

Three-Dimensional Computer Graphics

A Coordinate-Free Approach

Tony D. DeRose
University of Washington

Last Revised: October 2, 1992

Copyright ©1991, 1992

Tony D. DeRose

Contents

CHAPTER 1. Introduction	1
CHAPTER 2. Two-Dimensional Raster Algorithms	7
2.1 Scan-converting Line Segments	8
2.1.1 The Line Equation Algorithm	9
2.1.2 The Digital Differential Analyzer	9
2.2 Bresenham's Algorithm	11
2.3 The Device Abstract Data Type	14
2.4 The Simple Graphics Package	19
2.4.1 Two-Dimensional Windowing and Viewporting	19
2.5 Two-Dimensional Line Clipping	23
2.5.1 Cohen-Sutherland Line Clipping	25
2.5.2 The Clipping Divider	27
2.6 Windowing and Viewporting Revisited	28
CHAPTER 3. Coordinate-free Geometric Programming	
I	31
3.1 Problems with the Coordinate-based Approach	31
3.2 Affine Spaces	33
3.3 Euclidean Geometry	42
3.3.1 The Inner Product	43
3.4 Frames	44
3.5 *Matrix Representations of Points and Vectors	49
3.6 Affine Transformations	51
3.7 *Matrix Representations of Affine Transformations	58
3.8 Ambiguity Revisited	60
3.9 Coordinate-Free Line Clipping	62

3.10	A Brief Review of Linear Algebra	67
CHAPTER 4. Three-Dimensional Wireframe Viewing		69
4.1	Introduction	69
4.2	Point Creation	72
4.3	Clipping	73
4.4	Transformation to Screen Space	77
4.5	Scan Conversion	78
CHAPTER 5. Hierarchical Modeling		83
5.1	Simple Polygons	83
5.1.1	Clipping	84
5.1.2	Transforming Through Affine Maps	86
5.1.3	Scan-Conversion	86
5.2	Object Hierarchies	91
5.2.1	Transformation Stacks	91
CHAPTER 6. Hidden Surface Algorithms		93
6.1	Back Face Culling	93
6.2	Three-Dimensional Screen Space	95
6.3	The Depth Buffer Algorithm	96
6.4	Warnock's Algorithm	97
6.5	A Sweep Line Algorithm	98
CHAPTER 7. Coordinate-Free Geometric Program- ming II		101
7.1	Projective Transformations	101
7.1.1	The ProjectiveMap Data Type	108
7.1.2	*Matrix Representations of Projective Maps	108
7.2	Projective Maps and Perspective Viewing	110
7.3	Normal Vectors and the Dual Space	111
7.3.1	The Normal Data Type	115
7.3.2	*Matrix Representations of Dual Vectors	116
CHAPTER 8. Color and Shading		121
8.1	Tri-Stimulus Color Theory	121
8.1.1	Reproducing Spectral Responses with Frame Buffers	123
8.1.2	The CIE Color System	125
8.2	Lighting Models	125

8.2.1	Lambertian Shading	127
8.2.2	Ambient Lighting	131
8.2.3	Specular Reflection	132

Preface

This manuscript is intended as a rigorous introduction to the field of computer graphics at a level appropriate for advanced undergraduates and beginning graduate students in computer science. My intent is not to present a completely comprehensive survey of the field. Rather, my goal is to provide a firm, modern account of those topics within the subfield of three-dimensional raster graphics that can be given adequate treatment in a ten week session. I have therefore, unfortunately, been forced to eliminate discussions of many interesting topics. The text by Foley, van Dam, Feiner, and Hughes should be considered a primary reference for topics not covered here.

The manuscript is based on two courses (CSE 457 and 557) that I have taught over the past several years. The most distinguishing feature is the treatment of the geometric component of the material. Rather than using coordinate calculations, matrices, and matrix manipulations to accomplish geometric computations, a so-called coordinate-free approach is used. It is my feeling that a great deal of conceptual clarity and programming power is achieved by moving to the slightly higher level of abstraction provided by the coordinate-free framework.

Chapter 1

Introduction

The field of as computer graphics really got its start with one man: Ivan Sutherland. Sutherland was a graduate student in the late 50's and early 60's at the MIT Lincoln Laboratory. His landmark Ph.D. thesis described a system called *SketchPad* that was nothing less than a graphical, interactive, constraint-based system for the creation of two-dimensional engineering diagrams.

The display Sutherland used to develop SketchPad is now called a *calligraphic, vector, or stroke* device. Calligraphic displays operate by having a special purpose controller, called a *display processing unit*, govern the electric potential across the deflection plates inside a cathode ray tube. By varying the potentials appropriately, it is possible to cause the electron beam to sweep out a line segment. The picture is then built up by tracing out a potentially large number of line segments.

When the electron beam sweeps out a line, the phosphors coating the inside of the screen fluoresce, but as the beam passes by the intensity decays in a relatively short period of time. If a persistent picture is to be maintained on the screen, the display processing unit must repeatedly *refresh* the image by retracing all of the lines making up the picture, typically at rates between 30 and 60 times per second. The display processing unit must therefore buffer the line segments in a memory known as a *display list* (see Figure 1.1).

Calligraphic displays became quite popular and were successfully marketed by a company called Evans & Sutherland that Sutherland co-founded after he graduated from MIT and started the computer graphics laboratory at the University of Utah. Calligraphic displays were followed by several other display technologies, but they did

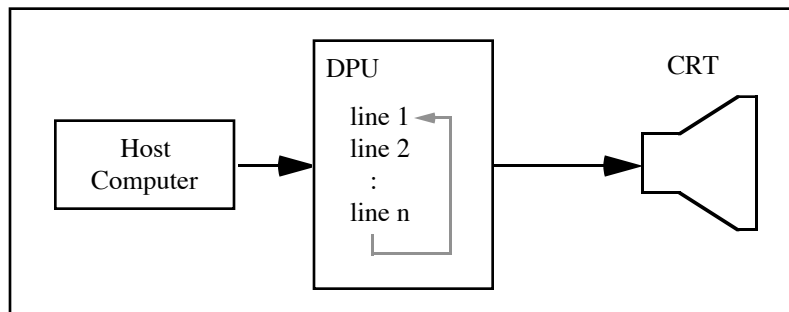


FIG. 1.1: Overview of a calligraphic display system

not have serious competition until the invention of the *raster frame buffer*.

The raster frame buffer was developed in the early 1970's by Dick Shoup while working at the Xerox Palo Alto Research Center. Instead of tracing out the image a line segment at a time, the frame buffer directs the electron beam to trace out the image in a left-to-right and top-to-bottom *raster scan* pattern, much like a standard television set (see Figure 1.2). Each of the left-to-right traces is known as a *scan-line*. As the beam traces along a scan-line, the intensity of the beam is modulated based on the contents of a two-dimensional array known as frame buffer memory. Each entry of the frame buffer memory is associated with a spot, known as a picture element or *pixel*, on the screen. In the simplest scheme, the frame buffer memory consists of one bit per pixel; a pixel is illuminated if and only if the corresponding bit in the frame buffer is set. By allocating several bits per pixel, a grey scale image can be created by arranging for the pixel intensity to be directly proportional to the corresponding value stored in the frame buffer.

The fidelity, or resolution, of a frame buffer image is controlled by the number of scan-lines, the number of pixels per scan-line, and the number of bits per pixel. Frame buffers consisting of 1024×1024 pixels with 8 or 24 bits per pixel are now relatively common. A 1024×1024 , 24 bit per pixel frame buffer is quite a consumer of RAM, requiring a total of 3 megabytes of memory.

There are two styles of color frame buffers, *color mapped* and *full color*. Pixels in a color mapped frame buffer are represented by

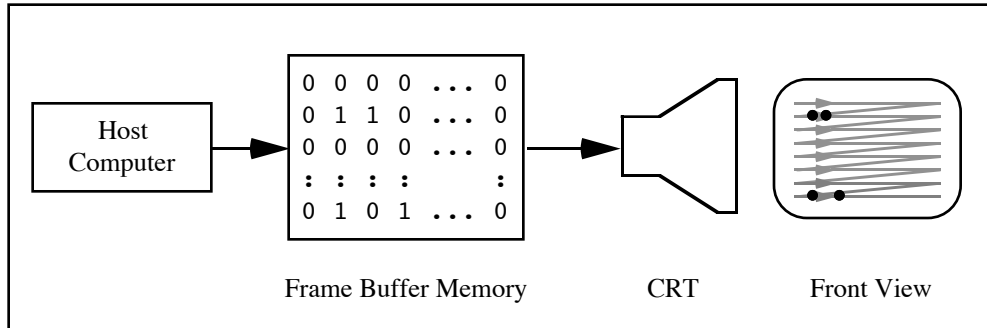


FIG. 1.2: Overview of a frame buffer display system

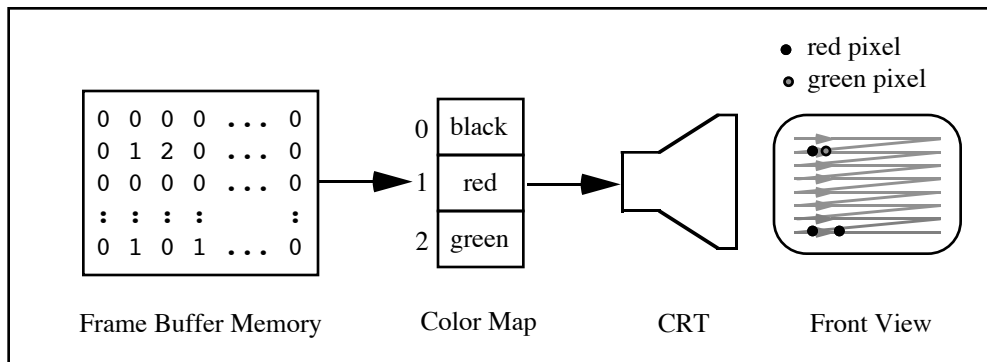


FIG. 1.3: Schematic of a color mapped display system.

an 8 to 12 bit color index. The color indices are turned into colors using a lookup table called a *color map*, as indicated in Figure 1.3. Referring to Figure 1.3, color index number 2, for example, is mapped into the color green. To understand exactly what is stored in each of the entries of the color map, we must look more closely at how color video monitors operate. Whereas monochrome monitors accept a single intensity signal or *channel*, a color video monitor requires three channels: one for red, one for green, and one for blue. Each color map entry i therefore stores three values to indicate the red, green, and blue intensities to associate with color index i . Each of the channel intensities is typically designated with 8 to 16 bit integers. A color map for turning 8 bits per pixel values into three 8 bit intensity channels would require $2^8 * 3 * 1 = 768$ bytes, and would allow color images consisting of up to 256 colors chosen from a *pallette* of 2^{24} possible colors.

The creation of high-quality smooth shaded color images requires many more than 256 colors. These images can only be accurately displayed on a full color frame buffer where at least 24 bits per pixel are available. Notice that color maps as described above would be prohibitively expensive in that the color map would be much larger than the frame buffer memory itself. Full color frame buffers therefore essentially do away with the color map, treating the 24 bits stored at each pixel as being composed of three 8 bit quantities indicating the intensities of each of the three color channels.

Higher quality full color frame buffers typically provide three lookup tables, one for each of the three color channels, that can be used to achieve certain special effects or to correct for non-linearities in the display. A non-linearity that is always present in display systems is caused by the behavior of the phosphors. The intensity I of a phosphor is proportional to δ^γ , where δ is the number of electrons striking the phosphor per unit time and γ is a constant that depends on a number of factors including the type of phosphor and the way it was deposited on the surface of the CRT. In a frame buffer display system, δ is in turn proportional to the value associated with a pixel, implying that the value of a pixel is non-linearly related to the intensity of the spot. The non-linearities can be compensated for by the lookup tables by storing at index i a value proportional to $i^{\frac{1}{\gamma}}$. Working through the chain of proportionalities it is easy to show that this ensures that the pixel value i is linearly related

to the intensity. This process has come to be known as *gamma correction* [5]. Since the value of γ must be known to initialize the lookup tables, true gamma correction requires that the value of γ be measured experimentally for each monitor.

Exercises

1. A spectrophotometer is a device that can accurately measure the intensity of a source of illumination. Describe a procedure for using a spectrophotometer to determine what values to store in a full color frame buffer's lookup tables to achieve gamma corrected images.

Two-Dimensional Raster Algorithms

In this and subsequent chapters we will build up techniques for creating color images of complex three-dimensional environments using full color frame buffers. The basic problem to be addressed may roughly be stated as:

Given: A mathematical description of a two or three-dimensional “scene” and a viewing position.

Find: A value for each pixel in the frame buffer such that the image on the screen is a reasonably accurate picture of what an imaginary viewer would see.

There are, admittedly, a number of ill-defined terms in the above statement, but each of these ideas will be made much more precise as we go along.

The first step in our study of raster graphics is to develop a variety of basic raster algorithms. The most primitive raster operation is the drawing of a dot, i.e., setting a pixel to some particular value. For the next several chapters we will consider only the construction of monochrome images. We assume that pixels can be set using a primitive operation:

```
fb_writePixel( x, y : integer; c : Color)
```

where, for now, the legal values for *c* are assumed to be **WHITE** or **BLACK**. The *x* and *y* parameters to `fb_writePixel()` indicate (ie, address) which pixel is to be modified. Unfortunately, addressing conventions differ from frame buffer to frame buffer. For instance,

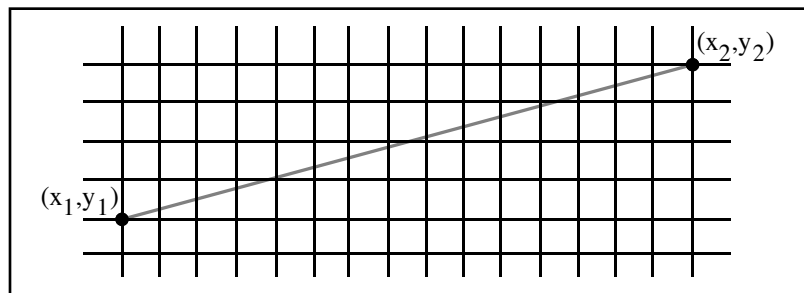


FIG. 2.1: To scan-convert the line segment connecting (x_1, y_1) to (x_2, y_2) , the intermediate pixels must be identified and illuminated. Grid lines denote pixel centers.

pixels on an X window display are addressed so that the upper left corner of the screen corresponds to $(x, y) = (0, 0)$, with x increasing to the right and y increasing downward. Other devices adopt the convention that $(0, 0)$ is the lower left corner, with x increasing to the right and y increasing upward. In this way each frame buffer defines its own *device coordinate system* that associates (x, y) to pixel locations. We say that x and y as above are *device coordinates*.

2.1. Scan-converting Line Segments

The process of painting or *rendering* a geometric entity such as a point, line, or circle into a frame buffer is called *scan-conversion*. Above we assumed that the scan-conversion of points was implemented by the primitive operation `fb_writePixel()`. In this section we examine the scan-conversion of the simplest non-trivial geometric entity, the line segment. Specifically, we consider the following problem:

Given: Two pixel locations (x_1, y_1) and (x_2, y_2) in device coordinates.

Find: The intermediate pixels to illuminate to represent the line segment connecting (x_1, y_1) to (x_2, y_2) as indicated in Figure 2.1.

We will solve this problem by beginning with a more or less obvious method, then refine the method until we derive an algorithm

that strikes a sensible balance between speed, accuracy, and ease of implementation. In the remainder of this section, we assume that device coordinates are such that x increases to the right and y increases upward.

We should first lay down a set of properties we would like our solutions to possess. Although some of the following properties seem obvious enough to ignore, we shall see apparently acceptable algorithms that fail to possess them.

Properties:

1. Lines should appear as straight as possible.
2. Lines should terminate exactly at (x_1, y_1) and (x_2, y_2) .
3. Lines should have relatively constant intensities.
4. The intensity of a line should be independent of slope.
5. The algorithm should be relatively efficient since line drawing is in the inner loop of many applications.

2.1.1. The Line Equation Algorithm The first algorithm for scan-converting the line segment $(x_1, y_1), (x_2, y_2)$ might be called the “Line Equation Algorithm” since it is based on the familiar equation $y = mx + b$ for lines, where m is the slope and b is the y intercept. Figure 2.2 presents a pseudo-code statement of the algorithm.

Although this algorithm is intuitive, it fails to possess several of the properties listed above. Notice that only one pixel is illuminated in each pixel column. This means that if L_1 and L_2 are two line segments of equal length emanating from (x_1, y_1) , with the slope of L_1 greater than the slope of L_2 , then fewer pixels will be illuminated for L_1 than for L_2 . This violates property 4 since it implies that the perceived intensity of the scan-converted line depends on the slope. Passing to the limit of infinite slope (i.e., a vertical line) we discover a more serious problem with the algorithm: it causes an arithmetic exception (a stoic term for “crashes”) when it attempts to divide by $(x_2 - x_1)$, which is, of course, zero for vertical lines.

2.1.2. The Digital Differential Analyzer The problems encountered with the Line Equation Algorithm can be partially remedied by noting that there is a symmetry in the problem that is not


```
LineEquationAlgorithm( x1, y1, x2, y2 : integer; c : Color)
begin
  m,b: real;
  x,dx: integer;

  m := (y2-y1)/(x2-x1);
  b := y1-m*x1;
  if (x2 - x1) > 0 then
    dx := 1.0;
  else
    dx := -1.0;
  endif;

  for x := x1 to x2 step dx do
    y := m*x + b;
    fb_writePixel( x, Round(y), c);
  endfor;
end
```

FIG. 2.2: A straightforward line drawing algorithm based on the line equation $y = mx + b$.

reflected in the algorithm. In the original problem statement for scan-converting lines, the x and y coordinates play completely symmetric roles, whereas the Line Equation Algorithm breaks this symmetry by always computing y as a function of x . The algorithm can therefore be improved by modifying it to interchange the roles of x and y if $|y_2 - y_1| > |x_2 - x_1|$. The algorithm can be further improved to reduce the number of floating point computations required in the inner loop. The key is to exploit the fact that the y value y_{i+1} needed for the $i+1$ st iteration can be computed incrementally from y_i . The relation

$$x_{i+1} = x_i + 1,$$

implies that

$$y_{i+1} = m(x_i + 1) + b = y_i + m,$$

and therefore y_{i+1} can be computed from y_i with a single addition. By replacing the statement $y := m*x + b$ with $y := y + m$, the multiplication is avoided in the inner loop. With these two improvements, symmetrization and incremental calculation, we have essentially derived an algorithm known as the Digital Differential Analyzer (for reasons that have nearly been lost to the mists of time...), or DDA for short. A pseudo-code statement of the DDA algorithm is shown in Figure 2.3.

2.2. Bresenham's Algorithm

There is a final improvement that we shall consider: the removal of *all* floating point computations to arrive at an entirely integer algorithm. The algorithm we now present was originally due to Jack Bresenham [3], although we have chosen to use the alternate derivation from Foley et al [11], which in turn is due to Pitteway [15]. (Historical aside: Bresenham originally devised the algorithm for drawing lines with pen-plotters, not frame buffers.)

As indicated above, Bresenham's algorithm will only require integer arithmetic. In fact, the only arithmetic operations required are integer addition, subtraction, and bitwise shifting. In what follows we make the simplifying assumption that $0 < y_2 - y_1 \leq x_2 - x_1$, i.e., that the slope of the line is between 0 and 1. The relaxation of this assumption is the subject of one of the exercises at the end of this chapter.

Bresenham's algorithm iterates over the pixel columns between

```
DDA( x1, y1, x2, y2: integer; c: Color)
begin
    length, dx, dy, i: integer;
    x, y, xincr, yincr: real;

    dx := x2 - x1;
    dy := y2 - y1;
    length := max( |dx|, |dy|);

    { Either xincr or yincr has magnitude 1. }
    xincr := dx/length;
    yincr := dy/length;

    x := x1;   y := y1;
    for i := 1 to length+1 do
        fb_writePixel( Round(x), Round(y), c);
        x:= x + xincr;
        y:= y + yincr;
    endfor;
end
```

FIG. 2.3: The Digital Differential Analyzer (DDA) scan-conversion algorithm

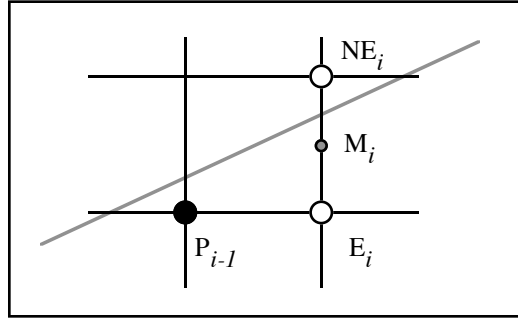


FIG. 2.4: The situation at an intermediate step of Bresenham's algorithm.

x_1 and x_2 , inclusive; on each iteration the pixel closest to the true line is chosen. Let P_{i-1} with coordinates (x_{i-1}, y_{i-1}) denote the pixel selected on the i -1st iteration of the algorithm. Referring to Figure 2.4, on the i th iteration the algorithm must choose between the pixels E_i and NE_i (these are the only two possibilities since the slope is restricted to be between 0 and 1). The algorithm will make the choice based on the value of an incrementally computed *decision variable*. To derive the decision variable it is convenient to express the line in *implicit form*; that is, as

$$F(x, y) = Ax + By + C = 0. \quad (2.1)$$

The coefficients A , B , and C in the implicit form can be readily computed from the line equation

$$y = \frac{\Delta y}{\Delta x}x + b \quad (2.2)$$

where

$$\begin{aligned} \Delta x &= x_2 - x_1 \\ \Delta y &= y_2 - y_1 \\ b &= y_1 - \frac{\Delta y}{\Delta x}x_1, \end{aligned}$$

by multiplying Equation 2.2 through by $2\Delta x$ to find that

$$F(x, y) = \underbrace{2\Delta y}_A x + \underbrace{(-2\Delta x)}_B y + \underbrace{2\Delta x b}_C = 0. \quad (2.3)$$

The justification for multiplying by $2\Delta x$ instead of Δx will become apparent shortly. From Equation 2.3 we observe that

1. If $F(x, y) < 0$, the point (x, y) is above the line.
2. If $F(x, y) > 0$, the point (x, y) is below the line.
3. A , B , and C are integers.

Observations 1 and 2 imply that if M_i denotes the midpoint between E_i and NE_i , and if $F(M_i) < 0$, then $P_i := E_i$ (that is, E_i should be chosen on the i th iteration); otherwise, $P_i := NE_i$. (Think about what should be done if $F(M_i) = 0$.)

The number $d_i = F(M_i)$ is the decision variable we were seeking. The convenient aspect of this particular choice of the decision variable is that it can be computed incrementally using only integer arithmetic. If E_i is chosen on the i th iteration, then

$$\begin{aligned} d_{i+1} &= F(x_{i-1} + 2, y_{i-1} + \frac{1}{2}) \\ &= d_i + A, \end{aligned}$$

and if NE_i is chosen on the i th iteration, a similar analysis shows that

$$d_{i+1} = d_i + A + B.$$

About the only remaining detail is to discover how to initialize the decision variable. It is not difficult to show that

$$d_1 = A + \frac{B}{2}.$$

The seemingly extraneous factor of two that was introduced into the definitions of A , B , and C was chosen precisely so that d_1 would be an integer. A pseudo-code statement of the complete algorithm is given in Figure 2.5.

2.3. The Device Abstract Data Type

We will eventually be developing relatively sophisticated application programs that read in geometric data, process them in various ways, and finally scan-convert them to create an image. As our formalism currently stands, application programs must know the specifics of the device coordinate systems for each of the devices they are to output

```

Bresenham( x1, y1, x2, y2 : integer; c : Color)
{ Draw the line segment from (x1,y1) to (x2,y2) assuming that the }
{ slope of the line is between 0 and 1 }
begin
    d, dx, dy, x, y : integer;
    incrE : integer;    { Amount to add when E chosen }
    incrNE : integer;  { Amount to add when NE chosen }

    { Compute loop invariant quantities }
    dx := x2 - x1;
    dy := y2 - y1;
    incrE := dy << 1; { << 1 means left shift by 1 bit }

    { Initialize incremental quantities }
    d := incrE - dx;
    incrNE := d - dx;
    x := x1;  y := y1;
    fb_writePixel( x, y, c);

    { Scan-convert the line segment }
    while (x < x2) do
        x := x + 1;
        if (d < 0) then
            { Choose E }
            d := d + incrE;
        else
            { Choose NE }
            d := d + incrNE;
            y := y + 1;
        endif
        fb_writePixel( x, y, c);
    endwhile;
end;

```

FIG. 2.5: Bresenham's algorithm for scan-converting lines whose slope is between 0 and 1.

to. Software engineering practices suggest that this information should be encapsulated to define an abstract data type (ADT) that models an idealized display device. The definition of the device ADT should abstract out as many of the details of specific devices as possible. Abstraction of detail is, however, in tension with the desire to take advantage of special hardware features of many modern frame buffers. For example, some graphics display systems currently provide hardware support for *bit blit*, the rapid copying of blocks of pixel values to and from the frame buffer memory. The designer of a portable device ADT may therefore be faced with difficult choices when deciding which high level operations to include and which to exclude.

In this section we will define a very simple idealized device and its corresponding ADT. Our primary goal is to abstract out details of device coordinate systems and color resolutions. Our idealized device will accept coordinates in an idealized coordinate system commonly known as *normalized device coordinates*. Normalized device coordinates, or NDC for short, are defined as shown in Figure 2.6. Normalized device coordinates are defined to closely match to sorts of coordinate systems typically encountered in analytic geometry. Points are addressed in NDC by specifying a pair of real-valued coordinates (x, y) . The origin is defined to be in the lower left corner, with the x axis pointing to the right and the y axis pointing upward. The visible portion of the NDC plane is defined to be the unit square $[0, 1] \times [0, 1]$; points lying outside the unit NDC square will not appear in the image. By using real rather than integers coordinates we have abstracted out the horizontal and vertical pixel resolutions of physical frame buffers. For the time being, we will assume that colors are selected from an enumerated type, containing at least the values WHITE and BLACK as before. A more sophisticated ADT for colors will be developed in Chapter 8.

As part of the definition of the idealized device, we also demand that the image of the NDC square actually appears as a square on whatever physical screen is being used. This is not as easy to accomplish as it sounds. Many physical frame buffers generate non-square pixels, meaning that the number of pixels covered by the NDC square must differ in the horizontal and vertical directions. We will revisit this issue later in this section.

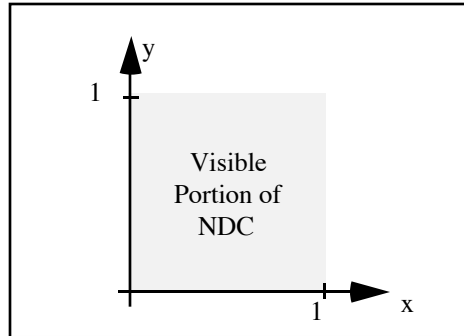


FIG. 2.6: Normalized device coordinates

Operations on the idealized device include:¹

- DeviceDrawDot(x, y : **real**; c : Color)
Draw a dot at the point (x, y) in the color specific by c .
- DeviceDrawLine(x_1, y_1, x_2, y_2 : **real**; c : Color)
Draw a line from (x_1, y_1) to (x_2, y_2) in the color specified by c .
- DeviceDrawText(x, y : **real**; str : **string**; c : Color)
Draw the string str starting at the point (x, y) . (A more sophisticated device ADT would include control over the size and perhaps the font the string is to be drawn in.)

An implementation of device ADT is a body of software, generally called a *device driver*, that maps the abstractions of the idealized device onto a concrete frame buffer. The device driver therefore encapsulates all device dependent information, making it easy to port applications from one device to another.

One of the principal responsibilities of a device driver is to transform normalized device coordinates (n_x, n_y) into device coordinates (d_x, d_y) appropriate for the frame buffer at hand. A pair of transformations $ToDev_x : n_x \mapsto d_x$ and $ToDev_y : n_y \mapsto d_y$ that operate on x and y coordinates, respectively, can be used for this purpose. As a specific example of the development of such coordinate transformations, suppose the device coordinates are as shown in Figure 2.7,

¹All coordinates are specified in NDC.

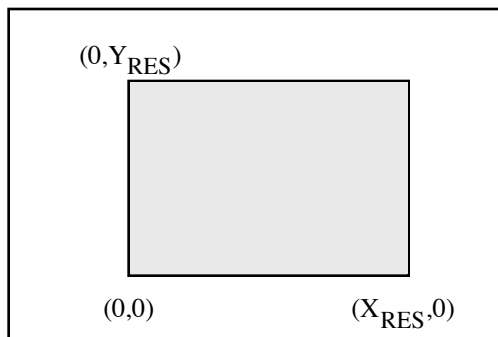


FIG. 2.7: A typical physical device coordinate system.

where the lower left corner of the screen corresponds to $(0,0)$, the lower right corner to $(X_{RES},0)$, the upper left corner to $(0,Y_{RES})$, and the upper right corner to (X_{RES},Y_{RES}) . The integers X_{RES} and Y_{RES} refer to number of pixels on the screen in the horizontal and vertical directions. Suppose further that the physical screen is wider than it is tall (a ratio of 4 to 3 is common, but by no means universal).

We shall construct the transformations $ToDev_x()$ and $ToDev_y()$ so that the image of the NDC unit square will overlay the largest central square portion of the screen, as indicated in Figure 2.8. Denote by x_0 and x_1 the x device coordinates of the left and right edges of the image of the NDC unit square. In practice these numbers must be determined by physically measuring the screen (who says computer science isn't a physical science?). The function $ToDev_x()$ is therefore subject to two constraints:

1. $ToDev_x(0) = x_0$.
2. $ToDev_x(1) = x_1$.

