen because gravity alone is insufficient.

2.1.3 Configurations

There are three basic configurations for connecting a plotter to a computer: off-line, on-line, and spooled (buffered). *Off-line* configuration means the plotter is not physically connected to the computer, so data must be manually transferred. The plotter reads plotting information from a tape that must be loaded and unloaded periodically. The plotting programs write this information on the tape and, at a designated time, the tape is removed and queued for the plotter.

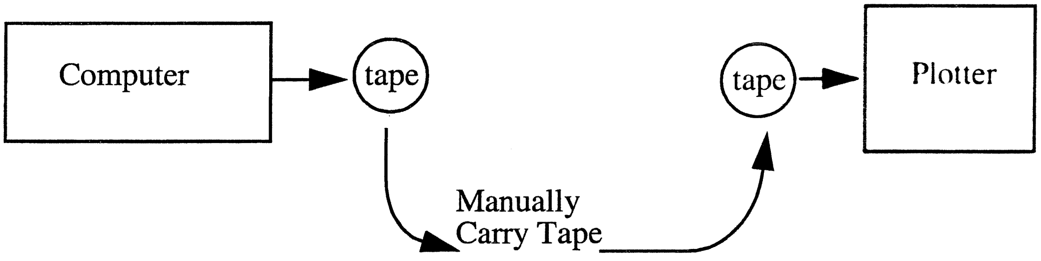


Figure 2-4. Off-line Plotter Configuration.

On-line configuration means there is a direct data communication interface between the computer and the plotter. As the plotting program executes in the computer, calls to plotting routines cause "plot commands" to be sent to the plotter. The plot commands are executed by the plotter hardware, i.e. the pen is moved. This is a typical configuration for a personal computer and a plotter.

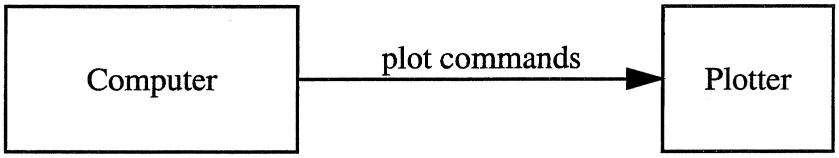


Figure 2-5. On-line Plotter Configuration.

It usually does not make sense to configure a plotter on-line with a multi-user, time-shared computer system because several plotting programs can be executing at once. In this case, it is necessary to couple the two previous approaches, on-line and off-line, in a manner that allows several plotting programs to create plot data, yet send only one program's data at a time to the plotter. *Spooling* is a method for accomplishing this.

2.1.4 Spooling

Current time-shared systems perform a combination of off-line and on-line plotting, called *spooling*. This is similar to how line printers are operated. Storing and plotting can occur simultaneously (with proper provisions for simultaneous access to files). Plotting programs create intermediate "plot files" when they execute. When a program finishes, the plot file is queued for plotting, that is, transferred onto a disk area set aside for plot files waiting to be plotted. This intermediate disk storage is a *buffer* between the programs creating plot data and the slower plotter, which can only process one plot at a time. Another computer program, sometimes executing in another computer, accesses the queued plot files periodically and sends them to the plotter. The plotting programs are off-line with respect to the plotter, and the intermediate program is on-line.

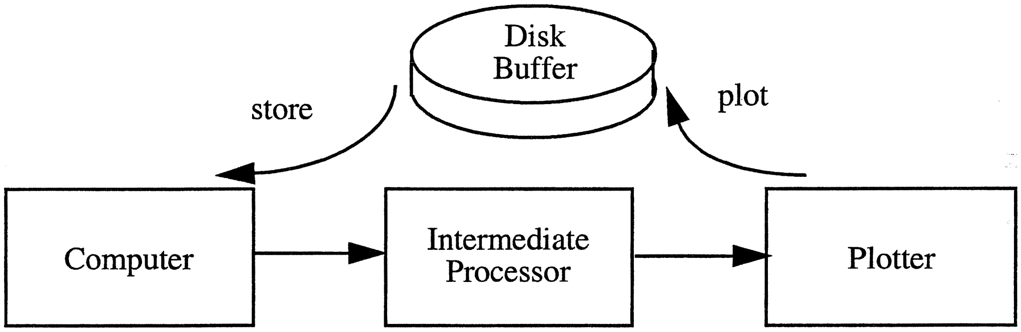


Figure 2-6. Spooling Configuration.

2.2 Electrostatic Plotters

Pen plotters are relatively slow devices. Electrostatic plotting was developed as a compromise between speed and accuracy. Electrostatic plotting is a process that, in effect, charges small dots on a piece of paper and then immerses the paper with dry powdered ink. The ink adheres to charged dots on the paper and falls off the rest (Figure 2-7).

One advantage of electrostatic plotting is that the printing process is almost entirely electronic, so the problems of speed and accuracy that plague mechanical pen plotters are eliminated. The drawback is that the resulting image must be drawn as an array of dots.

Each dot is called an *electrostatic plotter point* or an *epp*. The process involves the computer sending data to the plotter for each line of epps. The data for a line of epps is a series of

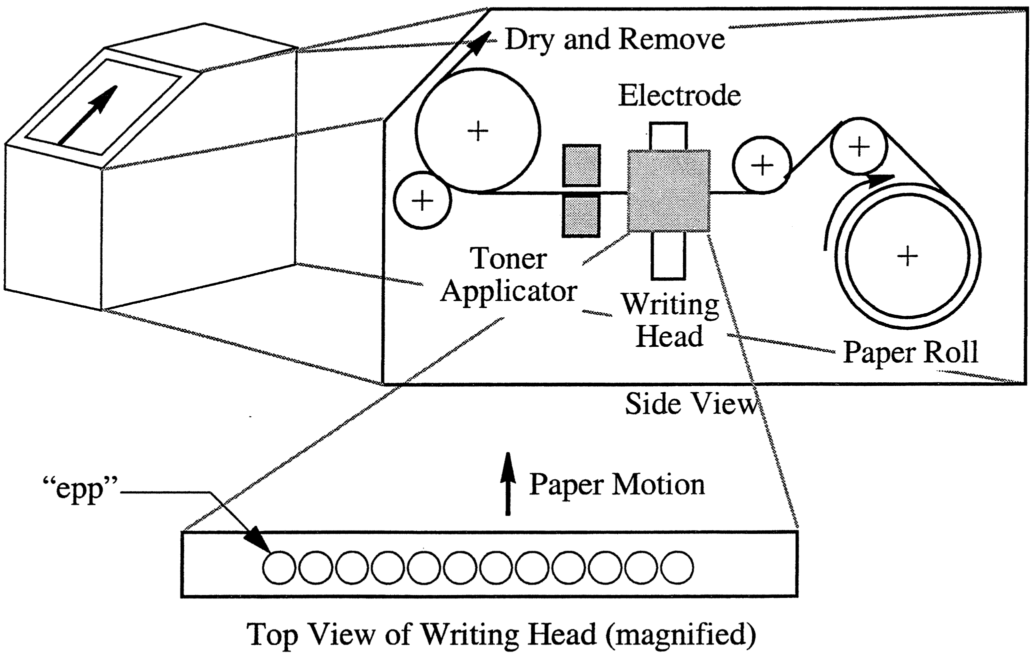


Figure 2-7. Elements of an Electrostatic Plotter.

binary numbers, 1or 0, representing "dot" (on) or "no dot" (off) for each writing head epp position. The computer must compute these epps based on the image that is to be drawn. After a complete line of epps is received, the plotter controls the writing head to charge the paper at epps whose values are "on." As the paper advances, new lines are charged and the charged lines reach the toner applicator where the ink is applied. The process continues down the entire page.

The density of epps, or resolution, is critical to the quality of the image. Typical electrostatic plotters have resolutions of 100 to 250 epps per inch. One hardware modification to increase resolution is to arrange the writing head in two rows of epps, one for even epps and another for odd. A single line of epps is printed on the paper in two steps that are synchronized in time so that the even and odd epps align along a line.

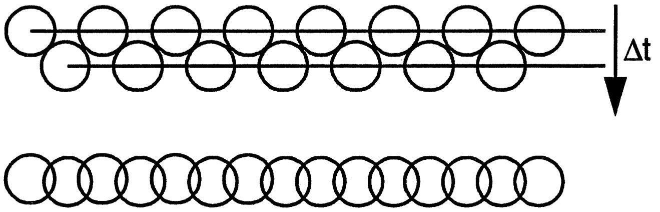


Figure 2-8. Staggered Epp Writing Head Operation.

Some past and present electrostatic plotter manufacturers are Versatec, Houston

Instruments, Varian, and Gould. Prices range from $5,000 to $50,000, widths vary from 8 inches to 6 feet, speeds vary between 1 to over 6 inch/sec., and resolutions vary from 80 to 250 epps per inch. Color printing can be done with multiple passes over the same paper, once for each color component. This will be described later in the chapter on color.

Electrostatic printers have been replaced by the more popular laser printers.

2.3 Laser Printers

Laser printers operate like electrostatic printers, except the printing "engine" is a laser beam instead of a writing head. The computer sends digital information (commands) to the controller, which carefully controls the intensity and deflection of the laser beam through a rotating polygon mirror. The mirror deflects the beam to the proper location (dot) on the photoreceptor drum. One line of dots along the drum is written at a time. The drum rotates, imparting the charge onto the paper passing under (over) it. The toner then adheres to the charged locations on the paper. Laser printers are capable of much higher resolutions than electrostatic plotters, from 300 to thousands of *dots per inch* (*dpi*).

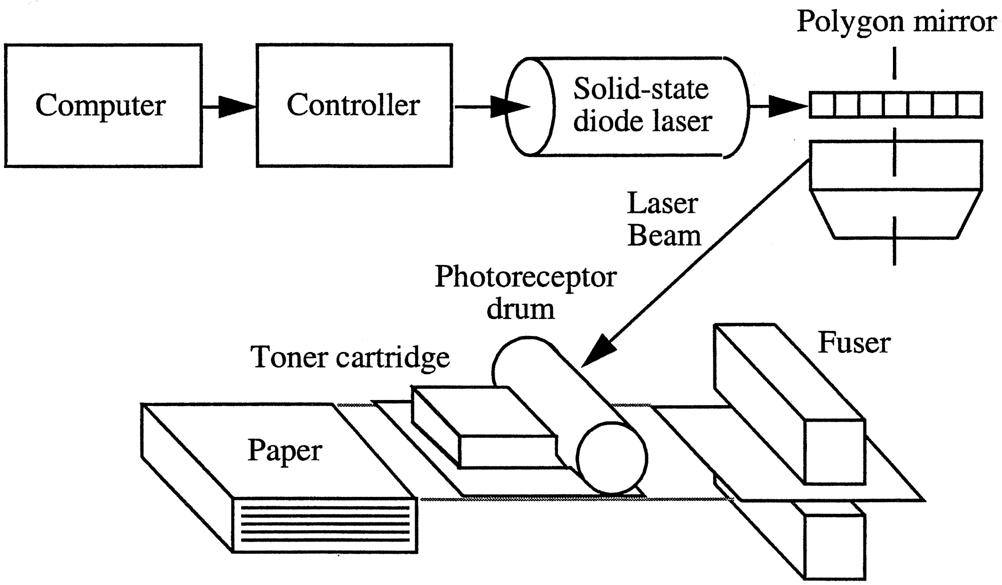


Figure 2-9. Elements of Laser Printers.

Some laser printer engine manufacturers are Canon, Ricoh, Kyocera, and Linotype. Prices range from $1,000 to over $50,000 and speeds vary from a few to 20 or more pages per minute.

2.4 Film Recorders

These devices write on film. Some act as a stand-alone output device. The image is drawn by drawing three monochrome (one color) images, each through the appropriate filter.

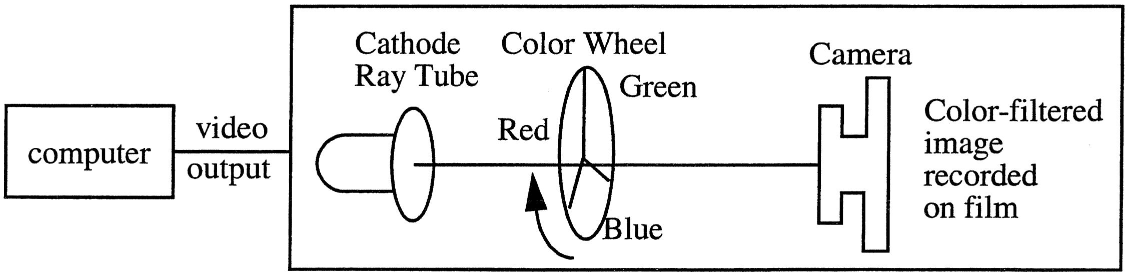


Figure 2-10. Elements of a Film Recorder.

Another variety of film recorder is used in conjunction with a raster display. The process is similar to the film recorder above, however the image is not computed for the recorder, but is instead captured from a screen display.

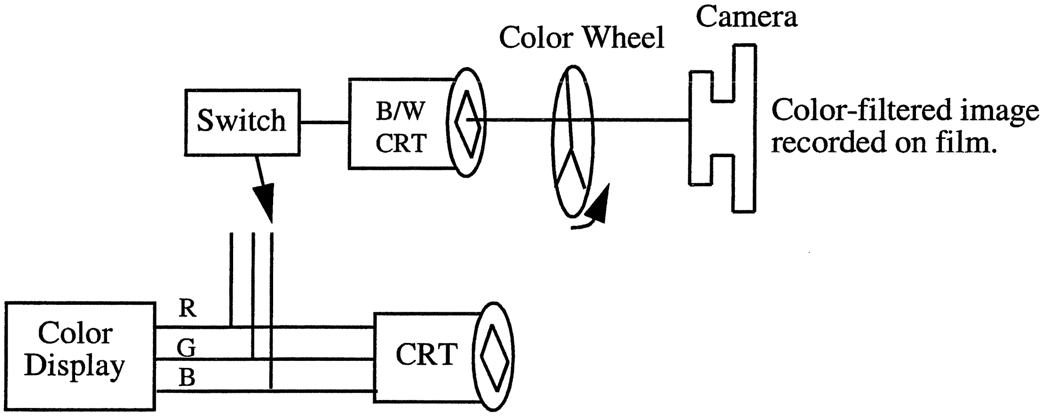


Figure 2-11. Elements of a CRT Film Hardcopy Recorder.

Chapter 3. Static Graphics Software

3.1 Hardcopy Plotting

We will begin by looking first at "generic hardcopy plotting." Early computer systems used batch computing, where each job ran without interaction, producing listing output and storing files for later examination. Plotting was done the same way. A program produced a "plot data file" and then directed the system to create the plot from this file. One can think of today's laser printing as the same process.

One of the first *plotting packages*, a library of functions that can be used to create plot data, was the Calcomp package developed in the 1960's (and versions are still in use today). Although many graphics packages have been developed over the years for increasingly more powerful graphics systems, all contain fundamental elements that are illustrated in the Calcomp package: initialization, drawing lines and symbols, and termination.

A typical Fortran program that calls Calcomp routines appears as shown below.

```
call plots
... other plotting calls ...
call plot( 0.0, 0.0, 999 )        2 draw
stop         X    Y    FLAG /      3 move
end                               999 stop
```

The routine `plots` initializes the plotting package internal variables and must be called before any other plotting routines. The routine `plot` with the given arguments terminates the plotting and must be the last routine called. In an off-line or spooled plotter configuration, `plots` initializes the intermediate plot data file. In an on-line plotter configuration, `plots` may setup the plotter interface and move the pen to the corner of the paper, or it may do nothing at all.

There are two basic drawing routines, one for lines and one for symbols (characters).

`plot( x, y, ipen )`

`Plot` moves the pen to the location (`x,y`) inches with respect to the origin, originally the lower left of the plotting page. If the value of the *pen code* `ipen` is 2, a line is drawn. If the value of `ipen` is 3, the pen is just moved, no line is drawn. Additionally, a negative pen code (-2 or -3) translates the origin to the destination coordinates. Changing the plotting origin is convenient, for example, when making several side by side pages of graphs (in the x direction). After completing the first page, the statement "`call plot( 11.0, 0.0, -3 )`" positions the pen and the origin 11 inches to the right of the current origin, a convenient location for the next page. In general, changing the origin facilitates drawing objects whose coordinates are not conveniently expressed with respect to the lower left corner of the page. We will use the symbols `DRAW` for a pen code of 2, `MOVE` for 3, and `ENDPLOT` for a pen code of 999.

`symbol( x, y, height, string, angle, ns )`

`Symbol` draws a string of characters, a series of characters along a line. The coordinates of the lower-left corner of the first character in `string` will be (`x,y`). The string's height in inches will be `height`. `string` is a character string, either a "quoted list of characters," a character variable, or a character array. The string will be plotted at `angle` degrees measured counterclockwise from the East direction. The argument `ns` specifies the number of characters plotted from `string`.

In Figure 3-1, the program creates the plot shown at the right.

```
plots();
plot( 0.0,  0.0, 3 );
plot( 8.5,  0.0, 2 );
plot( 8.5, 11.0, 2 );
plot( 0.0, 11.0, 2 );
plot( 0.0,  0.0, 2 );
symbol(1.,9.,1.5,'TEST',0.,4);
plot( 0.0, 0.0, 999 );
```

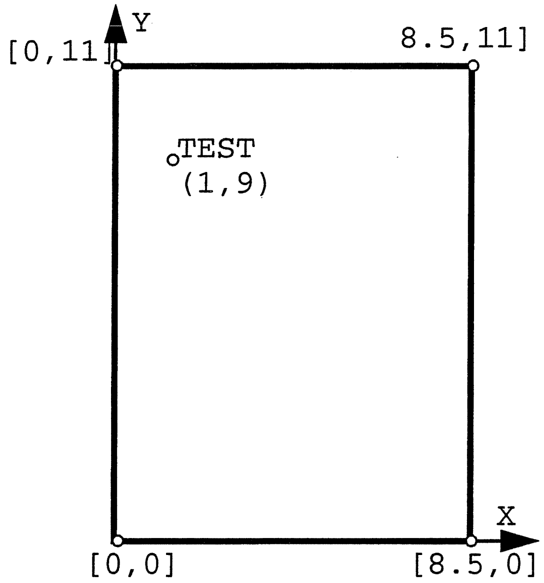
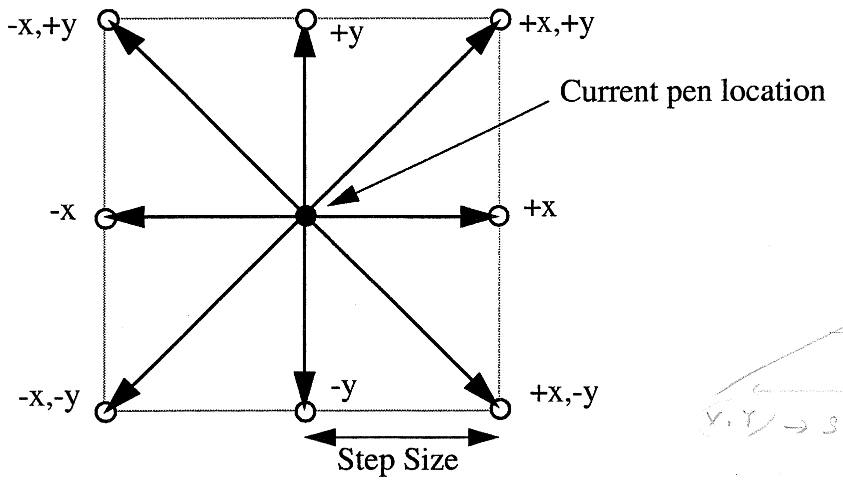


Figure 3-1. Sample Calcomp Program and Plot.

3.2 Lines

The Calcomp plotting package is an example of a basic set of static graphics routines. We are most concerned with general organization and function, not with specific names, argument lists, or other such details. We now take an inside look at the most basic of graphic functions, drawing straight lines.

A plotter can only draw straight lines corresponding to steps. Step sizes are small (typically 0.002 inch) so that fine detail can be drawn. The problem is drawing an arbitrary length straight line from a fixed set of small steps. Figure 3-2 shows a greatly magnified view from above the current pen location (black dot):



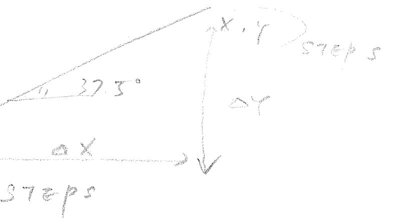


Figure 3-2. Possible Pen Steps.

There are 8 possible steps corresponding to horizontal, vertical and diagonal pen motions. Support software in the plotting package, in `plot`, must create each line as a sequence of these steps. This is accomplished by a *stepping algorithm*, the most famous of which is the Bresenham algorithm [Bresenham]. Of particular note with this algorithm is that it requires only addition and subtraction of integers, so it can be implemented in inexpensive microprocessors.

The first plotters accepted only simple binary commands that specified step directions and pen control, so line generation had to be done by the host computer software. Now, most plotters do this internally, so that only end point coordinates of the line need to be sent by the host computer.

3.2.1 The Bresenham Line Stepping Algorithm

Consider the situation shown in Figure 3-3, where the pen is at location [0,0] and a line is to be drawn whose exact equation is: u y = v x. Note that any line that does not pass through the origin can transformed into such a line using a simple change of variables.

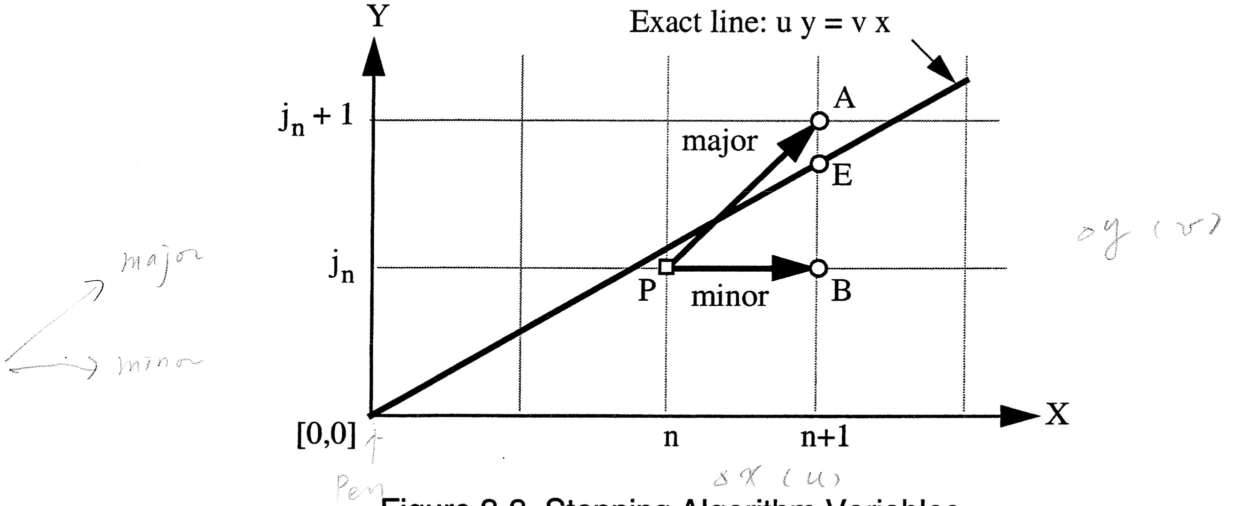


Figure 3-3. Stepping Algorithm Variables.

Assume that the pen has been moved (stepped) to the position labelled “P.” There are two possible next steps: a “major” diagonal step to point A and a “minor” horizontal step to point B. The term major indicates that the step involves both X and Y, whereas a minor step involves only one direction. The exact coordinates of the point on the line at x = n+1 is point E, whose y value is $(n + 1)\frac{v}{u}$. To decide which step, minor or major, would be closer to the exact line, first compute two positive vertical distances, AE and EB:

$$AE = (j_n + 1) - (n + 1) \frac{v}{u}$$

$$EB = (n + 1) \frac{v}{u} - j_n$$

Let the quantity d_{n+1} be the scaled difference of these two distances (assuming u > 0):

$$d_{n+1} = u \{ EB - AE \}$$

$$= u \{ [(n+1)\frac{v}{u} - j_n] - [(j_n + 1) - (n+1)\frac{v}{u}] \}$$

$$= 2v (n + 1) - u (2 j_n + 1).$$

For the previous step (substituting n-1 for n):

$$d_n = 2nv - u (2 j_{n-1} + 1).$$

Now express d_{n+1} in terms of d_n to formulate the incremental "stepping algorithm:"

$$d_{n+1} = d_n + 2 v - 2 u (j_n - j_{n-1})$$

and $d_1 = 2v - u$ (when $n = j_n = 0$).

Note that $(j_n - j_{n-1}) = 0$ if $d_n < 0$, which means take a minor step (previous step)

and $d_{n+1} = d_n + 2v$,

$= 1$ if $d_n > 0$, which means take major step

and $d_{n+1} = d_n + 2v - 2u$

For lines in the other four quadrants (and when $\Delta y > \Delta x$), it is necessary to define the proper minor and major steps according to the particular plotter and its commands, and the proper u and v quantities using absolute values. For example, given a line with a negative Δx and positive Δy, i.e. with the end point in the second quadrant, the minor step would be "-x" and the major step would be "-x+y." The quantities u and v must be computed as their first octant values.

This algorithm is illustrated in Figure 3-4.

```
major = 'proper plotter command for diagonal step';
minor = 'proper plotter command for minor step';
u = max( abs(dx), abs(dy) );
v = min( abs(dx), abs(dy) );
d = 2v - u;
for( counter = 0; counter < u; counter++ ) {
     if( d >= 0 ) {
          d = d + 2v - 2u;     /* or d += 2v - 2u; */
          PlotterCommand(major);
     } else {
          d = d + 2v;          /* or d += 2v; */
          PlotterCommand(minor);
     }
}
```

Figure 3-4. Stepping Algorithm Logic.

Note that the quantities 2v - 2u and 2v are constants, and 2v = v+v. Figure 3-5 is an example of the execution of the algorithm.

DX = 5, DY = 4, therefore: u = 5, v = 4, major = "+x+y", minor = "+x"

n	d_n	d_{n+1}	step
1	3	1	major
2	1	-1	major
3	-1	7	minor
4	7	5	major
5	5	3	major

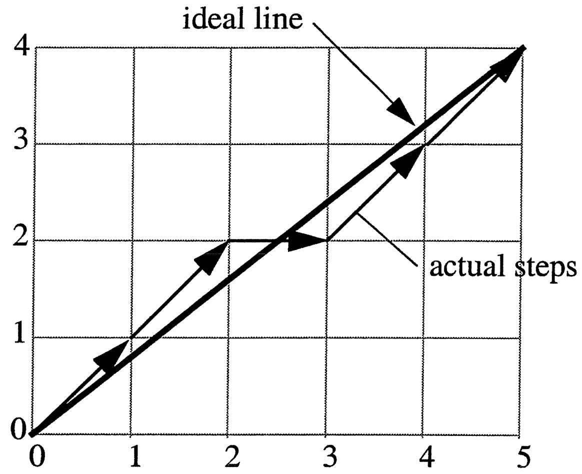


Figure 3-5. Example Steps.

Figure 3-6 illustrates an implementation of the stepping algorithm for a plotter whose step command is an integer in which each bit has the meaning shown in the figure. PlotterCommand is an operating system interface routine that sends the given integer, assumed to be a plotter command, to the plotter.

3.2.2 Plotting Lines

As examples of the internal operation of a plotting package, examine how the two basic drawing routines *could* be coded. To begin, it seems logical that a "user" would not want to use coordinates in units of plotter steps. This would be very inconvenient and makes the program specific to a particular plotter. Also, the user may want to easily change the plot scale (for example: to reduce the picture size for publication) and origin (for example: to produce a series of side by side graphs).

Assuming the existence of the stepping routine, Step, subroutine Plot could be written as shown in Figure 3-7. The variables xpen, ypen, xorg, yorg, factor, xres, yres are global variables maintained by the plotting package. Xpen and ypen are the current pen location of the plotter. Xorg and yorg store the origin and factor is the scale factor. Xres

```
void Step( int dx, int dy )
{
        int major, minor, u, v, d, count;

        u = dx;
        if( u < 0 ) {
                u = -u;                 // abs(dx)
                major = 4;              // -x
        } else
                major = 8;              // +x
        v = dy;
        if( v < 0 ) {
                v = -v;                 // abs(dy)
                major |= 1;             // OR -y bit
        } else
                major |= 2;             // OR +y bit
        if( u >= v ) {                  // major direction is x
                minor = major & 12;     // save only x bits
        } else {                        // else major direction is y
                minor = major & 3;      // save only y bits
                d = u;                  // and swap u & v
                u = v;
                v = d;
        }
        count = u;
        v += v;                 // 2v for minor d increment
        d = v - u;              // initial d is 2v - u
        u = d - u;              // 2v - 2u for major d increment
        while( --count >= 0 ) {
                if( d >= 0 ) {
                        d += u;
                        PlotterCommand( major );
                } else {
                        d += v;
                        PlotterCommand( minor );
                }
        }
}
```

plotter command:

32	16	8	4	2	1
PEN DN	PEN UP	+X	-X	+Y	-Y
0	0	1	0	0	1

example: $001001_2 = 9_{10}$ = step +X -Y

0 1 0 0 0 0 = 16 = PEN UP

Figure 3-6. An Implementation of the Stepping Algorithm.

and `yres` are the resolutions, which can be varied by the package as different plotters are selected for use. The routines `EndPlot`, `PenUp` and `PenDown` are system “interface” routines that communicate with the plotter (on-line) or enter plotter control commands to a plot file (off-line).

3.3 Characters

To draw a character requires that its graphic appearance, or *glyph*, is pre-defined in some drawable form. It is necessary to create a general definition of each character that can be scaled and positioned as needed. We establish a *definition grid*, a layout convention for defining characters.

```
float xpen, ypen, xorg, yorg, factor, xres, yres;

int Round( float x )
{
        if( x < 0 )
                return (int)(x - 0.5);
        return (int)(x + 0.5);
}
void Plot( float x, float y, int pen )
{
        int dx, dy, penabs;
        float xold, yold;

        if( pen == ENDPLOT ) {
                Endplot();
                return;
        }
        xold = xpen;
        yold = ypen;
        xpen = x * factor + xorg;
        ypen = y * factor + yorg;
        penabs = pen;
        if( pen < 0 ) {
                penabs = -pen;
                xorg = xpen;
                yorg = ypen;
        }
        if( penabs == MOVE )
                Penup();
        if( penabs == DRAW )
                Pendown();
        dx = Round( xres * (xpen - xold) );
        dy = Round( yres * (ypen - yold) );
        Step(dx, dy);
}
```

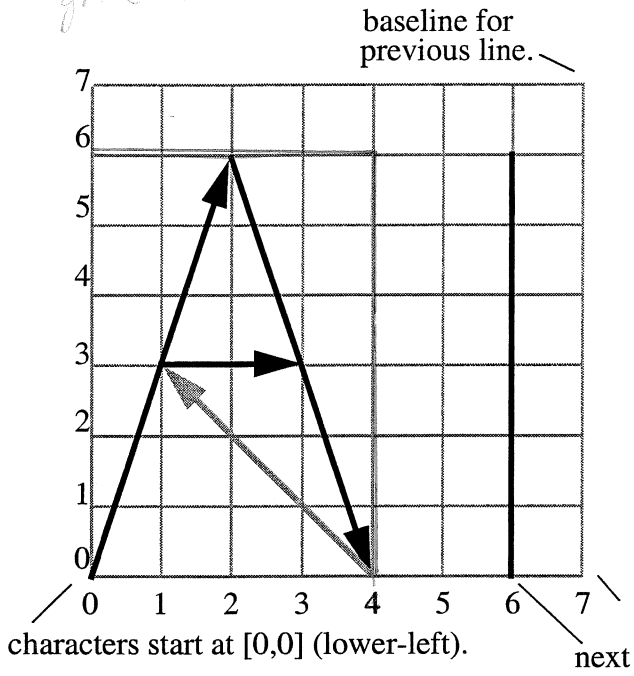
Figure 3-7. An Implementation of `Plot`.

A complete set of character definitions is called a *font.*

Look at an example, assuming a 7 by 7 definition grid to define a capital 'A' as shown in Figure 3-8. The routine to plot the symbol ‘A’ is coded in Figure 3-9. The variable ‘`f`’ is a scale factor that scales grid-units to inches.

This character is defined as *strokes* (lines). Other character definition formats exist, such as *outline fonts*, where each character is defined as a series of lines and curves that form a closed boundary.

By convention, characters are represented in computers as an ordered set of integers that correspond to graphic symbols. The standard in use today is the American Standard Code for Information Interchange, ASCII. All characters are represented in 7 bits in an 8-bit *byte* with values from 0 to 127. Only 96 of these are graphic (letters, numbers, punctuation), the others are nongrid 7x7


baseline for
previous line.
7
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7
"baseline"
characters start at [0,0] (lower-left).
next character begins here.

Data for "A"

I	AX	AY	PEN
1	0	0	3
2	2	6	2
3	4	0	2
4	1	3	3
5	3	3	2

Recall:
2 is DRAW,
3 is MOVE

Figure 3-8. Definition of Character "A."

in "grid units"

```
void A( float x, float y, float ht )
{
    static int ax[5] = {0, 2, 4, 1, 3};
    static int ay[5] = {0, 6, 0, 3, 3};
    static int pen[5] = {3, 2, 2, 2, 2};
    float f = ht / 7;
    int i;
    for( i = 0; i < 5; ++i )
        Plot( x + f * ax[i], y + f * ay[i], pen[i] );
}
```

Figure 3-9. Routine to Draw the Character "A".

graphic, e.g. carriage return, beep, delete. For example,

'A'	=	101 octal	=	65 decimal
'a'	=	141 octal	=	97 decimal

The first approach to coding the full Symbol routine might be to create 96 individual routines, one per graphic character, and use a list of IF statements to decide which routine to call. This also illustrates the issue of *character spacing* (Figure 3-10).

The variable dx is the accumulated horizontal distance from the initial x location to the next character to be drawn. In this case, it is computed using a convention that the spacing between characters shall be 6/7 of their height (a convention in the Calcomp package). This is *fixed width*

```
void Symbol( float x, float y, float ht, char *string, int ns)
{
        float dx = 0.0;
        int i;
        for( i = 0; i < ns; ++i ) {
                if( string[i] == 'A' )
                        A( x+dx, y, ht );
                else if( string[i] == 'B' )
                        <etc.>
                dx += ht * 6.0 / 7.0;
}
```

Figure 3-10. `IF` Statement Version of `Symbol`.

spacing. Another approach is *variable width spacing*, or *proportional spacing*, where the distance between characters depends on each individual character. In this example, this could be done by adding to each character definition a point with coordinates "*w* 0 3" , where *w* is the horizontal distance, in definition grid units, to the start of the next character and would vary for each character. Thus, the letter "i" could move a smaller distance, i.e. be thinner, than the letter "m."

3.3.1 Font Data Structures

As can be readily seen, there must be a better way to create a routine to draw any character string than the list of IF statements previously shown. A data structure is needed to represent a whole character set, i.e. a *font*:

There are (at least) three data structures that can be used to store the font, varying in the constraints they impose on the order that the character definitions occur in the data arrays and the size of each character, that is the number of points per character. Figure 3-11 illustrates the elements of the structures described below.

1. ASCII order, fixed size. 0 — 127

The character definitions in the data arrays must appear in increasing numerical order according to the ASCII codes. Also, the definitions must contain the same number of points, which we will assume is given in a predefined constant, CHARLEN. Therefore, the dimensions of `xgrid`, `ygrid` and `pen` are 128*CHARLEN (assuming 128 characters, with ASCII codes 0 to 127, will be stored). The data for a given character can be found using only the ASCII code. The index of the start of the definition for the character with ASCII code "ascii" is computed as CHARLEN * (ascii)

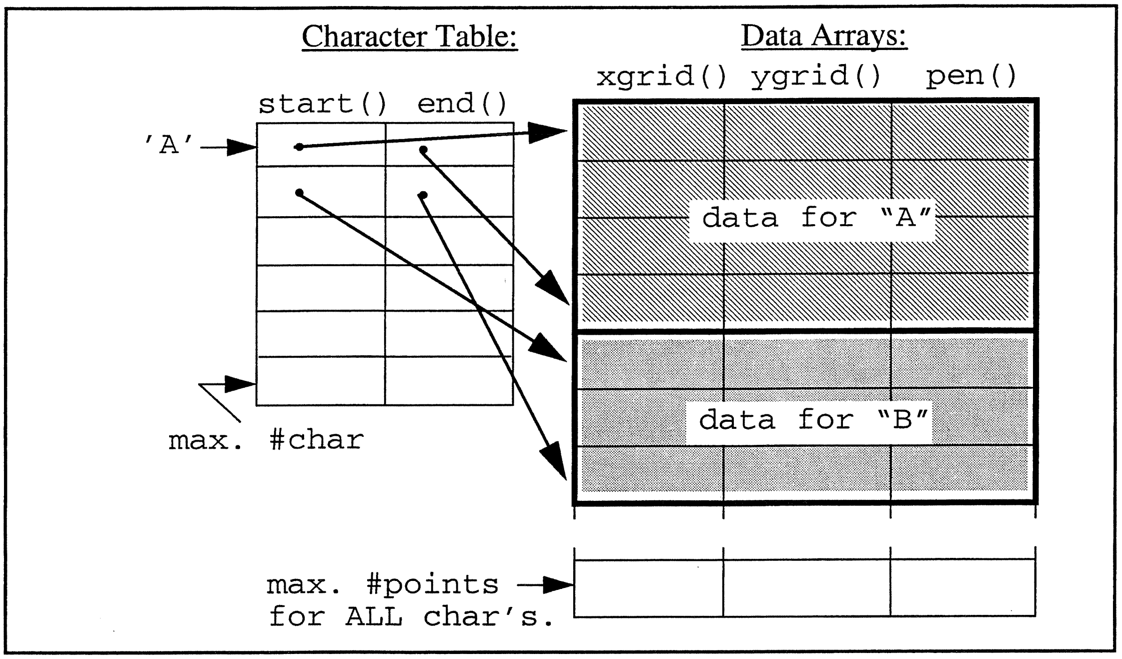


Figure 3-11. A Font Data Structure.

-1), and the definition has CHARLEN points.

2. ASCII order, variable size.

The definitions appear in increasing ASCII order, but the sizes of the definitions are allowed to vary. The is an improvement over the fixed size approach because now each character can be designed without concern for exceeding the size limit or wasting valuable data space. However, the starting index in the data arrays can no longer be computed from the ASCII code alone. Another data array must be created to store the starting indices in the data arrays: the `start` table. This table is indexed by the ASCII code. Due to the fact that the definitions are in ASCII order, the end of one definition can be computed from the start of the definition for the character with the next ASCII code.

3. Unordered, variable size. Requiring ASCII order can be overly constraining. The data can be more easily created and edited, for example, if definitions are allowed to appear in any order. The definitions are no longer in ASCII order, so the end of a definition can no longer be computed as above. This makes it necessary to extend the table with another data array to store the end index of each definition, the `end` array. (Some may find it more convenient to store the size of each character.) The arrays `start` and `end` are indexed by the ASCII code.

Figure 3-12 shows how routine Symbol could be coded using these data structures.

```
void Symbol( float x, float y, float ht, char *string, int ns )
{
      float f, dx;
      int ascii, i1, i2, i, j;
      static int xgrid[?]={...}, ygrid[?]={...}, pen=[?]{...};
      static int start[?]={...}, end[?]={...};
      f = ht / 7.0;
      dx = 0.0;
      for( i = 0; i < ns; ++i ) {
            ascii = string[i];
/* option 1: ascii order, fixed size */
            i1 = CHARLEN*(ascii)-1)+1;
            i2 = i1 + CHARLEN - 1;
/* option 2: asci order, variable size */
            i1 = start(ascii);
            i2 = start(ascii+1)-1;
/* option 3: unordered, variable size */
            i1 = start(ascii);
            i2 = end(ascii);
            for( j = i1; j <= i2; ++j )
                  plot( x+dx + f*xgrid[j], y+f*ygrid[j], pen[j] );
            dx = dx + ( ht * 6.0 / 7.0 );
      }
}
```

Figure 3-12. Symbol Using A Font Data Structure.

3.4 Window to Viewport Mapping

We now extend the low-level graphics capabilities by developing some fundamental mathematical elements for 2D graphics. To begin, consider the task of drawing data read from a file into a known plotting area. The problem is that the program does not know the data coordinates until the data is read. The user, i.e. the creator of the data, could be forced to manipulate (transform) the data to fit in the plotting area, but this is generally unacceptable and far too device specific. There is a better, more general approach, based on the graphical process known as *window to viewport mapping*.

First, establish some terms:

data window: a rectangle defined in *data units*, or *world coordinates.*

viewport: a rectangle defined in *device units*, or *device coordinates.*

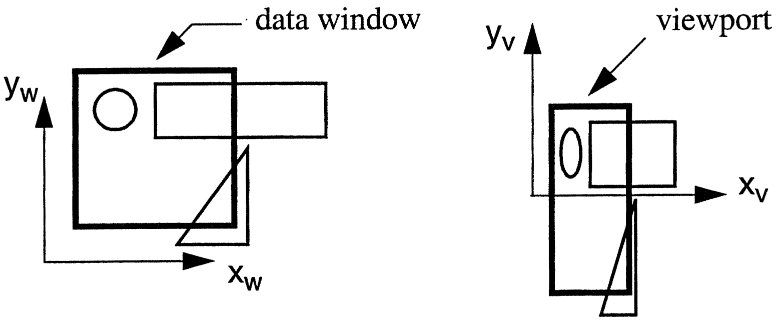


Figure 3-13. Data Window and Viewport.

3.4.1 Mapping Equations

The problem is to map, or transform, world coordinates into device coordinates so they can be drawn on the device. The variables that represent the location and dimensions of each space are shown in Figure 3-14. Think of mapping the window boundary to the viewport boundary. The derivation of the mapping from coordinates in window units, $[x_w,y_w]$, to the corresponding coordinates in viewport units, $[x_v,y_v]$, using the window and viewport parameters involves three steps:

1. Translate to data window coordinates relative to [wcx, wcy]:

$$x_1 = x_w - wcx$$

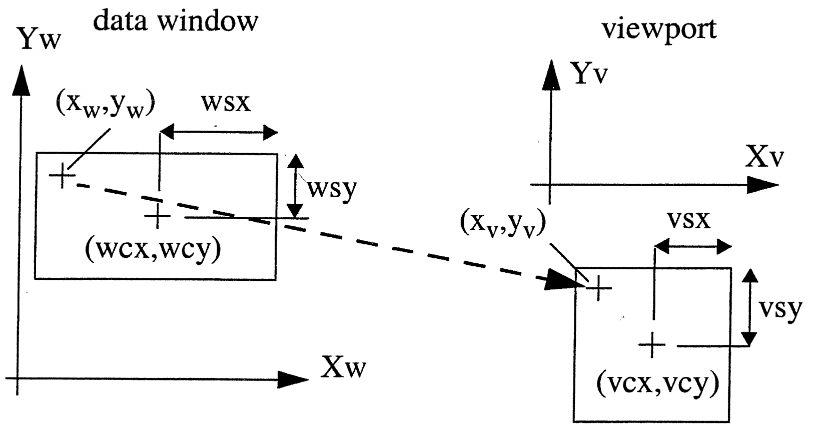


Figure 3-14. Data Window and Viewport Variables.

$$y_1 = y_w - wcy$$

2. Scale from window to viewport coordinates relative to [vcx, vcy]:

$$x_2 = x_1 \frac{vsx}{wsx}$$

$$y_2 = y_1 \frac{vsy}{wsy}$$

3. Translate to viewport coordinates relative to the viewport origin:

$$x_v = x_2 + vcx$$

$$y_v = y_2 + vcy$$

Combine these into single equations:

$$x_v = (x_w - wcx) \frac{vsx}{wsx} + vcx$$

$$y_v = (y_w - wcy) \frac{vsy}{wsy} + vcy$$

As a check, examine the units of the last equation for x_v and y_v:

$$(\text{device units}) = (\text{data units - data units}) \frac{(\text{deviceunits})}{(\text{dataunits})} + (\text{device units})$$

As expected, we get (device units) = (device units).

As a further validation, select a test point, the upper right corner of the data window, and map it to viewport coordinates. The coordinates of the upper right corner are: [wcx+wsx, wcy+wsy]. Substituting these for x_w and y_w into the equations above,

$$x_v = (wcx+wsx - wcx)\,\frac{vsx}{wsx} + vcx \qquad = vsx + vcx$$

$$y_v = (wcy+wsy - wcy)\,\frac{vsy}{wsy} + vcy \qquad = vsy + vcy$$

These are the expected values.

3.4.2 Uniform Scaling

If the two scale factors $\frac{vsx}{wsx}$ and $\frac{vsy}{wsy}$ are not equal, i.e. the *aspect ratios* (width/height) of the window and viewport are not equal, the data is unequally scaled in x and y, causing *distortion.*

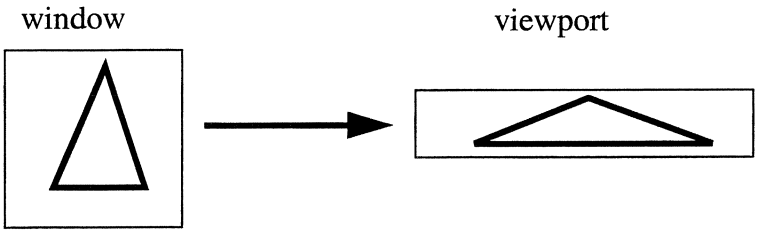
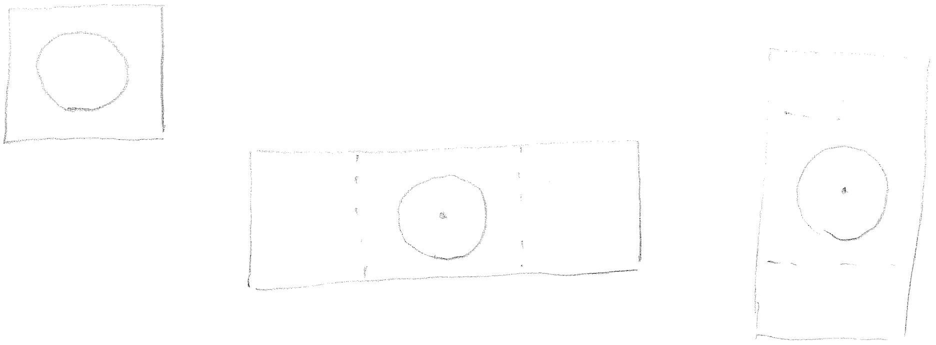


Figure 3-15. Distortion Due to Unequal Scaling in X and Y.

This can be avoided by altering the mapping process to perform *uniform scaling*. This involves using a single scale factor, $\text{factor} = \text{MIN}(\frac{vsx}{wsx}, \frac{vsy}{wsy})$, in place of two separate factors in the second step. In this case, the window boundary will map on or inside the viewport boundary.

In many applications, the data window is computed from the given data to be the smallest enclosing rectangle, often called the "*extent*." The extent is computed by scanning the data coordinates to find the extrema for x and y, which are the coordinates of the diagonals of the bounding rectangle: [xmin, ymin] and [xmax, ymax]. From these coordinates, the appropriate data



window variables are easily found.

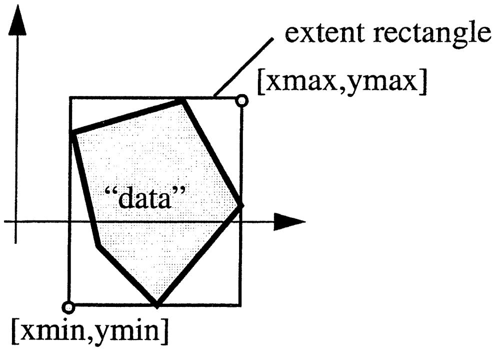


Figure 3-16. Data Extent.

3.5 2D Transformations

Window to viewport mapping involved coordinate transformations. In general, a transformation is a mapping:

1. of one set of points into another in a fixed reference frame, (Figure 3-17)

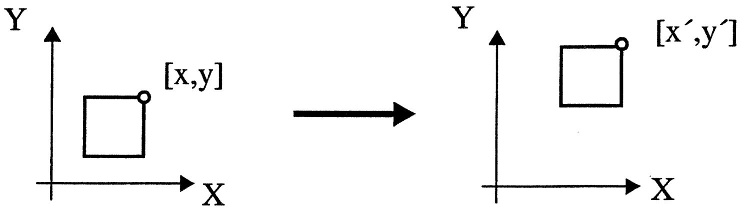


Figure 3-17. Coordinate Transformation.

2. or from one coordinate system to another (Figure 3-18). This is termed

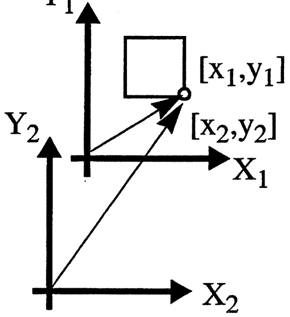


Figure 3-18. Change of Basis Transformation.

change of basis. For now, focus on the coordinate transformations, the mapping of a set of points (data) into another.

3.5.1 Review of Matrix Representation

Algebraic representations for transformations, i.e. variable substitutions in equations, limit computer implementations in several ways:

1. a special procedure is needed for each basic transformation,
2. it is difficult (sometimes impossible) to provide general transformation capabilities in algebraic form in the computer,
3. it is not generally possible to find the inverse of the algebraic equations resulting from a series of transformations. (For example, this is needed to restore the original position of an object after several transformations.)

We can represent the equations in a more general and computationally convenient way using matrices. In general, two equations of the form

$$x' = a\,x + b\,y$$
$$y' = d\,x + e\,y$$

are expressed in the *matrix equation*:

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & d \\ b & e \end{bmatrix}$$

when the coordinates x and y are represented as <u>row vectors</u>. When the coordinates are represented as <u>column vectors</u>, the matrix equation appears as,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Define symbols for the vectors and matrix:

$$V' = \begin{bmatrix} x' & y' \end{bmatrix} \qquad V = \begin{bmatrix} x & y \end{bmatrix} \qquad \mathbf{M} = \begin{bmatrix} a & d \\ b & e \end{bmatrix}$$

Using row vectors, the symbolic equation is : $V' = V\ \mathbf{M}$, and using column vectors, it is $V' = \mathbf{M}^T V$, where $\mathbf{M}^T$ is the matrix transpose of **M**. Given the symbolic equation, it is generally apparent whether row or column vectors are in use. However, given only a matrix, it is necessary to know if this matrix is to be applied to row vectors (vector before matrix) or column vectors (vector after matrix). These notes will use row vectors for coordinates.

For 2D operations, we are tempted to make **M** 2 by 2, but this disallows constants because all **M** elements multiply x or y. Expand the representations of **M** and V.

$$V = \begin{bmatrix} x & y & 1 \end{bmatrix} \quad \mathbf{M} = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}$$

This will work, but looking ahead we see that having **M** 3 by 2 will cause problems because we cannot invert a 3 by 2 matrix and we cannot multiply a 3 by 2 by a 3 by 2. For these reasons, we add a third column to **M**:

$$\mathbf{M} = \begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3.5.2 Basic Transformations

There are three "basic" transformations that can be combined to form more complex transformations: translation, scaling, and rotation.

1. Translation:

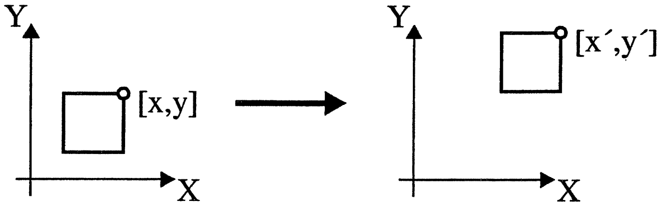


Figure 3-19. Translation.

$x' = x + Tx$

$y' = y + Ty$

The translation transformation in functional form and matrix form:

$$\mathbf{T}(Tx,Ty) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix}$$

2. Scaling:

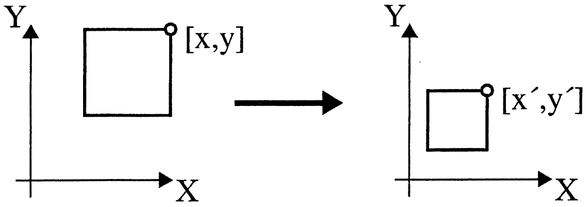


Figure 3-20. Scaling.

$x' = Sx\ x$

$y' = Sy\ y$

The scaling transformation in functional form and matrix form:

$$S(Sx,Sy) = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that position is affected by scaling. The above equations scale "about the origin." Mirror images can be done (for some geomtric forms) using negative scale factors. Distortion, or scaling with different factors in x and y, can be avoided simply by using a single scale factor Sx = Sy = 'S'.

3. Rotation:

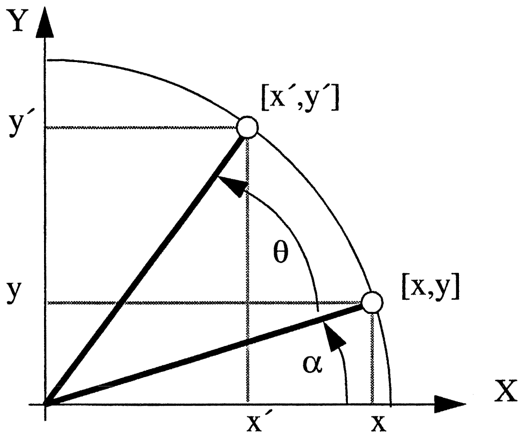


Figure 3-21. Rotation.

Given [x,y] and α, rotate by the angle θ.

$x' = R\cos(\ \alpha + \theta\ )$

$y' = R\sin(\ \alpha + \theta\ )$

Expanding,

$$x' = R\,(\cos\alpha\cos\theta - \sin\alpha\sin\theta)$$

$$y' = R\,(\sin\alpha\cos\theta + \cos\alpha\sin\theta)$$

and then recognizing that $x = R\cos\alpha$ and $y = \sin\alpha$,

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

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

The rotation transformation in functional form and matrix form:

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Like scaling, the rotation transformation is about the origin (Figure 3-22).

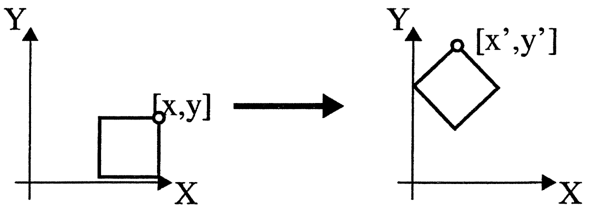


Figure 3-22. Rotation Occurs About the Origin.

For simplicity in writing and reading matrix equations, we generally write matrix transformations using the *functional forms*, which were shown above:

"**T**(Tx,Ty)" for translation,

"**S**(Sx,Sy)" for scaling,

"**R**(θ)" for rotation,

The *identity transformations*, or null transformations, are those that produce the identity matrix: **T**(0,0), **S**(1,1), **R**(0).

3.5.3 Concatenation (Composition)

A series of transformations can be combined (*concatenated*) into one. For example, consider scaling an object about a given point. We want to reduce the size of the box without changing its position [a,b] (Figure 3-23).

First carry out the derivation algebraically:

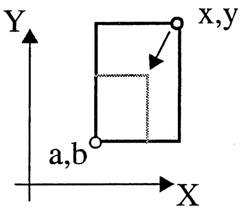


Figure 3-23. Scaling About a Point.

1. Translate so that position [a,b] becomes the temporary origin. [a,b] could be any reference point:

   $x_1 = x - a$ $y_1 = y - b$

2. Scale to the correct size:

   $x_2 = Sx\, x_1$ $y_2 = Sy\, y_1$

3. Translate again to restore the position [a,b]:

   $x_3 = x_2 + a$ $y_3 = y_2 + b$

Notice that x_3 is a function of x_2 and y_2 is a function of x_1. Substituting:

$x_3 = (x - a)\, Sx + a$ $y_3 = (y - b)\, Sy + b$

The order is very important.

Any number of basic transformations can be concatenated into one pair of equations, eliminating the intermediate results that would have required extra data storage and computation.

Concatenation is easy in matrix form. For the scaling example:

1. translate: **T**(-a,-b)
2. scale: **S**(Sx,Sy)
3. translate: **T**(a,b)

Combining these into a matrix equation:

$$
\begin{aligned}
v_3 &= v_2\, \mathbf{T}(a,b) \\
&= [\, v_1\, \mathbf{S}(Sx,Sy)\,]\, \mathbf{T}(a,b) \\
&= [\, v\, \mathbf{T}(-a,-b)\,]\, \mathbf{S}(Sx,Sy)\, \mathbf{T}(a,b)
\end{aligned}
$$

This last equation is easily converted to matrix form by substituting the appropriate matrices. We can define the *composite transformation matrix*, **M** as:

$$\mathbf{M} = \mathbf{T}(-a,-b)\, \mathbf{S}(Sx,Sy)\, \mathbf{T}(a,b)$$

and therefore,

$v_3 = v\,\mathbf{M}$.

Note again that the order of the matrices is critical! For practice, find **M** using the symbolic matrices:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{bmatrix} \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}$$

$$= \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ -aSx & -bSy & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}$$

$$= \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ (-aSx + a) & (-bSy + b) & 1 \end{bmatrix}$$

$M = M_1 M_2$

$M^{-1} = (M_1 M_2)^{-1}$

$= M_2^{-1} M_1^{-1}$

$T(a,b) \quad T(-a,-b)$

$R(\theta) \quad R(-\theta)$

$S(a,b) \quad S(\frac{1}{a}, \frac{1}{b})$

Multiplying out the matrix equation symbolically yields the two expected algebraic equations:

$$x' = x\,Sx + (-a\,Sx + a) \quad = \quad (x - a)\,Sx + a$$

$$y' = y\,Sy + (-b\,Sy + b) \quad = \quad (y - b)\,Sy + b$$

Thus, a single 3 by 3 matrix can represent any combination of basic 2D transformations in a compact, codable and simple form. In the computer, these are all numbers.

Functional form provides a convenient way of understanding and even solving transformation equations without doing matrix multiplication. For example, given the transformation equation in functional form, one can visualize the resulting picture by performing each basic transformation "by hand" on the objects. Try this with the previous "scaling about a point" example. This is easier (and much faster on a test) than performing the arithmetic.

Suppose we wanted to find the inverse of a series of transformations. Instead of computing the inverse of the concatenated 3 by 3 matrix, utilize the fact that the inverse of a product of matrices (transformations) is the product of the inverses in reverse order. The inverses of the basic transformations can be computed by inspection:

$\mathbf{T}(a,b)^{-1} = \mathbf{T}(-a,-b)$, $\mathbf{S}(a,b)^{-1} = \mathbf{S}(1/a,1/b)$, and $\mathbf{R}(\theta)^{-1} = \mathbf{R}(-\theta)$.

3.6 Two-Dimensional Clipping

Window to viewport mapping transforms coordinates from one coordinate system to another. Coordinates outside the window map to coordinates outside the viewport. It is often desirable or necessary to remove or "clip" portions of the data that lie outside a given boundary.

At first glance, one might be tempted to write a routine with a sequence of `IF` tests, each dealing with a special case. Fortunately, there is a more organized approach. An elegant algorithm, known as the Cohen - Sutherland clipping algorithm, was developed for line clipping. The algorithm is based on a special integer code, called a *region code* or *end point code*, that is computed for each end point of a line. This code compactly designates the boundary of the 2D region or zone in which the point lies, and has been designed for ease of use and efficiency in handling special cases.

Two-dimensional space is divided by four boundaries into 9 regions, all of which are semi-infinite (extend in one direction to infinity) except one, which is designated the "visible" region. Each boundary is assigned a bit code, or power of two integer, uniquely designating the violated boundary. The visible region is the center region and has a code of 0. One possible coding scheme is shown below:

TOP	BOTTOM	RIGHT	LEFT	IN
1000_2 (8_{10})	0100_2 (4_{10})	0010_2 (2_{10})	0001_2 (1_{10})	0

The region codes are computed based on these conditions as shown in Figure 3-24. Code to compute the clipping region code is illustrated in Figure 3-25. Note that points on the boundary of the visible region, as well as those inside, have a region code of IN.

Instead of attempting to detect and process the many possible cases of lines with end points in different regions, the algorithm uses the region codes to determine if the line is entirely visible or entirely invisible. If not, then one of the end points that is outside any boundary is re-computed to be the intersection of the line and the boundary, sometimes called "pushing" the end point to the boundary. This "clipped" line is then subjected to the same process. This "test and clip" iteration continues until the re-computed line is entirely visible or invisible.

The region codes greatly simplify and speed the visibility tests on a line. A line is visible

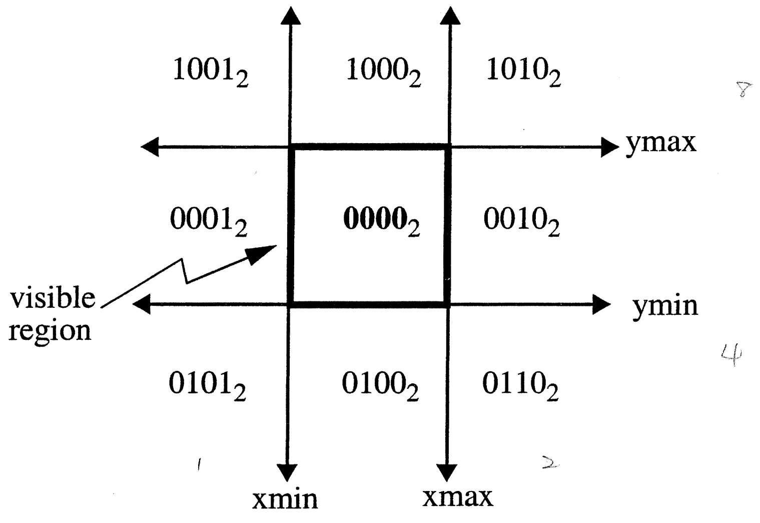


Figure 3-24. Clipping Region Codes.

```
code = IN;
if( x < xmin ) code = LEFT;
     else if( x > xmax ) code = RIGHT;
if( y < ymin ) code = code + BOTTOM;
     else if( y > ymax ) code = code + TOP;
```

Figure 3-25. Clipping Region Computations.

when both end point codes are zero, a simple and fast computation. The value of the region bit-coding scheme is seen in the test for an entirely invisible line. Without the region codes, one could code this test as an unwieldy statement of the form “if both end points are above the top boundary of the visible region, or if both end points are below the bottom boundary of the visible region, if both end points are left of the left boundary of the visible region, or if both end points are right of the right boundary of the visible region,” then the line is entirely invisible. With the region codes, however, this test is simple. Notice that the condition of both end points violating a boundary, i.e. above the top, below the bottom, right of right, left of left, means that the region codes will have 1’s in corresponding bit positions. This means that the integer “and” of the two region codes will be non-zero. In Fortran, the "and" computation is an implicit integer function,

`result = and( i1, i2 )` or for some compilers: `result = i1 .and. i2`

and the ‘&’ operator in C,

`result = i1 & i2.`

0110
0010
0010

"Both right"

The invisibility test reduces to a simple computation and test against zero. In effect, the “and” operation is four parallel `IF` statements.

Computing the clipped end point when a line crosses a clipping boundary is a geometric computation that is best done using the symmetric equation of a line. Given a line with end points $[x_1,y_1]$ and $[x_2,y_2]$, any point $[x,y]$ along that line satisfies the following equation,

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} .$$

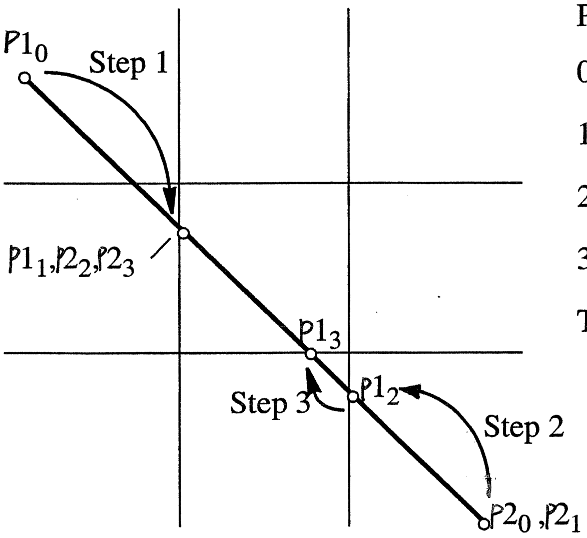
Top ymax

For a given boundary, either x (x = xmin for the LEFT boundary and x = xmax for the RIGHT) or y (y = ymax for TOP and y = ymin for BOTTOM) will be known, so the equation can be solved for the other coordinate. There are four boundaries, so there will be four pairs of equations. For example, if end point 1 of line crosses the RIGHT boundary, then end point 1 would be re-computed (clipped) as follows,

$$y_1 = \frac{y_2 - y_1}{x_2 - x_1}(\mathrm{xmax} - x_1) + y_1$$
$$x_1 = \mathrm{xmax}$$

Note that the order is critical. The y_1 computation must use the original x_1 coordinate.

Although the algorithm appears at first to clip unnecessarily many times, for example, a line may be clipped several times before being found to be outside the visible region, in fact it is efficient and very compact. Figure 3-26 illustrates how a line is processed during the iterations of the algorithm.



Pass	End Points	Action
0	$P1_0$ -$P2_0$	Original Line
1	$P1_1$ -$P2_1$	1_0 was LEFT
2	$P1_2$ -$P2_2$	2_1 was RIGHT
3	$P1_3$ -$P2_3$	1_2 was BELOW

The resulting line $1_3 2_3$ is visible.

Figure 3-26. Illustration of the Iterations of the Clipping Algorithm.

An implementation of the algorithm is shown in Figure 3-27.

```
void ClipLine2D( float x1, float y1, float x2, float y2 )
{
        int c1, c2, ctemp;
        float ftemp;
        c2 = RegionCode( x2, y2 );
        for(;;){
                c1 = RegionCode( x1, y1 );
                if( c1 == IN && c2 == IN ) {
                        /* process visible line from [x1,y1] to [x2,y2] */
                        return;
                }
                if( (c1 & c2) != 0 ) {
                        /* the line is invisible - skip or notify caller*/
                        return;
                }
                if( c1 == IN ) {  /* swap points 1 and 2 to simplify cases */
                        ctemp = c1; c1 = c2;c2 = ctemp;
                        ftemp = x1; x1 = x2;x2 = ftemp;
                        ftemp = y1; y1 = y2;y2 = ftemp;
                }
                if( c1 & LEFT ) {
                        y1 += ( y2 - y1 ) / ( x2 - x1 ) * ( xmin - x1 );
                        x1 = xmin;
                } else if( c1 & RIGHT ) {
                        y1 += ( y2 - y1 ) / ( x2 - x1 ) * ( xmax - x1 );
                        x1 = xmax;
                } else if( c1 & TOP ) {
                        x1 += ( x2 - x1 ) / ( y2 - y1 ) * ( ymax - y1 );
                        y1 = ymax;
                } else {      /* BOTTOM */
                        x1 += ( x2 - x1 ) / ( y2 - y1 ) * ( ymin - y1 );
                        y1 = ymin;
                }
        }
}
```

Figure 3-27. Implementation of the Clipping Algorithm.

3.7 Rasterization

The electrostatic plotter and laser printer are called *raster* devices because images are produced by dots displayed in a strict order. A plotter is a *random vector* device because lines can be drawn by continuous strokes in any direction in any order.

To display lines on a raster device, lines must be converted into *raster units*, e.g. epps or dots. This process is called *rasterization.*

Consider the process of computing the raster units on a page that will be printed using an electrostatic plotter or laster printer. Imagine superimposing on the page a grid of all possible *raster units* at their respective *raster locations*. Place over the grid a 2D coordinate system where the centers of raster units are whole numbers. During window to viewport mapping, the end point coordinates of a line are carefully rounded to the nearest raster location. The grid can be represented as a two dimensional array of raster values that is indexed by the row number, i.e. number of units vertically from the bottom (or top) of the page, and the column number, i.e. the number of units horizontally from the left of the page. Thus, the rounded [x,y] coordinates can be used as array indices. This suggests that a computer implementation of rasterization involves a two dimensional array of integers representing all the values at raster locations.

As shown in Figure 3-28, the problem of rasterizing a line given the starting and ending grid locations appears very similar to the plotter stepping problem. In fact, the stepping algorithm can be used almost directly, provided we give a different interpretation to a "step." In the case of

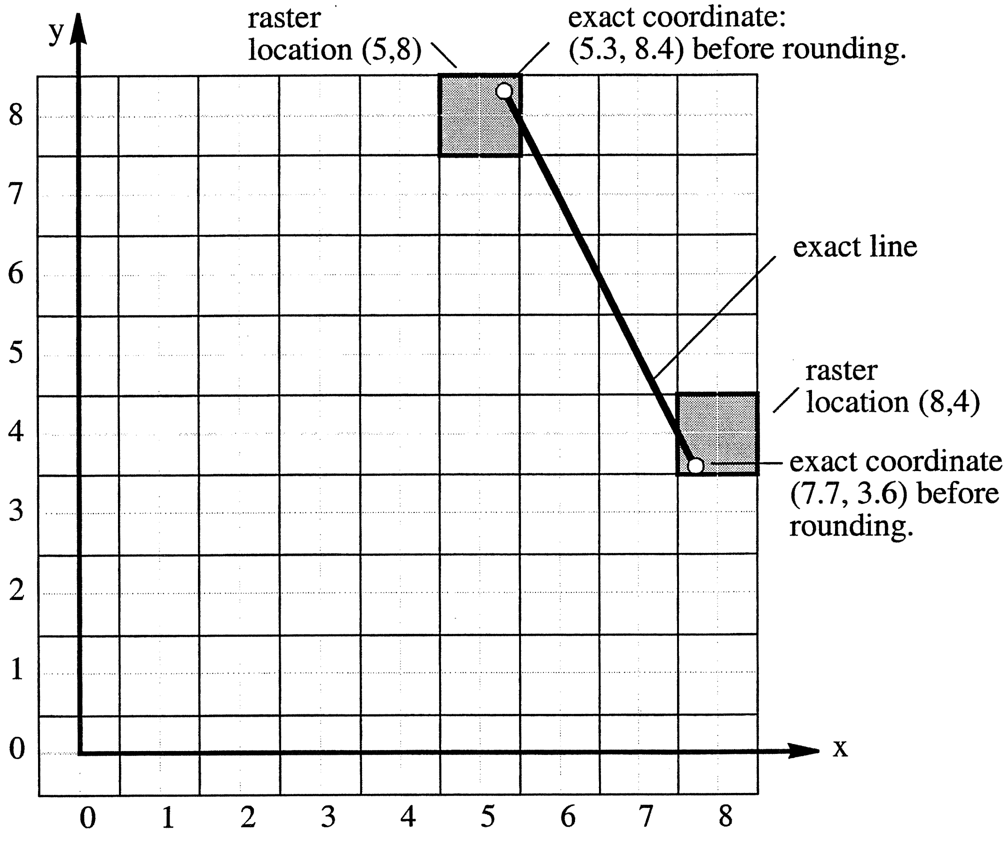

y
raster location (5,8)
exact coordinate: (5.3, 8.4) before rounding.
exact line
raster location (8,4)
exact coordinate: (7.7, 3.6) before rounding.
x
0
1
2
3
4
5
6
7
8
0
1
2
3
4
5
6
7
8

Figure 3-28. Raster Grid and a Line.

a plotter, a step was the motion of the stepping motors controlling the pen motion. A plotter step draws a small straight line that was horizontal, vertical, or diagonal.

For a raster device, however, a step must be interpreted in terms of raster locations on a grid. Instead of moving the pen, a step means to advance incrementally to the next raster location. The stepping code is easily modified to do this. The `PlotterCommand` calls should be replaced by statements that increment the [x,y] coordinates of the raster location and set the value in the array to "on" (typically the number 1 or 0).

Rasterization creates some special problems in computer storage. Suppose we try a "brute force" method, creating an array to represent all raster locations on a printer page and then computing the entire raster array before printing it. This approach is called a *Page System.* A plot on an 11" high page, one inch wide, at 200 raster units/inch contains: 200 x 200 x 11 = 440,000 values. If we store all information in memory,we would need almost 500,000 words of memory.

If we "pack" the data so that the "on/off" information for N raster units are stored in bits in a single word (16, 32 or more bits/word are typical), we still need around 7,333 to 13,750 words, just for that one inch.

Using a page system approach is not practical in some cases when the computer system lacks sufficient memory. Another approach is to make sequential passes over the data, rasterizing only one small *strip* at a time, called *Strip-Page Plotting*. For each strip, which amounts to a rectangular region of the plotting page (like a viewport), convert all linework (and symbols) into raster data, send this strip of data to the plotter, move the strip, then repeat the process until we have passed over all the data (Figure 3-29).

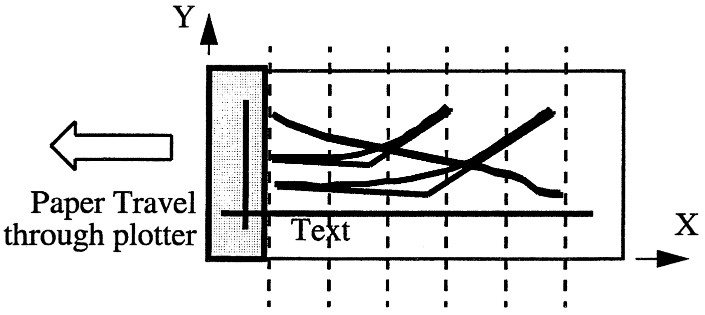


Figure 3-29. Strip-Page Plotting.

3.8 References

1. J. E. Bresenham, "Algorithm For Computer Control of A Digital Plotter," IBM Systems Journal, Volume 4, No. 1, 1965.

3.9 Review Questions

1. Derive the window to viewport mapping for viewports defined on devices that have the origin at the upper left of the screen, with the horizontal coordinate "h" increasing right, and the vertical coordinate "v" increasing down.
2. Derive a two-step window to viewport mapping procedure which first maps world data in the window into a unit viewport with center [0,0] and x and y from -1 to 1 (called normalized device coordinates), and then maps this data to the viewport. This procedure isolates device details from the basic mapping process and is useful when data must be clipped. The data is clipped in normalized device coordinates, which are independent of both the window and viewport, thus isolating the clipping process from them.
3. Often graphics packages map world data into "clipping coordinates" prior to performing 2D clipping in order to simplify the computations. The clipping coordinates of visible points lie between 0.0 and 1.0. After clipping, the end points of visible portions of lines are mapped to device coordinates and drawn. The process is: (1) apply the transformation [T1] to map world coordinates into clipping coordinates, (2) 2D clip, and (3) apply the transformation [T2] to map the clipped coordinates into device coordinates. Given a square window in world coordinates with center (wcx, wcy) and half-size ws and a square viewport in device coordinates with center (vcx, vcy) and half-size vs:

   (a) Show the basic transformations forming [T1] in symbolic functional form.

   (b) Show [T1] as a single matrix.

   (c) Find the expressions for the clipping coordinates of the world data point midway between the window center and right boundary.

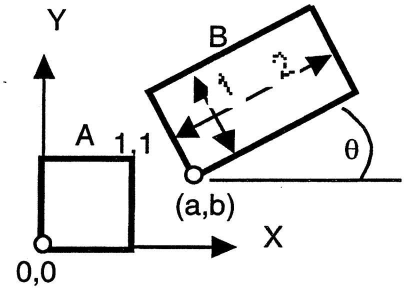
   (d) Show in symbolic functional form the basic transformations forming [T2] for drawing on GRAFIC windows.

4. Window to viewport mapping transforms world data into device data based on a window [wcx,wcy,wsx,wsy] and a viewport [vcx,vcy,vsx,vsy]. Derive this transformation as a single matrix with symbolic equations as its elements. Assume the X and Y data and device axes are parallel .

5. How could the font data and code for `Symbol` in Figure 3-12 be modified to produce proportional spacing?

6. Draw a diagram illustrating end point codes for the line clipping algorithm. Explain how and why they are used.

7. Show the list of basic transformations in functional form that will make object B from object A in the following figure.



8. Character fonts for raster displays are often defined as closed boundaries consisting of straight lines. A character instance is drawn by setting all pixels in the interior of its transformed boundary.

   (a) Diagram and explain the data structure necessary to store such a font.

   (b) Explain, in step by step form, how to implement the routine `DrawCharacter(ichar,h,v)`, which draws the character whose ASCII code is `ichar` at the location [h,v] on the screen using GRAFIC.

9. A low-cost laser printer must be sent an entire 8 by 10 inch page of pixels (in scan-line order). The page is printed after all pixels have been received. Each pixel is represented by a character: 'b' for black, 'w' for white. The printer has x and y resolutions of 300 pixels/ inch. Describe the software needed to produce a printed page on this device, starting with an application program that calls the following routines:

   1. STARTPICTURE is called to initialize plotting.

   2. LINE(x1, y1, x2, y2) draws a straight line from (x1,y1) to (x2,y2), where the coordinates are given in (real) inches from the lower left of the page.

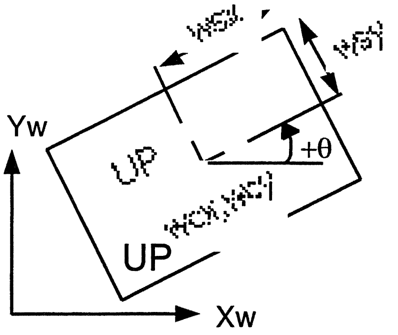
3. ENDPICTURE is called to signify plotting is finished.

(A) Draw a block diagram showing the general organization of the software, including the stages of processing, types of data, and flow of information starting with the application program and ending with the printing process.

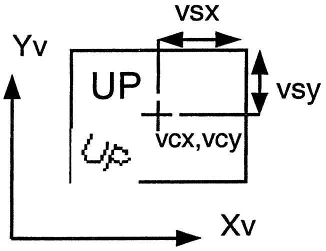
(B) Describe in detail the process of creating data for the printer after the application program has completed.

10. Given the rotated window and horizontal viewport defined below, derive the window to viewport mapping equations, first in functional transformation form, then in matrix form.

Window:



Viewport:



Chapter 4. Dynamic Graphics

Dynamic graphics describes the use of refresh displays for graphical interaction which can involve rapidly changing images and motion.

4.1 The Cathode Ray Tube

At the heart of all dynamic graphics systems is the *cathode ray tube* (CRT), the fundamental electronics for creating visible images on a screen. The basic elements of the CRT are shown in Figure 4-1. The beam starts as a flood of electrons generated in the filament, cathode and control grid. The flood of electrons is focused into a concentrated beam by the focusing system and accelerated into a narrow beam of high energy electrons by the accelerating plates. The beam is deflected by charges in two pairs of charged plates, one horizontal and one vertical, in the deflection system. The horizontal and vertical deflection voltages are controlled by signals from the computer.

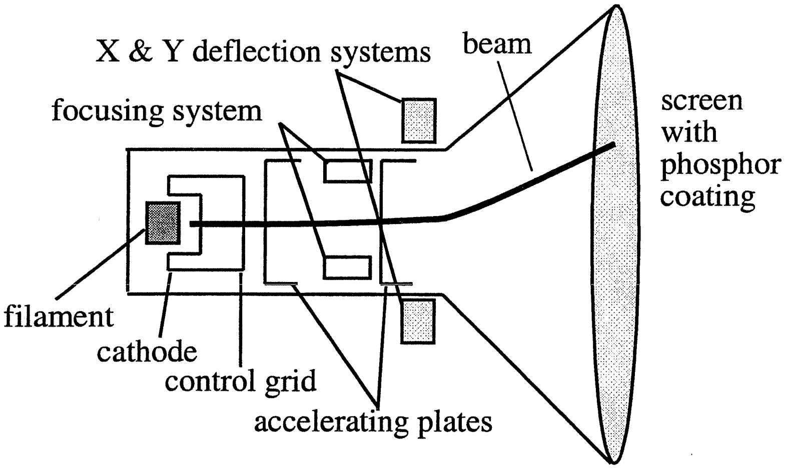


Figure 4-1. Elements of a CRT.

The basic display process begins with the computer setting the voltages in the horizontal and vertical deflection plates to appropriate values for the desired beam location. This is done through *digitial-to-analog* (D/A) converters connected between the computer and the display. The computer sends digital values to the D/A converters that convert the values to voltages. The output voltages of the D/A converters are connected to the X and Y deflection plates. The computer then pulses the beam intensity. Where the beam strikes the screen, the phosphor glows, causing a visible dot. The beam must be turned off quickly to avoid burning the phosphor. This forms the basic *point plotting display* (Figure 4-2).

