CSM213 Computer Graphics

Vincent Zammit, modified by Mike Rizzo and Kurt Debattista
Department of Computer Science and A.I.
University of Malta

February 1995, October 1997, February 2001

Contents

Chapter 1

Brief Overview of Graphics Systems

1.1 Introduction

- This chapter gives an introduction to some basic concepts underlying computer graphics.
- These notes summarize the main points covered in the lectures. Reading the following texts is recommended.

1.1.1 Recommended Texts

- **Introduction to Computer Graphics**. Foley, Van Dam, Feiner, Hughes and Phillips. Addison-Wesley Publishing Company.
- **Computer Graphics**. Donald Hearn and M. Pauline Baker. Prentice Hall International Editions.
- **Computer Graphics - A Programming Approach**. Steven Harrington. McGraw-Hill International Editions.

1.1.2 Course Structure

The course is divided into 3 parts:

1. Graphics Primitives
2. 2D and 3D Transformations
3. 3D Viewing and Representation

1.2 Applications of Computer Graphics

The use of computer graphics is becoming more popular in:

- **User Interfaces**: Today, virtually every software package provides an adequate graphical user interface.

- **Computer Aided Design**: Computers play an important role in the design and manufacture of vehicles, buildings, appliances, etc due to their ability to produce and amend designs interactively.
- **Visualisation**: Graphical representation of scientific, statistical, business, etc data gives the users an effective and illustrative presentation of numeric results.
- **Computer Aided Learning**: Interactive simulators and other graphics based learning systems are becoming an indispensable aid to education and specialised training.
- **Entertainment**: Advertisement, motion pictures, games, etc are constantly taking advantage of the ability of computers to develop realistic and detailed graphical images.

1.3 Graphics Systems

Graphics output devices can be classifies as:

- Random scan devices, or
- Raster scan devices.

1.3.1 Random Scan Displays

- Random scan, or *vector* devices accept a sequence of coordinates and display images by moving a drawing *pointer* (a pen for the case of a plotter, an electron beam for the case of a random scan cathode ray tube) as required.
- In a random scan system the image is stored in the system memory as a list of primitive drawing commands called the *display list* or *refresh buffer*. A *display processor*, or *graphics controller* reads and interprates the display list and outputs the plotting coordinates to the display device.
- Random scan devices produce precise and fast wireframe images.
- However they are not suitable for generating realistic images.
- Random scan devices include:
  - Plotters
  - CRT

1.3.2 Raster Scan Displays

- Raster scan devices treat an image as a rectangular matrix of intensity and colour values. They accept a sequence of colour/intensity values generating the image by displaying the individual spots as required.
- In a raster scan system the image is usually stored in a memory area called the *frame buffer* or *refresh buffer* as an array of *pixels*. The *video controller* scans the frame buffer and outputs the colour intensity values to the display device.

- Raster scan systems are well suited for displaying realistic images since pattern filled areas, shading, etc can be easily stored in the frame buffer as different colour intensity values.
- However the output lacks the accuracy and well definition of random scan systems.
- Raster scan devices include:
  - Raster scan monitors
  - Dot matrix printers

Chapter 2

Graphics Primitives: Line Drawing Algorithms

2.1 Scan Conversion

- Displaying graphics primitives (such as lines and circles) on a *raster scan* system involves the task of choosing which pixel to turn on and to which intensity.
- This process of digitizing a picture definition is called **scan conversion**.
- Some graphics controllers are capable of scan converting particular primitives, however in most low cost systems, scan converting has to be done by the software packages and/or libraries.
- This section deals with the implementation of relatively efficient scan converting algorithms of particular output primitives.

2.2 Line Drawing Algorithms

- The Cartesian equation of a line is:

$$y = mx + c$$

- Our aim is to scan convert a line segment usually defined by its endpoints: (x_a, y_a) and (x_b, y_b).
- Therefore, we need to draw the line segment:

$$y = mx + c$$

for $x_a \leq x \leq x_b$

where

$$\begin{aligned} m &= \frac{y_b - y_a}{x_b - x_a} \\ c &= y_a - x_a \frac{y_b - y_a}{x_b - x_a} \end{aligned}$$

- It is noted that for the particular case when $|m| = 1$, there will be exactly one pixel turned on in every column between x_a and x_b; and exactly one pixel turned on in every row between y_a and y_b.
- It is desireable that for $|m| < 1$, there is exactly one pixel in every column and at least one pixel in every row.
- Similarly, for $|m| > 1$, there will be exactly one pixel in every row and at least one pixel in each column.