If $ToDev_x()$ is chosen to be a linear function, it is completely determined by these two conditions:

$$ToDev_x(n_x) = (1 - n_x) x_0 + n_x x_1. \quad (2.4)$$

A similar process for $ToDev_y()$ shows that

$$ToDev_y(n_y) = n_y Y_{RES}. \quad (2.5)$$

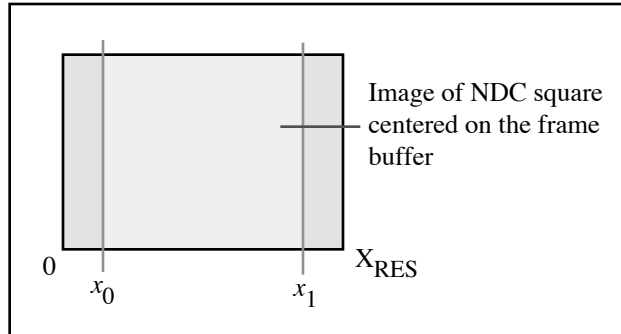


FIG. 2.8: The NDC square is mapped to the largest central square on the physical screen.

Given procedures to compute $ToDev_x()$ and $ToDev_y()$, an implementation of $DeviceDrawLine()$ could simply transform x_1, y_1, x_2, y_2 to device coordinates, then use Bresenham's algorithm to scan-convert the line.

2.4. The Simple Graphics Package

In this section we begin the development of a layer of graphics software designed to provide convenient, general facilities to higher level application programs. We shall refer to this set of routines as the Simple Graphics Package, or SGP. SGP will serve as a mediator between the device ADT on the low-level side, and the application program on the high-level side. To motivate the initial development of SGP, in the next several sections we consider the construction of a simple two-dimensional data plotting program. A good deal of the rest of the text is devoted to extending SGP to handle the specification and viewing of three-dimensional smooth shaded color images.

2.4.1. Two-Dimensional Windowing and Viewporting

Consider a program that reads in two dimensional data points and creates images such as the one shown in Figure 2.9 that plots yearly rainfall (in feet for Seattle, in inches for California). In this example the independent variable is the year and the dependent variable is the rainfall. Since the data points lie outside the unit square, we

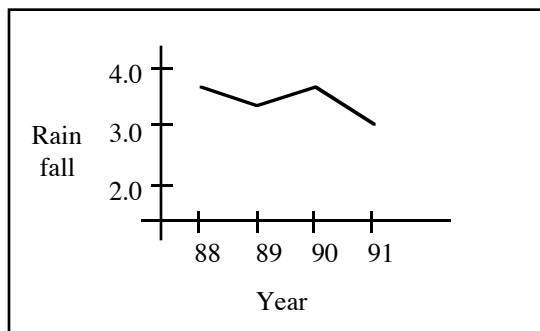


FIG. 2.9: A sample plot.

must first transform them to points in the unit square before using routines such as `DeviceDrawLine()`.

A conceptual framework for reasoning about what the transformation from data points to points in the unit square must satisfy is to imagine the final plot as a view into the two-dimensional “world” in which the data “lives”, as indicated in Figure 2.10. This data space, more generically called *world space*, is an arbitrarily large continuous plane upon which an abstraction of the image is imagined to exist. A view into world space is established by specifying a correspondence between a region of world space and a region of the NDC square. To emphasize the fact that the world space and the NDC square are conceptually distinct, the NDC square is imagined to be a portion of a separate infinitely large continuous plane known as *screen space*. Although one could envision more complicated schemes, one way to establish a correspondence between world space and screen space is to identify two rectangles, one in the world space, known as the *window*, and one in screen space, known as the *viewport*. Once these rectangles have been identified, points in the interior of the window can be mapped to the points interior to the viewport using a simple linear transformation.

The application program can communicate the position of the window and the viewport to SGP by having SGP export the following two routines:

- `SGPSetWindow(WINleft, WINright, WINtop, WINbottom)`

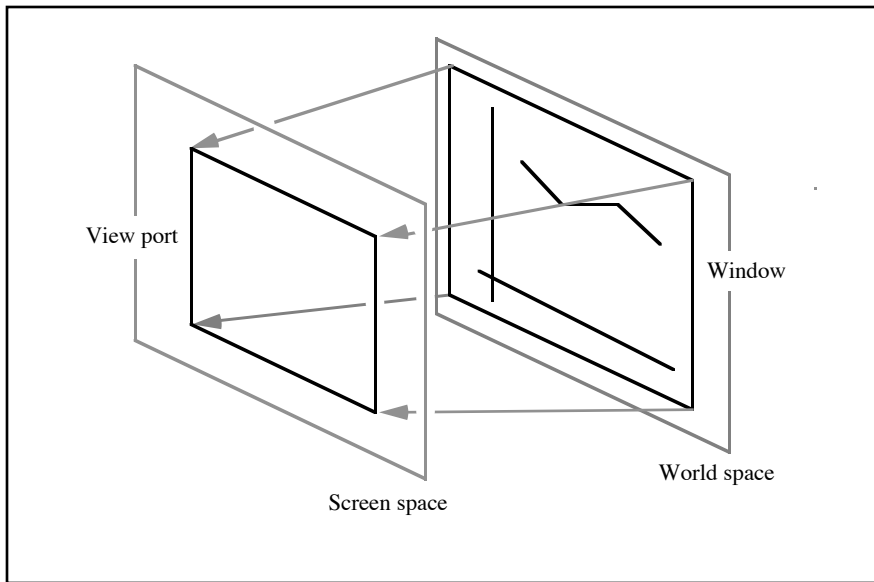


FIG. 2.10: The correspondence between world space and screen space is established by mapping a rectangle in the world (the window), into a rectangle in the screen (the viewport).

- `SGPSetViewPort(VPleft, VPright, VPtop, VPbottom)`

The arguments to these routines specify the left, right, top and bottom extents of the window and viewport, respectively. The parameters to `SGPSetWindow()` correspond to coordinates in the world space, whereas the parameters to `SGPSetViewPort()` correspond to coordinates in screen space. The calls necessary for establishing the connection indicated in Figure 2.10 might be something like:

```
SGPSetWindow( 85, 93, 4.5, -4.0)
SGPSetViewPort( 0.25, 0.75, 0.75, 0.25)
```

A point (w_x, w_y) in world space can be transformed into the corresponding point (n_x, n_y) in screen space using a pair of linear functions similar to `ToDevx()` and `ToDevy()`. Denoting these functions by `ToNDCx()` and `ToNDCy()`, it is not difficult to show that $(n_x, n_y) = (\text{ToNDC}_x(w_x), \text{ToNDC}_y(w_y))$ where

$$\begin{aligned} \text{ToNDC}_x(w_x) &= \frac{VP_{right} - VP_{left}}{WIN_{right} - WIN_{left}}(w_x - WIN_{left}) + VP_{left} \\ \text{ToNDC}_y(w_y) &= \frac{VP_{top} - VP_{bottom}}{WIN_{top} - WIN_{bottom}}(w_y - WIN_{bottom}) + VP_{bottom} \end{aligned}$$

Having established the correspondence between the world and screen spaces, SGP can take on the responsibility for automatically transforming drawing requests to screen space, allowing the application program to work more naturally in world coordinates. Specifically, SGP can export the routines

- `SGPDrawDot(x, y : real; c : Color)`
Draw a dot at the world space point (x, y) .
- `SGPDrawLine(x1, y1, x2, y2 : real; c : Color)`
Draw the world space line segment from (x_1, y_1) to (x_2, y_2) .
- `SGPDrawText(x, y : real; str : string; c : Color)`
Draw the string `str` starting at the world space point (x, y) .

These routines are most simply implemented by transforming the world space points into points in screen space by using `ToNDCx()` and `ToNDCy()`, followed by calling the corresponding routine exported by the device ADT.

```

program DataPlot;
begin
    x1, y1, x2, y2 : real;

    { Set up the window and viewport }
    SGPSetWindow( 85,92,0.0,4.5);
    SGPSetViewPort(0.25,0.75,0.25,0.75);

    { Draw the axes }
    SGPDrawLine( 87, 0.9, 87, 4.3);
    SGPDrawLine( 86, 1.0, 92, 1.0);

    { Draw the graph }
    read x1, y1;
    while more input do
        read x2, y2;
        SGPDrawLine( x1, y1, x2, y2);
        x1 := x2;   y1 := y2;
    endwhile;
end;

```

FIG. 2.11: The skeleton of a simple data plotting program based on SGP.

Using the SGP routines, the application program shown in Figure 2.11 could be used to generate (a simplified version of) the plot of Figure 2.9. The flow of control is summarized in the diagram of Figure 2.12 called the *two-dimensional graphics pipeline*.

To summarize, the device driver provides the abstraction of the screen space (i.e., NDC coordinates), and SGP provides the abstraction of world space. Application programs are therefore freed from many of the irrelevant details of the coordinate transformations required to correctly position the line segment on the screen.

2.5. Two-Dimensional Line Clipping

There are still a few details to deal with before leaving the two dimensional version of SGP. Consider, for instance, how a request such as

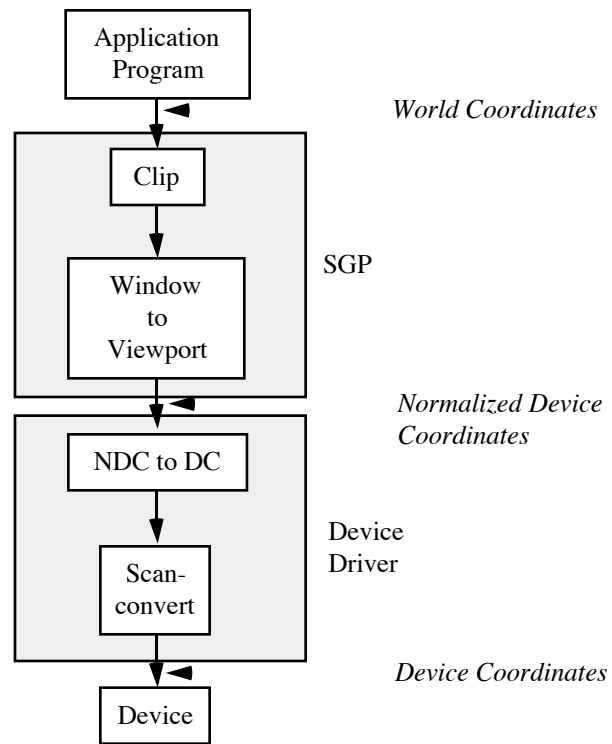


FIG. 2.12: The two-dimensional graphics pipeline as typically implemented in software.

SGPDrawLine(91, 3.0, 92, 10.0)²

should be dealt with. The usual action in such a case is to *clip* the line segment to the interior of the window. That is, we wish to trim away that part of the line segment that lies outside the window, and process the remainder as before. We will examine two line clipping algorithms in this section. The first is intended as a software solution whereas the second is particularly suited to a hardware implementation. A third line clipping algorithm, one that is easily extended to the clipping of polygons in two and three dimensions, is presented in Section 3.9.

2.5.1. Cohen-Sutherland Line Clipping Cohen and Sutherland developed a particularly efficient method for clipping line segments that is based on a clever classification of the endpoints of the segment. The Cohen-Sutherland algorithm is constructed to optimize the common cases, occurring when the line segment is either entirely within the window, or is entirely outside the window. The classification is based on the observation that if both endpoints are, say, above the window, then the entire line segment must be above the window, and can therefore be *trivially rejected*. The same situation holds when the endpoints are both left, right, or below the window. Each endpoint is therefore characterized by a four-bit vector, called an *outcode*, that indicates where the endpoint lies relative to the infinitely extended edges of the window. The meaning of each of the bits of an outcode is given in Figure 2.13. The outcodes effectively divide the world space into nine regions arranged around the window as shown in Figure 2.14

If P_1 and P_2 denote the endpoints of the line to be clipped, then if the bitwise “anding” of the outcodes of P_1 and P_2 yields a non-zero result, then either both points were left, right, above, or below the window. In such a case the entire line segment must be outside the window, meaning that the segment can be trivially rejected. This is the situation for line segment 1 in Figure 2.15. Line segment 2 of Figure 2.15 is entirely within the window. This can be detected by noting that both endpoints have the outcodes 0000 – such a line segment is said to be *trivially accepted*.

The remaining line segments in Figure 2.15 can neither be

²Don't laugh – in Seattle it could happen.

Bit Number	Meaning if Set
1	Point above window
2	Point below window
3	Point right of window
4	Point left of window

FIG. 2.13: Outcodes assigned to endpoints by the Cohen-Sutherland algorithm.

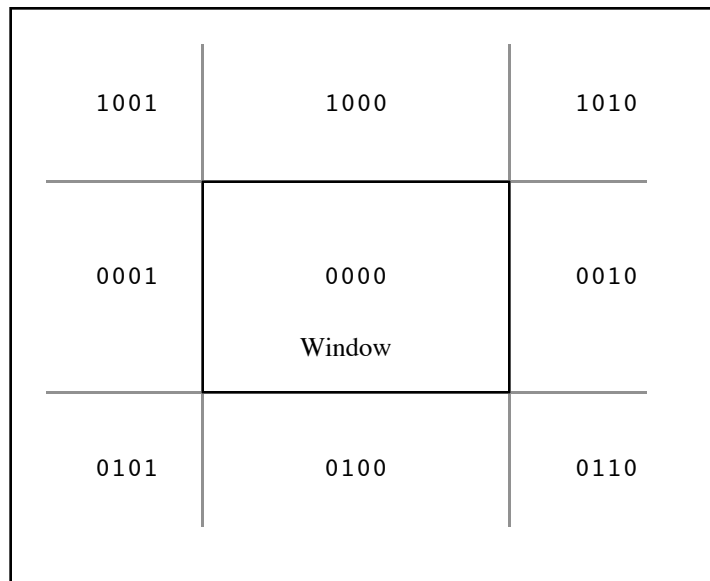


FIG. 2.14: The outcodes partition the world space into nine regions arranged around the window as shown above.

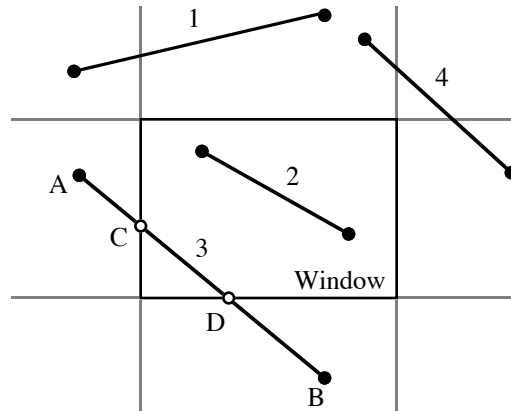


FIG. 2.15: Various line segments.

trivially accepted nor trivially rejected. They must be processed further by successively intersecting them with the infinitely extended edges of the window. Line segment 3, for instance, is processed by computing the point C where the segment intersects the left edge of the window. If (x_a, y_a) and (x_b, y_b) denote the coordinates of the endpoints A and B , respectively, then the coordinates (x_c, y_c) of C are given simply by

$$x_c = WIN_{left}$$

$$y_c = \frac{(x_b - WIN_{left})y_a + (WIN_{left} - x_a)y_b}{x_b - x_a}$$

The subsegment AC can be trivially rejected, leaving the subsegment CB for further processing. Clipping CB to the lower edge of the window allows the segment DB to be trivially rejected and the remaining segment CD to be trivially accepted.

2.5.2. The Clipping Divider As indicated in Figure 2.12, in software implementations of the graphics pipeline it is generally advantageous to clip the segments in world coordinates rather than postponing the clipping to NDC or device coordinates. The reason is that by clipping as early as possible in the pipeline, potentially many segments will be culled, thereby reducing the processing demands on later stages.

The situation is quite different when considering the mapping of the graphics pipeline into hardware. In this case it is more convenient to postpone the clipping phase so that it is done in device coordinates so that integer rather than floating point arithmetic can be used. The *clipping divider* [16] is an integer based divide and conquer method for clipping line segments in device coordinates. The algorithm tests the segment being processed for trivial accept and reject conditions. If neither of these cases holds, the midpoint of the segment is computed (requiring only shifts and adds), thereby breaking the segment into two subsegments. Each subsegment is processed recursively. The clipping divider can easily be implemented using current VLSI technology, requiring only an integer ALU, a stack, and some simple control logic.

2.6. Windowing and Viewporting Revisited

In preparation for the geometric discussions of the next chapter, let us reexamine the transformation between normalized device coordinates and Device Coordinates that was developed in Section 2.3. This transformation, by Equations 2.4 and 2.5, can be written in matrix form as

$$(d_x \ d_y) = (n_x \ n_y) \begin{pmatrix} x_1 - x_0 & 0 \\ 0 & Y_{RES} \end{pmatrix} + (x_0 \ y_0).$$

A trickier form can be used to replace the addition with multiplication of slightly larger matrices:

$$(d_x \ d_y \ 1) = (n_x \ n_y \ 1) \underbrace{\begin{pmatrix} x_1 - x_0 & 0 & 0 \\ 0 & y_1 - y_0 & 0 \\ x_0 & y_0 & 1 \end{pmatrix}}_{\mathbf{N}} \quad (2.6)$$

The transformation between world space and screen space can similarly be characterized by the matrix equation

$$(n_x \ n_y \ 1) = (w_x \ w_y \ 1) \underbrace{\begin{pmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ t_x & t_y & 1 \end{pmatrix}}_{\mathbf{S}} \quad (2.7)$$

where

$$\begin{aligned} f_x &= \frac{VP_{right} - VP_{left}}{WIN_{right} - WIN_{left}} \\ f_y &= \frac{VP_{top} - VP_{bottom}}{WIN_{top} - WIN_{bottom}} \\ t_x &= VP_{left} - f_x WIN_{left} \\ t_y &= VP_{bottom} - f_y WIN_{bottom} \end{aligned}$$

Combining Equation 2.6 with Equation 2.7 we find that

$$(d_x \ d_y \ 1) = (w_x \ w_y \ 1)\mathbf{W} \quad (2.8)$$

where the matrix \mathbf{W} is the product of \mathbf{S} and \mathbf{N} .

The somewhat mysterious appearance of the third component of 1 in tuples such as $(d_x \ d_y \ 1)$ and $(n_x \ n_y \ 1)$, and the use of 3×3 matrices for two-dimensional transformations will be thoroughly explained in the next chapter where we begin in earnest the mathematical study of geometry and geometric calculations.

The symbol juggling above shows that the calculations required to transform a point from world space into the corresponding point in screen space represented in device coordinates can be accomplished by building and multiplying a carefully chosen set of matrices. For this reason computer graphics texts develop geometric transformations from the point of view of matrix manipulations. We call this a *coordinate-based approach* since the matrices describe exactly how to combine the coordinates to achieve the (hopefully) desired geometric effect.

While a coordinate-based approach has its merits, not the least of which is a certain amount of familiarity, it also has some serious drawbacks that will be identified in the next chapter. We shall therefore pursue a *coordinate-free* treatment that emphasizes the geometric meaning of an operation instead of the low-level coordinate manipulations necessary to carry out its computation.

Exercises

1. Generalize Bresenham's algorithm to accept as input an arbitrary line segment.

2. Explain how to speed up Bresenham's algorithm by roughly a factor of two by exploiting a certain symmetry property possessed by the algorithm.
3. Do the DDA algorithm and Bresenham's algorithm produce the same results? If so, prove it. If not, provide a counterexample and characterize the ways in which the pixel patterns differ.
4. The line segments created by Bresenham's algorithm can appear to be rather "jagged". The jagged appearance of the segment can be reduced if the frame buffer has more than one bit per pixel used to create grey scale images. The idea is to partially illuminate all the pixels "near" the line so that pixels closer to the line are brighter. Develop and experiment with such a variant of Bresenham's algorithm, assuming a grey scale frame buffer that allocates 2^b bits per pixel.
5. There is a subtlety in the clipping divider method concerning arithmetic precision. Exactly what precision arithmetic is required by the algorithm? Why?

Coordinate-free Geometric Programming I

3.1. Problems with the Coordinate-based Approach

Graphics programs written in a coordinate-based way use matrix manipulations to express geometric operations. Unfortunately, a given matrix computation can have many geometric interpretations; the particular geometric interpretation is left to the imagination and discipline of the programmer. As an example, the code fragment shown in Figure 3.1 can be interpreted geometrically in at least three ways: as a change of coordinates, as a transformation from the plane onto itself, and as a transformation from one plane onto another (see Figure 3.2). The interpretation as a change of coordinates leaves the point unchanged geometrically, but changes the reference coordinate system (Figure 3.2(a)). The interpretation as a transformation of the plane onto itself moves the point, keeping the coordinate system fixed (Figure 3.2(b)). Finally, the interpretation as a transformation from one plane onto another involves two coordinate systems, one in the domain, and one in the range (Figure 3.2(c)). It is the interpretation as a transformation between planes that is appropriate for the matrix multiplications of Equations 2.6 and 2.7.

A common response to this ambiguity is that it does not matter which view is taken. Indeed, this is the response that most students of computer graphics come to believe. Unfortunately, this is not quite correct since it is possible to distinguish between the interpretations. In particular, lengths and angles do not change in the first interpretation, but they can in the second and third interpretations.

Above it was argued that a matrix computation could have many geometric interpretations. It is also the case that a matrix

$$\begin{aligned} \mathbf{P} &\leftarrow (p_1 \ p_2); \\ \mathbf{T} &\leftarrow \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}; \\ \mathbf{P}' &\leftarrow \mathbf{P} \mathbf{T}; \end{aligned}$$

FIG. 3.1: A typical matrix computation.

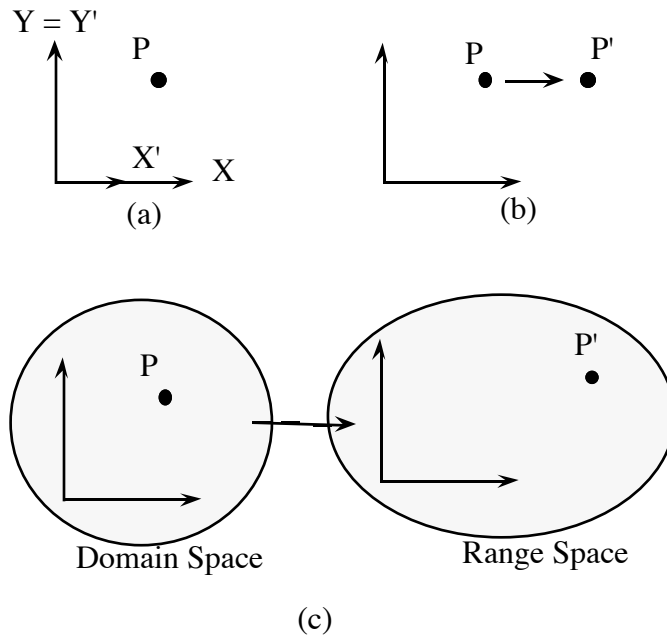


FIG. 3.2: Three interpretations of the code fragment of Figure 3.1.

computation can have *no* geometric interpretation. Some errors are allowed to creep in because there is no explicit representation of coordinate systems or spaces. The programmer is expected to maintain a clear idea of which coordinate system in which logical space (e.g., world coordinates, normalized device coordinates in screen space, etc.) each point is represented. As a consequence, the burden of coordinate transformations must be borne directly by the programmer. If extreme care is not taken, it is possible (and in fact common) to perform geometrically meaningless operations such as combining two points that reside in different spaces or are represented relative to different coordinate systems.

We will address the problems of ambiguity and validity by developing a coordinate-free geometric algebra (i.e., a collection of geometric objects together with operations for combining them) that promotes geometric reasoning rather than coordinate manipulations. Associated with the algebra will be an ADT that implements the abstractions provided by the algebra. The algebra and ADT are constructed so that only geometrically meaningful operations are possible. Moreover, all operations are geometrically unambiguous and their interpretation is clearly reflected by the code.

Although the development of the algebra is done in a coordinate-free way, the ADT must ultimately be implemented using coordinates. It is therefore important for the implementor of the ADT to understand how to translate geometric operations into coordinate calculations. In an effort to clearly separate the coordinate-free material from the coordinate-based material, the coordinate-based sections have been marked with an asterisk.

3.2. Affine Spaces

Although the geometric ADT will present abstractions based on Euclidean geometry, many of the geometric objects and operations that find use in computer graphics and related fields such as computer aided geometric design (CAGD) are founded in the more general branch of mathematics known as affine geometry. We have therefore chosen to develop the affine theory here, then specialize to Euclidean geometry in Section 3.3.

There are many different approaches to affine geometry [8, 10, 23]. One approach, first put forth by Weyl [23] (a modern account of which can be found in Dodson and Poston [8] and Flohr and Raith

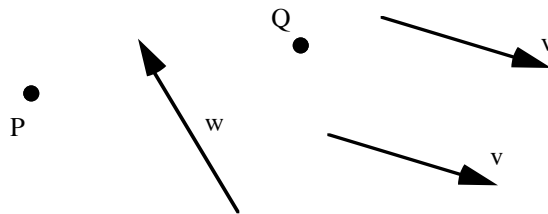


FIG. 3.3: Geometric interpretations of points and vectors.

[10]), makes a distinction between points and vectors, but does not define operations for combining them. The method we shall adopt is very similar to that used by Dodson and Poston. This development of affine geometry builds on vector spaces, so a brief review of the relevant parts of linear algebra is supplied in Appendix 3.10.

The most basic objects in the geometric algebra will be *affine spaces*, which in turn consist of *points* and *free vectors*. Intuitively, the only thing that distinguishes one point from another is its position. In more computer-sciencey jargon, points only have a position attribute. Free vectors on the other hand have the attributes of magnitude and direction, but no fixed position; the modifier “free” therefore refers to the ability of vectors to move about in the space. Free vectors will henceforth be referred to simply as vectors.

Geometrically we draw points such as P and Q as dots, and we draw vectors such as \vec{v} and \vec{w} as line segments with arrow heads (see Figure 3.3). (To avoid confusion about which symbols are points and which are vectors, we will conform to the convention that points will be written in upper case and vectors will be written in lower case and will be ornamented with a diacritical arrow.)

More formally, an affine space \mathcal{A} is a pair $(\mathcal{P}, \mathcal{V})$ where \mathcal{P} is the set of points and \mathcal{V} is the set of vectors. We shall use the notation $\mathcal{A}.\mathcal{P}$ and $\mathcal{A}.\mathcal{V}$ to refer to the points and vectors of an affine space \mathcal{A} . The vectors of an affine space are assumed to form a vector space. If n denotes the dimension of the vector space, then the affine space is called an affine n -space. An affine 1-space is more commonly called an *affine line*, and an affine 2-space is more commonly called an *affine plane*.

The set of points and the vector space of an affine space \mathcal{A} are

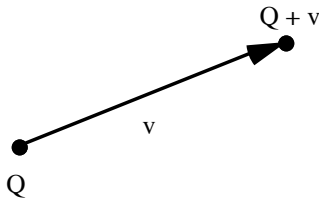


FIG. 3.4: Addition of points and vectors.

related through the following axioms:

- (i) *Subtraction*: There exists an operation of subtraction that satisfies:
 - a. For every pair of points P, Q , there is a unique vector \vec{v} such that $\vec{v} = P - Q$.
 - b. For every point Q and every vector \vec{v} , there is a unique point P such that $P - Q = \vec{v}$.
- (ii) *The Head-to-Tail Axiom*: Every triple of points P, Q and R , satisfies

$$(P - Q) + (Q - R) = P - R.$$

Before describing in more detail what the axioms mean geometrically, it is convenient to use them to define the operation of addition between points and vectors. Specifically, we define $Q + \vec{v}$ to be the unique point P such that $P - Q = \vec{v}$. The geometric interpretation of addition is shown in Figure 3.4.

In terms of the addition operation, axiom (ia) essentially states that there are no “points at infinity”, and axiom (ib) guarantees that there are no “holes” in the space; together these ensure that if point Q is fixed, then there is a one-to-one correspondence between vectors (\vec{v}) and points ($Q + \vec{v}$). The vector connecting the points Q and P can therefore be labeled as $P - Q$, as shown in Figure 3.5.

The geometric interpretation of axiom (ii), shown in Figure 3.6, indicates that the axiom is actually a statement of the familiar “head-to-tail rule” for vector addition, stated in terms of points rather than vectors. (Recall from elementary vector analysis that the vector addition $\vec{v} + \vec{w}$ can be constructed geometrically by aligning the head

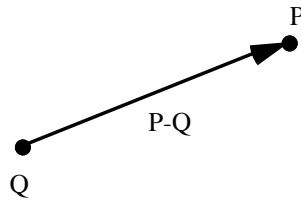


FIG. 3.5: Subtraction of points.

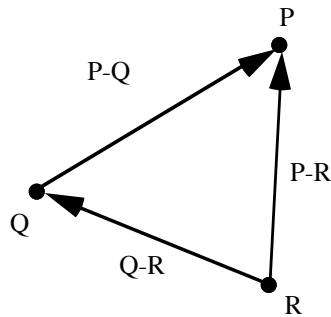


FIG. 3.6: The head-to-tail axiom.

of \vec{v} with the tail of \vec{w} . The sum is then the vector from the tail of \vec{v} to the head of \vec{w} .)

EXAMPLE 1. Examples of affine spaces abound. For instance, if you believe that time is infinite, then the time line is an example of a (one-dimensional) affine space. The points of the affine space correspond to dates, and the vectors of the affine space correspond to numbers of days. A date (a point) minus another date is a number of days (a vector). Thus, subtraction of points makes sense as a vector. The other axioms can also be shown to hold.

The theory of polynomials can be used as the source of another example of an affine space. Let the set of vectors be the set of homogeneous cubic polynomials (a polynomial is said to be homogeneous if its constant coefficient is zero). The set of points can then be taken to be the set of cubic polynomials whose constant term is 1. It is a simple matter to show that the axioms hold when

standard polynomial addition and subtraction are used to add points to vectors and to subtract points. The dimension of this affine space is 3 since that is the dimension of the space of homogeneous cubic polynomials.