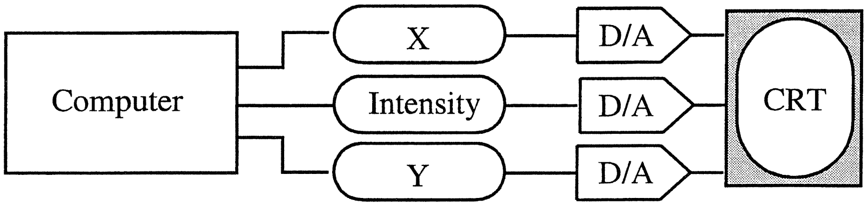
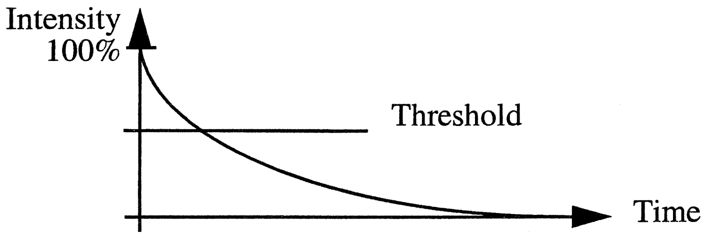


Figure 4-2. Configuration of a Point Plotting Display.

When the beam is turned off, the glow *decays* rapidly based on the *persistence*, or rate of light decay, of the screen phosphor. The glow decays very quickly, so the image must be *refreshed* (re-drawn) often, typically every 1/30 - 1/60 sec., even if the picture does not change. If the glow fades too much before it is refreshed, the eye will detect this intensity variation as *flicker*.



refresh

Figure 4-3. Decay in Intensity of Phosphor Light.

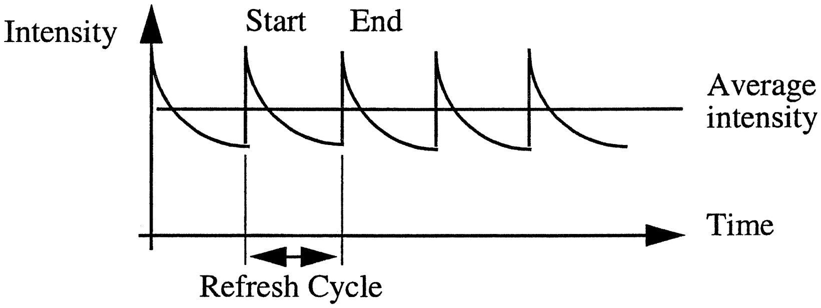


Figure 4-4. Intensity Variations Due to Refresh Cycles.

What would be needed to draw lines instead of points? The display hardware is only capable of drawing dots at locations on the screen, so it is a raster device, similar to the electrostatic plotter and laser printer. Therefore, a stepping algorithm could be adapted for use here, too. Given the screen locations of two end points of a line to be drawn, the task of the stepping algorithm is to compute the screen locations of the dots that best represent the line. As before, the stepping algorithm will start at one end point and compute "steps" to the next screen locations, ending at the other end point. In this case, a step consists of increments in the horizontal and vertical screen X and Y position.

Hardware can be added to the basic point plotting display to produce two different types of refresh display technology:

Vector continuous random lines: the beam motion is directly controlled by the display commands. The picture is actually drawn by the beam, analogous to pen plotting.

Raster ordered dots: the beam motion always follows the same path. The picture depends on the appearance of the dots, analogous to laser printing.

We will focus on raster technology first.

4.2 Raster Displays

As already mentioned, the common usage of the phrase *raster graphics* means the image is drawn in dots, that can be black & white, in shades of gray, or in colors. Raster graphics is based on the television CRT scanning display technology, or *video*.

4.2.1 Video Technology

A typical television has about 525 *scan-lines* (rows of dots across the screen) and 640 dots per scan-line. The signal received by a television defines every dot on the screen during every "*scan-cycle*". The dots are called picture elements, or "*pixels*."

The length of the scan-cycle is 1/30 second. In 1/30 second, the beam displays all pixels, illuminating them with the correct color/intensity. Televisions actually display the image using *interlace*. The even-numbered scan-lines are refreshed during one 1/60 second cycle, and the odd-numbered lines are refreshed during the next 1/60 second. Interlace smooths the displayed image by reducing the effects of beam motion variations.

Now consider creating the video signal from a computer instead of a camera and

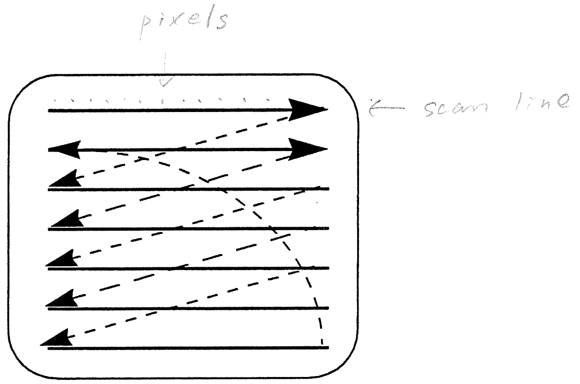


Figure 4-5. Beam Motion for Interlaced Raster Display.

transmitter. Such a raster display must draw each pixel 30 times per second to maintain the image. Just as a television constantly receives new images from a transmitter, the raster display hardware must repeatedly process its data to the display screen to refresh the *image*.

We call this hardware a *raster display processor*.

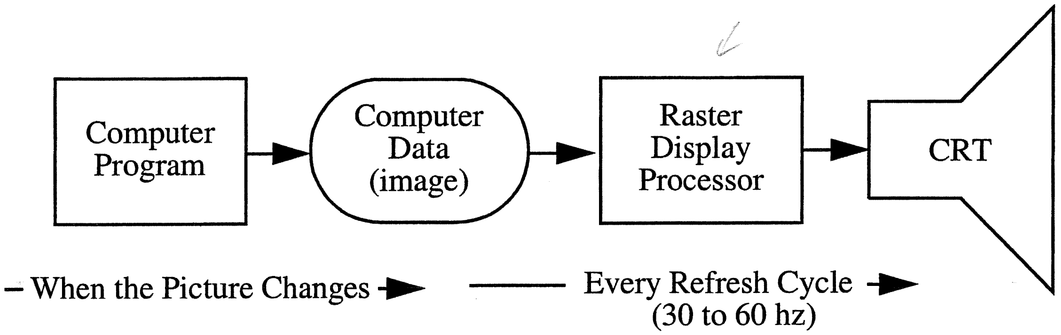


Figure 4-6. Raster Display Process.

4.2.2 The Bitmap Display

The simplest raster display processor is a *bitmap display*, or a black & white (B/W) display. It is so named because, like our discussion of the electrostatic plotter, it uses one memory bit to determine the on/off state of a particular pixel drawn by the display processor. The major difference between the bitmap display and the electrostatic plotter is that it is no longer a static display device. Pixels can be turned on or off each refresh cycle.

Digital data corresponding to the screen image must be maintained for the display processor. This data is stored in computer memory, called *pixel memory*. In the case of a bitmap display, pixel memory contains one bit (binary 0 or 1) for each pixel. Another term for pixel memory is *frame buffer*.

We refer to the numeric value of each pixel as *pixel value.* A bitmap display has pixel values that are 0 or 1, corresponding to black or white (or vice-versa).

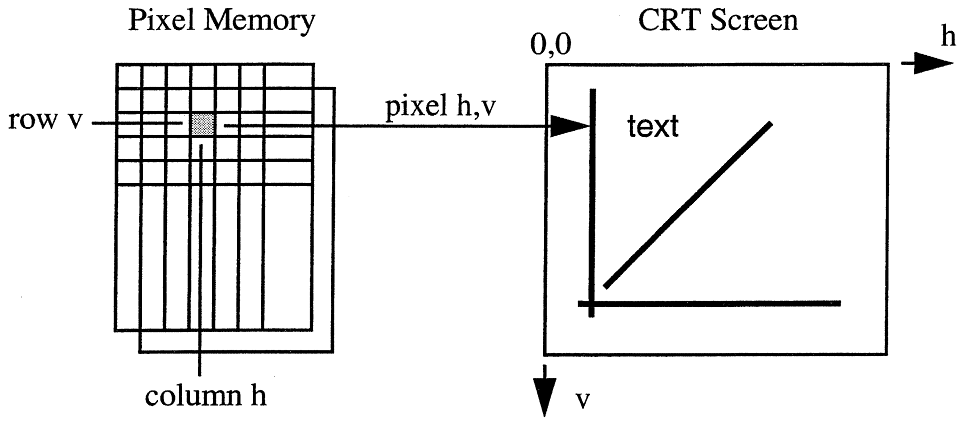


Figure 4-7. Pixel Memory Organization.

4.2.3 Pixel Memory Logic

Pixel memory can be considered as an array of numbers whose dimensions depend on the screen size. The memory is organized into `height` scan-lines, each containing `width` pixels. Think of this as a two dimensional array, `pixels[width,height]` (Figure 4-8). We impose

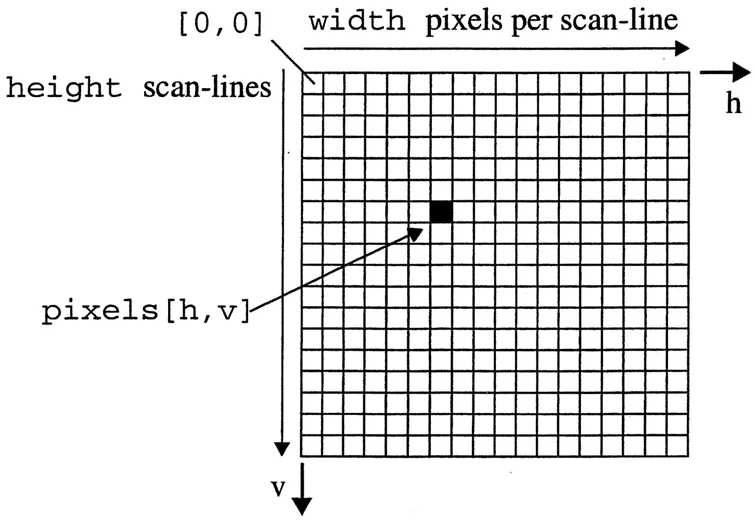


Figure 4-8. `Pixels` Array Variables.

a coordinate system on the pixel array by numbering each pixel by its position as the "hth" pixel in the "vth" scan-line: i.e. the coordinate [h,v], with the first pixel in the upper-left corner having coordinates [0,0]. Historically, the video display order dictates this choice of coordinate system.

It is now possible to alter the display by changing the values in the `pixels` array. For example, setting a pixel is simply the statement: `pixels[h,v] = BLACK`. This pixel will be displayed black by the display processor during the next refresh cycle. The statement `pixels[h,v] = WHITE` "clears" the pixel, i.e. sets it to white, which is often the "background color." One could think of this as "erasing."

A 512 by 512 black-and-white display requires 256K (K = 1024) bits of memory. Also, every pixel is displayed each refresh cycle, even if you have not drawn anything on the screen. That is, a pixel always has a value that is displayed.

4.2.4 Raster Drawing Operations

With the basic ability to set pixel values using their screen coordinates, now consider some basic raster drawing operations.

For example, straight lines can be drawn between pixel locations using a stepping algorithm in which a step is defined as changing pixel coordinates, like rasterization. For example, a minor raster step means incrementing or decrementing the h coordinate and setting the pixel value at the resulting [h,v]. Lines are drawn by stepping across all pixels nearest to the exact line, using the raster Bresenham stepping algorithm, and setting the pixel value for each step.

Another typical raster operation is *filling* an area, like a rectangle, with a given pixel value. The location and dimensions of a rectangle are specified in different ways, for example as the upper left [h,v] coordinates and the width and height (as in X windows), or alternatively the upper left and lower right coordinates (as in Macintosh Quickdraw). Given the rectangle, we simply use nested loops to set every pixel within the rectangle boundaries.

An interesting variation of the drawing processes is allowing *pixel patterns*, such as "50% gray". Typically, a pattern is an ordered sequence of varying pixel values specified over an area. For example, 50% gray would be alternating black and white pixels across and down scan-lines. It is possible in most packages to specify a pattern for the "pen" that is applied to lines or filled areas.

4.2.5 Gray Scale Displays

In order to display shades of gray the display processor hardware must allow a range of intensity values to be displayed at each pixel location, called a *gray scale display.* Instead of the two values on and off, pixel values are now integers that will be converted into intensities by the hardware each refresh cycle. The pixels array contains integers with sufficient bits to hold the range of pixel values.

Suppose we need 1024 *gray levels.* This means that pixel memory must hold pixel values between 0 and 1023, so at least 10 bits per pixel are needed (1024 possibilities = 2^{10}). The resulting pixel memory can be envisioned as a two dimensional array of 10 bit integers, i.e. `int pixels[512,512]`, that can be conceptualized as a three dimensional screen as shown in Figure 4-9. Pixel memory is now “512 by 512 by 10 bits deep”, requiring 2,621,440 bits.

intensity
0 ... 1023

pixel value
10 bits each

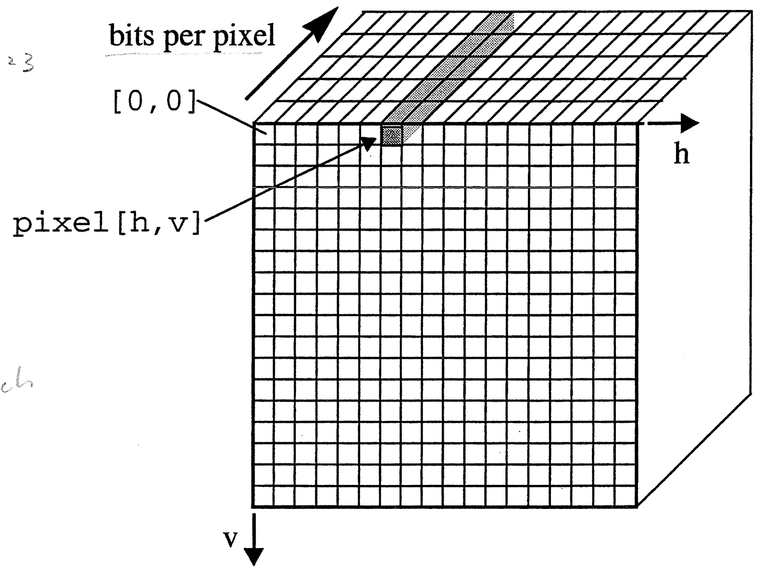


Figure 4-9. Three-Dimensional Conceptual View of Screen Pixel Memory.

4.3 Review Questions

1. Show the memory configurations, including necessary sizes of the hardware components, for two raster display systems capable of displaying 4096 simultaneous colors on a 1024 by 1024 screen: one with direct color and one with mapped color.
2. The Bresenham algorithm computes "steps" that approximate a straight line. The algorithm can be adapted for different devices.What is the Bresenham "step" for a pen plotter? What is the Bresenham "step" for a raster display?
3. Some raster displays have pixel values of black=1 and white=0, while others have white=1 and black=0. If the background color of a given display is white and the foreground (pen) color is black, show the pixel values and explain the visible results of drawing one line in XOR pen mode for each of these cases.
4. Draw a labelled diagram and describe the operation of the two system configurations for a 1280 pixels by 1024 scan-line raster display capable of displaying 4096 colors, at least 512 colors simultaneously — one with direct color and one with mapped (VLT) color.
5. Draw a labelled diagram and describe the display memory contents and the refresh process for:

   (a) a 1024 by 1024 B/W random vector display,

   (b) a 1024 by 1024 by 12 color raster display with a VLT.

Chapter 5. Color

Previously, we have concentrated on the hardware processes and programming techniques for producing single-color images, either black & white or shades of one color. Although the basic mechanism for producing color on a CRT turns out to be a relatively simple extension to what we have already seen, the production and perception of color is a considerably more complex problem that has been studied for hundreds of years. Creating a meaningful color display requires an understanding of various physical and perceptual processes, including a fundamental understanding of the physical properties of light, human color perception, color CRT operation, and computer models for color representation.

5.1 Light

Light is electromagnetic radiation, energy, that is emitted from light sources such as incandescent bulbs and the sun, reflected from the surfaces of objects, transmitted through objects, propagated through media such as air, and finally reaches our eyes. There are several models that have been proposed to describe the physical behavior of light. Generally, it is viewed as having both the properties of particles and waves. The model most appropriate for computer graphics describes light as an oscillating electromagnetic wave, as illustrated in Figure 5-1. As a wave, light has the wave properties of frequency ν, wavelength λ, and amplitude. It is known that frequency and wavelength are related by $\lambda = \frac{c}{\nu}$, where c is the speed of light in the medium (for a vacuum,

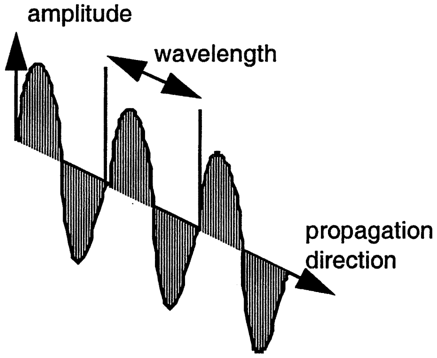


Figure 5-1. An Electromagnetic Wave.

c = 2.998 x 10^8 meters/sec). Wavelength is usually given in units of micrometers "µm" (10^{-6}) or nanometers "nm" (10^{-9}). The electric field intensity of the wave varies sinusoidally in time and space as a function of amplitude and frequency.

Visible light, the light that the human eye can perceive and thus the light that we can see, is only part of the entire electromagnetic spectrum, as shown in Figure 5-2 and Figure 5-3. The term "light" for our purposes means electromagnetic radiation with wavelengths in the range 380 nm to 770 nm. Generally, light that we see is not just a single wavelength, but consists of a continuous, non-uniform distribution of single-wavelength (monochromatic) components. The graph of intensities versus wavelengths forms a *spectral distribution*, or *spectral reflectance curve*, as illustrated in Figure 5-4. The term *spectral* means variance with, or dependence upon, wavelength.

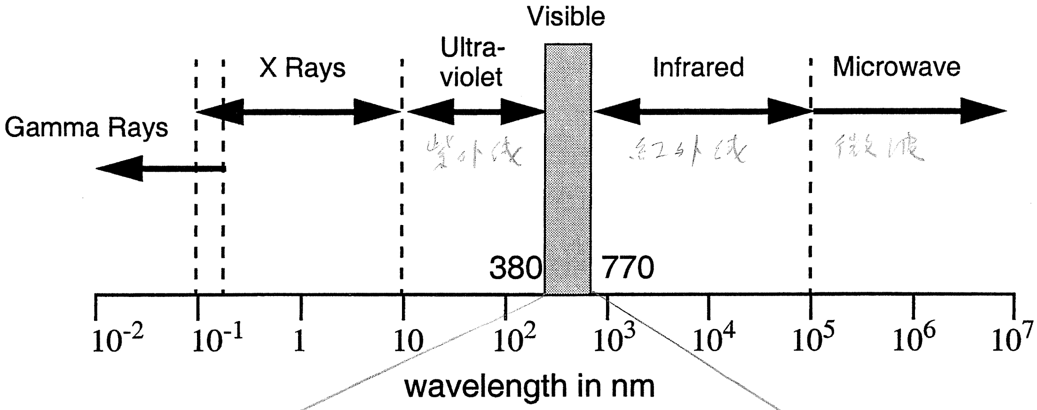


Figure 5-2. Electromagnetic Radiation.

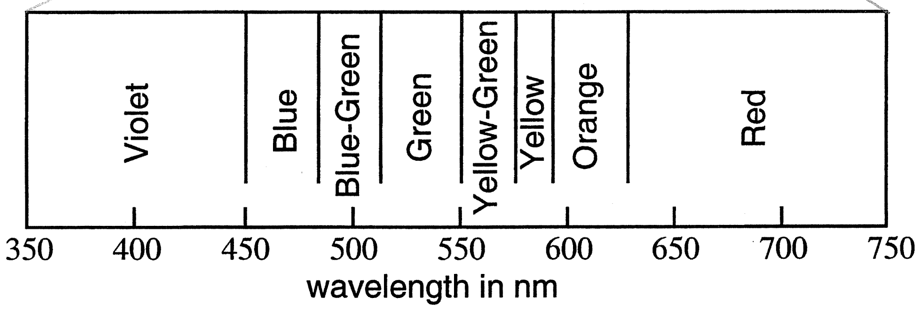


Figure 5-3. Colors in the Visible Light Spectrum.

5.2 Color Perception

Humans perceive color by differentiating the *chromatic* and *achromatic* characteristics of

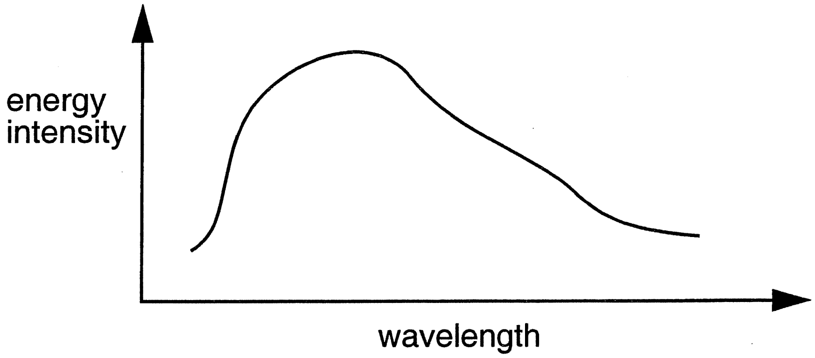


Figure 5-4. Spectral Curve of a Color of Light.

light. The achromatic characteristics are largely determined by the intensity and the chromatic by the wavelength. The differentiation is the result of many factors:

1. the physical characteristics of the object itself and its capacity to reflect or absorb certain wavelengths of light,
2. the properties of the light source illuminating the object,
3. the medium through which the light travels and the distance through the medium,
4. the properties of the surrounding objects or area,
5. the biochemical state of the eye and visual system upon stimulation,
6. the transmission characteristic of the receptor cells and neural centers,
7. the subject's previous experience with the object or sensation.

The first four factors pertain to the physical properties of the objects and their surrounding environment, and can be precisely determined. The last three deal with the physiological characteristics of the human visual system and vary from person to person, even with normal visual systems. For example, individuals will discern different transition points at which an object appears either red or orange, even though the physical situation does not vary. [Cychosz]

The human eye consists of a sensor array of photo-receptors known as the retina, and a lens system under muscle control that focuses the visual stimulation on the retina. The retina consists of two types of sensors: rods and cones. Rods are primarily used for night vision and respond achromatically to a given stimulus. Cones are used for color vision in normal lighting and respond primarily to wavelength.

Color is interpreted by the brain based on the stimuli from three types of cones that are distributed over the retina. Each cone absorbs either reddish light, greenish light or bluish light. The distribution of cones is not uniform. Sixty-four percent of the cones absorb red pigment, thirty two percent absorb green, and only two percent absorb blue. Figure 5-5 shows the relative

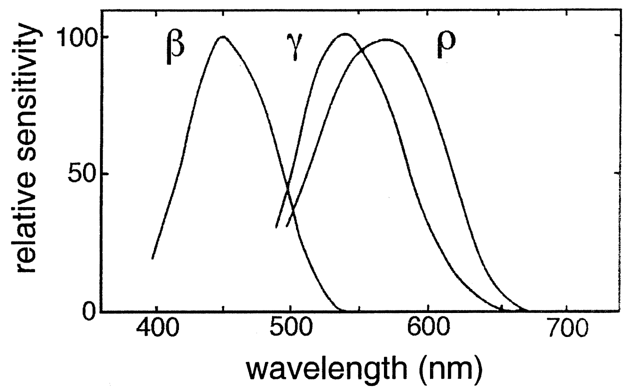


Figure 5-5. Spectral Response Curves for the Three Cone Types (β, γ and ρ).

sensitivities of the three cone types [Hunt 1987, page 13]. The stimulus information sent to the brain through the optic nerve by the receptors conveys a *luminance* (also called lightness, brightness and intensity) and two color ratios: a red-to-green ratio and a yellow-to-blue ratio.

As a result of this experimental data, it is believed that the eye responds primarily to intensities of three color *hues*, red (around 650 nm), green (around 530 nm) and blue (around 460 nm). This suggests that colors that we perceive can be approximated by combining amounts of three *primary colors*: red, green and blue. This is sometimes referred to as the *trichroma* or *tri-stimulus theory*, and dates back to the 1800's. This theory is applied in the design of color display systems and color printing equipment.

5.3 Color CRT Operation

From a programming standpoint, color raster displays function like three gray level display processors (Figure 5-6) whose outputs are passed through red, green or blue filters and focused on a single pixel on the screen. The color of each pixel is defined by intensities of red, green and blue that are sent to red, green and blue CRT guns. Pixel memory can be configured in two ways, called *direct color* and *mapped color*.

In *Direct Color* displays, the pixel value contains the component colors (Figure 5-7). Pixel memory can be configured in different ways depending on the arrangement of the RGB components in computer memory. Some systems represent the RGB components for each pixel

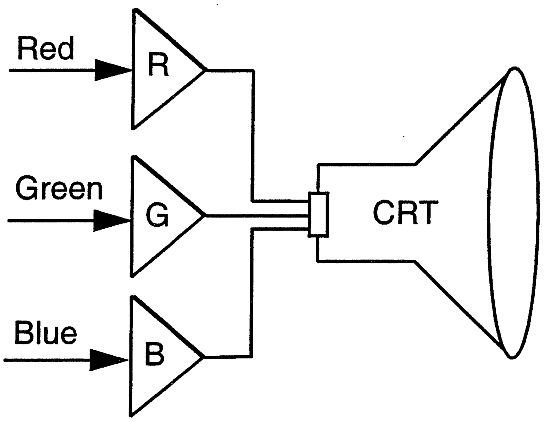


Figure 5-6. Color CRT Schematic.

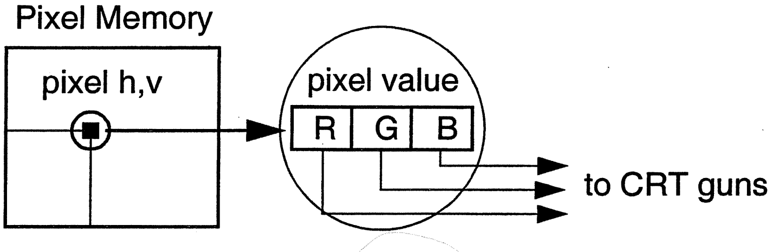


Figure 5-7. Direct Color Display.

within one computer word. Others separate the RGB components into three arrays in memory. In the first method, the RGB components of a pixel are obtained by accessing the pixel value at the appropriate array location in pixel memory and then dissecting the pixel value with binary operations. In the second method, the RGB components are individually accessed from the appropriate arrays.

For example, consider the simplest direct color display with one bit for each color component. Each pixel would have 3 bits, (i.e. "3 bits per pixel"), one for red, one for green, and one for blue. The display would be capable of displaying the 8 colors shown below.

R Component	G Component	B Component	Pixel Value	Color Name
0	0	0	0	Black
0	0	1	1	Blue
0	1	0	2	Green
0	1	1	3	Cyan
1	0	0	4	Red
1	0	1	5	Magenta
1	1	0	6	Yellow
1	1	1	7	White

In *Mapped Color* displays, the pixel value is an index into a separate array of memory in the display called a *Video Lookup Table*, or *VLT*. Each *entry* of the VLT contains color components that are set by the host computer. Each refresh cycle, the raster display processor accesses each

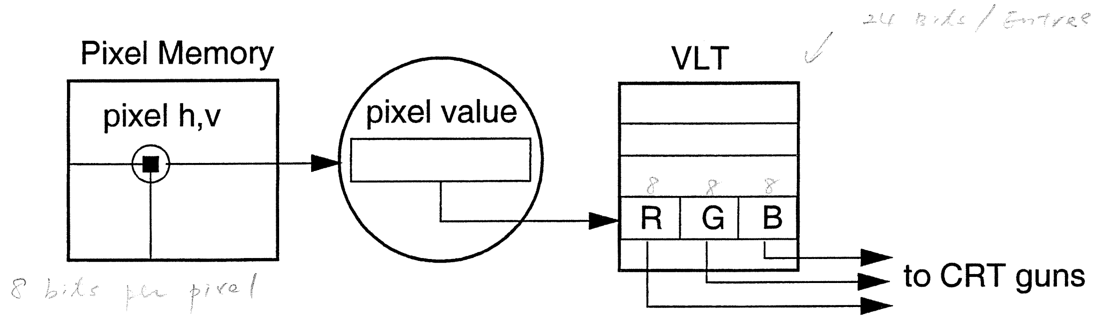


Figure 5-8. Mapped Color Display.

pixel value from pixel memory and uses the RGB components from the VLT location addressed by the pixel value for the pixel color. The VLT is also called a *Color Lookup Table*, or *CLUT*.

Mapped color systems provide a means for reducing the size of pixel memory (and therefore the cost of a system) when it is acceptable to choose a subset of colors from a large *palette* of possible colors. For example, most workstations today with color displays are mapped systems capable of displaying “256 colors chosen from a palette of 2^{24} possible colors.” (The number 2^{24} is over 16.6 million.) To understand the physical configuration of such a display, break down the quoted phrase in the previous sentence. “256 colors” means simultaneous colors on the screen at one time, so pixel memory must be able to store 256 different colors. Therefore, pixel memory must have $\log_2 256$, or 8, bits per pixel (i.e. $2^8 = 256$). “Palette” is a term often used for VLT, so “256 colors” also means that there must be 256 VLT entries that can be indexed by the pixel values. All that remains is the format of each VLT entry. The phrase “palette of 2^{24} possible colors” tells us that each entry must be able to store 2^{24} colors, i.e. each entry must contain 24 bits. The entry must be divided into three components for RGB, so we can safely assume that each VLT entry contains 24 divided by 3, or 8 bits per component. It would be very unusual not to use the same number of bits for each component. Figure 5-9 shows a schematic diagram of the configuration of such a display system.

A direct color system capable of displaying 2^{24} colors would require 24 bits per pixel. This is a significantly more costly system. The difference is sometimes subtle: the mapped system can display only 256 colors at once because each pixel can represent only 256 possible values. Each

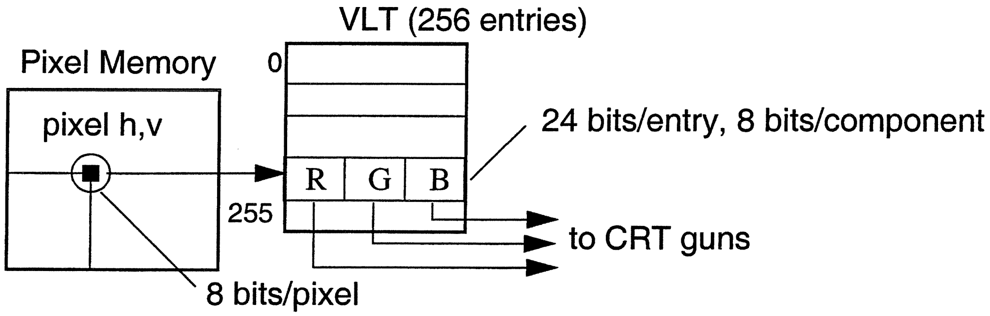
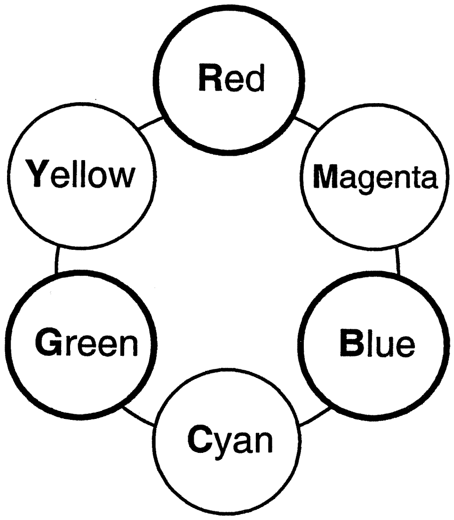


Figure 5-9. Configuration of a 256 Color Mapped Display System.

pixel value is mapped to a VLT entry, however, that can contain 2^{24} possible values (colors). The 24-bit direct system, sometimes called a "full color" system, can display 2^{24} colors at once, but has three times as much pixel memory (24 bits per pixel versus 8 bits per pixel). The additional memory required for the VLT is generally insignificant. In this example, 256 24-bit words is small compared to 512 by 512 or 1024 by 1024 pixels in pixel memory.

5.4 Additive and Subtractive Color

Color raster CRT's transmit light to the eye via illumination, so colors are formed by adding the three primary colors, red, green and blue, called the *additive primaries*. The secondary colors, cyan, magenta and yellow, are formed by combinations of these primaries (Figure 5-10).



G + B = Cyan (sky blue)

R + B = Magenta (purple)

R + G = Yellow (actually reddish green)

R+G+B = white

65535 = "1.0"

GLoad Color

Figure 5-10. Primary and Secondary CRT Colors.

A painter mixes yellow and blue (really cyan) to make green. This does not agree with CRT color, where green is a primary. The difference is that mixing paint and CRT color production involve different physical processes. Consider this from the point of view of light that reaches the eyes. For paint, we see reflected light resulting from incident light (typically white light, light containing all colors) reflecting from the surface of the paint. Physically, light represented by a certain spectral curve irradiates the surface from a certain direction and interacts with it in complex ways. Some of the light may be absorbed by the surface, some may be transmitted through the surface, and some reflects from the surface to our eyes. These interactions alter the spectral properties of the light, changing its spectral curve and therefore its perceived color. The reflected light has different spectral properties (color and intensity) than the incident light.

Consequently, the color of paint is reflected light from which certain amounts of each RGB component from the incident light have been removed (or filtered), thereby changing the light that reaches our eyes. The so-called *subtractive primaries*, cyan, magenta, yellow (CMY), actually filter one trichromatic component (RGB).

For example, consider what color is produced when white light shines through a layer of cyan cellophane and a layer of magenta cellophane (Figure 5-11) The cyan cellophane filters or blocks the red component, and the magenta cellophane blocks the green. Therefore, only blue passes through both layers.

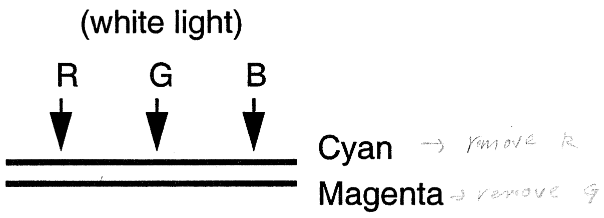


Figure 5-11. Light Transmitted Through Cellophane.

5.5 Color Representations

To this point, we have seen that light is a distribution of energy intensity at all wavelengths in the visible spectrum, that the human eye is a complex sensory system that responds primarily to three colors, and that the CRT produces color by illuminating three colors at given intensities at each pixel. How then can this information be used to represent color for computation and display in a program?

A number of schemes exist for representing color that are based on the trichromatic characteristics of color perception. Generally, these are mathematical and computational algorithms, *color models,* that determine three color parameters in terms of either a spectral curve for a color, or in terms of three color parameters in a different color representation. Some color models can be shown as a three-dimensional volume, the *color space*, in which colors are points defined by the three color parameters, the *color coordinates*.

The color spaces can be classified in three ways: perceptually-based uniform spaces, 知覺 perceptually-based non-uniform spaces, and device-directed spaces. Perceptually-based color spaces provide easy manipulation of color from a qualitative point of view and are useful for developing user-interfaces to specify colors for graphical applications. Uniform color spaces vary color uniformly across the space so that the "distance" between two colors is proportional to the perceived color difference. This is important for accurately simulating physical situations, such as theatrical lighting, and for accurate color comparisons. Device-based color spaces are founded on particular stimulant characteristics of a given device or color presentation medium. These spaces are developed statistically using color matching schemes and, in reality, only approximate a uniform space.

The following sections present a number of popular color spaces: device-based color spaces: RGB, CMY, CMYK, YIQ and XYZ; perceptually-based color spaces: HSI and HSV; and perceptually-based uniform color spaces: Luv and Lab.

5.5.1 RGB Color Space

The red, green and blue color components of the CRT can be viewed as a point in three-dimensional Cartesian space with the coordinates [R,G,B] (Figure 5-12), where the coordinates usually lie in the range [0,1]. The RGB space is the fundamental color representation for all colors that are to be displayed on a CRT. Eventually, colors in all other color spaces must be transformed into RGB coordinates for display.

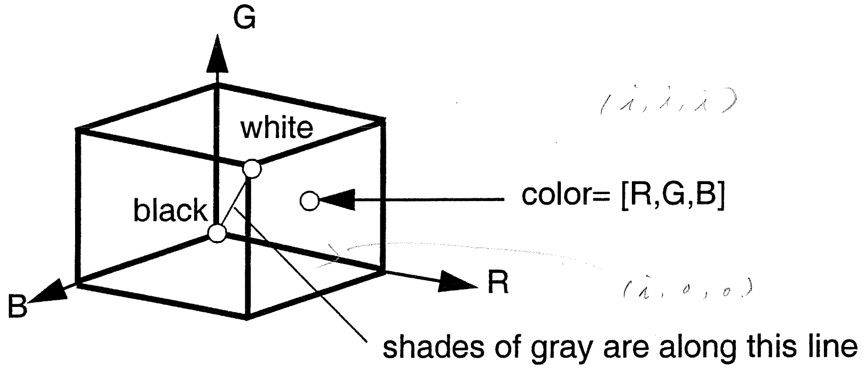


Figure 5-12. RGB Color Space.

5.5.2 CMY and CMYK Color Spaces

Color reproduction on hardcopy devices, such as printing, requires all colors to be represented as intensities of cyan, magenta and yellow. The conversion equations from RGB to CMY color coordinates are simple linear transformations:

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

and for CMY to RGB:

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

As usual, these equations assume the coordinates are real numbers in the range [0,1]. The CMY color space is shown in Figure 5-13.

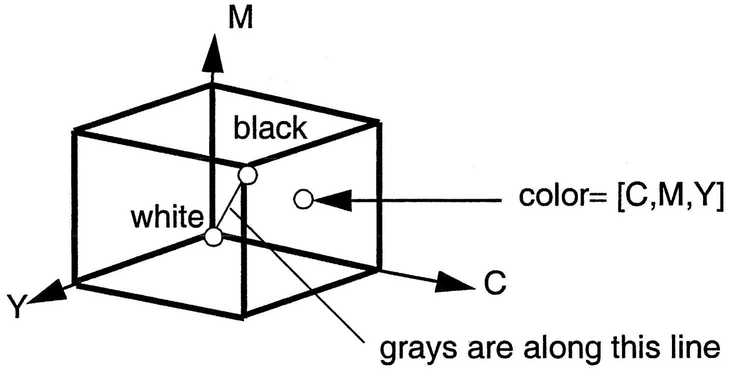


Figure 5-13. CMY Color Space.

As an example, consider how to produce Cyan on a CRT. The CRT requires color in RGB, so we must compute the RGB coordinates from the CMY. Cyan is [1,0,0] in CMY space.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

The color Cyan in RGB coordinates is [0,1,1], 1.0 intensity Green and 1.0 intensity Blue.

Now consider how to print Red. For this, we must compute the CMY coordinates from the RGB. Red is [1,0,0] in RGB space,

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

The color Red in CMY coordinates is [0,1,1], 1.0 intensity Magenta and 1.0 intensity Yellow.

Black is produced by equal amounts of CMY, which means combining equal amounts of three paints or three inks. In practice, this results in an unpleasant dark-brown color as well as consuming ink (or paint) from all three colors. Some printing devices include a black ink cartridge and consider black as a separate color. This is the CMYK color space, where 'K' stands for black. Unfortunately, it is a four-dimensional space that is not readily illustrated.

The conversion formulae for CMY to CMYK are [Foley]:

$$K = \min(C,M,Y) \qquad \begin{bmatrix} C \\ M \\ Y \\ K \end{bmatrix} = \begin{bmatrix} C \\ M \\ Y \\ 0 \end{bmatrix} + \begin{bmatrix} -K \\ -K \\ -K \\ K \end{bmatrix}$$

and for CMYK to CMY:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} C \\ M \\ Y \end{bmatrix} + \begin{bmatrix} K \\ K \\ K \end{bmatrix}$$

It is necessary to go through CMY to convert between CMYK and RGB.

5.5.3 YIQ Color Space

In 1953, the NTSC (National Television Standards Committee) recommended a set of transmission primary colors, known as YIQ, as a basis for the broadcast color television signal

[Smith]. This is basically an adjustment to the standard RGB space that was considered to be more appropriate for television transmission and viewing. Like many color models, it is related to the RGB space through a linear transformation that is compactly represented in the matrix equation:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

The conversion from YIQ to RGB involves the inverse of the matrix:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.620 \\ 1 & -0.272 & -0.647 \\ 1 & -1.108 & 1.705 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

The Y coordinate is the luminance, the brightness perceived by the viewer, and is the only signal received by a black-and-white television. The I and Q components are two chrominance values that determine the color of a pixel. They are derived from the modulation technique used in NTSC broadcast signals.

5.5.4 XYZ Color Space

In 1931, the *Commission Internationale de l’ E´clairage* (CIE) defined three standard primary colors, called X, Y and Z, as alternatives to R, G and B for color matching. The XYZ coordinates are based on three spectral matching functions, X(λ), Y(λ), and Z(λ) shown in Figure 5-14 that were developed based on color matching experiments and desired mathematical

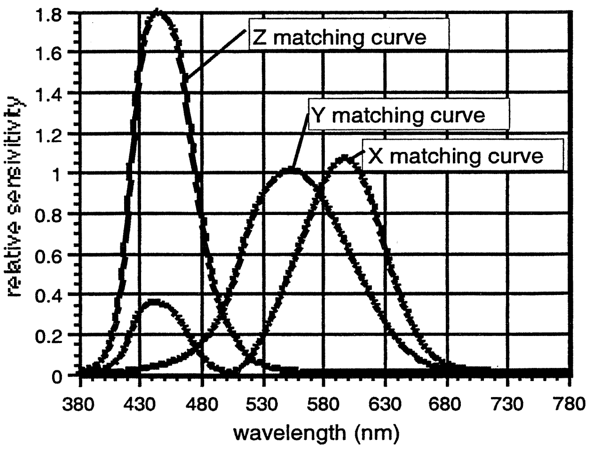


Figure 5-14. CIE Spectral Matching Curves.

properties. The Y(λ) curve matches the luminance response curve of the human eye, i.e. the eye's sensitivity to spectral wavelengths. These color matching functions are tabulated at 1 nm wavelength intervals [Wyszecki and Hall].

The CIE coordinates, [X,Y,Z], are computed from the spectral curve of a given color, I(λ), as integrals

$$X = k\int I(\lambda)X(\lambda)d\lambda \qquad Y = k\int I(\lambda)Y(\lambda)d\lambda \qquad Z = k\int I(\lambda)Z(\lambda)d\lambda$$

where k is a scaling constant appropriate for the lighting situation [Foley, p. 580]. In practice, I(λ) is also tabulated at 1 nm intervals and the integration is done numerically by summation.

The XYZ coordinates include luminance information in addition to chroma. To remove the luminance factor, XYZ coordinates can be scaled to normalized values, i.e. each coordinate is divided by the sum X+Y+Z, resulting in xyz coordinates (lower case), called *chromaticities*. Note that x+y+z = 1 by definition, so only the x and y coordinates are needed. The two dimensional plot of these [x,y] values is called a *chromaticity diagram* and bounds the colors in the XYZ color space. The CIE xyz chromaticity diagram is shown in Figure 5-5 .

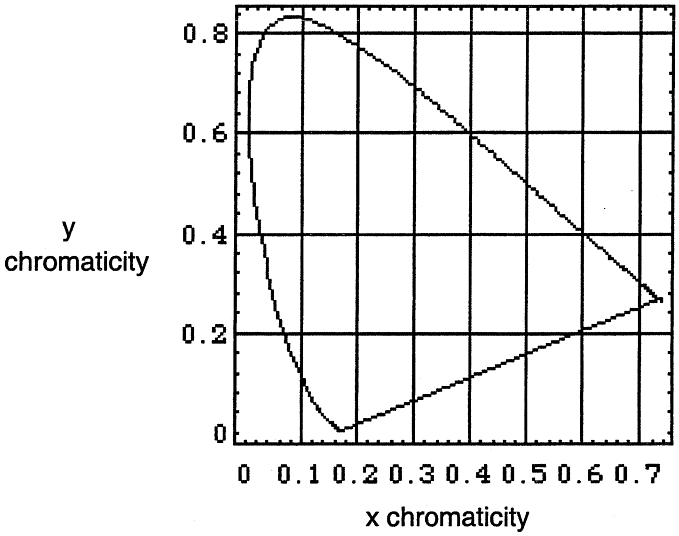


Figure 5-15. CIE xyz Chromaticity Diagram.

The chromaticities of a color can be measured with an incident light chromaticity meter. This can be done for a color monitor to calibrate its red, green, blue and white colors for precise color presentation. From the measured chromaticities it is possible to compute the transformation between RGB to XYZ. Mathematically, the chromaticities are the [x,y,z] values corresponding to the RGB primaries (red=[1,0,0], etc.). These reference colors are solved to form the transformation of the form

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} \mathrm{XYZtoRGB} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

where XYZtoRGB is a 3 by 3 transformation. White is used to scale the values. The resulting transformation relates the monitor's RGB values and the standard XYZ coordinates.

The NTSC provides a standard set of [x,y,z] chromaticities: red = [0.67, 0.33, 0.0], green = [0.21, 0.71, 0.08], blue = [0.14,0.08,0.78] and white = [0.313, 0.329, 0.358]. The resulting transformations from XYZ to RGB are,

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.967 & -0.548 & -0.297 \\ -0.955 & 1.938 & -0.027 \\ 0.064 & -0.130 & 0.982 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

[0.33, 0.33, 0.33]

and from RGB to XYZ:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.589 & 0.179 & 0.183 \\ 0.290 & 0.605 & 0.104 \\ 0 & 0.068 & 1.020 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

5.5.5 HSI Color Space

To produce several shades of blue ranging from light blue (almost white) to dark blue (almost black) is difficult using the RGB, CMY, YIQ or XYZ device-based color spaces. Perceptually-based colors spaces were developed to provide a more intuitive way to define colors for graphics programming. Perceptually-based color spaces use hue, saturation, and intensity (or brightness, lightness, value) as color coordinates.

Hue is the basic component of color and is responsible for the perceived color sensation. It is directly related to the dominant wavelengths of the given color. It is important to keep in mind that wavelengths are a physical property and that color is a psychological effect. All color perceived is a result of the stimulation of the photo-receptors in the visual system. Thus, the presence of reddish and greenish wavelengths of light does not produce yellowish wavelengths light, even though the resulting light would be perceived as yellowish because of the way individual sources stimulate the retina.

Saturation and *chroma* describe the purity of the color, or its *colorfulness*. The greater the presence of achromatic light (light with a horizontal spectral curve), the less saturated a color becomes. As saturation decreases, the dominant hue becomes more difficult to discern. Chroma is the colorfulness compared to white or the level of illumination. Gray is unsaturated, whereas pure monochromatic red of low intensity has high saturation and low chroma. Saturation generally is independent of intensity, whereas chroma depends on intensity.

Lightness and *Brightness* are often used in describing color. Brightness varies from black to bright, representing the strength of the stimulation as if the stimulant is self-luminous. Lightness, on the other hand, varies from black to white representing the reflected intensity of a diffuse surface that is externally illuminated. Thus, as the intensity of the illuminating source increases, the color of the surface becomes de-saturated as it approaches white in color. Here, we use intensity for lightness.

The hue-saturation-intensity (HSI) color space, also called hue-saturation-lightness (HLS), is a popular perceptually-based color space that is a double cone with either flat sides (called a "hexcone" as shown in Figure 5-16) or as a rounded double inverted cone with curved sides. Colors

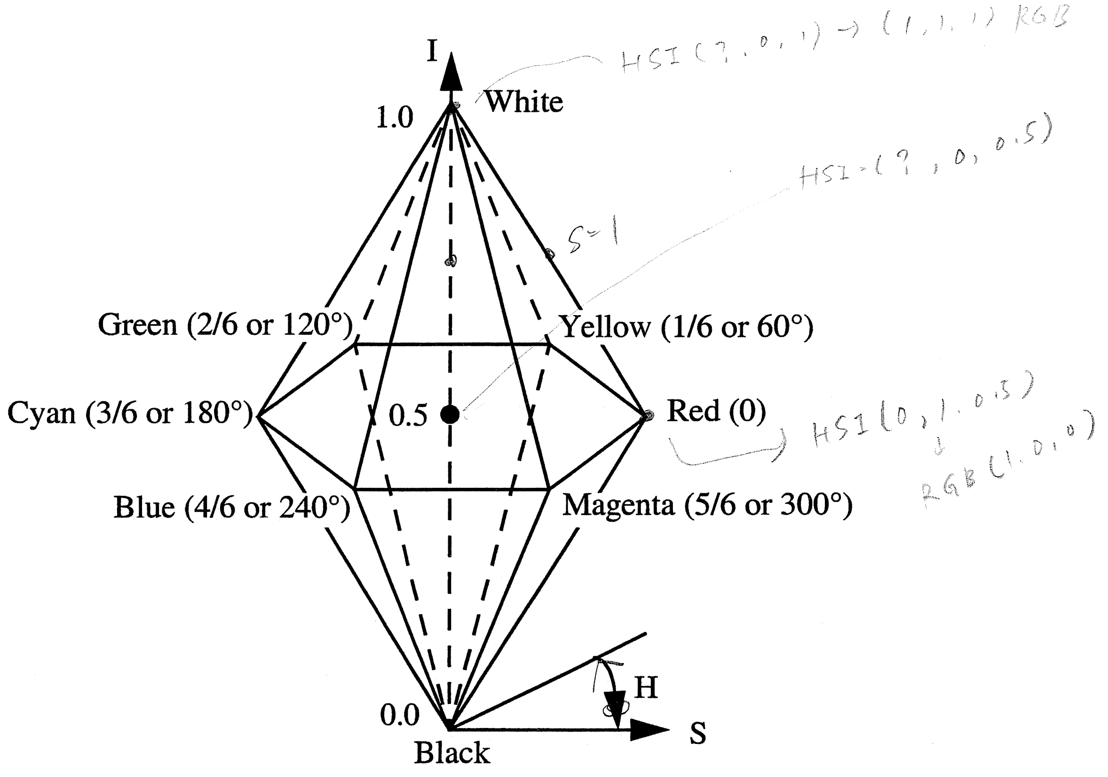


Figure 5-16. HSI Color Space.

are points within the volume of the double cone. Some representations draw the space as a cylinder with black on the bottom surface and white on the top surface. Think of slicing the color cone at a given intensity, producing a two-dimensional color wheel of hues varying around and saturations varying radially.

Hue (H) is the angular coordinate. It is represented by the angle around the equator of the cone, normalized to lie in the range [0,1] or [0,360]. Note that this does not follow physical wavelengths exactly. There is no reddish-blue in the electromagnetic spectrum.

Saturation (S) is the radial coordinate. Full (1.0) saturation is what one might describe as a "deep" color. Colors with zero saturation, shades of gray, are located along the axis connecting the tips of the cone. Full saturation is on the outer surface of the cone.

Intensity (I) is the vertical coordinate. Zero intensity of any hue and saturation is black. One-half intensity is along the equator. Full (1.0) intensity of any saturation and hue is white. The "pure" colors (fully saturated, without achromatic light) lie in the plane at half intensity.

A color given in HSI coordinates is converted to RGB coordinates in a two step process. The first step is to compute the relative RGB values, [R´,G´,B´], using only the hue (H) coordinate. This is similar in nature to finding "chromaticity ratios" without regard to luminance.

These values can be represented as periodic functions that cycle through the visible light spectrum as approximated by the RGB primaries. The linear versions of the functions, corresponding to the hexcone, are tabulated below and are graphed in Figure 5-17.

H	R´	G´	B´
0 to $\frac{1}{6}$	1	6H	0
$\frac{1}{6}$ to $\frac{2}{6}$	2-6H	1	0
$\frac{2}{6}$ to $\frac{3}{6}$	0	1	6H-2
$\frac{3}{6}$ to $\frac{4}{6}$	0	4-6H	1
$\frac{4}{6}$ to $\frac{5}{6}$	6H-4	0	1
$\frac{5}{6}$ to 1	1	0	6-6H
$\frac{1}{6}$ to $\frac{2}{6}$	2-6H	1	0

R'G'B' are converted to RGB as follows:

For $0 \leq I \leq 0.5$: $RGB = 2I(0.5 - S(R'G'B' - 0.5))$

For $0.5 < I \leq 1$: $RGB = 0.5 + S(R'G'B' - 0.5) + (2I - 1)(0.5 - S(R'G'B' - 0.5))$

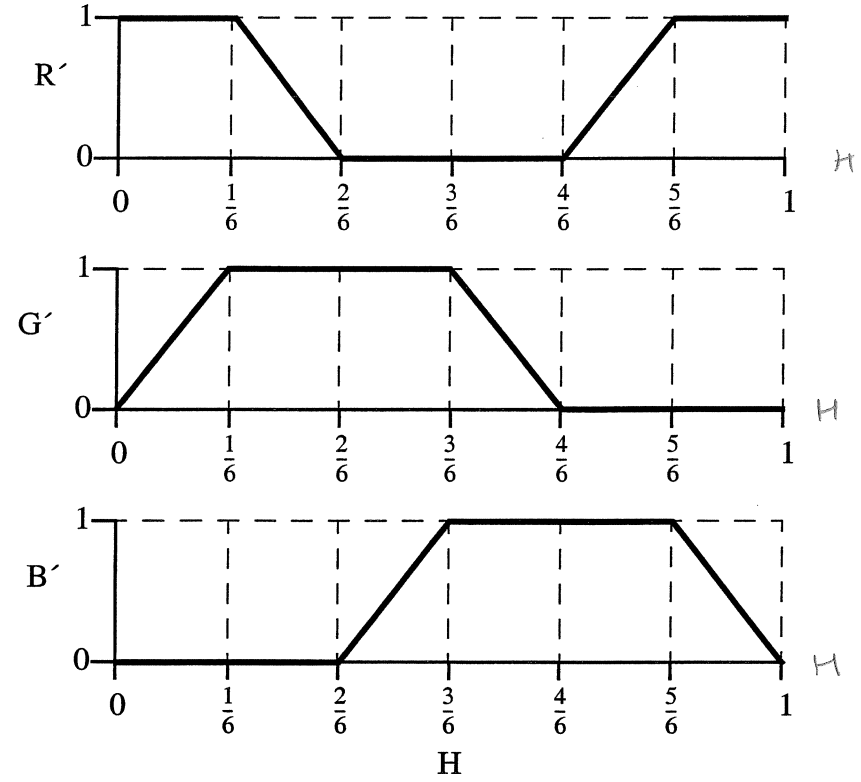


Figure 5-17. R′G′B′ Ratio Functions.

As an example, consider creating realistic images of objects by simulating of the effects of light reflecting from surfaces. White light reflecting from an object of a certain color produces different shades of that color. The problem, then, is to produce a range of shades of a given color, i.e. different intensities of a basic hue and saturation. HSI is a convenient color space for these computations. Given the hue and saturation of the basic color, vary the intensity (I) between 0 and 1 and produce an RGB for each. For example, the following table shows five shades of the color red.

	I	R	G	B
1	0.00	0.0	0.0	0.0
2	0.25	0.5	0.0	0.0
3	0.50	1.0	0.0	0.0
4	0.75	1.0	0.5	0.5
5	1.00	1.0	1.0	1.0

Examine the color paths of through the five colors in the HSI and RGB color spaces (Figure

5-18).

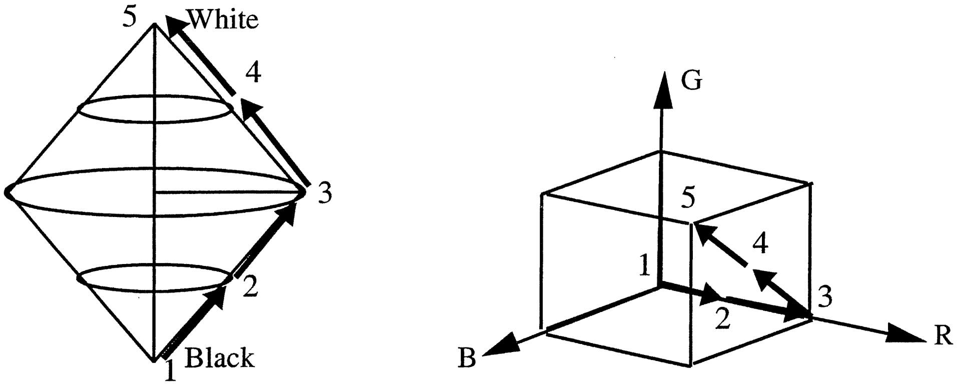


Figure 5-18. Spectrum Colors in HSI and RGB Spaces.

5.5.6 HSV Color Space

The Hue-Saturation-Value (HSV) space, also called Hue-Saturation-Brightness (HSB), is similar to the HSI color space. Geometrically, there is a single cone and the center of the top circle represents white (Figure 5-19). A color given in HSV coordinates is converted to RGB coordinates

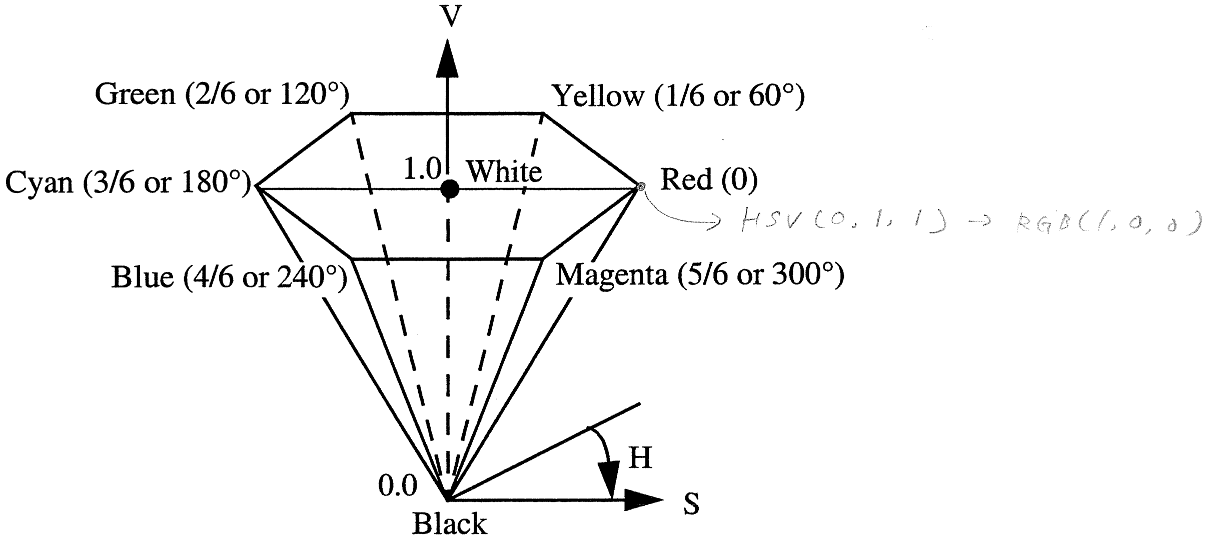


Figure 5-19. HSV Color Space.

in a two step process. The first step is to compute the R′G′B′ ratios as described previously for the

HSI space. Given the R′G′B′ ratios, RGB coordinates are computed as follows:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = V\left(\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + S\left(\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\right)\right)$$

5.5.7 Luv Color Space

The Luv space is a uniform color space developed by the CIE that is perceptually linear and is useful for color comparison. It relates to the XYZ space through the following non-linear equations [Hall p. 55]:

$$L = 116\left(\frac{Y}{Y_0}\right)^{1/3} - 16$$

$$u = 13L(u' - u'_0)$$

$$v = 13L(v' - v'_0)$$

where

$$u' = \frac{4X}{X + 15Y + 3Z} \qquad v' = \frac{9Y}{X + 15Y + 3Z}$$

$$u'_0 = \frac{4X_0}{X_0 + 15Y_0 + 3Z_0} \qquad v'_0 = \frac{9Y_0}{X_0 + 15Y_0 + 3Z_0}$$

$[X_0, Y_0, Z_0]$ is the color of the white reference point

The equations assume reference white has been normalized such that $Y_0 = 1$. The NTSC primaries can be used for standard monitors. Equal steps in L represent equal steps in the perceived lightness of related colors.

These equations can be solved for the conversion equations from CIE Luv to CIE XYZ:

$$Y = \frac{Y_0(16+L)^3}{1560896}$$

$$X = \frac{9(u+13Lu'_0)Y}{4(v+13Lv'_0)}$$

$$Z = \frac{Y(156L-3u-39Lu'_0-20v-260Lv'_0)}{4(v+13Lv'_0)}$$

Note that the order of the equations is important (X and Z depend on the value of Y). It is necessary to convert between the CIE XYZ coordinates and RGB coordinates to display CIE Luv colors.

5.5.8 Lab Color Space

The Lab space is perceptually-based uniform color space developed by the CIE that is a mathematical approximation to the well-known Munsell color system. The Munsell color system is based on a discrete set of equally spaced color samples. These samples are represented as discrete volumes and arranged as a cylindrical volume of varying radius. The Munsell hue is defined as the angle about the axis of the cylinder. The Munsell chroma (i.e., saturation) is defined as the radial distance and the Munsell value (i.e., lightness) is defined along the axis. The Munsell book consists of paint samples [Munsell] that is used to solve color matching problems. The Munsell color solid is illustrated in Figure 5-20.

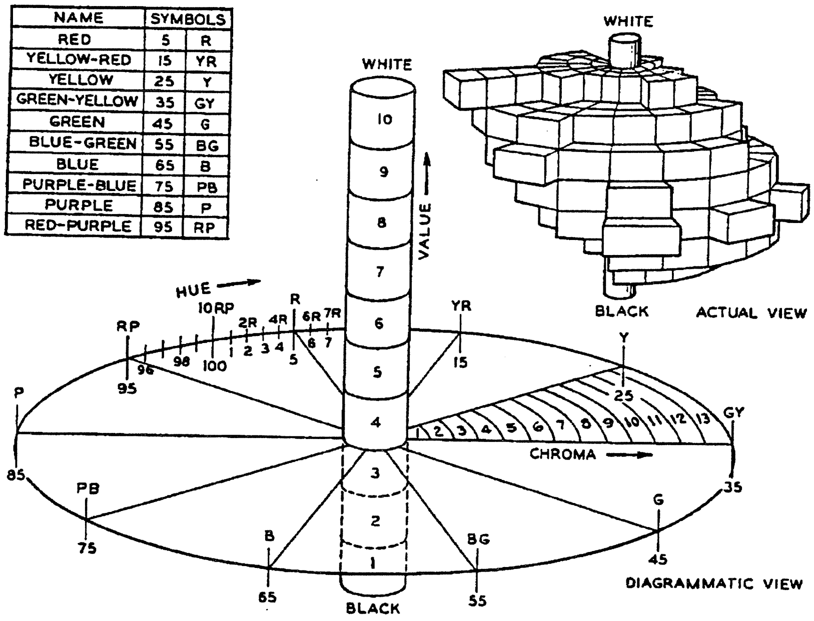
The XYZ to Lab equations are [Hall, p. 55]:

$$L = 116\left(\frac{Y}{Y_0}\right)^{1/3} - 16$$

$$a = 500\left(\left(\frac{X}{X_0}\right)^{1/3} - \left(\frac{Y}{Y_0}\right)^{1/3}\right)$$

$$b = 200\left(\left(\frac{Y}{Y_0}\right)^{1/3} - \left(\frac{Z}{Z_0}\right)^{1/3}\right)$$

where $[X_0,Y_0,Z_0]$ are the XYZ coordinates of the white reference point.



NAME	SYMBOLS	
RED	5	R
YELLOW-RED	15	YR
YELLOW	25	Y
GREEN-YELLOW	35	GY
GREEN	45	G
BLUE-GREEN	55	BG
BLUE	65	B
PURPLE-BLUE	75	PB
PURPLE	85	P
RED-PURPLE	95	RP

Figure 5-20. Munsell Color System [Williams, p. 19].

The CIE Lab to CIE XYZ equations are:

$$Y = \frac{Y_0(16 + L)^3}{1560896}$$

$$X = \frac{X_0\left(a^3 + 1500a^2\left(\frac{Y}{Y_0}\right)^{1/3} + 750000a\left(\frac{Y}{Y_0}\right)^{2/3} + 500^3\left(\frac{Y}{Y_0}\right)\right)}{500^3}$$

$$Z = \frac{Z_0\left(-b^3 + 600b^2\left(\frac{Y}{Y_0}\right)^{1/3} - 120000b\left(\frac{Y}{Y_0}\right)^{2/3} + 200^3\left(\frac{Y}{Y_0}\right)\right)}{200^3}$$

5.6 References

1. Cychosz, J.M., "The Perception of Color and Representative Space," Purdue CADLAB Technical Report No. JC-05, December 12, 1992.

2. Hall, R., *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, New York, New York, 1989.

3. Hunt, R. W. G., *The Reproduction of Colour in Photography, Printing and Television*, Fourth Edition, Fountain Press, Tolworth, England, 1987.

4. Incropera, F. P. and D. P. DeWitt, *Fundamentals of Heat and Mass Transfer*, Third Edition, John Wiley & Sons, New York, 1990.

5. Joblove, G. H. and D. Greenberg, "Color Spaces for Computer Graphics", Computer Graphics (SIGGRAPH '78 Proceedings), Vol. 12 No. 3, August 1978.

6. Munsell, A. H., A Color Notation, 10th ed., Baltimore, Md., 1946.

7. Murdoch, J. B., Illumination Engineering - From Edison's Lamp to the Laser, Macmillan Publishing Company, New York, 1985.

8. Murch, G. M., "Physiological Principles for the Effective Use of Color", IEEE Computer Graphics & Applications, November 1984, pp. 49-54.

9. Smith, A. R., "Color Gamut Transform Pairs", Computer Graphics (SIGGRAPH 78 Proceedings), Vol. 12 No. 3, August 1978.

10. Williams, R. G., Lighting for Color and Form, Pitman Publishing Corporation, New York, 1954.

11. Wyszecki, G. and W. Stiles, Color Science: Concepts and Methods, Quantitative Data and Formulae, second edition, Wiley, New York, 1982.

5.7 Review Questions

1. Draw the RGB and HSI color spaces. Label the locations and give the coordinates in each space of red, green, blue, black and white. Label the location and give the HSI coordinates of the color 50% brighter than pure red.
2. Given the normalized (0.0 to 1.0) RGB color [0.1, 0.2, 0.3], explain how to create the same color by mixing paints. Show necessary computations.
3. Show in step by step form how to compute an N color RGB spectrum from black to white of shades of a color of a given hue (H) and saturation (S).
4. Explain how colors are produced by a CRT versus how colors are produced by an ink ribbon printer, from the point of view of "light reaching the eyes." Explain how to create the color Yellow on a CRT and a printer.

Chapter 6. Vector Display Systems

We have seen that programming for a raster display processor is a matter of setting pixels in a frame buffer. Creating images for a *vector display processor*, on the other hand, is a very different matter. We now return to our discussion of display processors, this time to discuss the vector display processing unit.

As functionality is added to the vector DPU, its local intelligence increases and, in effect, it becomes a computer whose main function is controlling the CRT beam. Similar to computers used for computation, the DPU executes a repertoire of instructions. However, DPU instructions, called *display instructions*, are for drawing lines and characters, controlling refresh, and communicating with a host computer.

Also like a computer, to have the DPU do anything one must program it. In this case, it is called *display programming*.

6.1 Display Programming

Assume we have taken delivery of a new vector display system, that is "driven" from a host computer by instructions that are sent over a high-speed interface. The DPU, then, simply responds to instructions it receives from the host, one at a time.

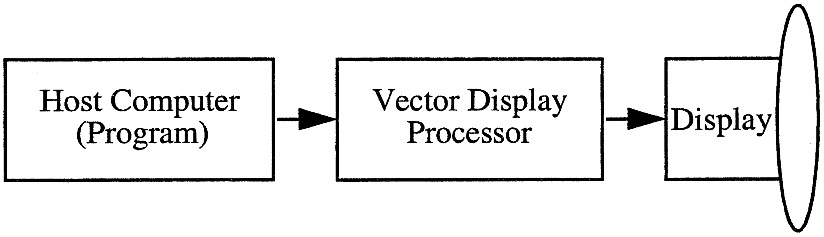


Figure 6-1. Vector Display Configuration.

We study its programming manual and learn that the screen coordinates range from -2048 to +2047 in x and y, with [0,0] at the screen center. The instructions must be sent as binary data to the DPU, but for convenience to humans they have been given symbolic forms, or *mnemonics*.

MOVEA x,y	move to the absolute 2D point (x,y)
MOVER dx,dy	move by dx and dy

DRAWA x,y	draw to the absolute point (x,y)
DRAWR dx,dy	draw by dx and dy

Now we must create the display program that will draw the desired picture when sent to the DPU, in this case a square 100 units on a side and located at [100,100].

	Instruction	Comment
1	MOVEA 100,100 ;	Move to (100,100)
2	DRAWR 100,0 ;	Draw by (100,0)
3	DRAWR 0,100 ;	Draw by (0,100)
4	DRAWR -100,0 ;	Draw by (-100,0)
5	DRAWR 0,-100 ;	Draw by (0,-100)

Figure 6-2. Display Instructions to Draw a 100 Unit Square.

6.1.1 Driving A Display without Memory

Next we manually translate ("assemble") the display instruction mnemonics into a list of binary data. (A program is usually used to convert these statement into binary instructions.) The problem now is to transfer the display instructions to the DPU to make the image appear. Support software in the host computer must be available to perform this data communication. We help a systems programmer develop these routines according to our specifications. For additional convenience, they should be callable from a high level language like C.

```
void ToDisp( int array[], int nwords );
```

sends `nwords` of display instructions to the display from the `array`.

Next we code a simple program to send this data to the display, a "display driving program" (Figure 6-3). The beam movement on the display screen as this program is "executed" is shown in

```
main()
{
     static int dlist[5] = {
     /*   list of integers corresponding to the */
     /*   display instructions in Figure 6-2 */
     };
     ToDisp( dlist, 5 );
}
```

Figure 6-3. First Display Driving Program.

Figure 6-4.

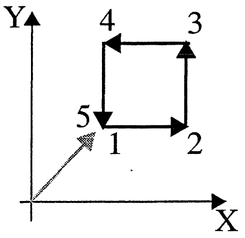


Figure 6-4. Beam Movement for First Display Program.

What happens when the program ends? The display driver program sent the display instructions once, so the beam followed the path, causing the screen phosphor to glow along the path. However, as we have seen, the display fades quickly away in less than 1/30 second.

It is necessary to *refresh* the display as long as the image should appear. The program must be edited to re-send the data periodically to correct this problem (Figure 6-5).

```
main()
{
     static int dlist[5] = {
     /*   list of integers corresponding to the */
     /*   display instructions in Figure 6-2 */
     };
     /* enter an infinite loop to periodically re-send data */
     /* to the display */
     for( ;; ) {
          ToDisp( dlist, 5 );
          /* compute */
          WaitForNextRefreshCycle();
     }
}
```

Figure 6-5. Second Display Driving Program with Refresh Loop.

There are different refresh methods for displays. What we just have seen is "*on-line execution from the host.* It is certainly the least expensive DPU we can buy, but the *refresh burden* is very demanding on our host computer. Consider also what happens if we add computations in the previous program at the comment. If these computations take too long, i.e. require more time than is left in the refresh cycle after the display instructions have been sent by ToDisp, then we again are faced with flicker.

To relieve the host computer of its *refresh burden*, newer systems contain their own memory to store the display list. They do *local refresh* from this *display memory*, thus freeing the host computer to do other things (like compute).

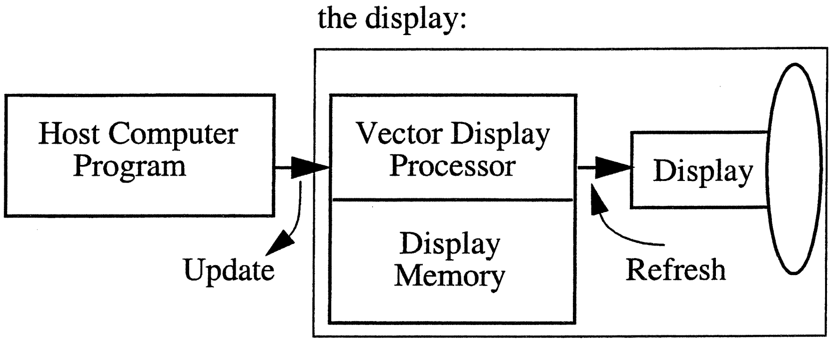


Figure 6-6. Vector Display with Memory.

The UPDATE arrow shows the interface carrying display instructions sent by the host computer to the display when the picture changes. The REFRESH arrow indicates the display instructions being executed from the DPU memory each refresh cycle to refresh the screen image, even if the picture has not changed. Thus, after sending the initial display program, the host does not have to re-send data if the picture remains the same.

Display memory adds functionality to the DPU that is accessed using some new instructions:

DHALT	stop the display and wait to restart
DJSR addr	execute a display subroutine call
DRET	return from a display subroutine call

As one example, the DPU refreshes the display from instructions stored in the display memory as follows:

1. The DPU starts at display memory location 1.
2. instructions are read from sequential display memory locations and executed until a DHALT is executed.
3. After the DHALT, the DPU waits until the next refresh cycle time, then returns to step 1.

Now we write another display program, this time to draw two squares, starting at display memory location 1 (Figure 6-7).

```
Display
Address      Instruction        Comment
1    DLIST:  MOVEA 100,100      ; Move to (100,100)
2            DJSR   BOX         ; Draw the Box
3            MOVEA 500,500      ; Move to (500,500)
4            DJSR   BOX         ; Draw another box
5            DHALT              ; Stop DPU
6    BOX:    DRAWR 100,0        ; Draw by (100,0)
7            DRAWR 0,100        ; Draw by (0,100)
8            DRAWR -100,0       ; Draw by (-100,0)
9            DRAWR 0,-100       ; Draw by (0,-100)
10           DRET               ; Return
```

Figure 6-7. Two Squares Display Program.

6.1.2 Driving A Display with Memory

As before, we manually translate the display instruction mnemonics into a list of binary data. We modify the support software in the host to perform this new data communication:

```
ToDisp ( int array[], int nwords, int daddr );
```

send nwords of display instructions to the display, loading them from array and storing them starting at display address daddr. The *beam movement* for this program is shown in Figure 6-9.

```
main()
{
     static int dlist[10] = {
     /*   list of integers corresponding to the */
     /*   display instructions in Figure 6-7 */
     };
     ToDisp( dlist, 10, 1);
}
```

Figure 6-8. Two Squares Display Driver.

What happens when the program terminates? The boxes remain displayed because the display is performing the refresh of the display instructions from its own memory.

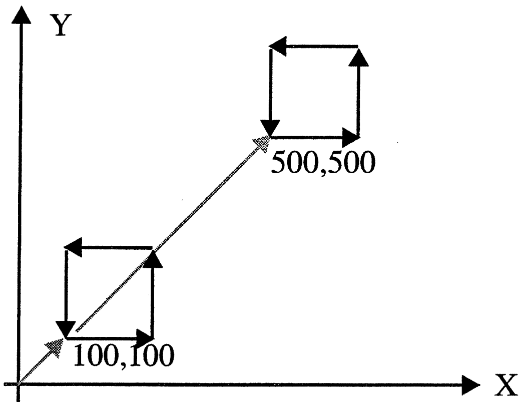


Figure 6-9. Beam Movement for Two Squares Display Program.

6.2 Dynamic Display Manipulation

Expand this driving program to interact with a device, in this case an analog-to-digital converter that is interfaced to the same computer as the display.

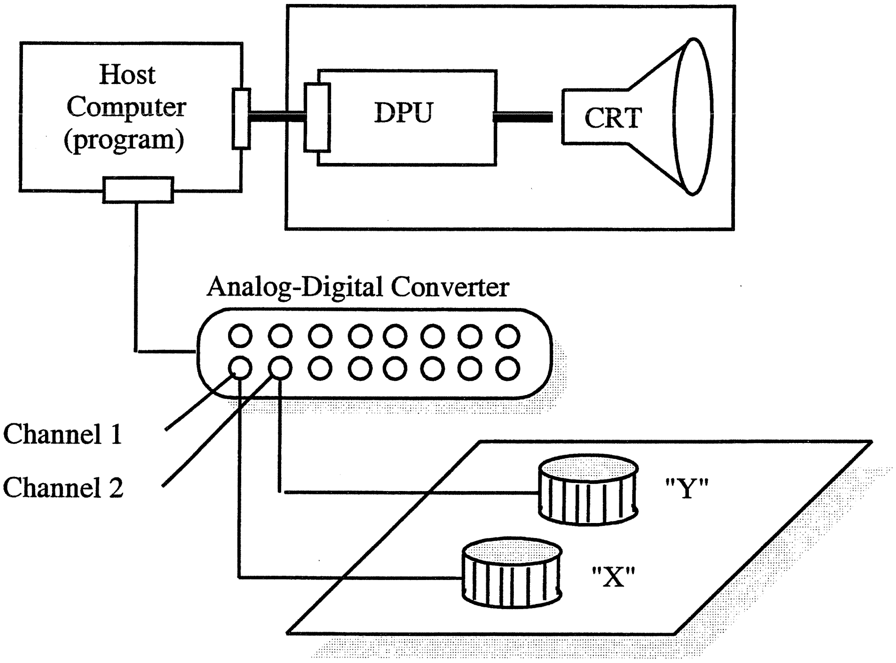


Figure 6-10. Hardware for Dynamic Display Manipulation.

Our interface routines, developed as the other, are:

```
     int ReadVolt( int ichan );
```

returns a numerical value that is the current voltage read from input channel `ichan`.

```
int MakeMove( int x, int y );
```

returns a "MOVEA x,y” display instruction given an x and a y coordinate.

The new driving program is shown in Figure 6-11.

```
main()
{
     static int dlist[10] = { /* data as before */ };
     int x, y;
     ToDisp( dlist, 10, 0 );
     for(;;) {
          x = ReadVolt( 1 );
          y = ReadVolt( 2 );
          /* Note: C arrays start at index 0 */
          dlist[0] = MakeMove( x, y );
          ToDisp( dlist, 1, 1 );
     }
}
```

Figure 6-11. Display Driver for Dynamic Display Manipulation.

What happens when the potentiometers are moved? The first box moves, because the program over-writes the MOVEA instruction at location 1 in display memory with a new instruction that was made by the MakeMove function from the x and y values read from the potentiometers.

Note that the display list is being changed as the DPU executes it. This is *dynamic display manipulation.*

One can now think of many other desirable functions: changing the size of the box, adding new boxes, deleting boxes “pointed to” by a “cursor” that we move with the potentiometers, etc. This involves the development of a graphics package of routines which provide a variety of functions:

1. manage display memory,
2. provide drawing functions for creating entities like lines, characters, etc.
3. provide logical organization of the display to permit adding, deleting, and other operations on groups of entities,
4. provide input functions for obtaining screen positions from devices.

6.3 Display Processors

There are many different types of raster and vector display processors. We will examine two generic configurations. First, introduce the elements of display systems:

CPU	central processing unit (hardware used to compute),
CPU memory	the program & the data,
DPU	display processing unit (hardware used to display),
DPU memory	the display list,
refresh	data path where the DPU executes display list,
update	data path where the host computer changes the display list.

Case 1: Remote Display Processor with Local Memory

This system was discussed earlier during the driving program examples.

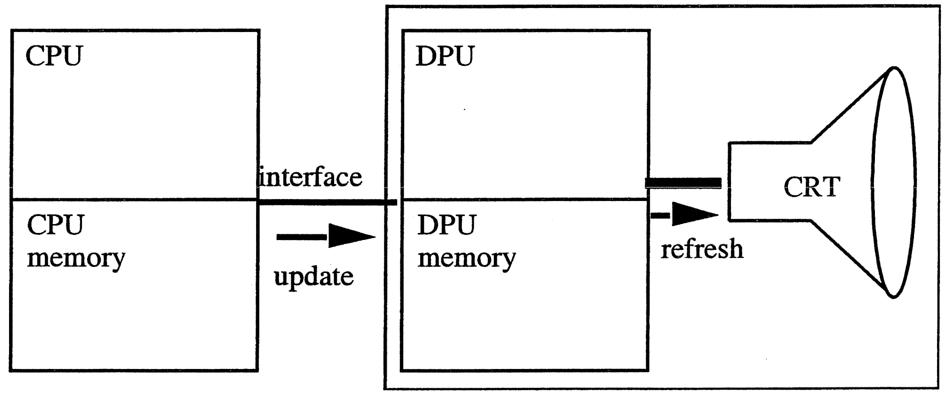


Figure 6-12. Remote DPU with Memory.