2.3 A straightforward line drawing algorithm

The above points can be used to implement a rather straightforward line drawing algorithm. The slope and the intercept of a line are calculated from its endpoints. If $|m| \leq 1$, then the value of the x coordinate is incremented from $\min(x_a, x_b)$ to $\max(x_a, x_b)$, and the value of the y coordinate is calculated using the Cartesian line equation. Similarly, for $|m| > 1$, the value of the y coordinate is varied from $\min(y_a, y_b)$ to $\max(y_a, y_b)$ and the x coordinate is calculated (using $x = \frac{y-c}{m}$).

The following algorithm assumes that $0 \leq m \leq 1$ and $x_a < x_b$. The other cases can be considered by suitable reflections with the coordinate axes and looping through the y coordinates instead of the x coordinates.

```
m := (yb - ya) / (xb - xa);
c := ya - m * xa;

for x := xa to xb do
  begin
    y := round (m*x + c);
    PlotPixel (x, y)
  end;
```

However, this algorithm is extremely inefficient due to the excessive use of floating point arithmetic and the rounding function `round`.

2.4 The incremental DDA Algorithm

Given the line $y = mx + c$, the previous algorithm can be improved if we note that if the change in x is δx, then the change in the value of y is:

$$\begin{aligned} \delta y &= (m(x + \delta x) + c) - (mx + c) \\ &= mx + m\delta x + c - mx - c \\ &= m\delta x \end{aligned}$$

Hence, while looping through the x coordinate ($\delta x = 1$) we can deduce the new value of y by adding δy ($= m$) to the previous one, instead of calculating it using the Cartesian line equation each time. Also, the above result can be used to get:

$$\delta x = \frac{1}{m}\delta y$$

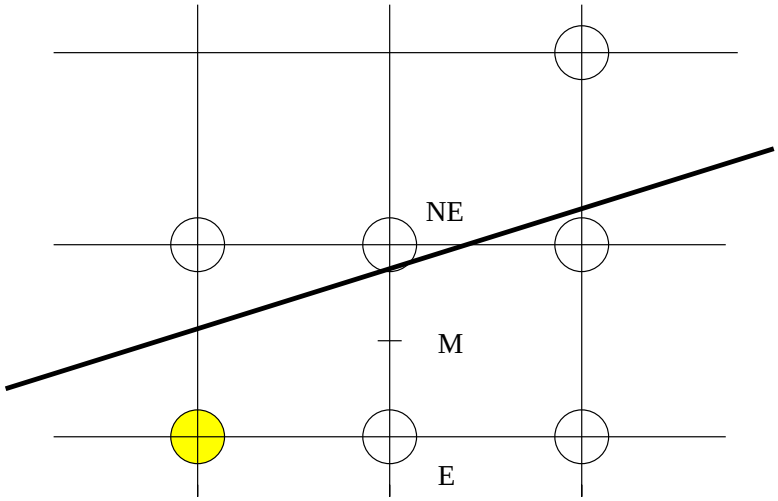


Figure 2.1: Checking whether the line is above the midpoint or not.

Therefore, while looping through the y coordinate, the new value of x can be achieved by adding $\frac{1}{m}$.

```
m := (yb - ya) / (xb - xa);
y := ya;

for x := xa to xb do
  begin
    PlotPixel (x, Round (y));
    y := y + m
  end;
```

- DDA stands for *Digital Differential Analyser*.
- The DDA algorithm is an improvement over the first one but it is still very inefficient.

2.5 The Midpoint Line Drawing Algorithm - Pitteway (1967)

Previously, we have shown that the difference between successive values of y in the line equation is $\delta y = m$. However, in practice, for $0 \leq m \leq 1$ the difference between the successive values of the y coordinate of the pixel plotted (`Round (y)`) is either 0 or 1. So, given that the previous pixel plotted is (x_p, y_p), then the next pixel $((x_{p+1}, y_{p+1}))$ is either $(x_p + 1, y_p)$ (let us call this point **E**, for East) or $(x_p + 1, y_p + 1)$ (**NE**).

Therefore, the incremental DDA algorithm can be modified so that we increment the value of the y coordinate only when necessary, instead of adding $\delta y = m$ in each step of the loop.

Given that the line equation is:

$$f(x) \quad = \quad mx + c$$

- **E** is plotted if the point **M** (the midpoint of **E** and **NE**) is above the line, or else

- **NE** is plotted if **M** is below the line.