Several simple deductions can be made from the head-to-tail axiom. By setting $Q = R$, we find that $(P - Q) + (Q - Q) = P - Q$, which implies that $Q - Q$ must be the zero vector $\vec{\mathbf{0}}$ since adding it to $P - Q$ results in $P - Q$. By setting $P = R$, we see that $(R - Q) + (Q - R) = \vec{\mathbf{0}}$, implying that $R - Q = -(Q - R)$. These facts, along with several others, are summarized in the following claim.

CLAIM 1. *The following identities hold for all points P , Q and R , and all vectors \vec{v} and \vec{w} .*

(a) $Q - Q = \vec{\mathbf{0}}$.

(b) $R - Q = -(Q - R)$.

(c) $\vec{v} + (Q - R) = (Q + \vec{v}) - R$.

(d) $Q - (R + \vec{v}) = (Q - R) - \vec{v}$.

(e) $P = Q + (P - Q)$.

(f) $(Q + \vec{v}) - (R + \vec{w}) = (Q - R) + (\vec{v} - \vec{w})$.

PROOF: Parts (a) and (b) were proved above. To prove (c), let point P be defined by $\vec{v} = P - Q$. The head-to-tail axiom then says that $\vec{v} + (Q - R) = P - R$. The proof is completed by substituting $Q + \vec{v}$ for P . Part (d) follows immediately from (c) by multiplying through by -1 . To prove (e), use the definition of addition together with the head-to-tail axiom to write P in the form $P = R + (P - Q) + (Q - R)$. Now, taking $Q = R$ we find that $P = Q + (P - Q) + \vec{\mathbf{0}}$, which completes the proof since adding the zero vector to the right side has no affect.

The proof of (f) is somewhat more difficult as it requires the two invocations of the head-to-tail axiom together with the use of parts

(a), (c) and (d):

$$\begin{aligned}
 (Q + \vec{v}) - (R + \vec{w}) & \\
 &= [(Q + \vec{v}) - R] + [R - (R + \vec{w})] && \text{by head-to-tail axiom} \\
 &= [(Q + \vec{v}) - R] + [(R - R) - \vec{w}] && \text{by part (d)} \\
 &= [(Q + \vec{v}) - R] - \vec{w} && \text{by part (a)} \\
 &= [(Q + \vec{v}) - Q] + [Q - R] - \vec{w} && \text{by head-to-tail axiom} \\
 &= [\vec{v} + (Q - Q)] + [Q - R] - \vec{w} && \text{by part (c)} \\
 &= (Q - R) + (\vec{v} - \vec{w}) && \text{by part (a)}
 \end{aligned}$$

□

Thus far, the objects in the algebra are space, point, vector, and scalar, and the operations are

$$\begin{aligned}
 \text{vector} + \text{vector} &\mapsto \text{vector} \\
 \text{scalar} * \text{vector} &\mapsto \text{vector} \\
 \text{point} - \text{point} &\mapsto \text{vector} \\
 \text{point} + \text{vector} &\mapsto \text{point}.
 \end{aligned}$$

For each object in the algebra there should be a corresponding data type in the ADT, and for each operation in the algebra there should be a corresponding procedure. We shall refer to the data types as Space, Point, Vector, and Scalar. The Vector and Point types can be tagged with the space in which they reside, making possible a wide range of geometric type checking. The procedures of the ADT thus far can be summarized as:

- Space \leftarrow SCreate(name:string, dim:integer)
Return an affine space of dimension dim. The name of the space is used for debugging purposes. Any number of spaces can be dynamically created.
- Vector \leftarrow VVAdd(v, w : Vector)
Return the vector sum of v and w. An error is signaled if v and w reside in different spaces.
- Vector \leftarrow SVMult(s : Scalar; v : Vector)
Return the vector v scaled by s.
- Vector \leftarrow PPDiff(p1, p2 : Point)
Return the vector p1-p2. An error is signaled if p1 and p2 reside in different spaces.

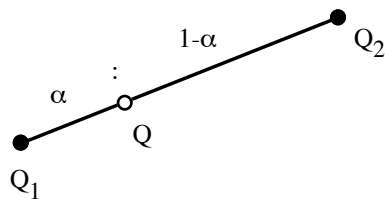


FIG. 3.7: Geometric interpretation of Equation 3.1.

- `Point ← PAdd(p : Point; v : Vector)`
Return the point $p+v$. An error is signaled if p and v reside in different spaces.

These routines are obviously not sufficient by themselves. In particular, there is currently no routines for creating Points and Vectors. These creation routines, `PCreate()` and `VCreate()`, are discussed in Section 3.4.

Notice the asymmetry in the way points and vectors are handled in the algebra. In particular, notice that it is possible to add vectors, but addition of points is not defined. Similarly, the process of multiplying a point by a scalar is undefined. The asymmetry should not be too surprising since points and vectors are being used in very different ways. In some respects the points are the primary objects of the geometry, whereas the role of the vectors is to allow movement from point to point by employing the operation of addition between points and vectors. In Section 3.4, we will see that the vectors are also used to introduce coordinates.

Although the addition of points may be forbidden, there are other convenient operations that can be defined. For instance, consider the expression

$$Q = Q_1 + \alpha(Q_2 - Q_1), \quad (3.1)$$

where Q_1, Q_2 are points and α is a scalar. This expression is meaningful in the context of our algebra because $Q_2 - Q_1$ is meaningful as a vector, implying that $\alpha(Q_2 - Q_1)$ is meaningful as a vector, implying that Q is meaningful as a point since it is the result of adding a point and a vector. Geometrically this means that point Q is one α th along the way from the point Q_1 to the point Q_2 , as shown in Figure 3.7.

If we forget for a moment that we are dealing with points, vectors, and scalars, we might be tempted to algebraically rearrange Equation 3.1 into the form

$$Q = (1 - \alpha)Q_1 + \alpha Q_2,$$

or perhaps in the more symmetric form

$$Q = \alpha_1 Q_1 + \alpha_2 Q_2, \quad \alpha_1 + \alpha_2 = 1. \quad (3.2)$$

This equation looks a bit odd since it appears that we are multiplying points by scalars (an undefined operation), then adding the result together (also undefined). We can formally get out of this bind by making a new definition.

DEFINITION 3.2.1. *The expression*

$$\alpha_1 Q_1 + \alpha_2 Q_2, \quad (3.3)$$

where $\alpha_1 + \alpha_2 = 1$ is defined to be the point

$$Q_1 + \alpha_2(Q_2 - Q_1).$$

An expression such as Equation 3.3 is called an *affine combination*. Affine combinations possess simple geometric interpretations. In particular, Equation 3.3 states that the point Q lies on the line segment Q_1, Q_2 so as to break the segment into relative distances $\alpha_2 : \alpha_1$, as shown in Figure 3.8. Conversely, if a point Q is known to break a line segment Q_1, Q_2 into relative ratios $a : b$, then Q can be expressed as

$$Q = \frac{bQ_1 + aQ_2}{a + b},$$

where for the sake of generality we have not assumed that a and b sum to one.

Affine combinations are supported in the ADT by the routine:

- Point \leftarrow PPac(P, Q : Point; a, b : Scalar)
The Point

$$a P + b Q.$$

is returned.

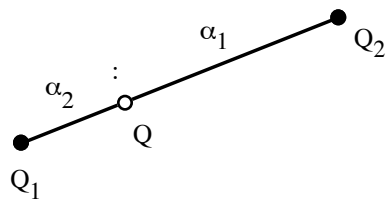


FIG. 3.8: Point Q breaks Q_1Q_2 into relative ratios $\alpha_2 : \alpha_1$.

The notion of an affine combination can be generalized to allow the combination of an arbitrary number of points. If Q_1, \dots, Q_k are points and $\alpha_1, \dots, \alpha_k$ are real numbers that sum to unity, then

$$\alpha_1 Q_1 + \alpha_2 Q_2 + \alpha_3 Q_3 + \dots + \alpha_k Q_k$$

is defined to be the point

$$Q_1 + \alpha_2(Q_2 - Q_1) + \alpha_3(Q_3 - Q_1) + \dots + \alpha_k(Q_k - Q_1). \quad (3.4)$$

The definition of affine combinations using Equation 3.4 is somewhat unnatural, as it treats Q_1 differently than the other points. It is therefore possible that the point obtained from Equation 3.4 might be different if the roles of Q_1 , and say, Q_2 were switched. Fortunately, this is not the case – the definition is independent of which point is used in place of Q_1 . The proof of this independence is the subject of Exercise 3.

REMARK: As an aside of interest to the purist, we mention another approach to affine geometry. An affine space can be defined as a set S that is closed under affine combinations. The points of the affine space are the elements of S ; the vectors are then defined to be equivalence classes of ordered pairs of points. The equivalence relation is constructed to build in the head-to-tail axiom. In particular, two pairs of points (Q_1, P_1) and (Q_2, P_2) are said to be equivalent if

$$\frac{Q_1 + P_2}{2} = \frac{Q_2 + P_1}{2}.$$

This condition has the geometric interpretation that $Q_1P_1P_2Q_2$ must form a parallelogram, as shown in Figure 3.9. It is not too hard to show that this condition is an equivalence relation on the ordered

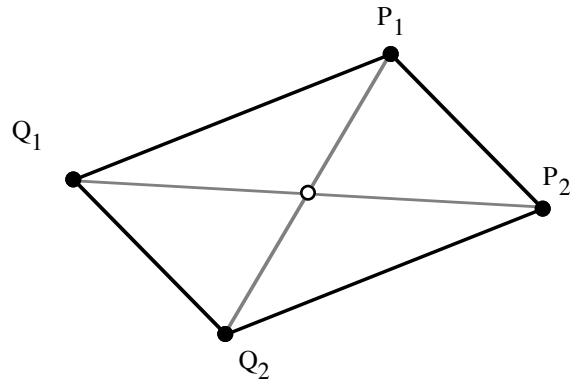


FIG. 3.9: Points $Q_1P_1P_2Q_2$ forming a parallelogram.

pairs of points, implying that the set of all ordered pairs of points are partitioned into equivalence classes. If $[Q, P]$ denotes the equivalence class containing the pair (Q, P) , then the set of all equivalence classes form a vector space, with scalar multiplication and addition defined as:

$$\alpha[Q, P] = [Q, (1 - \alpha)Q + \alpha P], \quad \alpha \in \mathfrak{R} \quad (3.5)$$

$$[Q_1, P_1] + [Q_2, P_2] = [Q_1, P_1 + P_2 - Q_2]. \quad (3.6)$$

The elements of the vector space thus formed are the vectors of the affine space. \square

3.3. Euclidean Geometry

In affine geometry metric concepts such as absolute length, distance, and angles are not defined. This is demonstrated by the fact that up to this point we have not used these concepts in the development of affine geometry. However, in graphics and computer aided design, it is often necessary to represent metric information, for without this information it is not possible to define right angles or to distinguish circles from ellipses.

When metric information is added to an affine space, the result is the familiar concept of a Euclidean space. In other words, a Euclidean space is a special case of an affine space in which

it is possible to measure absolute distances, lengths, and angles. Consequently, all results obtained for affine spaces also hold in Euclidean spaces. As simple examples, every triple of points in a Euclidean space obey the head-to-tail axiom, and the points of a Euclidean space are closed under affine combinations

3.3.1. The Inner Product In keeping with our algebraic approach to geometry, we shall incorporate metric knowledge by introducing a new algebraic entity called an *inner product*. An inner product for an affine space \mathcal{A} is a function that maps a pair of vectors in $\mathcal{A}\mathcal{V}$ into the reals. Rather than using a notation such as $f(\vec{u}, \vec{v})$ to denote an inner product, we use the more familiar form $\langle \vec{u}, \vec{v} \rangle$. Such a bi-variate function must possess the following properties to achieve the status of an inner product:

(i) *Symmetry*: For every pair of vectors \vec{u}, \vec{v} , $\langle \vec{u}, \vec{v} \rangle = \langle \vec{v}, \vec{u} \rangle$.

(ii) *Bi-linearity*: For every $\alpha, \beta \in \mathfrak{R}$ and for every $\vec{u}, \vec{v}, \vec{w} \in \mathcal{A}\mathcal{V}$,

- $\langle \alpha\vec{u} + \beta\vec{v}, \vec{w} \rangle = \alpha\langle \vec{u}, \vec{w} \rangle + \beta\langle \vec{v}, \vec{w} \rangle$.
- $\langle \vec{u}, \alpha\vec{v} + \beta\vec{w} \rangle = \alpha\langle \vec{u}, \vec{v} \rangle + \beta\langle \vec{u}, \vec{w} \rangle$.

(iii) *Positive Definiteness*: For every $\vec{v} \in \mathcal{A}\mathcal{V}$, $\langle \vec{v}, \vec{v} \rangle > 0$ if \vec{v} is not the zero vector, and $\langle \vec{0}, \vec{0} \rangle = 0$.

A Euclidean space \mathcal{E} can now be defined as an affine space together with a distinguished inner product; that is, $\mathcal{E} = (\mathcal{A}, \langle, \rangle)$. To conform more closely with standard practice, the inner product associated with a particular Euclidean space will generally be denoted by \cdot , and will generally be referred to as the dot product. Thus, we write $u \cdot v$ to stand for $\langle u, v \rangle$.

The dot product is used to define length, distance, and angles as follows:

- *The length of a vector*:

$$|\vec{v}| := \sqrt{\vec{v} \cdot \vec{v}}.$$

- *The distance between two points*:

$$\text{Dist}(P, Q) := |P - Q|.$$

- *The angle between two vectors*:

$$\text{Angle}(\vec{v}, \vec{w}) := \cos^{-1} \left(\frac{\vec{v} \cdot \vec{w}}{|\vec{v}| |\vec{w}|} \right).$$

Associated with every non-zero vector \vec{v} is a unique vector \hat{v} having unit length that points in the same direction as \vec{v} . The vectors \vec{v} and \hat{v} are, of course, related by

$$\hat{v} := \frac{\vec{v}}{|\vec{v}|}.$$

The definition of angles allows us to define the notion of *perpendicularity* or *orthogonality*. In particular, two vectors \vec{v} and \vec{w} are said to be perpendicular (or orthogonal) if $\vec{v} \cdot \vec{w} = 0$. We can also define the vectors to be *parallel* if $\hat{v} \cdot \hat{w} = 1$, and *anti-parallel* if $\hat{v} \cdot \hat{w} = -1$.

In the important special case of Euclidean 3-spaces, it is convenient to define another operation on vectors, namely the *cross product*. Given a pair of vectors \vec{v} and \vec{w} from a Euclidean 3-space, we define \times by the equation

$$\vec{v} \times \vec{w} = |\vec{v}| |\vec{w}| \sin \theta \hat{n},$$

where θ is the angle between the vectors and \hat{n} is the unique unit vector that is perpendicular to \vec{v} and \vec{w} such that \vec{v} , \vec{w} and \hat{n} satisfy the “right hand rule.”

Since Euclidean spaces are so useful in practical applications, we have found it convenient to make the convention that the Space datatype actually represents a Euclidean space. That is, in the code fragment

```
Space Screen;
World := SCreate( "World", 3);
```

the variable World is a Euclidean space, meaning that it comes pre-equipped with an inner product. Thus, if v and w are Vectors in World, then VVDot(v,w) returns v·w. Also defined is a routine VVCross(v,w) that returns v×w.

3.4. Frames

To perform numerical computations and to facilitate the creation of geometric entities, we must understand how affine spaces are coordinatized. In this section, we give two methods for imposing coordinates on affine spaces: frames and simplexes. Each method has its advantages, but we have chosen to use frames in the ADT

since they are more familiar to those used to traditional approaches to geometric programming.

Let $\mathcal{A} = (\mathcal{P}, \mathcal{V})$ be an affine n -space, let \mathcal{O} be any point, and let $\vec{v}_1, \dots, \vec{v}_n$ be any basis for $\mathcal{A}\mathcal{V}$. We call the column tuple

$$(\vec{v}_1, \dots, \vec{v}_n, \mathcal{O})^T = \begin{pmatrix} \vec{v}_1 \\ \vdots \\ \vec{v}_n \\ \mathcal{O} \end{pmatrix}$$

a *frame* for \mathcal{A} . Frames play the same role in affine geometry that bases play in vector spaces. The role of frames is more precisely indicated by the next claim.

CLAIM 2. *If $\mathcal{F} = (\vec{v}_1, \dots, \vec{v}_n, \mathcal{O})^T$ is a frame for some affine n -space, then every vector \vec{u} can be written uniquely as*

$$\vec{u} = u_1\vec{v}_1 + u_2\vec{v}_2 + \dots + u_n\vec{v}_n, \quad (3.7)$$

and every point P can be written uniquely as

$$P = p_1\vec{v}_1 + p_2\vec{v}_2 + \dots + p_n\vec{v}_n + \mathcal{O}. \quad (3.8)$$

The sets of scalars (u_1, u_2, \dots, u_n) and (p_1, p_2, \dots, p_n) are called the affine coordinates of \vec{u} and P relative to \mathcal{F} .

PROOF: The unique representation of \vec{u} follows from the fact that $(\vec{v}_1, \dots, \vec{v}_n)$ forms a basis for $\mathcal{A}\mathcal{V}$. From the definition of addition between points and vectors, there is a unique vector \vec{w} such that

$$P = \vec{w} + \mathcal{O}.$$

Since $(\vec{v}_1, \dots, \vec{v}_n)$ is a basis for $\mathcal{A}\mathcal{V}$, \vec{w} has a unique representation

$$\vec{w} = p_1\vec{v}_1 + p_2\vec{v}_2 + \dots + p_n\vec{v}_n.$$

Thus, P can be expressed uniquely as

$$P = p_1\vec{v}_1 + p_2\vec{v}_2 + \dots + p_n\vec{v}_n + \mathcal{O}. \quad \square$$

The notions of unit vectors and orthogonality allow the identification of an important kind of frame for Euclidean spaces. A frame

$(\vec{e}_1, \dots, \vec{e}_n, \mathcal{O})^T$ is said to be a *Cartesian frame* if the basis vectors are *ortho-normal*; that is, if the basis vectors satisfy

$$\vec{e}_i \cdot \vec{e}_j = \begin{cases} 1 & \text{if } i = j; \\ 0 & \text{otherwise} \end{cases}$$

Support for frames can be added to the ADT by introducing a new Frame data type, together with the routines:

- **Frame** \leftarrow FCreate(name : **string**; O : Point; v1,...,vk : Vector)
Returned is a new frame whose origin is O and whose basis vectors are v1,...,vk. An error is signaled if (a) the points and vectors do not reside in a common space, or (b) if the vectors do not form a basis. The name field is intended to be used for debugging purposes.

- **Point** \leftarrow PCreate(f : Frame; c1,...,ck : Scalar)
Denoting the origin of f as f.org and the basis vectors as f.v1,...,f.vk, the Point

$$f.org + c1 * f.v1 + \dots + ck * f.vk$$

is returned.

- **Vector** \leftarrow VCreate(f : Frame; c1,...,ck : Scalar)
The Vector

$$c1 * f.v1 + \dots + ck * f.vk$$

is returned.

- (c1,...,ck : Scalar) \leftarrow PCoords(p : Point; f : Frame)
Return the coordinates of p relative to f. An error is signaled if p and f do not reside in a common Space.

- (c1,...,ck : Scalar) \leftarrow VCoords(v : Vector; f : Frame)
Return the coordinates of v relative to f. An error is signaled if v and f do not reside in a common Space.

- **Point** \leftarrow FOrg(f : Frame)
Return the origin of the frame f.

- **Vector** \leftarrow Fv(f : Frame; i : **integer**)
Return the i-th basis vector of f (numbered starting at one).

There is still a sort of chicken and egg problem with the creation of Points and Vectors in the ADT. Points and Vectors can be created if one has access to a Frame, but to create Frames one must have access to a Point and a collection of Vectors forming a basis. This apparent circularity can be broken by making the convention that when a Space is created with `SCreate()`, it comes pre-equipped with a Frame known as the “standard frame.” If `S` is a Space, its standard frame can be accessed as `StdFrame(S)`.

EXAMPLE 2. Consider the code fragment shown in Figure 3.10(a), the geometric interpretation of which is shown in Figure 3.10(b). Although the example is somewhat contrived, it does serve to illustrate a number of important points:

1. The Frame `f2` is not Cartesian.
 2. Although `P` and `Q` are created with respect to different Frames, the system is able to determine that `f1` and `f2` span the same space, and hence `P` and `Q` reside in the same space. It therefore makes geometric sense to construct the midpoint `M` of the line segment `PQ`. The system is responsible for the bookkeeping required to construct a valid representation for `M`.
 3. Since `M` is known to reside in `S`, its coordinates relative to any Frame for `S` can be extracted. The first print statement will produce `(1.25, 1.0)`, and the second print statement will produce `(-1.5, 1)`.
-

Equation 3.8 can be written in a more symmetric form as an affine combination of $n+1$ points. Let $Q_i = \mathcal{O} + \vec{v}_i$ for $i = 1, \dots, n$, set Q_0 equal to \mathcal{O} , and let $p_0 = 1 - (p_1 + \dots + p_n)$. With these definitions, simple rearrangement allows Equation 3.8 to be rewritten as

$$P = p_0 Q_0 + p_1 Q_1 + \dots + p_n Q_n, \quad (3.9)$$

where, by construction, $p_0 + p_1 + \dots + p_n = 1$. Since every point can be written uniquely in the form of Equation 3.8, every point can also be written uniquely in the form of 3.9. In this form, the scalars (p_0, \dots, p_n) are called the *barycentric coordinates* of P relative to the n -simplex Q_0, \dots, Q_n . An n -simplex is a collection of $n+1$ points such that none of the points can be expressed as an affine combination

```

S : Space;
f1, f2 : Frame;
P, Q, M : Point;

S := SCreate( "Screen", 2);
f1 := StdFrame(S);
f2 := FCreate( PCreate(f1,1.5,0.5), VCreate(f1,0.5,0), VCreate(f1,0.5,0.5));

{ Create P relative to f1 }
P := PCreate( f1, 0.5, 1.0);

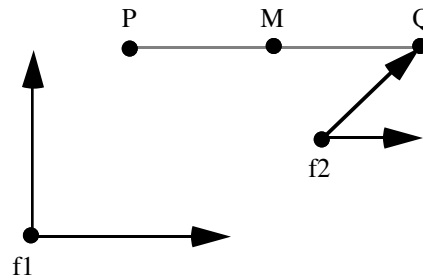
{ Create Q relative to f2 }
Q := PCreate( f2, 0, 1.0);

{ Compute the midpoint M }
M := PPac( P, Q, 0.5, 0.5);

{ Extract coordinates of M relative to f1 and f2 }
print( PCoords( M, f1)); print( PCoords( M, f2));

```

(a)



(b)

FIG. 3.10: A code fragment and its geometric interpretation.

of the others. Thus, a 1-simplex is a line segment, a 2-simplex is a triangle, a 3-simplex is a tetrahedron, and so forth.

Vectors can also be represented in barycentric form as follows. By letting $u_0 = -(u_1 + \dots + u_n)$, Equation 3.7 can be rewritten as

$$\vec{u} = u_0 Q_0 + u_1 Q_1 + \dots + u_n Q_n,$$

where, by construction, $u_0 + u_1 + \dots + u_n = 0$.

Simplexes and barycentric coordinates therefore offer an alternate method of introducing coordinates into an affine space. If the coordinates sum to one, they represent a point; if the coordinates sum to zero, they represent a vector. The notion of barycentric coordinates may at first seem somewhat obscure, but it is actually used in several situations in graphics and CAGD. We will find them useful, for instance, when we consider projective transformations in Section 7.1. Simplexes and barycentric coordinates also have important uses in the theory of Bézier curves and surfaces (cf. [6, 9]).

3.5. *Matrix Representations of Points and Vectors

In the previous section it was shown that points and vectors can be uniquely identified by their coordinates relative to a given frame. The most straightforward way to represent points and vectors in a computer is then to simply store their coordinates as a $1 \times n$ row matrix. However, for reasons that will only become fully apparent later, it is more convenient to augment the row matrix with an additional value that distinguishes between points and vectors [12]. To allow this augmentation to proceed in a rigorous fashion, we extend the original set of axioms for an affine space \mathcal{A} to include:

- (iii) *Coordinate Axiom:* For every point $P \in \mathcal{A}\mathcal{P}$, $0 \cdot P = \vec{0}$, the zero vector of $\mathcal{A}\mathcal{V}$, and $1 \cdot P = P$.

Armed with this axiom, we can rewrite Equation 3.8 in matrix notation as

$$\begin{aligned} P &= p_1 \vec{v}_1 + p_2 \vec{v}_2 + \dots + p_n \vec{v}_n + 1 \cdot \mathcal{O} \\ &= (p_1 \ p_2 \ \dots \ p_n \ 1)(\vec{v}_1 \ \vec{v}_2 \ \dots \ \vec{v}_n \ \mathcal{O})^T. \end{aligned}$$

Notice that the last component in the row matrix essentially says that P is a point, which explains the mystery of the additional

coordinate that was encountered in Section 2.6. Vectors can be represented in a similar fashion by rewriting Equation 3.7 as:

$$\begin{aligned}\vec{u} &= u_1\vec{v}_1 + u_2\vec{v}_2 \cdots u_n\vec{v}_n + 0 \cdot \mathcal{O} \\ &= (u_1 \ u_2 \ \cdots \ u_n \ 0)(\vec{v}_1 \ \vec{v}_2 \ \cdots \ \vec{v}_n \ \mathcal{O})^T.\end{aligned}$$

Thus, vectors are represented as row matrices whose last component is zero.¹

Suppose that a point P has coordinates $(p_1, \dots, p_n, 1)$ relative to a frame $\mathcal{F} = (\vec{v}_1, \dots, \vec{v}_n, \mathcal{O})^T$. It is natural to ask: what are the coordinates of P relative to a frame $\mathcal{F}' = (\vec{v}'_1, \dots, \vec{v}'_n, \mathcal{O}')^T$? To answer this question, we must find scalars p'_1, \dots, p'_n such that

$$p'_1\vec{v}'_1 + \cdots + p'_n\vec{v}'_n + \mathcal{O}' = p_1\vec{v}_1 + \cdots + p_n\vec{v}_n + \mathcal{O}.$$

It is more convenient to write this equation in matrix notation as

$$(p'_1 \ \cdots \ p'_n \ 1) \begin{pmatrix} \vec{v}'_1 \\ \vdots \\ \vec{v}'_n \\ \mathcal{O}' \end{pmatrix} = (p_1 \ \cdots \ p_n \ 1) \begin{pmatrix} \vec{v}_1 \\ \vdots \\ \vec{v}_n \\ \mathcal{O} \end{pmatrix}. \quad (3.10)$$

Each of the elements of \mathcal{F} can be written in coordinates relative to \mathcal{F}' . In particular, let these coordinates be such that:

$$\begin{aligned}\vec{v}_i &= f_{i,1}\vec{v}'_1 + \cdots + f_{i,n}\vec{v}'_n \\ \mathcal{O} &= f_{n+1,1}\vec{v}'_1 + \cdots + f_{n+1,n}\vec{v}'_n + \mathcal{O}'\end{aligned}$$

for $i = 1, \dots, n$. Substituting these equations into Equation 3.10 gives

$$(p'_1 \ \cdots \ p'_n \ 1) \begin{pmatrix} \vec{v}'_1 \\ \vdots \\ \vec{v}'_n \\ \mathcal{O}' \end{pmatrix} =$$

¹Those familiar with homogeneous coordinates are accustomed to adding an additional component when representing points. Note however that the addition of a component has been done here without having to mention homogeneous coordinates or projective spaces. This is not simply a trick, for we are representing affine entities, not projective ones. For instance, in the current context, a row matrix with a final component of zero represents a vector, whereas in projective geometry, a row matrix with a final component of zero represents an *ideal point* (more commonly known as a point at infinity).

$$(p_1 \ \cdots \ p_n \ 1) \begin{pmatrix} f_{1,1}\vec{v}'_1 + \cdots + f_{1,n}\vec{v}'_n \\ \vdots \\ f_{n,1}\vec{v}'_1 + \cdots + f_{n,n}\vec{v}'_n \\ f_{n+1,1}\vec{v}'_1 + \cdots + f_{n+1,n}\vec{v}'_n + \mathcal{O}' \end{pmatrix},$$

which can be rewritten as

$$(p'_1 \ \cdots \ p'_n \ 1) \begin{pmatrix} \vec{v}'_1 \\ \vdots \\ \vec{v}'_n \\ \mathcal{O}' \end{pmatrix} = (p_1 \ \cdots \ p_n \ 1) \begin{pmatrix} f_{1,1} & \cdots & f_{1,n} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ f_{n,1} & \cdots & f_{n,n} & 0 \\ f_{n+1,1} & \cdots & f_{n+1,n} & 1 \end{pmatrix} \begin{pmatrix} \vec{v}'_1 \\ \vdots \\ \vec{v}'_n \\ \mathcal{O}' \end{pmatrix}.$$

Linear independence of the vectors $\vec{v}'_1, \dots, \vec{v}'_n$ can be used to deduce that

$$(p'_1 \ \cdots \ p'_n \ 1) = (p_1 \ \cdots \ p_n \ 1) \begin{pmatrix} f_{1,1} & \cdots & f_{1,n} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ f_{n,1} & \cdots & f_{n,n} & 0 \\ f_{n+1,1} & \cdots & f_{n+1,n} & 1 \end{pmatrix}.$$

Thus, a change of coordinate systems can be accomplished via matrix multiplication. Notice that the matrix used to affect the change of coordinates has rows consisting of the coordinates of the elements of the “old frame” (frame \mathcal{F}) relative to the “new frame” (frame \mathcal{F}').

3.6. Affine Transformations

The next geometric object to be added to our collection is the affine transformation. Affine transformations are mappings between affine spaces that preserve the algebraic structure of the spaces. That is, affine transformations map points to points, vectors to vectors, and under certain conditions, frames to frames.