Generally, this display processor can be a raster DPU or a random vector DPU.

As a Random Vector DPU:

DPU memory consists of structured display commands, a display list.

As a Raster DPU:

DPU memory is unstructured pixel memory. In some cases, in addition to controlling the beam to display pixels, the DPU can also act as a general purpose computer, accepting "structured commands" from the host, such as "line" and "character". These commands are simply translated into operations on pixel memory. Their meaning is lost in DPU memory.

Case 2: The Raster Refresh Buffer

This is an interesting variation on a raster display that incorporates a computer into the display to simulate structured display operations. The DPU display list is maintained by DPU #1, an internal computer, which interprets structured instructions from the host, stores them in DPU memory #1 as a structured display list, and translates them into pixels in DPU memory #2, pixel memory. DPU #2 is a simple raster DPU that refreshes the screen from pixel memory. The translation process is termed repaint. The host computer can draw lines and then "erase" them. This causes DPU #1 to edit its display list, and then repaint pixel memory accordingly.

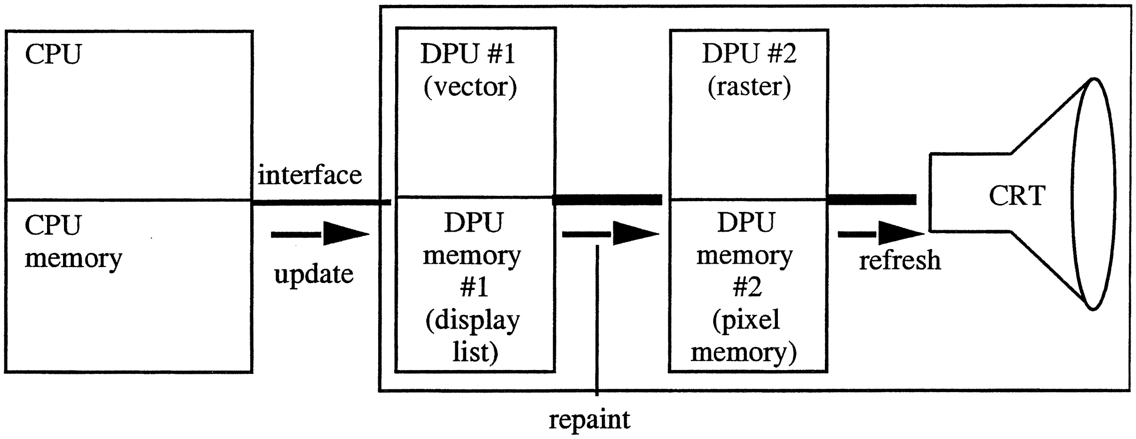


Figure 6-13. Raster Refresh Buffer.

The advantage of such a system is that a raster display now has structured commands, such as lines and characters, and erasure. The display independently handles the repaint process that all raster drawing requires.

The disadvantage is the cost and complexity of the additional hardware. Also, in the limit as the update rate approaches the refresh rate, such as in fast animation, one is limited by the speed of DPU #1.

6.4 Review Questions

1. A random vector display processor and a raster display processor read data from memory periodically in order to refresh the display on the screen. Describe the structure of this data for each display.
2. A picture consists of four lines that make a square on a CRT screen. Diagram and describe the display memory contents and display processor operation for this picture if the hardware is:

   (1) a random vector refresh display,

   (2) a black and white raster display.

Chapter 7. Graphics Systems

During the late 1970's and early 1980's, raster graphics technology advanced rapidly and raster devices quickly replaced vector graphics devices. As a result of rapidly decreasing costs of computer memory and video technology, raster graphics systems became less expensive than vector systems. Most importantly, raster devices could economically provide color, which had become essential for many graphics applications.

The software packages for vector systems were based on "graphical *segments*." A segment was the organizational unit for graphics information and provided a logical structure for dynamic display programming. Using segments was like drawing on a stack of transparent pages held over the screen. The programmer could draw only on the top page but the stack could be shuffled to allow drawing on any page. Segments (pages) could be deleted, blinked, and even transformed individually. All displayable entities were placed in segments by the programmer. All segments appeared on the screen at once.

It soon became clear that the segment-based graphics packages that had been used for vector display systems would not be appropriate for raster displays. Instead, the concept of a "*window*" was developed as the logical structure for organizing raster display systems.

7.1 Window Graphics Systems

The advent of workstations and personal computers greatly aided the development of window graphics systems. The basic window graphics systems in use today are described briefly in the next sections. The first two systems are "*kernel-based*" graphics systems, meaning the graphics functions and application must reside within the same computer. The last system described, the X Window System, is a "*network-based*" system, meaning the application and graphics functions can reside on different computers connected by a network.

Apple Macintosh™ Toolbox

Apple Computer, Inc. developed a comprehensive suite of software for the Macintosh computer that first shipped in 1984. The software consists of a hundreds of functions that are placed in ROM (read-only memory) contained in each Macintosh [1]. The set of functions is called the "Toolbox." A Macintosh program makes calls to Toolbox routines that are translated into system calls that enter the ROM code during execution.

Microsoft Windows™

Very similar to the Macintosh Toolbox, Microsoft Windows [2] was developed to support window graphics on IBM compatible personal computers. The first version of Windows appeared shortly after the Macintosh, and it has become the standard window environment for the IBM compatible world. Windows consists of a user environment that provides operating system functionality to the computer and contains graphics routines that are linked to programs executing Windows function calls. The routines support applications programming.

X Window System™

The X Window System [3] was developed jointly by Massachusetts Institute of Technology's Project Athena, Digitial Equipment Corporation and other companies starting in 1987. It was designed to support interactive raster graphics on networks of workstations. It has been adopted as a standard by nearly every workstation manufacturer.

X Windows is "*network-based*", meaning programs executing on one workstation calling X routines can display graphical data and interact with devices on another workstation or several other workstations. This makes it possible to run a program on a remote computer and interact with it on your local computer. This is termed a "*client-server*" model, where the *client* (*application*) program on a remote computer sends coded commands according to a "*protocol*", or "messages," over a network to a graphics *server* on the local computer. The commands are executed by the server to produce graphical output and other functions.

The functionality of the X Window System is made available to programmers through "*Xlib*," a library containing hundreds of C language routines for various functions. The Xlib routines are considered low-level functions, meaning they provide basic programming access to the protocol commands. Several software packages have been developed "on top of" Xlib routines to provide high-level functionality, for example, the OSF/Motif™ [4] user and programming environment. High-level refers to more sophisticated functionality intended for software developer support.

There are some common elements in each of these window systems:

windows are the fundamental organizational structure for drawing on the screen and communicating with the user,

events convey information about actions (user inputs, system happenings) to the program,

drawing primitives

are lines, characters, fill regions, color and pixel operations.

The following sections discuss these common elements.

7.2 Windows

Windows are in essence the raster counterpart to segments. They have a similar intent to segments, to allow a programmer to structure graphical display information in a manner appropriate for efficient and clear presentation and editing.

Some of the first window systems were implemented as "*tile systems*" in which the tiles, usually rectangular areas on the screen, cannot overlap. In other words, the screen is divided into a number of non-overlapping regions, or tiled windows.

"*Window systems*" relax this restriction and allow the regions to overlap.

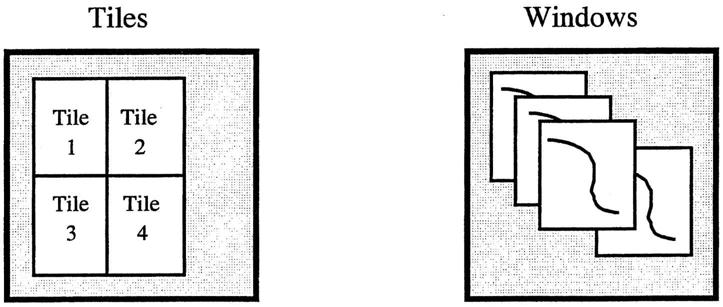


Figure 7-1. Tiles and Windows.

We can see that tile systems are considerably easier to implement. Tiles are a one-to-one mapping to screen pixels. Windows are a more difficult problem:

1. Each pixel can "belong" to a number of windows (Figure 7-2).
2. Only complex regions of windows may be visible. For example, the line a-b drawn in window A must be clipped against window B(Figure 7-3).

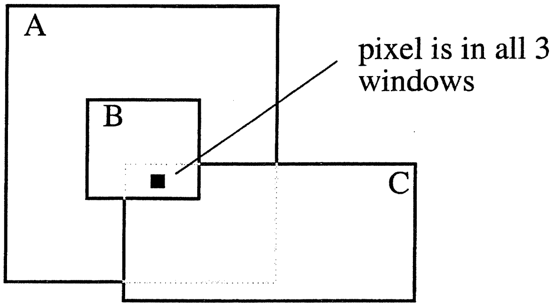


Figure 7-2. Overlapping Windows and Pixels.

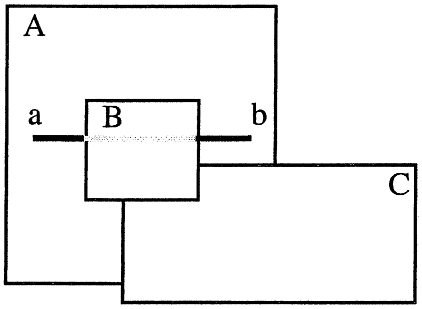


Figure 7-3. Overlapping Windows and Clipping.

3. There must be means for ordering the windows. The stacking order of the previous figures, from the "top", is B, C, A.

Windows are not only an organizational convenience for implementation. They are useful for efficient graphical communication with a user.

1. A window is a metaphorical means for "looking at the infinite world of data through a window." For example, scrolling, panning and zooming through a large picture or long document by moving the data through the window.
2. Windows are means for presenting many sets of data, or different views of one data set, to a user, like a stack of papers on a desk. The user can peruse the stack, move papers around as needed, pick one to edit, place them side-by-side for comparison, etc.
3. *Dialog windows* facilitate temporary communication with the user, such as warnings and data prompts, by re-using precious screen space.

7.3 Events

Graphics programs are notified of user actions, such as device inputs, and other noteworthy happenings through messages called *events*. Events control the execution of interactive programs and are central elements in all graphics systems. In general, all graphics programs are passive, waiting for events caused by actions to cause them to react.

At the heart of each graphics system is the event processor that accumulates information and actions from devices connected to the computer and other sources, and places the data associated with the event (such as the key typed on the keyboard) at the end of the "*event queue*." The typical graphics program contains at least one *event loop* that continually asks the question “is there an event for me to process?” In general, there is a function that removes an event from the head of the event queue and returns the data to the caller. For example, an event loop for a GRAFIC program is shown in Figure 7-4. The function `GNextEvent` returns four pieces of data: the

```
for( ;; ) {     /* "forever" */
     eventId = GNextEvent( &wid, &a, &b, &c );
     if( eventId == GKEYPRESS ) {
          if( a == 'a' )
               /* key 'a' has been pressed on keyboard */
     }
     if( eventId == GUPDATE ) {
          /* process window update event */
     }
}
```

Figure 7-4. A GRAFIC Event Loop.

window in which the event occurred (`wid`), and three data associated with the particular event (`a,b,c`). In the example, when a key is typed on the keyboard, an event of type `KEYPRESS` is returned and the first argument (called `a` in the figure) will contain the ASCII code of the key. When running the X Windows version of GRAFIC, `GNextEvent` will call Xlib’s `XNextEvent`; in Macintosh GRAFIC, the Toolbox function `GetNextEvent` is called; and in Microsoft Windows GRAFIC, `PeekMessage` is used in conjunction with a window “callback” procedure to accumulate events for `GNextEvent`. There are typically dozens of event types.

7.4 Drawing Primitives

Finally, all graphics systems have a set of functions for drawing. As expected, at the heart of each package are basic functions for drawing lines and characters. Most packages have functions for other types of primitives, such as arcs, polygons (a closed loop of lines), rectangles, circles and ellipses, markers (a symbol at a point) and fill regions. Fill regions are closed areas bounded by lines (and possibly arcs) into which a color can be "poured." Some systems provide sophisticated means for constructing fill regions, such as Boolean combinations (union, difference and intersection) of rectangles, circles, and other regions.

Drawing primitives and windows are linked through coordinates. The coordinates for lines are given in *window coordinates*, Cartesian [x,y] values that are measured relative to the origin of a given window. Each window contains its own origin, and it is possible to draw in any window at any time. In GRAFIC (and all the window systems described previously), the window origin is its upper left corner and window coordinates extend right and down.

Graphics systems also provide some representation for *drawing attributes*, such as color and *drawing mode* or function. Drawing mode refers to arithmetic and logical operations between the pixel value to be drawn and the corresponding pixel value in pixel memory. For example, think of drawing a line as an operation on the set of pixels in pixel memory where the line is to appear. For each of these pixels, the typical operation is to replace the pixel value in memory with the pixel value of the line. For example, drawing a red line means putting the pixel value of red into the pixels in pixel memory where the line is to appear. Now consider other operations than simply copying the pixel value into pixel memory.

相同：0
不同：1

In GRAFIC, for example, a pen mode called GXOR means perform the Boolean exclusive-or function with the "source" pixel value, the pen color, and the "destination" pixel value, the corresponding pixel in pixel memory. This is illustrated in Figure 7-5. The first line drawn extends from [100,100] to [400,100] in black. The pen mode is changed to GXOR, and a second line is drawn from [200,100] to [300,100]. For each pixel in the second line, the current color (black) is XOR'd with the corresponding pixel in pixel memory, in this case, also black. The result of black XOR black is white, as shown. Graphics packages typically provide a dozen or two such functions to control the drawing mode.

Drawing attributes are usually maintained in a *drawing state* of some kind. When a drawing

```
GPenColor( GBLACK );
GMoveTo( 100, 100 );
GLineTo( 400, 100 );
GPenMode( GXOR );
GMoveTo( 200, 100 );
GLineTo( 300, 100 );
```

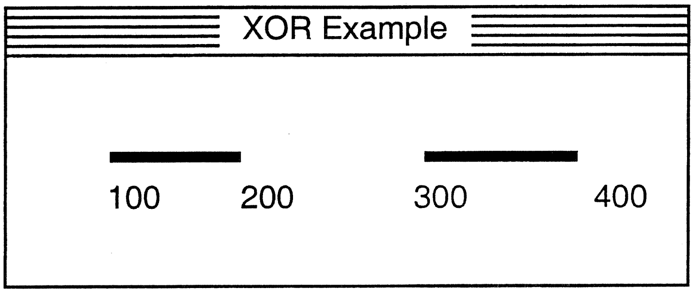


Figure 7-5. GRAFIC Calls Using XOR and Resulting Display.

primitive is executed, its appearance is determined by the drawing attributes in the current drawing state. For example, in GRAFIC, each window has a separate drawing state that defines the color and drawing mode for the next line, character or fill region that is created. Functions are available for changing the drawing state, such as `GPenColor` to set the color, and `GPenMode` to set the mode. In the Macintosh Toolbox, the drawing state is called the *GrafPort* ; in Microsoft Windows, it is the *device context*; and in X Windows, it is the *graphics context.*

7.5 References

1. Apple Computer, Inc., Inside Macintosh, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1985.

2. Microsoft Inc., Microsoft Windows 95 Programmer's Reference, Microsoft Press, Redmond, Washington, 1992.

3. Nye, A., Xlib Programming Manual for Version 11, Volumes One and Two, O'Reilly &Associates, Inc., 1990.

4. Open Sofware Foundation, OSF/Motif User's Guide, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

Chapter 8. Interactive Techniques

Recently, due largely to the rapid influx of personal computers in the home and workplace, considerable attention has been given to *graphical user interfaces* as an important aspect of programs and the key to "ease of use." The design of a good graphical user interface, or *GUI*, is both an art and a science. The following sections describe some basic GUI techniques.

The goals of a successful user interface should be to:

1. provide a natural person-machine interface, where natural means (a) appropriate and (b) efficient for the application,
2. fully exploit the power of the machine to augment the user's capabilities,
3. let the machine do the work.

A good GUI should follow and exploit the *principle of least astonishment*:

Make the expected happen; make what happens expected.

Exploit consistency and user anticipation.

8.1 Pointing and Positioning

An important aspect of any GUI is the capability of pointing to parts of the visual presentation on the screen before the user, typically through the use of pointing and positioning devices coordinated with moving graphical entities on the screen. *Cursors* are one predominant form of pointing and positioning. A cursor is a graphical symbol, or icon, that moves on the screen according to user actions. The mouse and pointer symbol is a common example of this today. As the user moves the mouse with hand movements, a cursor moves on the screen. The user can move the cursor "near" other graphical entities on the screen and direct the program to perform some action.

The form of the cursor can be an important source of information to the user. Special cursors can be used to convey problems or special input requirements as in Figure 8-1. As screen position indicators, cursors can take different forms that are appropriate for certain tasks. For

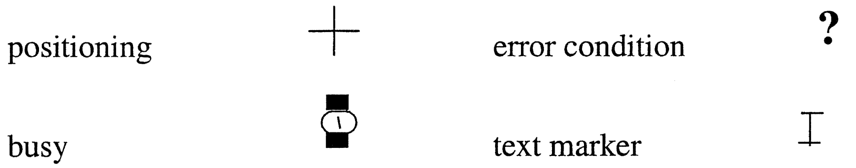


Figure 8-1. Examples of Special Cursors and Meanings.

example, *cross-hair cursors* are useful for aligning objects (Figure 8-2).

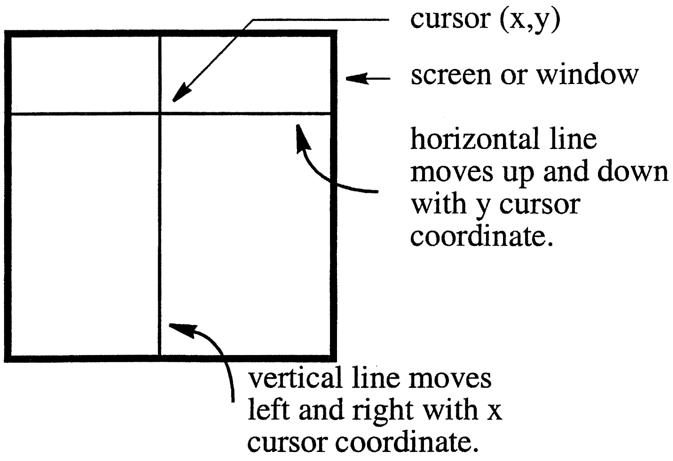

cursor (x,y)
screen or window
horizontal line moves up and down with y cursor coordinate.
vertical line moves left and right with x cursor coordinate.

Figure 8-2. Cross Hair Cursor.

Positioning constraints provide more exact control of cursor motion according to user-specified options. Constraining the motion of objects connected to the cursor to either horizontal or vertical components of the cursor position facilitates accurate positioning operations (Figure 8-3).

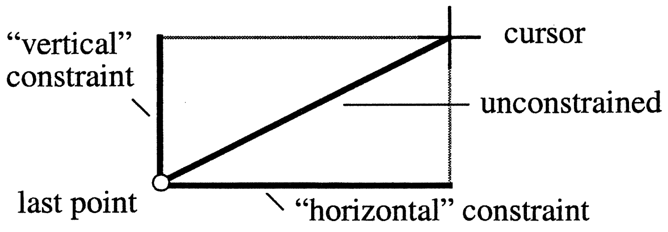


Figure 8-3. Constrained Cursor Motion.

Constraining the cursor location to fixed *grid* locations, sometimes called "snap to grid," means the cursor is moved to the nearest imaginary (or visible) grid location that is computed for each cursor movement by rounding off the exact position to the nearest grid location (Figure 8-4).

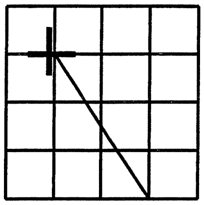


Figure 8-4.Cursor Positioning Constrained to a Grid.

8.2 Dragging

Feedback during positioning allows the user to "preview" the appearance of objects during re-sizing and moving operations. One of the most useful techniques for providing this feedback is to move objects with the cursor, called "dragging." Dragging means drawing and erasing an object as its position is changed with the cursor position.

With segment-based vector display systems, implementing dragging is relatively simple. The object to be dragged is placed in its own segment, distinct from segments displaying other data that will not change during dragging. As the cursor moves, the segment of the dragged object is continually erased and re-created, thereby creating the visual effect to the user of an object moving across the screen. In actuality, the object is not moving, it is an effect created by sequentially erasing the object drawn at the old position and drawing the object at a new position.

Although the principle is the same, implementing dragging with raster graphics systems requires special operations because there is a problem with the "erasing" steps. At first thought, erasing can be implemented easily by drawing over the previously drawn object in the background color (typically white).

Considering the object to be a straight line, the visual appearance of dragging is illustrated in Figure 8-5. This effect is called the *rubber-band* line. A first algorithm for the rubber-band line with a raster graphics package might have the following steps:

1. draw the line the from the start point to the old end point,
2. get the new cursor location, called the new end point,
3. redraw the old line, from start to old end point, in white to erase it,
4. assign the new end point coordinates to the old end point,
5. loop back to step 1.

The flaw in this approach is step 3, erasing the old line by drawing it in the background

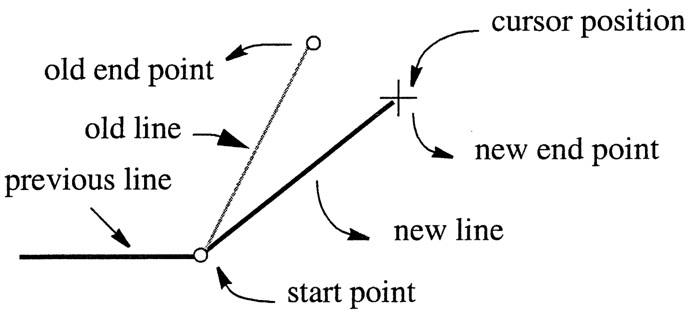


Figure 8-5. Dragging A Rubber-Band Line with the Cursor.

color, white. This operation replaces all pixels along the line with white, even those that belong to other lines passing through these pixels. For example, in Figure 8-6 we see the effect of erasing one line that crosses others, the classic "pixel in common" problem. As the dragging continues, the screen can become entirely erased.

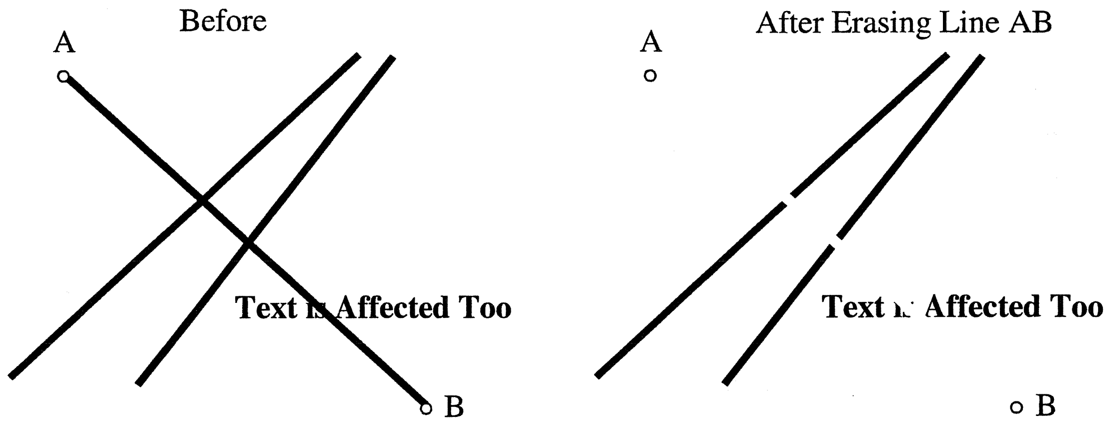


Figure 8-6.Illustration of the Raster Erasing Problem.

One inefficient and time-consuming solution is to redraw the entire image each time the line is redrawn. This will produce an annoying flash that will irritate the user. A better approach utilizes the drawing function exclusive-or (XOR), or the pen mode "GXOR" as it is called in GRAFIC. The XOR operation is a well known Boolean logic function whose binary logic table is shown in Figure 8-7. An important property of the XOR operator is that XOR'ing the same source

		A value	
		0	1
B value	0	0	1
	1	1	0

Results of A XOR B

Figure 8-7. XOR Truth Table.

value twice in a row to a destination value restores the original destination value. This is the key to dragging in raster graphics. Using a pen mode of GXOR, drawing the same object in the same color at the same location twice restores the original pixels in pixel memory.

Notice that for the pixel values of black and white, this works “neatly” as illustrated in Figure 8-8. In color, where pixel values can be arbitrary integers in general, the operation is still valid but produces arbitrary colors. Therefore, exclusive-or is useful only during dragging. After dragging is finished, the entire image must be re-drawn in GCOPY pen mode to restore the correct image.

Pen: B
B→W→B
W→B→W

For white = 0 and black = 1:

source	destination	source xor destination
black(1)	white(0)	black(1)
black(1)	black(1)	white(0)

Figure 8-8. XOR Operations on a Black and White Display.

With this information, an algorithm for the rubber-band line will have the following steps:

1. draw the line the from the start point to the old end point in GXOR mode,
2. get the new cursor location, called the new end point,
3. redraw the old line, from start to old end point, in GXOR mode to erase it,
4. assign the new end point coordinates to the old end point,
5. loop back to step 1.

Keep in mind that the XOR operation works on the pixel values as numbers. For example, consider the case of a mapped display, where pixel values are color table indices. Suppose that the color white is the pixel value 5 and black is 6. Drawing a black line over a white area means the source pixel values are black, or 6, and the destination pixel values are white, or 5. In GXOR pen mode, drawing the line means that the system XOR’s each source and destination pixel value for each pixel location along the line. In this case, 5 XOR 6 is 3 (5 is 101_2, 6 is 110_2, 101_2 XOR 110_2 = 011_2 or 3), so the resulting pixel values along the line are 3. But what color is pixel value 3? For a mapped display, it is the color in color table location 3, which may or may not be a color set by you! Drawing the same line in black a second time means that the pixel values will be 6 XOR 3, or 5, which restores the color white, as expected.

Figure 8-9 shows an implementation of rubber-band line dragging using GRAFIC routines. The routine `Drag` is called from the main event loop when dragging should begin.

```
void Drag( int h0, int v0 )
{
      WID wid;
      int event, c, moved, oldh, oldv, newh, newv;
      moved = 0;
      oldh = h0;
      oldv = v0;
      GPenMode( GXOR );
      do {
            event = GNextEvent( &wid, &newh, &newv, &c );
            if( event == GMOUSEMOTION ) {
                  if( moved != 0 ) {
                        /* redraw (erase) old XOR line */
                        GMoveTo( h0, v0 );
                        GLineTo( oldh,  oldv );
                  }
                  if( newh != oldh || newv != oldv || moved != 0 ) {
                        /* draw new XOR line */
                        GMoveTo( h0, v0 );
                        GLineTo( newh, newv );
                        oldh = newh;
                        oldv = newv;
                        moved = 1;
                  }
            }
      } while( event != GBUTTONUP );
/* redraw the final line in GCOPY mode for correct picture */
/* it is also necessary to redraw the entire picture */
      GPenMode( GCOPY );
      GMoveTo( h0, v0 );
      GLineTo( newh, newv );
}
```

Figure 8-9. GRAFIC Drag Code Example.

8.3 Picking

Picking is an important graphical input process where the user identifies an object drawn on the screen using an input action. Typically, this is accomplished with a positioning device, such as a mouse, with which the user causes a cursor on the screen to move near an object of interest and then indicates the action of picking with another input, such as depressing a mouse button or typing a key.

The program should acknowledge a successful pick (and a failed pick), by some response. The form of response is application dependent. Examples of good responses are:

1. *Highlighting* and unhighlighting the selected object(s). Highlighting means to graphically distinguish selected objects from others.
2. No response, when the reason is clear. For example, clicking the mouse button when the cursor is near nothing does not require a message saying "you picked nothing."

Examples of bad responses are:

1. Annoying, persistent or unclear actions, such as beeps, bells, blinking messages.
2. No response, when the reason is unclear. For example, not allowing an object to be picked may need some response.

Picking is often (nearly always) made difficult by problems of *ambiguous picks*, when more than one object may be identified with the picking action. There are many alternatives to resolving ambiguous picks, depending on the *application context* and the *screen context.*

Application context

means the understanding in the user's mind based on the knowledge required to get this far.

Screen context

means the visible graphical environment the user sees at the time.

In general, you (the developer) must have a *user model* in mind when designing the user interface.

Examine some basic approaches to picking objects.

8.3.1 Picking Text

Picking text is usually based on a "characteristic location," such as the lower left corner of the first character or inside the imaginary rectangle the bounds all the text.

8.3.2 Picking Lines

There are several different approaches to picking lines.

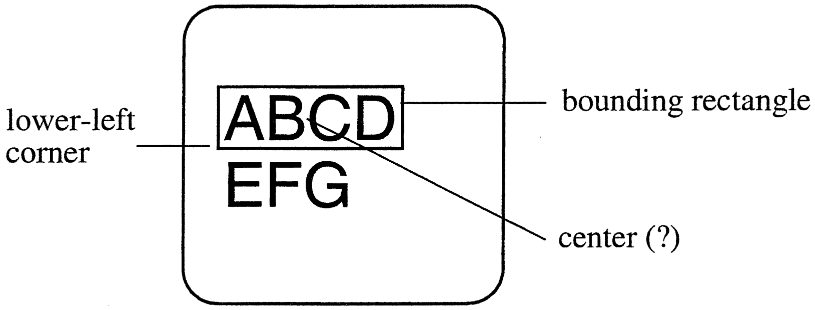


Figure 8-10. Characteristic Locations for Picking Text.

End points:

The Euclidean distance produces a circular picking region. Alternatively, the absolute value computations create a rectangular picking region that is nearly indistinguishable to the user from the circular, but are computationally faster.

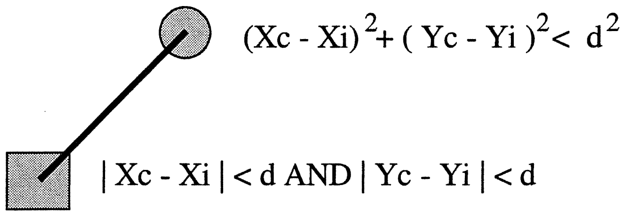

$(Xc - Xi)^2 + (Yc - Yi)^2 < d^2$
$| Xc - Xi | < d$ AND $| Yc - Yi | < d$

Figure 8-11. Picking Regions for Euclidean and Absolute Value Tests.

Anywhere along a line:

The problem is picking a bounded line (segment) with endpoints. We must combine the mathematical procedure for computing the distance to the unbounded line with the algorithmic process of limiting the picking region to the bounded portion of the line.

The distance from a point to an unbounded line is classically computed as,

$$d = \frac{|Ax + By + C|}{\sqrt{A^2 + B^2}}$$

where the line equation is $Ax + By + C = 0$, and A, B, and C are found from the line end points by solving the simultaneous equations:

$$Ax_1 + By_1 + C = 0$$

$$Ax_2 + By_2 + C = 0$$

Note that this involves dealing with a number of special conditions (A=0 and B=0, for example).

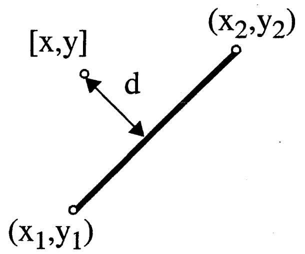


Figure 8-12. Distance from Point to a Line.

Another method, based on a vector solution that will be discussed later, is:

$$d = \frac{|\Delta x l_y - \Delta y l_x|}{\sqrt{\Delta x^2 + \Delta y^2}} \qquad \text{where:} \quad \begin{aligned} \Delta x &= x_2 - x_1 \\ \Delta y &= y_2 - y_1 \\ l_x &= x - x_1 \\ l_y &= y - y_1 \end{aligned}$$

Caution: the denominator above can be 0 when the line is actually a point.

This test alone, however, has a picking region that extends infinitely along the line:

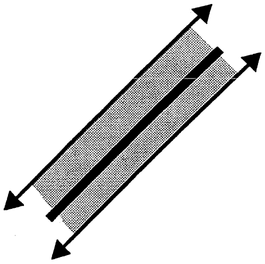


Figure 8-13. Picking Region for an Unbounded Line.

To bound the picking region, perform tests according to the Boolean expression:

picking region = “inside the smallest rectangle surrounding the line and its endpoints”

AND

“within distance d of the unbounded line”

This is shown visually in Figure 8-14.

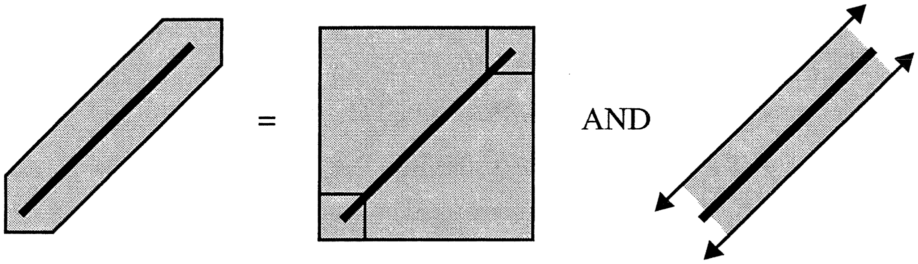


Figure 8-14. Picking Regions for a Bounded Line.

8.3.3 Picking Circles

Picking a circle means that the cursor is within a given distance of either side of the circle, resulting in the annular picking region illustrated in Figure 8-15. One approach would be to compute the distance from the cursor location [x,y] to the circle center [xc,yc] and compare this to the boundary radii of the two circles bounding the picking region, R+d and R-d. This approach is adequate for circles, but does not generalize for other conics (curves of degree two) such as ellipses.

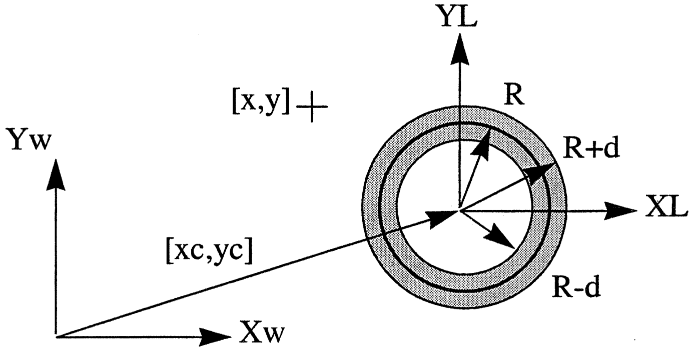


Figure 8-15. Picking Region for a Circle of Radius R and center [xc,yc].

A more general approach is based on the properties of conic equations, i.e. in this case the equation of a circle: $x^2 + y^2 - R^2 = 0$. Notice that if the point [x,y] is inside the circle then $x^2 + y^2 - R^2 < 0$ and if the point [x,y] is outside the circle then $x^2 + y^2 - R^2 > 0$. To simplify and speed the computations, first transform the cursor location into the local coordinate system with its origin at the center of the circle. This transformation is simply [x,y] - [xc,yc]. We now test the location of this point with respect to the picking region, which can be expressed logically as:

"inside the outer circle (radius R+d) AND outside the inner circle (radius R-d),"

or more conveniently,

"inside the outer circle (radius R+d) AND NOT inside the inner circle (radius R-d)."

The second form simplifies the logic to one routine "Boolean InsideCircle(x, y, radius)."

8.3.4 Picking Composite Objects

A composite object is an object that consists of a collection of simpler objects, such as lines, circles, or other entities. Picking composite objects is application dependent, meaning the user may or may not have characteristic locations for objects that are natural picking points or regions.

Vertices are often convenient, except for curved objects.

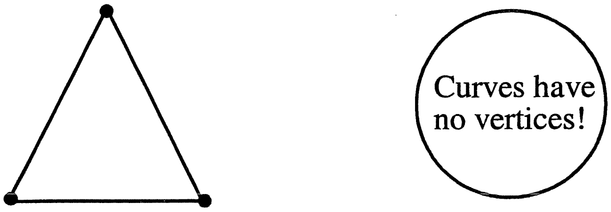


Figure 8-16. Vertex Picking Examples.

Picking an object that is composed of lines means picking any of its lines.

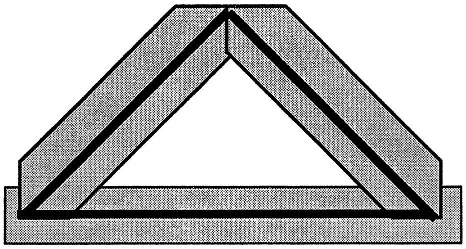


Figure 8-17. Picking an Object Composed of Lines.

Picking an object by picking near its *centroid*, i.e., the geometric center of an object, can sometimes be useful (but not often).

A *bounding rectangle* is the most often used picking region. This is visually understandable but also can be a gross over-approximation to the "inside" of an object.

Picking by number or name should be used only when numbers or names are inherent to

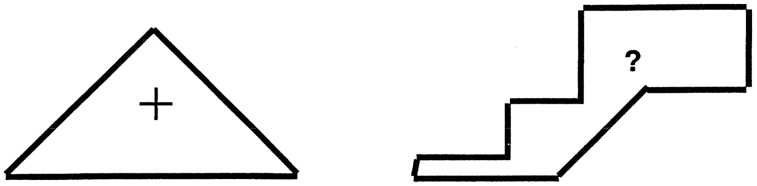


Figure 8-18. Centroids of Objects.

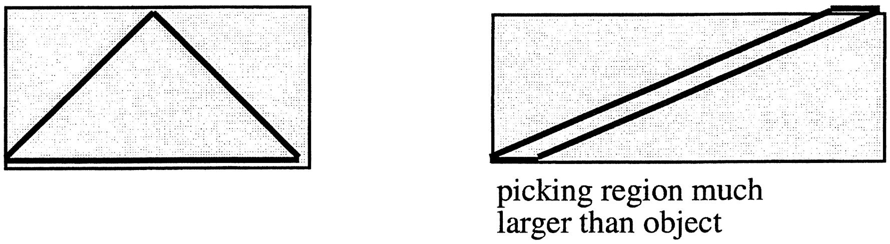


Figure 8-19. Bounding Rectangles of Objects.

the application, i.e. objects have names known by the user anyway. Even in this case, graphical picking should be permitted too.

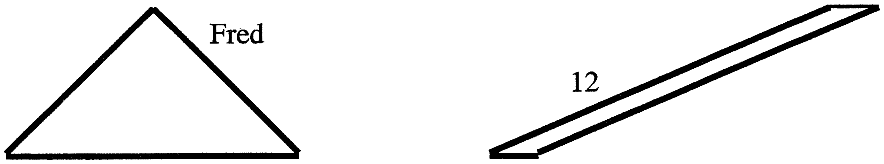


Figure 8-20. Showing Object Names.

Picking inside objects is a sophisticated approach to picking that is useful for simple objects, such as rectangles, but is very complicated for other objects, especially those with curves.

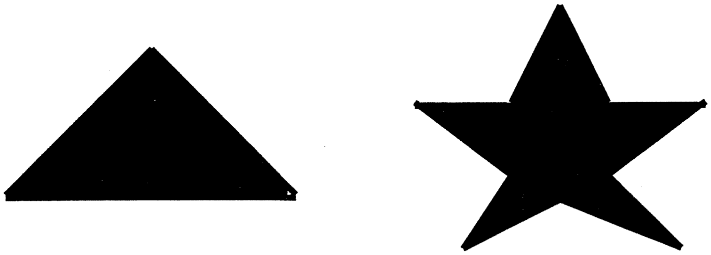


Figure 8-21. Interiors of Simple and Complex Objects.

Inside/outside tests, or point containment tests, will be discussed in a later chapter.

Picking inside a *selection region* is useful for selecting all objects in an area, usually a rectangle, on the screen.

This requires special tests for the intersection of the selection region with the "object

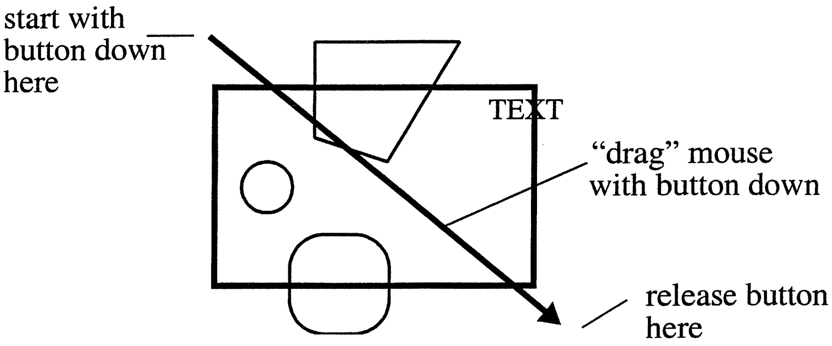


Figure 8-22. Picking Objects Inside a Selection Region.

region," either the bounding box, individual lines, or vertices.

8.3.5 Interaction Paradigms

Verb - Noun Paradigm

The interaction model in most early interactive graphics systems can be characterized as *verb-noun paradigm* because the user first selected the action (verb) and then was prompted for the entities on which to act (nouns).

Nouns are selected objects, such as text characters, graphical objects, markers, etc., whatever objects are pertinent to the application.

Verbs are commands that operate on objects.

This command structure approach usually required hierarchical menus and numerous prompts to request picks from the user. Research in user interfaces showed that such systems are very difficult to learn, difficult to remember and require an excessive number of user inputs.

One of the main drawbacks of the verb-noun paradigm is that the user (and program) is continually entering a *mode*, meaning a program state requiring special user inputs to complete. Modes usually require additional user inputs (e.g. enter mode, exit mode) and tend to challenge users to recall where they are and what is required to exit and complete the mode. In addition, verb-noun systems must handle many error conditions when the user picks invalid data items.

Noun-Verb Paradigm

Recent systems, first popularized by the Apple Macintosh graphical user interface, are based on a *noun-verb paradigm*. At first glance, the noun-verb paradigm appears to be a simple reversal of picks and commands, but in practice it greatly simplifies user interfaces. The user selects (and un-selects) objects (the nouns), and then executes the desired command (verb).

Noun-verb systems require that the program and user maintain a *selection*, meaning a set of selected objects. The selection is *highlighted* to visually distinguish selected objects from un-selected objects. For example, selected text is drawn inverted (or outlined by a darkened box, etc.), or a rectangle has tabs called handles drawn at its corners.

Knowing the selection before executing the command allows the program to:

1. enable and disable menu items and change their text to be appropriate for the selected objects and more informative to the user,
2. change the action performed to depend upon the selected objects.

For example, a "delete" command can be disabled (grayed and not allowed to be executed by the user) when no object is selected, different commands can be enabled when different numbers or types of objects are selected, and so on.

8.4 Review Questions

1. Summarize the "verb-noun" and "noun-verb" paradigms for graphical user interfaces. Give an example of how a "delete object" command could be changed to follow the verb-noun paradigm.
2. Explain in detail how the GRAFIC pen mode "`GXOR`" allows objects to be "dragged" across the screen without erasing the underlying picture. Give "before and after" examples with pixel values for a black and white display, and a mapped color display.

Chapter 9. Data Structures for Interactive Graphics

As you have probably seen to this point, graphics programs are always operating on data. In a non-graphical program, there is only one use for the data: analysis. By incorporating interactive graphics, data is now also used for interaction and display. Accordingly, we have two "types" or categories of data:

1. Problem data: for analysis (e.g., motion analysis of a mechanical device)
2. Graphical data: for display (e.g., creating and displaying the device)

In general, these types of data are not always distinct, i.e. there is overlap where data is used for both (Figure 9-1):

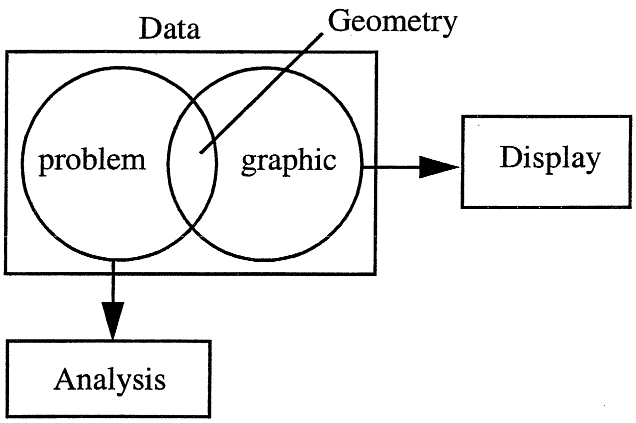


Figure 9-1. Problem and Graphical Data.

Examples:

Graphical Data	Problem Data
Geometry	Geometry
Visual Properties (color)	Physical Properties (density)
Graphics Package Information (windows, etc.)	

The following development of data structures combines methods for representation of objects with the needs for interactive graphical interaction.

9.1 Basic Data Storage Methods

First examine the data needed to create one object, one "entity" at a time. We will use points and straight lines as our entities, but the ideas can be generalized to other geometric and non-geometric forms. Keep in mind that the controlling program is interacting with a user whose commands activate and direct the actual storage and manipulation of the data, in this case lines.

As the object is drawn on the screen, which is purely a graphical operation of the program, it is also necessary to store the lines as data for later reference, either for analysis ("What is the length of the line?"), or for editing operations ("Delete this line").

9.1.1 Sequential lists

The most direct method of storing data is a *compacted sequential list*, or *CSL*, as shown in the following diagram.

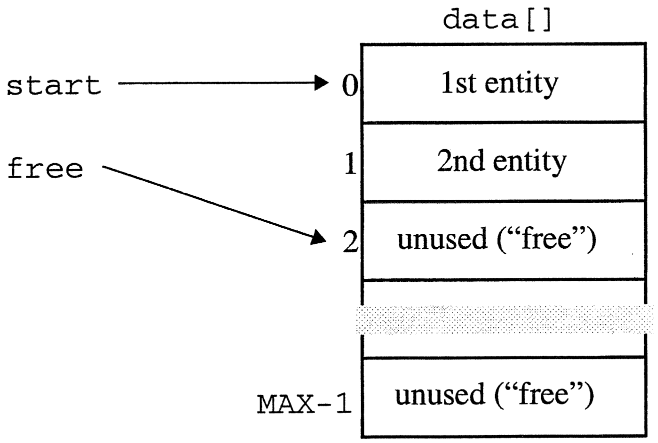
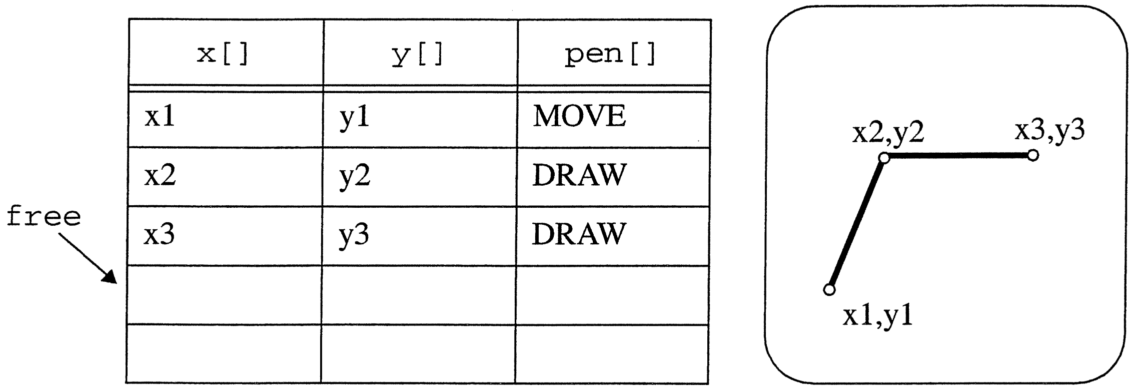


Figure 9-2. Compacted Sequential List Storage.

The variables `start` and `free` are *pointers* that indicate locations in the data array(s) that hold the first used *entity* and first free *cell*, respectively. In a CSL, `start` always points to the first cell, index 0, and is actually unneeded. In a CSL, entities are first, followed by free cells, in two contiguous blocks of cells.

The entities can be stored in several array(s) indexed by the same variable. For example, if one were storing a line drawing as a sequence of end points marked by pen codes, there would be three arrays (see Figure 9-3).



x[]	y[]	pen[]
x1	y1	MOVE
x2	y2	DRAW
x3	y3	DRAW

Figure 9-3. Points and Pen Codes Data Structure.

The ith entity consists of `x[i]`,`y[i]` and `pen[i]`.

In structured languages like C, C++ and Java, entities are more conveniently represented by structures or classes. For the example above, we could define the C data type `Entity` as a structure as follows.

```
typedef struct {
     int x, y;
     int pen;   /* constant meaning MOVE or DRAW */
} Entity;
```

If the size of the array is known at compile time, the data storage can be allocated statically as a single array of structures, `data`:

```
Entity data[MAX];
```

If the necessary size must be computed or otherwise found at run time, the `data` array can be allocated dynamically:

```
In C:      Entity* data = calloc( MAX, sizeof(Entity) );
In C++:    Entity* data = new Entity[MAX];
```

The x value of the ith entity is referenced as `data[i].x`.

Consider the four basic operations on our data structure: initialization, appending an entity, deleting an entity, and inserting an entity.

1. Initialization:

Set `free` to point to the first cell.

2. Appending an entity:

The general process is:

1. obtain a free cell,
2. append the cell to the last entity of the object,
3. fill the cell with the entity data.

For our points example, assume that point `newx, newy, newpen` is to be appended to the object. Figure 9-4 shows a code fragment for performing this operation.

```
/* step 1 (obtain free cell) */
      new = free;
      if( new >= MAX ) Error("OUT OF SPACE");
/* step 2 (append cell to last entity, remove from free) */
      free = free + 1;
/* step 3 (fill the cell with the entity data) */
      data[new].x = newx;
      data[new].y = newy;
      data[new].pen = newpen;
```

Figure 9-4. Appending an Entity to a CSL.

Note that for a CSL the statement `free = free + 1` appends the entity to the object and removes the cell from free storage in one step.

3. Deleting an entity.

If the cell is the last cell in the object, the process is simply decrementing the `free` pointer as illustrated in Figure 9-5.

¶ re-draw

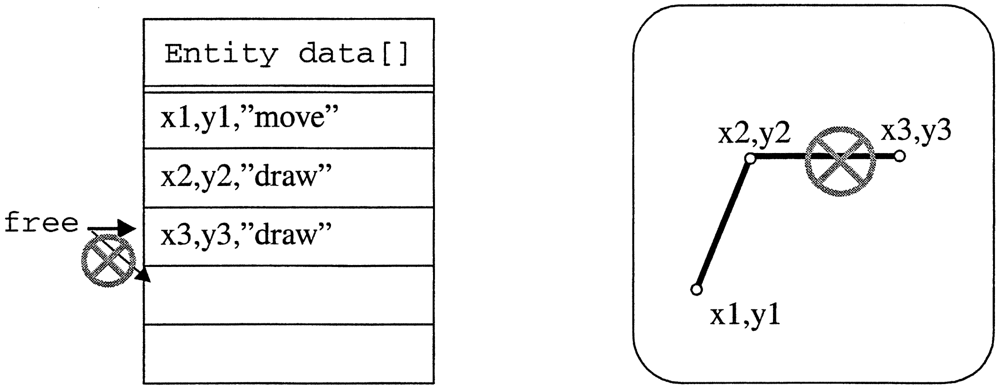
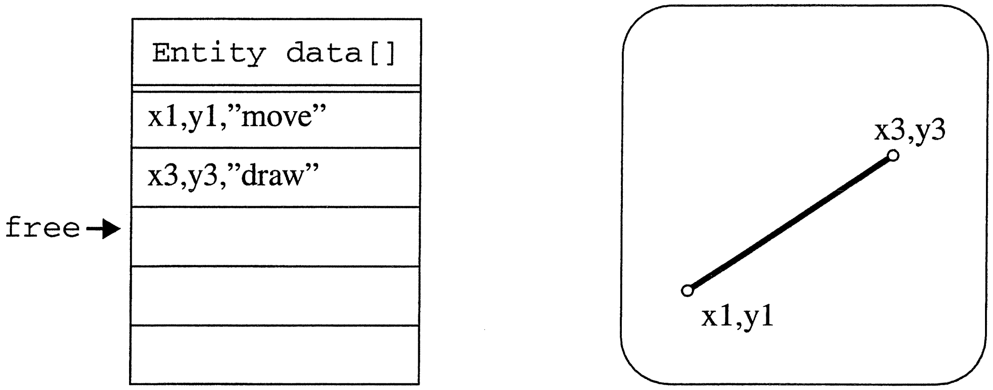


Figure 9-5. Deleting the Last Cell in a CSL.

If, however, the cell is not the last in the object, then we must *compact* the list to maintain the proper structure of the CSL for subsequent operations. For example, to delete the ith point, we must copy the cells from i+1 to free-1 "up" (to lower indices), or "shuffle up." This is necessary to fill the gap and to insure that all free cells are in a contiguous block at the end of the data array(s).

In our example, deleting the 2nd point would result in the structure in Figure 9-6.



Entity data[]
x1,y1,"move"
x3,y3,"draw"

Figure 9-6. Deleting an Entity (Point).

The loop to perform the compaction is illustrated in Figure 9-7.

```
   for( j = i + 1; j <= free - 1; j++ )
/* the next statement can assign a whole struct in C or C++, or
/* may need several assignments if separate arrays are used */
      data[j-1] = data[j];
   free = free - 1;
```

Figure 9-7. Compaction Loop.

4. Inserting an entity:

Inserting an entity between existing entities is like the reverse of the delete operation. To create a free cell before a given entity, it is necessary to copy the following entities down (to higher indices), or "shuffle-down." To insert a point before the ith point in our structure, we must move the entities from i through free-1 down (assuming there is a free cell), then fill the ith cell with the new entity data.

The loop to perform expansion is illustrated in Figure 9-8.

```
for( j = free - 1; j <= i; j = j - 1 )
   data[j+1] = data[j];
free = free + 1;
```

Figure 9-8. Expansion Loop.

There is an important point to make regarding data structures in interactive graphics applications. It is essential to remember that changing the data does not change the display. That is, the program must maintain the *integrity of the screen* representation of the internal data the user has created by redrawing when necessary. Often during the development of an interactive graphical program, the developer carefully and correctly codes the change in a data structure in response to a user command, but mistakenly forgets to redraw the display to show the change to the user. For example, the user issues a "delete line" command, but the line stays on the screen. The user then tries to pick the line, but the program fails to find it because it is no longer in the data structure. Further editing finally causes a screen redraw, and the user is surprised to see the deleted line gone, finally.

Conversely, another common error is the to change the display to the user, but failure to change the data. For example, the user issues a "delete line" command, and the line disappears from the display. After another command causes a re-draw of the screen, the pesky line magically reappears because the re-draw traverses the data structure, displaying each entity.

9.1.2 Linked Lists

In the CSL storage method, *adjacency* between array positions was used to determine which points were connected to form lines, and in general how to find the previous and next entities in the data. Thus, *connectivity* is implicit in a CSL.

Another method for storing data is a *linked list.* It appears similar to sequential storage, except the next entity need not follow the current one in storage. In other words, the data is linked through explicit pointers stored with each entity, as shown in Figure 9-9.

`Start` points to the first entity (not necessarily the first cell in the data) of the "entity list" that defines the object. `Free` points to the first cell of the "free cell list." Lists are terminated by a `NULL` pointer, a unique pointer value.

The display on the screen is the same, only the internal storage method has changed.

An array of integers can be added to hold the `next` pointers for the multiple array method. With structures, an "`int next;`" field is added to the definition of an entity. In the next section, we will use a C pointer variable for `next`.

It is usually easier to think of and draw a linked list as a graph of boxes connected by arrows as shown in Figure 9-10.

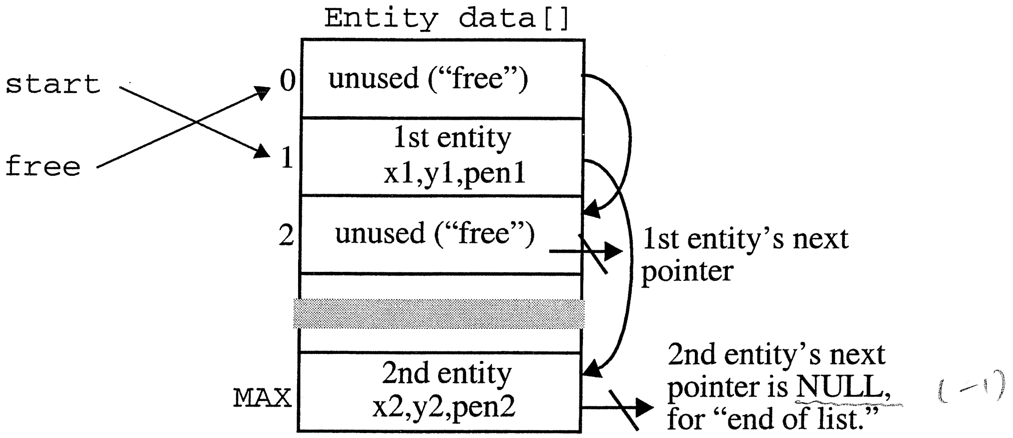


Figure 9-9. Linked List Storage.

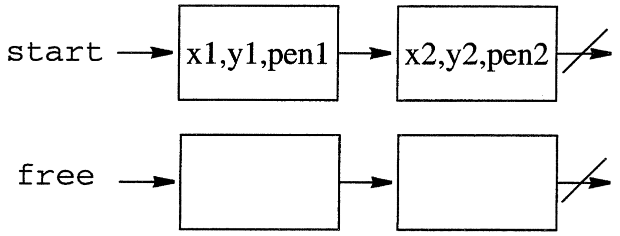


Figure 9-10. Diagram Form of Linked List.

The boxes are entities or cells, and the arrows are the links. The arrow with a line is a NULL pointer.

Now consider the basic operations as before, using the same example as before.

1. <u>Initialization</u>:

Initialization now means: set `start` to `NULL` and link all cells into the free list.

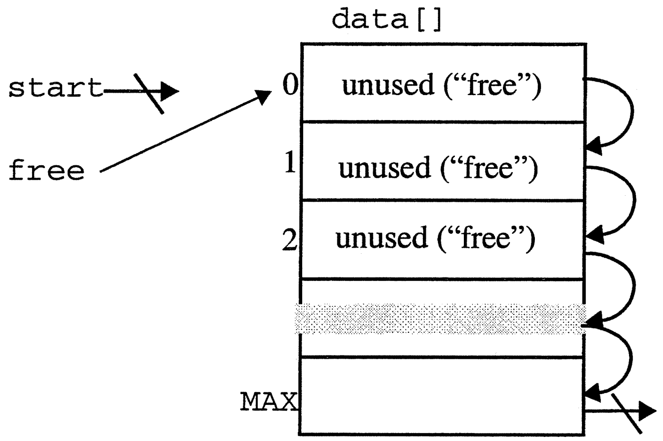


Figure 9-11. Initialization of a Linked List Structure.

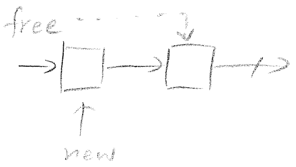
An initialization loop is illustrated in Figure 9-12.

```
start = NULL;
for( i = 0; i < MAX-1; i++ )
      data[i].next = i + 1;
data[MAX-1].next = NULL;
free = 0;
```

Figure 9-12. Linked List Initialization.

2. Appending an entity:

The three steps are the same: (1) get a free cell, (2) append the cell to the object, (3) fill the cell with the entity data. We now have an end condition to consider: when `start` is `NULL`, there is no last entity on which to append. Figure 9-13 illustrates one approach to appending to a linked-list.



```
/* step 1: get a free cell */
      new = free;
      if( new == NULL ) Error("OUT OF SPACE" );
      free = data[free].next;
/* step 2: find the last entity in the entity list */
      if( start == NULL ) {
            start = new;
            last = NULL;
      } else {
            last = start;
            while( data[last].next != NULL )
                  last = data[last].next;
      }
      if( last .NE. NULL ) data[last].next = new
/* step 3: */   (!=)
      data[new].x = newx;
      data[new].y = newy;
      data[new].pen = newpen;
      data[new].next = NULL; /* don't forget to init. next! */
```

Figure 9-13. Appending An Entity to a Linked List.

It may be more convenient and efficient to maintain a variable `last` that always points to the end of the list.

3. Deleting an entity:

Deletion shows the real advantage of linked lists for applications requiring dynamic data structures, i.e. data structures that are created, deleted, and edited interactively. Assume we have picked the entity to be deleted from the CSL and recorded a pointer to it in the variable `here`.

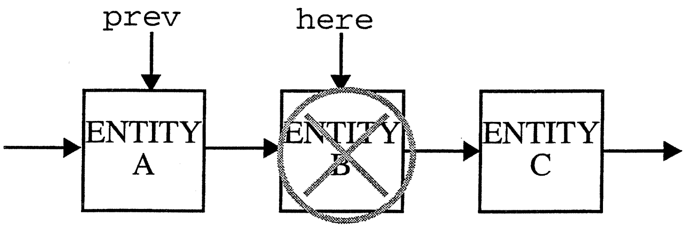


Figure 9-14. Linked List Deletion.

The pointer prev indicates the previous entity, i.e., the entity that points to here. Note that the first entity will not have a prev entity, so we handle this specially as another end condition. Given start, we can find prev as follows.

```
if( here == START )
      prev = NULL;
else {
      prev = start;
      while( data[prev].next != here )
            prev = data[prev].next;
}
```

Figure 9-15. Finding the prev pointer.

Now we are set to unlink the entity and return it to the free list.

```
/* first unlink the entity: */
      if( prev == NULL )
            start = data[here].next;
      else
            data[prev].next = data[here].next;
/* return the freed cell to the front of the free list */
      data[here].next = free;
      free = here;
```

Figure 9-16. Unlinking the entity referenced by here.

The free list is generally unordered, so returning the cell to the front is the easiest method.

4. Inserting an entity:

Inserting an entity is similar to appending one, so much so in fact that they can be combined without too much difficulty. Assume that we have already obtained a free cell and stored the data in the entity pointed to by new.

We use prev==NULL to mean at the front. The procedure can be coded as shown in Figure 9-18:

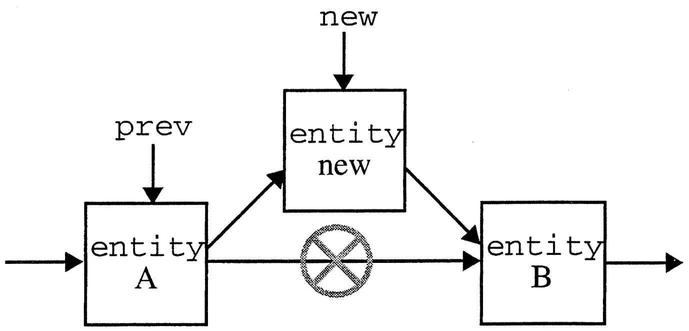


Figure 9-17. Inserting a new entity.

```
/* link the entity into the object after the entity 'prev' */
      if( prev == NULL ) {
            data[new].next = start;
            start = new;
      } else {
            data[new].next = data[prev].next;
            data[prev].next = new;
      }
```

Figure 9-18. Code to insert a new entity.

9.1.3 Linked Lists with C/C++ Pointers

Instead of integer pointers, it is more efficient and helps modularity to use C/C++ pointer variables to represent the start and next pointers in the Entity data. In this case, the Entity declaration would be:

```
typedef struct entity {
      int         x, y;
      int         pen;
      Entity*     next;      /* pointer to next Entity */
} Entity;
Entity* start; /* start pointer */
Entity* free;  /* free pointer */
```

(The line “Entity* next” will not compile as shown, but is written this way for easier reading.)

In the next figures, the previous code fragments are re-written to use the `next` pointer.

Figure 9-19 shows initialization using pointers, Figure 9-20 shows appending an entity, deleting_ an entity using pointers is shown in Figure 9-21 and inserting an entity is shown in Figure

```
start = NULL; /* NULL is usually 0 for C/C++ pointers */
for( i = 0; i < MAX-1; i++ )
      data[i].next = &data[i + 1];
data[MAX-1].next = NULL;
free = NULL;
```

Figure 9-19. Linked List Initialization Using Pointers.

```
/* step 1: get a free cell */
      Entity *last, *new;
      new = free;
      if( new == NULL ) Error(“OUT OF SPACE” );
      free = free->next;
/* step 2: find the last entity in the entity list */
      if( start == NULL ) {
            start = new;
            last = NULL;
      } else {
            last = start;
            while( last->next != NULL )
                  last = last->next;
      }
      if( last .NE. NULL ) last->next = new
/* step 3: */
      new->x = newx;
      new->y = newy;
      new->pen = newpen;
      new->next = NULL; /* don't forget to init. next! */
```

Figure 9-20. Appending An Entity to a Linked List Using Pointers.

```
      if( here == START )
            prev = NULL;
      else {
            prev = start;
            while( prev->next != here )
                  prev = prev->next;
      }
/* first unlink the entity: */
      if( prev == NULL )
            start = here->next;
      else
            prev->next = here->next;
/* return the freed cell to the front of the free list */
      here->next = free;
      free = here;
```

Figure 9-21. Finding `prev` And Unlinking An Entity Using Pointers.

9-22.

```
/* link the entity into the object after the entity 'prev' */
      if( prev == NULL ) {
            new->next = start;
            start = new;
      } else {
            new->next = prev->next;
            prev->next = new;
      }
```

Figure 9-22. Inserting a New Entity Using Pointers.

9.1.4 Using Compacted Sequential Lists and Linked Lists

Linked lists require more storage than CSL's. CSL's, however, require the movement of entities for insertion and deletion. If the entities consist of much data, and/or if there are large numbers of entities, CSL's can be slow. Also, in more complex data structures (as will be discussed next) where entities may be pointed to (referenced) by other entities, moving an entity requires that its reference(s) be updated accordingly. This can be clumsy, so linked lists would be preferred.

9.2 Storing Multiple Objects

Many applications operate on more than one object. This requires that a number of objects be available to construct a *scene*. Objects are "groups of entities" created by the user.

Now we must maintain groups of objects, or groups of groups of entities. Consider our basic storage techniques for representing groups of entities. We define the structure Object to contain a start and end pointer:

```
typedef struct {
      int  start;     /* Entity* start for C/C++ pointers */
      int  end;       /* Entity* end for C/C++ pointers */

} Object;
Object objects[MAXOBJ];
```

Compacted Sequential List:

A second data structure, the *object table*, itself a CSL, contains pointers to the `start` and

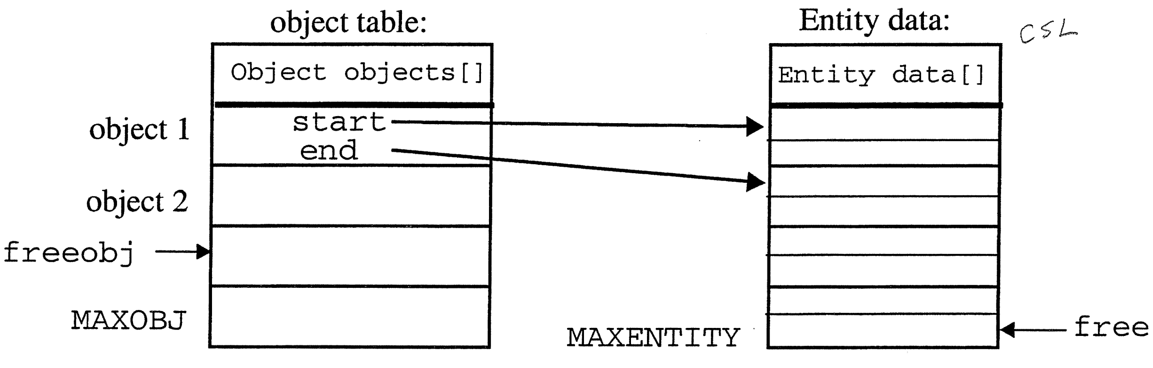


Figure 9-23. Object Table Using A CSL for Entity Data..

end of each object in the Entity data. For a linked list Entity data structure, the object table needs only the start pointer, and can appear as shown in Figure 9-24. All that is needed is the pointer

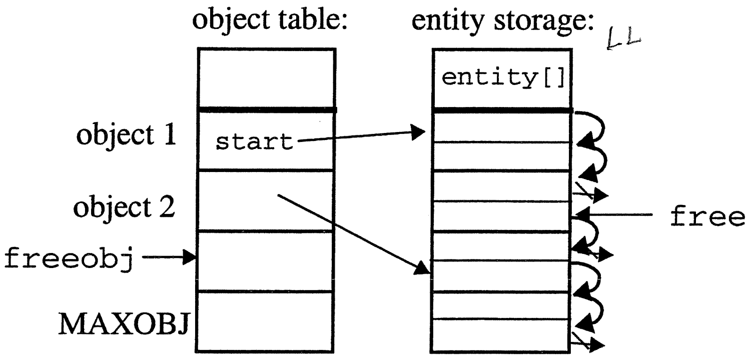


Figure 9-24. Object Table Using Linked Lists for Entity Data.

to the first entity (head) of a list. The object table is simply a CSL array of list head pointers.

9.3 Instancing with Multiple Objects

It is likely that what have been called objects would be used as *basic definitions* (or building blocks) to create a given scene. There may be two separate programs or two modes in one program, one to create the objects and another to create the scene given the object definitions.

It is wasteful and usually undesirable to duplicate the list of entities (no matter how they are stored) defining an object every time it is added to the scene. Instead, add another level of data to define the scene in terms of *instances* of the previously defined basic definitions. An *instance* is "a reference to a pre-defined object varying in position, size, orientation and attributes."

Consider the objects and scene shown in Figure 9-25. In addition to the object table and entity data, another data structure is needed to represent the instances of objects in the scene, an *instance table*, which stores data for each instance. For this, we could define a structure and type called Instance:

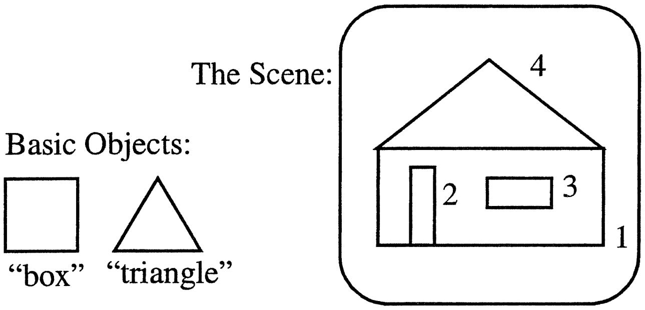
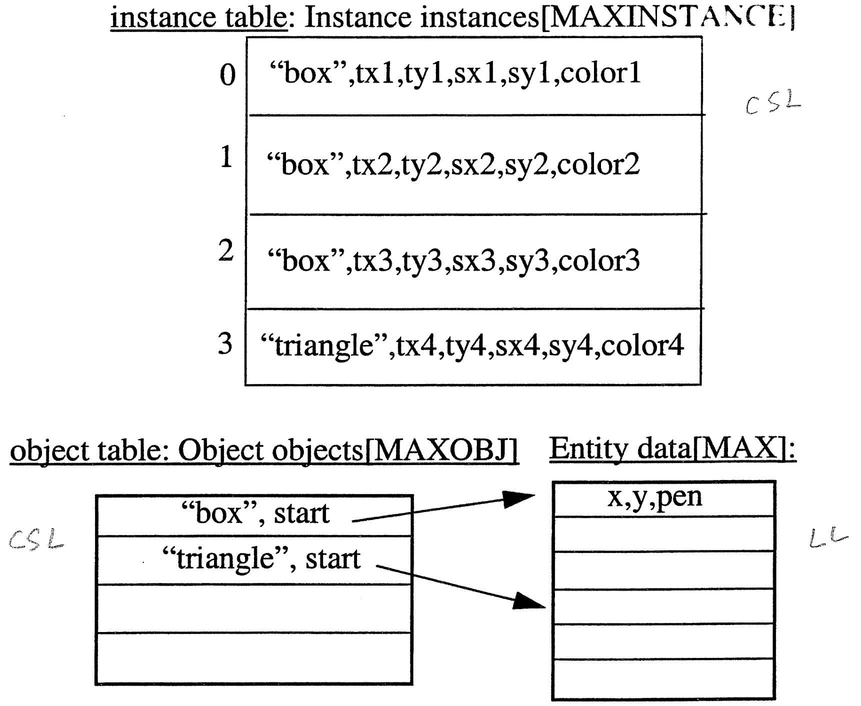


Figure 9-25. Instances of Basic Objects in a Scene.

```
typedef struct {
     int       object;    /* pointer to basic object */
     float     tx, ty;    /* the translation */
     float     sx, sy;    /* the scale factors */
     TPixel    color;     /* the color of the instance */
     /* etc. */
} Instance;
Instance instances[MAXINSTANCE];
```

The instance table for the scene in Figure 9-25 is illustrated in Figure 9-26. The instance table contains a reference to the basic object, i.e., a pointer to the object table or a string that is the name of the object that can be matched with the name stored in the object table. An integer array index for `object` is shown above, but a C/C++ pointer could be used as well as illustrated in previous sections. The instance table also contains *attributes* for each instance, which can be any geometric, visual or other properties that modify the appearance, location, scale, rotation, etc., of one instance of an object from another instance.

Each instance is a “transformed copy” of a basic object. One instance of a basic object has the same basic definition as another, i.e., each instance of a box contains the same entities (lines). Each instance, however, has a different location, size, color, etc. Often, the location, size and orientation of an instance is represented in a transformation matrix using the basic transformations of translation, scaling and rotation, respectively.



instance table: Instance instances[MAXINSTANCE]

0	“box”,tx1,ty1,sx1,sy1,color1
1	“box”,tx2,ty2,sx2,sy2,color2
2	“box”,tx3,ty3,sx3,sy3,color3
3	“triangle”,tx4,ty4,sx4,sy4,color4

CSL

object table: Object objects[MAXOBJ]

“box”, start
“triangle”, start

CSL

Entity data[MAX]:

x,y,pen

LL

Figure 9-26. Instance Data Structures.

The parameters in the instance table modify the appearance of the basic object when this instance is drawn. The object is transformed geometrically and visually as it is drawn. For this example, the instance drawing loop is sketched in Figure 9-27.
The coordinates are transformed by the instance transformation representing the location, size and orientation, and other visual properties can sometimes be implemented through simple graphics package calls. The instance table can be represented using a CSL or linked list structure, although the CSL is adequate for this example.

Picking objects on the screen means picking the instance of a basic object, which in turn

```
for( i = 0; i < #instances; i = i + 1 ) {
      GPenColor( instances[i].color );
      BuildTransform([M],instances[i].tx,instances[i].ty,
            instances[i].sx[i], instances[i].sy );
      for( each point 'j' of object[i] ) {
            Transform (data[j].x,data[j].y) by [M] into (x,y);
            if( data[j].pen[j] == MOVE )
                  GMoveTo(  x, y );
            else
                  GLineTo( x, y );
      }
}
```

Figure 9-27. An Example Loop to Draw Instances.

means picking a transformed copy of a basic object. Thus, the cursor coordinates must be compared to the transformed coordinates of each particular instance of a basic object. Another approach that is more computationally efficient is to transform the screen coordinates of the cursor into "basic definition coordinates" of each instance and then to compare these coordinates to the basic object coordinates. This involves multiplying the cursor coordinates by the inverse of the instance transformation of the object in question.

9.4 Hierarchical Instancing

To this point there is only one "level" of data. No instance is related in any way to another; each is simply "in" the scene. This may not be sufficient. For example, to reposition the "house" in the previous scene each entity must be moved individually. The present data structure has no concept of "house."

Another example is an airplane, with engines connected to wings connected to a fuselage. To properly represent and manipulate this data requires a *hierarchical* scene definition, or an *assembly.*

Extend the previous instance data structure with relational information, four pointers called `prev` (for previous), `next`, `parent` and `child`, which point to other instances in the instance table, which in turn point to four other instances. A graphical depiction of this structure is shown in Figure 9-28 where pointers are shown as horizontal and vertical arrows. Each box is an instance data "node" that contains the basic object reference and the geometric attributes, probably as a transformation matrix.

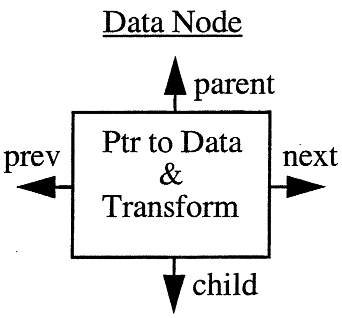


Figure 9-28. Graphical Depiction of Hierarchical Relationships between Entities.

In this representation, special significance is given to the parent and child pointers. They represent a hierarchical relationship between instances that relates the relative position, size and orientation of each child to its parent. The prev and next pointers are a doubly-linked list of children of a parent that has no such geometric significance.Linked list storage is required now. Figure 9-29 is an example of how an airplane could be modeled with such a structure.

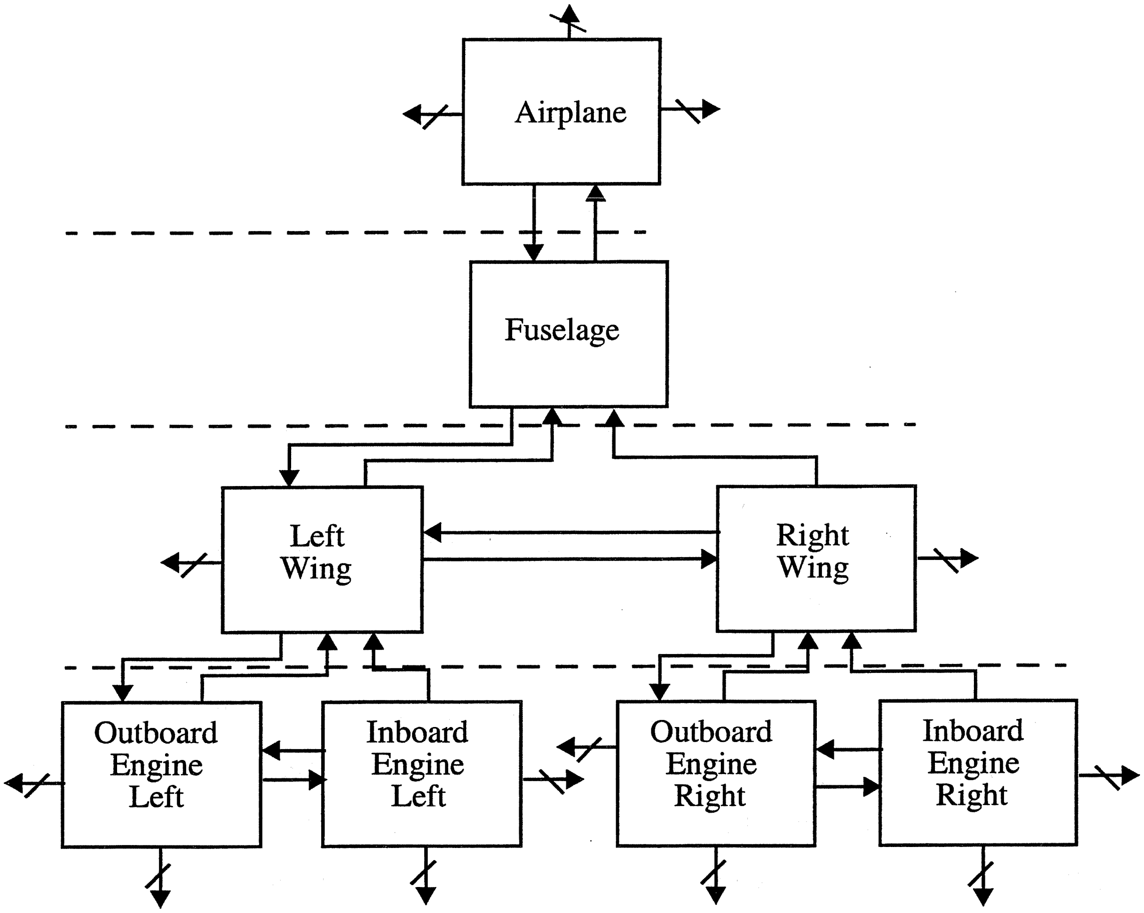


Figure 9-29. Airplane Example of Hierarchical Data Structure.

What is the "transform" in the data node? There are two possibilities: absolute (global) or relative (local). An absolute transform maps the data from local coordinates to world coordinates. Using absolute transforms, when the transform of any node is changed, all children must be changed appropriately. This is awkward, and somewhat defeats the hierarchic nature of the representation.

A relative transformation maps the child coordinates to its parent's coordinates. That is, each child is transformed relative to its untransformed parent. This "localizes" the transforms and makes the representation significantly more flexible. It also requires us to accumulate all transformations, parent to child, to "know" the absolute transform of any given child. In the example, the location, size and orientation of a wing that is defined in its own convenient coordinate system is defined by the location, size and orientation of the fuselage to which it is attached (i.e., its parent). Therefore, when traversing this hierarchical data structure, for example for drawing, it must be understood that transforming a node (data and associated transformation) in the tree also transforms all of its children. In other words, "move the plane, and everything connected to the plane moves too."

This hierarchical data structure is greatly aided by transformation matrices. For example, a division of a company has many designers working long hours developing the data for a wing. The designers naturally adopt a convenient coordinate reference frame in which to express the wing data, which we can consider for now as a list of points and pen codes. Similarly, another division of the company has developed the fuselage data and another the engine data, likewise selecting convenient reference frames (different than the others). Now, it is time to develop a model of the airplane with two identical wings (mirror images of each other) connected to the fuselage, each wing with two identical engines.

It would be unreasonable (and unnecessary) to ask one division to recompute their data in the reference frame of the other. Instead, we need to relate the reference frames, and use *instance transformations* to model the airplane structure as a hierarchy.

The wing data, W, defined in a wing reference frame (coordinate system), is first positioned relative to a fuselage, F, defined in its reference frame, by a transformation $\mathbf{M}_{W/F}$. Similarly, an engine, E, defined in an engine coordinate frame, is positioned relative to the wing by $\mathbf{M}_{E/W}$.

The coordinates of the wing expressed in the fuselage coordinate system W_F are:

$W_F = W * \mathbf{M}_{W/F}$ (W is the wing data in the wing coordinate system)

The coordinates of the engine expressed in the wing coordinate system E_W are:

$E_W = E * \mathbf{M}_{E/W}$

And now the engine can be expressed in the fuselage system as E_F:

$$E_F = E_W * \mathbf{M}_{W/F} = (E * \mathbf{M}_{E/W}) * \mathbf{M}_{W/F}$$

Notice that $E_F = E * (\mathbf{M}_{E/W} * \mathbf{M}_{W/F})$ or $E * \mathbf{M}_{E/F}$

where

$$\mathbf{M}_{E/F} = \mathbf{M}_{E/W} * \mathbf{M}_{W/F}.$$

This shows that as we "descend" from parent to child in the hierarchy, we can concatenate matrices to accumulate the net effect. This concatenation process continues as we further descend the structure. Consider, however, the transformations of the children of one parent, such as the fuselage. Transformations of one child do not affect transformations of other children with the same parent.

Consider the example illustrated in Figure 9-30. To compute the transformation relating the outboard engine left to the airplane, $\mathbf{M}_{OELA}$, we must pre-concatenate the transformations of each parent, from airplane to outboard engine left.

$$\mathbf{M}_{OELA} = \mathbf{M}_{OEL} * \mathbf{M}_{LW} * \mathbf{M}_F * \mathbf{M}_A$$

9.5 The Current Transformation Matrix

An efficient method for maintaining the transformations while traversing a hierarchical data structure is to use a "*current transformation matrix*," called **CTM**. The **CTM** holds the accumulated global transformations for the current node and all its parents.

When descending the data structure, parent to child, transformations are PRE-concatenated with the **CTM**, i.e. **CTM** = $\mathbf{M}_{child}$ * **CTM** (for row vector coordinates). The child node is then drawn with the current **CTM**: P' = P * **CTM**.

However, examine the **CTM** for the Left Wing versus the Right Wing of the Fuselage. After drawing one child of a parent, the starting **CTM** for the next child is the parent's **CTM**. This process repeats as the child becomes the parent, and so on. If we wish to maintain the prior parent's **CTM** as we display each parent-child-child in a depth-first manner, we must store all the prior

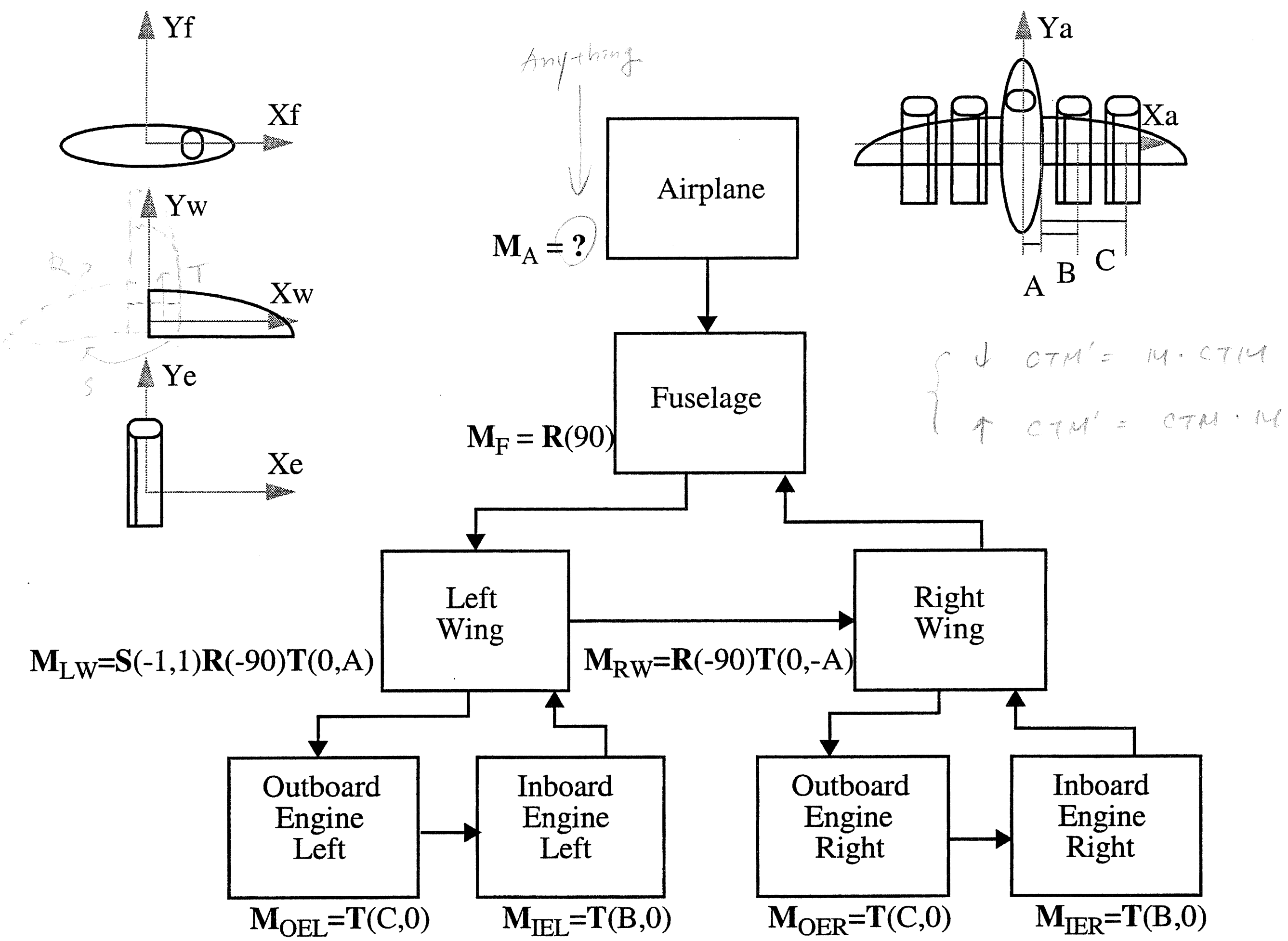


Figure 9-30. Numerical Airplane Hierarchy Example.

CTM's in a LIFO (last-in first-out) *CTM stack.* A stack is simply an array structure that has a top element index, or *stack pointer.*

Prior to concatenating a child's transformation with the **CTM**, we push the parent's **CTM** onto the stack. This is done by incrementing the stack pointer and copying the **CTM** into the stack entity at this pointer.

After the child has been drawn with its **CTM**, we must restore the prior **CTM** before drawing the next child by popping the top element off the stack into the **CTM**. This is done by copying the top element (pointed to by the stack pointer) into the **CTM** and decrementing the stack pointer. The stack entries, in this case, are transformation matrices. This can be implemented using a three dimensional array `CTMSTACK[i,j,k]`.

Figure 9-31 illustrates **CTM** tracing code for this hierarchical data structure.

The execution of the routine in Figure 9-31 using the data in Figure 9-32 is traced in Figure

```
void DrawObject ( ObjectPointer object, TransformMatrix ctm )
{
      TransformMatrix newctm; /* new matrix allocated each call */
      ObjectPointer child;

      MatrixMultiply: newctm ← object->xform × ctm
      DrawData( object->data, newctm );
      for(child=object->child; child != NULL; child=child->next )
            DrawObject( child, newctm );
}
```

Figure 9-31. Hierarchical Drawing Routine.

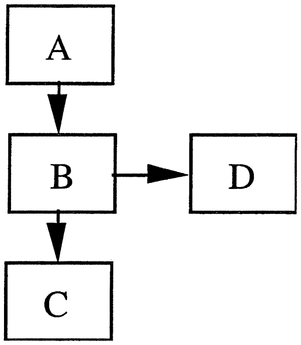


Figure 9-32. Example Hierarchy.

9-33. The execution of the routine in Figure 9-31 using the data in Figure 9-32 is traced in Figure

```
DrawObject( A, $\mathbf{I}$ )
      newctm$_1$ = $\mathbf{M}_A \times \mathbf{I}$
      DrawData( A, newctm$_1$ )
      for( child = B )
            DrawObject( B, newctm$_1$ )
                  newctm$_2$ = $\mathbf{M}_B \times$newctm$_1$
                  DrawData( B, newctm$_2$ )
                  for( child = C )
                        DrawObject(  C, newctm$_2$ )
                              newctm$_3$ = $\mathbf{M}_C \times$  newctm$_2$
                              DrawData( C, newctm$_3$ )
      ... for( child = D )
            DrawObject( D, newctm$_1$ )
                  newctm$_4$ = $\mathbf{M}_D \times$ newctm$_1$
                  DrawData( D, newctm$_4$ )
```

Figure 9-33. Trace of Execution DrawObject Routine.

9-33. This type of *modeling hierarchy* is used in a number of systems and is a standard in the *PHIGS* programming system.

Figure 9-34 is an example of a *procedural language* that implements a 3D modeling environment that produces similar effects to our hierarchical data structure. The system

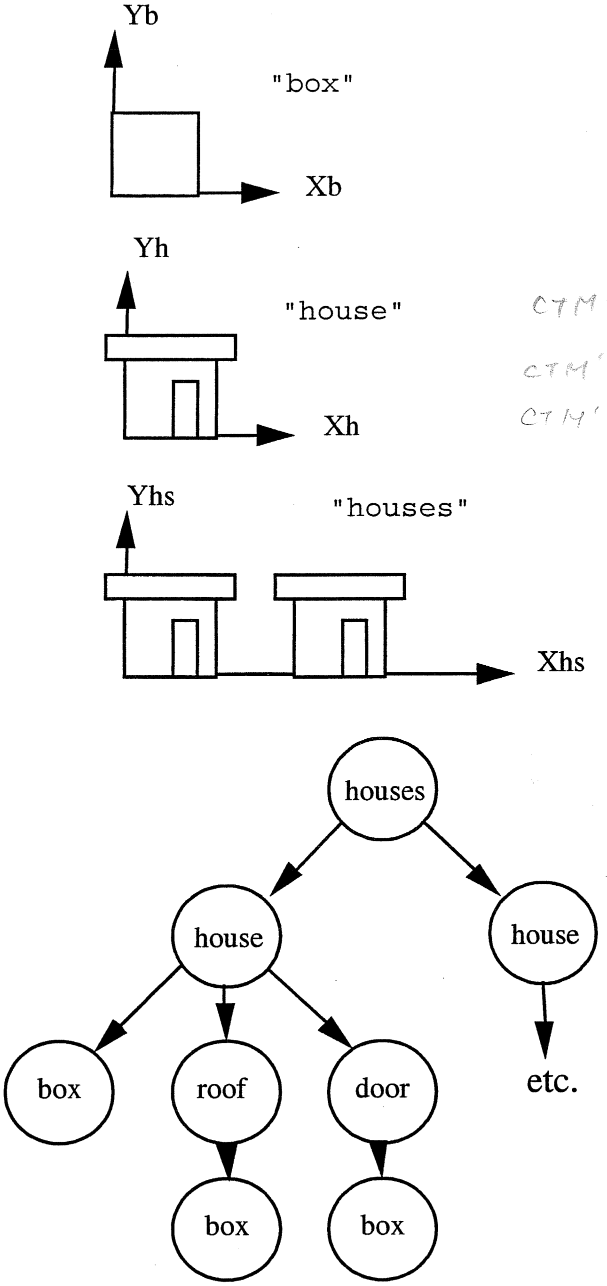
```
box := VECTOR_LIST  N=5
 0,    0
 0.25, 0
 0.25, 0.25
 0,    0.25
 0,    0;

roof := BEGIN_STRUCTURE
  TRANSLATE BY -.05,.25;
  SCALE BY 1.4,.2;
  INSTANCE OF box;
END_STRUCTURE;

door := BEGIN_STRUCTURE
  TRANSLATE BY .12,0;
  SCALE BY .2,.8;
  INSTANCE OF box;
END_STRUCTURE;

house := BEGIN_STRUCTURE
  INSTANCE OF box;
  INSTANCE OF roof;
  INSTANCE OF door;
END_STRUCTURE;

houses := BEGIN_STRUCTURE
  INSTANCE OF house;
  TRANSLATE BY .4,0;
  INSTANCE OF house;
END_STRUCTURE;
```



$CTM = [I]$

$CTM' = T(\,)\cdot[I]$

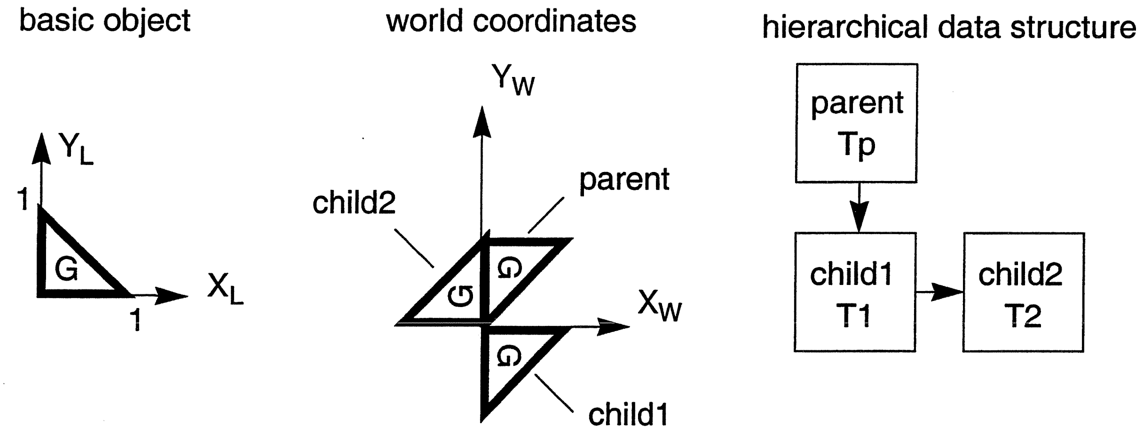
$CTM' = S\cdot T\cdot[I]$

Figure 9-34. The Evans & Sutherland PS300 Hierarchical Modeling Language.

automatically pushes the CTM before an INSTANCE call and pops the CTM afterwards.

9.6 Review Questions

1. Assume a program has displayed the hierarchical object above in a GRAFIC window using window to viewport mapping. The window surrounds the data and the viewport is the entire GRAFIC window. Describe, in step-by-step form, a picking algorithm for selecting an instance on the screen, given the cursor's viewport coordinates. Use the "first within tolerance" method. Show and explain all necessary equations and computations.
2. A hierarchical 2D data structure consists of a parent and two child instances of the same basic object. Show the relative transformations Tp, T1 and T2,in functional form. There are no scale transformations.

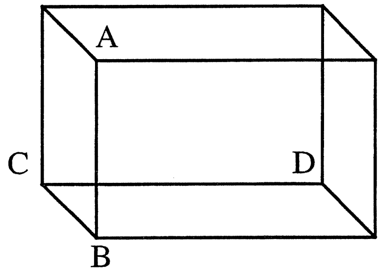


Chapter 10. 3D Rendering

Three-dimensional (3D) coordinates can be stored easily as data in the computer by adding the z coordinate. The problem is creating a two-dimensional (2D) *image* of a 3D *scene*.

There are three types of such *pictorial renderings*:

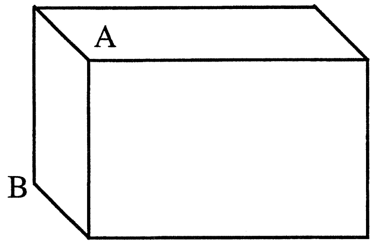
1. Wire Frame. The data is drawn by connecting points with lines or curves. This is the simplest computationally, but is spatially ambiguous. Can you see two different views in Figure 10-1?



Is line C-D closer or farther away than line A-B?

Figure 10-1. Ambiguity in Wire Frame Images.

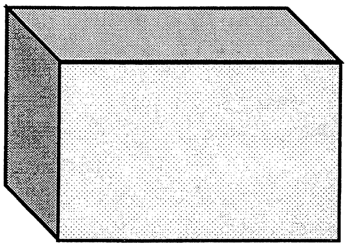
2. Hidden Lines Removed (HLR). Only parts of lines or curves which are not covered by other faces (surfaces) are drawn. HLR involves more complicated and time-consuming computations, but produces an image with less ambiguity.



Is point A closer or farther away than B?

Figure 10-2. Figure 10-1 with Hidden Lines Removed.

3. Hidden Surfaces Removed (HSR). HSR output requires raster displays. Those parts of faces not hidden by other faces are shaded (filled with pixels) with colors that indicate the intensity of light reflecting from the face. This involves more complex and time-consuming computations, and produces the least ambiguous images. Special lighting effects include reflection, refraction and shadows to make the image more realistic.



Shading is based on the intensity of light reflecting from the faces and indicates their relative angle to the eye.

Figure 10-3. Hidden Surface Image of Figure 10-1.

10.1 Depth Cueing

The basic problem in 3D rendering is to create the illusion of depth in the 2D image. Several techniques can be used singly or in combination to help remove ambiguity from the 2D image, especially for wire frame images.

1. Perspective projection causes 3D parallel lines to converge in 2D to a *vanishing point*, like the classic "railroad tracks" pictures.

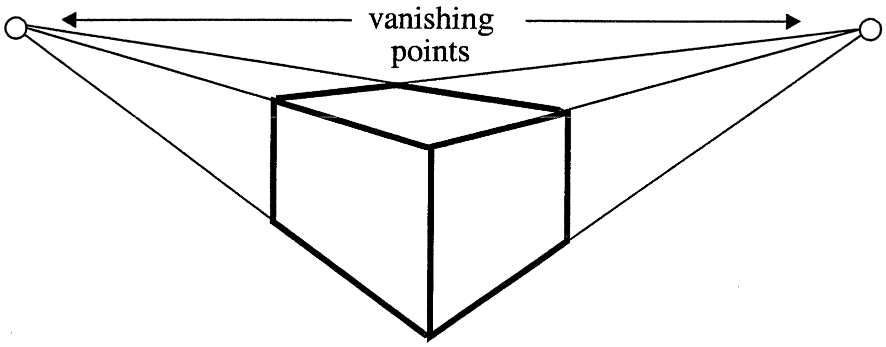


Figure 10-4. Perspective Image Showing Converging Lines.

Images that do not use perspective, that is, those in which all parallel lines in 3D remain parallel in 2D, are termed *orthographic*. The most common orthographic rendering, the four-view drawing, in essence "ignores" one coordinate when drawing each 2D orthographic view (Figure 10-5).

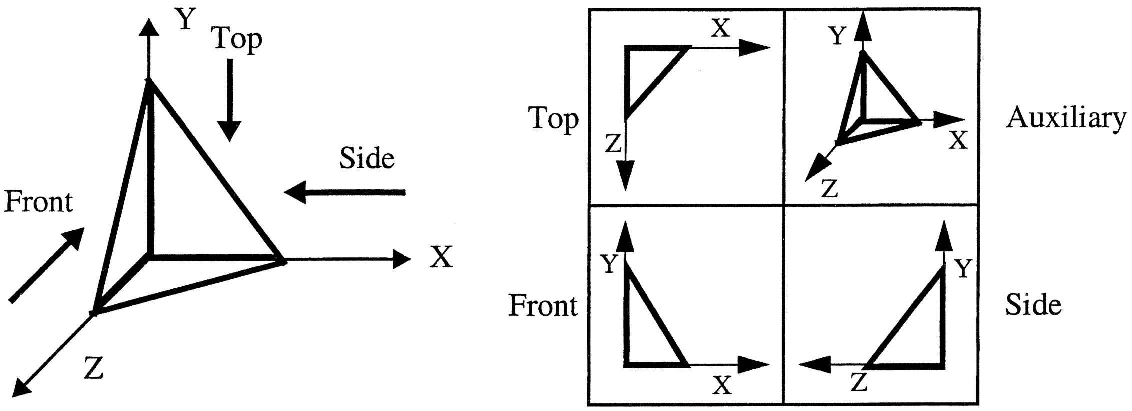


Figure 10-5. Orthographic Images in Engineering Drawing Arrangement.

2. Intensity Depth Cueing displays lines farther away from the eye at lower intensities. This can be done relatively easily in software for constant intensity lines on raster displays. Some high performance display systems offer this as an option.

3. Stereoscopic Viewing requires two projected views, one computed for the left eye and one for the right. Special viewing apparatus is often required. Color can also be used to cue the two images, such as the red and green lens stereo glasses worn in movie theaters.

4. Kinetic Depth revolves the object dynamically, causing lines farther away to move more than lines nearer to the eye. This requires dynamic 3D rotation, which is available on many display devices today (for additional cost, of course).

10.2 3D Data Structures

Points, curves and surfaces are the basic elements of 3D geometry. In general, considering only the mathematical forms, curves and surfaces are unbounded, meaning they have no start or end. Lines and planes extend infinitely. Practically, 3D geometry is bounded. A line has end points and a plane is bounded, i.e. has sides or boundaries. The terms vertex, edge and face are used to describe the topological elements of 3D objects related to points, curves and surfaces of geometry, respectively. A face is a surface bounded by edges, an edge is a curve bounded by vertices, and a vertex is a point on an object. For example, the cube in Figure 10-6 consists of topological elements of eight vertices, 12 edges bounded by these vertices, and 6 faces bounded by these edges.

Geometrically, there are 12 points, each edge is a curve that is a straight line, each face is a planar surface. As another example, a cylinder has no vertices, two circular edges, and three faces, two planar (bounded by the edges) and one cylindrical (bounded by the edges).

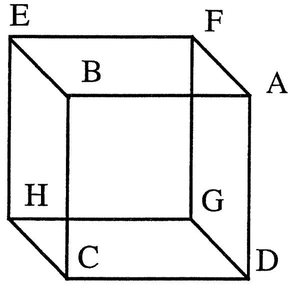


Figure 10-6. A Cube and its Eight Vertices.

Pictorial representations require different data storage methods. A box, for example, with vertices named A-G, as shown in Figure 10-6, can be stored several ways. The type of storage method can be dictated by the type of rendering(s) desired.

Wire Frame Representations

This is the "lowest level" of the data storage methods. Only curves (lines) are needed, so any of the storage methods described in the previous chapter can be used. In Figure 10-6, the data structure would store the 12 lines: AB, BC, CD, DA, EF, FG, GH, HE, BE, CH, FA, GD.

One approach would be to store the two end points of each line, i.e., (x_1,y_1,z_1) and (x_2,y_2,z_2). Notice that this duplicates each point three times (three lines meet at a point in a box), which can cause redundant calculations when transforming the object.

Another storage method, called points and lines, stores points and lines separately. A line consists of references to its two end points, analogous to saying "the line starts at vertex "i" and ends at vertex "j." The points and lines representation of the cube is shown in Figure 10-7. Notice that each vertex is stored once, affording efficiency in space and computation time, because transforming the object involves transforming each vertex only once.

In general, edges can be other than straight lines.This would make it necessary to store the type of curve or its equation with each edge. The "edge" is still a path between vertices, but now the description of this path, the curve, would be explicitly given.

points

index	x[]	y[]	z[]
1	x_A	y_A	z_A
2	x_B	y_B	z_B
3	x_C	y_C	z_C
4	x_D	y_D	z_D
5	x_E	y_E	z_E
6	x_F	y_F	z_F
7	x_G	y_G	z_G
8	x_H	y_H	z_H

lines

index	Start[]	End[]
1	1	2
2	2	3
3	3	4
4	4	1
5	5	6
6	6	7
7	7	8
8	8	5
9	2	5
10	8	3
11	6	1
12	4	7

Topology

Figure 10-7. Points and Lines Data Structure for the Cube in Figure 10-6.

Hidden Line Removal Representations

Removing hidden lines (curves) involves removing curves and parts of curves that are obscured by faces of the object. The data structure must store more than just lines and points, it must “know about” faces. Therefore, we must add a face structure that represents a surface that is bounded by edges.

In the cube of Figure 10-6, the faces are planes bounded by lines, i.e., polygons. Polygons are the simplest form of a face, and are very common among 3D data representations. A first storage method for polygons might be a list of points, ABCD, ADHE, EFGH, BCGF, ABFE, CDHG, where each adjacent pair of points in the list, including the last to the first, is a line. This suffers from the inefficiencies in space and redundant computations noted previously for lines.

Alternatively, a polygon can be stored as a list of references to vertices, similar to the points and lines logic, called a “points and polygons” structure. The lines structure of Figure 10-7 is replaced by a polygons structure that has four indices per polygon (in this limited example) corresponding to the four vertices. Edges would be understood implicitly to be lines between adjacent vertices, requiring that the list of vertices be properly ordered.

Hidden Surface Removal Representations

The data required for hidden surface removal is an extension of that for hidden line removal. In addition to the geometry of each face of the object, the light reflectance properties of the face must also be stored. Also, the lighting conditions of the surrounding environment are needed.

10.3 Planar Geometric Projections

Planar geometric projection involves transforming 3D coordinates onto an *image plane* (also called *projection plane*) in space, and using the resulting plane coordinates to draw the image. There are two forms of planar projections:

1. Parallel Planar Projection "cop"

Parallel projections have the *center of projection*, the "eye point" toward which points are projected, infinitely far from the image plane. There are two kinds of parallel projections:

(A) Orthographic Parallel Planar Projection: The projection direction is normal to the image plane. There are two kinds of orthographic projections:

Axonometric: the image plane is normal to a principal plane, i.e. top, front and side views.

Isometric: the image plane is not normal to a principal plane.

(B) Oblique Parallel Planar Projection: The projection direction is not normal to the image plane.

cabinet: cot(angle) = 1/2 (angle is about 63 degrees)

cavalier: 30 or 45 degrees.

2. Perspective Planar Projection

The center of projection is a finite distance from the image plane.

10.4 Perspective Projection

First we will study *wireframe perspective* drawing. Consider an *eye coordinate system* XYZ_e shown in Figure 10-8.

Let the XY plane (Z=0) be the image plane, also called the *picture plane*. A point in 3D space, (X,Y,Z), is projected onto the image plane by finding the intersection with the image plane of a line from the data point in question (X,Y,Z) to a given center of projection (COP). This line is called a *projector*. Thus (X_{1p},Y_{1p}) is the projected 2D image of (X_1,Y_1,Z_1), and similarly for (X_{2p},Y_{2p}) and (X_2,Y_2,Z_2).

After perspective projection, lines remain lines and planes remain planes (this will be

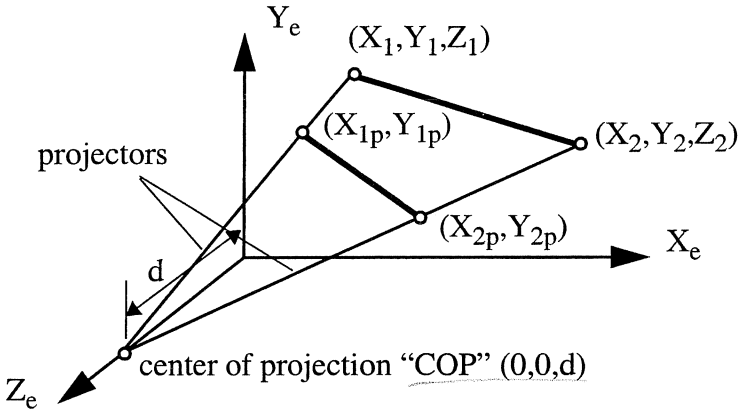


Figure 10-8. The Eye Coordinate System and Projected Lines.

shown mathematically later). Therefore, the image of a 3D line from point 1 to point 2 is the line connecting the projected points. Angles, however, are not preserved after perspective projection.

To derive the perspective equations, first look down the X_e axis (Figure 10-9).

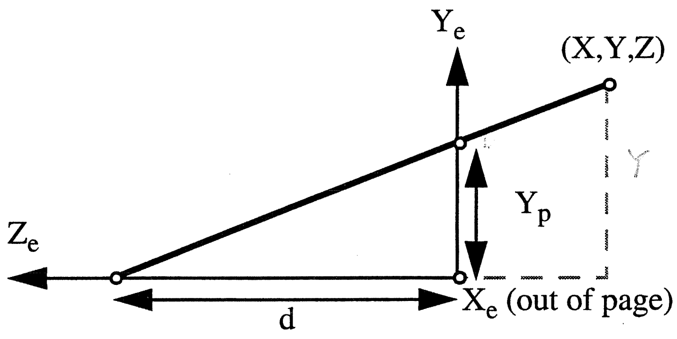


Figure 10-9. Viewing a Line Looking Down the X_e Axis.

Using similar triangles, relate Y_p to d, Y and Z:

$$Y_p = Y \frac{d}{d-Z}$$

$$\frac{Y_p}{d} = \frac{Y}{d-z}$$

Now find X_p by looking down the Y_e axis (Figure 10-10).

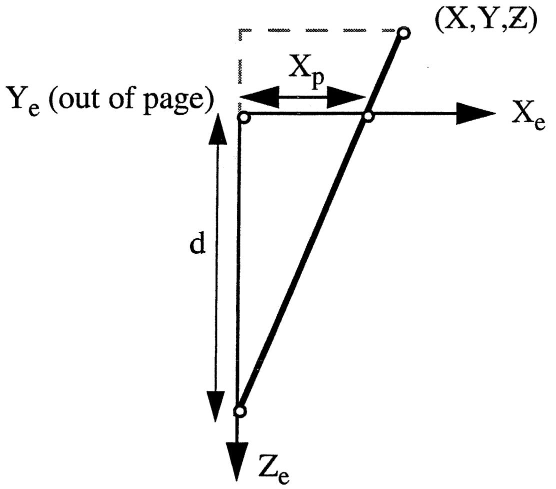


Figure 10-10. Viewing a Line Looking Down the Y_e Axis.

$$X_p = X\,\frac{d}{d-Z} = X\left(1-\frac{Z}{d}\right)^{-1}$$

$\frac{d}{d-Z}$, also shown as $\frac{1}{\left(1-\frac{Z}{d}\right)}$ is the *perspective factor* and is a function of the distance d of the eye from the image plane and the z coordinate of the given point.

After perspective, when the data lies on the picture plane, we can use the standard two-dimensional techniques to draw it.

Examine the perspective factor more closely.

1. When z → -∞, i.e., for data points far from the eye:

$$\lim_{z \to -\infty}\left(\frac{d}{d-Z}\right) = 0$$

Therefore (Xp,Yp) → (0,0). The points converge to the origin.

2. When d → ∞, i.e., when observing from infinitely far away:

$$\lim_{d \to \infty}\left(\frac{d}{d-Z}\right) = 1 \qquad X_p = X\left(\frac{d}{d-\overset{0}{Z}}\right)$$

Therefore (Xp,Yp) → (X, Y). This is parallel projection. So, perspective projection from infinitely far away becomes parallel projection.

"projectors become parallel"

3. When z > d, i.e., the data point is behind the COP:

$$\left(\frac{d}{d-Z}\right) = \text{“a negative number”}$$

Therefore (Xp, Yp) = (a negative number) × (X, Y). This causes inversion of the image, or “false projection.”

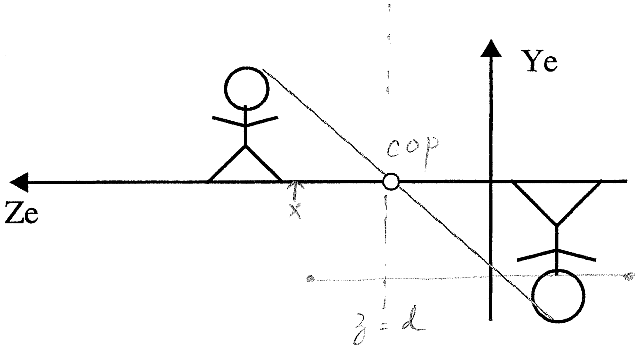


Figure 10-11. Inversion of Data Behind the COP.

4. When z → d, i.e., a point in the plane of the eye:

$$\lim_{z \to d}\left(\frac{d}{d-Z}\right) = \infty$$

The perspective factor approaches infinity, causing a severe lens distortion effect as objects approach the COP plane. Computationally, this is a problem that must be handled in code, as described later.

5. When z = 0, i.e. data points on the projection plane:

$$\left(\frac{d}{d-Z}\right) = 1$$

Points on the projection plane are not affected by perspective.

Figure 10-12 summarizes these different regions along the Z axis and their projection characteristics.

There are other definitions for the eye coordinate system. One of the early definitions for the eye coordinate system places the COP at the origin and the image plane at the location z=d in a *left-handed reference system.*

To derive the perspective factor for this system, again look at parallel orthographic projection, i.e., looking down the X_e axis (Figure 10-14).

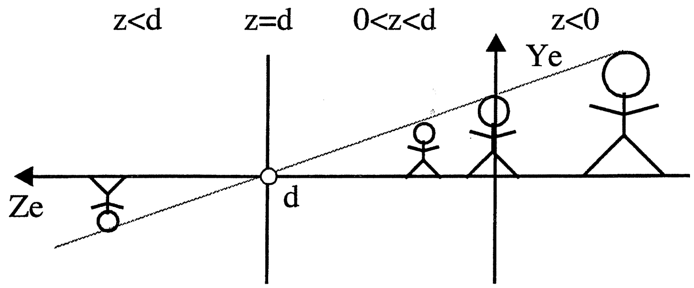


Figure 10-12. Effects of Data Z Values on Projection.

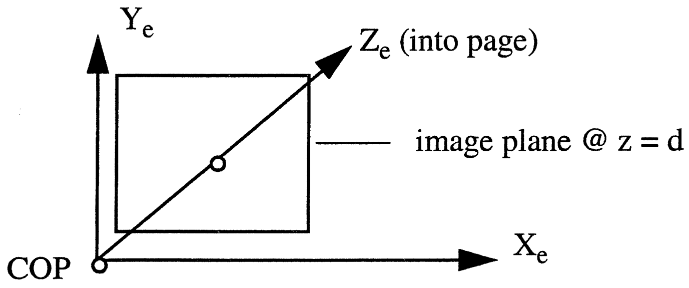


Figure 10-13. A Left-Handed Eye Coordinate System.

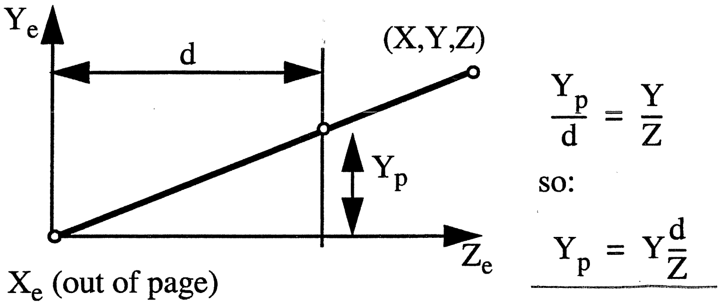


Figure 10-14. Looking Down X_e in the Left-Handed Eye Coordinate System.

Similarly, we find $X_p = X\frac{d}{Z}$. This appears simpler than the previous perspective equations, but has limitations for some applications, such as hidden surface removal, as we'll see later.

10.5 3D Modeling Transformations

Before advancing, consider 3D transformations, this time going directly to matrix and functional forms. The 3D vector consists of four elements: [x, y, z, 1], and the *homogeneous transformation matrix* is four by four.

1. Translation: **T**(Tx, Ty, Tz)

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1]\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ Tx & Ty & Tz & 1\end{bmatrix}$$

$V' = V \cdot M$ (row vector)

$V' = M \cdot V$ (col vector)

$$\uparrow$$

$$\begin{bmatrix}1 & 0 & 0 & Tx\\ & 1 & 0 & Ty\\ & & 1 & Tz\\ & & & 1\end{bmatrix}$$

2. <u>Scaling</u> : **S**(Sx, Sy, Sz)

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1]\begin{bmatrix}Sx & 0 & 0 & 0\\ 0 & Sy & 0 & 0\\ 0 & 0 & Sz & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

3. <u>Rotation</u>: $\mathbf{R}_X(\theta)$, $\mathbf{R}_Y(\theta)$, $\mathbf{R}_Z(\theta)$. ← $R(\theta, X)$ $R_x(\theta)$

There are three basic rotations about each principal axis:

Rotation about X, $\mathbf{R}_X(\theta)$:

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1]\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & \sin\theta & 0\\ 0 & -\sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Rotation Matrix Clues:

1. Rotating about X (Y or Z) does not change X's (Y's or Z's)
2. Row 4 and column 4 are always the same.
3. The problem is to locate the cos (C) and sin (S) elements:

$$\begin{matrix}C & S\\ S & C\end{matrix}$$

and to place the minus sign before one of the S terms. To do this, rotate an obvious point, like [0,1, 0] about the axis 90°. For X:

$$(0,0,1) = (0,1,0)\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & C & S & 0\\ 0 & -S & C & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

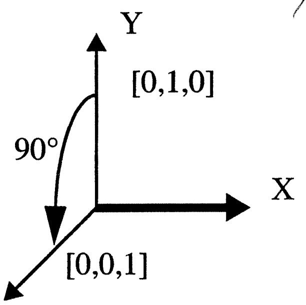
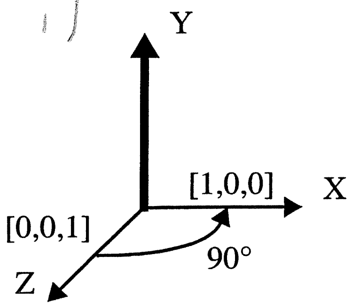


Figure 10-15. Test Rotation about the X Axis.

Rotation about Y, $\mathbf{R}_Y(\theta)$:

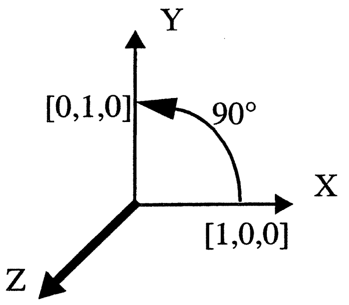
$$(0,0,1)\begin{bmatrix} c & 0 & -s & 0 \\ 0 & 1 & 0 & 0 \\ s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix}x' & y' & z' & 1\end{bmatrix} = \begin{bmatrix}x & y & z & 1\end{bmatrix}\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Rotation about Z, $\mathbf{R}_Z(\theta)$:

$$\begin{bmatrix}x' & y' & z' & 1\end{bmatrix} = \begin{bmatrix}x & y & z & 1\end{bmatrix}\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



10.6 Homogeneous Coordinates

We have simply ignored the fourth coordinate of vectors and the last column of matrices. For representing translation in matrix form and to make the transformation matrices square, we have set the fourth coordinate of a vector to 1: [x,y,z,1] and the last column of matrices to $[0,0,0,1]^T$.

Now we will generalize this notation and give deeper geometric interpretation to it. A point in 3D space is a projection of a point in 4D space given by the coordinates [wx, wy, wz, w], the *homogeneous coordinates* of a point.

Define *normalized homogeneous coordinates* as coordinates having a unity value of w. The [x, y, z] values are the 3D space coordinates only when w=1, i.e., only if the coordinates have been normalized. Thus, [2, 4, 6, 2], [0.5, 1.0, 1.5, 0.5] are the same normalized coordinates, [1, 2, 3, 1], which is the 3D point [1,2,3].

In general, to find the normalized 3D coordinates, simply divide by the "w" coordinate, i.e.,

$$\left[\frac{wx}{w}, \frac{wy}{w}, \frac{wz}{w}, 1\right]$$

To see the benefits of this convention, examine a general 4x4 homogeneous transformation matrix:

$$\left[\begin{array}{ccc|c} a & b & c & p \\ d & e & f & q \\ g & h & i & r \\ \hline l & m & n & s \end{array}\right]$$

1. (a-i): Scaling and rotation elements.
2. (l-n): Translation.
3. s: This is the homogeneous scaling element. Examine what happens when we transform a normalized coordinate by a matrix with a variable 's' element:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & s \end{bmatrix} \rightarrow \begin{bmatrix} x & y & z & s \end{bmatrix}$$

"After normalization," the coordinates are $\begin{bmatrix} \frac{x}{s} & \frac{y}{s} & \frac{z}{s} & 1 \end{bmatrix}$. The net effect is to uniformly scale the 3D coordinates by s^{-1}.

4. p,q,r: These are "projection elements". Begin by examining the effects of a transformation with a variable 'r' and 's' elements:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & s \end{bmatrix} \rightarrow \begin{bmatrix} x & y & z & (rz+s) \end{bmatrix}$$

Now normalize,

$$\begin{bmatrix} x & y & z & (rz+s) \end{bmatrix} \rightarrow \begin{bmatrix} \frac{x}{rz+s} & \frac{y}{rz+s} & \frac{z}{rz+s} & 1 \end{bmatrix}, \text{if } ((rz+s) \neq 0)$$

Letting $s = 1$ and $r = -\frac{1}{d}$, where d is the distance from the eye to the image plane,

$s + rz = 1 - \frac{z}{d}$

$$\begin{bmatrix} x\frac{1}{1-\frac{z}{d}}, & y\frac{1}{1-\frac{z}{d}}, & z\frac{1}{1-\frac{z}{d}}, & 1 \end{bmatrix}$$

This is "perspective projection." Thus, due to the convention that normalization introduces division, perspective can be represented as a transformation in matrix form. The perspective matrix, **P**, is:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{d} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

normalized : $w = 1$

or

$w = 0$

$(1,2,3,1) - (2,3,4,1)$

$= (-1,-1,-1,0)$

Where is the point [x,y,z,0]? Following the homogeneous convention, normalize the point:

$$\begin{bmatrix} \frac{x}{0} & \frac{y}{0} & \frac{z}{0} & \frac{0}{0} \end{bmatrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & 1 \end{bmatrix}$$

The interpretation of the homogeneous vector [x,y,z,0] is a point at infinity on a line from the origin through [x,y,z]. In other words, the vector [x,y,z,0] is a direction vector. This is a consistent interpretation when one considers that subtraction of two position vectors with the same 'w' coordinates will result in a direction vector with a 0 'w' coordinate. The ability to represent points at infinity is another benefit of homogeneous representation.

Project the point [0, 0, d, 1]:

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

This is the point at infinity on a line from the origin through [0,0,d]. Therefore, the COP projects to the point at infinity on the Z_e axis.

10.7 Viewing Transformations

The projection of 3D data onto an image plane at z = 0 requires that the data be defined relative to the eye coordinate system, where the observer's eye, i.e. the COP, is at a point (0,0,d) and is looking toward (0,0,0). In general, one may want to observe data from some other position and direction that, in general, can be specified as a *view vector*. The data and view vector are given in *world coordinates*. Consequently, data must be transformed from world coordinates into *eye coordinates* before perspective can be performed (analogous to translating before scaling when

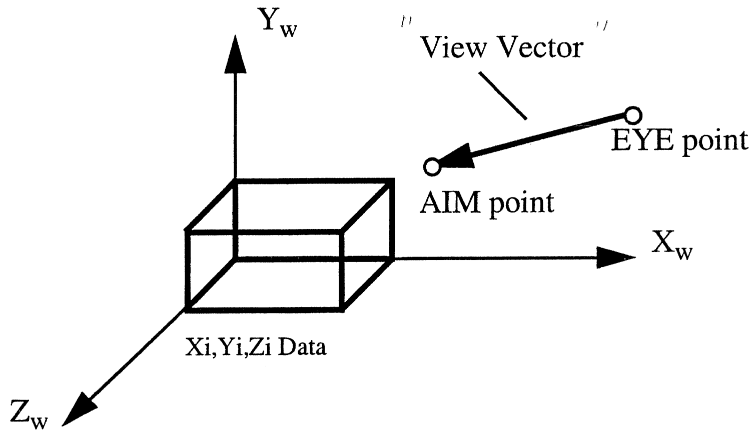


Figure 10-16. World Coordinate System and View Vector.

"scaling-about-a-point"). This transformation, called the *viewing transformation*, **V**, relates the eye coordinate system to the world coordinate system.

Where is the eye coordinate system? One answer is to use the view vector to define the eye coordinate system.

$AIM_w \rightarrow (0, 0, 0)_e$

$EYE_w \rightarrow (0, 0, d)_e$

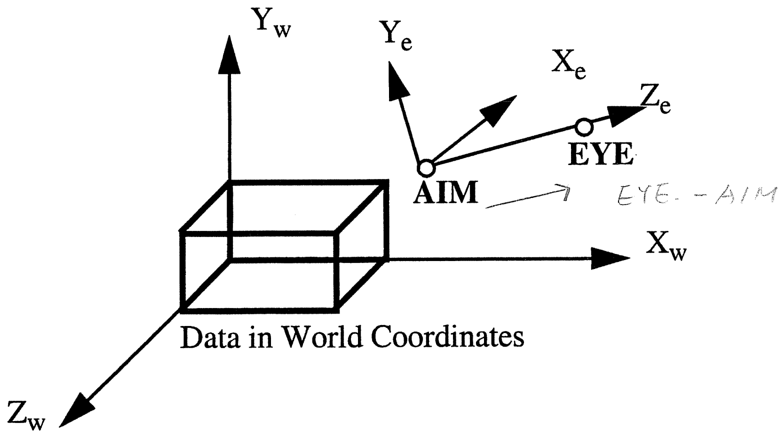


Figure 10-17. World and Eye Coordinate Systems.

For now, also make the arbitrary specification that the Y_e axis will be "vertical," i.e. the plane X_e=0 will be perpendicular to the plane Y_w=0. This means that the observer stands up-right with respect to Y_w when viewing the data in the world coordinate system. This specification will be relaxed later. Of course, the plane $Z_e = 0$ is the image plane, and the Z_e direction is the normal to the image plane.

The objective is a viewing transformation, **V**, that transforms a point in world coordinates into the corresponding point in eye coordinates:

$P_e = P_w \mathbf{V}$ p : point (position vector)

Visualize the process as transforming the view vector to align the Z_e axis with the Z_w axis. This can be accomplished using three basic transformations:

1. Translate the vector so that $\mathrm{AIM}_w \rightarrow (0,\ 0,\ 0)_e$
2. Rotate the result so that the transformed EYE' swings into the X_w=0 plane.
3. Rotate this result so that EYE" swings down onto the Z_w axis.

The matrix equation becomes:

$$P_e = P_w\, \mathbf{T}(?)\, \mathbf{R}(?,?)\, \mathbf{R}(?,?)$$

Hence,

$$\mathbf{V} = \mathbf{T}(?)\, \mathbf{R}(?,?)\, \mathbf{R}(?,?)$$

Step one. This is just a translation:

$$P' = P_w\, \mathbf{T}(\text{-AIM})$$

After the translation, the situation is as shown in Figure 10-18.

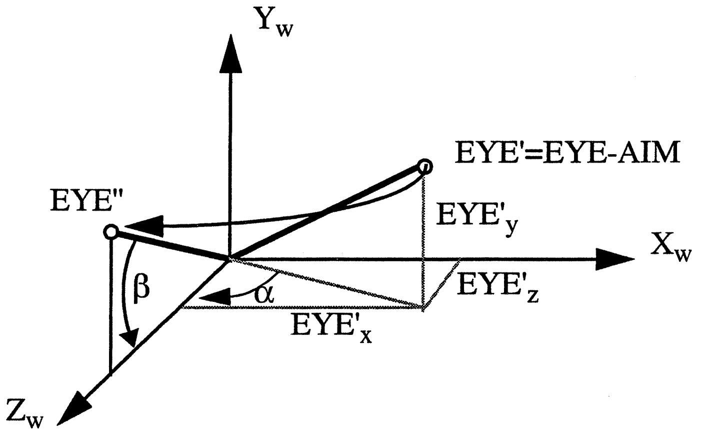


Figure 10-18. View Vector After Translation.

Step 2. Rotate α about Y_w to swing EYE' into the X_w=0 plane.

$$\text{Using trig:}\quad \alpha = \underset{\uparrow}{-}\tan^{-1}\left(\frac{\mathrm{EYE'}_x}{\mathrm{EYE'}_z}\right)$$

Check the sign by noting the rotation direction assuming positive values. Note that EYE'$_z$

can be 0, and that the $\tan^{-1}$ function returns angles between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, therefore we must code this using the ATAN2(EYE'_x,EYE'_z) function.

There is still a problem, however, in that EYE'_x and EYE'_z may both be zero if the view vector is parallel to the Y_w axis. In this case, ATAN2cannot be called. Set $\alpha = 0$.

Step 3. Swing EYE" down by β to (0,0,d) by rotating about X_w.

$$\beta = \tan^{-1}\left(\frac{EYE'_y}{\sqrt{EYE'^{\,2}_x + EYE'^{\,2}_z}}\right)$$

To check the transformations, try "data points" AIM and EYE. You should know what the results should be: (0,0,0) and (0,0,d).

For example, consider Figure 10-19. We first compute EYE' = EYE - AIM = (0,1,0). Then,

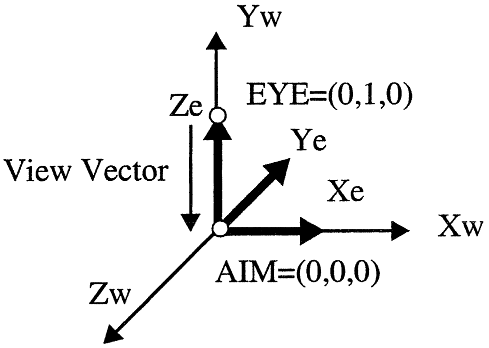


Figure 10-19. Example View Vector.

Xe = 1, 0, 0, 0

Xe_e = 1, 0, 0, 0

Ye = 0, 0, -1, 0

Xw_e = 1, 0, 0, 0

Ze = 0, 1, 0, 0

following the steps above,

$$\alpha = -\tan^{-1}\left(\frac{\text{EYE'}_x}{\text{EYE'}_z}\right) = -\tan^{-1}\left(\frac{0}{0}\right) = 0\,(\text{by assignment})$$

$$\beta = \tan^{-1}\left(\frac{\text{EYE'}_y}{\sqrt{\text{EYE'}_x{}^2 + \text{EYE'}_z{}^2}}\right) = \tan^{-1}\left(\frac{1}{\sqrt{0+0}}\right) = \frac{\pi}{2}$$

$$\mathbf{V} = \mathbf{T}(0, 0, 0)\mathbf{R}(0, \text{Y})\mathbf{R}\left(\frac{\pi}{2}, \text{X}\right)$$

$$\mathbf{V} = (\mathbf{I})(\mathbf{I})\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\frac{\pi}{2} & \sin\frac{\pi}{2} & 0 \\ 0 & -\sin\frac{\pi}{2} & \cos\frac{\pi}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As a check, we transform AIM and EYE,

AIM **V** = (0,0,0),

EYE **V** = (0,0,1).

Also note that the XYZ_e axes, treated like data, will align with the XYZ_w axes when transformed by **V**.

The EYE-AIM method just described can be far too restrictive for some applications. Imposing the condition that "d" is the distance between EYE and AIM can be awkward. It may be useful to consider this EYE-AIM view vector approach as an internal representation for the viewing information and give the user alternative methods for specifying the viewing position, direction and perspective distance.

An example of one alternative is using polar (or spherical) coordinates: R, azimuth and elevation, for a *viewing sphere* with a given center C. This is useful for viewing an unchanging object as though it were at the origin of a sphere of radius R. Consider viewing the earth as an example. Azimuth is the angle of rotation about the earth's axis, i.e. east-west motion, which in XYZ_w notation means a rotation about the Y axis. Elevation is the angle from the equator, i.e. north-south motion, that is a rotation about the X axis.

The radius of the viewing sphere, R, and its center, C, are usually computed to surround the 3D data. One can compute a "bounding sphere" for the data, or more simply, compute a

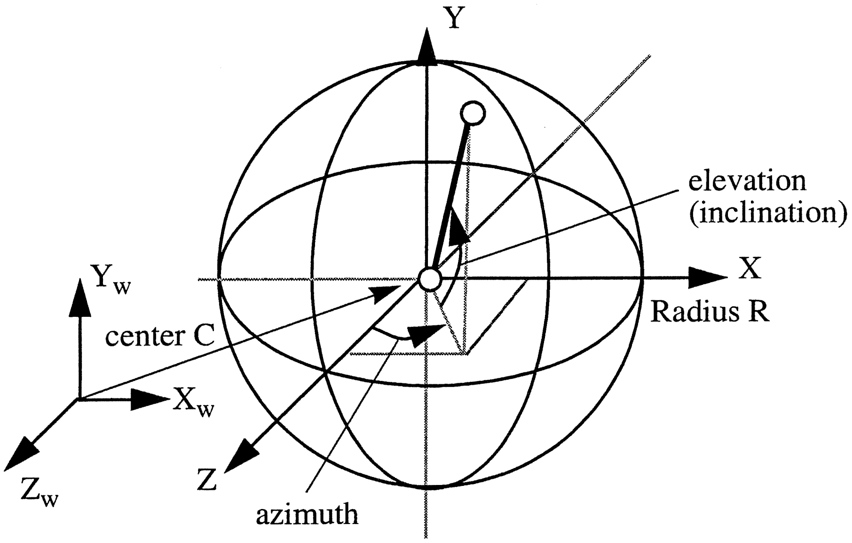


Figure 10-20. Viewing Sphere and Polar Coordinates.

parallelepiped with corners [xmin,ymin,zmin] and [xmax,ymax,zmax]. The sphere center C is then the midpoint of the diagonal connecting the corners, and R is half the length of the diagonal.

Given the viewing sphere, the user specifies viewing information as "elevation up, azimuth to the right." The problem is to compute EYE and AIM from this information, and then compute **V** as before. One approach is to set AIM = C, i.e. always look toward the center of the sphere. After all, this is the center of the data. To compute EYE, think of transforming a vector of length d along the Z axis (parallel to Z_w) through C by the elevation and azimuth rotations, then adding this vector to AIM:

$$EYE = [0,0,d,0]\, \mathbf{R}_X(\text{ -elevation })\, \mathbf{R}_Y(\text{ azimuth }) \oplus AIM$$

$EYE - AIM = (0, 0, d, 0)\, R_x\, R_y$

$P_w = P_e M$

$P_e = P_w M$

$M = R_x R_y$

$M^{-1} = R_y^{-1} R_x^{-1}$

$P_w M^{-1} = P_e$

10.8 Three Dimensional Clipping

There are several approaches to three-dimensional clipping, all of which have as their objective clipping lines in 2D and/or 3D to remove parts of lines that would fall outside the window.Three approaches will be presented in the following sections.

10.8.1 "Z-Clipping" with 2D Window Clipping

We saw that as z → d, perspective causes severe distortion i.e.

$$\frac{d}{d-z} \rightarrow \infty$$

and z > d produced an inverted image. Also, data far from the observer ($z_e << 0$) become possibly distracting dots at the origin. To eliminate these effects, two clipping planes parallel to the image plane are introduced:

1. a *near* clipping plane at $z = z_n$ to eliminate inversion and distortion,
2. a *far* clipping plane at $z = z_f$ (optionally) to provide a limited sight distance.

Clipping on the near plane is required prior to projection to avoid infinite projection factors and inverted data. Clipping on the far plane is optional and is useful for eliminating data "too far to see" and to produce special effects such as cross-sectioning.

Clipping on the near and far planes is termed "*z-clipping*." One approach to 3D clipping, based on the idea of utilizing existing 2D software for clipping and window to viewport mapping, is to:

1. clip the 3D line on the near and far planes (z-clip),
2. project the line into 2D on the image plane,
3. clip the projected line against the 2D window and then map it to the viewport.

Z-clipping can be implemented using logic based on the 2D Cohen-Sutherland clipping algorithm. The end point classification can be a simple Boolean value that signifies "visible" or "invisible." This code is computed based on the z coordinate alone as illustrated in Figure 10-21.

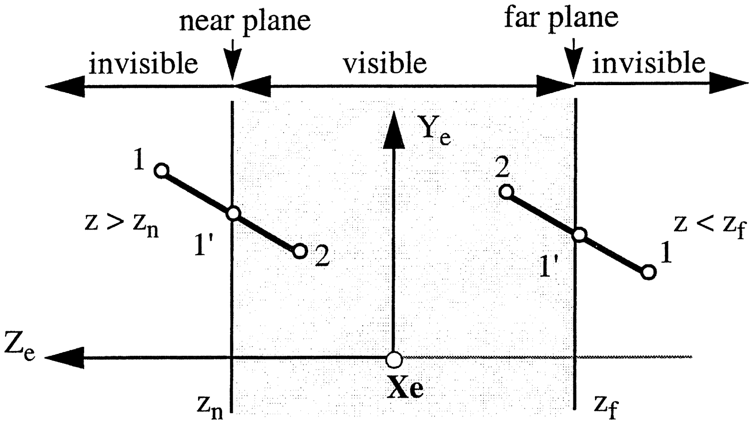


Figure 10-21. Z-Clipping End Point Regions.

To compute a clipped end point, use the "symmetric equations" for a 3D line:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} = \frac{z - z_1}{z_2 - z_1}$$ $z = z_n / z_f$

where [x,y,z] are the coordinates of any point on the line between the known points $[x_1,y_1,z_1]$ and $[x_2,y_2,z_2]$.

For a line crossing one of the planes, we know that z', the clipped z coordinate, will be z_n on the near plane and z_f on the far plane. Therefore, the x' and y' of point 1' shown in Figure 10-21 can be computed.

Lines can be rejected as wholly invisible during z-clipping and thus would not be further processed.

The process of "drawing a line" resembles a *pipeline* of operations starting with the end points in 3D world coordinates and ending with a clipped, projected line in 2D viewport coordinates. This is often called the *3D pipeline*. For the Z-Clipping approach to 3D clipping, the pipeline appears as shown in Figure 10-22.

Across the top of the pipeline element boxes are parameters that control the execution of each element of the pipeline. Across the bottom are the dimensionality and reference frame for the coordinates entering and exiting each element.

The diagram in Figure 10-22 can be used as a model for implementing the pipeline. Each block can be coded as a routine that inputs the end points of the line in 2D or 3D. The first routine, e.g. "Draw3DLine(x1, y1, z1, x2, y2, z2)," would be called to draw a line given the 3D local

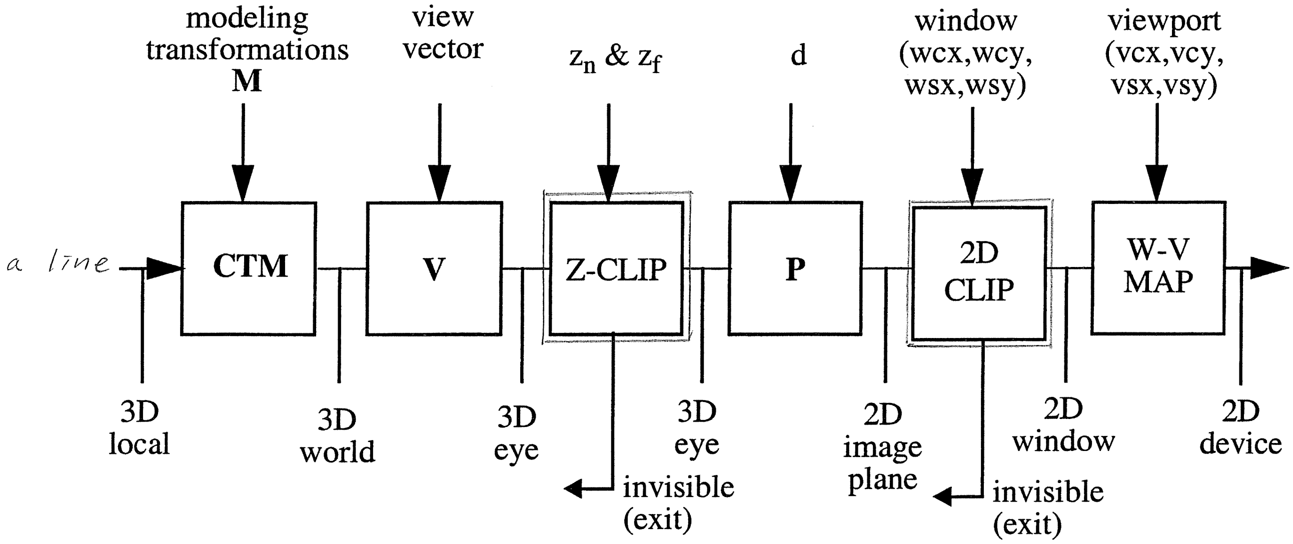


Figure 10-22. The 3D Pipeline for Z-Clipping.

coordinates of the end points. The application can then use this routine without concern for all the downstream details of its operation, thus forming a 3D package of sorts.

10.8.2 3D Frustum Clipping

Rather than perform clipping in two steps, 3D Z-clipping and then 2D window clipping after projection, the process can be accomplished one clipping step. This consolidates the elements of the pipeline for both efficiency and simplification of coding, but requires somewhat more complex computations.

Begin by visualizing four planes, called top, bottom, left and right, that pass through the COP and two corners of the window on the projection plane as shown in Figure 10-23. These four planes, combined with the near and far planes, form a six-sided "*viewing frustum*" bounded by the top, bottom, left, right, near, and far clipping planes. The viewing frustum bounds a "*viewing volume*." Points inside the viewing volume will be visible and points outside it will be invisible. Therefore, clipping a line against the 6 sides of the viewing frustum prior to projection will guarantee that the final 2D line lies within the window after projection.

Imagine holding the viewing frustum up to your eye and aiming it somewhere. The side clipping planes limit your "*field-of-view*" in the horizontal and vertical image plane directions. The horizontal Field-of-View angle (FOV) is the angle between the left and right clipping planes (see

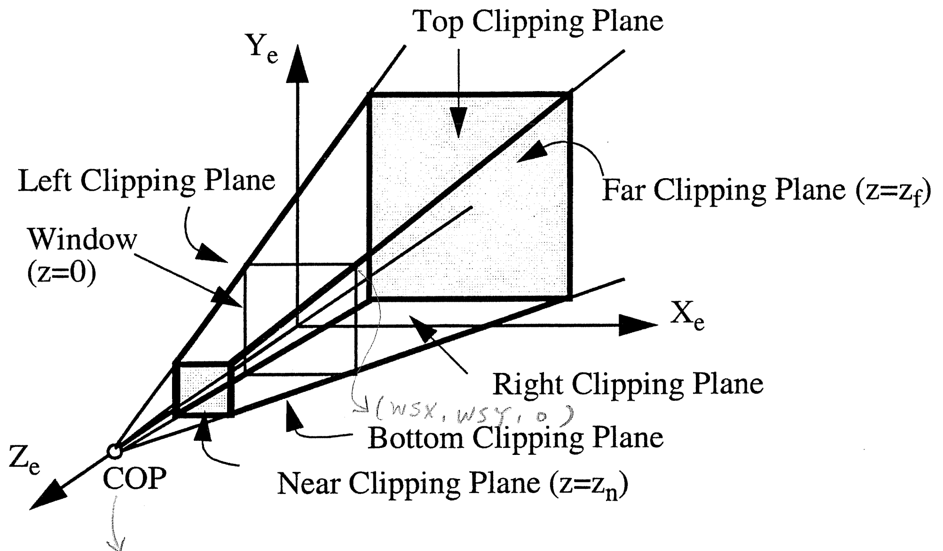


Figure 10-23. Viewing Frustum.

Figure 10-24), and the vertical FOV is the angle between the top and bottom clipping planes. The FOV angle is often used as a viewing parameter.

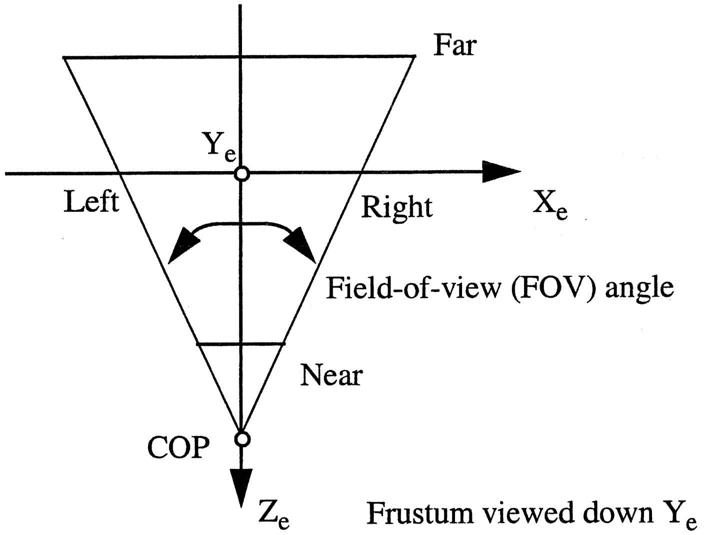


Figure 10-24. Horizontal Field-of-View Angle.

Examine the top and bottom clipping planes of the viewing frustum when viewed down the X_e axis as shown in Figure 10-25. The equations of the top and bottom planes are functions of z, the window size wsy, and the perspective distance d. These are infinite planes that divide 3D space in the Y direction into 3 regions: above the top clipping plane, visible, and below the bottom clipping plane.

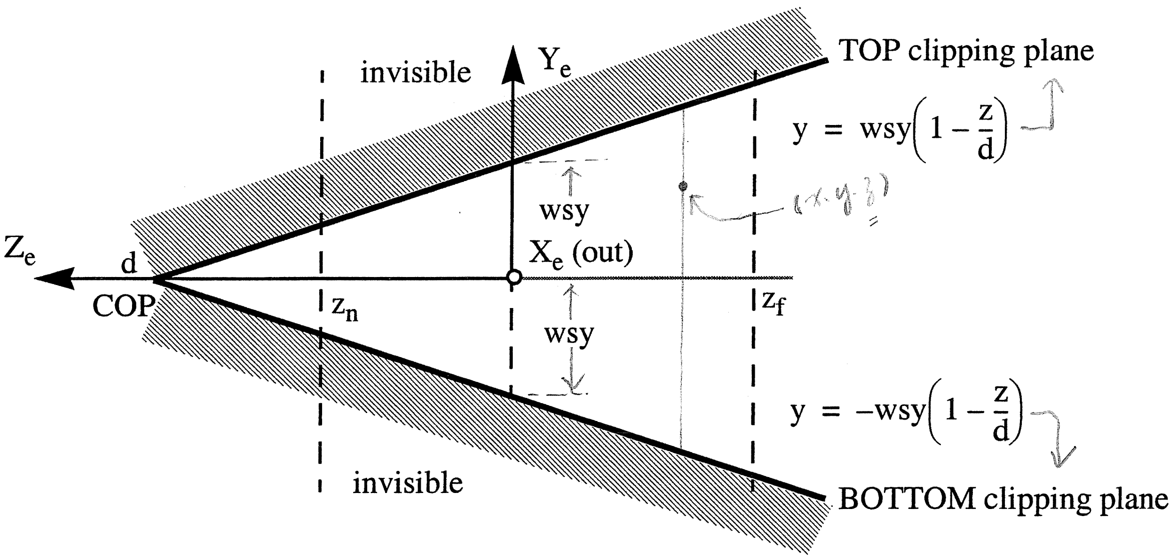


Figure 10-25. Side View (Looking Down X_e) of the Viewing Frustum.

Similarly, Figure 10-26 shows a top view of the viewing frustum and the equations for the left and right clipping planes. These are infinite planes that divide 3D space into three regions, left, visible, and right.

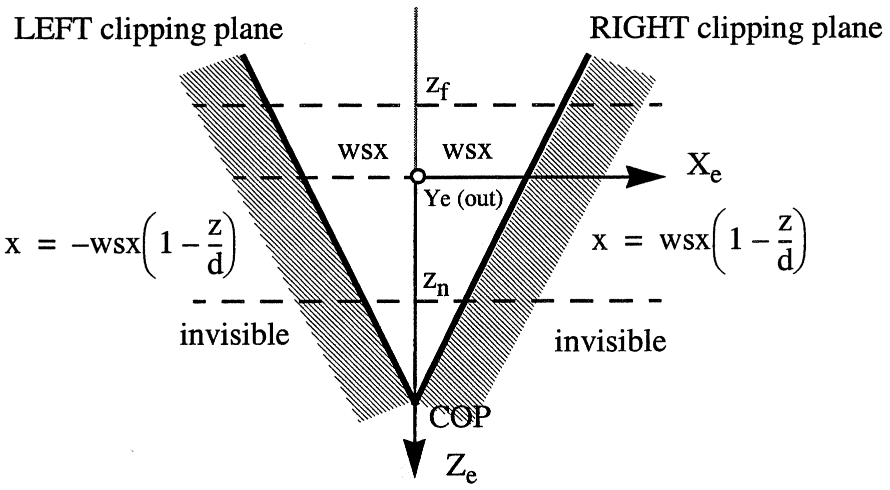


Figure 10-26. Top View (Looking Down Y_e) of the Viewing Frustum.

There is a common factor, $(1 - \frac{z}{d})$, that should look familiar. It is the inverse of the perspective factor. The 6 infinite planes form the boundaries of a "visible volume" that is described by the following equations.

Letting $wsx_p = wsx\left(1-\frac{z}{d}\right)$, $wsy_p = wsy\left(1-\frac{z}{d}\right)$:

$$\begin{array}{ccccc} -wsx_p & <= & x & <= & wsx_p \\ -wsy_p & <= & y & <= & wsy_p \\ z_f & <= & z & <= & z_n \end{array}$$

These equations can be coded into a *3D end point code* using a 6-bit binary code, where each bit represents the Boolean term "the point lies on the invisible side of the [TOP, BOTTOM, LEFT, RIGHT, NEAR, FAR] clipping plane." A zero code means the point is inside the frustum.

32	16	8	4	2	1
NEAR $z > z_n$	FAR $z < z_f$	TOP $y > wsy_p$	BOTTOM $y < -wsy_p$	LEFT $x < -wsx_p$	RIGHT $x > wsx_p$

Figure 10-27. Sample 3D End Point Region Code.

The frustum clipping algorithm can be implemented as an extension of the Cohen-Sutherland 2D clipping algorithm. The exit conditions are still based on the AND of the end point codes.

Clipping a line that crosses a boundary is a bit more complicated in 3D. Given two end points, P_1 and P_2, where P_1 is invisible and P_2 is visible, i.e., the line $P_1 P_2$ crosses a clipping plane, one must compute a new P_1 that lies on this plane. This was done in the 2D clipping algorithm using the symmetric equations of a line. Unfortunately, we now have a line in 3D space that intersects an oblique plane (not orthogonal to a principal axis).

One technique for this computation is to use the plane equation of the clipping plane, "$Ax + By + Cz + D = 0$". First note that this equation can be expressed as a *homogeneous vector dot product*:

$z - z_n = 0$

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = 0 \qquad \text{or,} \qquad P \bullet B = 0 \qquad \begin{bmatrix} 0 \\ 0 \\ 1 \\ -z_n \end{bmatrix} \leftarrow \text{near plane}$$

where B is the *plane coefficient vector* for a clipping plane. The equation of any point along the line $P_1 P_2$ can be expressed as a parametric vector function $P(\alpha)$, where the scalar parameter α

varies between 0.0 at P_1 to 1.0 at P_2:

$$P(\alpha) = P_1 + \alpha(\underset{\hookrightarrow \text{direction}}{P_2 - P_1})$$

First form the homogeneous dot product of the $P(\alpha)$ equation with the plane coefficient vector of the violated clipping plane, B. Then solve for the value of the parameter α for which the line intersects the plane by recognizing that $P(\alpha) \bullet B = 0$ at the intersection (i.e., when $P(\alpha)$ is on the plane).

$$P(\alpha) \bullet B = 0 = P_1 \bullet B + \alpha(P_2 - P_1) \bullet B$$

$$\alpha = \frac{P_1 \bullet B}{P_1 \bullet B - P_2 \bullet B}$$

The "*clipping plane coefficient vectors,* B," are easily formed from the equations of the clipping planes and are shown in Figure 10-28.

Plane Identifier	Plane Coefficient Vector	Algebraic Equation	End Point Test (Invisible is True)
LEFT	$[\,1, 0, -(\frac{wsx}{d})\,, wsx\,]$	$x = -wsx\left(1-\frac{z}{d}\right)$	$x < -wsx\left(1-\frac{z}{d}\right)$
RIGHT	$[\,-1, 0, -(\frac{wsx}{d})\,, wsx\,]$	$x = wsx\left(1-\frac{z}{d}\right)$	$x > wsx\left(1-\frac{z}{d}\right)$
BOTTOM	$[\,0, 1, -(\frac{wsy}{d})\,, wsy\,]$	$y = -wsy\left(1-\frac{z}{d}\right)$	$y < -wsy\left(1-\frac{z}{d}\right)$
TOP	$[\,0, -1, -(\frac{wsy}{d})\,, wsy\,]$	$y = wsy\left(1-\frac{z}{d}\right)$	$y > wsy\left(1-\frac{z}{d}\right)$
FAR	$[\,0, 0, 1, -z_f\,]$	$z = z_f$	$z < z_f$
NEAR	$[\,0, 0, -1, z_n\,]$	$z = z_n$	$z > z_n$

Figure 10-28. Frustum Clipping Plane Coefficient Vectors.

Using the fact that the plane equation, Ax+By+Cz+D=0, is the same as its negation, -Ax-By-Cz-D=0, the coefficient vectors can be formed so that the value of the dot product, $P \bullet B$, can be used to determine on which the side of the plane the point P lies. The plane coefficient vectors shown in Figure 10-28 have been formed such that $(P \bullet B)$ yields a *positive* number (or zero) if P is on the visible side of the plane and a negative value if the point is on the invisible side (and, of course, 0 if the point is on the plane). As a result, the end point code test also can be based on the

$P_1 \cdot B \quad P_2 \cdot B \to \alpha$

plane coefficient vector. This can further simplify and modularize the frustum clipping code. A single general plane clipping routine can be "passed" the plane coefficient vectors of the six clipping planes, classify the end points and clip to the boundaries using only the plane coefficient vector.

The pipeline for 3D frustum clipping is shown in Figure 10-29.

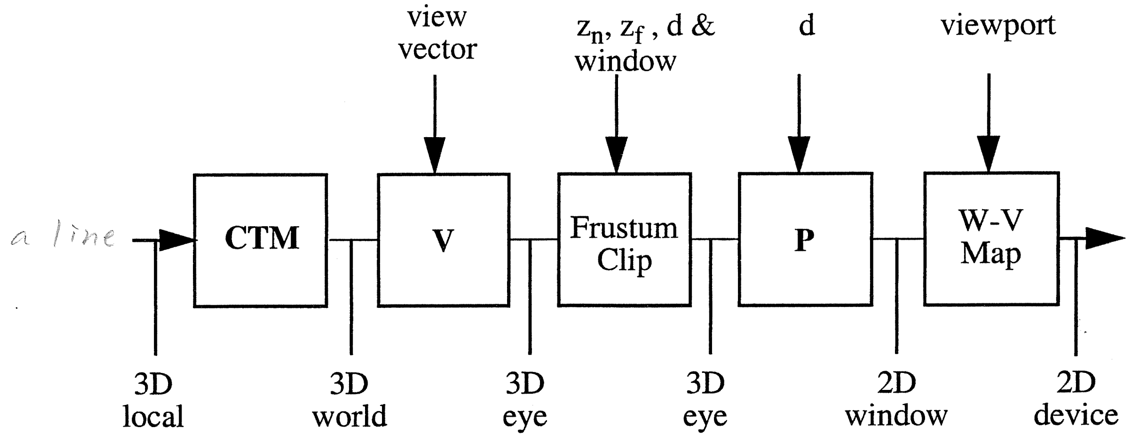


Figure 10-29. 3D Pipeline with Frustum Clipping.

10.8.3 3D Clipping in Homogeneous Coordinates

A disadvantage of 3D frustum clipping is that the plane coefficient vectors vary with the viewing frustum parameters (z_n, z_f, wsx,wsy,d) and must be re-computed whenever this viewing information changes. Also, note that except for clipping, the pipeline consists of transformations that could be concatenated into one matrix. The entire pipeline could become a vector to matrix multiplication followed by a simple normalization (perspective division) of a homogeneous vector. The problem, however, is clipping.

2D clipping is not the real problem, this can be done after projection on the viewport. The real problem is z-clipping on the near clipping plane that must be done before perspective to avoid inversion and division by zero.

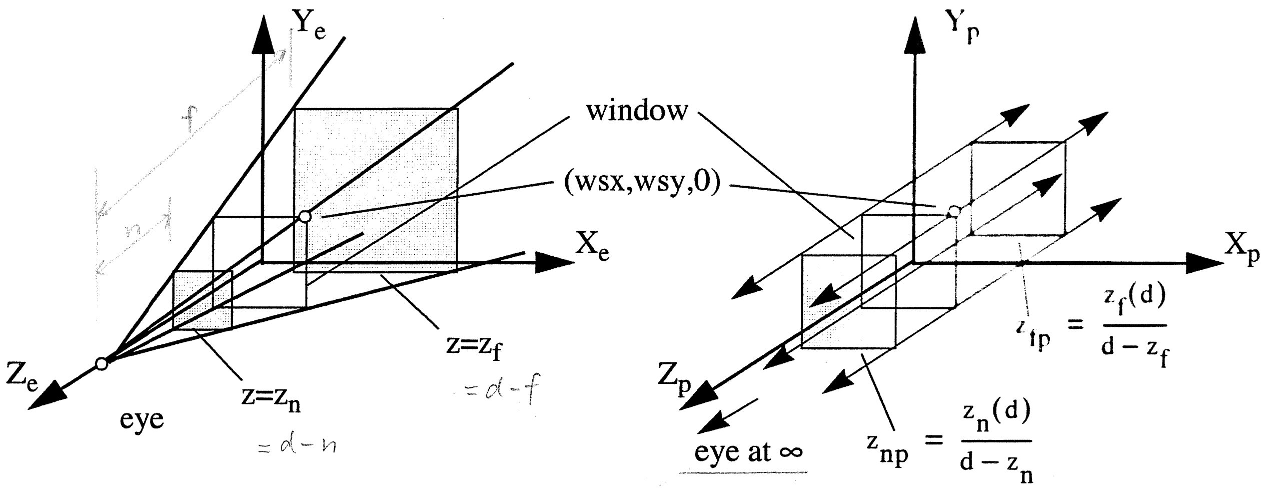
Looking more closely at this problem, perspective projection of a point occurs when the homogeneous vector is normalized to compute the 3D coordinates, i.e., when wx, wy and wz are divided by the w coordinate. Therefore, any 3D clipping computations done after transformation by the perspective matrix **P** must be performed prior to this, when the coordinates are in

↳ normalization

not w·x

homogeneous 4D space, i.e., [wx,wy,wz,w] with w not necessarily equal to one. This is not as complex as it first sounds, although it may appear to be more complex than the previous methods due to the nature of 4D coordinates.

Observe what happens to the viewing frustum after perspective projection (Figure 10-30).



viewing frustum before perspective

viewing frustum after perspective

Figure 10-30. Viewing Frustum Before and After Perspective.

The frustum is warped into a rectangular parallellipiped extending through the window between the projected near and far planes. Further examining the process, the two-dimensional image of the data is actually an orthographic parallel projection (front view, x and y values only) of the projected 3D coordinates.

Deeper insight into the perspective projection process is gained by examining three regions along the Z axis in the W-Z plane of 4D space before and after perspective, as shown in Figure 10-31.