To begin, let \mathcal{A} and \mathcal{B} be two affine spaces (it is sometimes the case that \mathcal{A} and \mathcal{B} are the same space). A map $F : \mathcal{A} \cdot \mathcal{P} \rightarrow \mathcal{B} \cdot \mathcal{P}$ is said to be an *affine transformation* (also called an affine map) if it preserves affine combinations. That is, F is an affine map if the

condition

$$F(\alpha_1 Q_1 + \cdots + \alpha_k Q_k) = \alpha_1 F(Q_1) + \cdots + \alpha_k F(Q_k) \quad (3.11)$$

holds for all points Q_1, \dots, Q_k and for all sets of α 's that sum to unity. (Notice the similarity between this definition and the definition of linear transformation given in Appendix 1.) Examples of affine transformations include: reflections, shear transformations, translations, rotations, scalings, and orthogonal projections. Perspective projections are not affine transformations, but they are *projective* transformations (see Section 7.1).

EXAMPLE 3. As a specific example of an affine transformation, consider the transformation $T : \mathcal{A.P} \rightarrow \mathcal{A.P}$ that performs translation along a fixed vector \vec{t} . This transformation can be defined by

$$T(P) = P + \vec{t}.$$

To show that T is an affine transformation, it suffices to show that

$$T(\alpha_1 P_1 + \alpha_2 P_2) = \alpha_1 T(P_1) + \alpha_2 T(P_2)$$

for every pair of points P_1, P_2 , and for every α_1, α_2 such that $\alpha_1 + \alpha_2 = 1$. This is not difficult to do, as the following derivation shows:

$$\begin{aligned} T(\alpha_1 P_1 + \alpha_2 P_2) &= (\alpha_1 P_1 + \alpha_2 P_2) + \vec{t} \\ &= P_1 + \alpha_2(P_2 - P_1) + \vec{t} && \text{def of affine comb} \\ &= (P_1 + \vec{t}) + \alpha_2[(P_2 - P_1) + (\vec{t} - \vec{t})] \\ &= (P_1 + \vec{t}) + \alpha_2[(P_2 + \vec{t}) - (P_1 + \vec{t})] && \text{Claim 1(f)} \\ &= T(P_1) + \alpha_2(T(P_2) - T(P_1)) \\ &= \alpha_1 T(P_1) + \alpha_2 T(P_2) && \text{def of affine comb} \end{aligned}$$

Induction on the number of terms in the affine combination can be used to show that T preserves arbitrary affine combinations.

An immediate consequence of their definition is that affine transformations carry line segments to line segments, and hence, planes to planes and hyperplanes to hyperplanes. This can be seen

by noting that the line segment connecting the points Q_1 and Q_2 can be written in parametric form as the affine combination

$$Q(t) = (1 - t)Q_1 + tQ_2, \quad t \in [0, 1]. \quad (3.12)$$

The image of the segment under an affine map F is therefore

$$F(Q(t)) = (1 - t)F(Q_1) + tF(Q_2), \quad t \in [0, 1], \quad (3.13)$$

which is a parametric description of the line segment connecting the images of Q_1 and Q_2 . Equations 3.12 and 3.13 actually show something substantially stronger. In particular, they show that the point breaking the line segment Q_1, Q_2 into relative ratio $t : (1 - t)$ is mapped to the point that breaks $F(Q_1), F(Q_2)$ into the same relative ratio, as shown in Figure 3.11. We therefore arrive at the important fact that affine maps preserve relative ratios.

Another important fact about affine maps is that they are completely determined if the image of an n -simplex is known. To see this, let $F : \mathcal{A} \cdot \mathcal{P} \rightarrow \mathcal{B} \cdot \mathcal{P}$ be an affine map, let \mathcal{A} be an affine n -space, and let Q_0, \dots, Q_n be an n -simplex in \mathcal{A} . In the previous section it was shown that every point in \mathcal{A} can be written uniquely as $P = p_0Q_0 + \dots + p_nQ_n$. The fact that F is affine implies that $F(P) = p_0F(Q_0) + \dots + p_nF(Q_n)$, which is completely determined if the points $F(Q_0), \dots, F(Q_n)$, i.e., the image of the simplex (Q_0, \dots, Q_n) , are known.

We can push the above argument further to yield another interesting result. Suppose that $S = (Q_0, \dots, Q_n)$ is an arbitrary simplex in \mathcal{A} and that $S' = (Q'_0, \dots, Q'_n)$ is an arbitrary collection of points in \mathcal{B} , not necessarily forming a simplex. We claim that there is a unique affine map from \mathcal{A} to \mathcal{B} that carries S into S' . The proof is immediate: existence follows by letting the map F in the previous paragraph be such that $F(Q_i) = Q'_i$, for $i = 0, \dots, n$; uniqueness follows from the fact that every point has a unique barycentric representation. In the case of affine planes (occurring when both \mathcal{A} and \mathcal{B} are affine two-spaces), this result says that every pair of triangles are related by a unique affine map. Similarly, every pair of tetrahedrons in an affine 3-space are related by a unique affine map.

In the introduction to this section it was claimed that affine transformations carry points to points and vectors to vectors. However, affine transformations are currently defined only on points.

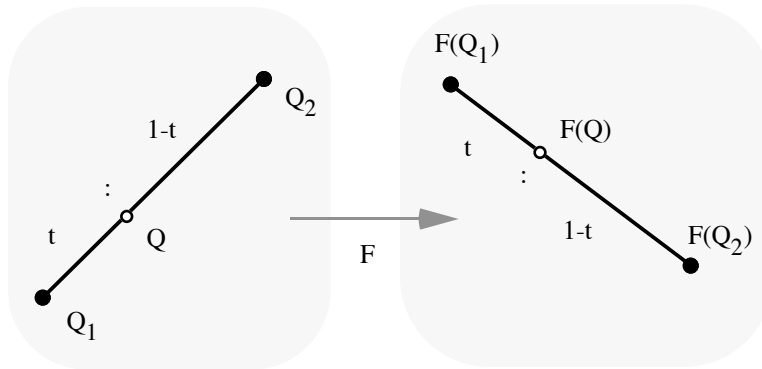


FIG. 3.11: The action of an affine map on a line segment.

Fortunately, we can extend their domains to include the vectors as well. Let $F : \mathcal{A}.\mathcal{P} \rightarrow \mathcal{B}.\mathcal{P}$ be an affine map, let \vec{v} be any vector in $\mathcal{A}.\mathcal{V}$, and let P and Q be any two points in $\mathcal{A}.\mathcal{P}$ such that $\vec{v} = P - Q$. We define $F(\vec{v})$ to be the vector in $\mathcal{B}.\mathcal{V}$ given by $F(P) - F(Q)$. In equation form,

$$F(\vec{v}) \equiv F(P - Q) := F(P) - F(Q).$$

Notice that the points P and Q used in the definition are not unique in that there are many pairs of points whose difference is \vec{v} . To verify that the definition of $F(\vec{v})$ is well-formed, it must be shown that if P, Q and P', Q' are two pairs of points whose difference is \vec{v} , then $F(P) - F(Q) = F(P') - F(Q')$. We leave the proof as an exercise.

Since the domain of an affine map such as $F : \mathcal{A}.\mathcal{P} \rightarrow \mathcal{A}.\mathcal{P}$ can be extended to include $\mathcal{A}.\mathcal{V}$, we consider F being defined on all of \mathcal{A} , and hence we write simply $F : \mathcal{A} \rightarrow \mathcal{B}$.

Using the definition of the action of an affine map on vectors, it is also not difficult to show that F is a linear transformation on the set of vectors. That is, F satisfies

$$F(u_1\vec{v}_1 + \cdots + u_n\vec{v}_n) = u_1F(\vec{v}_1) + \cdots + u_nF(\vec{v}_n), \quad (3.14)$$

for all u_1, \dots, u_n and for all $\vec{v}_1, \dots, \vec{v}_n$. The proof of this fact is rather instructive in that it demonstrates a use of the head-to-tail axiom. We will show that F satisfies the following two conditions:

1. $F(\vec{v} + \vec{w}) = F(\vec{v}) + F(\vec{w})$.

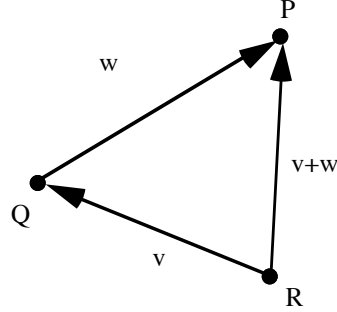


FIG. 3.12: The points P, Q , and R in the proof of condition 1.

2. $F(\alpha\vec{v}) = \alpha F(\vec{v})$.

Equation 3.14 can then be shown by using induction on the number of terms in the sum. To prove condition 1, let P, Q and R be points such that $\vec{v} = Q - R$ and $\vec{w} = P - Q$, as shown in Figure 3.12. By the head-to-tail axiom, $F(\vec{v} + \vec{w}) = F(P) - F(R)$. Using the head-to-tail axiom again in the range gives the desired result:

$$\begin{aligned} F(\vec{v} + \vec{w}) &= F(P) - F(R) \\ &= [F(P) - F(Q)] + [F(Q) - F(R)] \\ &= F(\vec{v}) + F(\vec{w}). \end{aligned}$$

To prove condition 2, we note that the vector $\alpha\vec{v}$ can be written as

$$\alpha\vec{v} = [(1 - \alpha)R + \alpha Q] - R.$$

The desired result can now be achieved in just a few steps:

$$\begin{aligned} F(\alpha\vec{v}) &= F([(1 - \alpha)R + \alpha Q] - R) \\ &= F([(1 - \alpha)R + \alpha Q]) - F(R) \\ &= (1 - \alpha)F(R) + \alpha F(Q) - F(R) \\ &= \alpha F(\vec{v}). \end{aligned}$$

Next, we show that F also satisfies

$$F(Q + \vec{v}) = F(Q) + F(\vec{v})$$

for every point Q and every vector \vec{v} . To do this, let P be such that $\vec{v} = P - Q$. Thus, $F(Q + \vec{v}) = F(Q + P - Q)$. Since the expression $Q + P - Q$ is an affine combination,

$$\begin{aligned} F(Q + P - Q) &= F(Q) + F(P) - F(Q) \\ &= F(Q) + [F(P) - F(Q)] \\ &= F(Q) + F(\vec{v}). \end{aligned}$$

Putting these facts together reveals that F preserves affine coordinates:

$$F(p_1\vec{v}_1 + \cdots + p_n\vec{v}_n + \mathcal{O}) = p_1F(\vec{v}_1) + \cdots + p_nF(\vec{v}_n) + F(\mathcal{O}), \quad (3.15)$$

showing that affine maps are completely determined once the image of a frame is known.

Affine transformations can be manipulated in the geometric ADT by adding an `AffineMap` data type. The fact found above that an affine map is completely determined once its action on a frame is known can be used as the basis of a coordinate-free method of specifying affine transformations. Specifically, we add to the ADT the routines:

- `AffineMap` \leftarrow `ACreate`(`f` : `Frame`; `O` : `Point`, `v1...vk` : `Vector`)
This is the most general affine map creation routine. Let S denote the space for which f is a frame, and let k denote S 's dimension. The point O and the vectors v_1, \dots, v_k must reside in a common space S' ; if they do not, an error is signaled. Returned is the (unique) affine map that carries $f.org$ to O , $f.v_1$ to v_1 , etc.
- `Point` \leftarrow `PXform`(`P` : `Point`; `T` : `AffineMap`)
If P resides in the domain space of T , then the image point $T(P)$ is returned; otherwise an error is signaled.
- `Vector` \leftarrow `VXform`(`v` : `Vector`; `T` : `AffineMap`)
If V resides in the domain space of T , then the image vector $T(v)$ is returned; otherwise an error is signaled.

EXAMPLE 4.

To demonstrate the use of the low-level affine map creation routine `ACreate()` we use it to construct a higher-level routine that

```

AffineMap Rotate2D( P : Point,  $\theta$  : Scalar)
{ Return a rotation about P by an angle  $\theta$  }
begin
  RotateFrame : Frame;
  e1, e2, e1', e2' : Vector;

  { Build the rotation frame }
  e1 := Fv( StdFrame( SpaceOf(P), 1);
  e2 := Fv( StdFrame( SpaceOf(P), 2);
  RotateFrame := FCreate( "rotate", P, e1, e2);

  { Build the images of e1 and e2 }
  e1' := VCreate( RotateFrame, cos( $\theta$ ), sin( $\theta$ ));
  e2' := VCreate( RotateFrame, -sin( $\theta$ ), cos( $\theta$ ));

  { Build and return the transformation }
  return ACreate( RotateFrame, P, e1', e2');
end

```

FIG. 3.13: The definition of a two-dimensional rotation operator using ACreate. The routine SpaceOf() is a polymorphic function that returns the space in which its argument (a Point, Vector, etc.) resides.

returns a affine map that represents rotation by an angle θ about an arbitrary point P in a two-dimensional space S. This is easily accomplished by creating a new frame called RotateFrame whose origin is P and whose basis vectors e1 and e2 are inherited from StdFrame(S). RotateFrame is chosen in this way because it is clear how its elements transform under the rotation. In particular, P maps to P, and e1 and e2 transform as:

$$\begin{aligned} e1 &\mapsto \cos(\theta)e1 + \sin(\theta)e2 \\ e2 &\mapsto -\sin(\theta)e1 + \cos(\theta)e2 \end{aligned}$$

The pseudo-code to carry out this process is shown in Figure 3.13.

If $F : \mathcal{A} \rightarrow \mathcal{B}$ and $G : \mathcal{B} \rightarrow \mathcal{C}$ are affine maps, then the composition map $H = G \circ F : \mathcal{A} \rightarrow \mathcal{C}$ is also an affine map. (See

Exercise 7 on page 65.) We therefore add the following routine to the ADT:

- `AffineMap ← AACompose(F, G : AffineMap)`
Returned is the affine map $G \circ F$. An error is signaled if the domain of G does not match the range of F .

3.7. *Matrix Representations of Affine Transformations

Just as points and vectors can be represented as matrices, so too can affine transformations. For notational simplicity, the following discussion will be restricted to maps between affine planes. This restriction is not limiting since all arguments carry through to affine spaces of arbitrary dimension.

Let \mathcal{A} and \mathcal{B} be two affine planes, let $F : \mathcal{A} \mapsto \mathcal{B}$ be an affine transformation, let $(\vec{v}_1, \vec{v}_2, \mathcal{O}_{\mathcal{A}})$ be a frame for \mathcal{A} , let $(\vec{w}_1, \vec{w}_2, \mathcal{O}_{\mathcal{B}})$ be a frame for \mathcal{B} , and let P be an arbitrary point whose coordinates relative to $(\vec{v}_1, \vec{v}_2, \mathcal{O}_{\mathcal{A}})$ are $(p_1, p_2, 1)$. We ask: what are the coordinates of $F(P)$ relative to $(\vec{w}_1, \vec{w}_2, \mathcal{O}_{\mathcal{B}})$? The solution requires little more than simple manipulation. We begin by expanding P in coordinates and use the fact that F preserves affine coordinates:

$$F(P) = F(p_1\vec{v}_1 + p_2\vec{v}_2 + \mathcal{O}_{\mathcal{A}}) \quad (3.16)$$

$$= p_1F(\vec{v}_1) + p_2F(\vec{v}_2) + F(\mathcal{O}_{\mathcal{A}}). \quad (3.17)$$

Since F carries vectors (points) in \mathcal{A} into vectors (points) in \mathcal{B} , the quantities $F(\vec{v}_1)$ and $F(\vec{v}_2)$ are vectors in \mathcal{B} and the quantity $F(\mathcal{O}_{\mathcal{A}})$ is a point in \mathcal{B} , and as such they each have affine coordinates relative to the frame $(\vec{w}_1, \vec{w}_2, \mathcal{O}_{\mathcal{B}})$. Suppose that

$$F(\vec{v}_1) = f_{1,1}\vec{w}_1 + f_{1,2}\vec{w}_2$$

$$F(\vec{v}_2) = f_{2,1}\vec{w}_1 + f_{2,2}\vec{w}_2$$

$$F(\mathcal{O}_{\mathcal{A}}) = f_{3,1}\vec{w}_1 + f_{3,2}\vec{w}_2 + \mathcal{O}_{\mathcal{B}}$$

Using these coordinates, Equation 3.17 can be manipulated as follows:

$$F(P) = \begin{pmatrix} p_1 & p_2 & 1 \end{pmatrix} \begin{pmatrix} F(\vec{v}_1) \\ F(\vec{v}_2) \\ F(\mathcal{O}_{\mathcal{A}}) \end{pmatrix}$$

$$\begin{aligned}
 &= (p_1 \ p_2 \ 1) \begin{pmatrix} f_{1,1}\vec{w}_1 + f_{1,2}\vec{w}_2 \\ f_{2,1}\vec{w}_1 + f_{2,2}\vec{w}_2 \\ f_{3,1}\vec{w}_1 + f_{3,2}\vec{w}_2 + \mathcal{O}_B \end{pmatrix} \\
 &= (p_1 \ p_2 \ 1) \begin{pmatrix} f_{1,1} & f_{1,2} & 0 \\ f_{2,1} & f_{2,2} & 0 \\ f_{3,1} & f_{3,2} & 1 \end{pmatrix} \begin{pmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \mathcal{O}_B \end{pmatrix} \\
 &= (p_1 \ p_2 \ 1) \mathbf{F} \begin{pmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \mathcal{O}_B \end{pmatrix} \\
 &= (p'_1 \ p'_2 \ 1) \begin{pmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \mathcal{O}_B \end{pmatrix}
 \end{aligned}$$

Thus, the point P with coordinates $(p_1, p_2, 1)$ gets transformed to the point $F(P)$ with coordinates $(p'_1 \ p'_2 \ 1)$, where $(p'_1 \ p'_2 \ 1) = (p_1 \ p_2 \ 1) \mathbf{F}$. For this reason, the matrix \mathbf{F} is called the *matrix representation of F relative to the frames $(\vec{v}_1, \vec{v}_2, \mathcal{O}_A)$ and $(\vec{w}_1, \vec{w}_2, \mathcal{O}_B)$* . Notice that

- The first row of \mathbf{F} is the representation of $F(\vec{v}_1)$.
- The second row of \mathbf{F} is the representation of $F(\vec{v}_2)$.
- The third row of \mathbf{F} is the representation of $F(\mathcal{O}_A)$.

As a consequence, affine maps are represented as matrices whose last column is $(0 \ 0 \ 1)^T$. Conversely, every matrix whose last column is $(0 \ 0 \ 1)^T$ represents some affine transformation.

EXAMPLE 5. As a specific example of the construction of a matrix representation of an affine transformation, consider the construction of a matrix representation of the translation T of Example 3. To do this, we must pick frames in both the domain and the range. Since T maps \mathcal{A} onto itself, the domain and range are the same space, so we may as well pick a single frame $(\vec{v}_1, \vec{v}_2, \mathcal{O})$ to serve double duty. Suppose that in this frame the vector \vec{t} has the coordinates $(a, b, 0)$. All we have to do to determine the matrix \mathbf{T} is to determine what T does to \vec{v}_1 , \vec{v}_2 , and \mathcal{O} .

We'll do the easy part first. The third row of \mathbf{T} consists of the coordinates of $T(\mathcal{O})$, which are $(a, b, 1)$ since

$$T(\mathcal{O}) = \mathcal{O} + \vec{t}$$

$$= a\vec{v}_1 + b\vec{v}_2 + \mathcal{O}.$$

The first row of \mathbf{T} consists of the coordinates of $T(\vec{v}_1)$. To see what these are, let R and S be two points such that $\vec{v}_1 = R - S$. Then, by definition of how affine maps behave on vectors,

$$\begin{aligned} T(\vec{v}_1) &= T(R - S) && \text{by def of } R,S \\ &= T(R) - T(S) && \text{by def } T \text{ on vectors} \\ &= (R + \vec{t}) - (S + \vec{t}) && \text{by def of } T \\ &= R - S && \text{by Claim 1(f)} \\ &= \vec{v}_1 && \text{by def of } R,S. \end{aligned}$$

In other words, T does not affect \vec{v}_1 . In fact, the derivation above works for all vectors, not just \vec{v}_1 , so T does not affect any vector. This means that the first row of \mathbf{T} is $(1 \ 0 \ 0)$, and the second row is $(0 \ 1 \ 0)$. Putting this all together, relative to the frame $(\vec{v}_1, \vec{v}_2, \mathcal{O})$, T is represented by the matrix

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix}.$$

3.8. Ambiguity Revisited

In Section 3.1 it was claimed that the ADT solves the ambiguity problem in that a given code fragment can have one and only one geometric interpretation. As a demonstration of how this is accomplished, we refer again to the code fragment of Figure 3.1, the geometric interpretations of which are shown in Figure 3.2.

Using the geometric ADT, each geometric interpretation is unambiguously reflected in the code. For instance, if the programmer intended a change of coordinates as indicated by Figure 3.2(a), the appropriate code fragment would be something like:

```
frame1, frame2 : Frame;
P : Point;
px, py : Scalar;

P := PCreate( frame1, p1, p2);
(px,py) := PCoords( P, frame2);
```

where `frame1` and `frame2` are two frames having the geometric relationship indicated in Figure 3.2(a). If the programmer was instead intending to effect a transformation on the space as indicated by Figure 3.2(b), the appropriate code would be something like:

```

T : AffineMap;
S : Space;
P', P : Point;
sf : Frame;
    ⋮
sf := StdFrame(S);
T := ACreate( sf, FOrig(sf), SVMult(2,Fv(sf,1)), Fv(sf,2));
    ⋮
P' := PAxform( P, T);
    ⋮

```

Finally, if a transformation between separate spaces is to be applied as indicated by Figure 3.2(c), the code would be something like:

```

T : AffineMap;
S1, S2 : Space;
P', P, O' : Point;
x', y' : Vector;
    ⋮
{ Compute O', x', and y' in S2 }
    ⋮
T := ACreate( StdFrame(S1), O', x', y');
    ⋮
{ Not that P and P' live in different spaces. }
P' := PAxform( P,T);
    ⋮

```

To reiterate, each of the code fragments above has an unambiguous geometric interpretation that is undeniably apparent from

the code. The fact that identical matrix computations are being performed at a lower level is invisible (and irrelevant).

3.9. Coordinate-Free Line Clipping

The use of coordinate-free concepts sometimes requires that problems be solved in ways that at first seem somewhat unnatural. Consider, for instance, the use of coordinate-free techniques to solve the clipping of two-dimensional line segments to the interior of a window. A coordinate-based solution to this problem was given in Section 2.5.1. We must first find a coordinate-free representation of the window. A straightforward representation would be to represent the window by its corner points. A more convenient representation for clipping, however, is to represent the boundary lines of the window rather than the corners. We shall find it most convenient to represent the window as the intersection of a collection of four linear *half-spaces* or *oriented hyperplane* H_1 , H_2 , H_3 , and H_4 . An oriented hyperplane H is a set of points

$$H = \{Q : \vec{n} \cdot (Q - P) \leq 0\}$$

where P is a point on the boundary of the hyperplane and \vec{n} is a vector that points outward and normal (i.e., perpendicular) to the boundary (see Figure 3.14(a)). The fact that inward and outward are distinguished motivates the use of the word “oriented”.

The window W to which a line segment is to be clipped can then be represented as the set of points

$$W = H_1 \cap H_2 \cap H_3 \cap H_4$$

as shown in Figure 3.14(b). To clip a line segment P_1P_2 to the interior of W , we must find the set of points $P_1P_2 \cap W$. Using the associativity of set intersection, we deduce that

$$\begin{aligned} P_1P_2 \cap W &= P_1P_2 \cap (H_1 \cap H_2 \cap H_3 \cap H_4) \\ &= (((((P_1P_2) \cap H_1) \cap H_2) \cap H_3) \cap H_4) \end{aligned}$$

implying that P_1P_2 can be successively clipped to each of the hyperplanes H_1, \dots, H_4 .

To see how to clip a segment P_1P_2 against a single oriented hyperplane H , defined by a point P and an outward normal vector \vec{n} ,

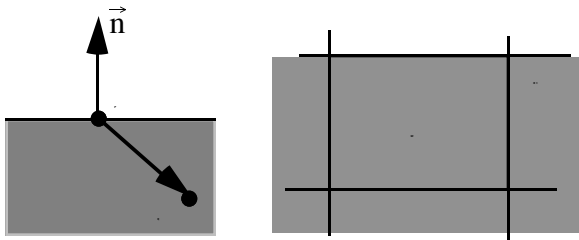


FIG. 3.14: (a) The representation of a half-space as a point P and an outward pointing normal vector \vec{n} ; (b) The representation of the window as the intersection of half-spaces.

it is convenient to think of the hyperplane as a function $H : \text{Points} \rightarrow \Re$ defined as

$$H(Q) = \vec{n} \cdot (Q - P). \quad (3.18)$$

The convenient thing about this definition is that the value $H(Q)$ is proportional to the signed distance of the point Q from the half-space boundary, with the constant of proportionality being the length of \vec{n} . That is, if d_Q is the signed distance of Q to the half-space boundary, then $H(Q) = |\vec{n}|d_Q$.

There are four cases of interest, depending on the signs of $H(P_1)$ and $H(P_2)$. If both are positive, then both endpoints are outside the half-space, meaning that the line segment can be *trivially rejected*. If both are negative, then both endpoints are inside the half-space, meaning that the line segment can be *trivially accepted*. The interesting cases occur when the signs of $H(P_1)$ and $H(P_2)$ differ, implying that one endpoint is inside and the other is outside. Suppose that $H(P_1) > 0$ and that $H(P_2) < 0$ (the other case is symmetric and will not be discussed). We must therefore compute the intersection point I between the line segment and half-space boundary (see Figure 3.15). The intersection I can be computed easily using an affine combination. Using similar triangles we find that $P_1I : IP_2 = H(P_1) : -H(P_2)$, meaning that

$$I = \frac{H(P_1)P_2 - H(P_2)P_1}{H(P_1) - H(P_2)}.$$

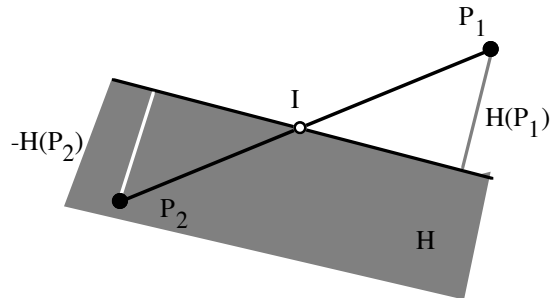


FIG. 3.15: If one endpoint is inside the half-space, and one is outside, the point of intersection I must be computed.

Once I is computed, the subsegment P_1I is discarded, and the subsegment P_2I is output as the result.

The above algorithm is the coordinate-free version of the Sutherland-Hodgman line clipping algorithm [17].

Exercises

1. Show that the definition of the action of F on vectors is well-defined in the sense that if P, Q and P', Q' are two pairs of points such that $P - Q = P' - Q'$, then $F(P) - F(Q) = F(P') - F(Q')$.
2. Let L_1 and L_2 be two lines that pass through the points Q_1, P_1 and Q_2, P_2 respectively. These lines are said to be *parallel* if $P_1 - Q_1$ is a scalar multiple of $P_2 - Q_2$. Show that parallel lines are mapped to parallel lines under affine maps. (A degenerate case can occur when both L_1 and L_2 are mapped to single points.)
3. Show that the definition of affine combinations as given in Equation 3.4 is independent of which point is used in place of Q_1 .
4. Show that scalar multiplication and addition of equivalence classes of points as in Equations 3.5 and 3.6 forms a vector space, then show that the axioms of affine spaces are satisfied.

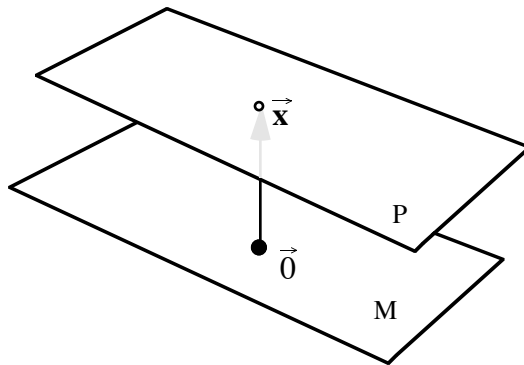


FIG. 3.16: The points \mathcal{P} of an affine space as a translated vector subspace \mathcal{M}

- Let \mathcal{L} be an ℓ -dimensional vector space, let \mathcal{M} be an $m < n$ -dimensional vector subspace of \mathcal{L} , let $\vec{x} \notin \mathcal{M}$ be a vector in \mathcal{L} , and let

$$\mathcal{P} = \{\vec{x} + \vec{v} : \vec{v} \in \mathcal{M}\}.$$

Figure 3.16 depicts the situation for $n = 3$ and $m = 2$. Define subtraction on elements of \mathcal{P} and a set \mathcal{V} such that $(\mathcal{P}, \mathcal{V})$ form an affine space. Prove that the axioms of affine spaces are satisfied by your definition.

This exercise shows that affine spaces can be constructed by translating a vector subspace (\mathcal{M}) away from the origin.

- Write a pseudo-code statement of a procedure `ARotate3D()` that takes as input a point P , a vector v (both assumed to reside in an affine three space), and an angle θ . The procedure should return an `AffineMap` that represents rotation by θ about an axis through P in the direction of v . By convention, positive angles correspond to clockwise rotation when viewed along v .
- Show that if $F : \mathcal{A} \rightarrow \mathcal{B}$ and $G : \mathcal{B} \rightarrow \mathcal{C}$ are affine maps, then the composed map $H = G \circ F : \mathcal{A} \rightarrow \mathcal{C}$ is also an affine map.
- Let $F : \mathcal{A} \rightarrow \mathcal{B}$ be the unique affine map that carries the frame \mathcal{F}_A in \mathcal{A} into the frame \mathcal{F}_B in \mathcal{B} . Show that the matrix representation of F relative to \mathcal{F}_A and \mathcal{F}_B is the

identity matrix. (This implies that the identity matrix does not necessarily represent the identity transformation.)