Region	Before Perspective	After Perspective
1.	$d < z < \infty$	$-\infty < z < -d$
2.	$0 < z < d$	$0 < z < +\infty$
3.	$-\infty < z < 0$	$-d < z < 0$

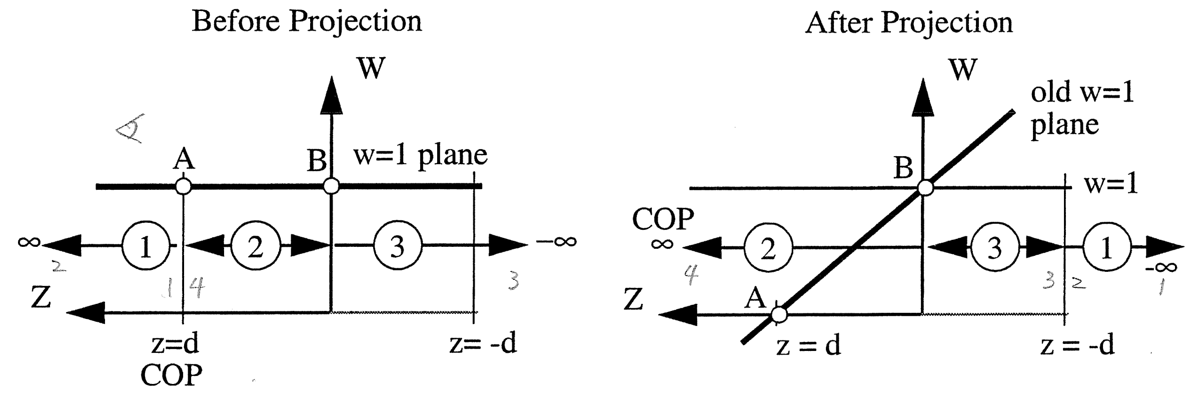


Figure 10-31. W-Z Plane View of Perspective Projection.

Several of the z projections involve limits. In region 1, as z approaches d,

$$\text{1.}\quad \lim_{\substack{z \to d \\ z > d}} \left(\frac{zd}{d - z} \right) = -\infty$$

Similarly, as z approaches ∞ and -∞ in regions 1 and 3,

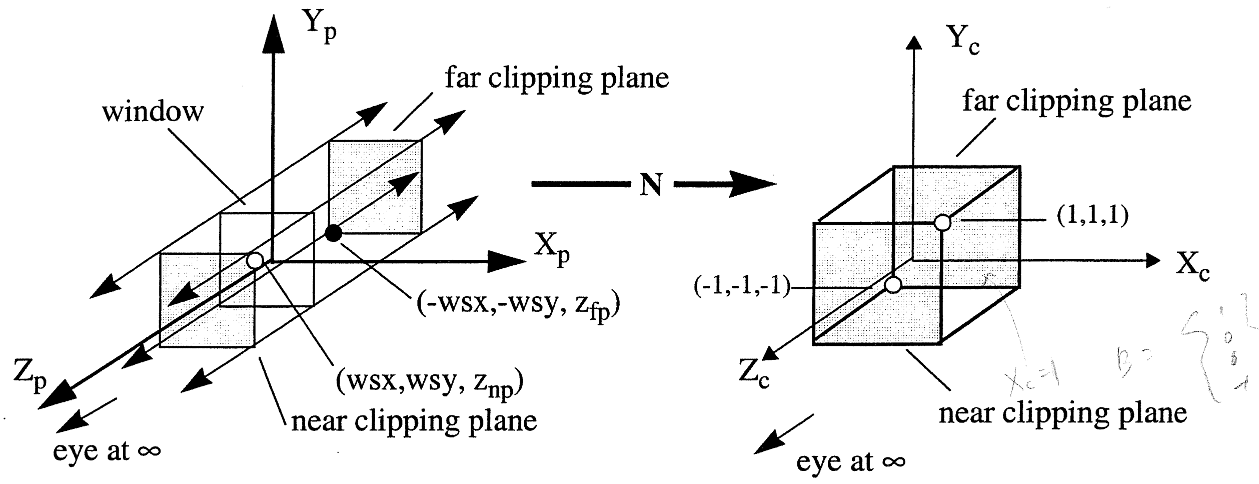
$$\text{2.}\quad \lim_{z \to \infty} \left(\frac{zd}{d - z} \right) = -d \qquad \text{3.}\quad \lim_{z \to -\infty} \left(\frac{zd}{d - z} \right) = -d$$

Finally, as z approaches d in region 2,

$$\text{4.}\quad \lim_{\substack{z \to d \\ z < d}} \left(\frac{zd}{d - z} \right) = \infty$$

In region 1, points behind the COP prior to projection transform into points in front of the eye after projection. Points in region 2 between the COP and the image plane are projected into greatly expanded semi-infinite region. Consider what happens to a point that is placed very near the COP. It will project to a point near the eye, i.e., near ∞. Likewise, points in the semi-infinite region 3 are compressed into a region between 0 and -d. This shows the highly non-linear nature of perspective projection.

Homogeneous clipping can be simplified considerably by transforming the coordinates after applying perspective using "a *normalization transform*, **N**," that transforms a point into "*clipping coordinates*." The x, y, and z coordinates of a point transformed by **P** are mapped into a *clipping coordinate system*, XYZ_C, in which the visible coordinates, coordinates that were originally inside the viewing frustum in XYZ$_e$, lie inside a cube centered at the origin ($x,y,z \in [-1,1]$). In effect, the view volume is transformed into a cube, as shown in Figure 10-32. After clipping lines (in 4D) against the clipping cube, they can be projected and mapped directly to the viewport.



viewing frustum after perspective

the clipping cube

Figure 10-32. Homogeneous Normalization After Perspective.

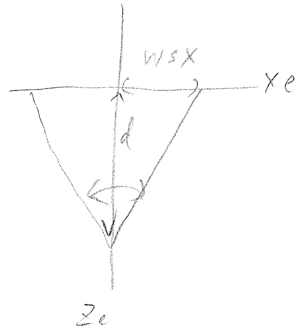
The normalization transformation **N** is constructed in two steps: (1) translate the center of the projected view volume to the origin, and (2) scale the resulting volume into a cube with x, y and z in [-1,1]. Refering to Figure 10-32, the center of the projected frustum lies along the z axis half-way between the near and far clipping planes. Let z_{np} be the z coordinate of the near clipping plane after projection, and z_{fp} be the z coordinate of the far clipping plane after projection. Using the standard perspective equations:

$$z_{np} = z_n \frac{d}{d - z_n} \quad \text{and} \quad z_{fp} = z_f \frac{d}{d - z_f}$$

The half-sizes of the projected volume are wsx, wsy, and $(z_{np}-z_{fp})/2$. We can now build **N**:

$$\mathbf{N} = \mathbf{T}\left(0, 0, -\frac{(z_{np} + z_{fp})}{2}\right)\mathbf{S}\left(\frac{1}{wsx}, \frac{1}{wsy}, \frac{2}{(z_{np} - z_{fp})}\right) \qquad \frac{1}{\frac{1}{2}(Z_{np}-Z_{fp})}$$

We concatenate the **P** and **N** matrices and multiply by d to simplify the terms. Notice that multiplying the matrix by d results in multiplying the resulting coordinates by d. However, scaling a homogeneous coordinate does not affect the normalized value. The resulting matrix is shown below. 'n' is the near distance, and 'f' is the far distance.



$$\mathbf{dPN} = \begin{bmatrix} \frac{d}{wsx} & 0 & 0 & 0 \\ 0 & \frac{d}{wsy} & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -1 \\ 0 & 0 & \frac{df+dn-2fn}{n-f} & d \end{bmatrix}$$

$\frac{d}{wsx} = \frac{d}{wsy}$

for "pz"

n = 0.1

f = 150

Consider the scaling elements $\frac{d}{wsx}$ and $\frac{d}{wsy}$ above. Referring to Figure 10-24, we see that

$$\frac{d}{wsx} = \cot\left(\frac{FOVx}{2}\right) \quad \text{and} \quad \frac{d}{wsy} = \cot\left(\frac{FOVy}{2}\right)$$

After the perspective and normalization transformations, the resulting homogeneous coordinates must be properly clipped against the clipping cube. Each endpoint is a homogeneous coordinate [wx, wy, wz, w] for which the visible 3D (projected) coordinates lie between -1.0 and 1.0.

(wx, wy, wz, w)

The 3D clipping volume is described by the equations:

→ $$-1 \quad <= \quad \frac{wx}{w}, \frac{wy}{w}, \frac{wz}{w} \quad <= \quad 1$$

W = 0 $B = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{Bmatrix}$

Recall, however, that clipping must occur prior to division. Therefore, clipping must be performed in the 4-dimensional homogeneous space. The above equations in 4D space are:

→ $$-w \quad <= \quad wx, wy, wz \quad <= \quad w$$

This involves clipping on the six planes wx = -w, wx = +w, and so on for wy and wz.

There is one special action necessary for lines with one end point that has a negative w value. Note that if both end points have negative w values, the line is wholly rejected. Recall that a negative w value results from perspective transformation of a point with a z coordinate behind the COP, i.e. $w = 1 - \frac{z}{d}$ is negative when z > d. This line must be clipped against the w=0 plane, removing the portion of the line behind the COP. Clipping on the w=0 plane must be done first, before clipping on the other planes. This is a simple matter of properly ordering the plane tests in the clipping code. Homogeneous clipping involves seven *hyperplanes* in 4D space.

Once again, the clipping algorithm is an extension of the basic Cohen-Sutherland 2D clipping logic, now clipping on a volume bounded by seven planes. A seven bit clipping code can be used, where the endpoint codes identify mutually exclusive regions bounded by the infinite planes called TOP, BOTTOM, LEFT, RIGHT, NEAR, FAR, WZERO (for the w=0 hyperplane). After checking the Boolean AND (quick reject) and OR (accept) of the end point codes, the line must be clipped by recomputing one of its end points.

Computing the intersection of the line with a clipping plane follows the same logic as described previously for frustum clipping. The plane equations, however, are much simpler and contain only constants (by design). Be sure to use homogeneous coordinates now.

$$\begin{bmatrix} wx & wy & wz & w \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = 0 \qquad \text{or,} \qquad P \bullet B_i = 0$$

The clipping plane coefficient vectors for the homogenous clipping coordinates, formed as before so that P • B <0 for invisible points, are given in Figure 10-33. Note that they are simple constants.

Plane Identifier	Plane Coefficient Vector	Algebraic Equation	End Point Test
WZERO	[0, 0, 0, 1]	w = 0	w < 0
LEFT	[1, 0, 0, 1]	wx = - w	wx < - w
RIGHT	[-1, 0, 0, 1]	wx = w	wx > w
BOTTOM	[0, 1, 0, 1]	wy = - w	wy < - w
TOP	[0, -1, 0, 1]	wy = w	wy > w
FAR	[0, 0, 1, 1]	wz = - w	wz < - w
NEAR	[0, 0, -1, 1]	wz = w	wz > w

Figure 10-33. Homogeneous Clipping Plane Data.

The pipeline for 3D homogeneous clipping consists of two matrix multiplications and the homogeneous clipping procedure: The homogenous line clipping algorithm is outlined in Figure 10-35.

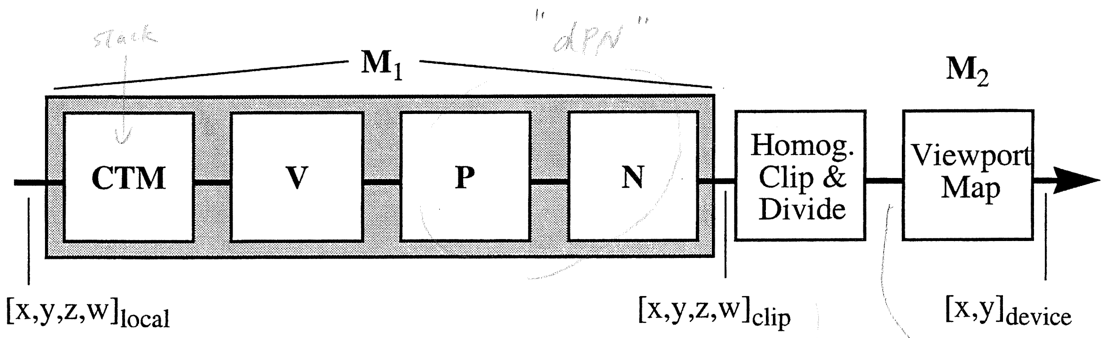


Figure 10-34. 3D Pipeline with Homogeneous Clipping. (wx, wy, wz, w)

(1, 1)
(-1,-1)

```
ClipLine( Point4D p1, Point4D p2 )
c2 = Zone( p2 )
LOOP:
      c1 = Zone( p1 )
      IF c1 is IN and c2 is IN THEN
            ProjectAndDraw( p1, p2 )
            RETURN.
      END
      IF (c1 AND c2) ¦ 0 THEN
            RETURN. "line is wholly invisible"
      "special case: p2 outside WZERO and p1 out: do WZERO first:"
      IF (c2 AND WZERO) OR (c1 is IN) THEN
            SWAP c1<-> c2 and p1<->p2

      IF c1 AND WZERO THEN
            B = wzeroplane
      ELSE IF c1 AND TOP THEN
            B = topplane
      etc... for the other planes in any order
```

$$\alpha = \frac{p1 \bullet B}{p1 \bullet B - p2 \bullet B}$$

$$p1 = p1 + \alpha(p2 - p1)$$

p2

```
ENDLOOP
```

Figure 10-35. Outline of Homogeneous Line Clipping.

See Blinn [1] for another approach to homogeneous clipping.

10.9 The GL Pipeline

One of the most popular 3D graphics packages is "GL" that was developed by Silicon Graphics, Inc. for its high performance workstations. The GL pipeline transformations use a somewhat different eye coordinate system and clipping coordinate system than given previously. The following sections derive the projection transformation for the GL perspective projection. This offers another approach to the 3D viewing and projection process that will strengthen our understanding of 3D viewing, perspective and clipping.

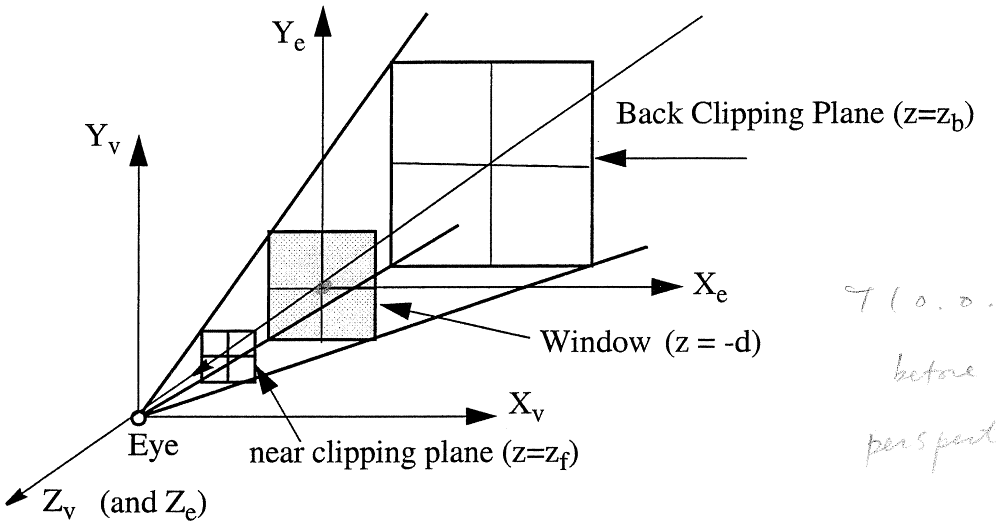
GL objects are defined in a right-handed coordinate system, but clipping coordinates are expressed in a left-handed clipping coordinate system. Left-handed systems have been used in many computer graphics systems and are sometimes regarded as a more natural reference for certain graphics computations because z values increase in the viewing direction, away from the eye. Recall that in a right-handed eye coordinate system, z values decrease (become more negative) in the viewing direction. For example, during the previous discussion of 3D clipping, the visibility tests for z coordinates may have seemed confusing because "farther away" means "smaller z." Another example is hidden surface processing, where z tests are performed on objects and left-handed coordinates may seem more natural.

10.9.1 GL Viewing Coordinate System

The GL viewing transformation maps data in world coordinates into an intermediate "viewing coordinate system" where the eye is at the origin and the image plane is at z = -d, as shown in Figure 10-38. This special coordinate system will be called XYZv. The locations of the near and back clipping planes are given as distances from the eye, *near* and *far*, to avoid the awkwardness to the user of working in intermediate coordinates, such as eye coordinates and clipping coordinates, and to make the plane locations relative to the eye position. Given the eye to clipping plane distances 'n' (eye to near) and 'f' (eye to far), the XYZv near and far clipping plane locations are:

$z_n = -n$ and $z_f = -f$.

Although this appears to be a suitable eye coordinate system for projection, there is a problem with performing perspective projection in XYZv, or any coordinate system where the



T(0.0.d)
before
perspective transf.

Figure 10-36. GL Viewing Frustrum.

COP is at the origin. Recall that the perspective factor for such a system is $\left(\frac{-d}{Z}\right)$, and the projection matrix is shown below (note the last row is all 0's):

$$\mathbf{P}_v = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{d} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The general projection is:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{d} \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} x & y & z & \frac{-z}{d} \end{bmatrix}$$

After perspective division:

$$\begin{bmatrix} x\left(\frac{-d}{z}\right), & y\left(\frac{-d}{z}\right), & -d, & 1 \end{bmatrix}$$

Notice that the z coordinate of all projected points is -d. We will see later that this projection is inadequate for hidden surface removal applications. The problem is that under projection, the COP must map to infinity, but this is not possible when the COP is the origin.

Therefore, we must transform XYZv into XYZe, with the COP at [0,0,d] and the image plane at the origin, and then perform the normal perspective transform **P**. Combining these two transforms into one results in a projection from XYZv coordinates, called $\mathbf{P_{tv}}$.

$$\mathbf{P_{tv}} = \mathbf{T(0,0,d)\ P}$$

$$\mathbf{P}_{tv} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{d} \\ 0 & 0 & d & 0 \end{bmatrix}$$

For clipping normalization, we must keep track of the projected coordinates of z_n and z_f. Transformation by $\mathbf{P_{tv}}$ yields a normalized z coordinate, z_p of:

$$z_p = \frac{-d(z+d)}{z}$$

Therefore, substituting -n for z_n and -f for z_f,

$$z_{np} = \frac{d(d-n)}{n} \text{ and } z_{fp} = \frac{d(d-f)}{f} .$$

As before, **N** will map the viewing frustrum into the clipping cube, however this time the z coordinate must be negated to convert from a right-handed to a left-handed system (Figure 10-37).

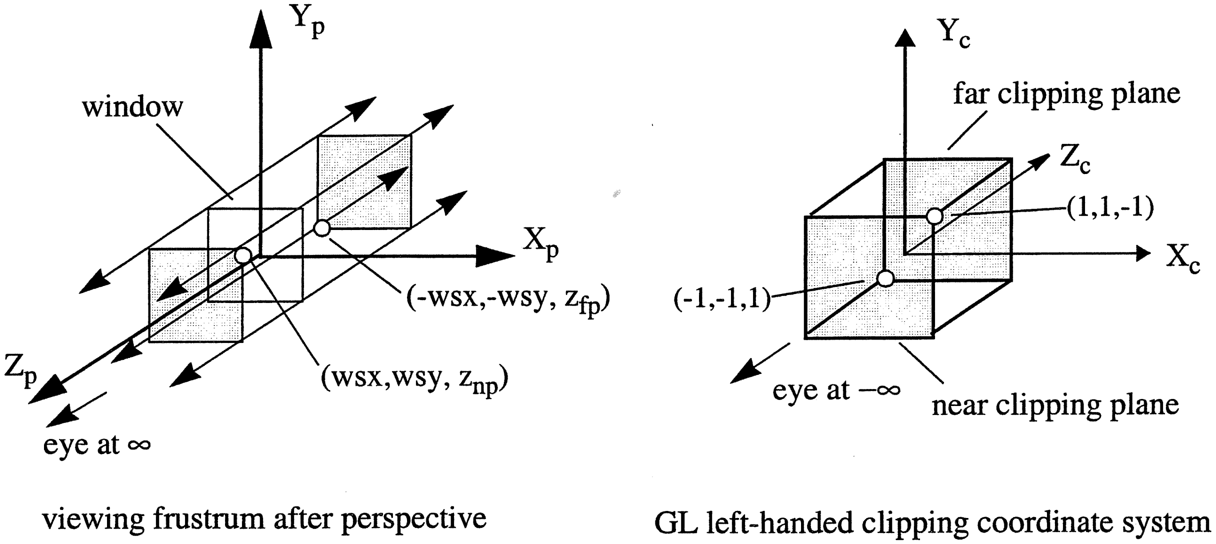


Figure 10-37. GL Homogeneous Normalization After Perspective.

The normalization transforms for the left handed clipping coordinate system are:

$$\mathbf{N}' = \mathbf{T}\left(0, 0, -\frac{(z_{np} + z_{fp})}{2}\right)\mathbf{S}\left(\frac{1}{wsx}, \frac{1}{wsy}, \frac{-2}{(z_{np} - z_{fp})}\right)$$

The advantage of the GL eye coordinate system is seen when we combine $\mathbf{P}_v$ and $\mathbf{N}'$ into a single transformation, **PvN**′, that transforms data from XYZv into XYZc in one step. For simplification, we multiply the matrix by the perspective distance d. This scales the resulting homogeneous coordinates by d but has not effect on the resulting 3D coordinates because of the division by w.

$$dP_{tv}N' = \begin{bmatrix} \frac{d}{wsx} & 0 & 0 & 0 \\ 0 & \frac{d}{wsy} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{(f-n)} & -1 \\ 0 & 0 & \frac{-2fn}{(f-n)} & 0 \end{bmatrix}$$

$1/\tan\left(\frac{FOV}{2}\right)$

This is a simple transformation that can be easily constructed directly from the pipeline parameters.

As checks, transform some "known" points:

[0,0,-n,1] -> [0,0,-1] (near plane center),

[0,0,-f,1] -> [0,0,+1] (far plane center),

[wsx,wsy,-d,1] -> [1,1,--] (upper right corner of window),

[-wsx,-wsy,-d,1] -> [-1,-1,--] (lower left corner of window).

10.9.2 Field-of-View

Note that there are two interesting ratios in the transformation above: d/wsx and d/wsy. We see that

$$\frac{wsx}{d} = \tan\left(\frac{fovx}{2}\right) \qquad \frac{wsy}{d} = \tan\left(\frac{fovy}{2}\right)$$

where "fovx" and "fovy" are the field of view angles that relate the perspective distance and the window sizes. The field of view angles are alternative parameters for specifying the perspective distance and window sizes. For a non-square window, there will be two fields of view, fovx and fovy. GL uses fovy and an aspect ratio, so fovx = fovy * aspect.

Replacing the d, wsx and wsy terms with fovy and aspect:

$$\mathbf{dPtvN'} = \begin{bmatrix} \dfrac{1}{\text{aspect}\tan\left(\dfrac{\text{fovy}}{2}\right)} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\left(\dfrac{\text{fovy}}{2}\right)} & 0 & 0 \\ 0 & 0 & \dfrac{-(f+n)}{(f-n)} & (-1) \\ 0 & 0 & \dfrac{-2fn}{(f-n)} & 0 \end{bmatrix}$$

The clipping plane coefficient vectors are nearly identical to the right-handed versions, only the near and back planes have changed.

Plane Identifier	Plane Coefficient Vector	Algebraic Equation	End Point Test
WZERO	[0, 0, 0, 1]	w = 0	w < 0
LEFT	[1, 0, 0, 1]	wx = - w	wx < - w
RIGHT	[-1, 0, 0, 1]	wx = w	wx > w
BOTTOM	[0, 1, 0, 1]	wy = - w	wy < - w
TOP	[0, -1, 0, 1]	wy = w	wy > w
NEAR	[0, 0, 1, 1]	wz = - w	wz < - w
FAR	[0, 0, -1, 1]	wz = w	wz > w

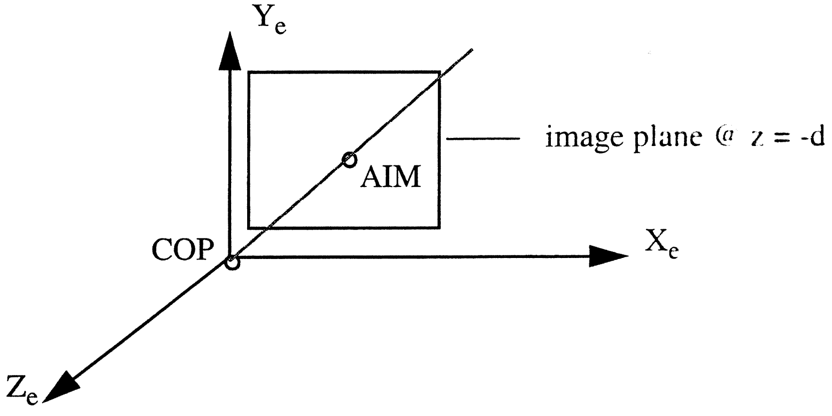
Figure 10-38. Homogeneous Clipping Planes for Left-Hand XYZc.

10.10 References

1. James F. Blinn, "A Trip Down the Graphics Pipeline: Line Clipping," IEEE Computer Graphics & Applications, January 1991, pp. 98-105.

10.11 Review Questions

1. A special eye coordinate system places the COP at the origin and image plane at z= -d in a right-handed eye coordinate system, shown below. Derive the perspective projection equations for this eye coordinate system. Show the **P** matrix.



2. Derive the perspective projection equations for x, y and z for a left-handed eye coordinate system with the center of projection at the origin and image plane at z = +d. Derive the corresponding **P** matrix.
3. The computation of the eye position given an AIM position using polar coordinates was given earlier as

   EYE = [0,0,d] **R**(-elevation, X) **R**(azimuth, Y) $\oplus$ AIM.

   Derive the viewing transform directly from the elevation and azimuth angles, without first computing EYE.
4. Why is it necessary to do near plane z clipping? Where in the 3D pipeline should it be done? Why is it desirable to do BACK plane z clipping? Where in the 3D pipeline should it be done?
5. The homogeneous clipping coordinates of the endpoint of a line after transformation by **V**, **P** and **N** are [1,3,-3,2]. Identify and give the plane equation of each clipping plane that is violated by this point.

6. Develop the perspective **P** and normalization **N** matrices for transforming data from a right-handed eye coordinate system, with the image plane at z=0, center of projection at (0,0,d), square window half-size of S, and near and back clipping plane locations ZF and ZB, into left-handed homogeneous clipping coordinates in which the visible x, y and z coordinates lie between 0.0 and 1.0 and z increases away from the eye. Draw and label an illustration of the affine viewing frustrum before and after the **P** and **N** transformations. Show the **N** transformation as a concatenation of basic transforms given in functional form with symbolic expressions as arguments.

Chapter 11. Computational Geometry

It should be clear by now that geometry is at the root of virtually all computer graphics problems. *Geometric modeling*, the representation of physical objects in the computer, is also at the heart of two areas receiving considerable attention today: computer-aided design (CAD) and computer-aided manufacturing (CAM). CAD, CAM and computer graphics share a mutual interest in geometry.

The term *computational geometry* describes an area of study addressing mathematical algorithms and computational methods for representing and manipulating geometry in the computer. We begin with a review of vector geometry fundamentals to establish terminology and basic methods on which later sections will build.

11.1 Review of Vector Geometry Fundamentals

There are significant advantages to using vector representation and vector equations to solve geometry problems instead of algebraic equations operating separately on each x, y and z coordinate:

1. more general solutions,
2. 2D and 3D solution methods are the same,
3. avoids many special cases,
4. clearer geometric interpretation,
5. code more closely resembles the mathematical form.

In Figure 11-1, the vector V is a *direction vector* that represents direction and length. It is also called a *relative vector*. The coordinates of a direction vector are relative displacements. The vector P in Figure 11-1 is a *position vector*, a vector that begins at the origin O. A position vector represents a location in space with respect to some reference frame origin. The origin is a position vector. The distinction between direction and position vectors is sometimes subtle and often overlooked. It will be a useful distinction later when computational algorithms are discussed.

We will represent all vectors in homogeneous coordinates:

V = [wx, wy, wz, w],

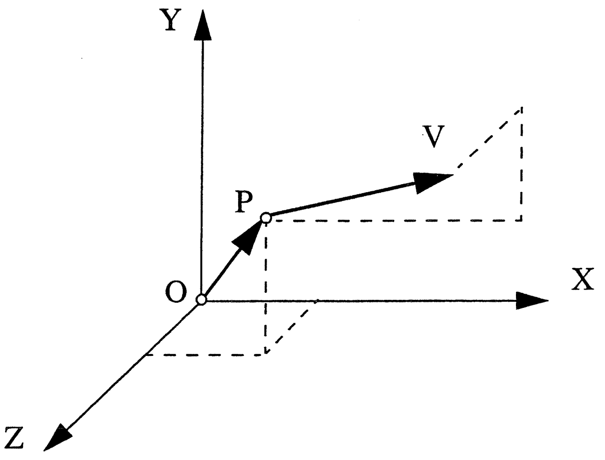


Figure 11-1. Position Vector P and Direction Vector V.

or sometimes unnormalized homogeneous coordinates are shown in capital letters:

V = [X, Y, Z, W].

Recall that the homogeneous vector [x, y, z, 0] was reasoned to be a point at infinity on a line through the origin and the point. Such a vector also results from the vector subtraction of two position vectors with identical w coordinates:

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ w \end{bmatrix} - \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ w \end{bmatrix} = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \\ 0 \end{bmatrix}$$

Therefore, a direction vector has a zero w coordinate. As expected, position vectors have non-zero w coordinates. This will be particularly useful when we consider transformations, such as translation, applied to direction and position vectors.

Geometrically, one can think of the vector coordinates as magnitudes to be applied to unit vectors along the principal axes in the vector equation (the dashed lines in Figure 11-1):

V = x i + y j + z k

where i, j, k are unit vectors, i.e. their length is one.

11.2 Basic Vector Utilities

In C and C++, one can define a type called `vector` that is an array,

```
typedef double vector[4];
vector v;
v[0] = 1.0;     /* or v[X] = 1.0; */
```

or a structure:

```
typedef struct {
     float (or double) x,y,z,w;
} vector;
vector v;
v.x = 1.0;
```

The structure `vector` form allows assignment of vectors, pass-by-value function arguments and vector return values from functions.

The following sections develop a basic set of vector utilities. The use of the `vector` structure is exploited in the example functions. The functions pass vectors by value, which encourages clear and correct coding, but is inefficient compared to passing vectors by address.

11.2.1 Vector Magnitude

The magnitude or length of a vector is shown as |V| and computed as:

$$|V| = \sqrt{x^2 + y^2 + z^2}$$

```
double VMag( vector v )
{
     return sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
}
```

Figure 11-2. Vector Magnitude Function.

Note that the magnitude funciton should be applied only to the direction vectors (w=0). The w coordinate is ignored. In our applications, the magnitude is meaningful only in 3D coordinates.

11.2.2 Vector Addition and Subtraction

Vector addition is shown as:

$V_1 \oplus V_2 =$ $[\, x_1 + x_2\, , y_1 + y_2, z_1 + z_{2,} w_{1\,+}\, w_2\,]$, and similarly for subtraction.

```
vector VAdd(vector v1, vector v2)
{
        v1.x += v2.x; /* remember that v1 is passed by value */
        v1.y += v2.y;
        v1.z += v2.z;
        v1.w += v2.w;
        return v1;
}
vector VSub(vector v1, vector v2)
{
        v1.x -= v2.x;
        v1.y -= v2.y;
        v1.z -= v2.z;
        v1.w -= v2.w;
        return v1;
}
```

Figure 11-3. Vector Addition and Subtraction Functions.

11.2.3 Vector Multiplication and Division by a Scalar

A vector is scaled by the factor f as : f V = [f x, f y, f z , f w].

```
vector VScale( vector v, double f )
{
        v.x *= f;
        v.y *= f;
        v.z *= f;
        v.w *= f;
        return v;
}
```

Figure 11-4. Vector Scaling Function.

11.2.4 Unit Vector

To make a unit vector, a vector of length one, of a vector:

$$\|V\| = \frac{V}{|V|}$$

```
vector VUnit( vector v )
{
        v = VScale( v, 1.0 / VMag(v) );
        v.w = 0.0;          /* make a direction vector */
        return  v;
}
```

Figure 11-5. Unit Vector Function.

Here one sees the advantages of vector-based programming: compactness and the code "resembles" the mathematical expression. The unit vector operation should be applied only to 3D vectors, and sets the result to a direction vector (w = 0). Its application is to create unit magnitude vectors of direction vectors.

11.2.5 Dot Product:

The *dot product* , also called the *scalar product*, is computed as:

$V_1 \bullet V_2 \equiv x_1\, x_2 + y_1\, y_2 + z_1\, z_2 + w_1\, w_2$.

```
double VDot( vector v1, vector v2 )
{
        return v1.x*v2.x + v1.y*v2.y + v1.z*v2.z + v1.w*v2.w;
}
```

Figure 11-6. Dot Product Function.

For direction vectors, the value of the dot product can be interpreted as (Figure 11-7):

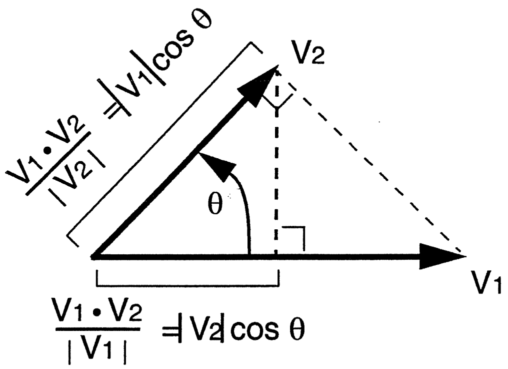
$V_1 \bullet V_2 = | V_1 | | V_2 | \cos \theta$,

and can be used to obtain the cosine of the angle between vectors:

$$\cos(\theta) = \frac{(x_1 x_2 + y_1 y_2 + z_1 z_2)}{|V_1||V_2|}$$

The dot product is commutative and distributive:

$V_1 \bullet V_2 = V_2 \bullet V_1$



$\frac{V_1 \bullet V_2}{|V_2|} = |V_1| \cos\theta$
V2
θ
V1
$\frac{V_1 \bullet V_2}{|V_1|} = |V_2| \cos\theta$

Figure 11-7. Geometric Interpretation of the Dot Product.

$$V_1 \bullet (V_2 \oplus V_3) = V_1 \bullet V_2 + V_1 \bullet V_3$$

The dot product also can be interpreted geometrically as the projection of one vector on another, as illustrated in Figure 11-7. This makes the dot product useful for some "geometric reasoning" computations. For example, if the dot product of two vectors is negative then cos(θ) is negative because vector magnitudes are always positive, which means that θ must be between 90 and 270 degrees, so we can say that "the vectors are facing in opposite directions." Similarly, if the dot product is zero, then cos(θ) is zero, so θ = 90 or 270 degrees, so "the vectors are perpendicular."

Note that $V \bullet V = | V | | V | \cos(0) = | V |^2$.

11.2.6 Cross Product

The *cross product*, also called the *vector product*, is computed as a vector:

$$V_1 \otimes V_2 = \left\| \begin{bmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{bmatrix} \right\| \equiv \begin{bmatrix} (y_1 z_2 - y_2 z_1) \\ -(x_1 z_2 - x_2 z_1) \\ (x_1 y_2 - x_2 y_1) \end{bmatrix}$$

The direction is determined by the "right hand rule," i.e., close your right hand in the direction of the first to second vector and your thumb points in the direction of the cross product.

The magnitude of the vector product is interpreted as:

$$| V_1 \otimes V_2 | = | V_1 | | V_2 | \sin\theta.$$

The cross product function should only be applied to direction vectors. The w coordinates are ignored.

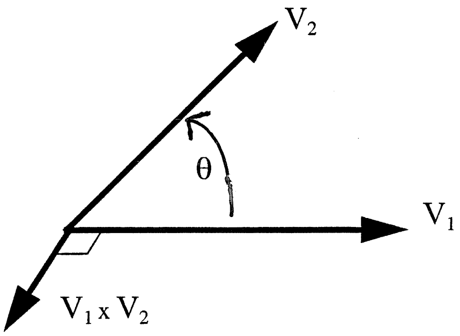


Figure 11-8. Direction of Cross Product Vector.

```
vector VCross(vector v1, vector v2)
{
      vector vout;
      vout.x = v1.y * v2.z - v2.y * v1.z;
      vout.y = v2.x * v1.z - v1.x * v2.z;
      vout.z = v1.x * v2.y - v2.x * v1.y;
      vout.w = 0.0;
      return vout;
}
```

Figure 11-9. Cross Product Function.

The cross product is not commutative, $V_1 \otimes V_2 = - V_2 \otimes V_1$ (opposite directions), but is distributive,

$$V_1 \otimes (V_2 \oplus V_3) = V_1 \otimes V_2 \oplus V_1 \otimes V_3 .$$

The cross product is also useful for geometric reasoning. For example, if the cross product of two vectors is zero, then, assuming the vectors magnitudes are non-zero, sin(θ) is zero, meaning θ=0 or 180 degress, so we can reason that the vectors are parallel and facing the same or opposite direction.

11.2.7 Vector Area

The area function is defined as the magnitude of the cross product of two vectors:

$$\mathrm{Area}(V_1 ,V_2) \equiv | V_1 \otimes V_2 |.$$

This is best explained geometrically. Consider two direction vectors placed at a common point, as in Figure 11-10 We extend the vectors to form a parallelogram with V_1 as the base. Define “height” as the projector from V_2 perpendicular to the base. From the definition of the cross product magnitude,

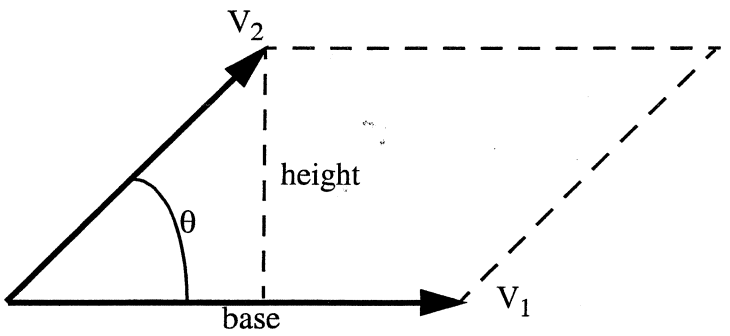


Figure 11-10. Geometric Interpretation of the Area Function.

$$| V_1 \otimes V_2 | = | V_1 | | V_2 | \sin(\theta),$$

but $| V_2 | \sin(\theta)$ is the value of height, and $| V_1 |$ is the value of base. Therefore, the magnitude of the cross product is the area of the parallelogram, and hence the reason for the name "Area."

What is the geometric significance when the Area(V_1,V_2) = 0? By definition, the area of the enclosing parallelogram is zero, hence the vectors must be colinear (pointing in the same or opposite directions).

```
double VArea( vector v1, vector v2 )
{
        return VMag( VCross( v1, v2 ) );
}
```

Figure 11-11. The Vector Area Function.

11.2.8 Vector Volume

The *vector volume* function is another vector computation that has a useful geometric interpretation. Given three direction vectors, the definition of the vector volume function is the dot product of the cross product:

$$\mathrm{Vol}(V_1 ,V_2 , V_3) \equiv (V_1 \otimes V_2) \bullet V_3.$$

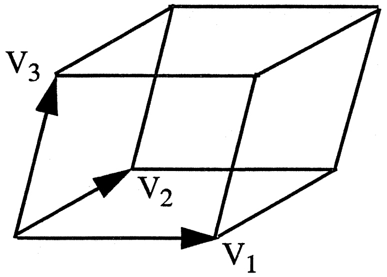


Figure 11-12. Vector Volume Function.

This is also known as the *scalar triple product.* The result, a scalar, is the volume of the parallelepiped formed by the extensions of the vectors.

If Vol(V_1 ,V_2 , V_3) = 0, then either two of vectors are parallel or the three vectors lie in a plane, i.e., contain no volume.

```
        double Vol( vector v1, vector v2, vector v3 )
{
        return VDot( VCross( v1, v2 ), v3 );
}
```

Figure 11-13. Vector Volume Function.

11.2.9 Vector Equation of a Line

We form the vector equation of a point set that describes a straight line through point P in the direction V:

$P(t) = P \oplus t\,V$,

where “t” is a parameter (Figure 11-14).

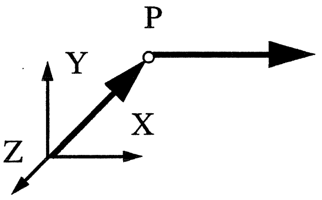


Figure 11-14. Vector Line Notation.

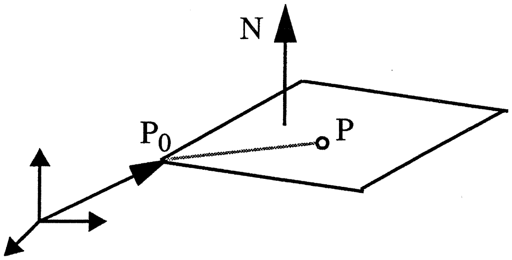
Depending on the range of the parameter t used in the equation, we can classify P(t) as a:

line	if t can have any value, $-\infty <= t <= \infty$,
ray	if t is limited to t >= 0,
segment	if t is limited to $t_0 <= t <= t_1$ (usually 0 <= t <= 1).

11.2.10 Vector Equation of a Plane

We form the vector equation of a point set that describes a plane through point P_0 with normal direction N by observing that any point P in the plane forms a line in the plane with P_0 that is perpendicular to the normal. The point-normal vector equation of the plane is given as:

$$(P - P_0) \bullet N = 0.$$



(P_0, N)

Figure 11-15. Vector Plane Notation.

Letting N = (A, B, C), P = (x, y, z) and $P_0 = (x_0, y_0, z_0)$, look closer at this equation. First expand the dot product,

$$(P \bullet N) - (P_0 \bullet N) = 0.$$

Now expand the vector equation into algebraic form,

$$(A\,x + B\,y + C\,z) - (A\,x_0 + B\,y_0 + C\,z_0) = 0.$$

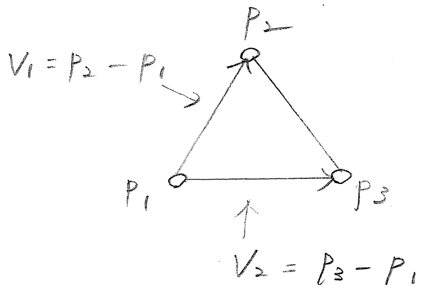
Compare this equation to the "standard" algebraic plane equation

$$A\,x + B\,y + C\,z + D = 0.$$

One sees that:

$$\underline{D = -(P_0 \bullet N)} = -(A\,x_0 + B\,y_0 + C\,z_0)$$

$$\text{"}B\text{"} = \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$



$$N = V_1 \otimes V_2 \Rightarrow A, B, C$$

$$D = -(P_1 \cdot N) \text{ or}$$
$$-(P_2 \cdot N)$$

11.3 Geometric Operations on Points, Lines and Planes

Now apply the basic vector functions to solve a number of common geometric operations on points, lines and planes.

11.3.1 Distance from a Point to a Line

Given a line defined by the point P and direction vector V, and a point Q as shown in Figure 11-16, one can compute the distance to the line using the vector functions. Derive the computations using geometric interpretation and insight. As we did with the vector Area function, begin by forming a parallelogram with P, Q and V by duplicating the line PQ and the vector V to form four sides. The distance "d" is its height. The area of this parallelogram is the base times height, or |V| * d.

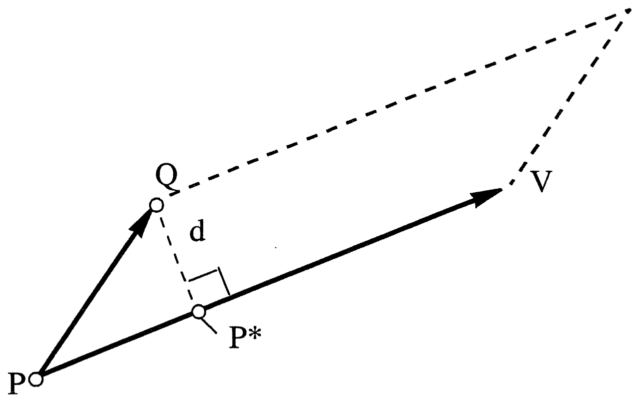


Figure 11-16. Distance d from a Point Q to a Line P-V.

Therefore,

$$d = \frac{|V \otimes (Q - P)|}{|V|} = \frac{\text{VArea(V, VSub(Q, P))}}{\text{VMag(V)}}$$

Let $P^* = P \oplus t\,V$, for some t. Then,

$$t = \frac{\pm|P^* - P|}{|V|} = \frac{(Q - P) \bullet V}{V \bullet V} = \frac{\text{VDot(VSub(Q, P), V)}}{\text{VDot(V, V)}}$$

Examine the cases shown in Figure 11-17

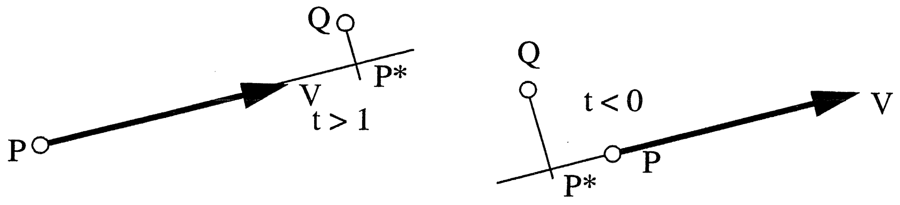


Figure 11-17. Examples of Distance of Point to Line.

11.3.2 Distance Between Two 3D Lines

The distance between two 3D lines, i.e. the smallest distance, occurs along a line perpendicular to both given lines. First, observe that two lines will either be parallel or not. If V_1 and V_2 are parallel, they lie in a plane, so one can think of the "area" between them.

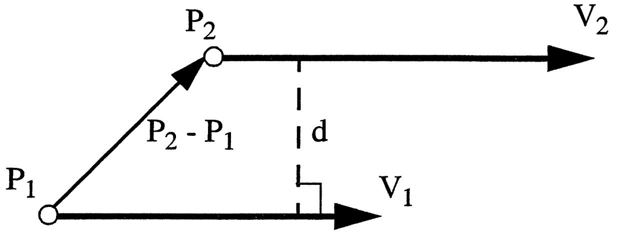


Figure 11-18. Distance Between Parallel Lines.

Using logic similar to that of finding the distance from a point to a line:

$$d = \frac{|V_1 \otimes (P_2 - P_1)|}{|V_1|} = \frac{\mathrm{VArea}(V_1, \mathrm{VSub}(P_2, P_1))}{\mathrm{VMag}(V_1)}$$

If V_1 and V_2 are not parallel, then they form a volume for which the distance d is the height:

$$d = \frac{|(V_1 \otimes V_2) \bullet (P_2 - P_1)|}{|V_1 \otimes V_2|} = \frac{\mathrm{Vol}(V_1, V_2, \mathrm{VSub}(P_2, P_1))}{\mathrm{VArea}(V_1, V_2)}$$

Note that the VArea result above will be zero if the lines are parallel. This can be used as a decision value in the algorithm to compute the distance between lines.

11.3.3 Intersection of Two Lines

Two 3D lines intersect when the distance between them is 0 (approximately, of course). To compute the point of intersection, P*, using vectors, solve the vector equation expressing the statement “the point on line 1 that is the same as the point on line 2”:

$$P_1\,(t) = P_2\,(s).$$

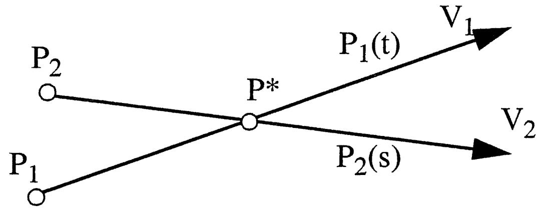


Figure 11-19. Intersection of Two Lines.

The problem is to find the parameter values for t and s that solve this equation. Substituting the vector expressions for the lines yields a vector equation in the variables s and t:

$$P_1 \oplus t\,V_1 = P_2 \oplus s\,V_2$$

Begin solving this equation by removing the absolute positions P_1 and P_2:

$$t\,V_1 = (\,P_2 - P_1\,) \oplus s\,V_2$$

Now cross both sides with V_2 to eliminate s:

$$t\,(\,V_1 \otimes V_2\,) = (\,(P_2 - P_1) \otimes V_2\,) \oplus s\,(\,V_2 \otimes V_2\,)$$

Of course, $(\,V_2 \otimes V_2\,) = 0$ by definition, so

$$t\,(\,V_1 \otimes V_2\,) = (\,(\,P_2 - P_1\,) \otimes V_2\,).$$

Dot this equation with $(V_1 \otimes V_2)$ to make a scalar equation in t,

$$t\,(\,V_1 \otimes V_2\,) \bullet (\,V_1 \otimes V_2\,) = (\,(P_2 - P_1) \otimes V_2) \bullet (\,V_1 \otimes V_2).$$

Finally,

$$t = \frac{[(P_2 - P_1) \otimes V_2] \bullet (V_1 \otimes V_2)}{(V_1 \otimes V_2) \bullet (V_1 \otimes V_2)}$$

Similarly, solving for s:

$$s = \frac{[(P_2 - P_1) \otimes V_1] \bullet (V_1 \otimes V_2)}{(V_1 \otimes V_2) \bullet (V_1 \otimes V_2)}$$

The denominator of these expressions is zero for $(V_1 \otimes V_2) = 0$, so this must be a degenerate condition. Geometrically, the vector $V_1 \otimes V_2$ is the common normal between the two vectors. If this is 0, then the vectors must be parallel or lie along the same line, and possibly also be overlapping. These must be handled as special cases in code.

How do we know if the intersection point is on the segment P_1 (t)? This is the advantage of parametric notation. The parameter value is a measure of the position of a point along the line. If $0 \leq t \leq 1$, the point is along the segment P_1 to P_1+V_1. If $t < 0$, the point is "before" P_1, and if $t > 1$, it is "after" P_1+V_1. Similar logic holds for s and the line between P_2 and P_2+V_2.

This is not as messy to code as it looks, see Figure 11-20.

```
Boolean LineLineIntersection( vector p1, vector v1,
        vector p2, vector v2, double *s, double *t )
{
        double numerator, ndotn;
        vector n, p21;
        n = VCross( v1, v2 );
        ndotn = VDot( n, n );
        if( VMag(n) < TOOSMALL )
                return FALSE;
        p21 = VSub( p2, p1 );
        numerator = VDot( VCross( p21, v2 ), n);
        *t = numerator / ndotn;
        numerator = VDot( VCross( p21, v1 ), n);
        *s = numerator / ndotn;
        return TRUE;
}
```

Figure 11-20. Line-line Intersection Function.

With s and t known, P* is computed using $P_1(t)$ or $P_2(s)$.

Figure 11-21 is an example of the use of the `LineLineIntersection` routine.

```
double s, t;
vector pstar;
if( LineLineIntersection(p1,v1,p2,v2,&s, &t) == TRUE ){
        pstar = VAdd( p1, VScale( v1, t ) );
        printf("the lines intersect at (%f,%f,%f)\n",
                pstar.x, pstar.y, pstar.z );
} else {
        printf("the lines do not intersect\n" );
}
```

Figure 11-21. Example of the Use of `LineLineIntersection`.

11.3.4 Intersection of a Line with a Plane

Another common problem in computational geometry is finding the point of intersection between a line and a plane. As in the case of the line-line intersection, this problem can be

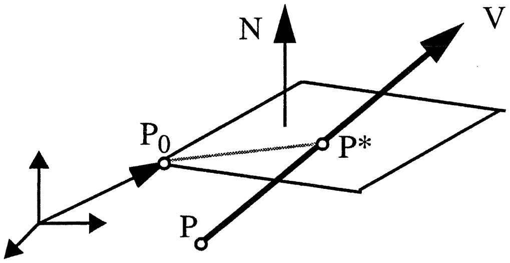


Figure 11-22. Line and Plane Intersection.

expressed geometrically as, “there is a point on the plane that is also on the line.” In vector notation, this becomes

$(P^* - P_0) \bullet N = 0$ “any point on the plane”

$P^*(t) = P \oplus t\,V$ “any point on the line”

Substituting P*,

$$[(P \oplus t\,V) - P_0] \bullet N = 0$$

Expanding:

$$t(V \bullet N) + (P - P_0) \bullet N = 0$$

Finally,

$$t = -\frac{(P - P_0) \bullet N}{V \bullet N}$$

The possibility of a zero in the denominator should always gain our attention. In this case, as in most vector-based computations, it is a warning of a special geometric situation. If V • N is (approximately) zero, the line is parallel to the plane, so there can be no intersection. Of course, the line could be in the plane.

If the plane is given as a plane coefficient vector, B, the line-plane intersection is computed as:

$$t = -\frac{\overset{\overset{(P-P_0)}{\parallel}}{\vec{P}} \bullet B}{V \bullet B}$$

$V \cdot N = V \cdot B$

$(x, y, z, 0) \cdot (\quad)$

11.4 Change of Basis in Vector-Matrix Form

Consider the coordinates of a position P in two different coordinate systems, XYZ_1 and XYZ_2, as shown in Figure 11-23.

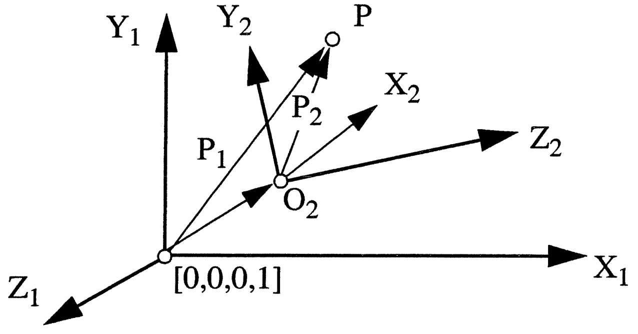


Figure 11-23. Two Coordinate Systems.

Given a vector in XYZ_1, P_1, we would like to find its coordinates with respect to XYZ_2, P_2, using a transformation **M**:

$$P_2 = P_1 \mathbf{M}.$$

The problem is to find the transformation **M**, called the change of basis. It can be assumed that we know the origin of the second coordinate system, O_2, and the unit direction vectors of the X_2, Y_2 and Z_2 axes, called I_2, J_2 and K_2, all with respect to XYZ_1. This information allows us to form four equations with known vectors for P_1 and P_2, as summarized in the following table:

P_1 (XYZ_1 Coordinates):	P_2 (XYZ_2 Coordinates):
I_2	[1,0,0,0]
J_2	[0,1,0,0]
K_2	[0,0,1,0]
O_2	[0,0,0,1]

This set of equations can be expressed in matrix form,

$$[I] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} I_2 \\ J_2 \\ K_2 \\ O_2 \end{bmatrix} \mathbf{M} = \begin{bmatrix} I_{2x} & I_{2y} & I_{2z} & 0 \\ J_{2x} & J_{2y} & J_{2z} & 0 \\ K_{2x} & K_{2y} & K_{2z} & 0 \\ O_{2x} & O_{2y} & O_{2z} & O_{2w} \end{bmatrix} M$$

where **M** is a 4 by 4 matrix. Solving for **M**,

$$[I] = [I\,J\,K][M]$$

$$[I\,J\,K]^{-1}[I] = [M] = [I\,J\,K]^{-1}[I\,J\,K][M] \qquad \mathbf{M} = \begin{bmatrix} I_2 \\ J_2 \\ K_2 \\ O_2 \end{bmatrix}^{-1}$$

Therefore, the transformation of a vector from one coordinate system to another can be found by finding the inverse of the matrix formed by putting the three unit direction vectors (in homogeneous form) in the first three rows and the origin position vector in the final row, of the second system with respect to the first.

The matrix inverse is a computationally expensive operation, so it is desirable to further refine this process by factoring the matrix on right hand side into a rotation matrix and a translation matrix,

↓ ↓

$$\begin{bmatrix} I_2 \\ J_2 \\ K_2 \\ O_2 \end{bmatrix} = \begin{bmatrix} I_2 \\ J_2 \\ K_2 \\ I \end{bmatrix} \begin{bmatrix} I \\ I \\ I \\ O_2 \end{bmatrix} = \begin{bmatrix} I_{2x} & I_{2y} & I_{2z} & 0 \\ J_{2x} & J_{2y} & J_{2z} & 0 \\ K_{2x} & K_{2y} & K_{2z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ O_{2x} & O_{2y} & O_{2z} & 1 \end{bmatrix}$$

where the I's represent the respective row of the identity matrix. The inverse of a product of matrices is the reverse order product of the respective inverses,

$$\mathbf{M} = \begin{bmatrix} I_2 \\ J_2 \\ K_2 \\ O_2 \end{bmatrix}^{-1} = \begin{bmatrix} I \\ I \\ I \\ O_2 \end{bmatrix}^{-1} \begin{bmatrix} I_2 \\ J_2 \\ K_2 \\ I \end{bmatrix}^{-1}$$

and the inverse of a translation is simply the translation by the negative vector. The rotation matrix is an orthonormal matrix, meaning the rows are orthogonal unit vectors. The inverse of an orthonormal matrix is simply its transpose, the matrix formed by changing the rows to columns. Using this information, the equation becomes,

$$M = \begin{bmatrix} I_2 \\ J_2 \\ K_2 \\ O_2 \end{bmatrix}^{-1} = \begin{bmatrix} I \\ I \\ I \\ -O_2 \end{bmatrix} \begin{bmatrix} I_2 \\ J_2 \\ K_2 \\ I \end{bmatrix}^{T} = \begin{bmatrix} I \\ I \\ I \\ -O_2 \end{bmatrix} \begin{bmatrix} I_{2x} & J_{2x} & K_{2x} & 0 \\ I_{2y} & J_{2y} & K_{2y} & 0 \\ I_{2z} & J_{2z} & K_{2z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where "$-O_2$" is actually computed as [-x, -y, -z, 1].

This is a compact, simple and efficient computation for **M**. The translation matrix is constructed simply by assigning the negative of O_2 to the last row of a matrix that has been initialized to identity, and the transposed rotation matrix is formed by assigning I_2, J_2 and K_2 as the first three columns of a matrix that has been initialized to identity. The transformation **M** is the product of these matrices.

As an example, this procedure can be used to find the viewing transformation, **V**. The first coordinate system is XYZ_w, the world coordinate system, and the second is XYZ_e, the eye coordinate system. The resulting transformation is **V**, the viewing transformation. World coordinates will be transformed into eye coordinates.

Given the view vector end points, EYE and AIM, one can find O_2, I_2,J_2 and K_2 using vector operations (see Figure 11-24).

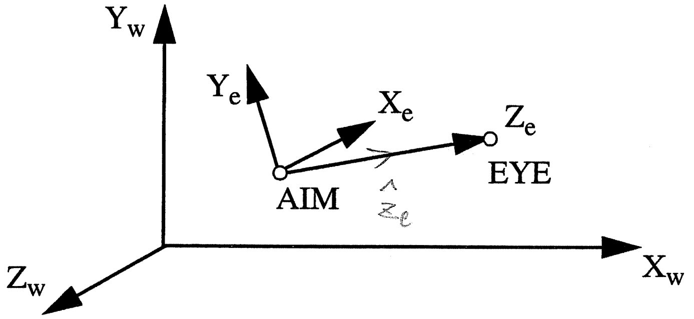


Figure 11-24. Change of Basis for the Viewing Transformation.

The process involves several steps:

1. K_2 = || EYE - AIM ||, the unit direction vector of the Z_e axis with respect to XYZ_w.
   $\hat{Z_e}$

2. The next step is to find I_2, the direction of the X_e axis. It is necessary to specify a direction, often called the UP direction, which, when crossed with K_2, forms the direction of I_2. To follow the viewing conventions established during the trigonometric version of the viewing transfmoration discussed earlier, UP = J ([0,1,0,0]), the unit vector along the Y_w axis. One must be careful, however, when UP is parallel to K_2, i.e. when UP $\otimes$ K_2 = 0, for example, looking along Y_w. In this case, one must specify another UP vector. To follow the conventions established in these notes, specify I, [1,0,0,0], the unit X_w axis.

```
IF ( $J \otimes K_2$ ) approx.= 0 THEN
        $I_2 = I$
ELSE
        $I_2 = \| J \otimes K_2 \|$
```

3. $J_2 = K_2 \otimes I_2$ (note that J_2 will be a unit vector because K_2 is perpendicular to I_2)
4. Create the transposed rotation matrix, **R**, from I_2, J_2, K_2 as described earlier.
5. Create the translation matrix, **T** = T(-AIM), because O_2 = AIM in this case.
6. Finally, compute **V** = **T R.**

As an example, consider the situation shown in Figure 11-25, with AIM = [0,0,0,1] and EYE = [1,0,0,1].

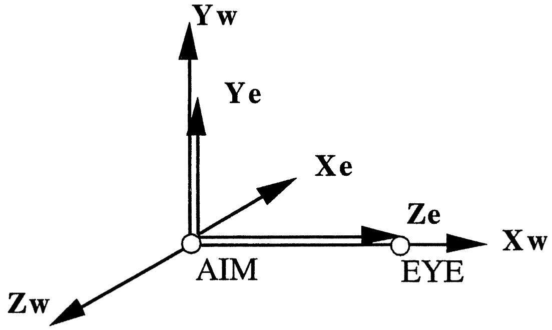


Figure 11-25. Example of Change of Basis Applied to the Viewing Transform.

Following the steps,

1. $K_2 = \| \text{EYE} - \text{AIM} \| = \| [1,0,0,1] - [0,0,0,1] \| = [1,0,0,0]$.

2. $J \otimes K_2 = [0,1,0,0] \otimes. [1,0,0,0] = [0,0,-1,0]$ (i.e., -K).

   This is not zero length, so $I_2 = \| J \otimes K_2 \| = [0,0,-1,0]$.

3. $J_2 = K_2 \otimes I_2 = [1,0,0,0] \otimes [0,0,-1,0] = [0,1,0,0]$.

4. Form the transposed rotation matrix, **R**:

$$\mathbf{R} = \begin{bmatrix} I_2 \\ J_2 \\ K_2 \\ I \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5. Form the negative translation, **T**, noting that O_2 = AIM:

$$\mathbf{T} = \begin{bmatrix} I \\ I \\ I \\ -O_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6. Finally, compute **V** = **T R**:

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

$Xe \cdot X = v[0][0]$
$Xe \cdot Y = v[1][0]$
$Xe \cdot Z = v[2][0]$

Look at the columns of the **R** and **V** matrices in the previous example. They are the unit direction vectors of XYZ_e with respect to XYZ_w. This means that given **V**, one can draw the axes of XYZ_e.

11.5 Instance Transformations

The logic used in the derivation of the change of basis transformation also is useful in forming *instance transformations* for drawing objects that are represented in a "local" coordinate system. Given an object represented in its own local coordinate system, we want to draw it at a given position and orientation in world space by transforming the local coordinates to world coordinates as lines are drawn. This is especially applicable when using the **CTM** matrix of a 3D pipeline.

For example, it is simple to compute the points on a circle with its center at the origin and with a unit radius. Refer to this as the *canonical* circle in its local coordinate system. Drawing an instance of a circle at a point on an arbitrary plane in 3D space is another problem, however. We would like to compute the local coordinates of the points on a circle and use an instance transformation to map them to points on the 3D circle in world coordinates. The problem is to find this transformation from the information describing the world position and orientation.

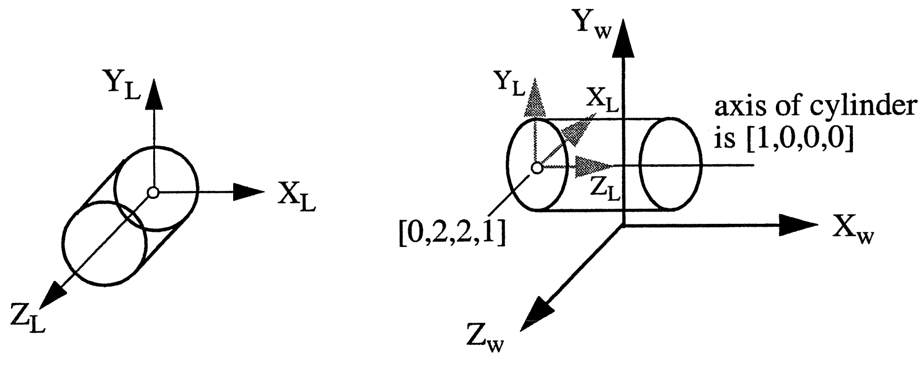
The problem, as usual, is finding the transformation **M** that maps XYZ_L coordinates into XYZ_w coordinates. The derivation of this transformation is similar to the derivation of the change of basis transformation. First, compute the origin and direction vectors of the local coordinate system in world coordinates from the position and orientation information given for the object. Next, form **M** from these vectors like the change of basis derivation, using the equation,

$P_w = P_L\,\mathbf{M}$:

$$\begin{bmatrix} X_L \\ Y_L \\ Z_L \\ O \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{M} = \mathbf{M}$$

This **M** is made by simple row assignments from the four vectors, the three unit direction vectorsX_L, Y_L, and Z_L, and the origin vector O, all expressed in world coordinates.

For example, compute the instance transformation to map a unit cylinder to the world position and orientation as shown in Figure 11-26. In the figure, the XYZ_L axes have been drawn



Basic cylinder
in local coordinates

Instance of cylinder
in world coordinates

Figure 11-26. Cylinder Instance Transformation Example.

in world coordinates. By observation we find that O = [0,2,2,1], X_L = [0,0,-1,0], Y_L = [0,1,0,0], and Z_L = [1,0,0,0]. Next, these vectors are assembled into the instance transformation **M**:

$$\mathbf{M} = \begin{bmatrix} X_L \\ Y_L \\ Z_L \\ O \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 2 & 2 & 1 \end{bmatrix}$$

Notice that the rotation sub-matrix is a simple $\mathbf{R}_Y(90)$.

If the direction vectors could not have been found by observation, then the viewing transformation vector operations described in the previous section could have been used to compute X_L, Y_L and Z_L.

Also, if the cylinder instance is to be scaled, such as changing the radius (or radii, e.g. making the cylinder elliptical) or length, then these transformations should be concatenated with the positioning and orienting transformation **M** to form the instance transformation. The scaling must be done in XYZ_L, before **M**.

11.6 Transforming Continuous Geometries

Transformations have been discussed in terms of points and lines. As one would expect, it is possible, and often desirable, to transform equations representing continuous geometry.

For example, consider the plane: A x +B y +C z +D = 0. First re-write the algebraic form in matrix form:

$\underline{P} \cdot \underline{B} = 0$

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = 0$$

$P' = P \cdot M$

Transform all points in the plane by the transformation **M**, mapping [x,y,z,1] → [x', y', z', 1]:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \mathbf{M}$$

The coefficients of the transformed plane, B', are found by post-multiplying the above equation by $\mathbf{M}^{-1}$B:

The original plane equation is

$P * B = 0.$

The tranformed points are

$P' = P\,\mathbf{M},$

so, inverting to find P,

$P = P'\,\mathbf{M}^{-1}.$

Subsituting this equation for P into the original plane equation above,

$(P'\,\mathbf{M}^{-1}) * B = 0.$

Re-group the matrix and B multiplication,

$P'\,(\mathbf{M}^{-1} B) = 0.$ $B^T (M^{-1})^T$ for col. vector

Or, $B' = \mathbf{M}^{-1} B$, and $P' * B' = 0$ is the new plane equation.

For example, consider the perspective projection of the plane y + z - 1 = 0, the plane along the projector through y = 1 (Figure 11-27). The plane equation is $[x\ y\ z\ 1]\ [\ 0\ 1\ 1\ -1\ ]^T$. The

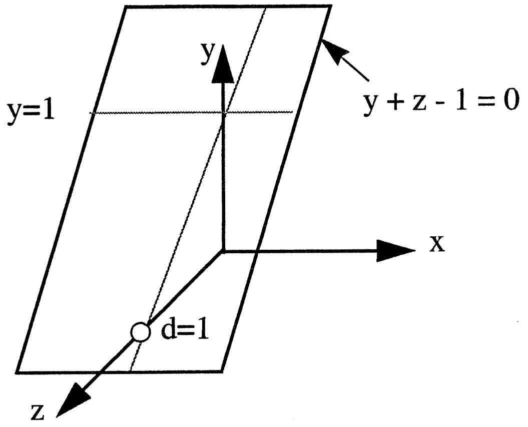


Figure 11-27. The Plane y + z - 1 = 0.

transformation **M** is the perspective projection matrix, **P**. For d = 1, this becomes:

$$\begin{bmatrix} A' \\ B' \\ C' \\ D' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ -1 \end{bmatrix}$$

So, the transformed plane equation is y - 1 = 0, or y = 1. This shows that the top clipping plane of the viewing volume transforms to a plane perpendicular to the y axis (Figure 11-28).

Using similar logic, the coefficients of a transformed conic, i.e., circle, sphere, parabola,

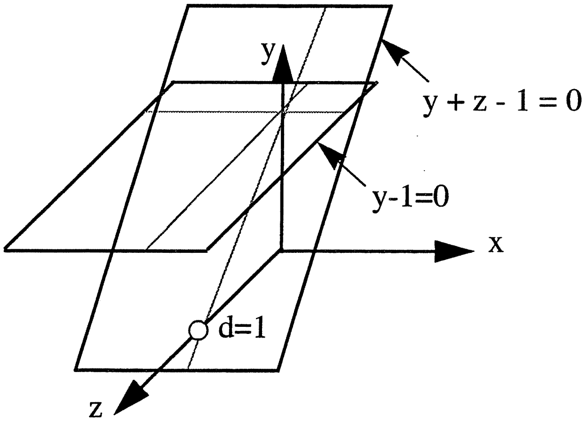


Figure 11-28. Original and Transformed Planes.

ellipse, or hyperbola, can be derived. For simplicity, we will consider only two-dimensions in the following. The general 2D conic equation is,

$$a_{20}x^2 + 2a_{11}xy + a_{02}y^2 + 2a_{10}x + 2a_{01}y + a_{00} = 0.$$

This can be put into matrix form,

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a_{20} & a_{11} & a_{10} \\ a_{11} & a_{02} & a_{01} \\ a_{10} & a_{01} & a_{00} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \mathbf{A} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0$$

$$\text{or, } P\,A\,P^T = 0$$

Transforming the curve is transforming all its points, so again apply the transformation **M** to [x, y, 1] to produce [x',y',1]. Inverting the equation to solve for [x,y,1] in terms of $\mathbf{M}^{-1}$ and [x',y',1], and substituting into the equation above, the transformed conic is found:

$$\mathbf{A}' = \mathbf{M}^{-1}\,\mathbf{A}\,(\mathbf{M}^{-1})^T.$$

As an example, consider translating a unit circle at the origin by [1,0,0,0]. The **A** matrix equation becomes:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0 \qquad \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Therefore, the transformed **A** matrix, **A'**, is found as follows:

$$\mathbf{A}' = \begin{bmatrix}1&0&0\\0&1&0\\1&0&1\end{bmatrix}^{-1}\begin{bmatrix}1&0&0\\0&1&0\\0&0&-1\end{bmatrix}\left(\begin{bmatrix}1&0&0\\0&1&0\\1&0&1\end{bmatrix}^{-1}\right)^{T} = \begin{bmatrix}1&0&0\\0&1&0\\-1&0&1\end{bmatrix}\begin{bmatrix}1&0&0\\0&1&0\\0&0&-1\end{bmatrix}\begin{bmatrix}1&0&-1\\0&1&0\\0&0&1\end{bmatrix}$$

$$\mathbf{A}' = \begin{bmatrix}1&0&-1\\0&1&0\\-1&0&0\end{bmatrix}$$

This is the equation: $(x-1)^2 + y^2 - 1 = 0$, which, as expected, is the equation of a unit circle with center [1 0].

In three dimensions, the conics are ellipsoids, spheres, paraboloids, etc. The matrix form is

$$\begin{bmatrix}x & y & z & w\end{bmatrix}\mathbf{A}\begin{bmatrix}x\\y\\z\\w\end{bmatrix} = 0$$

where the matrix **A** for 3D conics is 4 by 4. Exactly the same as the 2D case above, transforming the 3D conic by a transformation **M** creates a transformed conic represented by $\mathbf{A}' = \mathbf{M}^{-1}\,\mathbf{A}\,(\mathbf{M}^{-1})^{T}$.

11.7 Rotation About An Arbitrary Axis Through The Origin

Using basic transformations one can derive the transformation matrix that rotates an object about an arbitrary unit axis, u, through the origin.

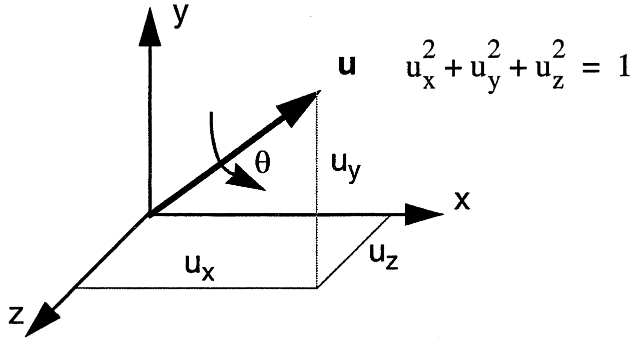


Figure 11-29. Rotation Axis Notation.

The approach is to first perform a change of basis that places the object in a coordinate system in which u becomes the z axis. Then rotate the object by the angle θ about this z axis and transform the object back into the original coordinate system. The first transformation is precisely the viewing transform, hence derive it using the change of basis in vector-matrix logic.

The direction K_2 is u. Next compute I_2:

$I_2 = \| (J \otimes K_2) \|$

Now compute $J_2 = K_2 \otimes I_2$

$$I_2 = \begin{bmatrix} \frac{u_z}{\sqrt{u_x^2 + u_z^2}} & 0 & \frac{-u_x}{\sqrt{u_x^2 + u_z^2}} \end{bmatrix}$$

$$J_2 = \begin{bmatrix} \frac{-u_x u_z}{\sqrt{u_x^2 + u_z^2}} & \sqrt{u_x^2 + u_z^2} & \frac{-u_y u_z}{\sqrt{u_x^2 + u_z^2}} \end{bmatrix}$$

Now create **R**, the transposed rotation transformation, using these as the columns:

$$\mathbf{R} = \begin{bmatrix} \frac{u_z}{\sqrt{u_x^2 + u_z^2}} & \frac{-u_x u_y}{\sqrt{u_x^2 + u_z^2}} & u_x \\ 0 & \sqrt{u_x^2 + u_z^2} & u_y \\ \frac{-u_x}{\sqrt{u_x^2 + u_z^2}} & \frac{-u_y u_z}{\sqrt{u_x^2 + u_z^2}} & u_z \end{bmatrix}$$

Form the composite transformation as (recall $\mathbf{R}^{-1} \rightarrow \mathbf{R}^{T}$):

$\mathbf{M} = \mathbf{R}\,\mathbf{R}(\theta,z)\,\mathbf{R}^{T}$

$$\mathbf{M} = \begin{bmatrix} \frac{u_z}{\sqrt{u_x^2 + u_z^2}} & \frac{-u_x u_y}{\sqrt{u_x^2 + u_z^2}} & u_x \\ 0 & \sqrt{u_x^2 + u_z^2} & u_y \\ \frac{-u_x}{\sqrt{u_x^2 + u_z^2}} & \frac{-u_y u_z}{\sqrt{u_x^2 + u_z^2}} & u_z \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{u_z}{\sqrt{u_x^2 + u_z^2}} & 0 & -\frac{u_x}{\sqrt{u_x^2 + u_z^2}} \\ -\frac{u_x u_y}{\sqrt{u_x^2 + u_z^2}} & \sqrt{u_x^2 + u_z^2} & -\frac{u_y u_z}{\sqrt{u_x^2 + u_z^2}} \\ u_x & u_y & u_z \end{bmatrix}$$

After considerable algebra, one arrives at a convenient form:

$$\mathbf{M} = \begin{bmatrix} u_x^2\text{vers}\theta + \cos\theta & u_xu_y\text{vers}\theta + u_z\sin\theta & u_xu_z\text{vers}\theta - u_y\sin\theta \\ u_xu_y\text{vers}\theta - u_z\sin\theta & u_y^2\text{vers}\theta + \cos\theta & u_yu_z\text{vers}\theta + u_x\sin\theta \\ u_xu_z\text{vers}\theta + u_y\sin\theta & u_yu_z\text{vers}\theta - u_x\sin\theta & u_z^2\text{vers}\theta + \cos\theta \end{bmatrix}$$

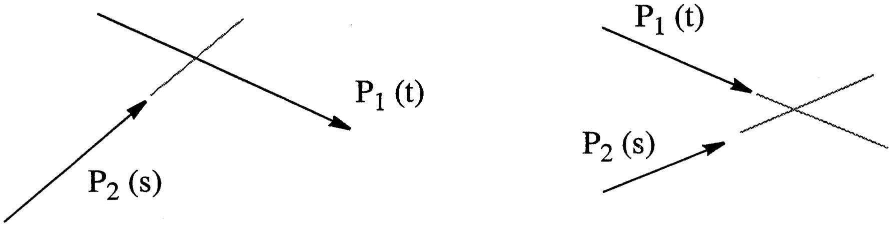
where versθ = 1 - cosθ.

11.8 Review Questions

1. Given AIM = [3,2,1] and **V** below, compute the unit direction vectors of the eye coordinate system and EYE for d = 2.

$$\mathbf{V} = \begin{bmatrix} 0.9487 & 0.0953 & -0.3015 & 0 \\ 0 & 0.9535 & 0.3015 & 0 \\ 0.3162 & -0.2860 & 0.9045 & 0 \\ -3.1623 & -1.9069 & -0.6030 & 1 \end{bmatrix}$$

2. How can one determine if two parallel vectors are in the same or opposite directions?
3. What would be the approximate values for s and t for these line-line intersection cases?



4. How would the condition "line in plane" be computed?
5. Given a viewing transformation **V**, show how to compute the translation and rotation matrices for the change of basis formulation. Also, show how the AIM location can be found from **V**.

Chapter 12. Hidden Line and Surface Removal

As you are (hopefully) discovering, it is not that difficult to produce a 3D wireframe drawing. It is considerably more difficult to make it look solid by removing the hidden lines or hidden surfaces.

Hidden line removal (HLR) and hidden surface removal (HSR) require more than just edge information. The objects must have surfaces that can hide edges.

HLR and HSR Algorithms are divided into two broad classes:

1. Line algorithms that output lines; for example, for a plotter,
2. Raster algorithms that output pixels; for example, for a raster display.

The fundamental problem in these processes is to determine the *visibility* of a geometric element, either a line or surface, with respect to all other objects in a scene. Determining the visible portion of an element is actually computing the portion that is not hidden by another element. Once the visibility computations are finished, the scene is drawn by displaying all visible portions of elements. Of course, visibility depends on the viewing information.

The concepts underlying visibility are introduced by first discussing back plane rejection.

12.1 Back-Plane Rejection

Given an object in 3D space bounded by planes and a viewing direction, one can sometimes reduce the data that must be processed by applying *back-plane rejection*, which is also called *poor man's hidden line removal*. This is done before proceeding with HLR or HSR.

The process involves determining if the plane in question is facing toward or away from the observer. This will determine if the plane is visible. To test visibility, a *line of sight* from the plane to the eye must be simulated (Figure 12-1).

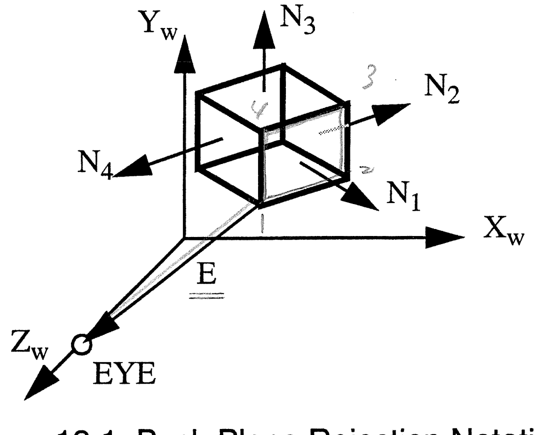


Figure 12-1. Back-Plane Rejection Notation.

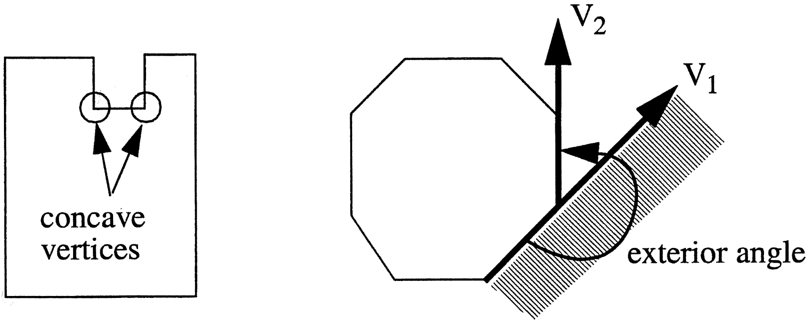
A plane is *back-facing* if

$E \bullet N < 0$,

and need not be considered in subsequent processing. Back-plane rejection can be applied only if the data is a closed solid when it is not possible to "see" the "back" side of a plane.

If the object is convex, back-plane rejection solves the HLR problem.

What does it mean for an object to be *convex* or concave (non-convex)? In general, a polygon is convex if the exterior angle between any two adjacent edges never is less than 180°. Another specification for convexity is that the plane normals computed as the cross products of all adjacent edges taken as direction vectors must face the same direction.



concave polygon convex polygon

Figure 12-2. Concave and Convex Polygons.

A convex object is best described as an object which, if shrink-wrapped with plastic, would have no cavities between the wrapper and its surfaces. We will see that hidden surface processing is considerably more complex for concave objects than convex.

12.2 3D Screen Space

When discussing the 3D wireframe pipeline, simple perspective was derived as a projection transformation from 3D to 2D. In order to perform HLR and HSR, it must be possible to determine if a given point in 3D space is “in front of” or “behind” another along a given *line of sight*. It was shown how line of sight is satisfied by projection. If two 3D points project to the same [x,y] coordinates on the picture plane, then they are along the same projector.

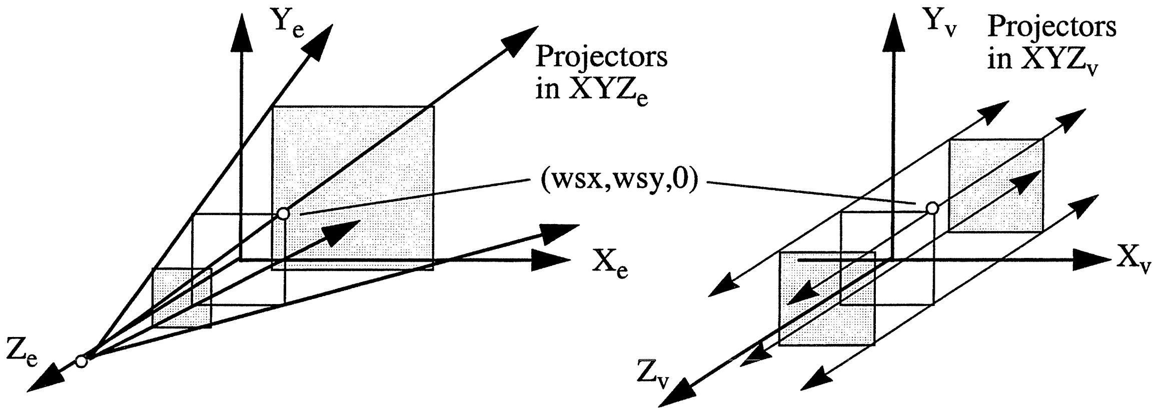


Figure 12-3. Lines of Sight Before and After Projection.

But, how can *depth comparisons* be performed after projection if the data is 2D? The answer is that, for HLR and HSR to determine visibility, one must retain the z coordinates of points "after" projection. The perspective projection transformation derived earlier, **P**, does create projected z coordinates for this purpose. However, some perspective projection transformations, namely those whose projectors pass through the origin, do not yield z coordinates that can be used for depth comparisons. We refer to 3D screen space coordinates as XYZ_V.

||
(h, v, z_e)
clipping coord.

12.3 Hidden Line Algorithms

The most common HLR approach is to compare each edge with each face to see if it is all or partially obscured. Approaches vary in that some compare edges to faces and others compare faces to edges. The problem is essentially one of clipping a line against a polygon.

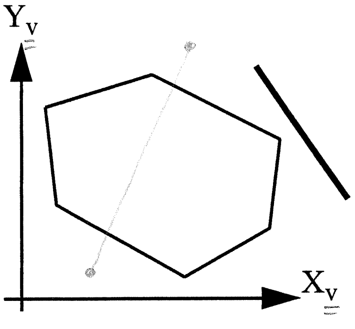
One of the first such algorithms was Robert's HLR algorithm. The general HLR idea is:

Step 1. Perform the viewing and projection transformations so that all lines lie in the 3D screen space.

Step 2. Compare each line with each polygon, in 2D. There are three possible cases:

2a. The line is "outside" the polygon. Action: keep the line.

O(n^3)



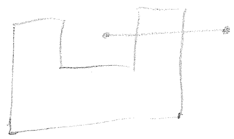


Figure 12-4. Line Outside a Polygon.

Note that determining if a line is outside a polygon requires (1) determining if any line of the polygon intersects the given line, and (2) determining if the end points of the given line lie inside the polygon (known as the "point containment test").

Examine the lines and polygon in Figure 12-5. Line AB has end points outside the polygon, but intersects it. Line CD has end points inside the polygon and also intersects it. Line EF is a simple case, the location of the end points can be used to classify the line as "crossing."

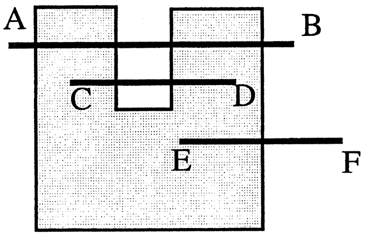


Figure 12-5. Lines and a Polygon.

2b. The line is "inside" the polygon in 2D. Determine if the line penetrates the plane of the polygon in 3D.

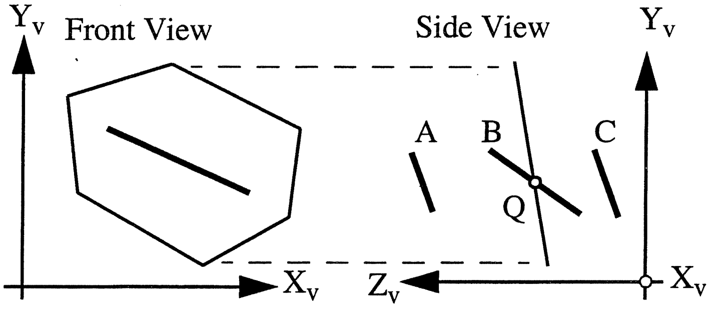


Figure 12-6. Two Views of a Line Inside a Polygon.

If it does not penetrate, determine if the line is completely in front of (e.g. line A), or completely behind (e.g. line C), the polygon. This involves 3D tests:

1. if it is in front, KEEP IT; if behind, eliminate it;
2. if the line does penetrate, keep only the portion in front of the polygon.

2c. The line intersects the polygon in 2D (line B).

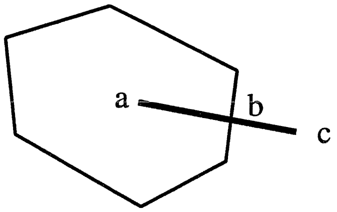


Figure 12-7. Line Intersecting a Polygon.

Break the line 'a-b-c' into 'a-b' and 'b-c'. Process 'a-b' by case 2b above, and process 'b-c' by step 2a above.

12.4 Raster Algorithms

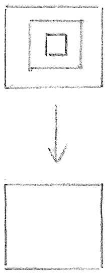
Raster technology is well suited to hidden surface (and line) removal. The following sections describe two well-known raster algorithms.

12.4.1 Painter's Algorithm (Newell-Newell-Sancha)

This simple approach uses the drawing order to create the final picture. The polygons are sorted based on decreasing distance (depth) from the eye. Each polygon is "filled", that is, the pixels within the projected boundaries of the polygon's image on the screen are written into the frame buffer, beginning with the polygon farthest from the observer and proceeding to the closest. Pixels within closer polygons are written after those in farther away polygons, and therefore overwrite them in pixel memory.

1. Project polygons into 3D screen space, (h, v. Zc)
2. Sort polygons by Z_v values,
3. Fill each in order, from farthest to nearest to the eye.

G Fill Begin
G Line To
G Fill End.



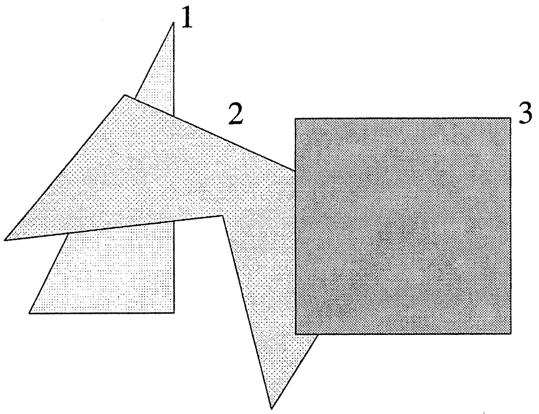


Figure 12-8. Overlapping Polygons in the Painter's Algorithm.

If two polygons overlap in 2D and intersect in Z_v, then depth sorting produces incorrect results. In this case, the two polygons must be split into non-intersecting polygons, which is a complex polygon clipping-like process. This greatly increases the complexity and diminishes the elegance of the algorithm.

The speed of the Painter's algorithm depends upon the filling speed of the graphics system because all polygons, visible or hidden, are filled. Filling is itself a computationally intensive process, so this is not the most efficient algorithm. It is, however, the simplest to implement.

12.4.2 Z-Buffer Algorithm

A more sophisticated approach, based somewhat on the Painter's algorithm, is to maintain a buffer (array) that holds the depth (usually z_v) for each pixel in the frame buffer. This depth buffer, sometimes called a z-buffer, is consulted prior to storing a pixel in pixel memory. This also requires that the polygon filling algorithm compute a depth for each pixel.

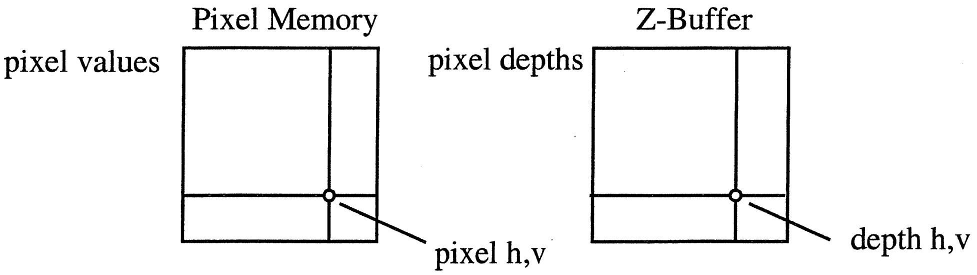


Figure 12-9. Z-Buffer Storage.

When a polygon is filled, a depth is computed for each pixel inside it and compared to the depth stored in the z-buffer for the corresponding pixel in pixel memory. If the polygon's pixel is "nearer to the eye" (greater-than for a right-handed system, less-than for a left-handed system), the new pixel depth is stored in the z-buffer AND the pixel value is stored in pixel memory. If the new pixel is farther away, then it is ignored. Therefore, after all polygons have been processed, the pixels remaining are the nearest pixels, i.e. the visible pixels.

Often, to avoid the excessive overhead of pixel-by-pixel graphics system calls, a separate array of pixel values is created for this algorithm. In this case, the picture is not drawn until all polygons have been processed. Then, the pixel values are written "en masse" to the display.

```
1)  Project polygons into 3D screen space.
2)  Initialize pixel and depth buffers (two dimensional arrays).
3)  FOR each polygon $P_i$ DO
        FOR each pixel & depth inside $P_i$ DO            (hi, vi, zci)
            IF depth "is closer than" stored depth[i,j] THEN
replace old value  {  depth[i,j] = depth
                   {  pixel[i,j] = pixel (or write the pixel to the screen directly)
            ENDIF
        ENDDO
    ENDDO
4)  Draw picture from pixel[][] data (if necessary)     "G Set Pixels"
```

The z-buffer approach has been implemented in hardware.

Graphic Card.

12.4.3 Watkins Algorithm

The Painter's algorithm is simple to implement, but cannot deal with complex intersecting polygons and "blindly" fills every polygon whether invisible or not. The z-buffer algorithm handles more complex geometry but requires large amounts of memory (the depth buffer). Both algorithms compute the picture in random pixel order, meaning pixels are computed anywhere on the screen, one or more times.

The Watkins algorithm was one of the first efficient *scan-line* hidden surface algorithms. The Watkins algorithm is designed expressly for display on a raster system, and bases all computations on scan-lines and pixels.

The entire scene is processed one scan-line at a time. For each scan-line, from the top of the viewport to the bottom, a horizontal plane, conceptually a *scan-plane*, is passed through the scene and intersected with all polygons. Because the intersection of a plane with a plane produces a line (or several lines for concave polygons), the horizontal scan-plane produces a set of lines, called *segments*. These segments are horizontal lines with respect to X_V - Y_V, but are in fact 3D lines in the 3D screen space.

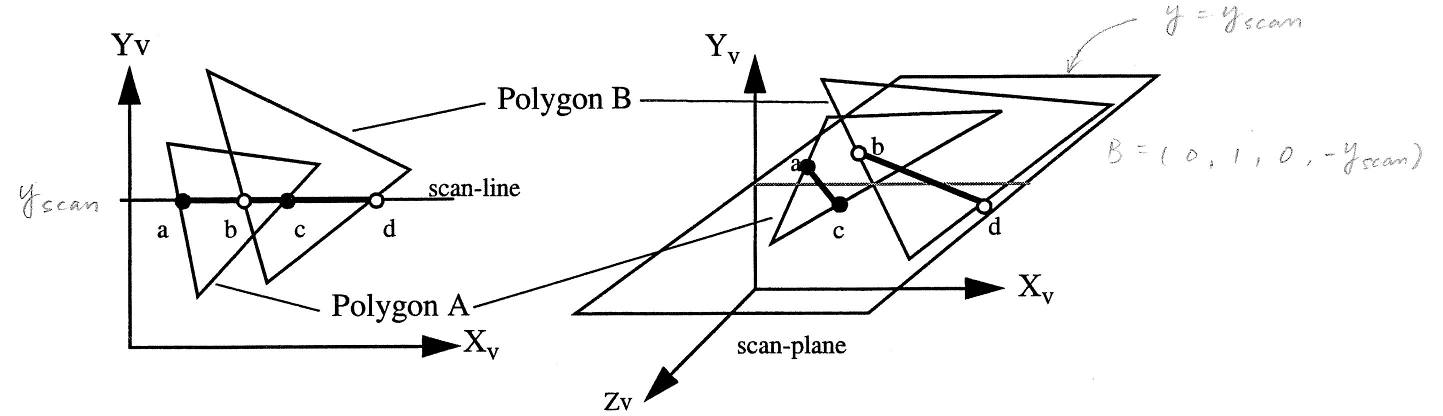


Figure 12-10. Scan-line and Scan-Plane.

Viewed in the scan-plane, the segments may overlap. Depth comparisons are made in the overlapping area(s) of segments to find which parts of segments are in front of others, i.e. visible. Overlapping segments are sub-divided and comparisons continue until no overlaps exist.

The resulting non-overlapping segments are rasterized into pixels and finally output to the display device. Each segment consists of a left x value and right x value.Then, we proceed to the next scan-line.

A segment is rasterized by computing the locations of the individual pixels along the horizontal scan-line at integral 'h' positions and writing these into pixel memory. In GRAFIC, the routine `SetPixels(h,v,n,pixels)` is used for this.

Due to the properties of lines and polygons in the 3D screen space, the Watkins' algorithm actually is *edge-driven*. That is, most computations are performed in 2D with some additional bookkeeping for the depth values.

Watkin's algorithm uses a *divide and conquer* (non-deterministic) algorithm for sub-dividing segments to remove overlaps. Simple segment conditions are processed, and other complex cases are sub-divided until they become simple.

(- [illegible])

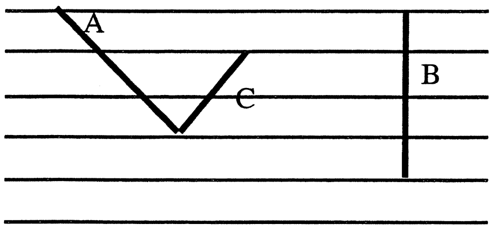
Scan-line coherence, the similarity between visible segment locations on one scan-line to the next, is used to eliminate unneeded subdivision. It is the scan-line coherence methodology and segment visibility resolving logic that distinguishes the Watkins algorithm from others, and makes it one of the fastest (and complex) HSR algorithms.

The Watkins segment rasterization phase can be changed to produce a raster image with only the edges, not the interior of surfaces. This is done by outputting "marker" pixels at the ends of resolved segments and "background" pixels at the interior pixels.

The Watkins algorithm can also output lines. Instead of rasterizing the resolved segments on a scan-line, the HLR version of Watkins remembers the segment boundaries, marked by the polygons that generate them, from one scan-line to the next.

By maintaining a list of growing lines, adjacent scan-lines can be compared to detect the appearance, disappearance, and continuation of lines.

Consider the following example:



scan-line	Action
2	New lines A and B.
3	New line C, extend A & B.
4	Extend A, B & C.
5	Extend A, B & C.
6	End A & C, Extend B.
7	End B.

When comparing the lines to the segments on the current scan-line, there are 3 possibilities:

1. A segment boundary has no counterpart growing line.

Start a new growing line with start and end positions at the given point.

2. A segment endpoint on the completed scan-line matches a growing line.

   Extend the growing line by setting its endpoint to the given point.

3. A growing line has no counterpart on the completed scan-line.

   The line ends: output it to the display or file being created and remove it from the data structure.

12.5 Polygon Clipping

HLR and HSR computations require that polygons be closed. For example, the first step in the Painter's Algorithm is to project the data, polygon, into XYZ_V. During this process, what should be done if an edge of a polygon lies outside the viewing frustum? Unfortunately, simply performing line clipping on the edges is wrong. Line-clipping algorithms are inadequate because the resulting polygon may not be closed, as illustrated in Figure 12-11.

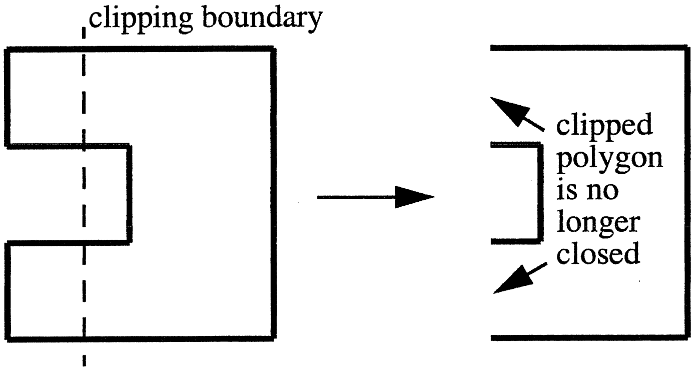


Figure 12-11. Line Clipping Applied to a Polygon.

Polygon clipping is needed to maintain closed polygons for later computations (such as filling and point containment tests).

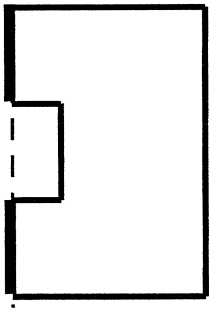


Figure 12-12. Polygon Clipping Adding Lines to Close a Polygon.

Polygon clipping appears to be considerably more difficult than line clipping (Figure 12-13).

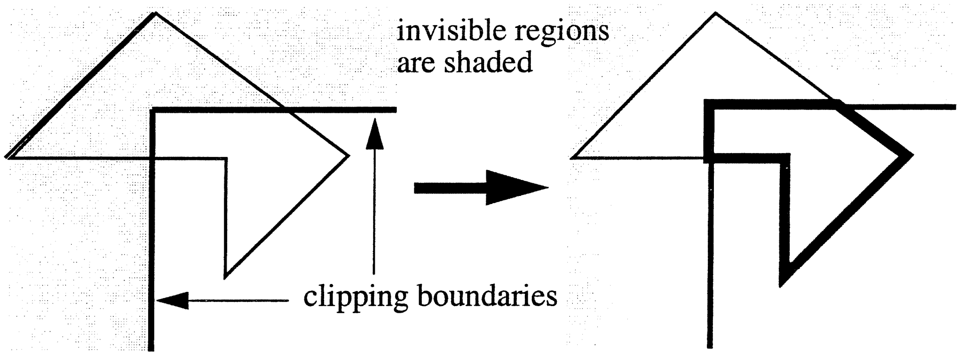


Figure 12-13. Complex Polygon Clipping.

The Sutherland-Hodgman algorithm [Sutherland] is an elegant approach to polygon clipping. The algorithm sequentially processes each edge around the perimeter of a polygon, testing the start and end vertices of the edge against the clipping boundaries. Given the start point of an edge, P_1, and the end point, P_2, there are four cases to consider as shown in Figure 12-14.

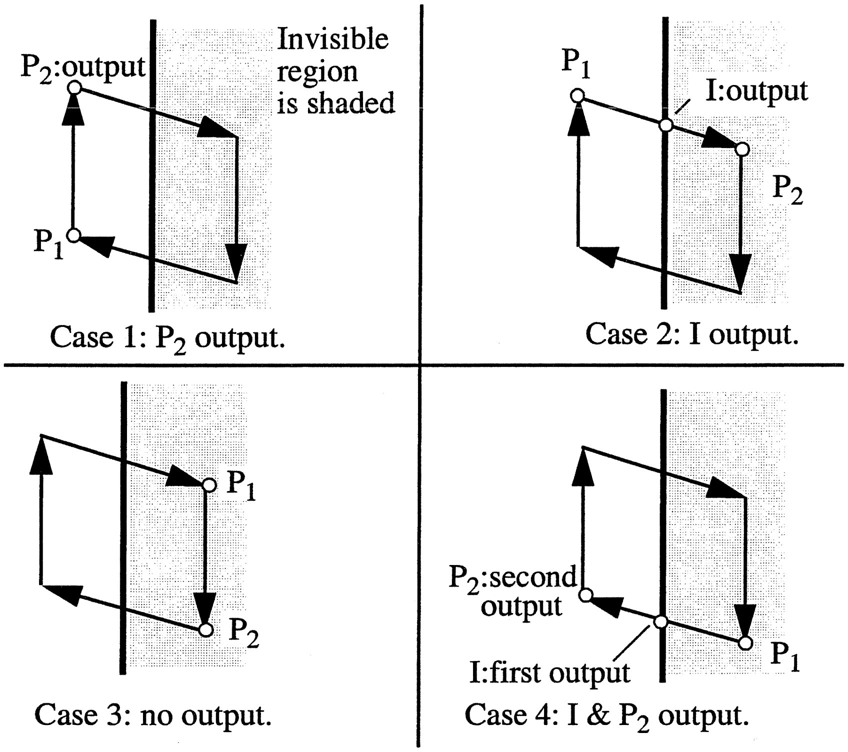


Figure 12-14. Four Clipping Cases for Polygon Clipping.

The algorithm processes each pair of edge end points, P_1 and P_2, clipping the edge against each boundary. Note that the process for each boundary is the same, i.e. as points are "output" from one boundary, they can be processed by the next. Each boundary evaluation is identical to the others, so the algorithm can be programmed using “re-entrant” code, i.e., one routine that processes P_1 and P_2 points can “call itself.”

A simpler approach is to clip the whole polygon against one boundary at a time, storing the output vertices in a second “buffer” (array of points). If any points are in the buffer after clipping on one boundary, then clip on the next boundary, this time reading points from the second buffer and outputting them to the first. Continue clipping on the remaining boundaries, reading from one buffer and storing into the other, toggling input and output each time. The buffers must be dimensioned at least twice the maximum number of vertices allowed per polygon. This can be understood intuitively by considering a polygon shaped like a saw blade.

Figure 12-15 shows a pseudo-code illustration of the basic procedure for clipping a polygon against one boundary, specified by an argument of data type `boundary` that represents the plane coefficient vector [A,B,C,D].

```
ClipPolygon(boundary B):
p1 = last vertex in input buffer;
FOR( p2 = each vertex of input buffer (1st to last) ) DO
      IF State(p1, B) != State(p2, B) THEN
            I = intersection of line p1-p2 with the boundary B;
            Append I to the output buffer;
      ENDIF
      IF State(p2, B) = IN THEN
            Append p2 to the output buffer;
      p1 = p2;
ENDFOR
```

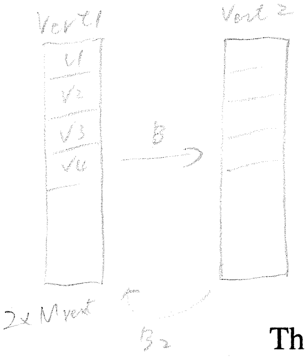


Figure 12-15. Pseudo-Code Polygon Clipping Routine.

The routine `State` in Figure 12-15 returns the state of the given point with respect to the given plane, either IN or OUT. An example of State in pseudo-code is shown in Figure 12-16.

```
State( Vertex P, boundary B):
IF P • B < 0 THEN
      State = OUT
ELSE
      State = IN;
ENDIF
END
```

Figure 12-16. An Example of the `State` Function.

For homogeneous polygon clipping, there would be seven planes, starting with WZERO. For example, consider Figure 12-17. The initial vertex buffer would contain the four

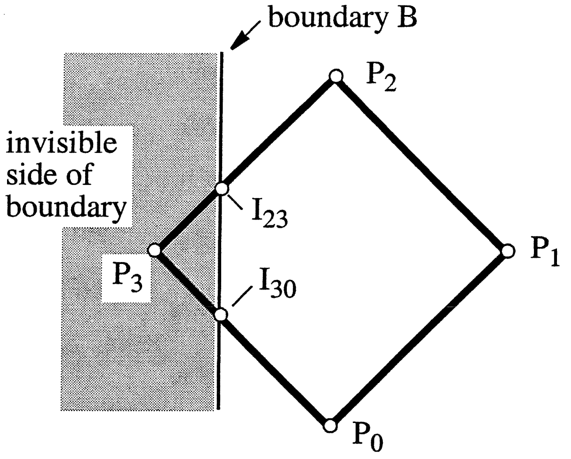


Figure 12-17. Example Polygon and Boundary.

ordered points { P_0, P_1, P_2, P_3 }. After processing by the code in Figure 12-15, the second vertex buffer would contain the five ordered points { I_{30}, P_0, P_1, P_2, I_{23} } that represent the clipped polygon.

Other approaches to polygon clipping have been published [Liang][Vatti].

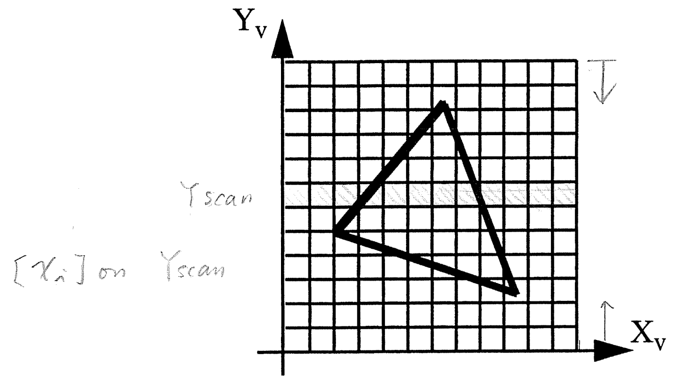
12.6 Scan Converting Polygons

Scan converting a 2D polygon whose vertices have been mapped to screen (pixel) coordinates involves computing the locations of all pixels whose centers lie inside or on the exact polygon boundary, commonly called *filling*. This can be done by scanning the polygon one infinite horizontal scan-line at a time ($y=Y_{scan}$), from top to bottom (or bottom to top).

The 2D scan converting algorithm consists of the following steps.

For each scan-line, Y_{scan}:

1. Compute the intersections of the scan-line with the edges of the polygon. A convex polygon will produce one pair of intersection points, while a concave polygon may produce a large number of pairs.

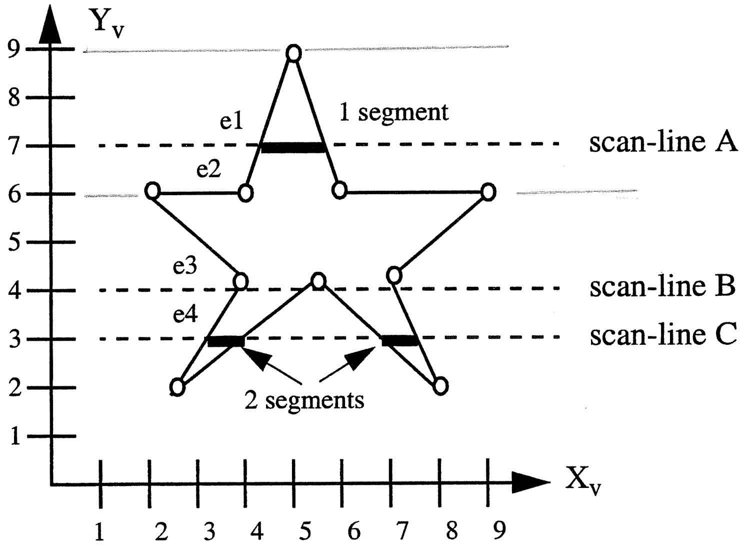


Grid locations are the coordinates of pixels. In some systems, this is the center of a pixel. In others, this may be a corner.

Figure 12-18. Pixel Grid and Polygon.

2. Sort these intersections by increasing x. Due to the arbitrary ordering of vertices around a polygon, the intersections will not generally be ordered left to right (increasing value).

3. Group the ordered intersections into pairs to form scan-line *segments* that lie inside or on the polygon. This is commonly called the "even-odd rule." For any closed object, as an infinite line crosses its boundaries, it toggles from outside to inside.



{X1, X2, X3, X4

Figure 12-19. Segments Formed by Intersections of Scan-Lines and a Polygon.

4. Rasterize each of the segments by "marching" horizontally along the segment from (X_{start}, Y_{scan}) to (X_{end}, Y_{scan}) assigning pixel values at pixel locations, i.e. grid coordinates.

Exact segment x coordinates must be rounded to the nearest integer pixel coordinates. Care must be taken to round properly. In general, the start (left) segment x should be rounded to the next larger integer value, and the end (right) segment x should be rounded to the next smaller integer value, as shown in Figure 12-20. The library routine CEIL(X) provides the rounding up function and FLOOR(X) provides the rounding down function.

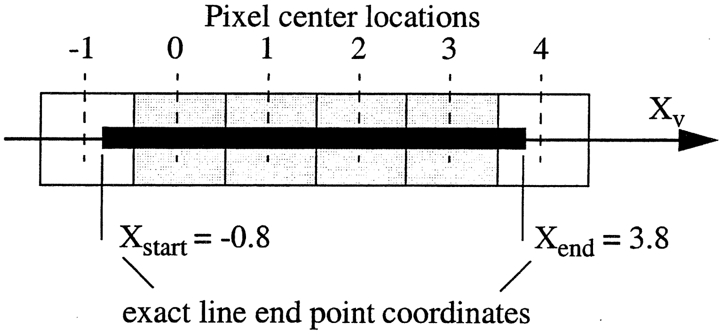


Figure 12-20. Segment Coordinate Rounding for Rasterization.

12.6.1 Vertex Intersections

It is possible for a scan-line to intersect a vertex if the vertex y coordinate is on a scan-line Y_{scan}. This will always happen for integer coordinates, and can happen even if real coordinates are used. One way to handle vertex intersections is to examine the previous and next vertices to determine if the vertex is a local minimum or maximum. If so, count the vertex intersection as two intersections with the same x coordinate. Otherwise, count it as one. This is further complicated by horizontal edges on scan-lines.

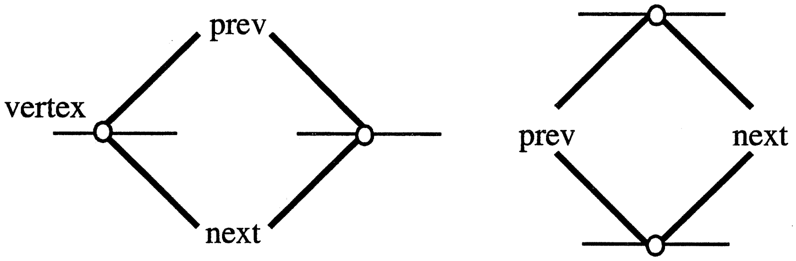


Figure 12-21. Vertex Intersections Along a Scan-Line.

Another approach is simply not to allow vertices to have y values on a scan-line by "adjusting" any offending y values, or all y values, prior to scan-converting the polygon. This requires that coordinates be stored as real (or rational integer) values. If a y value is on a scan-line, then add a small amount to its value to place it between scan-lines. The small amount should be large enough to avoid scan-line intersections but small enough to insure that the rounded values fall on the same scan-line. Some approaches simply add 0.5 to all rounded y coordinates. Because the scan-converting process finds pixels to the nearest integral x and y coordinate anyway, image quality will not suffer (for our purposes, anyway).

12.6.2 3D Scan Conversion

To provide pixel depths for z-buffer algorithms, the 2D scan-converting algorithm must be extended to include pixel depth computations. Now consider the scan-line as an edge-on view of an infinite plane: Y= Y_{scan}, as shown in Figure 12-22. This changes the scan-line - edge intersections from 2D to 3D, intersecting the plane Y= Y_{scan} with the 3D segment between two 3D vertices of the polygon. The result of each intersection calculation is now (X_i, Y_{scan}, Z_i), where X_i is the X coordinate as before, but Z_i is the Z value at the intersection. This list is sorted by X_i as before, and paired into segments.

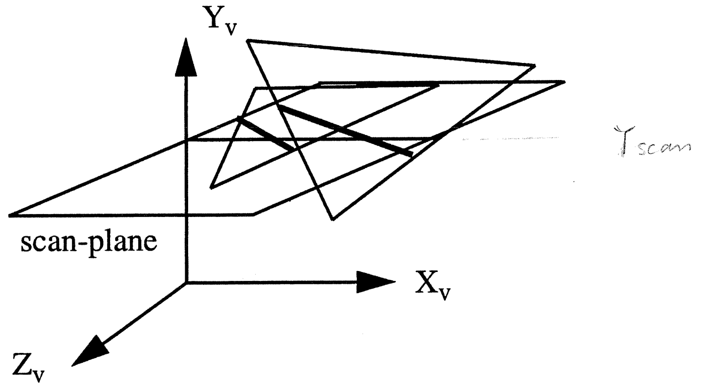
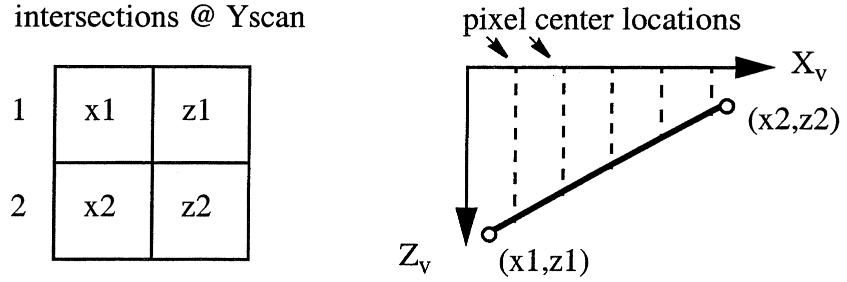


Figure 12-22. Segments and the Scan-Plane.

The segments are now 3D lines in the plane Y=Yscan, from (Xstart,Yscan,Zstart) to (Xend,Yscan,Zend). As the line is rasterized into integral X pixel addresses, the corresponding z depth at the rounded pixel x values must be computed (Figure 12-23). This z depth can be used in z-buffer comparisons.



intersections @ Yscan

1	x1	z1
2	x2	z2

Figure 12-23. Z Interpolation Along a Segment.

12.7 Scan-line Z Buffer Algorithm

Storing an entire screen of pixel values and pixel depths can be very costly and may not be necessary. Consider instead computing the image one scan-line at a time, and outputing each fully resolved scan-line to the output device in display order. The z-buffer depth comparisons will still be used to do hidden surface removal, but only along one scan-line. This is the scan-line z-buffer algorithm:

```
Map and clip polygons to $XYZ_v$
FOR each scan-line $Y_{scan}$ DO
        Clear the scan-line;
        FOR each polygon $P_i$ DO
                Compute sorted intersection segments $S_j$ of $P_i$ on $Y_{scan}$;
                FOR each segment $S_j$ DO
                        Rasterize $S_j$ into pixel values and pixel depths;
                        Compare and replace pixels in z-buffer with pixels from $S_j$;
                ENDDO
        ENDDO
        Draw the scan-line;
```

"Clear the scan-line" means to set all pixels to "BACKGROUND" and all depths to "FAR AWAY."

12.8 References

1. Sutherland, I. E. and Hodgman, G. W., "Reentrant Polygon Clipping," Communications of the ACM, Vol. 17, No. 1, 1974.
2. Liang, Y. and Barsky, B. A., "An Analysis and Algorithm for Polygon Clipping," Communications of the ACM, Vol. 26, No. 11, November, 1983.
3. Vatti, B. R., "A Generic Solution to Polygon Clipping," Communications of the ACM, Vol. 35, No. 7, July, 1992.

12.9 Review Questions

1. What is the "3D screen space" ? Be specific.
2. Explain the process of back-plane rejection. For what type of data is back-plane rejection valid? For what type of data will backplane rejection completely remove hidden lines?
3. Give a description, in step-by-step outline form, of the two hidden surface (polygons) algorithms named below. Provide sufficient detail to clearly show the flow of the algorithm, and to describe what needs to be done at each step:

   3A. the "Watkins Scan-line" algorithm,

   3B. the "Scan-line Z-Buffer" algorithm.
4. Given an ordered list of polygon vertices with an unspecified normal direction, describe how to compute if this polygon is convex or concave (non-convex).
5. Describe the contents and storage needs for the "z-buffer" for (1) the Z-Buffer HSR algorithm, (2) the Scan-line Z-Buffer algorithm.
6. Given an eye coordinate system with the COP at the origin and image plane at z = -d, what are the projected z coordinates? Is this projection useful for HLR and HSR?

Chapter 13. Realistic Display Effects

During the discussion of hidden surface removal, we often referred to the “polygon’s color” and other operations for storing the “pixel value” for the visible pixel belonging to an object. Suppose we have prepared 3D data consisting of a set of polygons covering a sphere, transformed the data to the screen space, and have performed HSR. We have assigned the color "gray" to each polygon in the data and the pixel values were stored in pixel memory holding the final image. Unfortunately, such an image would appear as a flat gray circle (with straight sides), without perceivable 3D depth characteristics, as illustrated in Figure 13-1. The problem is that we have ignored color variations due to lights, geometry and material.

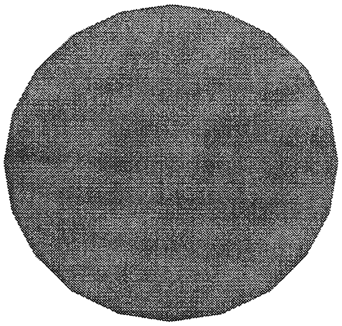


Figure 13-1. HSR Rendered Image.

To create a *realistic* image of 3D objects requires more than just hidden surface determination. It also requires simulation of the interaction of light with the geometry and surface properties of the objects that determines the light that reaches our eyes. Recalling the discussion of color, light is visible radiation that interacts with surfaces in complex ways. Incident light from many sources irradiates the surface of an object of a certain material. Light is absorbed by, reflected from, and transmitted through, the surface. The behavior is determined by the geometry of the surface and its material properties, including its color.

The terms *shading model*, *illumination model*, *lighting model* and *reflectance model* are used to describe models for the interaction of light and geometric objects. For our purposes, the illumination model is the mathematical model used to compute the color of a pixel corresponding to a visible point on the surface of an object. A shading model combines an illumination model with methods for varying the color across the surface of objects in an efficient manner.

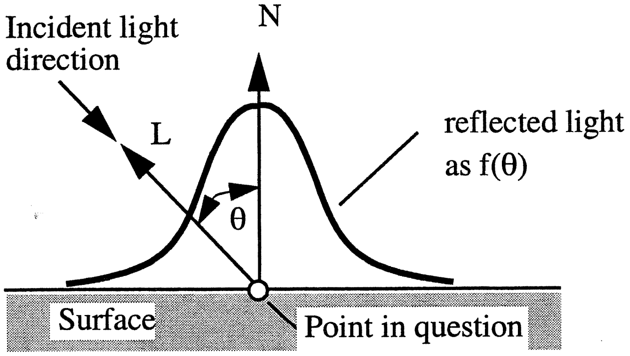
13.1 Illumination Models

Illumination models are based on the physical behavior of light and materials and involves complex energy transfer phenomena that have been the subject of study for hundreds of years. Early illumination models attempted to capture the gross behavioral characteristics of light and materials to produce computer generated images that would appear realistic to the eye. These models were somewhat perceptually based, in that they were developed empirically by experimentation to produce visually pleasing results using simple mathematical formulae to model mathematically undescribable physical behavior .

We begin by considering the fundamental components of a common illumination model, diffuse, specular, and ambient reflection. For the time being, consider the incident light to be achromatic light that can be characterized by an intensity. Later we will consider more complex models for light.

13.1.1 Diffuse Reflection

Micro-facets, microscopic plates on the surface of an object, reflect the incident light in all directions. For objects with "dull or matte surfaces," the micro-facets on the surface create a distribution of reflected light that is concentrated in the direction of the surface normal, as shown in Figure 13-2. This is characteristic of "*diffuse reflectance*." The intensity of the reflected diffuse light can be approximated by the cosine curve shown in Figure 13-2, where L is the unit direction vector opposite to the direction of incident light, N is the unit surface normal vector, and θ is the



$\cos\theta = \hat{L} \cdot \hat{N}$

Figure 13-2. Diffuse Reflectance Notation.

angle between L and N. We know that $\cos\theta = L \bullet N$ if L and N are unit vectors.

The intensity of reflected light for diffuse reflection is

$$I = K_d \cos(\theta),$$

whereK_d is a surface property representing how much incident light is reflected diffusely from a given surface.

13.1.2 Specular Reflection

For polished surfaces, bright spots, or *highlights*, appear when the light source, surface normal, and the eye are in the right positions to capture some of the *specular reflection.* Looking at an ideal reflector, i.e., a mirror, a ray of incident light would be reflected only along R_L, which would be seen only if the eye were located exactly as shown in Figure 13-3. If the surface is not a perfect reflector, as most are not, then the reflected light is scattered around R_L and the intensity diminishes as the eye moves away from the direction of perfect reflection, R_L.

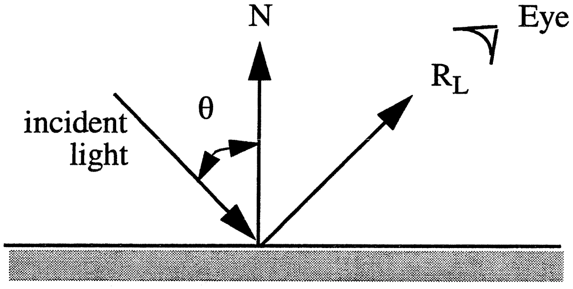


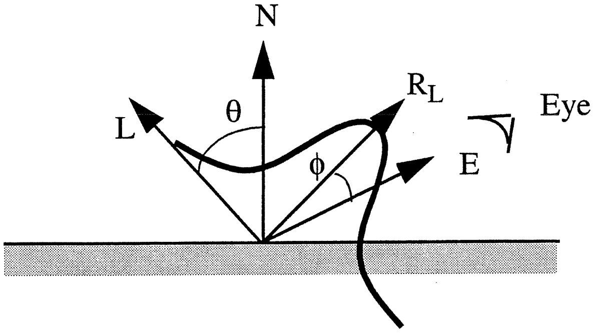
Figure 13-3. Specular Light Reflection.

There are two common approaches to simulating specular reflection. The first, and least commonly used today, involves the angle between the ideal reflection direction, R_L, and the vector to the eye, E, as shown in Figure 13-4.

The specular component of light is

$$K_s \cos^n\phi,$$

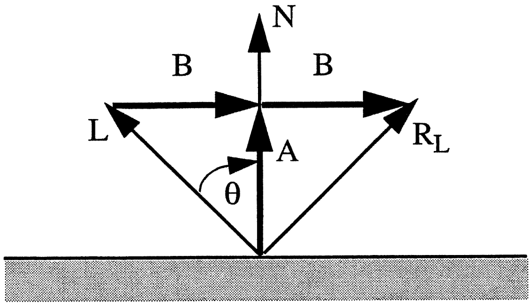
where K_s is a surface property representing the amount of incident light specularly reflected from a surface, n is the specular exponent, a surface property representing the distribution of specularly reflected light about the reflection vector R_L (values usually are 2 to 100), and ϕ is the angle



$\cos \phi = R_L \cdot E$

Figure 13-4. Specular Reflection Vectors.

between R_L and E. In this case, it is computed using $\cos\phi = R_L \bullet E$ if R_L and E are unit vectors. Given L and N, the vector R_L is computed by sweeping L through N by an angle θ.



$$A = (N \bullet L) N$$

$$B = A - L = (N \bullet L) N - L$$

$$R_L = A + B = 2A - L$$

$$\boxed{R_L = 2(N \bullet L) N - L}$$

Figure 13-5. Reflection Vector R_L Computations.

Note that it is possible to have $R_L \bullet E < 0$, which cannot be allowed. This means $\cos(\phi) < 0$, so $\phi >$ 90°, which makes it impossible for reflection. In this case, the specular reflection should be set to 0.

Instead of using the R_L vector for computing the specular component, a more popular approach is to define another vector, H, which is the unit bisector of L and E as shown in Figure 13-6. H is computed using vector operations:

$$H = \| E + L \|$$

assuming E and L are unit vectors.

This simplifies the computations considerably and eliminates the logical problems of $R_L \bullet E<0$.

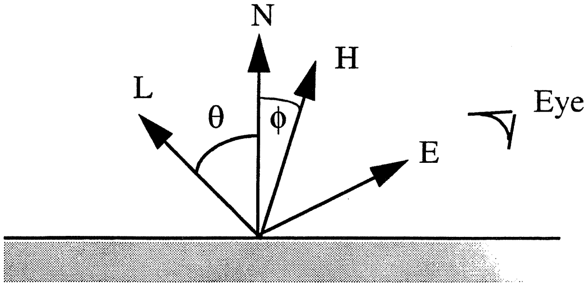


Figure 13-6. H Vector Computations.

If H and N are unit vectors, the $\cos \phi = H \bullet N$.

13.1.3 Ambient Light

$I = K_d \cos\theta + K_s \cos^n \phi$

In a real environment, light is transmitted from lights and reflected from objects to other objects in a very complex manner described by laws of physics. To account for this in a simple way, most lighting models include an ambient component, called "*ambient light*," that is considered the minimum intensity for any surface. Ambient light has no incoming direction and is constant across all surfaces, or per surface. Simply put, it compensates for complex lighting conditions that cannot be simulated (or are too time-consuming to simulate). The ambient light produces constant intensity surfaces, $K_a I_a$,

where K_a is a surface property representing how much surrounding ambient light a particular surface reflects, I_a is a "*global illumination property*", representing the amount of ambient light in the lighting environment surrounding the objects. In this case, it is the intensity of ambient light in the environment.

13.1.4 Ambient - Diffuse - Specular Model

Combining ambient, diffuse and specular reflection yields:

$$I = K_a I_a + K_d \cos\theta + K_s \cos^n \phi$$

or,

$$I = K_a I_a + K_d (L \bullet N) + K_s (H \bullet N)^n$$ ← achromatic.

This is a basic lighting model used in many computer graphics applications.

Typically, the values for the surface properties K_a, K_d and K_s are scaled so that:

0.1~0.5 0.5

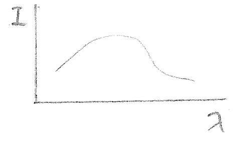
$K_a + K_d + K_s <= 1.0.$

Logically, this limits the total amount of reflected light to be no greater than the incident light. Also, if a surface's coefficients add to less than one, it will reflect less light than a surface with a larger sum.

To model the reflection of colored light, we again are faced with two modeling approaches: the physically-based simulation of light interacting with surfaces, or the early empirical solution that utilizes the tri-stimulus theory of color. The simulation approaches model colored light as a specular curve and represent the surface and material properties as specular functions instead of the constant coefficents that we have seen. The reflection models operate on the specular curves to produce the net reflected light as a specular curve. See [Hall] for a discussion of this process.

Based on the tri-stimulus theory of color, light is modeled as intensities of primary component colors, a trichromatic vector, typically RGB. Therefore, the incident light, I, in the previous equations changes from an intensity to a vector,

$$I = \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$



Likewise, the surface model coefficients become coefficient vectors,

$$\vec{K}_a = \begin{bmatrix} K_{a,\,red} \\ K_{a,\,green} \\ K_{a,\,blue} \end{bmatrix} \quad \vec{K}_d = \begin{bmatrix} K_{d,\,red} \\ K_{d,\,green} \\ K_{d,\,blue} \end{bmatrix} \quad \vec{K}_s = \begin{bmatrix} K_{s,\,red} \\ K_{s,\,green} \\ K_{s,\,blue} \end{bmatrix}$$

I_L =

R = 0 | = 0

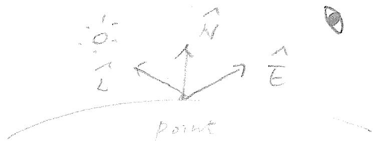
G = 0 0 = 0

B = 1 0 = 0

The full color lighting model is the vector equation:

$$\Rightarrow \quad \vec{I} = \vec{K}_a \vec{I}_a \oplus \vec{K}_d (\vec{L} \bullet \vec{N}) \oplus \vec{K}_s (\vec{H} \bullet \vec{N})^n$$ [Ka, Kd, Ks, n]

The products of vectors, for example $K_a I_a$ above, are computed by multiplying corresponding array elements to produce a vector as the result. Similarly, the product $K_d (L \bullet N)$ above is a vector multiplied by a scalar, producing a vector as a result.



$$\hat{H} = \| \hat{E} + \hat{L} \|$$

$$\begin{bmatrix} 2 \\ 6 \\ 12 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

13.1.5 Light Sources

$$I = K_a I_a + K_a I_{L_i}(L+N) + K_s I_{L_i}(L+N)^n$$

The illumination model operates on two sources of light: ambient light and incident light. Ambient light has no direction, and is a user-defined "fudge-factor" to account for inter-surface reflections. Incident light, however, has a direction that depends upon the type of light source. There are a number of different models for different types of light sources. Each light source model contains a color of the source light, position and direction information, and parameters which control the behavior of the light. The color is either an achromatic intensity (typically assumed to be white light) or a trichromatic vector.

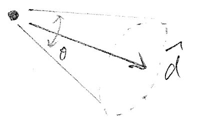
The simplest and most often used is the "*point light source*", like a light bulb. A point light source is modeled as a location and color (for direct color models). The incident light direction is computed using the world coordinates of the point in question and the light position:

$$L = \| P_L - P \|$$

where P is the point in question and P_L is the position of the point light source.

Another useful light source is "*parallel light*", such as sunlight. Parallel light is represented by a direction vector. The light direction vector, L, is simply the negation of this vector, and is the same for all points on all objects.

The "*spotlight source*" is representative of the many other types of light source models. Typically, a spotlight source is represented by a position, direction, cone angle and attentuation function. The direction of incident light is computed similar to the point light source, using the spotlight position and the point in question. The intensity of the incident light, however, is varied according to the geometry of the spotlight direction, spotlight cone angle and the incident light direction using the attenuation function. These computations are similar in nature to the computations for diffuse reflection, where the angle between the incident light direction and spotlight direction determines the factor applied to the spotlight color to determine the incident light intensity. The light is a maximum along the spotlight direction, falling off to a minimum value at the spotlight cone angle.



13.1.6 Multiple Light Sources

To model a lighting environment with more than one light source, one executes the illumination model for each light individually and sums the results. It is usually necessary to "clamp" intensities to a maximum value, such as unity, during this summation. During lighting computations, when a light intensity exceeds unity, it is set ("clamped") to unity.

13.2 Polygon Shading Methods

13.2.1 Flat Shading

For scan-line and z-buffer hidden surface processing of polyhedral objects, i.e., objects described by planar polygons, a "single" shade (color and intensity) can be computed for each polygon using an appropriate lighting model. Typically, the light source is considered to be sufficiently far from the polygon's surface so that the rays are essentially parallel. Therefore, any point on the polygon can be used to compute the L and E vectors and all pixels for this polygon have the same value. This is called flat shading.

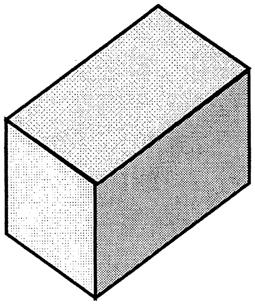
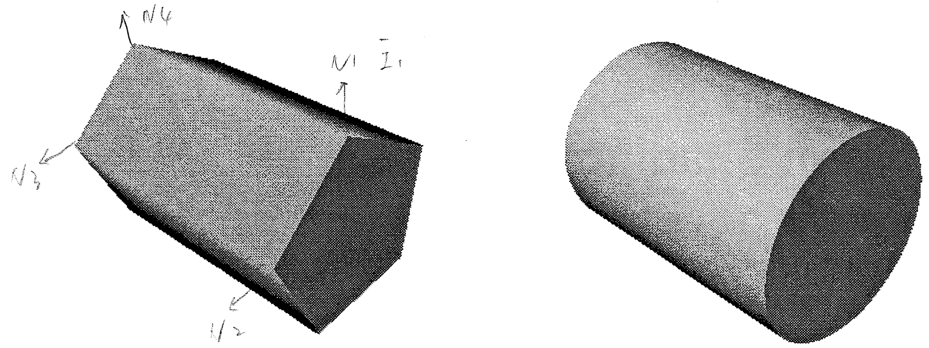


Figure 13-7. Flat Shaded Object.

13.2.2 Smooth Shading

Rendering shaded images of objects with curved surfaces is often done by approximating the surfaces by some number of planar polygons. This is called *tessellation* and the polygons are called *facets*. This presents special problems to rendering methods that use flat shading. The human eye easily discerns small changes in intensities, so one easily sees “that is an image of a sphere made of many flat polygons.”

Henri Gouraud developed a shading model for rendering tessellated objects by interpolating the intensities across the boundaries of the polygons. *Gouraud shading*, or *intensity* *interpolation*, involves computing intensities at the vertices of polygons. An example of smooth shading is shown in Figure 13-8. If the normals at the vertices can be computed from surface equations, such as the equations of a cylinder as shown in Figure 13-8, then these normals are stored at the vertices and used for the lighting computations. When the surface equations are not known, as is the case in systems that approximate all surfaces with planar polygons, then the shared vertices where facets from the same surface meet, the *geometric normals* are averaged to form a *lighting normal* that will be used for intensity computations. The averaged normal at a shared vertex is the vector average of the plane normals of all polygons sharing the vertex. In Figure 13-9, for example,



Flat shaded cylinder

Smooth shaded cylinder

Figure 13-8. Examples of Smooth Flat and Smooth-Shaded Cylinders.

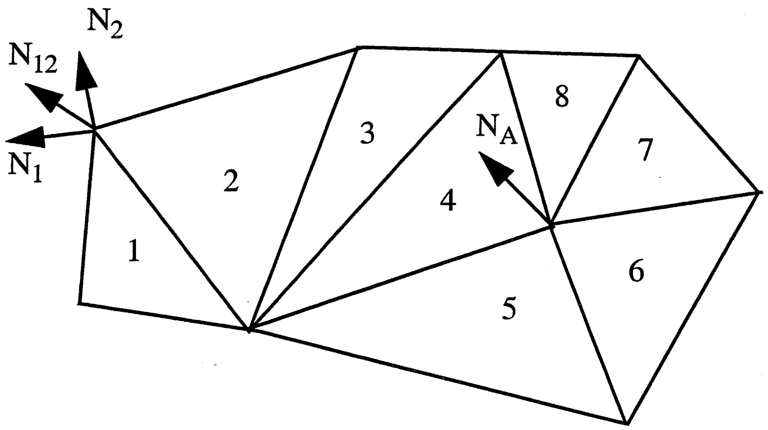


Figure 13-9. Normal Averaging for Smooth Shading.

$$N_{12} = \left\| \, \| N_1 \| + \| N_2 \| \, \right\| \text{ and } N_A = \left\| \sum_{i=4}^{8} \| N_i \| \right\| .$$

Scan-line computations must now linearly interpolate the color of pixels based on the intensity or trichromatic color components across a scan-line segment and along the edges from one scan-line to the next. For example, in Figure 13-10, the colors of pixels along the indicated

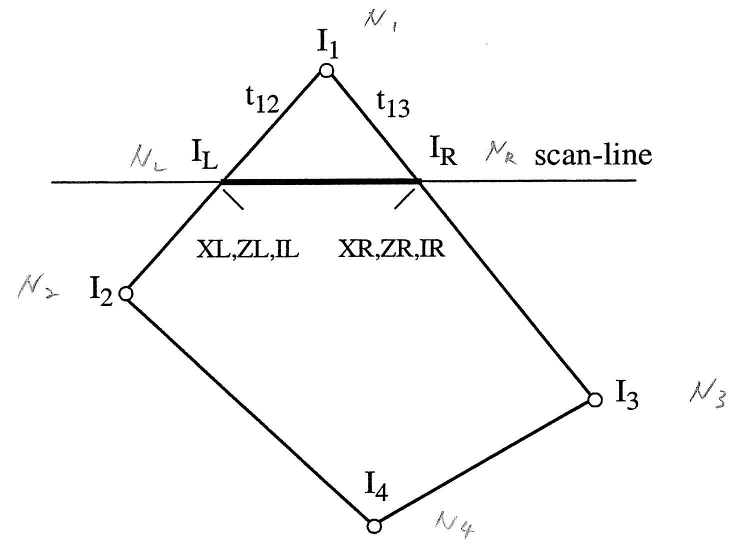


Figure 13-10. Intensity Interpolation While Filling a Polygon.

scan-line is a linear interpolation of the colors I_L and I_R, which are themselves linear interpolations along the edges I_1I_2 and I_1I_3, respectively:

$$I_L = I_1 + t_{12} (I_2 - I_1) \text{ and } I_R = I_1 + t_{13} (I_3 - I_1),$$

where t_{12} and t_{13} are parameters that vary from 0 to 1 along each edge. The intensity across segment I_L-I_R is interpolated with a parameter s that varies from pixel to pixel across the scan-line:

$$I_s = I_L + s (I_R - I_L).$$

Unfortunately, Gouraud shading produces noticeable *mach bands* of intensity patterns. These are due to discontinuities in the intensity variations across polygon boundaries.

B. T. Phong suggested averaging the normal itself, instead of the intensity. This reduces the bands produced by Gouraud shading. Notice the nonlinear relation between the intensity I and the normal N in the lighting model equation. Phong shading, or normal interpolation, produces smoother images, but requires more complicated per-pixel computations that must be done in world space. The full lighting model must be computed for each pixel.

13.3 Ray Tracing

Ray tracing (also called ray casting) is a versatile and elegant method for rendering images with realistic display effects. The eyes integrate the light that falls on the retina from all directions to form the image we see. This is difficult to compute, so instead consider a single ray of light passing out of the eye through each pixel, asking the question “What contributed to light along this ray that reached the eye through this pixel?”

The idea is to “trace” or “cast” lines from the eye position through each pixel on the image plane into space. Whatever is “seen” along the ray is displayed as the pixel color on the raster screen.

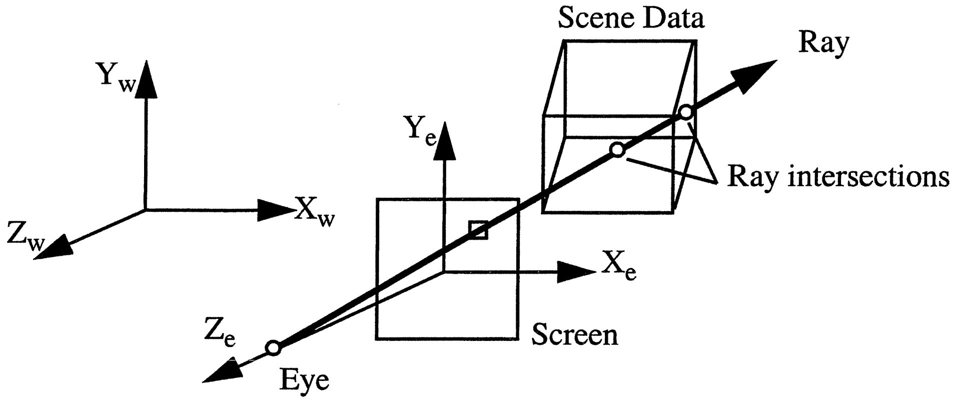


Figure 13-11. Ray Tracing.

Ray tracing requires the computation of the intersections of a ray with the geometry describing the scene, typically planes, but also spheres, cylinders, and other geometric forms. For each ray, all intersections with all objects are computed. Line of sight visibility is determined along a ray easily: the ray intersection nearest to the eye is the visible point, called the "visible hit."

When the ray is modeled as a parametric vector equation of a line, such as P + t V for t Š 0, the ray parameter ′t′ is the distance measure along the ray that determines the nearest hit. The ray tracing process is basically simple:

```
Procedure RayTrace:
FOR each pixel on the screen in world coordinates DO
    Compute the vector ray equation through the pixel;
    Set the "visible_hit" parameter to "none";
    FOR each object in the scene DO
        Compute the ray intersection(s) with the object;
        IF an intersection is nearer than visible_hit THEN
            set visible_hit to new intersection parameters;
        ENDIF
    ENDFOR
    IF there was a visible_hit THEN
        Set the pixel color using the illumination model;
    ELSE
        Set the pixel color to the background color;
    ENDIF
ENDFOR
```

The ray tracing process simplifies and enhances the rendering process considerably:

1. There is no need to transform the geometry into screen coordinates because rays can be cast just as easily in world coordinates.
2. Projection is a by-product of the process because each ray is a projector. If the rays eminate from the eye point, then the resulting image will have perspective projection. If the rays are all parallel and perpendicular to the screen, the image will appear to be orthographically projected.
3. Clipping is not needed because rays are cast only through pixels on the screen. In addition, intersections with geometry at or behind the eye, i.e. with zero or negative ray parameters, are easily ignored as invisible.
4. The world coordinates of the intersection point and the surface normal at this point are available for illumination model computations and more sophisticated rendering effects.
5. More complex geometry than simple polygonscan be used because the computations only require the intersection with a ray.

Before we abandon scan-line and z-buffer rendering altogether, however, there is a significant disadvantage to ray tracing: it is many times more computationally expensive. Ray tracing is elegant, but very time-consuming, compared to scan-line methods. Indeed, much of the research in ray tracing is directed towards improving the computation times.

13.3.1 Shadows

One rendering effect that adds considerable realism to an image is shadows. Shadows occur when light from a light source is obscured from portions of the surfaces of objects by other objects. We have simply ignored this phenomenon in the development so far. Shadows modify the behavior of the illumination model.

Ray tracing facilitates shadow computations. After the visible hit point, called Q from now on, has been computed from the ray eminating from the eye, another ray is cast from the visible hit toward each light source. If this *shadow ray* intersects any object before reaching the light (see Figure 13-12), then this light source cannot contribute any light reflection toward the eye for this pixel. In a ray tracing program, the shadow ray is "just another ray" to be intersected with the geometry of the scene. It is possible to have the same function process the initial ray and the shadow ray.This contributes to the elegance of the process.

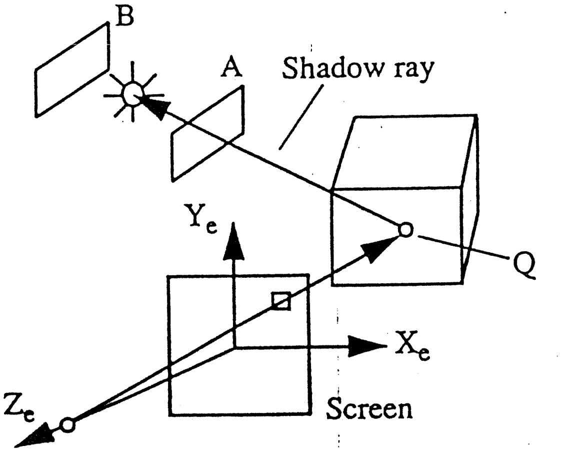


Figure 13-12. Shadow Ray Vectors.

The shadow ray, however, must be processed differently than the initial ray from the eye. The ray parameter for the initial ray is only used for relative distance comparisons, so its magnitude is generally unimportant. It is a true ray with a semi-infinite parameter range ($t > 0$). The magnitude of the "V" direction vector in the initial ray equation is not important. For the shadow ray, however, it is necessary to know if any object intersects the ray between Q and the light. This means that the shadow ray is really a segment, with upper and lower parameter bounds. Typically the shadow ray equation is:

$P(t) = Q + t\,(P_L - Q)$ for $0 < t <= 1$.

If the visible hit point is determined to be shadowed from a light source, then only ambient light reflects from the point. Therefore, the illumination model is truncated to the ambient portion for this light for this point:

$I = K_a\, I_a$.

13.3.2 Reflection and Refraction

The illumination model has considered only light reflected directly from lights to objects to the eye. In a real environment, however, light reflects from surfaces to objects as well. Such *global illumination* computations add to realism at increased computational cost.

After computing the initial ray intersection, Q, one can ask the question: "What light could have reached the eye as a result of reflection from the surface at Q?" The question is answered by casting a ray in a direction R corresponding to ideal reflection with the direction E to the eye (Figure 13-13). This ray simulates *reflection* sources and becomes a separate incident light path to trace. The direction R is found with vector computations similar to the specular reflection vector R_L discussed in section 13.1.2:

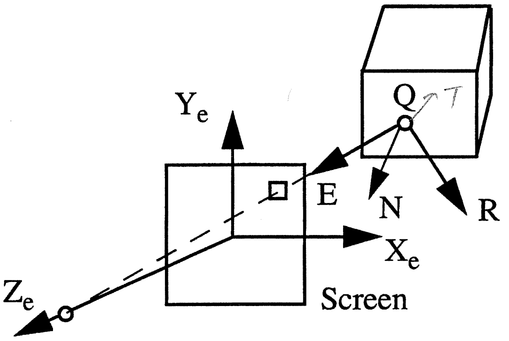


Figure 13-13. Reflection Ray Vectors.

$R = 2\,(E \bullet N)\,N - E$.

If the object is *transparent*, that is, light can be transmitted through it, a ray can be cast through the surface to simulate the *refracted* light reaching the pixel. The direction of the transmitted ray, or refraction ray, is determined by Snell's law of refraction that relates θ_t, the angle between the transmitted ray T and the normal N, and n_t, the coefficient of refraction of the transmitted medium, to θ_i, the angle of the incident ray and n_i, the coefficient of refraction of the incident medium: $n_i \sin \theta_i = n_t \sin \theta_t$ (Figure 13-14).

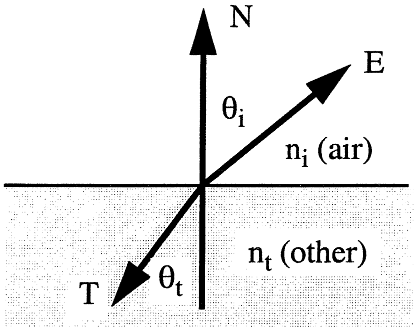


Figure 13-14. Vectors for a Transmitted Light Ray.

Each reflected and transmitted ray is further processed as another incident ray, in turn spawning its own reflections and refractions. The result is a ray producing process that creates a binary tree of rays, all to produce the color for one pixel. The resulting data structure resembles a tree of intersection data nodes (Figure 13-15). Once the tree has been computed, the resulting

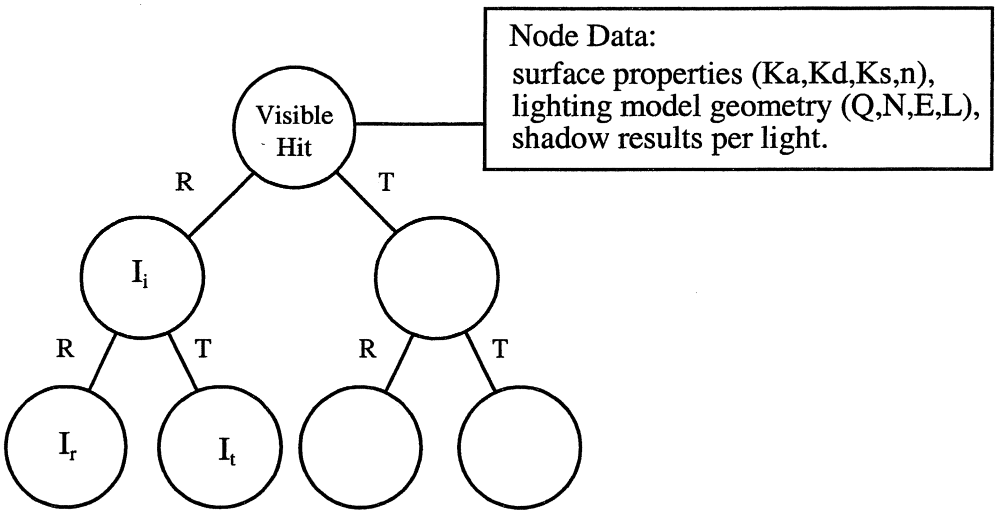

Node Data:
surface properties (Ka,Kd,Ks,n),
lighting model geometry (Q,N,E,L),
shadow results per light.
Visible Hit
R
T
I_i
R
T
R
T
I_r
I_t

Figure 13-15. Ray Tree for a Single Ray.

intensity (color) must be computed for this pixel. This is done, typically using a recursive procedure, for each data node (bottom up), using a formula of the form:

$$I_{node} = K_i I_i + K_r I_r + K_t I_t$$

where:

I_i is the intensity computed with the node's data for the incident ray and the light sources using an illumination model,

I_r is the intensity from the reflection ray branch of the tree,

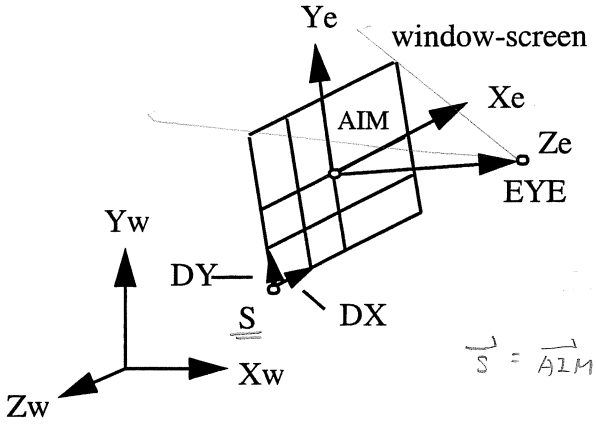
I_t is the intensity from the transmission ray branch of the tree,

K_i, K_r, K_t are the surface properties that determine the contribution of each ray (and sum less than or equal to 1.0).

In theory, parallel, perfectly reflecting objects can create an infinite number of rays. In practice, the surface properties are used during the ray casting process to attenuate (lessen) the intensity of each ray at each intersection. When the ray intensity of one of the rays (reflection or refraction) decreases below a certain limit, no new rays are cast. In addition, some ray tracing programs simply stop casting rays after a certain tree depth is reached.

13.4 Ray Geometry

Ray tracing is performed in world space. The problem becomes finding the pixel locations in world coordinates (Figure 13-16). Let S be the position in XYZ_w of the lower left corner of the window. The viewport (screen) pixels must be mapped onto a grid in this window. S then becomes the starting pixel location. (The other corners could do as well, of course.)



$\vec{S} = \vec{AIM} - WSX\ \hat{Xe} - WSY\ \hat{Ye}$

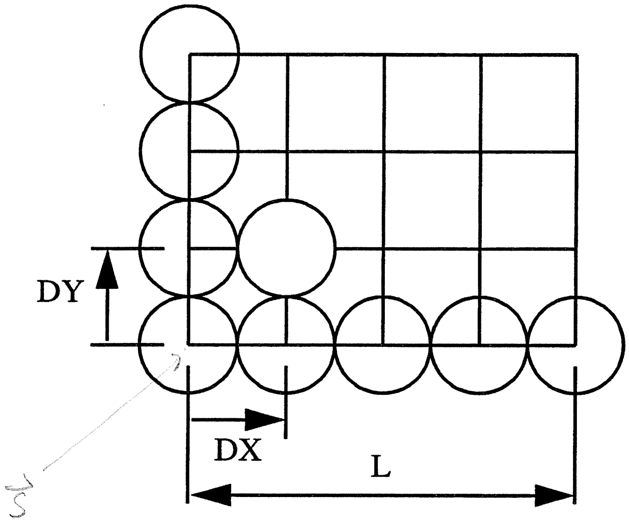
Figure 13-16. Vectors for Ray Geometry.

Let the unit direction vectors of the eye coordinate system in XYZ_w be called x_e, y_e, z_e.

Given these, AIM, and wsx and wsy (the window half-sizes), S is found using vector operations in the image plane in XYZ_w space.

$S = AIM - wsx\ x_e - wsy\ y_e$

Let DX be the world vector between the centers of two adjacent pixels on the same scan-line. That is, DX is a "horizontal step." Given wsx, wsy and the viewport resolutions (number of pixels across and down the screen), DX can be found as shown in Figure 13-17. With S, DX and DY, it is possible to "march across the screen" pixel by pixel, tracing rays at each location.



$$\vec{DX} = \frac{L}{(Res_x - 1)} = \frac{2wsx}{(Res_x - 1)}\hat{x}_e$$

$$\vec{DY} = \frac{2wsy}{(Res_y - 1)}\hat{y}_e$$

$p(t) = COP + (P_i - COP)t$

Figure 13-17. DX and DY Vector Computations.

The ray equation is $P(t) = P + t\ V$.

For perspective projection, all rays start at the eye, so P = EYE. Each ray passes through a pixel, so these are the two points on the line. The direction vector $V = (P_i - EYE)$ where P_i is the given pixel location.

For orthographic projection, P is formed with the same [x,y] coordinates as P_i and a z coordinate that is appropriate for clipping points behind the eye. In other words, select the z coordinate of P such that ray intersections with the ray parameter $t \leq 0$ are invisible. All rays are parallel and perpendicular to the screen, so $V = -K = [0,0,-1,0]$, a constant.

Advancing to the next pixel on a scan-line is a computation using the vector DX:

$P_{i+1} = P_i + DX \qquad \text{for } i = 0..Res_x\text{-}1$

Advancing to the next scan-line can be done using one of two methods. Given that P_i is at the last pixel on a scan-line, advance to the next by returning horizontally to the first pixel location, then moving up to the next scan-line:

$$P_{i+1} = (P_i - 2 \text{ wsx } x_e) + DY$$

Another approach is to save the location of the first pixel on the current scan-line, thereby avoiding the computations above. Call this S_i. Advance to the scan-line now simply by computing:

$$P_{i+1} = S_i + DY$$

$$S_i = P_{i+1}$$

13.5 Ray-Polygon Intersections

Given a ray, P(t) = P + t V, and a planar polygonal surface through a point P_0 and a normal N satisfying $(P - P_0) \bullet N = 0$, we first compute Q, the point of intersection of the ray and the infinite plane. Next we must determine if Q lies inside the bounded surface, a polygon in 3D space. It is assumed that the boundary consists of an ordered set of straight line segments between vertices. This procedure is commonly called the *point containment test*, or the *inside/outside test*.

Typically, the plane vertices are stored in a list data structure of some kind, ordered consistently with the given normal. For example, “the vertices traverse the boundary in a CCW orientation when viewed down the normal,” or something similar.

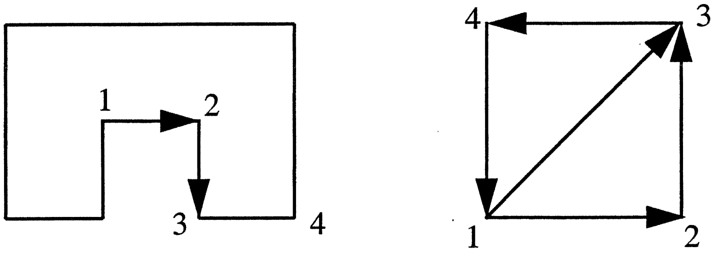


Figure 13-18. Concave and Convex Polygons and Vertices Forming Their Boundaries.

First consider point containment for convex polygons.

13.5.1 Point Containment for Convex Polygons

If the polygon in question is convex, then its geometric properties can be used to simplify and speed the point containment test. The interior region of convex polygons can be described as the *intersection of half-spaces* formed by the edges (sides). This can be expressed as "if a point is inside each half-space formed by the infinite extension of each edge then the point is inside the polygon." For efficiency, invert the rule to "if a point is outside any half-space (etc.), then the point is outside the polygon."

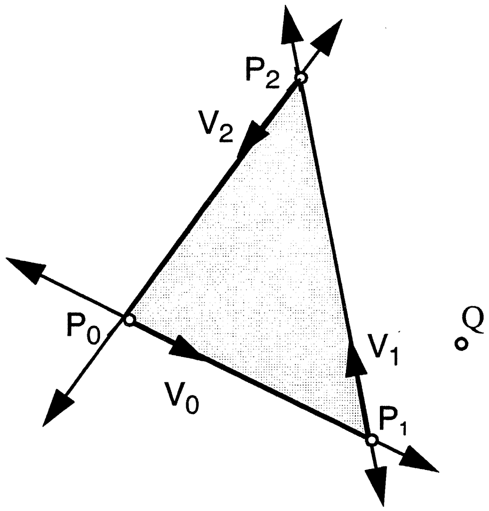


Figure 13-19. Convex Polygon as the Intersection of Half-Spaces Formed by Lines.

This is the basis for the "*quick-rejection*" algorithm shown in Figure 13-20.

```
FOR   each edge $P_i$-$V_i$   DO
      if [ ( Q - $P_i$ ) ⊗ $V_i$ ] * N > 0   THEN
            RETURN "OUTSIDE";
END
RETURN "INSIDE";
```

Figure 13-20. Point Containment Algorithm for Convex Polygons.

13.5.2 Semi-Infinite Ray Point Containment Test

One technique for determining point containment for non-convex polygons is the semi-infinite ray test. Cast a ray in the plane from Q in any convenient direction and count the intersections with the boundary segments. This process is time-consuming and is complicated by intersections with vertices. The algorithm in Figure 13-22 simply restarts if the ray intersects a vertex.

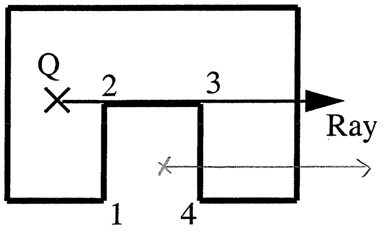


Figure 13-21. Semi-Infinite Ray for Point Containment Tests.

```
RESTART:
      Set random ray direction in plane;
      count = 0;
      FOR each edge of polygon DO
            IF ray intersects edge THEN
                  IF intersection is at a vertex THEN
                        GO TO RESTART;
                  ENDIF
                  count = count + 1;
            ENDIF
      ENDDO
      IF count IS ODD THEN
            RETURN "INSIDE"
      RETURN "OUTSIDE"
```

Figure 13-22. Semi-Infinite Ray Point Containment Algorithm.

13.5.3 Sum of Angles Point Containment Test

Another somewhat more efficient technique for non-convex polygons is the sum of angles method. This is based on the property that if a point is inside a closed polygon, the sum of the angles between vectors from the point to successive pairs of ordered vertices will be ±360 degrees ($\pm 2\pi$).