9. Let $F : \mathcal{A} \rightarrow \mathcal{B}$ and $G : \mathcal{B} \rightarrow \mathcal{C}$ be affine maps, and let $\mathcal{F}_\mathcal{A}$, $\mathcal{F}_\mathcal{B}$, and $\mathcal{F}_\mathcal{C}$ be frames in \mathcal{A} , \mathcal{B} , and \mathcal{C} , respectively. Show that if \mathbf{F} is the matrix representation of F relative to $\mathcal{F}_\mathcal{A}$ and $\mathcal{F}_\mathcal{B}$, and \mathbf{G} is the matrix representation of G relative to $\mathcal{F}_\mathcal{B}$ and $\mathcal{F}_\mathcal{C}$, then \mathbf{FG} is the matrix representation of $G \circ F$.
10. Suppose that an affine map $T : \mathcal{A} \rightarrow \mathcal{B}$ has a matrix representation \mathbf{T} relative to frames $\mathcal{F}_\mathcal{A}$ and $\mathcal{F}_\mathcal{B}$, and suppose that $\mathcal{F}'_\mathcal{A}$ is a frame in \mathcal{A} such that coordinates relative to $\mathcal{F}_\mathcal{A}$ are changed into coordinates relative to $\mathcal{F}'_\mathcal{A}$ by multiplying by a matrix \mathbf{F} . Show that the matrix representation of T relative to $\mathcal{F}'_\mathcal{A}$ and $\mathcal{F}_\mathcal{B}$ is $\mathbf{F}^{-1} \mathbf{T}$.
11. Show that the function $H(Q)$ defined in Equation 3.18 is an affine map from points into real numbers. An affine map that maps into the reals is called an *affine functional*.
12. (Due to Ron Goldman.) Show that every non-singular affine transformation of the affine plane is the composite of one scale, one translation, one rotation, and one shear. A scale transformation is one where $\mathcal{O} \rightarrow \mathcal{O}$, $\vec{v}_1 \rightarrow a\vec{v}_1$, and $\vec{v}_2 \rightarrow b\vec{v}_2$, where $(\vec{v}_1, \vec{v}_2, \mathcal{O})$ is a frame, and where a and b are non-zero scalars. A shear is a transformation such that $\mathcal{O} \rightarrow \mathcal{O}$, $\vec{v}_1 \rightarrow \vec{v}_1$, $\vec{v}_2 \rightarrow a\vec{v}_1 + b\vec{v}_2$.

3.10. A Brief Review of Linear Algebra

A *vector space* over the reals is a set \mathcal{V} , each element of which is called a *vector*, that satisfies the following properties:

- (i) Addition of vectors and multiplication by real numbers (scalars) is defined.
- (ii) The set is closed under linear combinations. That is, if $\vec{v}, \vec{w} \in \mathcal{V}$, and $\alpha, \beta \in \mathfrak{R}$, then $\alpha\vec{v} + \beta\vec{w} \in \mathcal{V}$.
- (iii) There is a unique *zero vector* $\vec{\mathbf{0}} \in \mathcal{V}$, such that

- For every vector $\vec{v} \in \mathcal{V}$, $\vec{\mathbf{0}} + \vec{v} = \vec{v}$.
- For every vector $\vec{v} \in \mathcal{V}$, $0 \cdot \vec{v} = \vec{\mathbf{0}}$.

Some examples of vector spaces are listed below. For each example, think about how multiplication and addition of vectors is defined.

1. $\mathfrak{R}^2 = \{(x, y) | x, y \in \mathfrak{R}\}$.
2. $P^3 =$ the set of all polynomials of degree ≤ 3 .
3. $C([0, 1]) =$ the set of all continuous functions defined on the unit interval.

The vectors $\vec{v}_1, \dots, \vec{v}_k$ are said to be *linearly independent* if

$$c_1\vec{v}_1 + c_2\vec{v}_2 + \cdots + c_k\vec{v}_k = \vec{\mathbf{0}} \Leftrightarrow c_i = 0, \quad i = 1, \dots, k$$

where c_1, \dots, c_k are scalars. Otherwise, the vectors are said to be linearly dependent.

The *dimension* of a vector space is defined to be the largest number of linearly independent vectors. For example, the dimension of \mathfrak{R}^2 , written $\dim \mathfrak{R}^2$ can be shown to be 2, and $\dim P^3$ can be shown to be 4.

A sequence $(\vec{v}_1, \dots, \vec{v}_n)$ of linearly independent vectors in a vector space of dimension n is called a *basis*. As a simple example, a basis for \mathfrak{R}^2 is $((1, 0), (0, 1))$. Another basis for \mathfrak{R}^2 is $((1, 1), (0, 1))$, and a basis for P^3 is the familiar *power basis* $(1, x, x^2, x^3)$. Another familiar basis for P^3 are the cubic *Bernstein polynomials* $(x^3, 3x^2(1-x), 3x(1-x)^2, (1-x)^3)$.

Bases are essential for imposing coordinates on vector spaces. Their importance is underscored by the following theorem.

THEOREM 3.10.1. *Let $(\vec{v}_1, \dots, \vec{v}_n)$ be a basis for a vector space \mathcal{V} . For every $\vec{w} \in \mathcal{V}$, there exists a unique set of scalars c_1, \dots, c_n such that*

$$\vec{w} = c_1\vec{v}_1 + c_2\vec{v}_2 + \cdots + c_n\vec{v}_n. \quad (3.19)$$

The numbers (c_1, \dots, c_n) are called coordinates of \vec{w} relative to the basis $(\vec{v}_1, \dots, \vec{v}_n)$.

Proof. For a rigorous proof, see any standard text in linear algebra such as O’Nan [14].

The two important points about this theorem are: (1) coordinates are always relative to some basis, and (2) relative to a particular basis, the coordinates are unique. It is therefore meaningless to talk about the coordinates of a vector without talking about the basis relative to which the coordinates are taken.

Let \mathcal{V}, \mathcal{W} be vector spaces (in many graphics and CAGD applications \mathcal{V} and \mathcal{W} are the same space). A map $T : \mathcal{V} \mapsto \mathcal{W}$ is a *linear transformation* if for every $\alpha_1, \dots, \alpha_k \in \mathfrak{R}$, and for every $\vec{v}_1, \dots, \vec{v}_k \in \mathcal{V}$,

$$T(\alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + \cdots + \alpha_k\vec{v}_k) = \alpha_1T(\vec{v}_1) + \alpha_2T(\vec{v}_2) + \cdots + \alpha_kT(\vec{v}_k).$$

Exercises

1. For the vector space \mathfrak{R}^2 , what are the coordinates of $(4, 3)$ relative to the basis $((1, 1), (0, 1))$?
2. For the vector space P^3 , what are the coordinates of $2x$ relative to the basis $(1, x, x^2, x^3)$? What are the coordinates of $2x$ relative to the basis $(x^3, 3x^2(1-x), 3x(1-x)^2, (1-x)^3)$?
3. Show that a linear transformation $T : \mathcal{V} \mapsto \mathcal{W}$ carries the zero vector in \mathcal{V} into the zero vector in \mathcal{W} .

Three-Dimensional Wireframe Viewing

4.1. Introduction

In this chapter we consider the construction of a (C language) program to display three-dimensional line segments in *orthographic projection* using the geometric ADT. This program, called `wireframe`, serves as the starting point for the rendering project.

Orthographic projection is defined as follows. A point Q in a three-dimensional space \mathcal{A} is orthographically projected onto a *projection plane* $\pi \in \mathcal{A}$ by constructing a line, called a *projector* through Q perpendicular to π . The image point Q' of Q under the projection is the point where the projector intersects π (see Figure 4.1). This defines a mapping $P : \mathcal{A} \rightarrow \mathcal{A}$ that actually turns out to be an affine map (see Exercise 1 on page 81).

The other important type of projection is *perspective projection*,

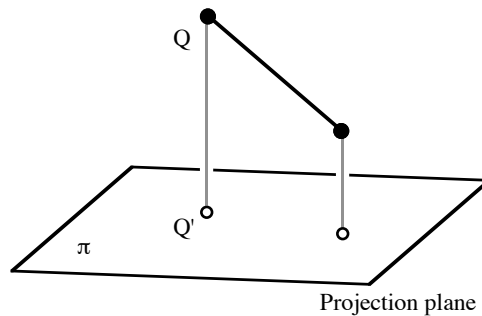


FIG. 4.1: Orthographic projection

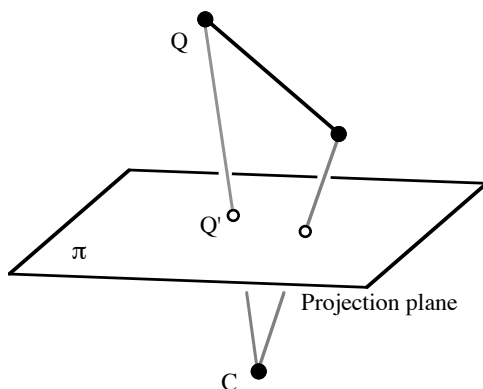


FIG. 4.2: Perspective projection

where the projectors all emanate from a point C called the *center of projection*, as shown in Figure 4.2. It can be shown that perspective projection is *not* an affine map (see Exercise 2 on page 81), but it is a projective map, as discussed in Chapter ??.

The `wireframe` program receives its input from a file that contains the world coordinates of the line segment endpoints, generating as output an orthographic view of the line segments. (The extension of the program to perspective viewing is left as an exercise.) Also input to the program are several viewing parameters:

- A view point called `Eye`, a viewing direction vector called `ViewDir`, and an orientation or “up” vector called `UpVector`. These parameters specify the position of the viewer within the world space: the viewer is positioned at the point `Eye`, looking in a direction `ViewDir`. The projection plane is taken to be the plane through `Eye` perpendicular to `ViewDir`. The role of `UpVector` is to orient the viewer. The view will be such that `UpVector` appears vertical in the final image (see Figure 4.3).
- The width and height of the window on the projection plane. These parameters are stored in variables `WinHsize` and `WinVsize`.
- A viewport specification given as in Section 2.6, stored in the global variables `VPleft`, `VPright`, `VPtop`, and `VPbottom`.

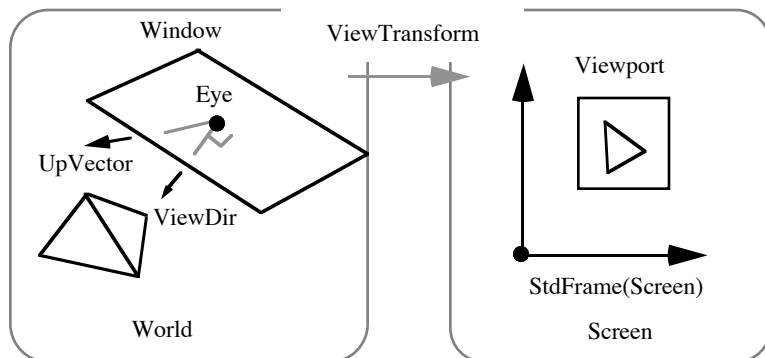


FIG. 4.3: Window and viewport specification.

The general strategy is to create two spaces, **World** and **Screen**. The **World** space is a Euclidean three-space in which the objects to be viewed are placed. The **Screen** space is a two-dimensional space that corresponds to the physical frame buffer, with the visible portion of the **Screen** space defined to be the unit square subtended by the standard frame in **Screen** space (see Figure 4.3). A viewing frame (called **ViewFrame**) is constructed from the viewing parameters and a clipping volume is constructed about the viewing frame. The clipping volume is a rectangular parallelepiped as shown in Figure 4.4.

Line segments are then processed in four steps:

1. *Point Creation*: When a pair of endpoint coordinates are read from the data file, they are immediately converted to Points in the **World** space.
2. *Clipping*: The line segment between the newly created endpoints is clipped to the clipping volume.
3. *Transformation to Screen Space*: The clipped line is (affinely) mapped into the **Screen** space.
4. *Scan Conversion*: The device coordinates of the clipped, projected line segment are extracted and scan converted using a line drawing algorithm such as Bresenham's algorithm.

We shall now examine each of these steps in more detail to demonstrate their implementation using the geometric ADT.

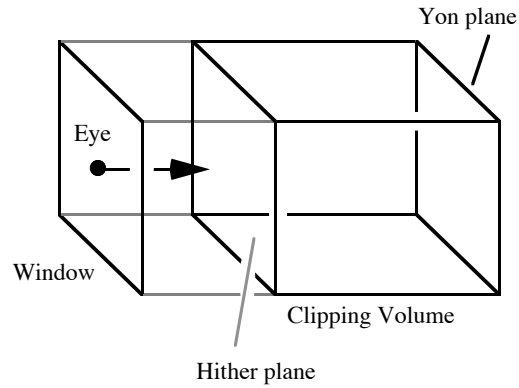


FIG. 4.4: The clipping volume for orthographic projection. The indicated vector is the viewing direction; the front and back clipping planes are at distances `Hither` and `Yon`, respectively.

4.2. Point Creation

The creation of points given coordinates has already been discussed. However, for completeness, step 1 can be implemented by a C procedure `ReadSegment`:

```
typedef struct {
    Point p1, p2;
} Segment;

/*
** Read the world coordinates of two points, and return
** a Segment structure. No check is done for end-of-file.
*/
Segment ReadSegment()
{
    Scalar x1, y1, z1, x2, y2, z2;
    Segment seg;

    scanf("%lf %lf %lf %lf %lf %lf",
          &x1, &y1, &z1, &x2, &y2, &z2);
    seg.p1 = PCreate( WorldFrame, x1, y1, z1);
```

```

    seg.p2 = PCreate( WorldFrame, x2, y2, z2);
    return seg;
}

```

where `WorldFrame` is a synonym for `StdFrame(World)`.

4.3. Clipping

In analogy with Section 3.9, the clipping volume is represented as the intersection of a collection of (six) oriented hyperplanes as shown in Figure 4.4. Each hyperplane is represented as a point and an outward pointing vector (or as described in Section 3.9 and Exercise 11 on page 66, as affine functionals). In our implementation, hyperplanes are represented using the C structure

```

typedef struct {
    Point b;
    Normal n;
} Hyperplane;

```

For now it is sufficient to think about the datatype `Normal` as being a `Vector` used specially to represent perpendicularity (we'll come back to this issue in the next chapter). Since we are thinking of hyperplanes as affine functionals, we define a procedure to apply them to points:

```

/*
** Evaluate the affine functional associated with the
** Hyperplane Pi at the point q.
*/
Scalar EvalHyperplane(Pi, q)
Hyperplane Pi;
Point q;
{
    return NVApply( Pi.n, PDiff(q,Pi.b));
}

```

Don't worry at this point about the meaning of `NVApply`; it is simply a pedantic way of writing `VVDot`.

The clipping volume is then defined to be the intersection of the negative half-spaces defined by the oriented hyperplanes. As in Section 3.9, line segments can be clipped to volumes defined in this

way by successively clipping them against each of the planes.

The clipping of line segments to planes in three dimensions is *identical* to the process described in Section 3.9. A C implementation of the algorithm to clip a line segment to a single hyperplane is shown below. Much of the code is devoted to correctly treating the boundary cases where one or both endpoints are sufficiently close to the boundary of the hyperplane that numerical error could cause problems. The constant EPSILON is a predefined value that is intended to be an upper bound on numerical error.

```

/*
** Clip a line segment against the given Hyperplane.
** Return TRUE if a portion of the segment survives
** the clipping; return FALSE otherwise.
*/
static int ClipLineAgainstHyperplane( P, p1, p2)
Hyperplane P;
Point *p1, *p2;
{
    Point lp1, lp2; /* Local copies of endpoints. */
    Scalar pp1, pp2; /* H(p1) and H(p2) */
    Point intersect; /* Point of intersection. */

    lp1 = *p1;
    lp2 = *p2;

    pp1 = EvalHyperplane( P, lp1);
    pp2 = EvalHyperplane( P, lp2);

    /* If the endpoints are within EPSILON of */
    /* the boundary treat them as if they are */
    /* exactly on the boundary. */
    if (fabs(pp1) < EPSILON) {
        pp1 = 0.0;
    }
    if (fabs(pp2) < EPSILON) {
        pp2 = 0.0;
    }
}

```

```
/* At this point |pp1| and |pp2| are at */
/* least as big as EPSILON or exactly 0.0 */
/* so it is safe to test for equality */
/* with zero. */

if ((pp1 > 0.0) && (pp2 > 0.0)) {
    /* Both points are outside --- trivial reject */
    return FALSE;
}
if ((pp1 <= 0.0) && (pp2 <= 0.0)) {
    /* Both points are inside --- trivial accept */
    return TRUE;
}
/* Check to see if one of the endpoints is */
/* on the boundary. If so, then the line */
/* should either be trivially rejected or */
/* trivially accepted. */
if (pp1 == 0.0) {
    if (pp2 > 0.0) {
        /* lp1 is on the boundary, lp2 is */
        /* outside: trivial reject */
        return FALSE;
    } else {
        /* lp1 is on the boundary, lp2 is */
        /* inside: trivial accept. This case */
        /* should have been caught above, but*/
        /* I'm paranoid. */
        return TRUE;
    }
}

if (pp2 == 0.0) {
    if (pp1 > 0.0) {
        /* lp2 is on the boundary, lp1 is */
        /* outside: trivial reject */
        return FALSE;
    } else {
        /* lp2 is on the boundary, lp1 is */
        /* inside: trivial accept. This case */

```

```

        /* should have been caught above, but */
        /* I'm still paranoid.                */
        return TRUE;
    }
}

/*-----*/
/* The line segment definitely crosses the plane.*/
/* In fact, the Hyperplane cuts the line into   */
/* ratios |pp1| to |pp2|. This is used          */
/* to compute the point of intersection.        */
/*-----*/
intersect = PPr( lp1, lp2, pp1, -1.0*pp2);

/* Figure out which endpoint to throw out */
if (pp1 < 0.0) {
    /* Throw out lp2 */
    *p1 = lp1;
    *p2 = intersect;
} else {
    /* Throw out lp1 */
    *p1 = intersect;
    *p2 = lp2;
}
return TRUE;
}

```

This routine points out that the coordinate-free implementation of clipping has the added benefit that the code has no notion of the dimension of the space in which the line segments live. This means that the procedure above can be used for two-dimensional, three-dimensional, or even n-dimensional line clipping. In two-dimensional clipping, for instance, the oriented hyperplanes are the oriented lines that bound the visible window. Notice too that the hyperplanes are not required to be in any special orientation (as long as the clipping volume is convex). This allows irregular windows and clipping volumes to be used without increasing the complexity of the code.

4.4. Transformation to Screen Space

The transformation that carries points in the `World` space into points in the `Screen` space should be such that the window on the projection plane is carried into the viewport, as indicated by Figure 4.3. This transformation, called `ViewTransform`, is an affine map since orthographic projection is assumed.

As described in Section 3.6, an affine transformation such as `ViewTransform` is completely characterized once its action on a frame is known. The `ViewFrame` is set up for this purpose. The origin of the `ViewFrame` is taken to be the point `Eye`, the x direction vector is taken to be the vector from `Eye` to the right edge of the window, the y direction vector is the vector from `Eye` to the top of the window, and the z direction vector is the unit vector in the direction of `ViewDir`. The convenience of this definition of the `ViewFrame` is that we have a simple characterization for its image under `ViewTransform`. Specifically, let `VPCenter` denote the center point of the viewport, and let `VPx` and `VPy` denote the vectors from `VPCenter` to the right and top edges, respectively, of the viewport. The origin of `ViewFrame` (the `Eye` point) therefore maps to `VPCenter`, its x direction vector maps to `VPx`, its y direction vector maps to `VPy`, and its z direction vector maps to the zero vector in `Screen`. The global variable `ViewTransform` therefore be constructed fairly simply:

```
BuildViewTransform()
{
    Point VPCenter;
    Vector VFx, VFy, VFz, VPx, VPy;

    VFz = VNormalize( ViewDir);
    VFx = SVMult(
        VNormalize(VVCross(VFz,UpVector)),
        WinHsize/2.0);
    VFy = SVMult(
        VNormalize(VVCross( VFx, VFz)),
        WinVsize/2.0);
    ViewFrame = FCreate("View",Eye,VFx,VFy,VFz);

    VPCenter = PCreate( StdFrame(Screen),
```

```

                                (VPlleft+VPrighl)/2.0,
                                (VPltop+VPlbottom)/2.0);
VPx = VCreate( StdFrame(Screen),
              (VPrighl-VPlleft)/2.0, 0.0);
VPy = VCreate( StdFrame(Screen),
              0.0, (VPltop-VPlbottom)/2.0);

ViewTransform = ACreate( ViewFrame,
                        VPlcenter, VPx, VPy,
                        VZero(Screen));
}

```

where $VZero(S)$ returns the zero vector in the space S . Line segments can then be mapped from the World space into the Screen space by:

```

/*
** Send the segment "seg" through the viewing
** transformation.
*/
Segment TransformSegment( seg)
Segment seg;
{
    Segment ScreenSegment;

    ScreenSegment.p1 = PAxform( seg.p1, ViewTransform);
    ScreenSegment.p2 = PAxform( seg.p2, ViewTransform);

    return ScreenSegment;
}

```

4.5. Scan Conversion

Once the segment has been mapped to Screen space, scan conversion can occur after the device coordinates for the endpoints have been determined. Using the geometric ADT, this is most easily accomplished by defining a “device frame” called `DeviceFrame` in the Screen space. This frame is defined such that coordinates relative to the device frame represent device coordinates. As an example, consider a device that has the origin in the upper left hand

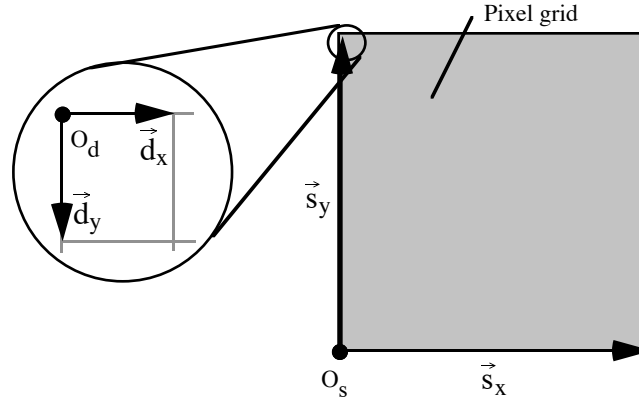


FIG. 4.5: The definition of the device frame $(O_d, \vec{d}_x, \vec{d}_y)^T$ assuming a device whose origin is in the upper left corner, with x increasing to the right and y downward. The vectors \vec{d}_x and \vec{d}_y are defined to be the length and width of a pixel.

corner, with x coordinates increasing to the right and y increasing downward. Suppose too that pixels are addressed from 0 to X_{RES} in the x direction and from 0 to Y_{RES} in the y direction.

Referring to Figure 4.3, the wireframe example establishes the convention that the visible portion of the Screen space is the unit square subtended by `StdFrame(Screen)`. Let the origin of `StdFrame(Screen)` be denoted by O_s , the x-direction vector by \vec{s}_x , and the y-direction vector by \vec{s}_y . If the device frame has origin O_d , x-direction vector \vec{d}_x , and y-direction vector \vec{d}_y , then (see Figure 4.5):

- $O_d = O_s + \vec{s}_y$. This sets the origin of the device frame to the upper left hand corner of the visible region of Screen space.
- $\vec{d}_x = \frac{1}{X_{\text{RES}}} \vec{s}_x$. This says that device x-coordinates increase to the right and range from 0 to X_{RES} . In other words, the vector \vec{d}_x is one pixel in length, as shown in Figure 4.5.
- $\vec{d}_y = -\frac{1}{Y_{\text{max}}} \vec{s}_y$. This says that device y-coordinates increase to downward and range from 0 to Y_{RES} .

Having established `DeviceFrame` as an initialization step, a

segment in `Screen` space can be scan converted by extracting coordinates relative to `DeviceFrame`, then invoking a standard scan-converter such as Bresenham's algorithm:

```
DeviceInitialize()
{
    Point Os,Od;
    Vector Xs, Ys, dx, dy;

    Os = F0rg(StdFrame(Screen));
    Xs = Fv(StdFrame(Screen), 1);
    Ys = Fv(StdFrame(Screen), 2);

    Od = PVAdd( Os, Xs);
    dx = SVMult( 1/XRES, Xs);
    dy = SVMult( -1/YRES, Ys);

    DeviceFrame = FCreate( "Device", Od, dx, dy);
}

/*
** Draw a segment on the device by extracting
** device coordinates then calling Bresenham's
** algorithm.
*/
DeviceDrawLine(seg)
Segment seg;
{
    Scalar x1, y1, x2, y2;

    /* Extract coords relative to DeviceFrame */
    PCoords( DeviceFrame, seg.p1, &x1, &y1);
    PCoords( DeviceFrame, seg.p2, &x2, &y2);

    /* ... and draw the line. */
    Bresenham((int)x1,(int)y1,(int)x2,(int)y2,BLACK);
}
```

Exercises

1. Show that orthographic projection preserves affine combinations, and is therefore an affine map.
2. Show that perspective projection is not an affine map.

Hierarchical Modeling

In this chapter we take a first look at methods for representing structured (*i.e.*, hierarchical) collections of geometric models that contain sufficient information to allow the creation of smooth shaded color images with hidden surfaces removed.

5.1. Simple Polygons

Line segments are insufficient as modeling primitives for creating realistic images since they do not contain enough information to remove hidden surfaces. A two-dimensional modeling primitive must therefore be introduced. The simplest such primitive is a triangle. More generally, one could use quadrilaterals or polygons with an arbitrary number of vertices. More complex objects such as cubes, chairs, telephones, etc., can be tiled using polygonal facets. We begin with the definition of the notion of a “simple” polygon.

A curve is said to be *simple* if it does not intersect itself. The Jordan curve theorem says that a closed simple planar curve divides the plane into two sets, a finite “inside” and an infinite “outside” [7]. A simple polygon P can be defined as the finite set bounded by a closed, simple, planar, piecewise linear curve.

A simple polygon P is typically represented by a sequence of co-planar vertices V_1, \dots, V_n . The bounding edges of the polygon are therefore the line segments $V_i V_{i+1}$, $i = 1, \dots, n$, where indices are to be taken modulo n . Simple polygons can be either *convex* or *concave*. A convex polygon is one where the points of the polygon form a convex set (a set S is convex if its elements are closed under convex combinations).

Since we are interested in using simple polygons as modeling

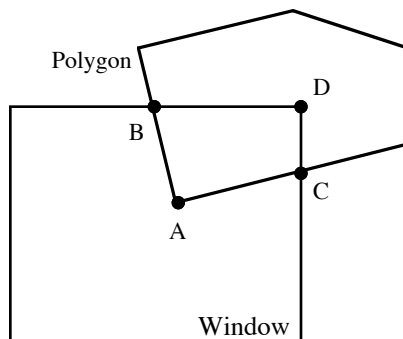


FIG. 5.1: There is more to polygon clipping than repeated line clipping. Repeated line clipping would result in the vertices A , B , and C . However, the vertices of the clipped polygon are A , B , C and D . Thus, further processing would be required to determine that D is the vertex that needs to be added to complete the description of the clipped polygon.

primitives, we must develop algorithms for processing them through the graphics viewing pipeline. That is, we must develop algorithms for clipping, transforming, and scan-converting simple polygons. The remainder of this section is devoted to these tasks.

5.1.1. Clipping The problem we consider first may be stated as:

Given: A convex simple polygon P with vertices V_1, \dots, V_n contained in a two-dimensional space \mathcal{A} , and a convex window W (the clipping region) represented as the intersection of oriented hyperplanes H_1, \dots, H_k .

Find: The vertices $V'_1, \dots, V'_{n'}$ of the polygon $P' = P \cap W$.

One might be tempted to solve this problem by using the Sutherland-Hodgman line clipping algorithm to clip the edge V_1V_2 to W , then V_2V_3 to W , and so on, to produce a sequence of vertices $V'_1, \dots, V'_{n'}$ that would be conjectured to form the vertices of $P' = P \cap W$. The failure of this approach is indicated in Figure 5.1.

The algorithm we present here is due to Sutherland and Hodgman [17]. It proceeds as in the Sutherland-Hodgman line clipping



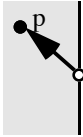

Case	Geometry	Action
1) $p \in H, s \in H$		Output(p)
2) $p \notin H, s \notin H$		None
3) $p \in H, s \notin H$		Output(I) Output(p)
4) $p \notin H, s \in H$		Output(I)

FIG. 5.2: A summary of the Sutherland-Hodgman polygon clipping algorithm.

algorithm in that it successively clips the entire polygon to each of the oriented hyperplanes defining the clipping region. The problem of clipping a polygon to a convex clipping region is thereby reduced to the following problem:

Given: A polygon P with vertices V_1, \dots, V_n to an oriented hyperplane H .

Find: The vertices of Q_1, \dots, Q_m of $P \cap H$.

The method is to march around the vertices of P keeping track of a previous vertex s and a current vertex p . On each iteration of the algorithm, there are four cases to consider based on the containment of s and p in H . The action to take in each of the cases is summarized in Figure 5.2. The action Output(Q) adds point Q to the end of the sequence of vertices Q_1, \dots, Q_m being built up for $P \cap H$.

This same algorithm extends readily for clipping polygons in an affine space \mathcal{A} of arbitrary dimension to a clipping region represented as the intersection of oriented hyperplanes.

The proper clipping of concave polygons is somewhat more difficult. Such an algorithm has been developed by Weiler and Atherton [22].