Using V_i to mean the vector from Q to P_i, note that:

$$\begin{aligned} (V_i \otimes V_{i+1}) \bullet N &= |V_i|\ |V_{i+1}|\ \sin\theta \\ V_i \bullet V_{i+1} &= |V_i|\ |V_{i+1}|\ \cos\theta \end{aligned}$$

$$\text{so, } \tan\theta = \frac{\sin\theta}{\cos\theta} = \frac{(V_i \otimes V_{i+1}) \bullet N}{V_i \bullet V_{i+1}}$$

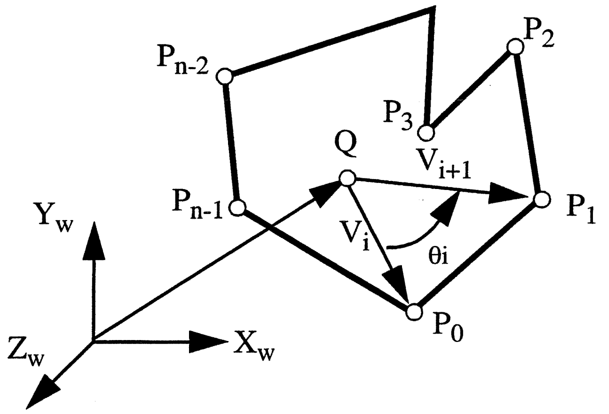


Figure 13-23. Sum of Angles Algorithm Notation.

```
$\theta_{total} = 0$
FOR each vertex $P_i$ of the polygon DO /* including $P_{n-1}$ to $P_0$! */
      $V_i = P_i - Q$
      $V_{i+1} = P_{i+1} - Q$
      sine =  $(V_i$ x $V_{i+1}) \bullet N$
      cosine = $V_i \bullet V_{i+1}$
      IF  |sine| ≈ 0 AND |cosine|≈0 THEN  "on vertex: so RETURN INSIDE"
      $\theta$ = ATAN2(sine,cosine)
      IF $\theta \approx \pi$ THEN "on edge: so RETURN INSIDE"
      $\theta_{total} = \theta_{total} + \theta$
ENDDO
IF $|\theta_{total}| \approx 2\pi$ THEN
      RETURN INSIDE
ELSE
      RETURN OUTSIDE
ENDIF
```

Figure 13-24. Sum of Angles Point Containment Algorithm.

13.6 Orienting Lighting Normals skip

Depending on the form of the data representing objects, it may be necessary to determine which side of a surface is "visible to the observer." Similar to the discussion for back plane rejection, the data may be closed or open. Closed data means that it is not possible to see the back of a surface, so surface normals are "oriented" and can be used directly as lighting normals in the illumination model.

Open data may be oriented or not, meaning that surface normals may be used to determine the front or back of a surface, or not. For example, for a single plane, as the eye position is rotated around the data, the plane normal will sometimes face the observer and other times face away from the observer, determined by the result of E • N.

For un-oriented data, the proper surface normal must selected for the lighting normal when the viewing position changes. In effect, this makes each surface a thin solid. Consider an edge-on view of the visible surface at the point of ray intersection, Q, as shown in Figure 13-25. Consider the different combinations of the two different eye positions (labelled "Eye a" and "Eye b"), two light positions ("Light a" and "Light b") and two different surface normals ("Na" and "Nb"). The objective is to select the normal, N_a or N_b, to give the proper lighting effect. There are 4 cases to consider depending on the eye and light locations. Select a lighting normal N_l from the surface normal, N, as follows:

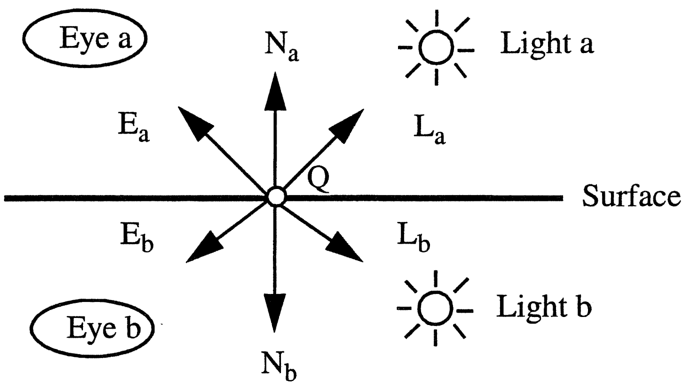


Figure 13-25. Possible Normals for Different Eye Positions for a Surface.

```
N_l = "current surface normal"
IF( E • N < 0 ) THEN
     "flip lighting normal: N_l = - N_l"
ENDIF
IF( L • N_l < 0 ) THEN
     "eye hidden from light: use ambient light only"
ELSE
     "light is visible to eye: use the full lighting model"
ENDIF
```

Figure 13-26. Lighting Normal Algorithm.

13.7 Intersecting a Ray with a Convex Polyhedron

The interior of a convex polyhedron (a volume whose bounding faces are convex polygons) can be described as the "intersection of half-spaces" bounded by the infinite planes formed by each of the faces. Similar to the point containment test for convex polygons, the algorithm for determining if a point is on a convex polyhedron involves testing on which side of each face the point in question lies. $P \cdot B$

Given a ray, P(t) = P + t V, one approach to finding the points of intersection is:

1. Compute the set $\{ t_i \}$ of intersections with the infinite planes of the object. For a parallelepiped, i = 0..5; for a tetrahedron, i = 0..3.

2. Classify each point $\{ Q_i \} = \{ P + t_i V \}$ by forming the homogeneous dot product with the other planes B_j, i.e., $Q_i \bullet B_j$, where B_j is a homogeneous plane vector $[A, B, C, D]^T$, and B_j has been formed such that [A,B,C] is the right-hand outward normal of the face. The result of the dot product will classify the point: e.g. $[1, 0, 0, -1]$

$Q_i \bullet B_j$	=	0	point on plane, IN (or ON),
$Q_i \bullet B_j$	>	0	point is on +normal side, definitely OUT,
$Q_i \bullet B_j$	<	0	point is on -normal side, maybe IN.

3. After classification, there will be:
    1. no points in $\{ Q_i \}$ that classify as IN, in which case the ray misses the polyhedron,
    2. two points in $\{ Q_i \}$ that classify as IN. In this case, let Q_1 be the point with the smallest t_i, the enter point, and Q_2 be the other, the exit point. The normal for the enter point is the plane normal; the normal for the exit point is the negative of the normal.

A second approach is even simpler and faster. Observe that if the ray intersects the convex polyhedron, the entering t value must be less than the exiting t value. Conversely, if the ray intersects the infinite planes made by the faces of the object outside of the face boundaries, the entering t value will not be less than the exiting value. The entering intersection will have $V \bullet N < 0$, and the exiting $V \bullet N > 0$ (see Figure 13-27).

The algorithm shown in Figure 13-28 combines the two steps in the previous algorithm into

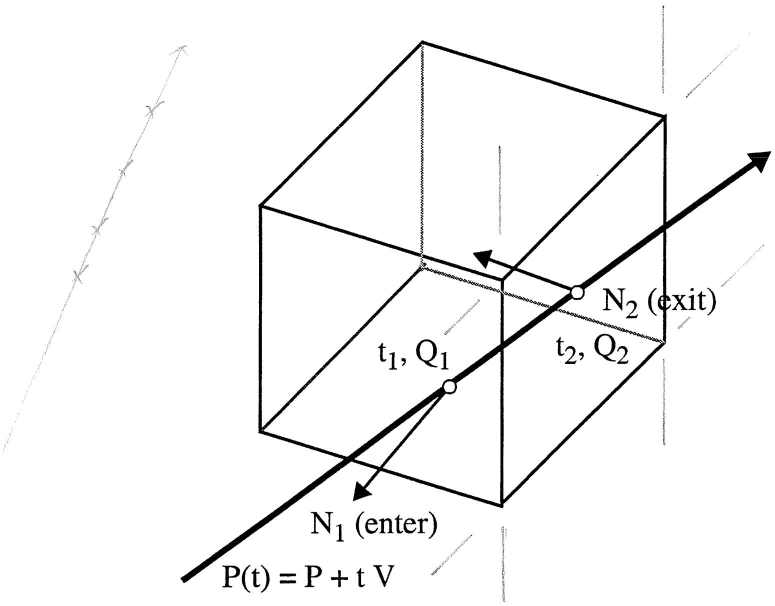


Figure 13-27. Ray Intersecting a Polyhedron.

Q_i 6 points

" $B = CTM^{-1} \cdot B^T$ "

one loop.

```
Initialize $t_{enter} = -\infty$, $t_{exit} = +\infty$.
FOR each plane $B_i$ of the object:
    IF $V \bullet B_i \approx 0$ THEN
        IF $P \bullet B_i > 0$ THEN
            no hits, exit
        ELSE
            skip to the next plane
        ENDIF
    ENDIF
    $t = - P \bullet B_i / V \bullet B_i$
    IF $V \bullet B_i < 0$ AND $t > t_{enter}$ THEN $t_{enter} = t$, $B_{enter} = B_i$.
    IF $V \bullet B_i > 0$ AND $t < t_{exit}$ THEN $t_{exit} = t$, $B_{exit} = B_i$.
ENDFOR
IF $t_{enter} <= t_{exit}$ AND $t_{enter} != -\infty$ THEN
    RETURN 2 hits:   $t_{enter}$ on $B_{enter}$, normal $+B_{enter}$ and
                     $t_{exit}$ on $B_{exit}$, normal $-B_{exit}$
ELSE
    RETURN "no hits".
ENDIF
```

Figure 13-28. Ray - Convex Polyhedron Intersection Algorithm.

13.8 Intersecting a Ray with a Conic Solid

The general equation of a conic solid is

$$A\,x^2 + B\,y^2 + C\,z^2 + D\,x + E\,y + F\,z + G\,xy + H\,yz + I\,xz + J = 0.$$

This equation can be represented in matrix form:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \mathbf{A} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

where **A** is a matrix of constant coefficients. For example, a sphere of radius R and center at the origin has the following **A** matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -R^2 \end{bmatrix}$$

Letting P = [x,y,z,1], the equation is,

$$P\,\mathbf{A}\,P^T = 0.$$

The intersection of a ray with a general conic can now be computed using this more convenient matrix form. The ray equation is the familiar

$$P(t) = P + t\,V.$$

Note that all coordinates are assumed to be homogeneous, so V = [x, y, z, 0]. Substituting P(t) for P in the matrix equation above yields:

$$(P + tV)\,\mathbf{A}\,(P + tV)^T = 0.$$

After some manipulations, this reduces to:

$$t^2 (VAV^T) + t (PAV^T + VAP^T) + PAP^T = 0.$$

This is a quadratic formula in an unknown parameter t. Letting

$$a = VAV^T,\ b = PAV^T + VAP^T, \text{ and } c = PAP^T,$$

yields the standard form of the quadratic formula: $a\,t^2 + b\,t + c = 0$ with the well known solution:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The real roots of this equation, if any, are the parameter values of the intersection points of the ray and the conic. The discriminant inside the radical, $b^2 - 4ac$, determines the solution:

1. if it is negative, there are no real roots and hence no intersection between the ray and conic,
2. if it is 0, there are two identical real roots, so the intersection is a point of tangency between the ray and conic,
3. if it is positive and non-zero, there are two real roots corresponding to the two intersection points between the ray and the conic. Generally, two identical roots can be handled the same as two distinct roots.

The ray-conic intersection routine simply checks the sign of the discriminant and returns "no hits" if it is negative or "2 hits at t_1 and t_2" if it is positive. The values for t_1 and t_2 are the minimum and maximum of the two roots, respectively.

To find the normal vector to a conic at a given point, Q, one begins with the basic equation of the transformed conic. Letting P = [x,y,z,1] and **A** be the transformed coefficient matrix,

$$PAP^T = f(x, y, z) = 0$$

The normal vector is found using the vector gradient operator, ∇, defined as:

$$N = \nabla f(x, y, z) = \frac{\partial}{\partial x}f\hat{i} + \frac{\partial}{\partial y}f\hat{j} + \frac{\partial}{\partial z}f\hat{z}$$

Applying the gradient operator to the conic at point Q:

$$N = \nabla(PAP^T)\Big|_{P=Q} = \left(\frac{\partial}{\partial x}(PAP^T)\hat{i} + \frac{\partial}{\partial y}(PAP^T)\hat{j} + \frac{\partial}{\partial z}(PAP^T)\hat{k}\right)\Bigg|_{P=Q}$$

The only variables involved in the partial derivatives are x, y and z in P. For example, evaluating the x partial derivative,

$$\frac{\partial}{\partial x}(PAP^T) = \frac{\partial P}{\partial x}(AP^T) + (PA)\frac{\partial P^T}{\partial x}$$

$$\frac{\partial P}{\partial x} = \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix} = \hat{i}$$

$$\begin{aligned}\frac{\partial}{\partial x}(PAP^T) &= \hat{i}(AP^T) + (PA)\hat{i}^T \\ &= 2(PA)\hat{i}^T\end{aligned}$$

Similarly,

$$\frac{\partial}{\partial y}(PAP^T) = 2(PA)\vec{j}^T$$

$$\frac{\partial}{\partial z}(PAP^T) = 2(PA)\vec{k}^T$$

Summing the gradient,

$$N = \nabla(PAP^T) = (2(PA)\vec{i}^T)\vec{i} + (2(PA)\vec{j}^T)\vec{j} + (2(PA)\vec{k}^T)\vec{k}$$

Dropping the factor of 2 (N is normalized anyway) and combining the vector components into a matrix, the result is:

$$N = QA$$

The w coordinate of N must be set to 0.

13.9 Ray-Tracing CSG Solids

CSG (Constructive Solid Geometry) is a method for representing volumetric solids as primitive solids combined using the Boolean set operations Union, Difference and Intersection.

Consider the following example of set operations applied to 2D areas:

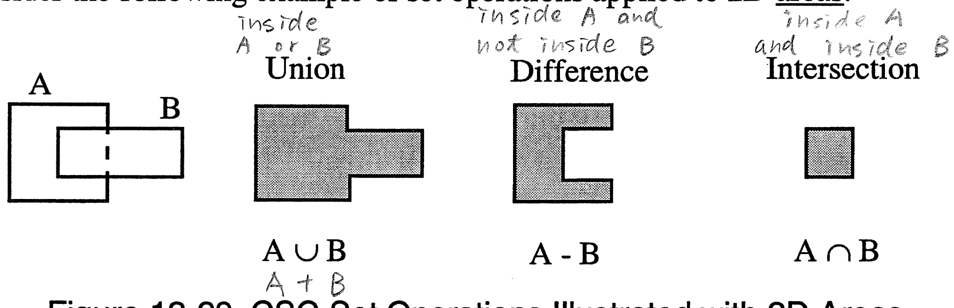


Figure 13-29. CSG Set Operations Illustrated with 2D Areas.

Letting TRUE mean “inside or on” and FALSE mean “outside,” construct truth tables for the operators as shown in Figure 13-30.

In 3D, this logic applies to volumes. For example, in Figure 13-31 a hole is placed through a block by subtracting a cylinder from it. The truth tables, however, are the same. Each primitive must be able to answer the query: is the point [x,y,z] inside “you”? CSG can be used with ray tracing to create realistic images of complex objects composed of Boolean combinations of simpler

UNION

		B	
	∪	T	F
A	T	T	T
	F	T	F

"inside A OR inside B"

$A \cup B$

DIFFERENCE

		B	
	—	T	F
A	T	F	T
	F	F	F

"inside A AND outside B"

$A \cap \sim B$

INTERSECTION

		B	
	∩	T	F
A	T	T	F
	F	F	F

"inside A AND inside B"

$A \cap B$

Figure 13-30. Truth Tables and Geometric Meaning of Set Operations.

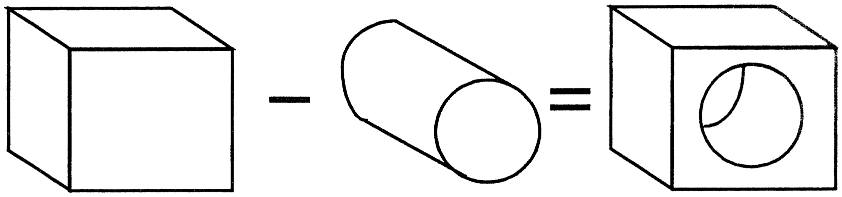


Figure 13-31. Boolean Difference Applied to Volumes.

primitives. This involves:

1. computing the intersections of the ray with each of the primitives and storing them (parameter value, object identifier, etc.),
2. sorting the intersections by increasing ray parameter value,
3. evaluating the resulting intersection list using the CSG "expression," i.e., the list of primitives and the Boolean operators applied to them:

$$\text{primitive}_1 + \text{primitive}_2 - \text{primitive}_3 + \ldots\ \text{primitive}_n$$

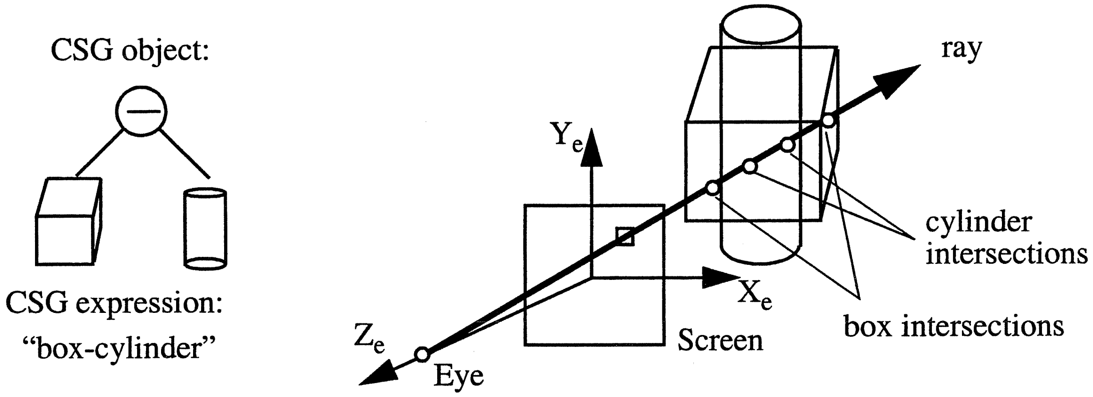


Figure 13-32. CSG Expression and Ray Intersections.

To evaluate the intersections using the CSG expression, consider the ray as the real line starting at the left at -∞ (far behind the eye), and proceeding to the right to +∞ (far in front of the eye) as illustrated in Figure 13-33.

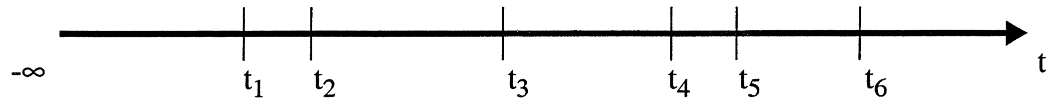


Figure 13-33. Ray Intersections Shown as Points on the Real Line.

For each primitive, its current "ray state" is maintained as a Boolean variable S_i for i = 1..#objects. Initially, assume the ray begins outside all primitives, so S_i = FALSE. If it is possible for the ray to start inside a primitive, then additional computations are needed to determine the initial S_i for each primitive. Begin processing each intersection along the ray, nearest to farthest (left to right on the real line), evaluating the Boolean expression using the current ray states of each primitive, S_i. As each intersection is reached, denoted by the parameter t_j, "toggle" the ray state of the corresponding primitive, S_j. Toggling means to change TRUE (IN) to FALSE (OUT), and vice-versa, corresponding to entering the primitive on the first hit and leaving it on the second.

The process terminates when:

1. the expression evaluates to TRUE (IN), this is the visible intersection,
2. the last intersection is passed without evaluating to IN, there is no intersection with the CSG object. (Consider the ray that intersects only a negative object.)

This is illustrated in Figure 13-34.

```
Assign initial S values to each primitive
FOR each intersection $I_i$ in ascending ray parameter order DO
      $S_i$ = NOT $S_i$
      IF EvaluateCSGObject() == TRUE THEN
            IF ray parameter value of $I_i$ < 0.0 THEN
                  RETURN "no hit: behind the eye"
            ELSE
                  RETURN "visible hit is intersection $I_i$"
ENDFOR
RETURN "no hit"
```

Figure 13-34. CSG Ray Tracing Logic.

Given the individual ray states of the primitives, evaluating the Boolean expression utilizes the truth tables. Consider the CSG expression as a list of primitives, each with a SENSE of "+" or

"-". An algorithm for evaluating the expression is shown in Figure 13-34.

```
BOOLEAN EvaluateCSGObject:
result = OUTSIDE
FOR each primitive$_i$ DO
    IF SENSE(primitive$_i$) == "+" THEN
        result = result OR S$_i$
    IF SENSE(primitive$_i$) == "-" THEN
        result = result AND (NOT S$_i$)
ENDFOR
RETURN result
```

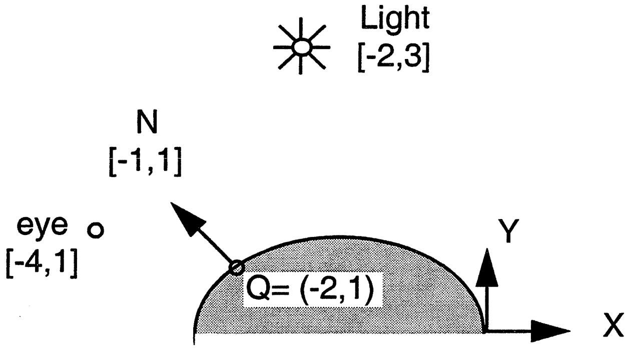
Figure 13-35. Evaluation of a CSG List with Primitive States.

13.10 Review Questions

1. Given Q (point on the surface) = [1, 1, 0], PL (light position) = [-1, 1, 0] and N (normal at Q) = [0, 1, 1], compute the parametric vector ray equation of the shadow ray that can be used in a ray tracing program to determine if the light can contribute to the intensity at Q. For what range of values of the shadow ray parameter will intersections with other objects cause Q to be in a shadow?
2. Given the lighting situation: EYE = [1, 0, 1], PL (light position) = [-1, 1, 0], **Q** (point on the surface) = [1, 1, 0] and N (normal at **Q**) = [0, 1, 1], compute the unit vectors L, H, and E.
3. Describe, in step-by-step outline form, how one does smooth shading of a curved surface approximated by polygons (i.e. a sphere) using the scan-line z–buffer HSR algorithm. Make illustrations, and show all equations pertaining to smooth shading.
4. Given AIM = (1,0,1), EYE = (0,0,1), window half-size = 1, and horizontal and vertical (full screen) resolutions = 101, find: (a) Xe, Ye and Ze, the unit axes of the eye coordinate system expressed in XYZw (If you can't find them, assume values and continue; (b) the position vector for the lower left corner pixel on the window; (c) DX, the vector between adjacent pixels on a scan-line; (d) DY, the vector between adjacent vertical pixels between scan-lines. First show symbolic equations, then compute numeric values; (e) For ORTHOGRAPHIC projection, find the vector equation of the ray through S.

5. A tetrahedron is a closed object bounded by four triangles intersecting at the four corner points. If the points are labelled A, B, C and D, then the triangles are ABC, BDC, ACD and ADB. Given the four corner points of a tetrahedron, A=(1,0,1), B=(2,0,3), C=(2,2,2), D=(3,0,1), the lighting model coefficients Ka = 0.2, Kd = 0.3, Ks = 0.4, and n = 2, the eye located at infinity on the +z axis, and the light located at infinity on the +x axis: compute the intensity of light from each face of the tetrahedron that reflects light to the eye using the ambient-diffuse-specular lighting model with the given parameters. Show equations in symbolic form prior to performing arithmetic.

6. For the ray tracing situation shown below: (a) compute and draw L, the unit direction to the light source; (b) compute and draw E, the unit direction toward the eye; (c) compute and draw R, the unit direction of ideal reflection; (d) compute cos(θ), and label θ on the diagram; (e) compute cos(ϕ) and label ϕ on the diagram. First show symbolic equations, then compute numeric values.



Chapter 14. Curves

14.1 Parameterization

When storing the definition of a line, it is not necessary to store any intermediate points, just the end points.

Figure 14-1. A Line Segment.

$P(t) = P_1 + t\ (P_2 - P_1)$ or $P(t) = (1 - t)\,P_1 + t\,P_2$

Similarly, to draw an arc or circle, you only need the center C and radius R:

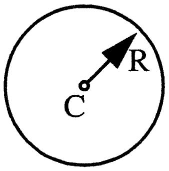


Figure 14-2. Circle.

$x(t) = R\ \cos(t),\ y(t) = R\ \sin(t)$ for $\theta_1 <= t <= \theta_2$

To draw the circle, one would connect a series of points around the circle by straight lines. The number of lines could vary from circle to circle to provide the appropriate quality of display.

A general 3D curve can be described parametrically, for example, a helix (Figure 14-3):

$x(t) = R\cos(t)\ ,\quad y(t) = R\sin(t)\ ,\quad z(t) = C\,t$

It is possible to derive other parametric forms for analytic curves, such as second degree equations, i.e. conics: circles, ellipses, parabolas, and hyperbolae. For example, a circle can be represented by the parametric equations in Figure 14-4. Unfortunately, the parameter range of t needed to cover the whole circle is $-\infty < t < \infty$. This is impractical and also shows that the point distribution around the circle is highly non-linear.

It is computationally impractical to use a parameter range including the value infinity. By introducing a one dimensional homogeneous representation for the parameter, a more computationally convenient range is obtained. Replace t in the preceding equations with the ratio

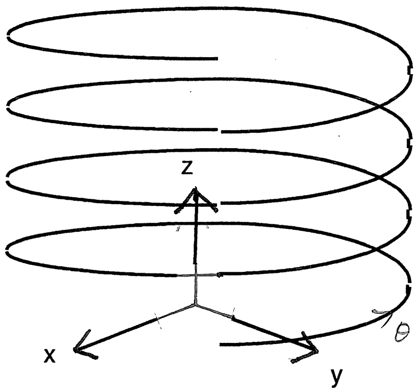
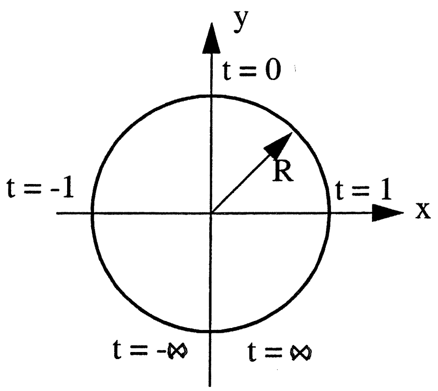


Figure 14-3. Helix.



$$x = \frac{2tR}{1+t^2}$$

$$y = \frac{(1-t^2)R}{1+t^2}$$

Using homogeneous coordinates:

$$\begin{bmatrix} wx & wy & w \end{bmatrix} = \begin{bmatrix} 1 & t & t^2 \end{bmatrix} \begin{bmatrix} 0 & R & 1 \\ 2R & 0 & 0 \\ 0 & -R & 1 \end{bmatrix}$$

Figure 14-4. Parametric Equations for a Circle.

$w = 1+t^2$

$w x = 2Rt$

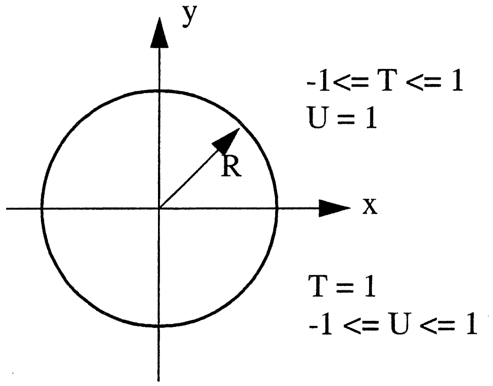
$\frac{wx}{w} = \frac{2Rt}{1+t^2}$

$\frac{T}{U}$. The resulting parameterization is more notationally complex as it now involves two parameters, T and U. Represent t = ∞ as the ratio of $\frac{T}{0}$

" $t = \frac{T}{U}$ "

The complete circle is now traced in two parts. The top half is completed by holding U=1 with -1<=T<=1. The bottom half is then traced by holding T constant at 1 and letting U vary over the range -1 to 1, as shown in Figure 14-5.

Comparing the "*parametric representation*" for curves, C(t) = [(x(t) y(t) z(t)], to the "*implicit representation*", C(x,y,z) = 0, there are advantages and disadvantages for each. Parametric representation is best for drawing curves and operations requiring an ordered point set and for transformations of curves. Implicit representation is best for geometric intersection computations and for determining if a given point is on a curve. Ideally, one would like to have both the parametric and implicit representations. However, this is a very complex process at times and may not be possible at all for some curves.



$$t = \frac{T}{U}$$

$$x = \frac{2UTR}{U^2 + T^2}$$

$$y = \frac{(U^2 - T^2)R}{U^2 + T^2}$$

Figure 14-5. Homogeneous Parameterization of a Circle.

The mathematical process of converting a parametric form into its corresponding implicit form has been termed *implicitization*. Finding the parameter value corresponding to a given (x,y,z) coordinate on a parametric curve is termed *inversion.* These processes are important computational geometry methods used in computer-aided design (CAD) systems.

14.2 Spline Curves

The presentations so far have been relatively simple shapes and their defining equations. To be of any practical value in applications, the methods discussed here must be extensible to more “free-form” curve shapes. Drafts-people have been designing free-form curves for years in 2D, using a flexible beam called a spline and weights called ducks. The spline is formed using as many ducks as are necessary to control the shape. Then a pencil is drawn along the spline to produce the desired curve. The stiffness of the spline makes the curve “smooth.” This process can be simulated

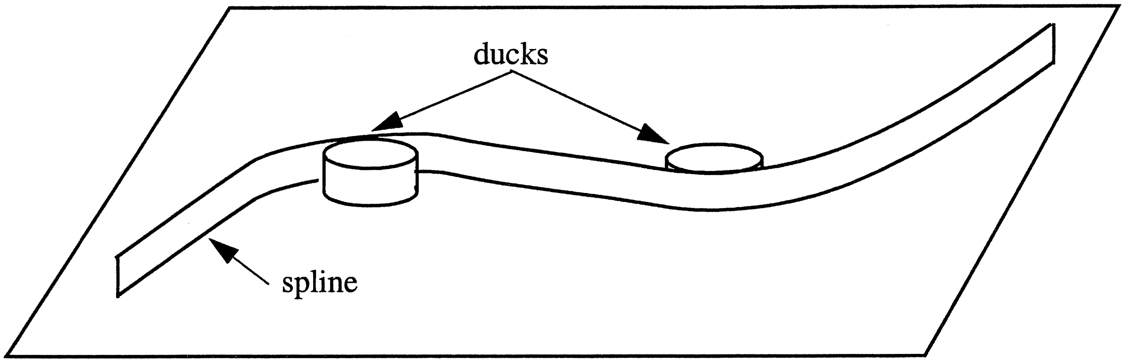


Figure 14-6. Curve Drawing with a Spline and Ducks.

mathematically using equations to represent the curves. The most common form for these equations is the parametric polynomial.

14.3 Polynomial Forms

The characteristics of the spline and ducks can be modeled mathematically using differential equations. However, in a *shape design system* this accuracy and complexity are generally unnecessary. Fortunately, a similar effect is achievable using the analytically and computationally simpler *polynomial* forms that define *algebraic* curves. These algebraic curves can be defined implicitly, but for nearly two decades the parametric representation has been the representation of choice in shape design systems.

A 3D parametric polynomial curve is shown below using 't' as the independent parameter and the standard *"power polynomial" basis*:

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + \dots + a_n t^n$$

$$y(t) = b_0 + b_1 t + b_2 t^2 + b_3 t^3 + \dots + b_n t^n$$

$$z(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 + \dots + c_n t^n$$

or in vector form, → A_0 (vector)

$$P(t) = [\, x(t)\ \ y(t)\ \ z(t) \,] = A_0 + A_1 t + A_2 t^2 + \dots + A_n t^n .$$

where: $A_i = [\, a_i\ b_i\ c_i \,]$

What should be the degree of the polynomial, n?

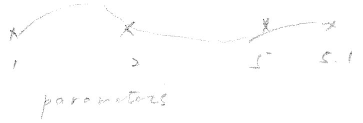
n	P(t)	Description
1	$A_0 + A_1 t$	linear (straight lines)
2	$A_0 + A_1 t + A_2 t^2$	quadratic (planar conics)
3	$A_0 + A_1 t + A_2 t^2 + A_3 t^3$	cubic (the lowest order "space" curve)

As shown in the circle example, the parametric definition is valid over the range $(\underset{-\infty}{-x} <= t <= \underset{\infty}{x})$, but most often we are only interested in a segment of the curve. This finite segment is defined by imposing bounds on the acceptable parameter range, usually 0 <= t <= 1.

For *cubic polynomials* (n=3), a convenient matrix notation is often used:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix}}_{T} \underbrace{\begin{bmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{bmatrix}}_{\mathbf{A}}$$

power basis vector coefficient matrix



A curve is defined by its *coefficient matrix* **A**. The goal is to define **A** in an intuitive and concise manner in order to produce a curve of the desired shape.

One way to accomplish this is to force the curve to *interpolate* (pass through) four points spaced along the curve at pre-assigned parameter values. Given 4 known points, including their respective parameter values along the intended curve, we can derive the **A** matrix for the curve.

Given $P_0 = (x_0, y_0, z_0), \ldots, P_3 = (x_3, y_3, z_3)$ and (t_0, t_1, t_2, t_3), we have a system of 12 linear equations in 12 unknowns, and therefore we can solve for the coefficient matrix **A**.

We know that $P(t_i) = P_i$ for $i = 0..3$, so we can form the matrix equation:

$$\begin{bmatrix} x_0 & y_0 & z_0 & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 1 & t_1 & t_1^2 & t_1^3 \\ 1 & t_2 & t_2^2 & t_2^3 \\ 1 & t_3 & t_3^2 & t_3^3 \end{bmatrix} \mathbf{A}$$

This can be solved for **A**:

$$\mathbf{A} = \begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 1 & t_1 & t_1^2 & t_1^3 \\ 1 & t_2 & t_2^2 & t_2^3 \\ 1 & t_3 & t_3^2 & t_3^3 \end{bmatrix}^{-1} \begin{bmatrix} x_0 & y_0 & z_0 & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{bmatrix}$$

Given **A**, we can step along the cubic curve segment using sequentially increasing values of t ($t_0 <= t <= t_3$) to compute a set of points that can be connected by straight lines (Figure 14-7).

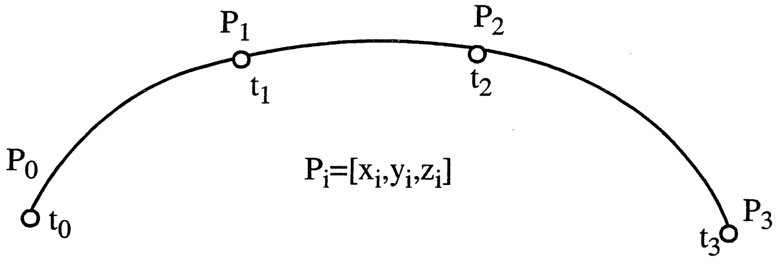


Figure 14-7. Cubic Polynomial Interpolation through Four Points.

Note that moving any one of the four points or changing its associated parameter value will change the coefficients, thus altering the shape of the whole curve (Figure 14-8). We now begin to see the concept of *curve sculpting*. This will not always produce a "nicely-behaved" or smooth curve because "wiggles" can result when a curve is forced to interpolate unevenly distributed points.

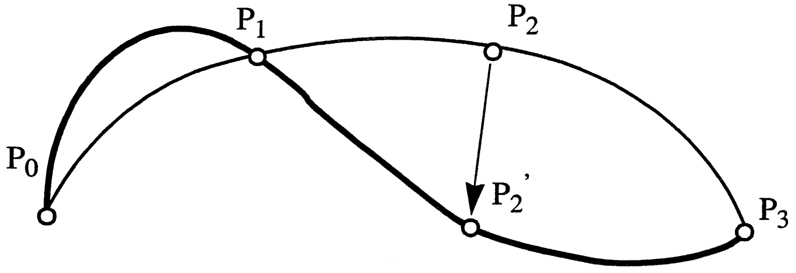


Figure 14-8. Effects of Moving a Point on a Curve.

One difficulty encountered with this formulation is that of matching slopes at the junction point (sometimes called a "*knot*") between segments when creating a "*spline curve*."

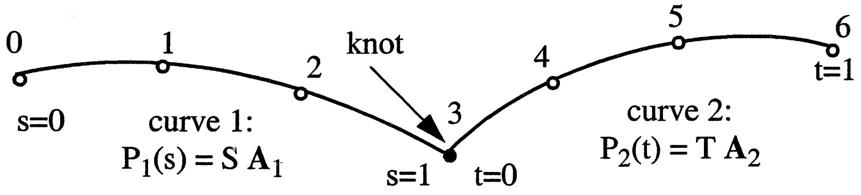


Figure 14-9. Two Curves Meeting at a Knot Point.

$\frac{dy}{dx} = \frac{\frac{dy}{dt}}{\frac{dx}{dt}}$

For many applications, a more appropriate and intuitive approach is achieved by specifying two endpoints and two *parametric tangents*. Taking the derivative of the curve with respect to the parameter:

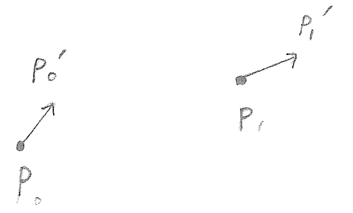
$\left(\frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt}, 0 \right)$

$$\frac{d}{dt}P(t) = P'(t) = \left[x'(t)\ y'(t)\ z'(t)\right] = \left[0\ 1\ 2t\ 3t^2\right]\mathbf{A}$$

We now use the curve end points (P_0 and P_1) and the end parametric tangents (P'_0 and P'_1)

to derive the coefficient matrix. Assuming t varies in the range [0,1]:

$$\begin{bmatrix} P_0 \\ P_1 \\ P_0' \\ P_1' \end{bmatrix} = \begin{bmatrix} P(0) \\ P(1) \\ P'(0) \\ P'(1) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \mathbf{A}$$



where:

$$P'_0 = P'(0) = \frac{d}{dt} P(t) \Big|_{t=0}$$

$$P'_1 = P'(1) = \frac{d}{dt} P(t) \Big|_{t=1}$$

resulting in,

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix}^{-1} \begin{bmatrix} x_0 & y_0 & z_0 & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_0' & y_0' & z_0' & 0 \\ x_1' & y_1' & z_1' & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 & y_0 & z_0 & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_0' & y_0' & z_0' & 0 \\ x_1' & y_1' & z_1' & 0 \end{bmatrix}$$

This is known as *Hermitian interpolation.*

Slope continuity between adjacent curves can now be directly controlled using a common tangent vector at the end of one curve segment and at the start of the next.

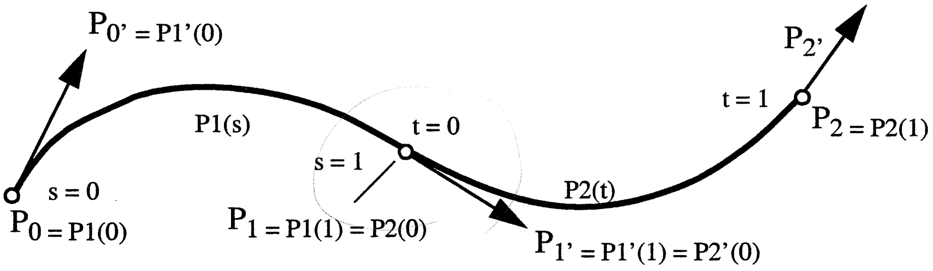


Figure 14-10. Slope Continuity between Spline Curves.

We can now sculpt curves by moving one of the two endpoints or by changing one of the two tangent vectors. Note that a tangent vector has both magnitude and direction, and that changing either value alters the curve definition.

We can now apply a transformation matrix, **M**, to all points on a curve:

$$[\, wx \; wy \; wz \; w \,] = [\, x \;\; y \;\; z \;\; 1] \; \mathbf{M} = \mathrm{T\,A\,M} = \mathrm{T\,A'}$$

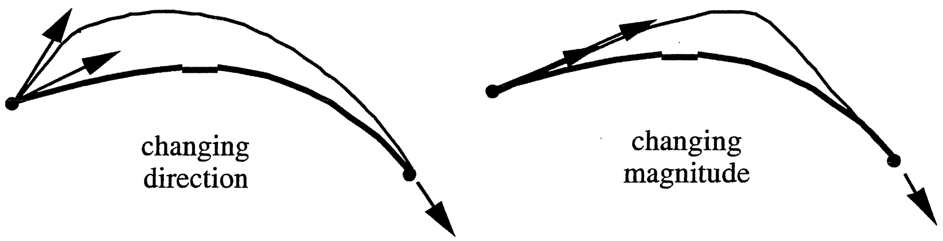


Figure 14-11. Effects of Changing Parametric Tangent Magnitude and Direction.

Now, instead of processing individual straight line segments through the 3D wireframe pipeline, we are able to transform an entire curve. The transformed curve is represented by the transformed coefficient matrix, **A**'. Using homogeneous coordinates, this can even include perspective.

14.4 Bézier Formulation

Another popular curve form is the Bézier curve, shown in Figure 14-12. A degree n Bézier curve is defined by a set of (n+1) ordered *control points.* When connected by straight lines, the points form what is commonly called a *control polygon.* The control points act somewhat like the ducks of the old drafting spline, however, only the first and last control points lie on the curve. The remaining points simply act as weights, pulling the curve towards each point. The resulting curve is said to "mimic" the overall shape of the control polygon, which is one reason Bézier curves are popular for interactive curve design.

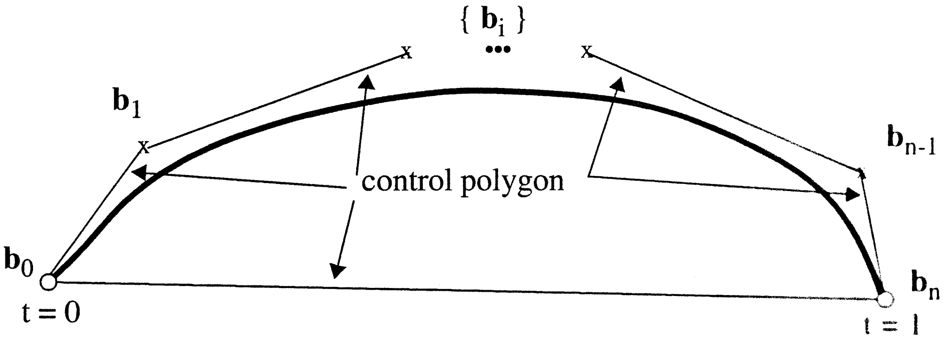
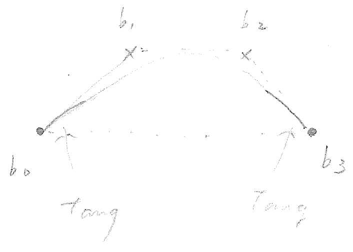


Figure 14-12. Bézier Control Points and Curve.

The general nth degree Bézier curve is defined as follows.

$$P(t) = \sum_{i=0}^{n} B_{n,i}(t)\, b_i$$



where $B_{n,i}(t)$ are Bernstein basis functions,

$$B_{n,i}(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}$$

n is the degree of the curve (largest exponent of t), and b_i, i = 0.. n are the control points.

The constant factor in the Bernstein basis function is the *binomial coefficient*: $\frac{n!}{i!(n-i)!}$. It is expanded for some values of n and i in Figure 14-13. Recall that 0! = 1. Notice that for i = 0 and for i = n, the binomial coefficient is 1.

$P(t) = B_{1,0}(t)\, b_1 + B_{0,1}(t)\, b_2 = (1-t) b_1 + t\, b_2$

i →

n ↓	0	1	2	3	4
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

Figure 14-13. Binomial Coefficients.

A cubic curve is formed when n = 3 (i.e., 4 control points).

$$P(t) = \sum_{i=0}^{3} B_{3,i}(t)\, b_i = b_0B_{3,0}(t) + b_1B_{3,1}(t) + b_2B_{3,2}(t) + b_3B_{3,3}(t)$$

$$= b_0(1-t)^3 + b_1(3t(1-t)^2) + b_2(3t^2(1-t)) + b_3t^3$$

This can be represented in matrix form:

$$P(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$= T\mathbf{M}_{be}\begin{bmatrix} b \end{bmatrix}$$

where:

T is the (power) polynomial basis vector,

$\mathbf{M}_{be}$ is called the Bézier basis change matrix,

[b] is the matrix of Bézier control points.

If more control over the shape of a curve is desired, additional control points may be added (increasing the degree of the polynomial). Using more than one curve segment can also provide additional control, especially in cases where the curve must pass through several points.

One can show that the tangents at the ends of a Bézier curve are parallel to the first and last legs of the control polygon, respectively. Thus, in general for slope continuity, or C^1 continuity, the next-to-last control point on curve #i, the common end point to curve #i and #i+1, and the second control point on curve #i+1 must be collinear (Figure 14-14).

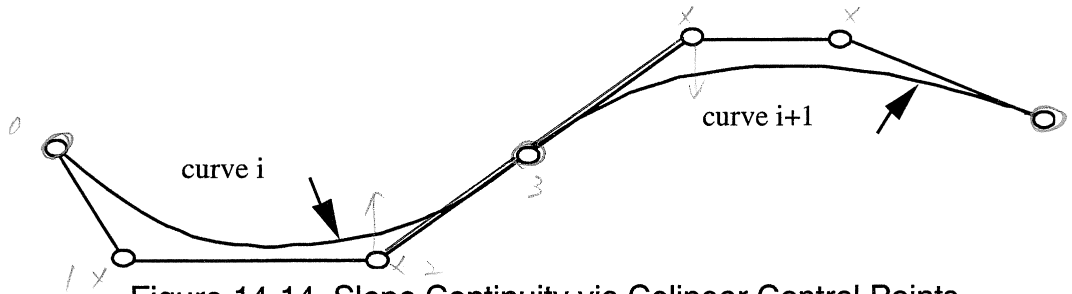
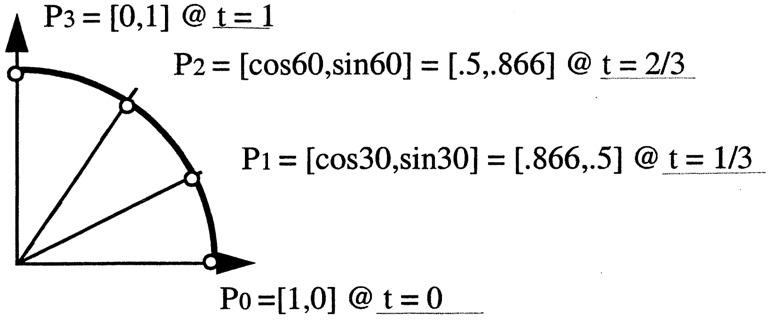


Figure 14-14. Slope Continuity via Colinear Control Points.

Second parametric derivative (curvature or C^2) continuity between curve segments can also be achieved by proper placement of the third and third-to-last control points.

Suppose we want a parametric cubic polynomial representation for a quarter circle in the first quadrant (Figure 14-15).



P3 = [0,1] @ t = 1
P2 = [cos60,sin60] = [.5,.866] @ t = 2/3
P1 = [cos30,sin30] = [.866,.5] @ t = 1/3
Po =[1,0] @ t = 0

Figure 14-15. Interpolation Points for a Quarter Circle.

For point interpolation, the equation [P] = T **A** becomes: $T = [\,1 \;\; t \;\; t^2 \;\; t^3\,]$

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0.866 & 0.5 & 0 & 1 \\ 0.5 & 0.866 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0.3333 & 0.1111 & 0.0370 \\ 1 & 0.6667 & 0.4444 & 0.2963 \\ 1 & 1 & 1 & 1 \end{bmatrix} \mathbf{A}$$

Solving for **A**:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0.04423 & 1.6029 & 0 & 0 \\ -1.4856 & -0.1615 & 0 & 0 \\ 0.4413 & -0.4413 & 0 & 0 \end{bmatrix}$$

Now find the Bézier control points for the same curve. To do this, first examine the matrix equation for the two curve forms, equating the power basis polynomial form with the Bézier form:

$$P(t) = \underline{T\,\mathbf{A}} = \underline{T\,\mathbf{M}_{be}\,[\,b\,]}$$

We see that $\mathbf{A} = \mathbf{M}_{be}\,[\,b\,]$, so $[\,b\,] = [\,\mathbf{M}_{be}\,]^{-1}\,\mathbf{A}$. For this example,

$$[b] = \mathbf{M}_{be}^{-1}\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{1}{3} & 0 & 0 \\ 1 & \frac{2}{3} & \frac{1}{3} & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0.04423 & 1.6029 & 0 & 0 \\ -1.4856 & -0.1615 & 0 & 0 \\ 0.4413 & -0.4413 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \underline{1 \quad 0} & 0 & 1 \\ \underline{1.015 \quad 0.535} & 0 & 1 \\ \underline{0.535 \quad 1.015} & 0 & 1 \\ \underline{0 \quad 1} & 0 & 1 \end{bmatrix}$$

Examine the Bézier control polygon on the quarter circle:

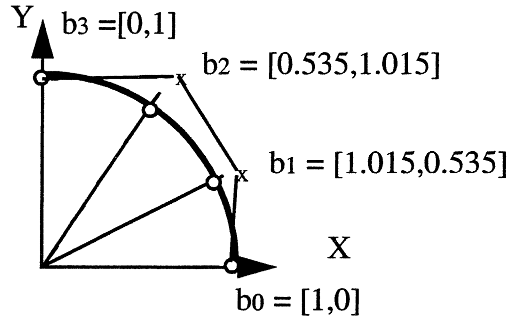


Figure 14-16. Bézier Control Points for the Quarter Circle Example.

14.5 B-Spline Formulation

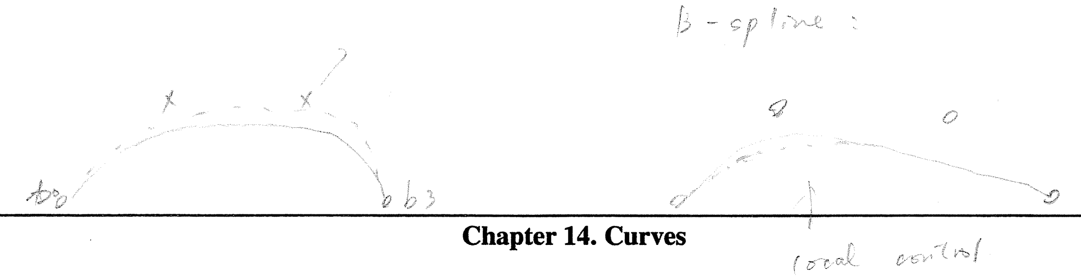
Another popular curve formulation is the B-spline representation. Like the Bézier formulation, the B-spline formulation also involves control points. In addition, however, the B-spline formulation involves *parametric knots* which give more degrees of freedom for controlling shape.

order = degree + 1 ;

The general $\underline{m^{th}\text{ order}}$ B-spline curve is defined as the summation:

$$P(t) = \sum_{i=0}^{m-1} N_{m,i}(t) P_i$$

where the B-spline basis function, $N_{m,i}(x)$, is defined recursively:



parameter

$$N_{m,i}(x) = \frac{(\overset{\downarrow}{x} - x_i)}{(x_{i+m-1} - x_i)} N_{m-1,i}(x) + \frac{(x_{i+m} - x)}{(x_{i+m} - x_{i+1})} N_{m-1,i+1}(x)$$

$$N_{1,i}(x) = \begin{cases} 1 & \text{if } (x \in [x_i, x_{i+1}]) \\ 0 & \text{if } (x \notin [x_i, x_{i+1}]) \end{cases}$$

cubic $m = 4$

$N_{4,i}$

$N_{3,i}$ $N_{3,i+1}$

where: 'x_i' are the parametric knots,

'm' is the order of the curve (degree + 1),

't' is the parameter. $\{x_i\} \leftarrow$ given

The array of knots, called the *knot vector*, is a non-decreasing sequence of parameter values that gives *local control* to the shape of the curve. This means that moving a control point only affects the curve near the point, not the whole curve (as does the power basis polynomial formulation). From the B-spline basis definition, one sees that for a given parameter value, only the adjacent 'm' knots affect the shape.

Unlike the Bézier formulation, it is possible to vary the order of the curve, 'm', without varying the number of control points. As the order decreases, the curve moves closer to the control points, ultimately becoming a piecewise linear function (straight lines between the control points) for order = 2. Control points can be repeated and so can knot values.

Although the knot values can be any values, not just 0 to 1, usually the knot vector consists of integers starting with 0. If the knots are uniformly spaced, then it is termed a *uniform B-spline*; if not, it is a *non-uniform B-spline*. $\rightarrow \{x_i\} = \{0, 0, 0, 1, 1, 1\}$

In general, B-spline curves do not pass through any of their control points. However, they can be formulated to pass through the first and last control points, similar to Bézier curves. This requires that the first and last knot values be repeated 'm' times. Examine the B-spline basis to verify this.

If 'k' is the number of knots, 'v' is the number of control points, and 'X' is the maximum knot value, then we can compute:

$X = (v - 1) - m + 2$

$k = 2m + X - 1$

~~Examples of B-splines formulated in this manner in the following figures.~~

Hence~~, Bernstein~~ basis functions are a special case of B-spline basis functions. In the figures above, the 3rd order B-spline with 3 control points is a quadratic (degree 2, order 3) Bézier curve; the 4th order B-spline with 4 control points is the cubic Bézier; and the 5th order B-spline with 5 control points is a quartic Bézier.

14.6 Parabolic Blending (skim)

Parabolic blending is a curve formulation based solely on points on the curve. Given a set of four points, compute a curve between the middle two points by linearly blending parabolas through the first three points and second three points, respectively.

In general, a parabola can be represented in vector-matrix form as:

$$P(s) = [\, s^2 \;\; s \;\; 1 \,]\, \mathbf{A}$$

where **A** is a 3 by 3 matrix of coefficients that define the curve.

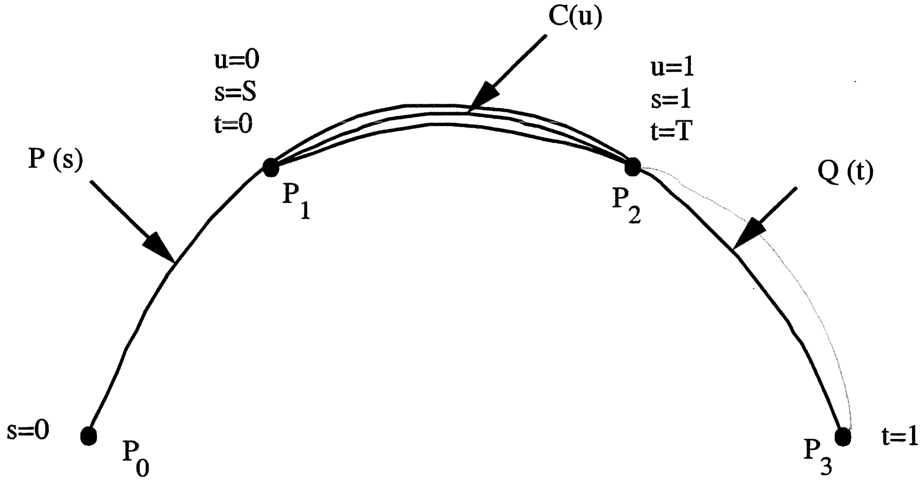


Figure 14-17. Parabolic Blending of C(u) from P(s) and Q(t).

Given three points defining the parabola such that:

$$P_0 = P(0),\; P_1 = P(x),\; P_2 = P(1)$$

we can combine these equations into one matrix equation: ?5

$$\begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ x^2 & x & 1 \\ 1 & 1 & 1 \end{bmatrix} \mathbf{M}(x)$$

This equation can be solved for **M**(x):

$$\mathbf{M}(x) = \begin{bmatrix} 0 & 0 & 1 \\ x^2 & x & 1 \\ 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix} = \frac{1}{x^2 - x} \begin{bmatrix} x-1 & 1 & -x \\ 1-x^2 & -1 & x^2 \\ x^2-x & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix}$$

The first parabola, P(s), passes through the first three points, and the second parabola, Q(t) passes through the last three points. We now linearly blend these two parabolas (2nd order curves) using the parameter u to obtain a cubic curve between P_1 and P_2:

$C(u) = (1 - u)\,P(s) \oplus u\,Q(t)$

Note that parameters s and t are linear functions of u:

$s = S + u\,(1\text{-}S)$ and $t = T\,u$

Combining all this, the equation becomes:

$$C(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} \left(-\frac{(1-S)^2}{S}\right) & \left(T + \frac{(1-S)}{S}\right) & \left(S + \frac{1}{T-1}\right) & \left(\frac{T^2}{1-T}\right) \\ \left(\frac{2(1-S)^2}{S}\right) & \left(\frac{2(S-1)}{S} - T\right) & \left(\frac{T-2}{T-1} - 2S\right) & \left(\frac{T^2}{T-1}\right) \\ \left(-\frac{(1-S)^2}{S}\right) & \left(\frac{(1-2S)}{S}\right) & S & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

where S is the value of s on P(s) at P_1, and T is the value of t on Q(t) at P_2, for u in [0,1].

Values for S and T are determined by considering slope continuity between adjacent blended segments, like $C_1(u)$ and $C_2(u)$ in the figure above.

For S= 0.5 and T = 0.5:

$$C(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

This is a simple cubic interpolation scheme given points that must lie on the curve. Each set of four points corresponds to one parabolically blended curve, as illustrated in Figure 14-18.

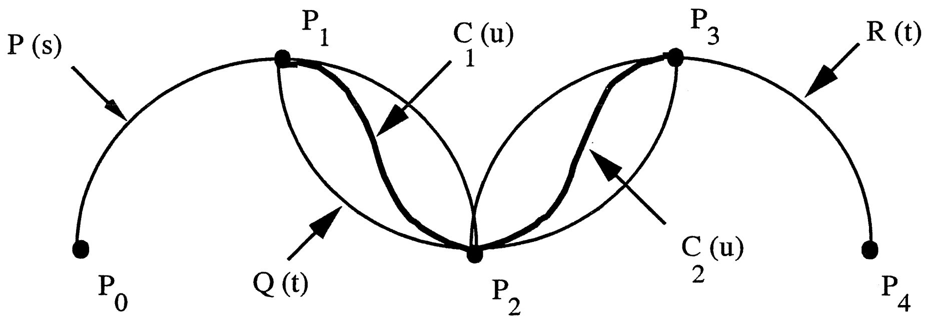


Figure 14-18. Parabolic Blending of Spline Curves.

14.7 Rational Formulations

As presented, parametric curves based on polynomials cannot exactly represent the conic forms (circular arcs, etc.). For our quarter circle example, Figure 14-19 shows the error in the circle radius ($d = 1 - x^2 - y^2$) at the x and y coordinates generated by $P(t) = T\,\mathbf{A}$. Note that the error is zero only at t=0, $\frac{1}{3}, \frac{2}{3}$ and 1, the interpolated points.

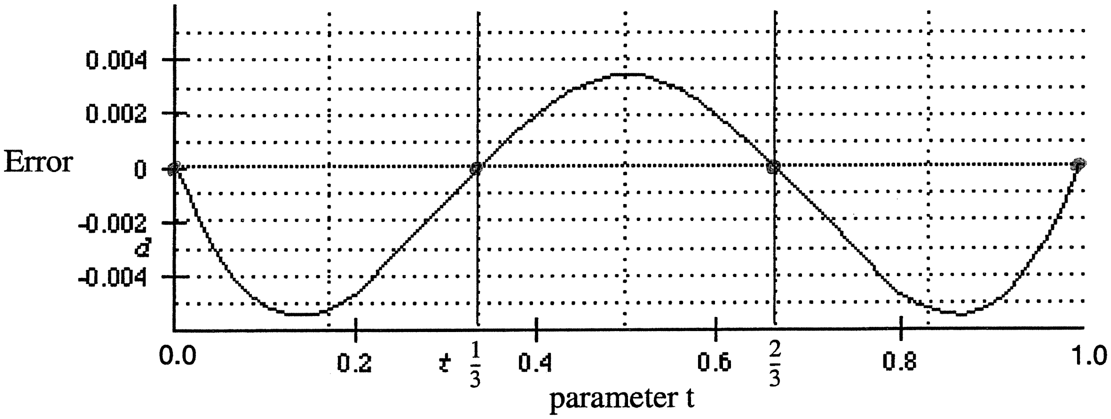


Figure 14-19. Error Between Exact Circle Radius and Polynomial Interpolation.

To provide greater flexibility, the curve representations can be expanded using homogenous coordinates.

$$wx(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + \ldots + a_n t^n$$

$$wy(t) = b_0 + b_1 t + b_2 t^2 + b_3 t^3 + \ldots + b_n t^n$$

$$wz(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 + \ldots + c_n t^n$$

$$\rightarrow w(t) = d_0 + d_1 t + d_2 t^2 + d_3 t^3 + \ldots + c_n t^n$$

In vector form:

$$P(t) = [\; wx(t) \; wy(t) \; wz(t) \; w(t) \;].$$

Notice that the 3D coordinates of a point now are the ratio of two polynomials and hence the term *rational curves*:

$$x(t) = \frac{wx(t)}{w(t)} \qquad y(t) = \frac{wy(t)}{w(t)} \qquad z(t) = \frac{wz(t)}{w(t)}$$

As an example, recall the parameterization of a circle given earlier:

$$\begin{bmatrix} wx & wy & w \end{bmatrix} = \begin{bmatrix} 1 & t & t^2 \end{bmatrix} \begin{bmatrix} 0 & R & 1 \\ 2R & 0 & 0 \\ 0 & -R & 1 \end{bmatrix}$$

The quadratic (n=2) Bézier formulation yields:

$$P(t) = \begin{bmatrix} 1 & t & t^2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -2 & 2 & 0 \\ 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

Equating the matrices as before and solving for the control points:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \mathbf{M}_{be}^{-1}\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 2 & 0 \\ 1 & -2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 & R & 1 \\ 2R & 0 & 0 \\ 0 & -R & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & \frac{1}{2} & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & R & 1 \\ 2R & 0 & 0 \\ 0 & -R & 1 \end{bmatrix} = \begin{bmatrix} 0 & R & 1 \\ R & R & 1 \\ 2R & 0 & 2 \end{bmatrix}$$

Compare these results with the previous cubic Bézier control points:

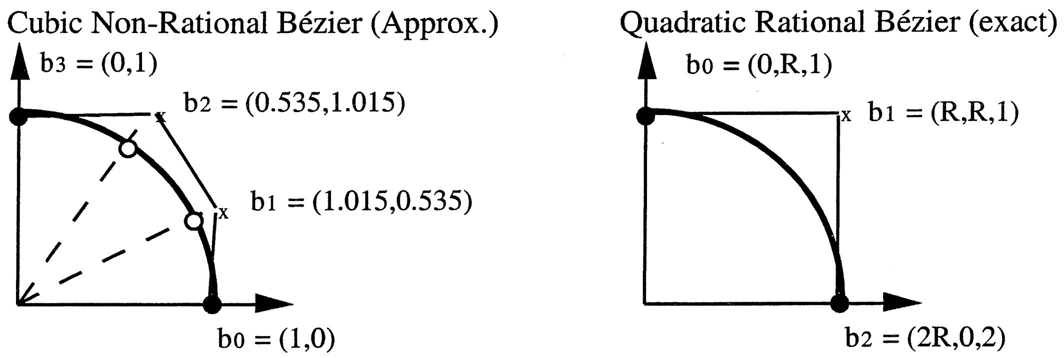


Figure 14-20. Comparison of Non-Rational and Rational Circle Forms.

Non-Uniform Rational B-Spline

$w(t)$

↑

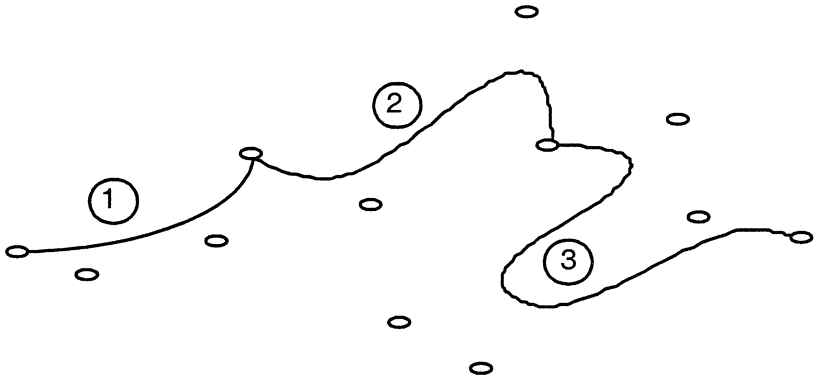
N U R B

↓

$\{x_i\}$

14.8 Review Questions

1. Draw the Bézier control polygons and state the degree for each of the following curves:



2. The general nth degree Bezier curve equation is:

$$P(t) = B^n(t) = \Re_{i=0}^{n} \left(\frac{n!}{i!\,(n-i)!} \right) t^i (1-t)^{n-i}\, b_i$$

The vector form of the cubic Bézier curve (n=3) is:

$$B^3(t) = b_0 (1-t)^3 + 3\, b_1\, t\, (1-t)^2 + 3\, b_2\, t^2 (1-t) + b_3\, t^3$$

After expanding the coefficients, the matrix form of this cubic equation is:

$$B^3(t) = [\, t^3\, t^2\ t\ 1\,] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = [T][M_b][b]$$

For the <u>quadratic</u> (n=2) Bézier curve, find: (1) the vector form, and (2) the matrix form, written similar to the examples above. (Recall that 0! = 1.)

Chapter 15. Surfaces

The parametric forms of curves can be extended to surfaces by adding another parametric dimension, from P(t) to P(s, t). The objective is to define a complete surface in terms of certain discrete entities which provide some geometric insight as to the resulting shape. As before with the curve segments, we are considering only a finite portion of an infinite surface. This is generally referred to as a *surface patch.* $\vec{P}(t) \rightarrow \vec{P}(s,t)$

15.1 Lofted Surfaces

The easiest approach is also the oldest: a technique called *lofting*. A *linear blend* is made between two curves (typically each is in a parallel plane). The two base curves use a common parameter 't' and range (typically [0,1]), and an additional parameter 's' is introduced to blend these two curves linearly. The ranges of the two parameters, s and t, form a two dimensional space termed the *parametric domain* of the surface.

Given the two base curves: $P_1(t)$ and $P_2(t)$, the lofted surface is defined as:

$$P(s,t) = (1-s)\,P_1(t) + s\,P_2(t) \qquad 0 <= s,t <= 1$$

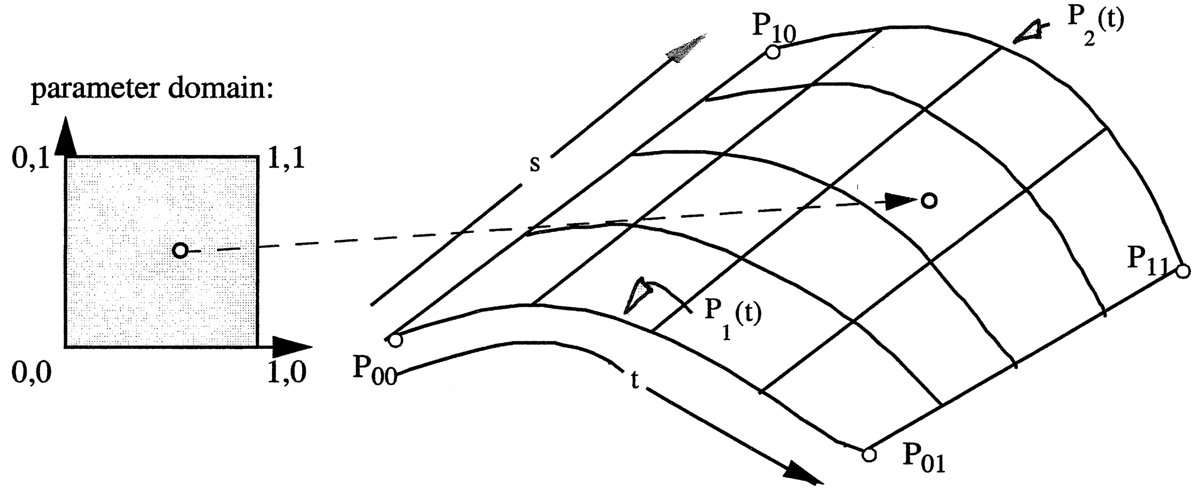


Figure 15-1. Lofted Surface as a Linear Blend of Curves.

This type of surface is also called a *ruled surface*. Lofting is adequate for certain applications (e.g. airfoils) but tends to leave the surface somewhat “flat” in one direction (Figure

15-2).

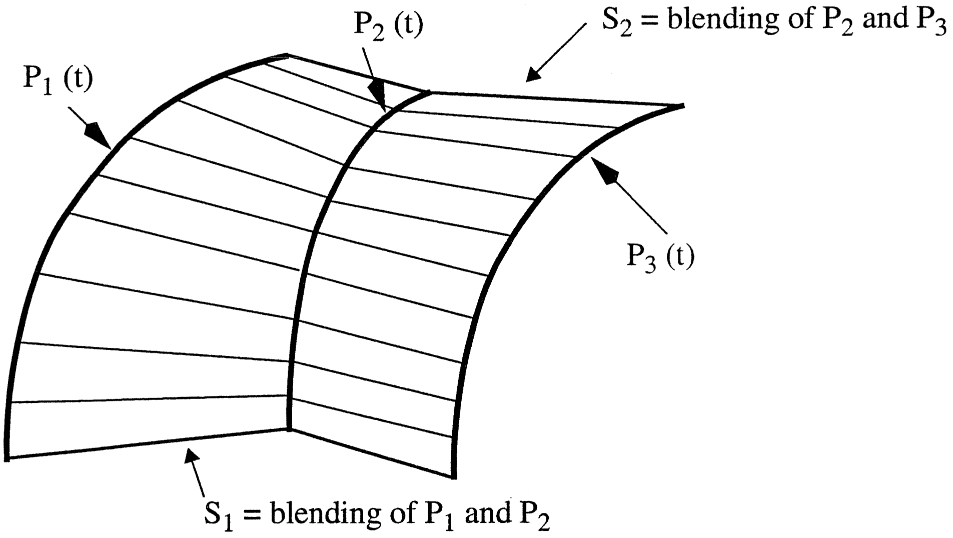
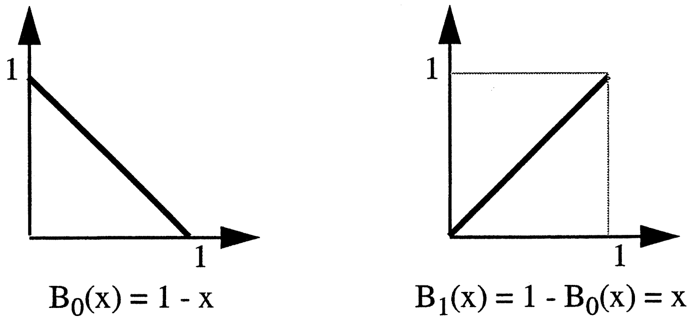


Figure 15-2. Slope Discontinuity at Interior Curves of Lofted Surfaces.

15.2 Linear Interior Blending

One might intuitively feel that any four space curves with four common corner points could be interpolated to yield a surface definition. The interpolation is done with linear *blending functions*, such as the linear functions $B_0(x)$ and $B_1(x)$ shown in Figure 15-3. It is essential that the blending functions sum to unity for the surface to contain its corner points.



$B_0 + B_1 = 1$

Figure 15-3. Linear Blending Functions.

The surface is now a linear blend of the curves and the corner points:

$$
\begin{aligned}
P(s,t) = \quad & P_1(t)\ B_0(s) \ + \ P_4(s)\ B_0(t) \\
& + P_3(t)\ B_1(s) \ + \ P_2(s)\ B_1(t) \\
& - P_{00}\ B_0(t)\ B_0(s) \\
& - P_{01}\ B_1(t)\ B_0(s) \\
& - P_{11}\ B_1(t)\ B_1(s) \\
& - P_{10}\ B_0(t)\ B_1(s)
\end{aligned}
$$

where: P_1, P_2, P_3, P_4 are bounding curves,

$P_{00}, P_{01}, P_{11}, P_{10}$ are the corner vertices. See Figure 15-4.

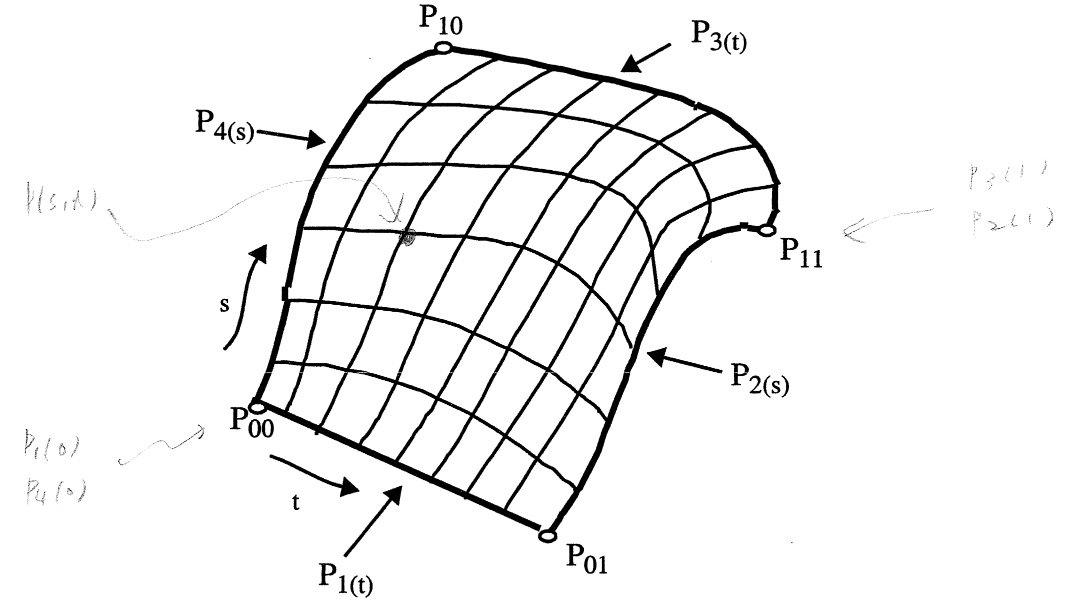


Figure 15-4. Surface with Linear Interior Blending of Four Curves.

Note the boundary conditions for the boundary curves, i.e. when s and t are 0 and 1:

$$P_1(0) = P_4(0) = P_{00}$$

$$P_1(1) = P_2(0) = P_{01}$$

$$P_2(1) = P_3(1) = P_{11}$$

$$P_3(0) = P_4(1) = P_{10}$$

Now check the s = t = 0 boundary condition for the surface:

$$\begin{aligned}
P(\, s{=}0,\, t{=}0 \,) = {} & \overset{P_{00}}{\cancel{P_1(0)}}\ \overset{1}{\cancel{B_0(0)}} + \overset{P_{00}}{\cancel{P_4(0)}}\ \overset{1}{\cancel{B_0(0)}} \\
& + \overset{P_{10}}{\cancel{P_3(t)}}\ \overset{0}{\cancel{B_1(s)}} + \overset{P_{01}}{\cancel{P_2(0)}}\ \overset{0}{\cancel{B_1(0)}} \\
& - P_{00}\ \overset{1}{\cancel{B_0(0)}}\ \overset{1}{\cancel{B_0(0)}} \\
& - P_{01}\ \overset{0}{\cancel{B_1(0)}}\ \overset{1}{\cancel{B_0(0)}} \\
& - P_{11}\ \overset{0}{\cancel{B_1(0)}}\ \overset{0}{\cancel{B_1(0)}} \\
& - P_{10}\ \overset{1}{\cancel{B_0(0)}}\ \overset{0}{\cancel{B_1(0)}}
\end{aligned}$$

$B_0(x) = 1 - x$

$B_1(x) = x$

$$\begin{aligned}
P(\,0, 0\,) \quad & = P_{00} + P_{00} - P_{00} \\
& = P_{00} \text{ as expected.}
\end{aligned}$$

Note that each corner vertex belongs to two boundary curves and will therefore be added twice when s and t are 0 or 1. This is the reason for the last four terms involving the corner vertices in the P(s,t) equation.

As a second check, evaluate P(s, 0):

Boundaries :

$b(s,t)$

$s = 0,\ t = 0 \rightarrow 1$

$s = 1,\ t = 0 \rightarrow 1$

$t = 0,\ s = 0 \rightarrow 1$

$t = 1,\ s = 0 \rightarrow 1$

$$\begin{aligned}
P(\, s,\, t{=}0 \,) = {} & \overset{P_{00}}{\cancel{P_1(0)}}\ B_0(s) + P_4(s)\ \overset{1}{\cancel{B_0(0)}} \\
& + \overset{P_{10}}{\cancel{P_3(0)}}\ B_1(s) + P_2(s)\ \overset{0}{\cancel{B_1(0)}} \\
& - P_{00}\ \overset{1}{\cancel{B_0(0)}}\ B_0(s) \\
& - P_{01}\ \overset{0}{\cancel{B_1(0)}}\ B_0(s) \\
& - P_{11}\ \overset{0}{\cancel{B_1(0)}}\ B_1(s) \\
& - P_{10}\ \overset{1}{\cancel{B_0(0)}}\ B_1(s)
\end{aligned}$$

$$\begin{aligned}
P(\, s, t{=}0 \,) \quad & = P_{00}\, B_0(s) + P_4(s) + P_{10}\, B_1(s) - P_{00}\ B_0(s) - P_{10}\ B_1(s) \\
& = P_4(s) \text{ as expected}
\end{aligned}$$

Any four space curves (parameterized from 0 to 1) may be used to construct a surface. If $P_2(s)$ and $P_4(s)$ are linear, the equations reduce to a ruled surface.

Ray Cast :

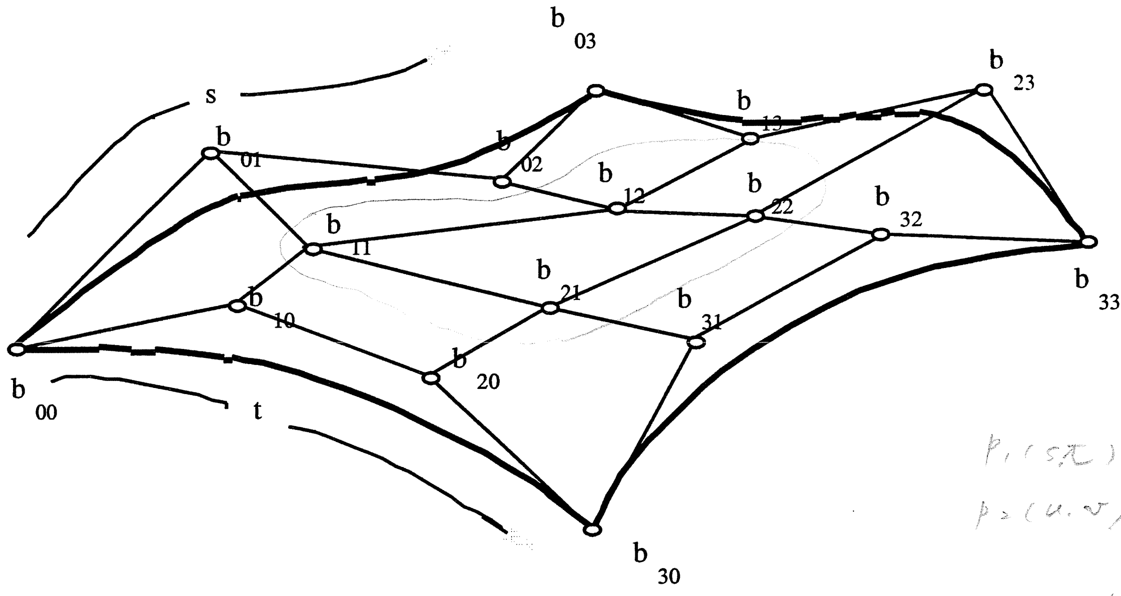
$\begin{cases} P + tV \\ p(u, v) \end{cases} \rightsquigarrow$ slow

15.3 Bézier Surfaces

The Bézier surface formulation is a Cartesian product surface. The general "n by m" surface is formulated as follows:

$$P(s, t) = \sum_{i=0}^{n} \sum_{i=0}^{m} B_{n,i}(s) B_{m,j}(t) b_{ij}$$

where $B_{n,i}(x)$ is the Bernstein basis function defined in the previous chapter, 'n' is the degree of the surface in the 's' direction, and 'm' is the degree of the surface in the 't' direction. The b_{ij} are (n + 1) * (m + 1) control points that form a *control net*. (Figure 15-5)



$P_1(s,t)$

$P_2(u,v)$

Figure 15-5. Bézier Surface Control Net.

$P_2'(x,y,z) = 0$

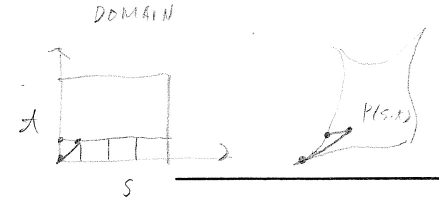
The bi-cubic surface is shown below in blending function format.