5.1.2. Transforming Through Affine Maps The image of a simple polygon P with vertices V_1, \dots, V_n under an affine transformation T is again a simple polygon P' with vertices $T(V_1), \dots, T(V_n)$. A polygon can therefore be mapped through an affine transformation simply by mapping each of its vertices.

5.1.3. Scan-Conversion In this section we examine two algorithms for scan-converting polygons. The first is a general algorithm appropriate for scan-converting a simple polygon with an arbitrary number of edges. The second algorithm is a simplified version optimized for scan-converting triangles. In the following sections we assume that the vertices are given in device coordinates.

A Sweep Line Algorithm The basic idea in this algorithm is to process the polygon a scan-line at a time, considering the scan-lines in, say, bottom to top order. For each scan-line, all intersections I_0, \dots, I_{k-1} between the scan-line and the edges of the polygon are found. The intersections are maintained in a list sorted by increasing x coordinate. For each pair of intersection points I_{2i}, I_{2i+1} , all pixels between these points (called a *span*) are illuminated as indicated in Figure 5.3.

One subtlety with the algorithm is that intersections between the polygon and the scan-line must be counted carefully since there is an implicit assumption that the number of intersections I_0, \dots, I_{k-1} is even. Consider for instance scan-lines S_2 and S_3 in Figure 5.3. Vertex V_3 is counted twice on scan-line S_2 whereas the vertex V_1 is counted only once in scan-line S_3 . The general rule is to count a vertex twice if it is a *local min* or a *local max*; it is counted once otherwise. A vertex V_i is considered a local min if both V_{i-1} and V_{i+1} lie on scan-lines above V_i ; local max vertices are defined similarly.¹

The real cleverness in the algorithm comes from the way the

¹You should think about how to handle horizontal edges.

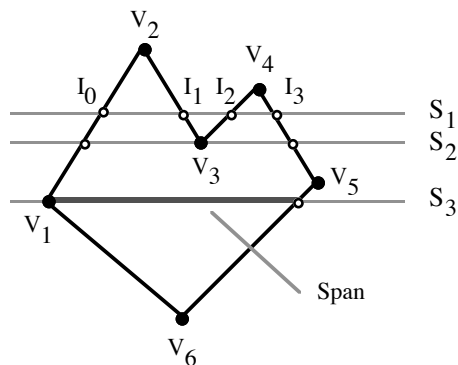


FIG. 5.3: The sweep-line algorithm identifies entire spans of pixels at a time.

spans are computed incrementally as the current scan-line “sweeps” up the image. Two data structures are used to speed the process. The first data structure, called the *active edge list* or AEL, represents the intersection points I_0, \dots, I_{k-1} . It contains a set of edge records, one for each edge that intersects the current scan-line, sorted by the x coordinate of the intersection points. The edge records are defined as

```

EdgeRecord = record
     $y_{exit}$  : integer;
     $x, x_{inc}$  : real;
end;
```

For an edge record e , the field $e.x$ contains the x coordinate of the intersection of the edge with the current scan-line (the scan-line serves to implicitly specify the y coordinate of the intersection). Successive pairs of edge records on the AEL define the spans appropriate for the current scan-line. After the current scan-line has been processed, the AEL is updated by:

1. Discarding edges that become inactive; that is, edges that do not intersect the next scan-line. The field y_{exit} is used for this purpose. It is set to indicate the y coordinate of the last scan-line for which the edge is active. Thus, all edges e such that

```

BuildBucketTable( $P$ )
begin
  foreach edge  $(x_i, y_i)(x_{i+1}, y_{i+1}) \in P$  do
     $e :=$  new EdgeRecord;
    if  $(y_i > y_{i+1})$  then
      { Make sure  $(x_i, y_i)$  is lowest endpoint. }
      Swap  $(x_i, y_i)$  and  $(x_{i+1}, y_{i+1})$ 
    endif;
     $e.x := x_1$ ;
     $e.x_{inc} := (x_{i+1} - x_i)/(y_{i+1} - y_i)$ ;
    Insert  $e$  into  $B_{y_i}$  in sorted order.
  endforeach;
end;

```

FIG. 5.4: Creation of the y-bucket table.

$e.y_{exit}$ is less than the y coordinate of the next scan-line are removed from the AEL.

2. Updating the x coordinates of the intersections by adding x_{inc} to x . The values $e.x$ and $e.x_{inc}$ for an edge e are initialized as shown in Figure 5.4.
3. Adding newly active edges. A second data structure, called the *y-bucket table*, is used to quickly identify the edges that become active on a given scan-line. The y-bucket table is an open hash table containing one bucket B_i per scan-line. It is initialized by building an edge record per edge of the polygon as shown in Figure 5.4. An edge e is placed in bucket B_i if i is the y coordinate of the lowest endpoint of e . Edges within each bucket are sorted by increasing x field as a primary sort key and by increasing x_{inc} value as a secondary sort key. An example of the y-bucket table is shown in Figure 5.5.

Scan-conversion of Triangles The scan-conversion of triangles is much simpler than the scan-conversion of an arbitrary simple polygon. Consider the scanning of a triangle with vertices $V_1 = (x_1, y_1)$, $V_2 = (x_2, y_2)$, $V_3 = (x_3, y_3)$. Without loss of generality we

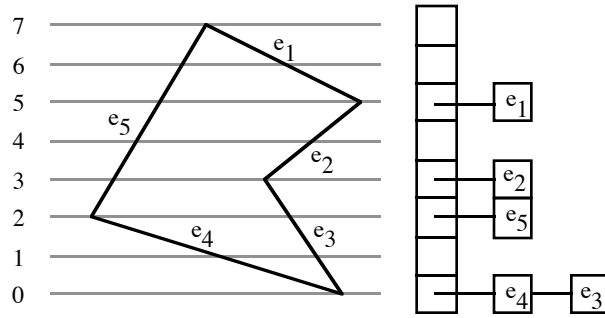


FIG. 5.5: An example of the y-bucket table

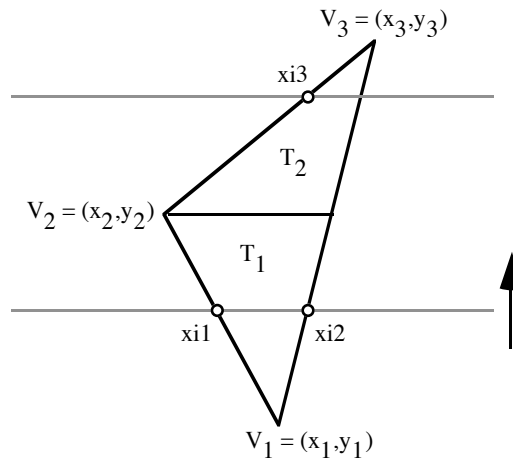


FIG. 5.6: Triangle scan-conversion

can assume that $y_1 \leq y_2 \leq y_3$. We divide the triangle along the scan-line through V_2 into two subtriangles T_1 and T_2 as shown in Figure 5.6. The original triangle can be scanned by first scanning out T_1 starting at V_1 and moving upward, followed by scanning of T_2 . The two intersections between the current scan-line and the triangle are maintained and incrementally updated. This process is summarized by the following pseudo-code:

```

ScanTriangle(x1,y1,x2,y2,x3,y3 : integer)
{ Scan the triangle assuming  $y_1 \leq y_2 \leq y_3$  }
begin
  y, i : integer;
  inc1, inc2, inc3, xi1, xi2, xi3 : real;

  if (y3 = y1) then
    { The triangle is degenerate; scan the edges. }
    ScanSpan( min(x1,x2,x3), max(x1,x2,x3), y1);
    return;
  endif;

  xi2 := x1;
  inc2 := (x3 - x1)/(y3 - y1);

  if (y1  $\neq$  y2) then
    { Scan  $T_1$  }
    xi1 := x1;
    inc1 := (x2 - x1)/(y2 - y1);

    { Scan up to, but not including, scan-line  $y_2$ . }
    { Scan-line  $y_2$  will be scanned in the conversion }
    { of subtriangle  $T_2$  }
    for y := y1 to y2-1 do
      ScanSpan( xi1, xi2, y);
      xi1 := xi1 + inc1;
      xi2 := xi2 + inc2;
    endfor;
  endif;

  if (y2  $\neq$  y3) then

```

```

    { Scan  $T_2$  }
    inc3 := (x3 - x2)/(y3 - y2);
    xi3 := x2;
    for y := y2 to y3 do
        ScanSpan( xi2, xi3, y);
        xi2 := xi2 + inc2;
        xi3 := xi3 + inc3;
    endfor;
else
    {  $T_2$  was empty; scan out scan-line  $y_2$  }
    ScanSpan( xi1, xi2, y2);
endif;
end;

```

The routine ScanSpan() referred to above is responsible for scanning the pixels on scan-line y . A strategy for avoiding gaps between triangles that are supposed to abut is to have ScanSpan() scan the pixels whose x coordinates are from the floor of the left endpoint of the span to the ceiling of the right endpoint. This tends to enlarge the triangle slightly, but it helps to fill gaps that can open between triangles if both left and right endpoint coordinates are rounded.

5.2. Object Hierarchies

See transparencies

5.2.1. Transformation Stacks

Exercises

1. Show that the Sutherland-Hodgman polygon clipping algorithm can fail if the polygon P to be clipped is concave.
2. Modify the triangle scan-conversion algorithm so that pixels covered by abutting triangles are painted once and only once.

Hidden Surface Algorithms

In this chapter we consider the problem of accounting for total or partial occlusion of one object by another. That is, we consider the *hidden surface problem*. Hidden surface algorithms can be classified into two broad categories: image space or object space algorithms. Image space algorithms resolve the hidden surface problem at the resolution of the final image; that is, only at displayable pixels. Object space algorithms, on the other hand, produce a resolution independent solution.

6.1. Back Face Culling

Before discussing general hidden surface algorithms that determine complete visibility information, there is a test that can often be used to quickly identify invisible polygons. The test, known as *back face culling*, is based on the assumptions that polygons are used to tile closed opaque objects, and that none of the polygons of an object are clipped by the clipping volume. In other words, the polygons present after clipping are assumed to enclose well-defined volumes. Back face culling is therefore inappropriate for an object such as the one shown in Figure 6.2 where polygons cover only five of the six sides of a cube.

The test also requires that polygons have associated outward normals that point outward from the enclosed volume (see Figure 6.1). It is common to infer the outward normal by imposing an ordering on polygon vertices. A common convention (for convex polygons) is the “right hand rule”: if the polygon is defined by vertices V_1, \dots, V_n , then for any $i = 1, \dots, n$ the outward normal is defined to point along $(V_{i+1} - V_i) \times (V_{i+2} - V_{i+1})$.

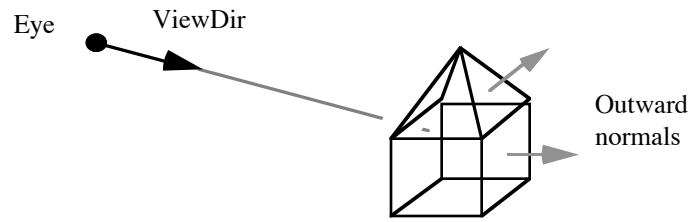


FIG. 6.1: Back face culling.

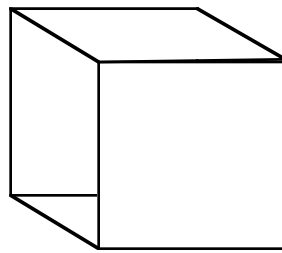


FIG. 6.2: An object for which back face culling is inappropriate since the polygons do not enclose a volume.

With these conventions, the outward normal of a polygon can be used to quickly determine a sufficient condition for invisibility of the polygon. Intuitively, if the outward normal points away from the viewer, then the polygon will be totally occluded by one or more other polygons, and hence it can be culled from further processing. More specifically, a polygon can be culled in this way if $(P - \text{Eye}) \cdot \vec{n} > 0$, where P is any point in the plane of the polygon, Eye is the viewpoint, and \vec{n} is the outward normal. This test is only a sufficient condition for invisibility. The full necessary and sufficient conditions require a general purpose hidden surface algorithm.

6.2. Three-Dimensional Screen Space

Most hidden surface algorithms in use today are of the image space variety, resolving the hidden surface problem during scan-conversion. To use such an algorithm, we must extend the graphics pipeline since the current pipeline projects from the three-dimensional world space down to the two-dimensional screen space, thus losing critical information about relative depth. If an image space algorithm is to be used, the screen space must be generalized from two dimensions to three dimensions, and the viewing transformation from world space to screen space must be set up so the depth ordering of objects (as seen from the viewpoint) is preserved.

To simplify the following discussion, we assume for the time being that the scene is to be viewed in orthographic rather than perspective projection. The extension to perspective viewing will be covered in Section 7.2.

The geometric situation is indicated in Figure 6.3. By setting up the viewing transformation so that it maps the clipping volume to the indicated parallelepiped in the screen space (called the *view box*), when objects are transformed through the map their depth orderings are preserved. Since the screen space is now three-dimensional instead of two-dimensional, the definition of the device frame must be extended to include a third vector. It is convenient to introduce the third device frame vector \vec{d}_z as shown in Figure 6.3. In Figure 6.3, the standard frame for the screen space is $(\vec{s}_x, \vec{s}_y, \vec{s}_z, \mathcal{O}_s)^T$, and the device frame is denoted by $(\vec{d}_x, \vec{d}_y, \vec{d}_z, \mathcal{O}_d)^T$. Assuming the same type of device as in Section 4.5, we have the relations

$$\mathcal{O}_d = \mathcal{O}_s + \vec{s}_y$$

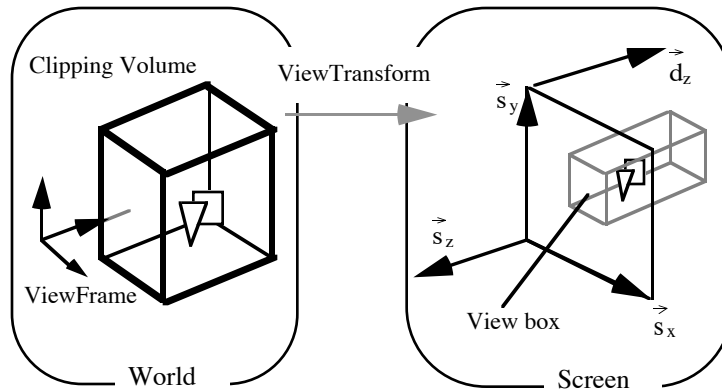


FIG. 6.3: The viewing transformation is such that the clipping volume is mapped to the view box so as to preserve the depth ordering of points along projectors.

$$\begin{aligned}\vec{d}_x &= \frac{1}{X_{\text{RES}}}\vec{s}_x \\ \vec{d}_y &= -\frac{1}{Y_{\text{RES}}}\vec{s}_y \\ \vec{d}_z &= -\vec{s}_z.\end{aligned}$$

These frames have been set up so that if we transform a point into the screen space and extract coordinates (x, y, z) relative to the device frame, then (x, y) determines the pixel to illuminate, and z reflects the relative depth of the point within the scene such that increasing z corresponds to increasing depth. Thus, if two points P_1 and P_2 have device frame coordinates (x, y, z_1) and (x, y, z_2) , respectively, then P_1 occludes P_2 if and only if $z_1 < z_2$.

6.3. The Depth Buffer Algorithm

The first hidden surface algorithm we shall consider is a very simple one called the *depth buffer* algorithm. The algorithm, originated by Ed Catmull in 1974 [4], is also called the *z-buffer* algorithm.

The basic idea behind the algorithm is to maintain an array of depth values during the scan-conversion of the primitives in the scene. The depth buffer allocates one entry for each pixel (x, y) , denoted $\text{depth}[x, y]$, that contains the depth (*i.e.*, the “z-value”) of

the nearest object covering pixel (x, y) that has been processed thus far. The depth buffer is initialized by setting each entry to a value larger than any realizable depth. (The arrangement described in Section 6.2 guarantees that the largest possible depth is 1.) During the scan-conversion of a primitive object O , if O is determined to cover pixel (x, y) , then O is visible at (x, y) if and only if the depth of O at (x, y) is less than the current value of $\text{depth}[x, y]$. If O is visible at (x, y) , then the pixel is shaded and the depth buffer is updated to record the fact that O is now the closest object visible at (x, y) . This process is summarized by the code fragment:

```
if depth of  $O$  at  $(x, y) < \text{depth}[x, y]$  then  
    fb_writePixel( $x, y$ , Shade( $O, x, y$ ));  
    depth[ $x, y$ ] := depth of  $O$  at  $(x, y)$ ;  
endif
```

The Shade() routine implements one of the shading algorithms described in Chapter 8.

The depth buffer algorithm is very easy to implement, especially if the scene is tessellated into triangles prior to scan-conversion. To this end, the triangle scan-conversion algorithm of Section 5.1.3 can be extended to incrementally compute the depth of the triangle during scan-conversion (see Exercises 4 and 5 on page 100). In fact, the depth buffered scan-conversion of triangles is simple enough that a number of graphics workstations currently implement it in hardware. Another advantage of the depth buffer algorithm is that it is an *on line* algorithm, meaning that it can fully process display primitives one at a time. Most other hidden surface algorithms require that all primitives be available before any occlusion determinations can be made.

A major disadvantage of the depth buffer algorithm, at least as described above, is that the images it creates suffer from discretization effects. For instance, if two triangle interpenetrate, the line of intersection will appear as a fairly ragged edge.

6.4. Warnock's Algorithm

Warnock's algorithm [20] is an early example of a divide and conquer method that resolves hidden surfaces for the entire viewport by determining if the image to be rendered is "sufficiently simple" to

allow hidden surfaces to be determined with simple tests. If the image within the viewport is too complex, the viewport is split horizontally and vertically into four subviewports, the polygons are split and distributed amongst the subviewports, and the algorithm is called recursively.

Recursion stops when either a subviewport is the size of a pixel or the image within the subviewport is “simple”. An image within a subviewport is considered to be simple if at most one polygon is visible within the subviewport. This can occur either if the subviewport contains at most one polygon or if the subviewport is covered by a single polygon. If the subviewport is the size of a pixel and the image is not simple, the pixel is painted based on the color of the nearest polygon covering the center of the pixel.

6.5. A Sweep Line Algorithm

In the late 60's and early 70's a number of algorithms were developed that build up an image a scan-line at a time [1, 2, 21, 24]. The basic idea behind these algorithms is to use the sweep line algorithm for polygon scan-conversion discussed in Section 5.1.3. In the following it is assumed that the polygons do not interpenetrate. (If polygons P_1 and P_2 do interpenetrate, P_1 can be split along the plane of P_2 into two subpolygons using the Sutherland-Hodgman polygon clipping algorithm.)

The y-bucket table and active edges lists will again be used as in Section 5.1.3, with the exception that all edges of all polygons are dealt with simultaneously. Thus, the y-bucket table is initialized by inserting an edge record for each edge in the scene. The edge records are essentially as before, with an added field that points back to a polygon record indicating which polygon the edge belongs to. Polygon records store data such as the plane equation for the polygon, color attributes, and so forth.

The active edge list (AEL) is again used to maintain the set of edges intersected by the current scan-line, sorted in left to right order. The critical observation in the algorithm is that visibility changes can occur only at points of intersection between the scan-line and elements of the AEL (this is only true because of the assumption of non-interpenetrating polygons). This observation means that the hidden surface problem can be resolved for entire spans at a time. Referring to Figure 6.4, span s_1 will be painted the color of polygon

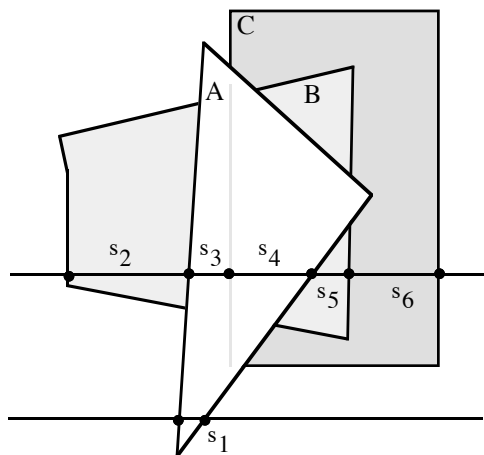


FIG. 6.4: Entire spans are resolved by the sweep line algorithm.

A , span s_2 with the color of B , spans s_3 and s_4 with the color of A , and so on.

To determine which polygon is visible within each span, a list of polygons beneath the current span is maintained. This list, called *Inside*, is sorted by increasing depth. When a new span is processed, the polygon P corresponding to the left edge of the span is either added to or deleted from the *Inside* list: P is deleted if it is already present, otherwise it is added to the list in depth sorted order. The polygon at the front of *Inside* is the one used to determine the color of each span.

Here is a brief critique of the sweep line algorithm:

- (+) works well when spans are large.
- (+) each pixel gets painted once.
- (+) amenable to antialiasing.
- (-) fairly difficult to program and debug.
- (-) not on-line.

Exercises

1. Show that if after clipping polygons tile closed opaque objects, then a polygon is occluded if $(P - \text{Eye}) \cdot \vec{n} > 0$, where P is a point on the polygon, \vec{n} is the outward pointing normal, and Eye is the view point.
2. The right hand rule given in Section 6.1 is appropriate for convex polygons only. Give a more general rule that can be applied to concave as well as convex polygons.
3. For what class of scenes is back face culling both necessary and sufficient for polygon invisibility?
4. In the depth buffered scan conversion of triangles a “z increment” must be computed such that if the triangle has depth z at pixel (x, y) , then it has depth $z + \text{zinc}$ at pixel $(x + 1, y)$. (x, y and z denote three-dimensional device coordinates.) If the triangle has vertices V_1, V_2 , and V_3 in screen space, show that

$$\text{zinc} = -\frac{\vec{d}_x \cdot \vec{n}}{\vec{d}_z \cdot \vec{n}}$$

where \vec{d}_x and \vec{d}_z denote device frame basis vectors as in Figure 6.3, and where \vec{n} is a vector perpendicular to the plane of the triangle.

5. Extend the triangle scan-conversion algorithm of Section 5.1.3 to perform depth buffered scan conversion.

Coordinate-Free Geometric Programming II

In Chapter 3 a collection of basic geometric entities were introduced (points, vectors, etc.). While these objects are sufficient for many applications, two new objects must be added to fully support the generation of smooth shaded images of scenes viewed in perspective. These objects are projective transformations and normal vectors.

7.1. Projective Transformations

In Section 4.1 it was mentioned that perspective projections are not affine transformations. To model perspective, we must generalize to the projective transformations. Affine transformations were shown to carry lines to lines and to preserve ratios of distances along lines. These two properties can in fact be used as the definition of affine transformations. That is, if we define $\text{Ratio}()$ by

$$\text{Ratio}(P, Q, R) := \bar{QR} : \bar{PQ} = \frac{\bar{QR}}{\bar{PQ}}$$

for collinear points PQR , where \bar{AB} denotes the length of a line segment AB , then a map T is affine if for all collinear triples of points Q_0, Q, Q_1 , their images Q'_0, Q', Q'_1 are collinear and

$$\text{Ratio}(Q_0, Q, Q_1) = \text{Ratio}(Q'_0, Q', Q'_1). \quad (7.1)$$

Projective maps do not preserve ratios, but they do preserve *cross ratios*. The cross ratio of four collinear points Q_0RQQ_1 can be

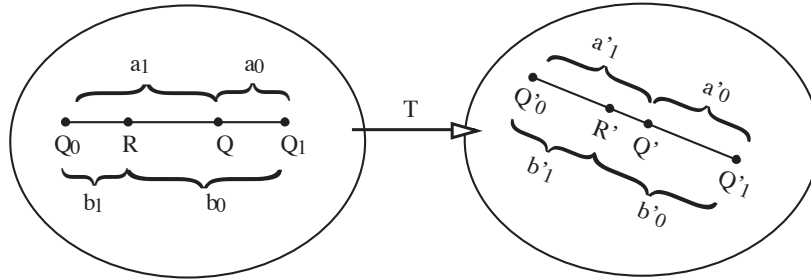


FIG. 7.1: The cross ratio.

defined as¹

$$CR(Q_0, R, Q, Q_1) := \frac{\text{Ratio}(Q_0, Q, Q_1)}{\text{Ratio}(Q_0, R, Q_1)}.$$

Projective maps can therefore be defined as follows:

DEFINITION 7.1.1. *A map $T : \mathcal{A} \rightarrow \mathcal{B}$ between affine spaces is said to be projective if lines map to lines in such a way that for all collinear quartuples Q_0RQQ_1 the following holds:*

$$CR(Q_0, R, Q, Q_1) = CR(Q'_0, R', Q', Q'_1) \quad (7.2)$$

where the primed points are the images of the unprimed points under T , as indicated in Figure 7.1.

Strictly speaking, Definition 7.1.1 as stated is not completely precise. The difficulty is that the transformation may be undefined for a small set of lines (see Example 6). One way to make the definition precise is to extend the affine domain and range spaces to their *projective completions* (see Chapter ??). Roughly speaking, a projective completion is obtained by adding “points at infinity” in a way that avoids special cases that arise in affine spaces. Fortunately, for purposes of perspective viewing the anomalies in Definition 7.1.1 will not be encountered.

EXAMPLE 6.

¹There are several different definitions that can be adopted for cross ratios, but they are all equivalent in the sense that one is preserved if and only if the others are preserved.

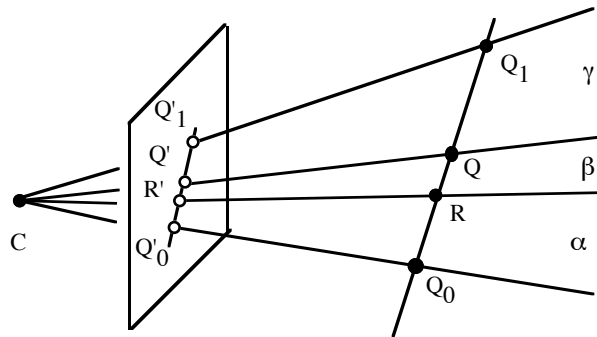


FIG. 7.2: Central projection maps lines to lines and preserves cross ratios, hence it is a projective map.

In Section 4.1 it was claimed that perspective projection, also known as *central projection*, was a projective map. To verify the claim, we must demonstrate that lines map to lines and that cross ratios are preserved under central projection. The following development is due to Farin [9].

Referring to Figure 7.2, it is clear that lines not containing C map to lines under perspective projection. In particular, a line Q_0Q_1 maps to the line of intersection between the projection plane and the plane containing C , Q_0 , and Q_1 . A line through C maps to the point of intersection between the line and the projection plane. The one exception to this is for lines through C and parallel to the projection plane; in fact, the map is not defined for these lines since the line and the projection plane do not intersect. As mentioned above, for now we shall simply ignore this small set of anomalous lines.

For those lines that are mapped to lines under central projection, we must show that cross ratios are preserved. We first notice that

$$\text{Ratio}(Q'_0, Q', Q'_1) = \frac{\text{Area}(Q'_0, Q', C)}{\text{Area}(Q', Q'_1, C)}$$

where $\text{Area}(A, B, C)$ denotes the area of the triangle whose vertices are A, B, C . If $\alpha = \angle Q_0CR$, $\beta = \angle RCQ$, and $\gamma = \angle QCQ_1$, then the law of sines can be used to show that

$$\text{Area}(Q_0, Q, C) = \frac{\overline{Q_0C} \overline{QC} \sin(\alpha + \beta)}{2}$$

implying that $CR(Q_0, R, Q, Q_1)$ can be written as

$$\begin{aligned} CR(Q_0, R, Q, Q_1) &= \frac{\frac{Q_0\bar{C} \ Q_1\bar{C} \ \sin(\alpha + \beta)}{Q\bar{C} \ Q_1\bar{C} \ \sin(\gamma)}}{\frac{Q_0\bar{C} \ RC \ \sin(\alpha)}{RC \ Q_1\bar{C} \ \sin(\beta + \gamma)}} \\ &= \frac{\sin(\alpha + \beta) \sin(\beta + \gamma)}{\sin(\gamma) \sin(\alpha)} \end{aligned} \quad (7.3)$$

Equation 7.3 shows that $CR(Q_0, R, Q, Q_1)$ depends only on angles between projectors, and not on distances from the points to the center of projection. Since the projectors for Q_0, R, Q, Q_1 are shared by Q'_0, R', Q', Q'_1 , we conclude that

$$CR(Q_0, R, Q, Q_1) = CR(Q'_0, R', Q', Q'_1),$$

thus completing the proof that central projection is a projective map.

Since the cross ratio is a generalization of the simple ratio, every affine transformation is also a projective transformation. It is possible to show that the composition of two projective maps yields a projective map (see Exercise 1 on page 118); hence, the composition of an affine map with a projective map also yields a projective map.