$x(s,t)$

$$P(s, t) = \left[(1-t)^3, 3t(1-t)^2, 3t^2(1-t), t^3\right] \left[b\right] \begin{bmatrix} (1-s)^3 \\ 3s(1-s)^2 \\ 3s^2(1-s) \\ s^3 \end{bmatrix}$$

The parameter functions can be factored into two matrices as before:

$$P(s, t) = \mathbf{T}\mathbf{M}_{be}\left[b\right]\mathbf{M}_{be}^{T}\mathbf{S}^{T}$$



$\vec{N} = \vec{P_s} \times \vec{P_t}$

where [b] is a tensor (matrix of vectors) that contains the points on the control net, and T and S are again polynomial basis vectors and $\mathbf{M}_{be}$ is the Bézier basis change matrix as discussed in the previous chapter.

Like the Bézier curve, the Bézier surface closely mimics its control net. The surface contains only the corner points and the three control points nearest each vertex control the *surface tangents*. Parametric surface tangents are computed by partial derivatives in the 's' and 't' parametric directions:

$$\frac{\partial}{\partial s}P(s,t) = P_s(s,t) \qquad \frac{\partial}{\partial t}P(s,t) = P_t(s,t)$$

where,

$$P_s(s,t) = \left[x_s(s,t)\ y_s(s,t)\ z_s(s,t)\right] = \left[\frac{\partial}{\partial s}x(s,t)\ \frac{\partial}{\partial s}y(s,t)\ \frac{\partial}{\partial s}z(s,t)\right]$$

Given parametric tangents, we compute <u>geometric tangents</u> using ratios:

$$\frac{dy}{dx}(s,t) = \frac{y_s(s,t)}{x_s(s,t)} \qquad \frac{dz}{dx}(s,t) = \frac{z_s(s,t)}{x_s(s,t)} \qquad \frac{dz}{dy}(s,t) = \frac{z_s(s,t)}{y_s(s,t)}$$

Note that because they are ratios of the parametric tangents, the geometric tangents have an extra degree of freedom (a scale factor).

Surface normals can also be computed for any parametric point on a surface. The normal is defined by the cross product of the parametric tangent vectors:

$$N(s,t) = P_s(s,t) \otimes P_t(s,t)$$

Surface normals are important for a number of computational geometry applications, including machining of surfaces (locating the position of a cutting tool involves moving along the normal by the radius of the cutter), realistic rendering (lighting model calculations), and finding offset surfaces (the surface which is everywhere equidistant from another surface).

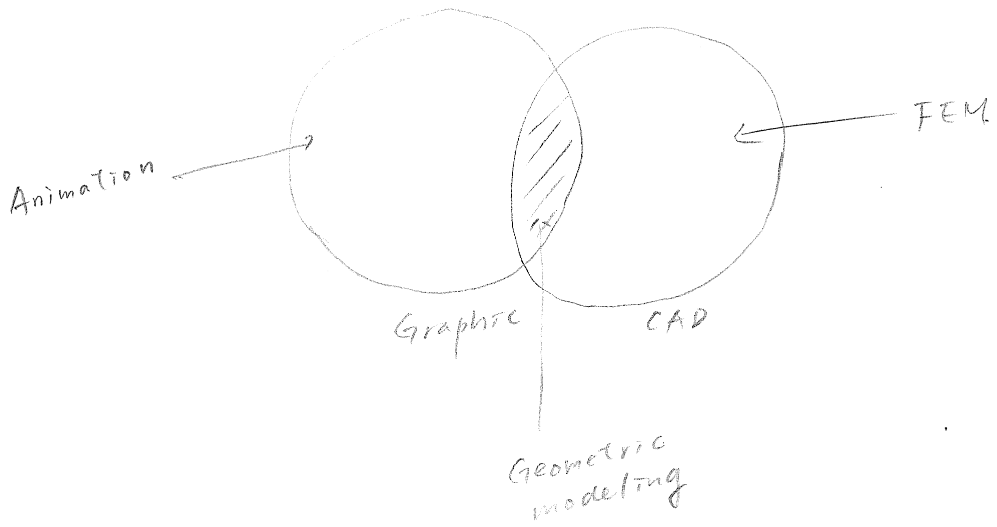
15.4 B-Spline Surfaces

The B-spline curves also generalize to surfaces. Simply use $\mathbf{M}_{bs}$ in place of the $\mathbf{M}_{be}$ in the surface equations and fill $\mathbf{B}_x$, $\mathbf{B}_y$, and $\mathbf{B}_z$ with B-spline control net coordinates.

$$x(s,t) = T\, \mathbf{M}_{bs}\, \mathbf{B}_x\, \mathbf{M}_{bs}{}^T\, S^T$$

$$y(s,t) = T\, \mathbf{M}_{bs}\, \mathbf{B}_y\, \mathbf{M}_{bs}{}^T\, S^T$$

$$z(s,t) = T\, \mathbf{M}_{bs}\, \mathbf{B}_z\, \mathbf{M}_{bs}{}^T\, S^T$$



Chapter 16. Geometric Modeling

The representation and manipulation of geometry is a key ingredient in nearly all computer graphics applications. This is also true in the fields of computer-aided design (CAD) and computer-aided engineering (CAE), in which systems strive to create “software prototypes” of physical objects on which mathematical simulations can be performed to predict the behavior of the objects in real life situations. The current and potential benefits of CAD and CAE systems are improvements in the design and manufacture of new products, where “what if” questions can be answered by computer simulations instead of building and testing costly, time-consuming prototypes.

The technical foundation for CAD systems is the ability to represent the geometry of the parts and assemblies being designed in sufficient detail to allow the computer to assist in design, analysis and manufacture. This field, geometric modeling, has grown with the fields of computer graphics and CAD over the years. A brief chronology of this development is given next.

16.1 Historical View of Geometric Modeling

1950's

Computer graphics first appears as a tool for design and manufacturing in large aircraft and auto industries. Early “research” shows great potential.

1960's

Computer graphics hardware is very costly and the software development effort is enormous. Only large companies can justify the investment.

~1963

Sutherland's “Sketchpad” demonstrates utility of interactive computer graphics for engineering problem-solving in general.

1970's

The appearance of low-cost minicomputers, coupled with new storage tube CRT terminals, spawn 2D "turnkey CAD" systems development. Small companies can now develop software and deliver a system affordable to a larger segment of industries. The CAD industry is born.

~1973

Braid's "Designing with Volumes" thesis and his "BUILD" system demonstrate the potential for solid modeling in CAD. Also, the PADL research project at Rochester establishes the viability of solid modeling.

1970's

Turnkey CAD systems flourish, and add 3D capabilities. A technical debate begins between the drafting versus solid modeling paradigms for CAD. Solid modeling is computationally expensive, making it impractical for most minicomputer systems in place in industry.

early 1980's

New powerful 32-bit "midi"-computers enter the marketplace. Turnkey systems struggle to re-write their systems to exploit the power of these computers. The viability of solid modeling increases, encouraging further development by CAD vendors. Solid modeling also gains support as applications appear in computer-aided manufacturing, CAM, such as computer automated machining planning for numerical control machining (NC). Research efforts at universities and industries in CAD grow considerably.

mid-1980's

Personal computers revolutionize the computer marketplace. PC CAD quickly takes over marketplace, redefining the cost structures and options available to customers, and opening the CAD market to all industries, large and small. The established turnkey market stagnates. Solid modeling has become a part of every large CAD system.

1990's +

Parametric and constraint-based design systems gain attention. Integration of design and manufacturing becomes more important.

16.2 CAD and Geometric Modeling

CAD systems share an obsession with geometry. As CAD systems evolved, so did the treatment of geometry:

2D wireframe models

geometry consists of lines and curves in 2D: (x,y),

objects are created using drafting methods,

3D is simulated using multiple views.

3D wireframe models

like 2D wireframe data, but 3D: (x,y,z),

space curves now possible.

surface models

surfaces with boundary curves between points,

"skin" of aircraft, auto,

NC possible with manual guidance.

solid models

oriented surfaces + *topology,*

"robust" shape description,

complete spatial enumeration,

automation (shape understanding) is aided greatly.

features models

are an extension of solid models to include product data in a form that supports integration among design, analysis and manufacturing programs.

16.3 Solid Geometric Modeling

A serious limitation with wireframe geometry for modeling real objects is that some necessary information is missing. Wireframe and surface representations do not completely and unambiguously define the total geometry. These systems require that the *spatial occupancy* of the object be interpreted by a human. Consider the information necessary to compute the volume of an object stored as a set of polygons. Do the structures discussed so far contain sufficient information? Generally not.

Another problem is that it is possible to accidentally design objects that are physically impossible to build, due to dangling edges and faces, and faces or edges that "pass through themselves." To have some assurance that the resulting geometry represents an object that is physically realizable, a more rigorous way to construct and store 3D geometry is necessary. At the same time, there is interest in developing computer-aided design systems with capabilities for designing in three dimensions with more powerful construction methods than simple points, lines and surfaces.

Solid modeling began in the early 1970's at universities in England and the U.S. The primary goal for solid modeling was, and still is, to develop a complete and unambiguous computer representation for physically realizable objects that can be used for all engineering design, analysis and manufacturing purposes with a high degree of automation. Solid modeling is key to computer-aided design (CAD), computer-aided manufacturing (CAM) and automation.

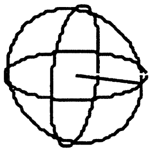
A simple test can be made to assess the robustness and completeness of a geometric model:

Is the point (x,y,z) inside, outside or on the object?

Solid modeling systems can answer this question without human intervention. Can wireframe systems (for example wireframe or surface models)? No.

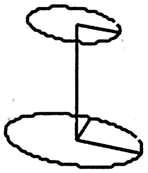
CSG solid modeling was described in section 13.9 on page 239. Solid modeling programs expedite computer aided geometric design by eliminating many of the cumbersome operations necessary to construct models with conventional computer aided drafting systems. Consider the alternative way of constructing the difference of a cylinder and block (Figure 13-31) with traditional drafting approaches: intersecting each surface with each of the others, then deleting unwanted edges, "trimming," i.e. computing and deleting off unwanted pieces of other curves, and stitching together the remaining sections of curves and surfaces.

Intuitively, one realizes that these operations produce valid solids (except for some surface-on-surface situations). Although different modelers use different sets of primitives, the ones generally common to all are the Sphere and Cone, etc. as illustrated in the figures.



object = Sphere(P, radius)

Figure 16-1. Sphere Primitive and Functional Notation.



object = Cone(P, axis, radius1, radius2)

Figure 16-2. Cone Primitive and Functional Notation.

Swept Solids, also called$2\frac{1}{2}$ D solids:

Planar profile projected by *extrusion*: "Slab."

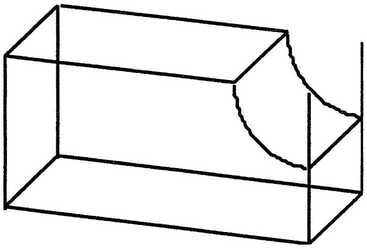


Figure 16-3. Extruded or 2-1/2D Primitive.

Planar profile revolved about an axis: *Surface of revolution.*

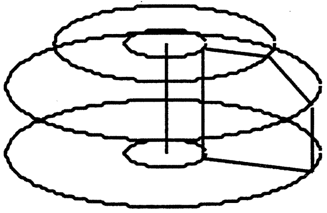


Figure 16-4. Surface of Revolution Primitive.

16.4 Solid Modeling Methods

No solid modeling system today is able to model all manufactured parts exactly. Each system is limited in some way; generally due to its inability to deal with complex geometry, such as curved surfaces. In these cases the system either gives up ("I can't do that") or makes some approximation and continues.

In general, solid modeling methods vary according to how:

1. the model is constructed:

   languages and/or graphical interaction,

   set operations,

   sweeps,

   Euler operations

2. the model is stored (i.e. the internal data structure):

   CSG

   BREP

   Hybrid (combination of CSG and BREP)

We will examine two of the fundamental approaches: *constructive solid geometry* (CSG) and *boundary representation* (BREP).

16.4.1 Constructive Solid Geometry (CSG)

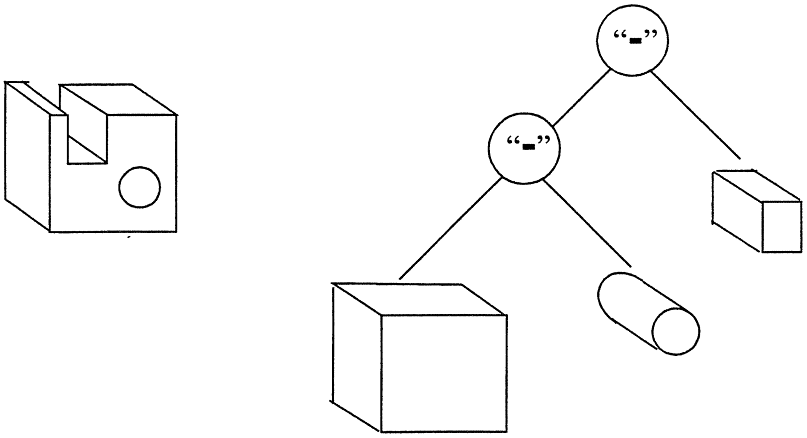
We have already seen an example of CSG, or *primitive instancing*, used as a construction method. There is a natural appeal to combining objects in this way. For example, making a hole, slot, pocket, etc. is made much simpler with CSG operations. CSG also strongly relates to manufacturing operations, another reason for its popularity in CAD and CAM.

Some engineering applications relate to CSG construction methods somewhat naturally, for example:

1. creating a mold for a given part using CSG modeling by "subtracting" the part from a block.
2. simulating machining operations by "subtracting" the cavities made by the tools from the part.

3. checking for interference between parts by computing their Boolean “intersection.”

Objects can be stored in *CSG trees*, also. A CSG tree is a binary tree data structure with Boolean operations at its branches and primitives at the leaves.



Boolean

Figure 16-5. CSG Tree Representation (Right) of an Object (Left).

Stored CSG models are in fact only a definition of “how to make” the model from primitives. This is called an *unevaluated* form, because each application that wants to use the model must evaluate it for its own purposes.

One of the methods used to evaluate a CSG model for realistic display is ray casting as discussed earlier.

16.4.2 Boundary Representation

Probably the most popular method for storing geometric models is boundary representation, or BREP. As the name implies, BREP models maintain a complete description of all the surfaces bounding the object.

A typical BREP stores all the faces (surfaces), edges and vertices on the object in a richly linked data structure. This structure stores:

1. geometry: surface equations, points, curve equations.
2. topology: how the geometry is connected.

    vertices, edges, loops, faces, shells, objects.

The typical structure is called a “face-edge-vertex” graph, or FEV. The best description of this structure is “everything points to everything else.”

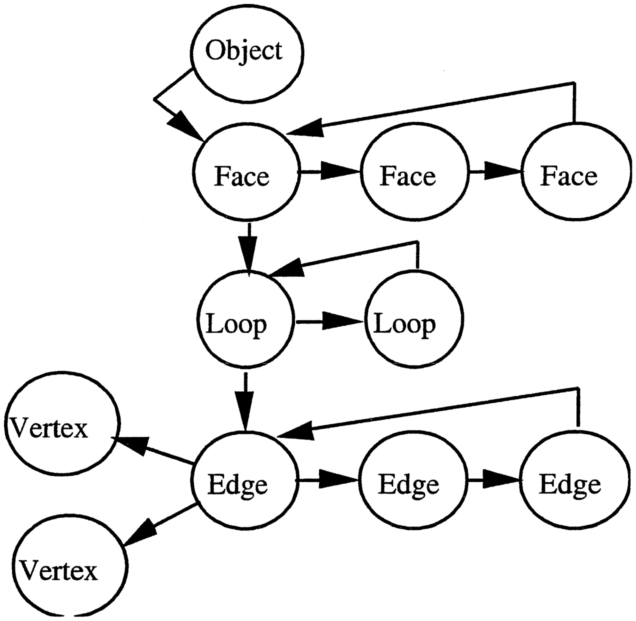


Figure 16-6. Illustration of a BREP Data Structure.

16.4.3 Evaluating CSG Objects: A 2D Example

We can describe enclosed 2D areas with expressions, where the operators are '+' for Boolean addition, '-' for subtraction and '*' for intersection, and the operands are primitive objects, or primitives, such as “CIRCLE(Xc, Yc, Radius)” and “BOX(Xc, Yc, Size).”

For example, the expression:

OBJECT = BOX1(0, 0, 1) + CIRCLE(1, 0, 1) - BOX2(0, 0, 0.5)

creates the OBJECT shown in Figure 16-7.

Letting the operator “P” mean “primitive,” OBJECT could be stored in a binary tree data structure, a “CSG tree,” shown in Figure 16-8.

We can perform computations on a CSG tree to determine if a given point (x,y) is inside or outside the object described by the tree. For our example, let the value TRUE mean “inside or on the object,” and FALSE mean “outside.” We will design a function `TestPoint(x,y,tree)` which will return TRUE (actually the value 1) or FALSE (the value 0) for the CSG tree pointed to by “tree.”

First define two data structures, `CSGTree` and `Primitive`, as illustrated in Figure 16-9.

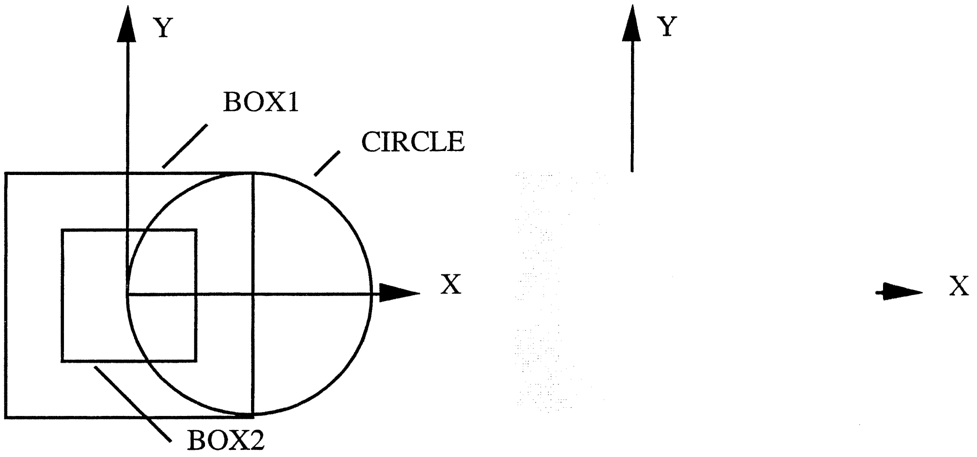


Figure 16-7. Example Object (right) formed by Primitives (left).

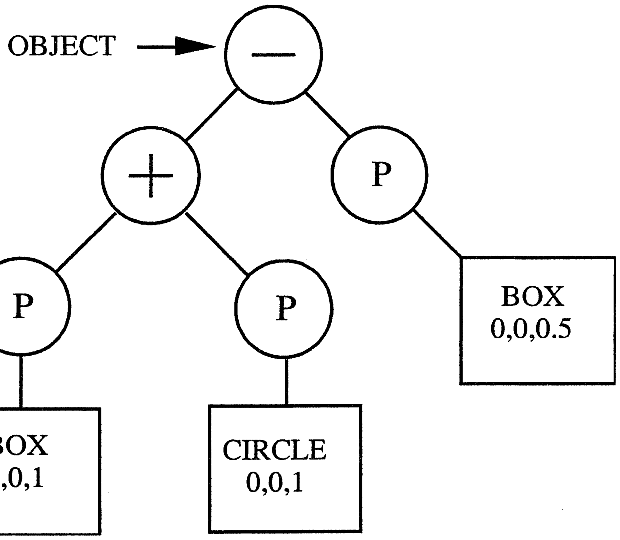


Figure 16-8. CSG Tree Representation of the Object in Figure 16-7.

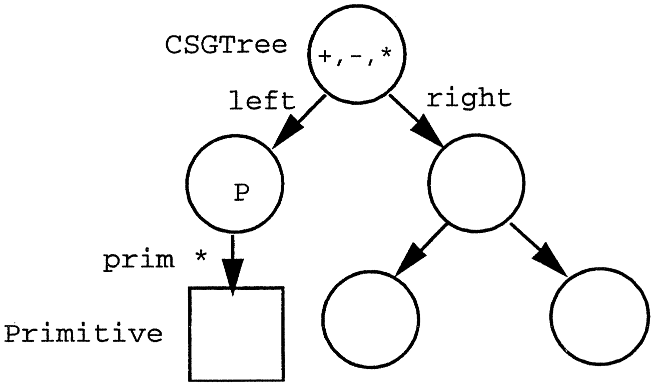


Figure 16-9

C code for the `CSGTree` structure could be:

```
typedef struct CSGTree {
        int operator;
        struct CSGTree *left, *right;
        struct Primitive *prim;
    } CSGTree;
```

and Primitive could be any appropriate data representation for storing the primitives' data.

Define the values for the "operator" field: ADD, SUBTRACT, INTERSECT and PRIMITIVE. The fields "left" and "right" point to sub-trees for the +, - and * binary operators. They are not used for the 'P' operator. The field "prim" points to a structure of type Primitive only if the operator is 'P'.

The function to evaluate if a given point is inside or on a CSG object is coded (in C) using recursion as shown in Figure 16-10.

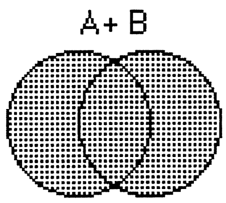
```
Boolean TestPoint( x, y, tree )
      float x, y;
      CSGTree *tree;
{
      switch( tree->operator ) {
            case ADD:
               return( TestPoint(x,y,tree->left) ||
                        TestPoint(x,y,tree->right) );
            case SUBTRACT:
               return( TestPoint(x,y,tree->left) && !
                        TestPoint(x,y,tree->right) );
            case INTERSECT:
               return( TestPoint(x,y,tree->left) &&
                        TestPoint(x,y,tree->right) );
            case PRIMITIVE:
                  if( tree->prim->type == CIRCLE )
                      return( TestCircle(x,y, tree->prim) );
                  if( tree->prim->type == BOX )
                      return( TestBox(x,y, tree->prim) );
      }
}
```

Figure 16-10. CSG Evaluation Function TestPoint.

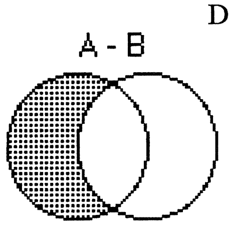
The "primitive routines" TestCircle and TestBox must return TRUE if the given point is inside the particular instance of the primitive, or FALSE otherwise. All calls to TestPoint end up in calls to primitive routines.

Adding new primitives is simply a matter of coding their TestXXX routines.

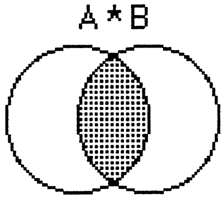
For some applications, such as ray casting CSG trees, two-state logic is inadequate because it is necessary to know when a point is ON the object as well. Figure 16-11 gives the three-state logic tables for the Boolean operators.



			B	
	UNION	IN	ON	OUT
	IN	IN	IN	IN
A	ON	IN	ON	ON
	OUT	IN	ON	OUT



			B	
	DIFFERENCE	IN	ON	OUT
	IN	OUT	ON	IN
A	ON	OUT	ON	ON
	OUT	OUT	OUT	OUT



			B	
	INTERSECTION:	IN	ON	OUT
	IN	IN	ON	OUT
A	ON	ON	ON	OUT
	OUT	OUT	OUT	OUT

Figure 16-11. Three-State CSG Truth Tables.

Index

The GRAFIC Package

Computer Aided Design and Graphics Laboratory

Purdue University

D. C. Anderson

1999 Version

CONTENTS

Overview

GRAFIC is a set of libraries of C and C++ callable routines for programming interactive graphics applications on workstations and personal computers. Versions of the GRAFIC library are available for workstations running X Windows™, Apple Macintosh™ PowerPC personal computers, and Intel compatible personal computers running Microsoft Windows 95 or NT™. Except for minor differences, the same source code will function as expected for these platforms simply by re-compiling and linking with the appropriate GRAFIC library. The GRAFIC package was developed to provide basic, cross-platform, multi-language support for programming window-based raster graphics applications without the complexities and language dependencies typical of window graphics systems. The libraries can be accessed at URL http:www.cadlab.ecn.purdue.edu.

The next sections introduce the basic concepts for GRAFIC programming, followed by descriptions of the routines. Information specific to programming using the Macintosh and Windows versions of GRAFIC is provided in the closing sections.

GRAFIC Windows

When a GRAFIC program is running, the screen usually has several windows. A window is an area of the screen available for drawing lines, text and filled areas. At any time there may be several windows on the screen, some created using a window manager program, some created by one or more GRAFIC programs, and others created by other users and their programs running on the same workstation.

GRAFIC windows have a *title bar*, a thin rectangular area across the top of the window with centered text, called the *window title*, that is specified as an argument in calls to the `GInitGrafic`, `GCreateWindow` and `GSetWTitle` routines. Beneath the title bar is the *menu bar*, another rectangular area across the window in which pull-down menu titles are displayed. Menus are created with `GReadMenu` and `GNewMenu`. Calls to drawing routines produce lines, text, and filled areas in the *drawing area*, the remaining rectangular area below the menu bar. On a Macintosh, the menu bar is not inside the window. Instead, the menu bar for the front window is displayed along the top of the main screen.

All drawing is done in *window coordinates*, which are the coordinates of a screen picture element, or *pixel*, in the drawing area. The coordinates are denoted as (h,v), for horizontal and vertical, to distinguish them from the familiar (x,y). This is mainly to emphasize that v increases downward. The origin (0,0) for window coordinates is the upper-left corner of the window beneath the menu bar (even if no menu bar is created) as shown in Figure 1. Screen coordinates are measured from the screen origin, the upper left corner of the screen. Throughout this document, the term *window coordinates* refers to (h,v) measured with respect to a window origin, and *screen coordinates* refer to (h,v) measured with respect to the screen origin.

There are two ways to create GRAFIC windows. For many programs only one GRAFIC window is needed. This is best accomplished by calling `GInitGrafic` with the window title string as the argument, for example:

```
    GInitGrafic("My Window Title");
```

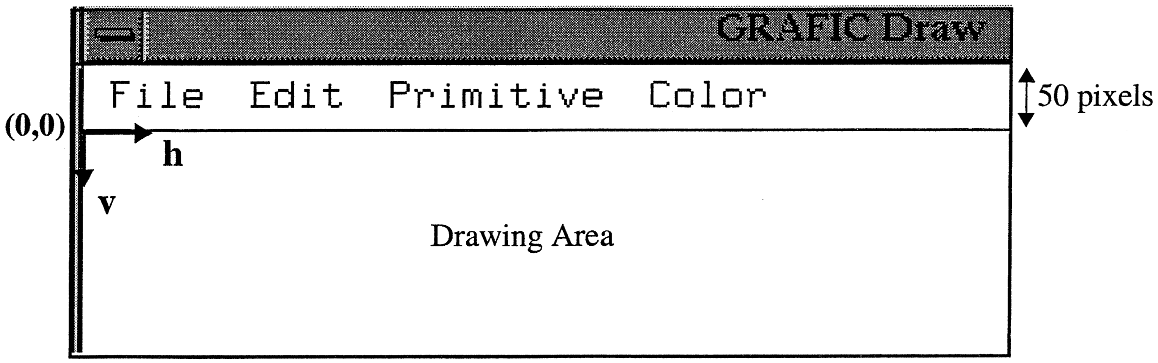


Figure 1. GRAFIC window layout.

Windows can also be created as needed with GCreateWindow, which returns a window identifier, or window ID (abbreviated "WID"), that can be saved and used to identify a window for routines that need a WID as an argument or that return a WID.

A list of all GRAFIC windows created is maintained in a stacking order from front to back like a stack of papers. Windows in front (i.e. on top) overwrite any overlapping areas of those behind them. The *front window*, whose WID is returned by GFrontWindow, is the first GRAFIC window in this list. GSelectWindow makes a window the front window.

All drawing is done in the *drawing window*, which is not necessarily the *front window*. GSetWindow makes a window the drawing window. It is possible to draw in any GRAFIC window and to change the stacking order of windows using calls to GSetWindow and GSelectWindow.

Pixels and Color

The display screen consists of a two dimensional array of pixels with a set number of pixels per *scan-line* (a row of pixels across the screen) and a set number of scan-lines. Typical screens have from 600 to 1100 pixels per scan-line and 400 to 900 scan-lines. The typical *resolution* of a screen is around 80 pixels per inch. The width and height in pixels and millimeters for the screen can be obtained using GGetScreenInfo. It is good practice to use GGetScreenInfo to obtain screen characteristics when creating windows instead of assuming specific screen dimensions.

Displays come in three varieties: black and white (B/W), monochrome (shades of one color), and color. A B/W display, as the name implies, can display pixels in two colors: black and white. This is commonly called a *bitmap* display because each pixel requires only one bit whose value is 1 for black and 0 for white, or vice-versa.

Colors are displayed as additive combinations of three primary components: red, green and blue (abbreviated RGB). Color displays can be *direct color*, where the pixel value contains the three RGB component values, or *mapped color*, where the pixel value is an index into a *color table* that contains the color components. The pixel value for mapped color is an index to a color table entry that contains the color components. To display each pixel on a mapped color display, the display processor uses the pixel value as an index to access the RGB component values from the color table and then creates this color at the pixel location on the screen. The color table is often called a *Video Lookup Table* (VLT) or a *Color Lookup Table* (CLUT). Currently, all pixel values in GRAFIC are color table indices. GRAFIC allows programs to run on B/W displays, and silently maps colors to black and white.

For mapped color, there is one color table that is shared by all users of a workstation, so it is necessary to reserve color table entries for exclusive use if it is necessary to set and change their RGB values. This is done by calling GLoadColor with the desired RGB components. GLoadColor will return a color table index that can be used as the pixel value for the specified color in later calls to GPenColor, GFillEnd and GSetColor. The color remains set in the color table for the duration of program execution. A color table entry that has been reserved with GLoadColor can be changed using GSetColor. All pixels that have been drawn with the pixel value of the changed color table entry will immediately change to the new color without redrawing. It is not possible to predict the value returned by GLoadColor and the values may differ from one execution of the same program to another. GBLACK and GWHITE are pre-defined colors that are represented by macros in the GRAFIC include file "grafic.h".

There is a limited number of color table entries available, typically less than 256, so sometimes it is necessary to "free" reserved entries with GFreeColor. This may be necessary for certain applications and is a courtesy to allow full access to the color table to other users on multi-user workstations. Color table entries are automatically freed when a program terminates.

Events

Your program is notified of user actions while the program is running, such as depressing a mouse button, moving the mouse, picking menus, typing keys, and window updates (exposure of previously covered areas of a window), through events returned by GNextEvent. All events are recorded in a queue – none are lost. This means that it is possible for a user to "get ahead" of a program, that is, have many events queued and waiting to be processed, especially when your program performs lengthy computations between GNextEvent calls.

Events are identified by numbers denoted by symbols (defined in the GRAFIC include file). Each call to GNextEvent returns the event identifier as the function value and from one to four data values that are stored in the variables passed as arguments. A program must first check the event identifier and then branch to the appropriate section of code that interprets the arguments accordingly. If the event queue is empty, GNextEvent returns the event identifier GNULLEVENT.

Menus

GRAFIC provides a simple means for defining and interacting with menus. A menu bar consists of one or more pull-down menus with *menu titles*, each of which contains one or more *menu items*. Each menu item that is enabled can be picked by the user. Items can be enabled and disabled using GEnableMItem.

A menu item is picked by positioning the mouse over a menu title in the menu bar and depressing and holding down any mouse button. This pulls down the menu, displaying the menu items in a list below the menu title. This menu can be released (pulled up) and another menu can be pulled down by dragging the mouse over another menu title with the button still depressed. Dragging the mouse over an enabled menu item causes it to be drawn highlighted, as illustrated in

Figure 2. Releasing the mouse button over an enabled menu item creates a GMENUPICK event. For example, releasing the mouse button with the cursor in menu "Primitive" and menu item "New Oval" as shown in Figure 2 would cause GNextEvent to return a GMENUPICK event with the first three arguments (called wid, a and b in the GNextEvent documentation) assigned the WID of the window containing the menu and the values 3 and 2, which should be interpreted as menu number 3 and item number 2. Menus are numbered left to right starting at 1 and items are numbered from top to bottom starting at 1.

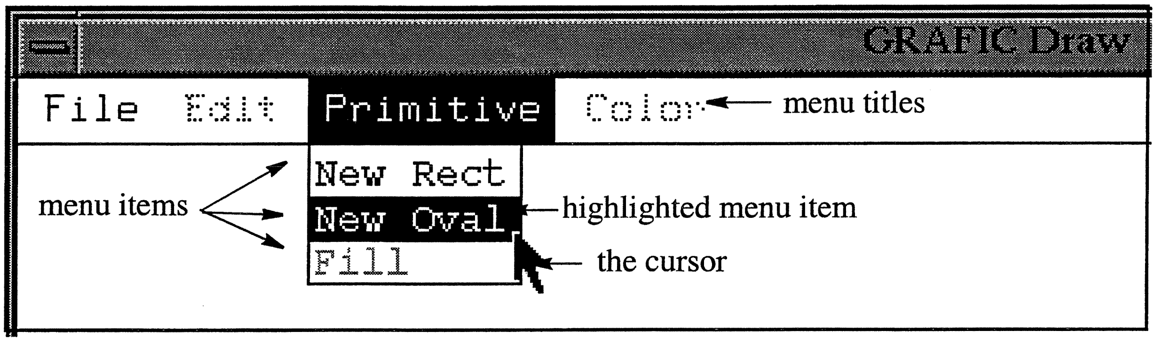
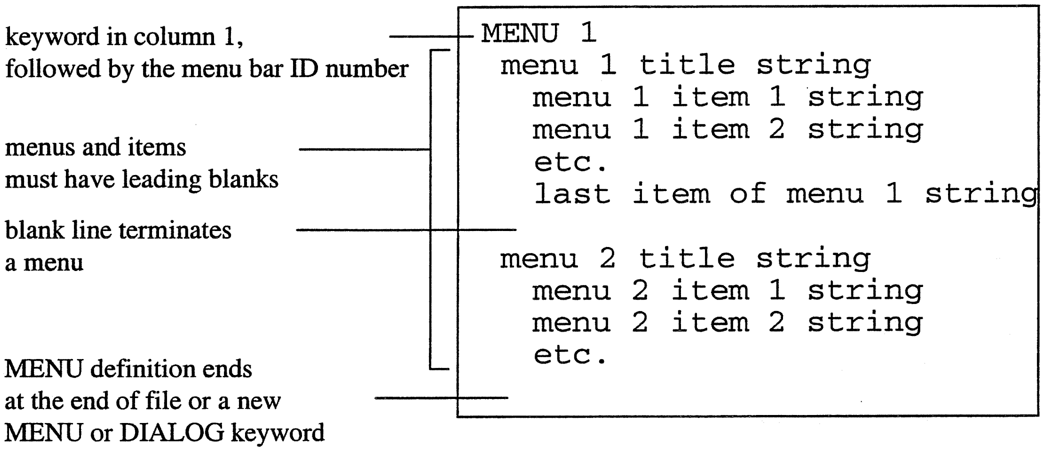


Figure 2. GRAFIC menus.

For X Windows, GRAFIC menus can be defined in a *resource file* that is a text file that lists the menu titles and menu item strings. GResourceFile is called to open the resource file. For the Macintosh and Windows, resources are handled differently and calls to GResourceFile are ignored. See the Macintosh and Windows sections for details. Only one resource file may be open at a time, but several resource files may be used during the execution of a program. GResourceFile silently creates a sorted list of all the menu and dialog resources and their locations in the file for fast access when they are needed.

A menu bar is created in the drawing window with a call to GReadMenu, passing the menu bar ID number as the argument. There can be different menu bars stored in one resource file. However, a program should call GReadMenu only once for each window.

The menu definition resource file format is illustrated in Figure 3.



```
MENU 1
 menu 1 title string
   menu 1 item 1 string
   menu 1 item 2 string
   etc.
   last item of menu 1 string

 menu 2 title string
   menu 2 item 1 string
   menu 2 item 2 string
   etc.
```

Figure 3. Menu resource format.

For example, consider the menus shown in Figure 4.

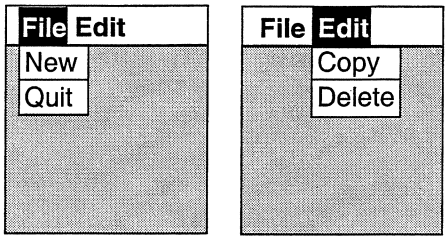


Figure 4. Example menus.

These menus can be created with the resource file “demo.r” shown below.

```
MENU 1
  File
     New
     Quit

  Edit
     Copy
     Delete
```

The following C code creates the menus from “demo.r,”

```
GResourceFile(“demo.r″);
GReadMenu(1);
```

Menus can also be created from strings using GNewMenu. This eliminates the need for a separate resource file but also reduces the convenience of redefining menu strings outside your program.

The following code fragment uses GNewMenu to create the same menus as above:

```
char *menus[2];
menus[0] = “File | New | Quit″;
menus[1] = “Edit | Copy | Delete″;
GNewMenu(2, menus);
```

See the descriptions of GReadMenu and GNewMenu for more details. GNewMenu is not implemented in the Macintosh and Windows versions.

Window Graphics Environment

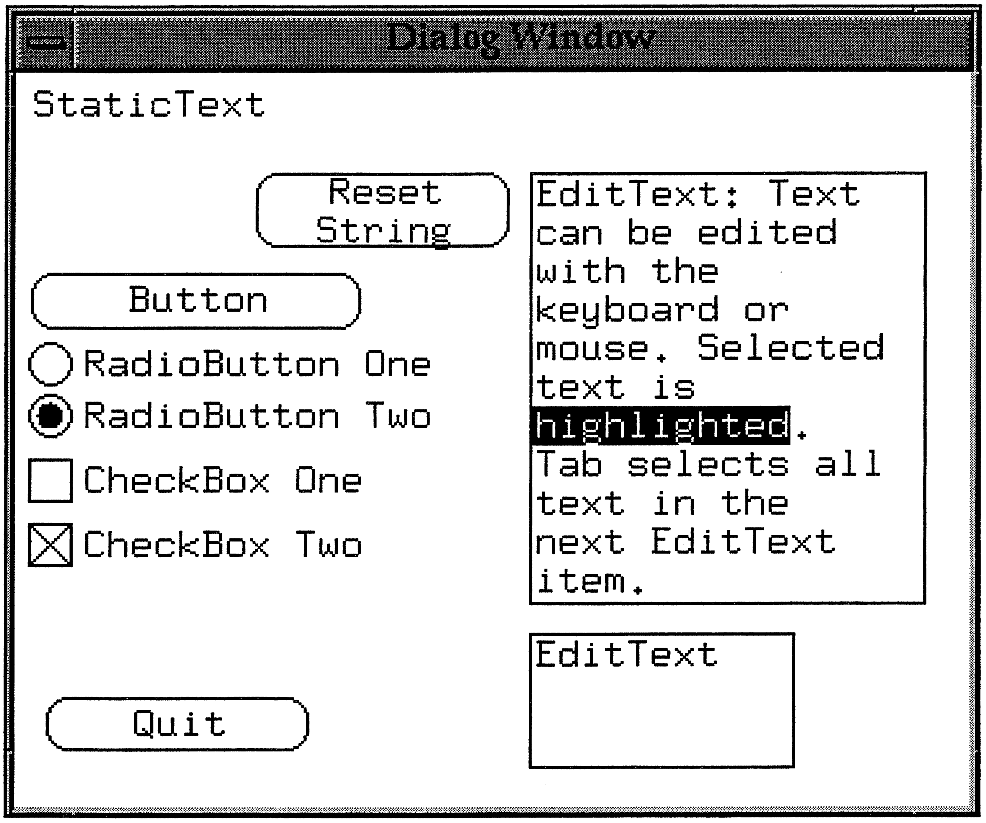
Each GRAFIC window maintains data about its graphics environment as shown below.

Data	GRAFIC routines controlling the data
window location and size	`GSetWindowInfo`
window title	`GSetWTitle`
stacking order	`GSelectWindow`
window cursor	`GSetCursor`
menu bar	`GReadMenu, GNewMenu`
pen position	`GMoveTo, GLineTo`
pen color	`GPenColor`
pen size	`GPenSize`
pen mode	`GPenMode`

When changing to a different drawing window via `GSetWindow`, the graphics environment of that window is recalled by GRAFIC. For example, one can be drawing in different colors and different pen modes in two windows, and switch back and forth as needed without repeatedly calling `GPenColor` and `GPenMode`. Also, as the cursor moves from one window to another, it will change to the set cursor for each window.

Dialogs

A *dialog* consists of a dialog window and dialog items inside the window. The dialog can display uneditable text with *StaticText* items and allow the user to interact with data through *Button*, *RadioButton*, *CheckBox*, and *EditText* items. Once a dialog is displayed, it receives and processes all events until terminated (disposed). The dialog in Figure 5 illustrates each type of item.



EditText: Text can be edited with the keyboard or mouse. Selected text is highlighted. Tab selects all text in the next EditText item.

EditText

Figure 5. GRAFIC dialog window and dialog items.

For X Windows, dialogs are defined as DIALOG resources in a resource file. For Macintosh and Windows GRAFIC, dialogs are created graphically and stored in the ".rsrc" or ".rc" file, respectively. GGetDialog creates a dialog from the resource file given the resource ID and displays it on the screen. Alternatively, GNewDialog allows C and C++ programs to create a dialog from initialized data structures. It is possible to invoke a dialog when another is displayed, for example, to prompt the user "Is it ok to delete the file named in the first dialog?" GDisposeDialog closes and deletes the active dialog. The previous dialog, if any, then becomes the active dialog again. All dialogs must be disposed before events can be processed by GNextEvent.

The format of a dialog in a resource file is as follows:

```
DIALOG ID
        title
        left top width height
        number_of_items

        item 1 data

        item 2 data
        ...
```

ID is the integer corresponding to the argument to GGetDialog that identifies the dialog. Title is a single line of text that is displayed as the title of the dialog window. Leading white space (blanks and tabs) is removed. A blank line can be used for a blank title. Left top width height are the screen dimensions and the coordinates of the upper left corner of the dialog on the screen. On the next line, the integer number_of_items is the number of dialog items that follow. Following this are the item definitions, where each item consists of four lines and items are separated by blank lines:

```
type
text
left top width height
setting
```

Type is one of the following words, independent of case: Button, RadioButton, CheckBox, StaticText, or EditText. Text is a single line of text that is displayed in the item with leading white space removed. Left top width height are the window coordinates and dimensions of the rectangular bounds of the item relative to the dialog window origin. Setting is one or two words specifying the initial state of the item. The first word is either ENABLED or DISABLED and tells GRAFIC if GModalDialog should return the item's number during user interaction as described later. For RadioButton and CheckBox items, an additional word, ON or OFF, specifies the item setting.

The text associated with an item is returned by GetDItemText. Item text can be changed with SetDItemText. For EditText and StaticText items, the associated text is the text string displayed for the item. For Button, RadioButton and CheckBox items, the associated text is the label displayed with the item icon.

For RadioButton and CheckBox items, the state of the item can be set and retrieved with GSetDItemValue and GGetDItemValue.

The following paragraphs describe each item type.

Button

A Button is an oval shaped item with centered text inside. When any mouse button is pressed inside the bounds of the item, the oval and text are shown highlighted. The size of a button must be greater than 16 pixels in both dimensions.

RadioButton

A RadioButton has a circular icon in the top left corner of its bounds. The text is displayed to the right of the icon. When the button is released inside a RadioButton item, the item toggles state (i.e., its value changes from on to off, or off to on) and the corresponding icon is displayed. RadioButtons are typically used for groups of options where only a single option may be on. The user must insure that the proper RadioButtons are on or off using the routines `GGetDItemValue` and `GSetDItemValue`.

CheckBox

A CheckBox is similar to the RadioButton except that the icons are small squares. CheckBoxes are typically used for groups of options that can have several on or off at the same time. The routine `GGetDItemValue` and `GSetDItemValue` are used to get and set CheckBox values.

StaticText

A StaticText item displays its text left justified inside its bounding rectangle. The item number will be returned if the item is enabled and a mouse button is pressed inside its bounds, but no highlighting occurs.

EditText

A EditText item allows the user to interactively enter and edit text. A rectangle is drawn around the bounds. In the case of multiple EditText items in a dialog, one will be the current EditText item that shows the text cursor or highlighted text and will be affected by typed keys. Any of the EditText items can be made the current one by clicking any mouse button inside it. A cursor will be drawn marking the insertion point for new text. The location of the insertion point can also be changed by clicking a mouse button at the new location. Typing a backspace key will delete the character to the left of the insertion point. Text can also be selected by dragging the mouse over it with a button depressed. Selected text is highlighted. The selected text can then be deleted by typing a backspace key, or it will be replaced by any typed character.

Part or all of an EditText item's text can be selected using `GSelDItemText`. Two integer values are passed that identify character locations within the text. The first value is the character position of the start of the selected text (1 for the beginning). The second value is the index of the character at the end of the selected text, not including that character. If the two values are equal, that location will be made the insertion point. Values larger than the text length will be set to the actual text length. `GSelDItemText` also makes the item the current EditText item.

Once a dialog is displayed using `GGetDialog` or `GNewDialog`, it receives all events, and events for other windows are discarded. `GModalDialog` should be called in a loop to handle events for the dialog. `GModalDialog` will process all drawing and interaction for the dialog, including highlighting and text operations. `GModalDialog` returns only after a mouse button is released inside an enabled dialog item or a key is typed in an enabled EditText item. The return value is the item number of the affected dialog item. Items are numbered in the order they appear in the dialog item list in the resource file, starting with one. Also, when a return (or enter) key is typed, `GModalDialog` will return a 1 if the first item is enabled. Therefore, the first item should always be the default button, typically the "OK" or "Cancel" button.

Routines

The GRAFIC routines are described below in alphabetical order.

Names in all capital letters in the code font, such as `GBUTTONDOWN`, are macros defined in the include file "`grafic.h`" .

Certain arguments have universal meanings:

Argument	Meaning
`pixel`	an integer specifying the pixel value that identifies a color (usually a color table index). `GBLACK` and `GWHITE` are pre-defined macros that may be constants or function calls.
`h,v`	integer horizontal and vertical window or screen coordinates in pixels.
`left, top, width, height`	integer coordinates and dimensions in pixels. Rectangular screen areas are always given in terms of the upper-left corner (`left, top`) and the `width` and `height` .
`string`	Standard null-terminated string conventions are followed.
`wid`	a window id, type WID.

GCreateWindow

```
WID GCreateWindow(const char *title, int left, int top,
          int width, int height)
```

Purpose: Create a new window with the upper-left corner at `(left,top)` on the screen and with the given `title`. The dimensions `width` and `height` are the size of the drawing area. The window automatically becomes the front window and the drawing window. The returned value is the `WID` that can be saved for later references to the window.

Arguments: `title` title string for the window.

`left,top`
screen coordinates of upper-left corner of the drawing area.

`width,height`
width and height of the drawing area in pixels.

Notes:

- In X GRAFIC, anything drawn before the first `GUPDATE` event may be lost.

GDisposeDialog

```
void  GDisposeDialog( void )
```

Purpose: Erase the dialog window and free all memory associated with the active dialog.

Notes:

- After `GDisposeDialog`, the active dialog will be set to the previous dialog if there was one.

- Dialogs must be disposed before events can be processed by `GNextEvent`.

GDisposeWindow

`void GDisposeWindow( WID wid )`

Purpose: Delete the given window.

Arguments: `wid` the identifier of the window to delete.

Notes:
- After calling `GDisposeWindow`, `wid` is invalid.

GEnableDItem

`void GEnableDItem( int item, int setting )`

Purpose: Enable or disable the given dialog item number from being returned by `GModalDialog`.

Arguments:

`item`	the item number in the dialog. Items are numbered from 1 in the order they appear in the `DIALOG` resource.
`setting`	A value or an expression that evaluates to 0 to disable or non-zero to enable.

GEnableMItem

`void GEnableMItem( int menu, int item, int setting )`

Purpose: Enable or disable a menu item. A disabled item will appear gray and cannot be picked by the user.

Arguments:

`menu`	the menu number. The first menu is number 1.
`item`	the item number. The first item is number 1.
`setting`	A value or an expression that evaluates to 0 to disable or non-zero to enable.

Notes:
- `menu` and `item` belong to the menu bar of the drawing window.

GFillBegin

`void GFillBegin(void)`

Purpose: Begin the definition of a filled area. The area is created by calls to routines `GMoveTo` and `GLineTo` and can be self-intersecting. After the area has been defined, it is filled by calling `GFillEnd`. The only GRAFIC calls permitted between calls to

GFillBegin and GFillEnd are GMoveTo and GLineTo.

GFillEnd

void GFillEnd(void)

Purpose: Terminate filled area definition and fill the interior with pixels of the current pen color and pen mode.

Notes:

- The area must be closed (last point identical to the first).
- Areas with interior cavities (holes) are created by drawing the exterior boundary, then moving to an interior point with GMoveTo and drawing the interior boundary.
- In Windows GRAFIC, interior holes are not allowed.

GFillOval

void GFillOval(int left, int top, int width, int height)

Purpose: Fill an oval with the current pen color using the current pen mode.

Arguments: left, top the window coordinates of the upper left corner of the rectangle that bounds the oval.

width
height the dimensions of the rectangle in pixels.

GFillRect

void GFillRect(int left, int top, int width, int height)

Purpose: Fill a rectangle with the current pen color using the current pen mode.

Arguments: left, top the window coordinates of the upper left corner of the rectangle.

width
height the dimensions of the rectangle in pixels.

Notes:

- The rectangle extends to the right and down from (left,top).
- GFillRect is faster than using GFillBegin and GFillEnd to fill a rectangle.

Example: The following routine erases (fills with GWHITE) an entire window:

```
void EraseWindow( WID wid )
{
   int left, top, width, height;
   GSetWindow( wid );
   GGetWindowInfo( wid, &left, &top, &width, &height );
   GPenColor( GWHITE );
   GFillRect( 0, 0, width, height );
}
```

GFreeColor

`void GFreeColor( TPixel pixel )`

Purpose: Free a color table entry previously reserved by a call to `GLoadColor`. After this call, the color table entry will be available for future calls to `GLoadColor`.

Arguments: `pixel` the color table index of the entry.

Notes:

- `pixel` must correspond to a pixel value returned by `GLoadColor`.
- `GFreeColor` does nothing on B/W and direct color displays.
- `GFreeColor` is needed in rare cases when a program must repeatedly reserve and free color table entries.
- Color table entries are automatically freed when a program terminates.

GFrontWindow

`WID GFrontWindow( void )`

Purpose: Return the `WID` of the front window.

Notes:

- Zero (`NULL`) is returned if there are no windows.
- GRAFIC knows only about the windows created by your program.

GGetDialog

`void  GGetDialog( int id )`

Purpose: Read and begin execution of a dialog from the current resource file. The dialog may not appear until `GModalDialog` is called.

Arguments: `id` the dialog identifier number corresponding to the "`DIALOG ID`" line in the resource file or the DIALOG resource number in a Windows or Macintosh resource file.

GGetDItemText

void GGetDItemText(int item, char* text)

Purpose: Return the text contained in the given dialog item.

Arguments:

`item`	the item number.
`text`	variable address to receive the item text.

Notes:

- `text` must be able to hold at least `GMAX_STRING`+1 characters.
- The displayed text string will be returned for EditText and StaticText items. For Button, RadioButton and CheckBox items, the label text of the item will be returned.

Example:

The following code illustrates how to convert text in a dialog item into a real number.

```
float GetDItemNumber( int item )
{
   float number;
   char text[128];
   GGetDItemText( item, text );
   if( 1 != sscanf( text, "%f", &number ) )
            -- the item does not contain a valid number --
   return number;
}
```

GGetDItemValue

int GGetDItemValue(int item)

Purpose: Return the value (0 is off, 1 is on) of the given RadioButton or CheckBox dialog item.

Arguments: `item` the item number in the dialog.

GGetScreenInfo

```
void GetScreenInfo( int *wpixels, int *htpixels,
                    int *wmm, int *htmm, int *planes)
```

Purpose: Return the screen dimensions and number of bits per pixel.

Arguments: `wpixels`,
`htpixels` variable addresses for the width and height of the screen in pixels.
wmm,

`htmm` variable addresses for the width and height of the screen in millimeters.

`planes` variable address for the number of bits per pixel of the screen. Usually, this will be 1 on B/W displays, 4 or 8 on mapped displays with color lookup tables, and 16, 24, or 32 on direct color displays.

GGetWindowInfo

```
void GGetWindowInfo( WID wid, int *left, int *top, int *width,
        int *height )
```

Purpose: Return the current size and location of the given window.

Arguments: `wid` the GRAFIC `WID`.

`left,top` variable addresses for the screen coordinates of the window drawing area.

`width,`
`height` variable addresses for the dimensions of the window drawing area in pixels.

Notes:
- `GFrontWindow` can be used to obtain the `WID`.

Arguments: `wid` The `WID` of the window.

GInitGrafic

```
void GInitGrafic( char *title )
```

Purpose: Initialize the GRAFIC package and create a window.

Arguments: `title` the string for the window title. If it is `NULL` (value 0) or "" (null string), no window will be created.

Notes:
- This must be the first GRAFIC routine called and in main().
- In X GRAFIC, no drawing should be done before the first `GUPDATE` event for the window.

GLineTo

```
void GLineTo( int h, int v )
```

Purpose: Draw a line from the current pen location to `(h,v)`, or add a line to the definition of a filled area.

Arguments: `h, v` the window coordinates of the end point of the line.

GLoadColor

TPixel GLoadColor(unsigned int r, unsigned int g, unsigned int b)

Purpose: Create a color entry in the color table and return its index (the pixel value) for future GPenColor calls.

Arguments: r, g, b the color components. Components are integers between 0 and 65535.

Notes:

- Returns 0 if a color table entry cannot be reserved. Note that 0 is usually GBLACK or GWHITE, so your program can interpret a 0 as failure.
- In Windows GRAFIC, for direct displays, an RGB pixel value is returned.

GModalDialog

int GModalDialog(void)

Purpose: Process events for the active dialog window and return the item number of the enabled item affected by releasing a mouse button inside its bounding rectangle or typing a character when there is an enabled EditText item.

Example:

The following C code fragment illustrates the typical "dialog loop" using GModalDialog. Assume that the dialog item list consists of an OK button, a Cancel button, and an EditText item. The constants OKITEM=1, CANCELITEM=2, TEXTITEM=3 make the code more readable.

```
GGetDialog( 1 );
/* Use GSetDItemText & GSetDItemValue calls here to initialize
   dialog items according to current program values */
do {

   item = GModalDialog();
   /* do checks here on current item values/text*/
} while( item > CANCELITEM );
if( item == OKITEM ) {
   /* Use GGetDItemText & GGetDItemValue calls here to get text &
   values from items and set program values */
}
GDisposeDialog();
```

GMoveTo

void GMoveTo(int h, int v)

Purpose: Move the pen location or add a vertex to the definition of a filled area.

Arguments: h, v the window coordinates of the pen location.

GNewDialog

`void GNewDialog( GDialogData* data )`

Purpose: Create a new dialog from stored data structures (X Windows only).

Arguments: `data` the address of `GDialogData` data initialized using the `DIALOG` and `DIALOGITEM` C pre-processor macros.

The general format of a dialog data initialization contains a `GDialogItemData` array variable containing `DIALOGITEM` macros followed by a `DIALOG` macro that references the `GDialogItemData` variable. There can be many such initializations, but remember that the `GDialogItemData` variable and `GDialogData` names must be distinct. The item types for the `DIALOGITEM` macro are defined in the include file grafic.h.

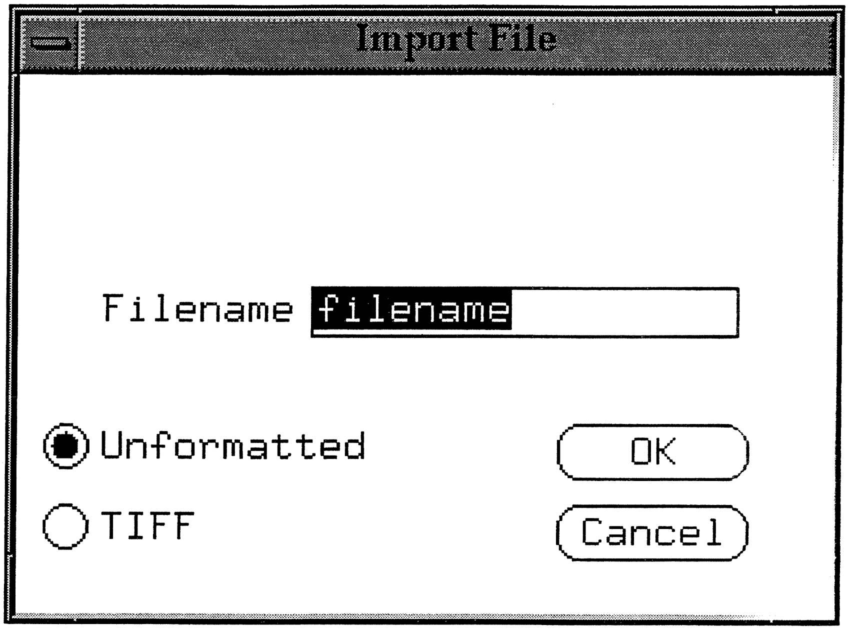
```
GDialogItemData itemlabel[] = {
DIALOGITEM(type,text,left,top,width,height,enabled,setting),
<as many DIALOGITEM statements as items in the dialog>
};
DIALOG(dialogname,title,left,top,width,height,itemlabel)

/* later in the code: */
GNewDialog( &dialogname );
```

Example:

The following data initialization statements illustrate how to create the dialog shown in the illustration.

```
GDialogItemData itemlist1[] = {
DIALOGITEM(GButton,"OK",200,130,70,20,1,0),
DIALOGITEM(GButton,"Cancel",200,160,70,20,1,0),
DIALOGITEM(GEditText,"",110,80,155,16,0,0),
DIALOGITEM(GRadioButton,"Unformatted",10,130,120,16,1,1),
DIALOGITEM(GRadioButton,"TIFF format",10,160,120,16,1,0),
DIALOGITEM(GStaticText,"Filename",30,80,75,16,0,0),
};
DIALOG(dialog1,"Import File",500,100,300,200,itemlist1)
GNewDialog( &dialog1 );
```



Filename filename

● Unformatted

○ TIFF

GNewMenu

void GNewMenu(int nmenu, char *menus[])

Purpose: Create and install menus from the given strings into the drawing window (X Windows only).

Arguments: `nmenu` The number of strings in `menus`.

`menus` An array of strings addresses, each of which will become a separate menu. Each string is of the form:
"menu-title | item1 | item2 | ... | last-item"
The vertical bar ("|") separates the title and items, which may have embedded blanks. Items may be specified as disabled by prefixing them with a left parenthesis ("(").

Notes:

- Only in X Windows GRAFIC.

GNextEvent

int GNextEvent(WID *wid, int *a, int *b, int *c)

Purpose: Return information about the next event in the arguments and return the event identifier as the function value.

Arguments: `wid` address for the `WID` pertinent for the event .

`a, b, c` addresses for the returned event-specific information.

The following table lists the contents of the a,b,c arguments for each event:

Event	**wid**	**a**	**b**	**c**
GBUTTONDOWN	wid	h	v	button & key data
GBUTTONUP	wid	h	v	button & key data
GMOUSEMOTION	wid	h	v	button & key data
GKEYPRESS	wid	char		
GMENUPICK	wid	menu	item	
GENTERWINDOW	wid			
GLEAVEWINDOW	wid			
GNULLEVENT				
GUPDATE	wid			
GWMCOMMAND	wid	command		

Notes:

- GWMCOMMAND 'a' values are the window manager command: GWMQUIT, for quitting an application, and GWMCLOSE, for closing a window. These commands result from window manager actions on the window.
- "Button and key data" consists of combinations of the button constants GLEFTBUTTON, GMIDDLEBUTTON, GRIGHTBUTTON and the key constants GSHIFTKEY and GCONTROLKEY. For example, depressing the left mouse button with the shift key depressed creates a GBUTTONDOWN event with the 'c' value of (GLEFTBUTTON+GSHIFTKEY). To detect this case, the code could be:

```
if((c & GSHIFTKEY) != 0 && (c & GLEFTBUTTON) != 0)...
```

- GMOUSEMOTION events are caused whenever the mouse moves in a GRAFIC window.
- GUPDATE events are created whenever a portion of a window is uncovered and therefore needs to be redrawn by your program. This can happen as a result of creating, disposing, resizing or moving windows.
- GENTERWINDOW and GLEAVEWINDOW events are generated when the mouse moves across window drawing area boundaries, including moving between the menu bar and the drawing area.
- GNULLEVENT events are returned if no other event is in the queue.

GOval

void GOval(int left, int top, int width, int height)

Purpose: Draw the outline of an oval tangent to a rectangle with the current pen color using the current pen mode.

Arguments: left,top the window coordinates of the upper left corner of the rectangle that bounds the oval.

width, height
the dimensions of the rectangle in pixels.

GPenColor

void GPenColor(TPixel pixel)

Purpose: Set the color for future drawing.

Arguments: `pixel` the pixel value.

Notes:

- `pixel` is the pixel value returned by `GLoadColor` or the pre-defined GRAFIC color `GBLACK` or `GWHITE`.

GPenMode

void GPenMode(int mode)

Purpose: Set the pen drawing mode for future lines, text and filled areas.

Arguments: `mode` the drawing mode, either `GCOPY`, `GINVERT` or `GXOR`.

Notes:

- `GCOPY` copies the current pen pixel value into the screen pixels that would be affected by the drawing operation (text, line or fill). This is the most common drawing mode, and is the initial drawing mode after calling `GInitGrafic`.
- `GINVERT` inverts (0 to 1,1 to 0) each bit in the pixel values of the screen pixels that would be affected by the drawing operation (text, line or fill). The current pen pixel value is ignored. This is most useful for creating "reverse video" effects.
- `GXOR` performs a Boolean exclusive-or of the pixel values of the screen pixels that would be affected by the drawing operation (text, line or fill) and the pen pixel value. GXOR'ing the same pixel locations twice restores the original screen image. This is most useful for dragging objects across the screen without having to re-draw the underlying image.
- Note that the screen pixel values resulting from `GINVERT` and `GXOR` may refer to color table entries that have not been set by your program.
- `GXOR` is not supported on all workstations. For example, some models of Silicon Graphics IRIS™ workstations do not support `GXOR`, and the resulting colors are unspecified.

GPenSize

void GPenSize(int size)

Purpose: Set the pen size for future `GLineTo` and `GOval` calls in the drawing window.

Arguments: `size` the pen size in pixels.

Notes:

- In X Windows and Windows GRAFIC, the pixels are approximately centered around the addressed pixel. In the Macintosh, they extend to the right and below the addressed pixel.

- Pen size 0 is a single-pixel pen.
- There is a speed penalty for pen sizes greater than 0.

GPostEvent

`void GPostEvent( int event, WID wid, int a, int b, int c)`

Purpose: Place the given `event` with its parameters at the end of the event queue to be returned by a future call to `GNextEvent`.

Arguments:

`event`	the event identifier.
`wid`	the window id for the event.
`a, b, c`	the appropriate parameters for the event. See `GNextEvent` for descriptions of events and their parameters.

Notes:

- Posting `GUPDATE` events is a useful way of causing a re-draw of a window.

GReadMenu

`void GReadMenu( int id )`

Purpose: Read the given menu bar from the current resource file and install it in the current drawing window.

Arguments:

`id`	the menu bar id number used in the resource file in the “`MENU ID`” line or the MENU number in a Windows and Macintosh resource file.

Notes:

- There is only one menu bar per window. An existing menu will be replaced.
- The menu bar may not appear until the next call to `GNextEvent`.

GResourceFile

`void GResourceFile( char *fileName )`

Purpose: Ready the given file for future `GReadMenu` and `GGetDialog` calls.

Arguments: `fileName` the Unix file name of the resource file.

Notes:

- In Macintosh and Windows GRAFIC, `GResourceFile` is ignored.
- `GResourceFile` should be called after `GInitGrafic` and before any `GReadMenu` or `GGetDialog` calls.

GSelDItemText

void GSelDItemText(int item, int start, int end)

Purpose: Set item number `item` in the active dialog to be the active EditText item and set the text selection range.

Arguments:

`item`	the item number in the dialog.
`start`	the starting character of selected text or the insertion point.
`end`	the end of selected text, not inclusive.

Notes:

- The first character index is 1.
- If `start` equals `end`, the selection is made the text insertion point, which appears as a vertical bar between characters. Typed characters will be inserted at the bar.
- If `start` is less than `end`, the characters between `start` and `end` are made the selected and highlighted text. A typed character will replace the selected characters.
- If `start` or `end` is greater than the string length, it will be set to the string length. This allows all text to be selected using `GSelDItemText( item, 1, 99 ).`
- Typing the TAB key causes all the text in the next EditText item in the dialog to be selected (in a circular last-to-first manner).

GSelectWindow

void GSelectWindow(WID wid)

Purpose: Raise the given window to make it the front GRAFIC window (if not already).

Arguments:

`wid`	the window identifier of the existing window to be raised.

Notes:

- Only windows created by GRAFIC are considered, so other windows on the screen may overlap the window.
- The window is only raised, it is not made the drawing window.

GSetDItemText

void GSetDItemText(int item, char *text)

Purpose: Replace the text in dialog `item` with the given `text`.

Arguments:

`item`	the item number.
`text`	address of the text string.

Notes:

- `text` should be at most `GMAX_STRING` characters long, including the '\0' terminator character.

- `text` will become the displayed text string for EditText and StaticText items. For Button, RadioButton and CheckBox items, `text` will become the label text of the item.

Example:

The following routines convert a real number into text and place it in a dialog item.

```
void SetDItemNumber( int item, float number )
{
   char text[128];
   sprintf( text, "%f", number );
   GSetDItemText( item, text );
}
```

GSetDItemValue

```
void  GSetDItemValue( int item, int value )
```

Purpose: Set the value of the given Checkbox or RadioButton item in a dialog.

Arguments:

`item`	the item number in the dialog.
`value`	the value or an expression which evaluates to 0 for off, non-zero for on.

Notes:

- Only RadioButton and CheckBox items have values. `GSetDItemValue` should not be called for other types of items.

GSetColor

```
void GSetColor( TPixel pixel, unsigned int r, unsigned int g,
          unsigned int b )
```

Purpose: Change the color components in a color table entry.

Arguments:

`pixel`	the pixel value.
`r, g, b`	the new color components (see `GLoadColor`).

Notes:

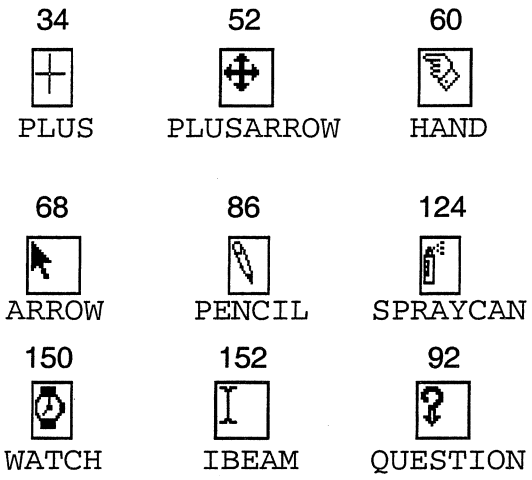
- The pixel value must have been previously returned by `GLoadColor`.
- This changes the color table entry for the given pixel value, which will immediately change the color of all pixels on the screen that have this pixel value.
- In Windows Grafic, for mapped color, the colors will not change immediately and must be re-drawn. For direct color, GSetColor does nothing.
- Ignored on B/W and direct color displays.

GSetCursor

void GSetCursor(int id)

Purpose: Set the cursor for the drawing window.

Arguments: `id` the pre-defined cursor identifier.
The GRAFIC cursors are shown below with their identifiers:



Notes:

- In Windows GRAFIC, the cursors are system-defined cursors which, unfortunately, are not identical in appearance to those shown above.

GSetMItem

void GSetMItem(int menu, int item, char* string)

Purpose: Change the string displayed in the given menu item.

Arguments:

`menu`	the menu number, starting at 1.
`item`	the item number, starting at 1.
`string`	the string to replace the given menu item.

Notes:

- `menu` and `item` refer to the menu in the drawing window.
- In Windows GRAFIC, `GSetMItem` also enables the item. It may be necessary to call `GEnableMItem` afterwards to disable the item.

GSetPixels

void GSetPixels(int left, int top, int width, int height, TPixel pixels[])

Purpose: Set a rectangular area of screen pixel values.

Arguments: `left`,
`top` the window coordinates of the upper left corner.
`width` the number of pixels per row.
`height` the number of rows.
`pixels` the array of pixel values, in row major order.

Notes:

- The pen location is not affected.
- The pixels are copied from the array to the screen. The pen mode does not affect the operation.

GSetWindow

```
void GSetWindow( WID wid )
```

Purpose: Make the window `wid` the drawing window, the destination of future drawing.

Arguments: `wid` the `WID` of an existing window.

Notes:

- The window's stacking order is not changed. The drawing window can have other windows obscuring parts of it.

GSetWindowInfo

```
void GSetWindowInfo( WID wid, int left, int top, int width, int
        height )
```

Purpose: Set the size and location of the given window drawing area.

Arguments: `wid` the `WID` of the window.
`left`, `top` the screen coordinates of the upper left corner of the window drawing area.
`width`,
`height` the dimensions the window drawing area in pixels.

GSetWTitle

```
void GSetWTitle( WID wid, char* title )
```

Purpose: Set the title of the given window.

Arguments: `wid` the `WID` of the window.
`title` the title string.

Notes:

- `title` can have at most `GMAX_STRING` characters, including the '\0' terminator.

GStringHeight

```
int GStringHeight( char* string )
```

Purpose: Return the height of the string in pixels.

Notes:
- Currently, the string argument is ignored so a ‘0’ (`NULL`) can be used.

GStringWidth

```
int GStringWidth( char *string )
```

Purpose: Return the width in pixels of a string.

Arguments: `string` the null terminated string of characters.

GText

```
void GText( char *string )
```

Purpose: Display a string of characters at the current pen location.

Arguments: `string` the null terminated string of characters.

Notes:
- The lower left corner of the first character of the string begins at the current pen location.

Windows GRAFIC

A version of the GRAFIC library is available for Intel compatible personal computers for Microsoft Windows 95 and NT™ using Microsoft Visual C/C++ (version 5 and later). The C routines are identical in operation to their X counterparts, with the few exceptions listed below. This documentation describes some of the differences and idiosyncrasies between program operation on X GRAFIC versus Windows GRAFIC, and explains how to create programs using GRAFIC. See the URL given in the Overview for supported compiler systems and installation instructions.

main() and WinMain

Microsoft Windows Win32 applications begin execution with a call to WinMain, not the standard main as in Unix and nearly all other C programs. For compatibility, a macro "main" is defined in *grafic.h* that expands into a call to a GRAFIC routine (GraficMain).

Thus, for portability across Macintosh's, Unix workstations and PC's, declare your main program as follows:

```
main()
{
        GInitGrafic("title");  /* as usual */
        /* other code as usual */
        return 0;              /* instead of exit(0) */
}
```

This setup will compile and run on all platforms.

Resource Files

Windows GRAFIC uses standard Windows resource files and routines, which are entirely different than X GRAFIC. Create a resource file for your project using the resource editor.

Text

Text is always drawn in GCOPY mode. Windows does not support pen modes for text.

Menus

The standard Windows menu interface is used. Menus appear across the top of each window. GReadMenu(id) reads the MENU resource with the given id number.

You must encode the menu and item numbers into the "control id" field of each menu item. The high order (left) byte is the menu number and the low order (right) byte is the item number. This is easily done using C hexadecimal notation: "0x0201" (decimal 513) corresponds to menu 2, item 1.

There is no GNewMenu in Windows GRAFIC.

Dialogs

The standard Windows dialog interface is used. `GGetDialog(id)` reads the DIALOG resource with the given `id` number.

The dialog item types correspond nearly identically to the standard GRAFIC types. A Button is a "push button" or "Default push button," a Checkbox is an "Auto check box," a RadioButton is an "Auto radio button," StaticText is "Static Text," and EditText is "Edit Text." One of your dialog's buttons should be a "Default push button" Button type, the button returned when a return key is typed (typically OK).

Buttons, RadioButtons, CheckBoxes and EditText must be enabled to work the standard GRAFIC way. The "control id" corresponds to the item number.

Color

In Windows GRAFIC, the type `TPixel` is `long`. In direct color mode, a `TPixel` value contains three bytes specifying the direct RGB color of a pixel. In mapped color mode, a `TPixel` value contains an index for a color palette and some coding bits. This happens transparently at execution time. Use `TPixel` for all pixels to insure proper operation.

Execution Errors

Windows GRAFIC routines check for a number of potentially damaging errors during execution. When an error occurs, the routine `GraficError` is called and passed a string giving a terse description of the error. The routine `GraficError` supplied with the Windows GRAFIC library simply calls the Windows routine `FatalAppExit`, which invokes a standard dialog that shows the message and terminates the application.

Macintosh GRAFIC

A version of GRAFIC is available for Macintosh PowerPC computers for C and C++ using Metrowerks CodeWarrior. The C routines are identical in operation to their X counterparts; with the few exceptions listed below. This documentation describes some of the differences and idiosyncrasies between program operation on X Windows versus the Macintosh.

Files

Macintosh GRAFIC consists of the text file "grafic.h" and the library file "grafic.lib".

Resource Files

Macintosh GRAFIC uses the standard Macintosh resource files and resource manager routines. There is no text resource file in Macintosh GRAFIC. Resources needed for an application, such as menus and dialogs, must be created using the program "ResEdit" and should be placed in a separate resource file that is added to the project. Resource files are automatically compiled and added to an application. The resource files can be edited by simply double-clicking the files in the project window.

Menus

The standard Macintosh menu interface is used. Menus appear across the top of the screen, not inside windows. The menu bar contains the menu for the front window. The `id` in `GReadMenu(id)` gives the resource ID of the appropriate `MBAR` resource.

The resource ID's for each menu in a menu bar must be numbered consecutively, left to right, starting from 1. Take care when creating menus using ResEdit to set the Menu ID correctly using both the "Get Info" command in the FILE Menu and the "Edit Menu & MDEF ID" command in the MENU menu. GRAFIC automatically inserts an Apple menu. The Apple menu contains a "Quit Grafic Program" command that is processed by `GNextEvent` automatically.

Keyboard equivalents are supported by Macintosh GRAFIC. Simply put the proper key character in the `MENU` item "key equiv" box using ResEdit.

Dialogs

Create `DLOG` and `DITL` resources and add them to your ".rsrc" file using ResEdit or other dialog utilities. The `DLOG ID` must match the corresponding `GGetDialog(id)` argument.

Mouse Buttons

The Macintosh mouse has a single button, whereas X and Windows have three, identified as `GLEFTBUTTON`, `GMIDDLEBUTTON` and `GRIGHTBUTTON` by `GNextEvent`. In Macintosh GRAFIC, the command key can be used to map the mouse button to other button values. When the mouse button is pressed or released, `GNextEvent` returns `GRIGHTBUTTON` if no neither key is depressed and `GLEFTBUTTON` if the command key is depressed, as summarized below. `GMIDDLEBUTTON` is not supported.

Key depressed	Button code ('c' argument) returned by GNextEvent
command key	GLEFTBUTTON
neither	GRIGHTBUTTON

Window Manager Functions

Macintosh GRAFIC provides a number of window manager-like actions like X and Windows GRAFIC, which have window manager support for resizing windows and per-window cursors. This functionality is contained in the routine GNextEvent. When the cursor is in the lower right corner of a window and the command key is down, the cursor changes to a down-right arrow symbol (↘|). This indicates the re-sizing area of the window. Depressing the mouse button now will re-size the window showing the standard Macintosh gray outline.

Clicking in the close box in a window (in the upper left corner) will queue a GWMCOMMAND event of type GWMCLOSE with the appropriate WID. Executing the "Quit GRAFIC Program" item in the Apple menu will queue a GWMCOMMAND event of type GWMQUIT.

Color

Macintosh GRAFIC will work on all Macintosh models, with and without color. Although the color routines will function on black-and-white systems and on color systems in black-and-white mode, pixel values returned by certain routines will not be useful. For example, GLoadColor returns either GBLACK or GWHITE. In full color mode, GRAFIC still supplied only 256 colors using a VLT.

Execution Errors

Macintosh GRAFIC routines check for a number of potentially damaging errors during execution. When an error occurs, the routine GraficError is called and passed a string giving a terse description of the error. The routine GraficError supplied with the Macintosh GRAFIC library simply calls the Macintosh toolbox routine DebugStr, which causes a debugger trap that shows the message. If Apple's *MacsBug* is installed, entering the command "es" will abort the program and return you to the monitor. If no debugger is installed, the built-in Macintosh debugger will be entered. It is a small window with a ">" prompt. Check your Macintosh documentation for the proper commands.

Setting up a Microsoft Visual C++ Project for Grafic Programs

Installing Grafic Files

Grafic files are contained in the file "wingrafic.zip" available from http://widget.ecn.purdue.edu/~me573/Grafic/index.html, in "Obtaining Grafic", "Windows Grafic". You should unzip the text include file grafic.h into C:\ProgramFiles\DevStudio\VC\include, and the binary library file grafic.lib into C:\Program_Files\DevStudio\VC\lib. (On your system, the DevStudio\VC directory may be installed on a different disk and/or in a different directory than C:\Program_Files.)

Starting Microsoft Visual C++

After starting "Microsoft Visual C++" (VC) from the Start->Programs menu, the "Microsoft Developer Studio" window appears. The left window pane is the workspace, which initially shows the "InfoView" tab listing the contents of the documentation. The right side is blank, and will contain the text and resource windows later. Along the top are many buttons whose function can be determined by moving the cursor slowly to each and stopping, causing the "tip" window to appear. There is no active project, so this is just the default startup display. After a project is created and saved, opening the ".dsw" file starts VC. The workspace contains the list of files, project settings, and other data that must be saved.

Creating a new project and workspace

Execute the menu command File->New... to invoke the New dialog to start a new project. Select "Win32 Application" from the list of Projects, enter the desired project name in the "Project name:" text box, and set the desired folder in the "Location:" text box. Leave "Create new workspace" and "Platforms: Win32" checked.

Two new tabs are added to the workspace pane: FileView and ClassView. Clicking the tabs changes between them. Most of the time, you'll want FileView. Initially, there are no files.

Creating a new source file

Execute menu command File->New... and select "C++ source file" or "C/C++ header file", then enter the desired File name and Location. Then add the file to the project.

Adding a file to the project

Execute menu command Project->Add To Project->Files... to invoke the "Insert Files into Project" dialog. Set the desired "Files of type" filter, navigate the folders, and select the files to add. Text files (such as .c and .cpp source files), resource files (.rc) and libraries (.lib) can be added this way. For Grafic projects for the course, you'll usually be adding a ".rc" and "resource.h" files, and creating your own .c or .cpp file.

Project settings

Execute Project->Settings... to invoke the "Project Settings" dialog. For simple Grafic programs, select the "Link" tab. In the "Object/library modules:" text box, insert "grafic.lib" as the first library file, before kernel32.lib. On the PC's in ME353, the files grafic.h and grafic.lib have been installed with VC.

Editing project files

Once files have been added to the project, they can be opened by double clicking the name in the workspace window. The file is displayed in a window in the right pane, and can be edited using typical click and type operations. Right-clicking on files in the workspace pane invokes a menu for various operations.

Building the executable file

The Build menu is used to compile and link files into the executable file. For example, executing "Build->build miniproj.exe" will compile any edited source files and resource files, and link the object code into "miniproj.exe". The Build->Compile command can be used to compile a specific file, for example, the file being edited.

The output pane appears along the bottom of the window. It has several tabs: Build, Debug, Find in Files 1 and Find in Files 2. The Build pane shows the progress during Build commands. The Debug pane shows information during debugging. For most purposes, VC takes care of these panes for you. You can close the output pane by clicking the "x" icon, and re-open it using View->Output.

Setting debugger breakpoints

You can instruct the debugger to stop (break) at statements in your source files by right-clicking on the source line and executing the "Insert/Remove Breakpoint" menu command. The first time this will insert a breakpoint and a red dot will appear in the left column of the file. The second time, the breakpoint will be removed. You can see all breakpoints using the Edit->Breakpoints menu command.

Executing the program with the debugger

After a successful Build, you can execute and debug the program with Build->Start Debug->Go (or F5). This will compile any out-of-date source files and re-link the executable, and then set the program into execution.