Referring again to Figure 7.1, imagine that the points $Q_0, R,$ and Q_1 are fixed and that Q is variable. We would like to obtain an expression for $Q' = T(Q)$ in terms of the fixed points and their images $Q'_0, R',$ and Q'_1 . To do this, let $a_0 : a_1 = \text{Ratio}(Q_0, Q, Q_1)$ and let $b_0 : b_1 = \text{Ratio}(Q_0, R, Q_1)$; similarly let $a'_0 : a'_1 = \text{Ratio}(Q'_0, Q', Q'_1)$ and let $b'_0 : b'_1 = \text{Ratio}(Q'_0, R', Q'_1)$, as shown in Figure 7.1. Since T is projective, Equation 7.2 holds, implying that

$$\frac{a_0 : a_1}{b_0 : b_1} = \frac{a'_0 : a'_1}{b'_0 : b'_1},$$

or, equivalently, that

$$a'_0 : a'_1 = a_0 \lambda_0 : a_1 \lambda_1$$

where $\lambda_0 = b'_0/b_0$, and $\lambda_1 = b'_1/b_1$. Thus, Q' can be expressed as

$$Q' = \frac{a'_0 Q'_0 + a'_1 Q'_1}{a'_0 + a'_1}$$

$$\begin{aligned}
 &= \frac{(a'_0 : a'_1)Q'_0 + Q'_1}{(a'_0 : a'_1) + 1} \\
 &= \frac{(a_0\lambda_0 : a_1\lambda_1)Q'_0 + Q'_1}{(a_0\lambda_0 : a_1\lambda_1) + 1} \\
 &= \frac{a_0\lambda_0Q'_0 + a_1\lambda_1Q'_1}{a_0\lambda_0 + a_1\lambda_1}. \tag{7.4}
 \end{aligned}$$

Before going further, it is convenient to introduce some simplifying notation. Let c_0, \dots, c_n be arbitrary scalars and let Q_0, \dots, Q_n be arbitrary points. We define the bracket notation $[\cdot]$ by

$$[c_0Q_0 + \dots + c_nQ_n] := \frac{c_0Q_0 + \dots + c_nQ_n}{c_0 + \dots + c_n}.$$

Using this definition of $[\cdot]$, expressions such as $[a\vec{v} + b\vec{w} + cO]$, where O is a point can be shown to be

$$[a\vec{v} + b\vec{w} + cO] = \frac{a}{c}\vec{v} + \frac{b}{c}\vec{w} + O.$$

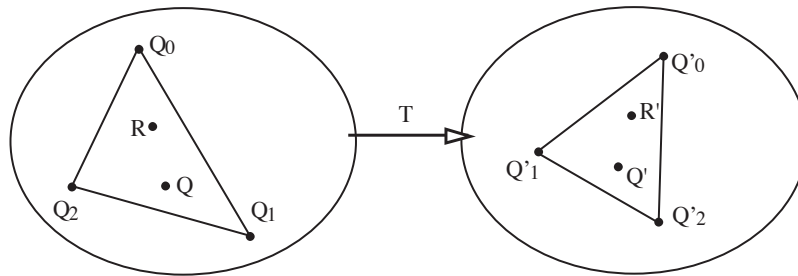
Equation 7.4 can now be written more simply as

$$Q' = [a_0\lambda_0Q'_0 + a_1\lambda_1Q'_1]. \tag{7.5}$$

Equation 7.5 states that if Q'_0, R' , and Q'_1 are known, then $T(Q)$ can be computed for any other point Q on the line Q_0Q_1 . In other words, T as a map on the line Q_0Q_1 is completely determined once the image of three distinct points is known. Contrast this to the situation for affine maps: An affine map on a line is completely determined once the image of two points is known.

Figure 7.3 illustrates the case of a projective map on a plane. Let (a_0, a_1, a_2) and (b_0, b_1, b_2) be the barycentric coordinates of Q and R respectively relative to the triangle $Q_0Q_1Q_2$, and let (b'_0, b'_1, b'_2) be any numbers such that $R' = [b'_0Q'_0 + b'_1Q'_1 + b'_2Q'_2]$. In the case that Q'_0, Q'_1 , and Q'_2 are affinely independent, these numbers are unique up to a scale factor. Using a process similar to the one leading to Equation 7.5, the image of Q under T is given by

$$\begin{aligned}
 T(Q) &= T([a_0Q_0 + a_1Q_1 + a_2Q_2]) \\
 &= [a_0\lambda_0T(Q_0) + a_1\lambda_1T(Q_1) + a_2\lambda_2T(Q_2)] \\
 &= [a_0\lambda_0Q'_0 + a_1\lambda_1Q'_1 + a_2\lambda_2Q'_2] \tag{7.6}
 \end{aligned}$$

FIG. 7.3: A projective map T on a plane.

where $\lambda_i = b'_i/b_i, i = 0, 1, 2$. Thus, for two dimensions, a projective map is completely determined once the action on a triangle plus one other point (R) is known. For the general case of n dimensions, a straightforward generalization of Equation 7.6 holds, implying that a projective map from an n dimensional affine space is completely determined once its action on $n + 2$ points is known (an n -simplex plus one other point).

REMARK: There are two “hidden” assumptions in the above discussion. First, as mentioned above, the $n + 2$ points used to characterize a projective map must be in *general position* (see Exercise 2). A collection of $n + 2$ points is said to be in general position if whenever one of the points is deleted, the remaining $n + 1$ points are affinely independent. Thus, four points in two-dimensions are in general position if no three are collinear; similarly, five points in three-dimensions are in general position if no four are coplanar. The second implicit assumption is that the points Q'_0, \dots, Q'_n form an n -simplex. This assumption crept in when we introduced the barycentric coordinates (b'_0, \dots, b'_n) of R' . The requirement that Q'_0, \dots, Q'_n form an n -simplex is impossible to satisfy if T is not invertible. For instance, in the three-dimensional case, if T collapses the whole 3-space into a plane (as in Example 6), then a 3-simplex in the plane cannot occur since all sets of 4 points in a plane are affinely dependent. Fortunately, Equation 7.6 still holds as long as R' is in the affine span of Q'_0, \dots, Q'_n . \square

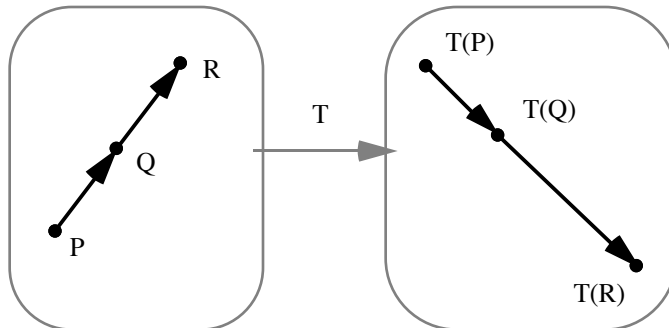


FIG. 7.4: The action of a projective map on a line segment.

Even though projective transformations carry points to points,² lines to lines, and more generally, hyperplanes to hyperplanes, they do not preserve the structure of affine spaces (they do, however, preserve the structure of *projective spaces*). In particular, they do not map vectors to vectors. In fact, it is not possible to extend the domain of a projective transformation to include vectors. In an attempt to do so, we might be tempted to offer a definition similar to the one used for affine spaces. That is, suppose T is a projective transformation and suppose P and Q are points such that $\vec{v} = P - Q$. If we define the action of T on \vec{v} by $T(\vec{v}) = T(P - Q) = T(P) - T(Q)$ we run into a fundamental difficulty. To be well-formed, the definition of $T(\vec{v})$ should be such that it does not matter which pair of points are used as long as their difference is \vec{v} . Unfortunately, the value of $T(\vec{v})$ does depend on the choice of points. An explicit example of this difficulty is shown in Figure 7.4. The point Q is the midpoint of P, R , implying that $Q - P = R - Q$. However, since T does not map Q to the midpoint of $T(P), T(R)$, we find that $T(Q) - T(P) \neq T(R) - T(Q)$. Thus, if we used Q and P to compute $T(\vec{v})$, we would arrive at a different result than if we had used Q and R .

²Actually, as mentioned earlier, some points in the domain can be mapped to points at infinity in the range. These points therefore do not have images in the affine range space.

7.1.1. The ProjectiveMap Data Type The fact that projective transformations cannot map vectors must be reflected in the geometric algebra and the ADT. The geometric algebra can deal with the situation simply by leaving it undefined; the ADT can handle the problem by signaling a type-clash if a request is made to map a vector through a projective transformation.

The ADT can be augmented to support projective maps by adding a ProjectiveMap data type along with the following procedures:

- ProjectiveMap \leftarrow PMCreateP($P_0, \dots, P_{n+1}, P'_0, \dots, P'_{n+1} : \text{Point}$)
Return the projective map that carries P_i to P'_i , $i = 0, \dots, n+1$. The points P_0, \dots, P_{n+1} must reside in a common n -space, and the points P'_0, \dots, P'_{n+1} must reside in a common m -space. A further restriction is that the affine span of P'_0, \dots, P'_m is the entire range of the transformation.
- ProjectiveMap \leftarrow PMPMCompose($F, G : \text{AffineMap or ProjectiveMap}$)
Return the projective map $G \circ F$. An error is signaled if the domain of G does not match the range of F .
- Point PPMxform($P : \text{Point}, \text{PM} : \text{ProjectiveMap}$)
Return the point P transformed by projective map PM .

7.1.2. *Matrix Representations of Projective Maps For an affine map T , we were able to find a matrix that would transform coordinates of a point P into coordinates for $T(P)$. We would now like to consider the situation when T is projective. It turns out to be slightly easier to construct a matrix that transforms *barycentric coordinates* for P into (frame) coordinates for $T(P)$.

For notational simplicity, we will again do only the two-dimensional case; the general case follows immediately. Let $T : \mathcal{A} \rightarrow \mathcal{B}$ be a projective map, and let $Q_0, Q_1, Q_2, R \in \mathcal{A}$ be in general position. We will construct a matrix \mathbf{T} that transforms barycentric coordinates (a_0, a_1, a_2) for P relative to Q_0, Q_1, Q_2 into coordinates for $T(P)$ relative to a frame $F_B = (\vec{w}_0, \vec{w}_1, O_B)^T$ in \mathcal{B} . To do this, let (b_0, b_1, b_2) be the barycentric coordinates of R relative to Q_0, Q_1, Q_2 ,

and let (b'_0, b'_1, b'_2) be the barycentric coordinates for $T(R)$. Since T is projective,

$$\begin{aligned} T(P) &= T([a_0Q_0 + a_1Q_1 + a_2Q_2]) \\ &= [a_0\lambda_0T(Q_0) + a_1\lambda_1T(Q_1) + a_2\lambda_2T(Q_2)], \end{aligned}$$

where $\lambda_i = b'_i/b_i$. Thus,

$$T(P) = \left[\begin{pmatrix} a_0 & a_1 & a_2 \end{pmatrix} \begin{pmatrix} \lambda_0 & 0 & 0 \\ 0 & \lambda_1 & 0 \\ 0 & 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} T(Q_0) \\ T(Q_1) \\ T(Q_2) \end{pmatrix} \right].$$

The points $T(Q_0)$, $T(Q_1)$, $T(Q_2)$ possess coordinates q_{ij} relative to F_B , so we can write

$$T(p) = \left[\begin{pmatrix} a_0 & a_1 & a_2 \end{pmatrix} \underbrace{\begin{pmatrix} \lambda_0 & 0 & 0 \\ 0 & \lambda_1 & 0 \\ 0 & 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} q_{00} & q_{01} & 1 \\ q_{10} & q_{11} & 1 \\ q_{20} & q_{21} & 1 \end{pmatrix}}_{\mathbf{T}} F_B \right] \quad (7.7)$$

The matrix \mathbf{T} is a representation of T in the following sense. Given (a_0, a_1, a_2) , $(p'_0, p'_1, 1)$ can be computed from \mathbf{T} as follows. First compute (x_0, x_1, x_2) by matrix multiplication:

$$(x_0 \quad x_1 \quad x_2) = (a_0 \quad a_1 \quad a_2) \mathbf{T}.$$

Thus,

$$\begin{aligned} T(P) &= [(x_0 \quad x_1 \quad x_2) F_B] \\ &= [x_0\vec{w}_0 + x_1\vec{w}_1 + x_2O_B] \\ &= p'_0\vec{w}_0 + p'_1\vec{w}_1 + O_B \end{aligned}$$

where $p'_0 = x_0/x_2$, $p'_1 = x_1/x_2$. To reiterate, given (a_0, a_1, a_2) , $(p'_0, p'_1, 1)$ can be computed using a two-step procedure:

$$\begin{aligned} (x_0 \quad x_1 \quad x_2) &:= (a_0 \quad a_1 \quad a_2) \mathbf{T} \\ (p'_0 \quad p'_1 \quad 1) &:= \frac{1}{x_2} (x_0 \quad x_1 \quad x_2). \end{aligned}$$

It is only slightly more difficult to find a matrix \mathbf{T}' that carries frame coordinates for p into frame coordinates for $T(p)$ (see Exercise 4 on page 118). Once found, P is transformed through T by applying the two-step procedure above using \mathbf{T}' in place of \mathbf{T} .

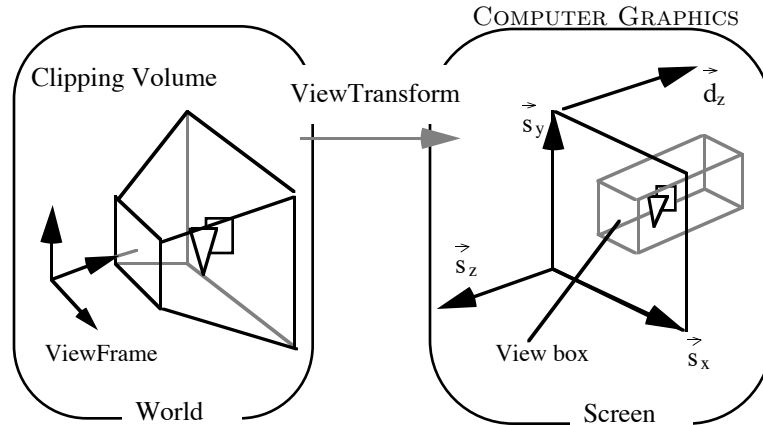


FIG. 7.5: For perspective viewing the viewing transformation is such that the truncated pyramid is mapped to the view box.

7.2. Projective Maps and Perspective Viewing

The idea underlying the creation of hidden surface renderings of scenes viewed in perspective is essentially the same as was used in Section 6.2 where a transformation was constructed to map the clipping volume to the view box (see Figure 6.3). The only difference in perspective viewing is that the clipping volume forms a truncated pyramid instead of a parallelepiped. In other words, to support perspective viewing, we construct a transformation as indicated in Figure 7.5 that carries the truncated pyramid into the view box. This transformation is clearly not affine since parallelism is not preserved. It is, however, projective, meaning that we can fully specify the transformation by specifying the images of five points in general position. Using the five points indicated in Figure 7.6, the desired transformation can be constructed as

```
ViewTransformPM : ProjectiveMap;
ViewTransformPM := PMCreateP( P0, P1, P2, P3, C,
                              P'0, P'1, P'2, P'3, C' : Point)
```

Once the primitives have been mapped to the view box, the hidden surface algorithms of Chapter 6 can be used without modification.

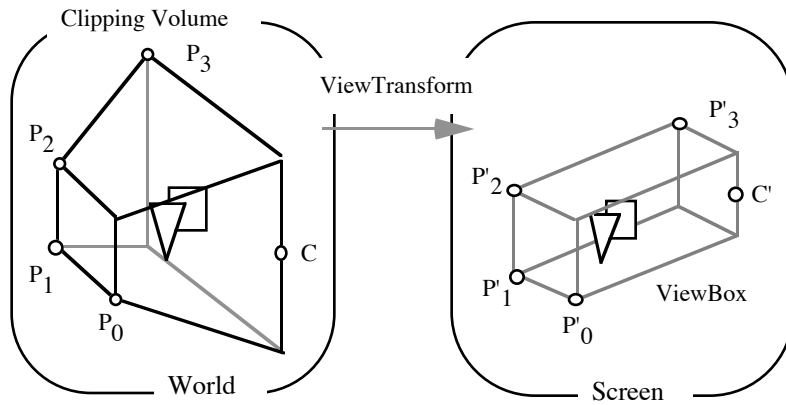


FIG. 7.6: The perspective viewing transformation can be specified as the one carrying P_0, P_1, P_2, P_3, C to $P'_0, P'_1, P'_2, P'_3, C'$.

7.3. Normal Vectors and the Dual Space

In many graphics and modeling applications it is convenient to introduce the idea of a *normal vector*. For instance, in polygonal modeling it is common to represent objects by polyhedra where each vertex is tagged with a normal vector that is used in computing shading information (see Chapter 8). Normal vectors are also important for ray tracing applications since the surface normal determines the direction of a reflected ray, and is one of the determining factors in the direction of a refracted ray.

Unfortunately, the term “normal vector” implies that these objects behave just like other vectors. While this is nearly correct, there are important situations where subtleties can occur. A simplified situation is shown in Figure 7.7 for a hypothetical two-dimensional polygonal modeling application. The left portion of Figure 7.7 represents the definition space of a polygonal approximation to a circle. The right portion of the figure is the image of the polygon under the indicated embedding, in this case a non-uniform scaling. Notice that if the normal vectors are transformed as vectors, then their images do not end up perpendicular to the image of the circle (an ellipse). In fact, they have become more horizontal when they should have become more vertical. Such incorrectly transformed normals can cause visual errors in shading and reflection.

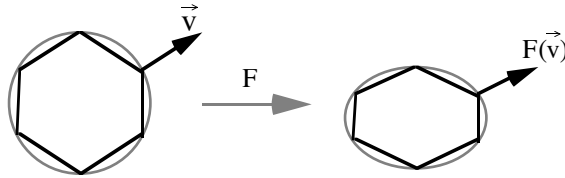


FIG. 7.7: Normals transforming as vectors. The gray circle on the left is being approximated by a set of chords. The vector \vec{v} is normal to the circle at the indicated vertex. If \vec{v} is mapped through the non-uniform scaling F , then $F(\vec{v})$ is not perpendicular to the image of the circle.

One way to understand the problem encountered above is that vectors were being used to represent two different kinds of information. The first (and fundamental) use of vectors is to represent parallelism. (Recall that two lines PQ and $P'Q'$ are parallel if $P - Q$ is a multiple of $P' - Q'$.) The transformation rule for applying affine maps to vectors was constructed precisely to preserve parallelism. The use of vectors to represent outward pointing normals is attempting additionally to use vectors to represent perpendicularity: an outward pointing normal is perpendicular to the tangent line (or tangent plane) of the object being modeled. Since perpendicularity is not preserved under affine maps, vectors fail to be good representatives of tangent lines and tangent planes.

A remedy that is firmly rooted in the fundamentals of geometry is to introduce a new class of objects, classically known as *the dual vectors*, into the algebra. Intuitively, a dual vector will be used to directly represent oriented tangent hyperplanes. It will then be possible to construct a transformation rule for dual vectors that preserves tangency. To make these ideas more precise, we must take a short excursion into the concepts of linear functionals and dual spaces.

Warning: The discussion to follow is intended for the purist who is interested in the algebraic details of dual spaces and how they relate to normal vectors; for those interested primarily in results, the remainder of this section should probably be skipped, at least

on first reading. The results of this section can be summarized as follows:

- Dual vectors should be used when representing perpendicularity to tangent lines and tangent planes.
- Dual vectors are linear functionals, meaning that a dual vector ϕ can be applied to a vector \vec{u} to produce a scalar.
- There is a one-to-one association between vectors and dual vectors in a Euclidean space. A vector \vec{v} is in association with a dual vector λ if for every vector \vec{w} , $\lambda(\vec{w}) = \vec{v} \cdot \vec{w}$. If \vec{v} and λ are paired in this association, they are called duals of one another.
- Dual vectors are represented by the Normal data type in the ADT. It is only really necessary to distinguish between a Vector and a Normal when mapping through affine maps that do not preserve angles, such as shears and non-uniform scaling.

For the moment, let us leave the realm of affine and Euclidean geometry and work instead in the context of vector spaces. A *linear functional* λ on a vector space \mathcal{V} is a map from \mathcal{V} into the reals that satisfies the linearity condition

$$\lambda(\alpha\vec{v} + \beta\vec{w}) = \alpha\lambda(\vec{v}) + \beta\lambda(\vec{w}),$$

for all $\vec{v}, \vec{w} \in \mathcal{V}$ and for all $\alpha, \beta \in \mathfrak{R}$. It turns out that the set of all linear functionals on a vector space \mathcal{V} itself forms a vector space, generally denoted by \mathcal{V}^* (cf. Lang [13]). The vector space \mathcal{V}^* of linear functionals is called the *dual space* of \mathcal{V} . To reinforce the dual nature of the spaces \mathcal{V} and \mathcal{V}^* , the elements of \mathcal{V} are more accurately known as *primal vectors* and the elements of \mathcal{V}^* are called *dual vectors*.

An inner product on the vector space can be used to establish an association between primal vectors and dual vectors. In particular, using the bracket notation for the inner product, if \vec{v} is held fixed, the expression $\langle \vec{v}, \vec{u} \rangle$ is a linear functional whose argument is \vec{u} ; that is, $\lambda(\vec{u}) := \langle \vec{v}, \vec{u} \rangle$ is a linear functional on \mathcal{V} and is therefore a dual vector (associated with the vector \vec{v}). To avoid having to invent a symbol to act as the argument \vec{u} , it is more common to write $\lambda := \langle \vec{v}, \cdot \rangle$.

Using this association, we can define the functional $\langle \vec{v}, \cdot \rangle$ to be the dual of \vec{v} . In equation form,

$$\mathcal{D}_{\vec{v}} := \langle \vec{v}, \cdot \rangle.$$

In this form we recognize that \mathcal{D} is actually a linear mapping from \mathcal{V} to \mathcal{V}^* since

$$\mathcal{D}_{\alpha\vec{v}+\beta\vec{w}} = \alpha\mathcal{D}_{\vec{v}} + \beta\mathcal{D}_{\vec{w}}.$$

In fact, \mathcal{D} is one-to-one and onto, implying that it is also invertible. The definition of \mathcal{D} provides another interpretation of the quantity $\langle \vec{v}, \vec{w} \rangle$. By construction, $\langle \vec{v}, \vec{w} \rangle = \mathcal{D}_{\vec{v}}(\vec{w})$, implying that $\langle \vec{v}, \vec{w} \rangle$ can be obtained by first dualizing \vec{v} , then applying the resulting linear functional to \vec{w} .

It was mentioned in the introduction to this section that dual vectors represent oriented hyperplanes. To see this, notice that $\mathcal{D}_{\vec{v}}(\vec{w})$ vanishes whenever \vec{w} is perpendicular to \vec{v} since perpendicularity implies that $\langle \vec{v}, \vec{w} \rangle = 0$. Recall that in a vector space the set of vectors perpendicular to a fixed vector forms a hyperplane that contains the zero vector. Thus, the linear functional $\mathcal{D}_{\vec{v}}$ represents an oriented hyperplane through the origin perpendicular to \vec{v} . The hyperplane is oriented because we can distinguish a positive and negative side. The vector \vec{w} is on the positive side of the hyperplane if $\mathcal{D}_{\vec{v}}(\vec{w}) > 0$; it is on the negative side if $\mathcal{D}_{\vec{v}}(\vec{w}) < 0$.

To translate the above results from vectors in a vector space into a Euclidean setting, we observe that the freedom of vectors to move about in Euclidean space means that a dual vector defines only the orientation of the hyperplane, but does not fix it absolutely in space. The hyperplane can be fixed by specifying a point through which the hyperplane must pass. Thus, in a Euclidean space an oriented hyperplane is represented as a point B together with a dual vector $\mathcal{D}_{\vec{n}}$. A point Q is in the positive half-space, negative half-space, or on the boundary if the number $\mathcal{D}_{\vec{n}}(Q - B)$ is positive, negative, or zero, respectively. This representation for hyperplanes has in fact already been used in Sections 3.9 and 4.3 in conjunction with the Sutherland-Hodgman clipping algorithm.

At this point it is not clear that we have gained any new insight from the introduction of dual spaces and dual vectors. After all, one could interpret the above discussion as saying nothing more than a plane is defined by a point P and a vector \vec{v} . The advantage of the

dual vector approach is in the determination of how dual vectors, and hence planes and hyperplanes, transform under affine maps.

Let $F : \mathcal{A} \mapsto \mathcal{B}$ be an invertible affine map and let \vec{v} be a vector in $\mathcal{A}\mathcal{V}$. We would like to extend the domain of F to include the dual vectors in such a way that perpendicularity is preserved. This goal can be achieved if we define the action of F on a dual vector $\mathcal{D}_{\vec{v}}$ as

$$F(\mathcal{D}_{\vec{v}}) := \mathcal{D}_{\vec{v}} \circ F^{-1}.$$

To see that perpendicularity is preserved with this definition, let \vec{w} be any non-zero vector in $\mathcal{A}\mathcal{V}$, let \vec{w}' be its image under F ; similarly, let $\mathcal{D}'_{\vec{v}}$ be the image of $\mathcal{D}_{\vec{v}}$ under F . A consequence of the definition is that $\mathcal{D}_{\vec{v}}(\vec{w}) = \mathcal{D}'_{\vec{v}}(\vec{w}')$, since

$$\begin{aligned} \mathcal{D}_{\vec{v}}(\vec{w}) &= \langle \vec{v}, \vec{w} \rangle \\ &= \langle \vec{v}, F^{-1} \circ F\vec{w} \rangle \\ &= \langle \vec{v}, F^{-1}\vec{w}' \rangle \\ &= \mathcal{D}'_{\vec{v}}(\vec{w}'). \end{aligned}$$

Thus, if \vec{w} lies in the hyperplane defined by $\mathcal{D}_{\vec{v}}$ (ie, $\mathcal{D}_{\vec{v}}(\vec{w}) = 0$), then \vec{w}' will lie in the hyperplane defined by $\mathcal{D}'_{\vec{v}}$ (ie, $\mathcal{D}'_{\vec{v}}(\vec{w}') = 0$).

REMARK: As another remark for the purist, we note that dual vectors, i.e., linear functionals, are an instance of the notion of a *covariant tensor*. More specifically, dual vectors are covariant tensors of order one. In general, a covariant tensor of order k is a k -linear map from a vector space into the reals (cf. [Spivak '79]); that is, a map $T(\vec{v}_1, \dots, \vec{v}_k)$ is a covariant tensor of order k if it is linear in each of its arguments. Tensors are useful objects in fields such as differential geometry, continuum mechanics, and relativity theory. Programs designed to solve problems in these areas might therefore benefit from having tensors included in the algebra and the ADT. \square

7.3.1. The Normal Data Type Dual vectors are represented in the geometric ADT by the Normal data type. The routines for manipulation of Normals are:

- Normal \leftarrow NCreate(f : Frame; c1, ... , ck : Scalar)
Return the Normal whose coordinates in frame f are c1, ... , ck.

The coordinates of a Normal are defined to be the coordinates of the vector dual to the normal. Thus, NCreate is equivalent to VDual(VCreate(f , c1, ... , ck)).

- (c1,...,ck : Scalar) \leftarrow NCoords(ϕ : Normal; f : Frame)
Return the coordinates of ϕ relative to f.
- Normal \leftarrow VDual(V : Vector)
Return the dual to vector V.
- Vector \leftarrow NDual(ϕ : Normal)
Return the vector dual to ϕ .
- Scalar \leftarrow NVApply(ϕ : Normal, V : Vector)
Apply ϕ to V, that is, return $\phi(V)$.
- Normal \leftarrow NAXform(ϕ : Normal, T : AffineMap)
Return the image of ϕ under the affine map T ; T is assumed to be invertible.

7.3.2. *Matrix Representations of Dual Vectors Rather than representing dual vectors as row matrices as was done for points and vectors, dual vectors are most naturally represented as column matrices. This is most easily seen by considering the coordinate computation of the quantity $\mathcal{D}_{\vec{v}}(\vec{w}) = \langle \vec{v}, \vec{w} \rangle$. If we expand \vec{v} and \vec{w} into their coordinates relative to a *Cartesian frame* $(\vec{e}_1, \dots, \vec{e}_n, \mathcal{O})^T$, we find that

$$\mathcal{D}_{\vec{v}}(\vec{w}) = \langle v_1\vec{e}_1 + \dots + v_n\vec{e}_n, w_1\vec{e}_1 + \dots + w_n\vec{e}_n \rangle.$$

Bi-linearity is used to rewrite this as

$$\mathcal{D}_{\vec{v}}(\vec{w}) = \sum_{i,j} v_i w_j \langle \vec{e}_i, \vec{e}_j \rangle \quad (7.8)$$

Since the basis vectors are ortho-normal, all cross terms (those with $i \neq j$) vanish leaving

$$\mathcal{D}_{\vec{v}}(\vec{w}) = v_1 w_1 + \dots + v_n w_n.$$

Notice that this computation can be written in matrix form as the product of a row vector and a column vector:

$$\mathcal{D}_{\vec{v}}(\vec{w}) = (w_1 \ \dots \ w_n \ 0) (v_1 \ \dots \ v_n \ 0)^T$$

. The row vector is the matrix representation of \vec{w} relative to $(\vec{e}_1, \dots, \vec{e}_n, \mathcal{O})^T$, so we can interpret the column vector as the matrix representation of $\mathcal{D}_{\vec{v}}$ relative to the same frame. Multiplication of these matrices corresponds to the application of $\mathcal{D}_{\vec{v}}$ on \vec{w} , and hence results in the value $\mathcal{D}_{\vec{v}}(\vec{w})$.

Recall that \mathcal{D} was defined as a linear mapping between a vector space and its dual space. The fact that a vector \vec{v} having coordinates $(v_1, \dots, v_n, 0)$ is represented by the row matrix $(v_1 \ \cdots \ v_n \ 0)$ and has a dual $\mathcal{D}_{\vec{v}}$ represented by the column matrix $(v_1 \ \cdots \ v_n \ 0)^T$ implies that the mapping \mathcal{D} is realized by the matrix transpose operator. Since the transpose operator is its own inverse, the inverse of \mathcal{D} is also realized by the matrix transpose operator.

Given that dual vectors can be represented as column matrices, we now consider the question of how these matrices transform under the action of affine maps. More precisely, let $F : \mathcal{A} \mapsto \mathcal{B}$ be an affine map whose matrix representation relative to Cartesian frames in \mathcal{A} and \mathcal{B} is \mathbf{F} , and let \mathbf{w} and \mathbf{v} be the row and column matrices, respectively, that represent \vec{w} and $\mathcal{D}_{\vec{v}}$ relative to the chosen basis. Similarly, let \mathbf{w}' and \mathbf{v}' be the matrix representations of the images of \vec{w} and $\mathcal{D}_{\vec{v}}$ under \mathbf{F} . With these definitions, it is \mathbf{v}' that we seek. This column vector can be obtained by expanding $\mathcal{D}_{\vec{v}}(\vec{w})$ in matrix notation:

$$\begin{aligned} \mathcal{D}_{\vec{v}}(\vec{w}) &= \mathbf{w} \mathbf{v} \\ &= \mathbf{w} \mathbf{F} \mathbf{F}^{-1} \mathbf{v} \\ &= \mathbf{w}' \mathbf{F}^{-1} \mathbf{v} \end{aligned} \tag{7.9}$$

It was shown earlier that $\mathcal{D}_{\vec{v}}(\vec{w})$ was invariant under affine maps, implying that $\mathcal{D}_{\vec{v}}(\vec{w}) = \mathcal{D}'_{\vec{v}}(\vec{w}') = \mathbf{w}' \mathbf{v}'$. Comparing this with Equation 7.9 reveals that

$$\mathbf{w}' \mathbf{F}^{-1} \mathbf{v} = \mathbf{w}' \mathbf{v}',$$

which can be rewritten as

$$\mathbf{w}' (\mathbf{F}^{-1} \mathbf{v} - \mathbf{v}') = 0.$$

If \mathbf{w}' were a totally arbitrary row vector we could use the non-singularity of \mathbf{F} to deduce that

$$\mathbf{v}' = \mathbf{F}^{-1} \mathbf{v}. \tag{7.10}$$

There is, however, a problem with taking Equation 7.10 to be the matrix expression for the transformation of dual vectors.³ Consider the case when F is a translation, meaning that \mathbf{F} is of the form given in Example 5. The inverse matrix is therefore

$$\mathbf{F}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{pmatrix}.$$

Using this matrix in Equation 7.10 can result in \mathbf{v}' having a non-zero last component, meaning that it cannot be used as the representation of a dual vector. The solution to this dilemma is hidden in the fact that \mathbf{w}' is not totally arbitrary because its last component must be zero. This means that all but the last component of \mathbf{v}' must agree with $\mathbf{F}^{-1} \mathbf{v}$, but the last component can, without loss of generality, be chosen to be zero. This choice can be forced by introducing a $(n+1) \times (n+1)$ matrix \mathbf{Z} that contains all zero elements except for ones on the first n diagonals. We therefore take as our transformation rule for dual vectors the matrix expression

$$\mathbf{v}' = \mathbf{Z} \mathbf{F}^{-1} \mathbf{v}. \quad (7.11)$$

One further caveat is in order: Equation 7.11 is only valid when coordinates are expressed relative to Cartesian frames. If coordinates are expressed in a non-Cartesian frame, the transformation rule becomes slightly more complicated. The simplification occurs for Cartesian frames because the cross terms in Equation 7.8 are guaranteed to vanish. We leave the generalization to arbitrary frames as an exercise.

Exercises

1. Show that the composition of two projective maps is projective.
2. What goes wrong if the pre-image points used to define a projective map are not in general position?
3. Show that planes map to planes under projective maps.
4. Extend the results of Section 7.1.2 to determine a matrix \mathbf{T}' that carries frame coordinates for a point P into frame coordinates for the point $T(P)$, where T is a projective map.

³Thanks to Richard Bartels for pointing this out.

5. Derive the matrix transformation rule for dual vectors when arbitrary (i.e., non Cartesian) frames in \mathcal{A} and \mathcal{B} are chosen.

Color and Shading

To create images with smoothly varying, physically plausible, color variations, we must develop models that approximate the interaction of light with surfaces. Before developing these lighting models, we shall look more closely at what is meant by “color” by taking a short detour through the basics of tri-stimulus color theory.

8.1. Tri-Stimulus Color Theory

Consider the artificially simple situation of a human observer looking at a single colored light source. We know from elementary physics that the light source can be physically characterized by an emission spectrum $I(\lambda)$ that assigns an intensity of emission to each wavelength λ of the electromagnetic spectrum. A device such as a spectrophotometer could be used to physically measure the emission spectrum from the light source. The human observer on the other hand would describe the perceived color of the light, perhaps using words such as “bluish yellow” or “light green”. The key word here is *perceives*: whereas nature generates spectra, humans perceive color.

The distinction between spectra and color has been made strikingly clear through experiments in color theory. It is possible for instance to present two very different spectra to an observer without the observer being able to tell them apart. (Two distinct spectra that are perceived as being identical are called *metamers*.) The implication of such experiments is that the number of perceivable colors is much smaller than the number of spectra. This implication is corroborated by biological evidence. A (normal) human retina possesses three different types of color receptors. Roughly speaking, these receptors respond most strongly to long wavelengths, medium

FIG. 8.1: Response curves for the short (S), medium (M), and long (L) wavelength human color receptors.

wavelengths, and short wavelengths, of light respectively. The activity levels for each of the receptors for a variety of wavelengths have been measured, resulting in the *response curves* shown in Figure 8.1. Since the long wavelength receptor peaks in the red portion of the spectrum, it is commonly referred to as the red receptor. The medium and short wavelength receptors are commonly known as the green and blue receptors, respectively.

Tri-stimulus color theory attempts to explain the association between spectra and color by modeling the low-level or “early” human vision system as a linear mapping $V : \Lambda \rightarrow \mathbf{C}$, where Λ is the (infinite dimensional) vector space of continuous functions of wavelength, and \mathbf{C} is a three-dimensional vector space called *color space*. Specifically, tri-stimulus theory states that V is given by given by

$$V(I(\lambda)) = \vec{c} = \ell\vec{\ell} + m\vec{m} + s\vec{s}. \quad (8.1)$$

where the vectors $\vec{\ell}, \vec{m}, \vec{s}$ form a basis for \mathbf{C} . The scalars $\ell, m,$ and s indicate the level of activity of each of the the three color receptors in response to stimulation by the spectrum $I(\lambda)$; they are computed

according to

$$\begin{aligned} \ell &= \int_{-\infty}^{\infty} L(\lambda)I(\lambda)d\lambda \\ m &= \int_{-\infty}^{\infty} M(\lambda)I(\lambda)d\lambda \\ s &= \int_{-\infty}^{\infty} S(\lambda)I(\lambda)d\lambda \end{aligned}$$

where L , M , and S are the receptor response functions of Figure 8.1.

Since the range of V has a smaller dimension than its domain, it is necessarily a many-to-one mapping, thereby offering an explanation for the existence of metamers.

Although tri-stimulus theory is adequate for many computer graphics tasks, it is not adequate for all purposes since there are many phenomena that the theory fails to predict. For instance, we have all noticed that the color of a piece of fabric or a paint chip seems to change depending on what other colors are nearby. Theories that adequately explain such effects are still an active area of research.

8.1.1. Reproducing Spectral Responses with Frame Buffers

A natural problem to consider in the context of computer graphics is:

Given: A spectrum $I(\lambda)$.

Find: r , g , b values to store in a frame buffer so that the color viewed on the screen evokes the same perceptual response as $I(\lambda)$.

To examine this question more closely, we first note that the specific r , g , b values will clearly depend on the specifics of the monitor. In particular, it will be critical to know the emission spectra $R(\lambda)$, $G(\lambda)$ and $B(\lambda)$ of each of the red, green, and blue phosphors, respectively. Assuming the monitor has been gamma corrected, the synthesized spectrum $I'(\lambda)$ appearing on the monitor corresponding to a pixel value (r, g, b) is

$$I'(\lambda) = rR(\lambda) + gG(\lambda) + bB(\lambda). \quad (8.2)$$

Given $I(\lambda)$ we wish to compute (r, g, b) such that $I(\lambda)$ and $I'(\lambda)$ are metamers; that is, we require

$$V(I(\lambda)) = \vec{c} = V(I'(\lambda)). \quad (8.3)$$

To solve the problem, we first note that as a consequence of the linearity of integration, the mapping V is a linear transformation, meaning that for any two spectra $X(\lambda)$ and $Y(\lambda)$, and for any scalars a and b ,

$$V(aX(\lambda) + bY(\lambda)) = aV(X(\lambda)) + bV(Y(\lambda)).$$

Linearity of V implies that

$$\begin{aligned} V(I'(\lambda)) &= rV(R(\lambda)) + gV(G(\lambda)) + bV(B(\lambda)) \\ &= r\vec{r} + g\vec{g} + b\vec{b}, \end{aligned}$$

where the vectors \vec{r} , \vec{g} , and \vec{b} are $V(R(\lambda))$, $V(G(\lambda))$ and $V(B(\lambda))$, respectively. Substituting Equations 8.4 and 8.1 into Equation 8.3, we find that

$$\vec{c} = r\vec{r} + g\vec{g} + b\vec{b} = \ell\vec{\ell} + m\vec{m} + s\vec{s}. \quad (8.4)$$

In vector space terms, Equation 8.4 states that (r, g, b) are the coordinates of \vec{c} in the basis $(\vec{r}, \vec{g}, \vec{b})$, and (ℓ, m, s) are the coordinates for \vec{c} in the basis $(\vec{\ell}, \vec{m}, \vec{s})$. We therefore recognize the calculation of (r, g, b) from (ℓ, m, s) as a simple change of coordinates; hence there must exist a 3×3 matrix \mathbf{A} such that

$$\begin{pmatrix} r & g & b \end{pmatrix} = \begin{pmatrix} \ell & m & s \end{pmatrix} \mathbf{A}.$$

In fact, it can be shown that (see Exercise 1 on page 134)

$$\mathbf{A} = \begin{pmatrix} \alpha_{RL} & \alpha_{RM} & \alpha_{RS} \\ \alpha_{GL} & \alpha_{GM} & \alpha_{GS} \\ \alpha_{BL} & \alpha_{BM} & \alpha_{BS} \end{pmatrix}^{-1} \quad (8.5)$$

where

$$\begin{aligned} \alpha_{XY} &= \int_{-\infty}^{\infty} X(\lambda)Y(\lambda)d\lambda \\ X &\in \{R, G, B\}, \quad Y \in \{L, M, S\}. \end{aligned}$$

The change of basis matrix \mathbf{A} can be computed once the emission spectra are known for the monitor on which the input spectra are to be reproduced. Once \mathbf{A} is computed, the (r, g, b) triple corresponding to a spectrum $I(\lambda)$ can be found by first computing (ℓ, m, s) , then multiplying by \mathbf{A} to obtain (r, g, b) .

Our coordinate-free machinery can be used to explain the above process as follows. The human visual system corresponds to the basis $(\vec{\ell}, \vec{m}, \vec{s})$ for the color space \mathbf{C} since coordinates relative to this basis indicate the level of activity in human color receptors. Each monitor corresponds to a different basis $(\vec{r}, \vec{g}, \vec{b})$, where the relationship between the “monitor basis” and the “human basis” depends on the phosphor emission spectra of the monitor. The monitor basis is therefore the color equivalent of the (spatial) device frame imposed on screen space.

8.1.2. The CIE Color System An organization called the Commission Internationale de l’Éclairage (CIE) has devised a color reproduction process that is somewhat more accurate than the process above (inaccuracies in the above process are due to deficiencies in the tri-stimulus theory). Rather than computing (r, g, b) by first computing (ℓ, m, s) from the spectrum, the CIE developed a two step process wherein the coordinates (x, y, z) of $V(I(\lambda))$ relative to an agreed upon standard basis $(\vec{x}, \vec{y}, \vec{z})$ for color space are first computed. The (r, g, b) coordinates appropriate for a particular monitor are computed from (x, y, z) using a change of basis matrix.

Increased accuracy was achieved by the CIE by experimentally determining lookup tables that represent the mapping from spectra to (x, y, z) . This was done by presenting human observers with colored lights corresponding to large number of different wavelengths. For each wavelength, the subjects were asked to adjust the red, green, and blue values on a high quality color monitor until the color on the monitor matched the color of the spectrum. This resulted in a table that mapped directly from spectra to (r, g, b) for that particular monitor. The effects of the monitor were then factored out by performing a change of coordinates from $(\vec{r}, \vec{g}, \vec{b})$ to $(\vec{x}, \vec{y}, \vec{z})$. The famous *CIE chromaticity diagram* corresponds to the colors lying on the plane

$$x\vec{x} + y\vec{y} + z\vec{z}, \quad x + y + z = 1.$$

8.2. Lighting Models

The problem to be addressed in this section can be stated roughly as follows:

FIG. 8.2: A lighting model is used to determine the intensity of light propagating from a point P visible to the viewer through pixel p . The vector $\hat{\ell}$ points toward the point light source and the vector \hat{v} points toward the viewer.

Given: A point P on the surface of a geometric primitive that has been determined to be visible to the viewer through a pixel p , a unit vector \hat{v} from P to the viewer, and a unit vector $\hat{\ell}$ from P toward a point light source (see Figure 8.2).

Find: The intensity and color of light radiating from P back toward the viewer. (The computed radiated intensity is used to set the color of pixel p .)

The above problem is solved by developing a model of the physical interaction of light with materials. Such a model has come to be known as a *shading* or *lighting* model. Our approach will be to begin with a very simple lighting model, then successively embellish it to obtain a series of lighting models that offer increasing realistic shading effects at the price of higher computational cost.

FIG. 8.3: The situation at a point P to be illuminated.

8.2.1. Lambertian Shading This is a simple lighting model based on the assumption that the incident light is uniformly reradiated in all directions. This is the so-called *diffuse* or *Lambertian* assumption (named after the French physicist Lambert). To develop a mathematical model based on the diffuse assumption, we use a linearity assumption together with the definition of intensity as the ratio of power (energy/sec) to area. The linearity assumption is that the outgoing energy is proportional to the incident energy. This assumption is very accurate under normal viewing conditions, but it is certainly violated for violent conditions such as bombardment by powerful lasers.

We seek an expression that relates the outgoing intensity I_{out} to the incident intensity I_{in} . To do this, we first determine the incident energy striking a differential surface area $\Delta A_{surface}$ of the surface around P . Referring to Figure 8.3, the incident power $E_{in}(\hat{\ell})$ in the direction of $\hat{\ell}$ is

$$E_{in}(\hat{\ell}) = I_{in}(\hat{\ell})\Delta A_{in} \quad (8.6)$$

where ΔA_{in} is the cross sectional area of the incident beam that corresponds to the area $\Delta A_{surface}$ on the surface, and where $I_{in}(\hat{\ell})$

denotes the incident intensity arriving from direction $\hat{\ell}$. Using the linearity assumption, we write

$$E_{out}(\hat{v}) = \phi(\hat{v}, \hat{\ell}) E_{in}(\hat{\ell}) \quad (8.7)$$

where $\phi(\hat{v}, \hat{\ell})$ is the constant of proportionality that relates the incoming power from the direction $\hat{\ell}$ to the outgoing power in the direction of \hat{v} ; it is a property of the material, characterizing how energy is reflected off the surface. Combining Equations 8.6 and 8.7 and using the relation

$$I_{out}(\hat{v}) = \frac{E_{out}(\hat{v})}{\Delta A_{out}},$$

we find that

$$I_{out}(\hat{v}) = \phi(\hat{v}, \hat{\ell}) I_{in}(\hat{\ell}) \frac{\Delta A_{in}}{\Delta A_{out}} \quad (8.8)$$

$$= \phi(\hat{v}, \hat{\ell}) I_{in}(\hat{\ell}) \frac{\Delta A_{in}}{\Delta A_{surface}} \frac{\Delta A_{surface}}{\Delta A_{out}}. \quad (8.9)$$

Referring to Figure 8.3, the ratio of ΔA_{in} to $\Delta A_{surface}$ is $\cos \theta_{in}$; similarly, the ratio of ΔA_{out} to $\Delta A_{surface}$ is $\cos \theta_{out}$, implying that

$$I_{out}(\hat{v}) = \phi(\hat{v}, \hat{\ell}) I_{in}(\hat{\ell}) \frac{\cos \theta_{in}}{\cos \theta_{out}} \quad 0 \leq \theta_{in}, \theta_{out} \leq \pi/2. \quad (8.10)$$

The diffuse reflection assumption states that $I_{out}(\hat{v})$ is independent of \hat{v} , meaning that

$$\phi(\hat{v}, \hat{\ell}) = \rho(\hat{v}, \hat{\ell}) \cos \theta_{out}. \quad (8.11)$$

The function $\rho(\hat{v}, \hat{\ell})$ is called the *bidirectional reflectance*. The bidirectional reflectance for a diffusely reflecting surface is assumed to be a constant k_d . Using Equation 8.11, Equation 8.10 can be rewritten as

$$I_{out}(\hat{v}) = k_d I_{in}(\hat{\ell}) (\hat{\ell} \cdot \hat{n})_+ \quad (8.12)$$

where $(x)_+$ is defined to be x if $x \geq 0$, and zero otherwise.

Equation 8.12 is the *diffuse lighting model*. As a consequence of the linearity assumption, multiple light sources can be modeled simply by summing the contributions from each light source. Notice however that $I_{in}(\hat{\ell})$ as used in Equation 8.12 is the intensity of the

light source as measured at P . The conservation of energy principle from physics predicts that the intensity of a point light source obeys the inverse square law – the intensity falls off as one over the distance from the source squared. Thus,

$$I_{\text{in}}(\hat{\ell}) = \frac{I_0}{r^2}$$

where r is the distance from the light source to P , and I_0 is the intensity of the source as measured at unit distance. Although the inverse square law is physically accurate for a point source, the lighting effects produced using it are extremely harsh. One reason for the harshness is that true point light sources are never encountered in everyday experience. The lighting effects can be improved in a number of ways. The most physically justified method is to model light sources with finite extent by integrating Equation 8.12 over the area of the light source. This is in fact the approach taken by a number of *global illumination* methods that will be discussed in Chapter ???. A much less expensive (empirical) approximation is to model the intensity fall off as

$$I_{\text{in}}(\hat{\ell}) = \frac{I_0}{\gamma + r}$$

where γ is a constant associated with the light source.

It should also be noted that $I_{\text{out}}(\hat{v})$ is the intensity leaving P toward the viewer. The pixel p through which P is visible should be set to the intensity as measured at p . Thus, strictly speaking, p should be set to an intensity $I_{\text{out}}(\hat{v})/d^2$, where d is the distance from P to p . Once again, this fast decay in intensity produces illumination effects that are too harsh. It is therefore very common in graphics to simply ignore the intensity fall off between P and p .

Colored light sources reflecting off colored surfaces are most accurately modeled by treating I_{in} , I_{out} , and k_d in Equation 8.12 as functions of wavelength. Once the final spectrum $I(\lambda)$ is determined, it can be mapped to a color by applying the mapping V given in Equation 8.1. The physically correct method is therefore to perform all calculations in Λ , then map to \mathbf{C} to obtain a color.

In computer graphics, however, it is traditional to handle color by maintaining red, green, and blue color components separately. For example, color light sources are typically characterized by their

red, green, and blue intensities; that is, a light source with spectrum $I(\lambda)$ is most often represented by the (r, g, b) coordinates of $V(I(\lambda))$. The diffuse reflectances of colored materials are similarly characterized by three coefficients $(k_{d,\text{red}}, k_{d,\text{green}}, k_{d,\text{blue}})$. For instance, a predominantly red material would be characterized by a value of $k_{d,\text{red}}$ near one, and values of $k_{d,\text{green}}$ and $k_{d,\text{blue}}$ near zero.

The approach traditionally taken in computer graphics is essentially to perform all calculations in \mathbf{C} rather than in Λ . There are a number of difficulties with this (virtually universal) approach. For instance, the physically correct color corresponding to the calculation $I_1(\lambda) + k(\lambda) * I_2(\lambda)$ in Λ is

$$V(I_1(\lambda) + k(\lambda) * I_2(\lambda)) = V(I_1(\lambda)) + V(k(\lambda) * I_2(\lambda)). \quad (8.13)$$

The component-wise color computed by the usual computer graphics approach can be formally described by

$$V(I_1(\lambda)) + V(k(\lambda)) \otimes V(I_2(\lambda)), \quad (8.14)$$

where \otimes is defined follows: if $\vec{x} = r_x \vec{r} + g_x \vec{g} + b_x \vec{b}$ and $\vec{y} = r_y \vec{r} + g_y \vec{g} + b_y \vec{b}$, then

$$\vec{x} \otimes \vec{y} = r_x r_y \vec{r} + g_x g_y \vec{g} + b_x b_y \vec{b}.$$

The two results given in Equations 8.13 and 8.14 will, in general, not be equal since V does not preserve multiplication of functions. More precisely, if $X(\lambda), Y(\lambda) \in \Lambda$, $V(X(\lambda) * Y(\lambda))$ is not, in general, equal to $V(X(\lambda)) \otimes V(Y(\lambda))$.

A number of groups, led primarily by Cornell University, have been advocating the physically correct approach of performing lighting calculations in Λ , using for instance, piecewise linear approximations of functions. Unfortunately, the approach is still not widespread. In the remainder of this chapter, we succumb to the sin of performing calculations in \mathbf{C} using component-wise computations for the red, green, and blue color coordinates.

Putting all the pieces together, we set the color components of pixel p to

$$I_{out,c}(\hat{v}) = \sum_{\text{light source } i} \rho_c(\hat{v}, \hat{\ell}_i) \frac{I_{0,c}^i}{\gamma_i + r_i} (\hat{\ell}_i \cdot \hat{n})_+ \quad (8.15)$$

$$c \in \{\text{red, green, blue}\}$$

where $\rho_c(\hat{v}, \hat{\ell}) = k_{d,c}$.

Finally, we mention another simple type of light source called a *directional light source*. A directional source models a distant point light source such as the sun. Unlike a point light source where the vector $\hat{\ell}$ depends on the point P being illuminated, directional light sources are modeled using a vector $\hat{\ell}$ that does not depend on P , thus capturing the fact that incoming rays from the light are parallel. Another difference is that since the light rays are parallel, there is no intensity fall off with distance.

8.2.2. Ambient Lighting The images produced by the lighting model of Equation 8.15 are still rather harsh. One reason for the harshness is that only direct illumination is modeled – there is no account taken of indirect illumination that is so often present in natural environments. In an office, for instance, the floor underneath a desk receives some light even though the area is not directly illuminated by a light source. The global illumination algorithms in Chapter ?? are specifically designed to accurately model indirect illumination. A reasonably effective but very simple empirical method is to model only *ambient* illumination. Ambient illumination refers to a completely uniform level of light that is visible to all surfaces. The light model can be augmented to including ambient illumination by replacing Equation 8.15 with

$$I_{out,c}(\hat{v}) = \rho_{a,c}I_{a,c} + \sum_{\text{light source } i} \rho_c(\hat{v}, \hat{\ell}_i) \frac{I_{0,c}^i}{\gamma_i + r_i} (\hat{\ell}_i \cdot \hat{n})_+ \quad (8.16)$$

$$c \in \{\text{red, green, blue}\}$$

where

- $\rho_c(\hat{v}, \hat{\ell}) = k_{d,c}$ is the diffuse bidirectional reflectance.
- $\rho_{a,c}$ is the ambient bidirectional reflectance. It is common to assume that the material diffusely reflects the ambient illumination, meaning that $\rho_{a,c} = k_{d,c}$.
- $I_{a,c}$ is the intensity of the ambient illumination.

8.2.3. Specular Reflection Specular (shiny) surfaces are not modeled well by diffuse reflection since they exhibit preferential reradiation. The observed intensity of a point P on a near-perfect mirror surface, for instance, depends on the position of the viewer relative to the light source. If the viewer is positioned so that the reflection of the light source is visible at P , the intensity at P appears to be very bright. The variation of the intensity of a point with the position of a viewer is not an effect captured by the diffuse model.

Torrance and Sparrow developed a model of specularly reflecting surfaces by treating the surface as being composed of microscopic perfectly reflecting flat facets called *microfacets* [18, 19]. The microfacets are assumed to have normal vectors oriented in a distribution about the macro surface normal \hat{n} , thereby modeling the microscopic flaws that prevent real materials from being perfectly mirror-like. Torrance and Sparrow show that the bidirectional reflectance function for this model of specular reflection is of the form

$$\rho_s(\hat{v}, \hat{\ell}) = \sigma \frac{DG}{(\hat{n} \cdot \hat{v})(\hat{n} \cdot \hat{\ell})}$$

where

- σ is a constant.
- D is a function describing the distribution of microfacet normals.
- G is a self-shadowing factor, accounting for the shadowing of some microfacets by other microfacets.

Although the Torrance-Sparrow model was known in the physics literature in the late 60's, Phong Bui-Tuong, working at the University of Utah in the computer graphics laboratory in the mid '70s, independently developed an empirical model of specular reflection that is qualitatively very similar to the Torrance-Sparrow model. In particular, Bui-Tuong used a model of specularity that is equivalent to the bidirectional reflectance

$$\rho_{s,c}(\hat{v}, \hat{\ell}) = k_{s,c} \frac{(\hat{r} \cdot \hat{v})_+^p}{(\hat{n} \cdot \hat{\ell})_+}, \quad c \in \{\text{red, green, blue}\}$$

where \hat{r} is a unit vector pointing in the *mirror direction* as shown in Figure 8.4. This model of specular reflection has come to be known

FIG. 8.4: The quantities necessary for the Phong lighting model.

as the *Phong lighting model*. The maximum intensity occurs when the viewing direction \hat{v} aligns with the mirror direction \hat{r} , since the term $(\hat{r} \cdot \hat{v})_+^p$ reaches its maximum value of one. The number p , called the *Phong exponent*, controls the rate at which a specular reflection falls off as \hat{v} moves away from \hat{r} , thereby controlling the width of specular highlights as seen on the surface. Small values of p cause broad highlights whereas large values of p create narrow, sharply focused highlights. The Phong exponent then roughly corresponds to the “shininess” of the surface, with large values of p corresponding to very shiny materials. In terms of the Torrance-Sparrow microfacet model, p corresponds to the width of the distribution of microfacet normals about the macro surface normal \hat{n} .

The values $k_{s,\text{red}}, k_{s,\text{green}}, k_{s,\text{blue}}$ characterize the degree to which the surface specularly reflects in each of the color channels. These values effectively control the color of specular highlights.

A lighting model incorporating the effects of ambient, diffuse, and the Phong model of specularity is given by

$$I_{out,c}(\hat{v}) = \rho_{a,c} I_{a,c} + \sum_{\text{light source } i} \rho_c(\hat{v}, \hat{\ell}_i) \frac{I_{0,c}^i}{\gamma_i + r_i} (\hat{\ell}_i \cdot \hat{n})_+ \quad (8.17)$$

where

$$\rho_c(\hat{v}, \hat{\ell}_i) = k_{d,c} + k_{s,c} \frac{(\hat{r}_i \cdot \hat{v})_+^p}{(\hat{n} \cdot \hat{\ell}_i)_+}$$

Exercises

1. Prove that the matrix \mathbf{A} that changes coordinates relative to $(\vec{\ell}, \vec{m}, \vec{s})$ to coordinates relative to $(\vec{r}, \vec{g}, \vec{b})$ has the form given in Equation 8.5.
2. Develop a formula for computing the mirror direction vector \hat{r} from \hat{n} , and $\hat{\ell}$.

References

- [1] W. Bouknight. A procedure for generation of three-dimensional half-toned computer graphics representations. *Communications of the ACM*, 13(9):527–536, September 1970.
- [2] W. Bouknight and K. Kelly. An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources. *SJCC*, pages 1–10, 1970.
- [3] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.
- [4] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Computer Science Department, University of Utah, December 1974.
- [5] E. Catmull. A tutorial on compensation tables. In *Proceedings of SIGGRAPH*, pages 279–285, 1979.
- [6] C. de Boor. B-form basics. In G. Farin, editor, *Geometric Modeling: Algorithms and New Trends*, pages 131–148. 1987.
- [7] Manfred P. do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.
- [8] C. Dodson and T. Poston. *Tensor Geometry: The Geometric Viewpoint and its Uses*. Pitman, London, 1979.
- [9] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Addison-Wesley, New York, second edition edition, 1990.
- [10] F. Flohr and F. Raith. Affine and Euclidean geometry. In H. Behnke, F. Bachmann, K. Fladt, and H. Kunle, editors, *Fundamentals of Mathematics, Volume II*, pages 293–383. MIT Press, 1974.
- [11] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, New York, 1990.
- [12] R. Goldman. Vector geometry: A coordinate-free approach. Siggraph Course Notes, Course No. 19, 1987.
- [13] Serge Lang. *Algebra*. Addison-Wesley, Redwood City, California, second edition, 1984.
- [14] M. O’Nan. *Linear Algebra*. Harcourt Brace Jovanovich, Inc., New York, second edition edition, 1976.
- [15] M. L. V. Pitteway. Algorithm for drawing ellipses or hyperbolae with a digital plotter. *Computer Journal*, 10(3):282–289, 1967.
- [16] R. Sproull and I. Sutherland. A clipping divider. In *FJCC*, pages 765–775. Thompson Books, Washington, D.C., 1968.
- [17] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.
- [18] K. Torrance and E. Sparrow. Theory of off-specular reflection from roughened surfaces. *Journal of the Optical Society of America*, 57(9):1105–1114, 1967.

- [19] K. Torrance, E. Sparrow, and R. Birkebak. Polarization, directional distributional, and off-specular peak phenomena in light reflected from roughened surfaces. *Journal of the Optical Society of America*, 56(7):916–925, 1966.
- [20] J. Warnock. A hidden-surface algorithm for computer generated half-tone pictures. Computer Science Department TR 4-15, NTIS AD-753 671, University of Utah, 1969.
- [21] G. Watkins. A real-time visible surface algorithm. Computer Science Department UTEC-CSc-70-101, NTIS AD-762 004, University of Utah, June 1970.
- [22] K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. In *Proceedings of SIGGRAPH*, pages 214–??, 1977.
- [23] H. Weyl. *Space, Time, and Matter*. Methuen & Co., London, 1922. Translated from German by H. Brose.
- [24] C. Wylie, G. Romney, D. Evans, and A. Erdahl. Halftone perspective drawings by computer. *FJCC 1967*, pages 49–58, 1967.