Therefore, **E** is plotted if:

$$\begin{aligned} f(x_{p+1}) &\leq M_y \\ mx_{p+1} + c &\leq y_p + \frac{1}{2} \\ m(x_p + 1) + (y_a - mx_a) &\leq y_p + \frac{1}{2} \\ m(x_p - x_a + 1) &\leq y_p - y_a + \frac{1}{2} \end{aligned}$$

defining

$$\begin{aligned} \Delta x &= x_b - x_a \\ \Delta y &= y_b - y_a \end{aligned}$$

$$\begin{aligned} \frac{\Delta y}{\Delta x}(x_p - x_a + 1) &\leq y_p - y_a + \frac{1}{2} \\ \Delta y(x_p - x_a + 1) - \Delta x(y_p - y_a + \frac{1}{2}) &\leq 0 \\ 2\Delta y(x_p - x_a + 1) - \Delta x(2y_p - 2y_a + 1) &\leq 0 \end{aligned}$$

Now, let us define the left hand side of the above inequality by:

$$C(x_p, y_p) = 2\Delta y(x_p - x_a + 1) - \Delta x(2y_p - 2y_a + 1)$$

So for all p the point

- $(x_p + 1, y_p)$ is plotted if $C(x_p, y_p) \leq 0$, or else
- $(x_p + 1, y_p + 1)$ is plotted if $C(x_p, y_p) > 0$.

But how does the value of $C(x_p, y_p)$ depend on the previous one? If we choose **E**, then $(x_{p+1}, y_{p+1}) = (x_p + 1, y_p)$. Therefore,

$$\begin{aligned} C(x_{p+1}, y_{p+1}) &= C(x_p + 1, y_p) \\ &= 2\Delta y(x_p + 1 - x_a + 1) - \Delta x(2y_p - 2y_a + 1) \\ &= 2\Delta y(x_p + 1 - x_a) - \Delta x(2y_p - 2y_a + 1) + 2\Delta y \\ &= C(x_p, y_p) + 2\Delta y \end{aligned}$$

and, if we choose **NE**, then $(x_{p+1}, y_{p+1}) = (x_p + 1, y_p + 1)$. Therefore,

$$\begin{aligned} C(x_{p+1}, y_{p+1}) &= C(x_p + 1, y_p + 1) \\ &= 2\Delta y(x_p + 1 - x_a + 1) - \Delta x(2y_p + 2 - 2y_a + 1) \\ &= 2\Delta y(x_p + 1 - x_a) - \Delta x(2y_p - 2y_a + 1) + 2\Delta y - 2\Delta x \\ &= C(x_p, y_p) + 2(\Delta y - \Delta x) \end{aligned}$$

Moreover, the initial value of C is:

$$\begin{aligned} C(x_a, y_a) &= 2\Delta y(x_a - x_a + 1) - \Delta x(2 * y_a - 2 * y_a + 1) \\ &= 2\Delta y - \Delta x \end{aligned}$$

- Note that the value of C is always an integer.
- The value of y is always an integer.
- The value of C can be computed from the previous one by adding an integer value which does not depend on the x and y-coordinates of the current pixel.

```
Procedure MidpointLine (xa, ya, xb, yb: Integer);

  Var
    Dx, Dy, x, y: Integer;
    C, incE, incNE: Integer;

  Begin

    Dx := xb - xa;
    Dy := yb - ya;
    C := Dy * 2 - Dx;
    incE  := Dy * 2;
    incNE := (Dy - Dx) * 2;

    y := ya;
    for x := xa to xb - 1 do
      begin
        PlotPixel (x, y);
        If C <= 0 then
          C := C + incE
        else
          begin
            C := C + incNE;
            inc (y)
          end
      end;
    PlotPixel (xb, y)

  End;
```

2.6 The Bresenham Line Drawing Algorithm

The midpoint algorithm selects the next pixel by checking whether the line passes over or beneath the midpoint **M**. The Bresenham algorithm selects the pixel which is nearest to the line. It can be easily seen that these two selection criteria are equivalent, so that the Bresenham and the midpoint line drawing algorithms are basically identical.

2.7 The Double Step Midpoint Line Drawing Algorithm

Wu and Rokne (1987) proposed a midpoint line drawing algorithm that attempts to draw to pixels at a time instead of one. The Wu and Rokne double step algorithm draws one of four

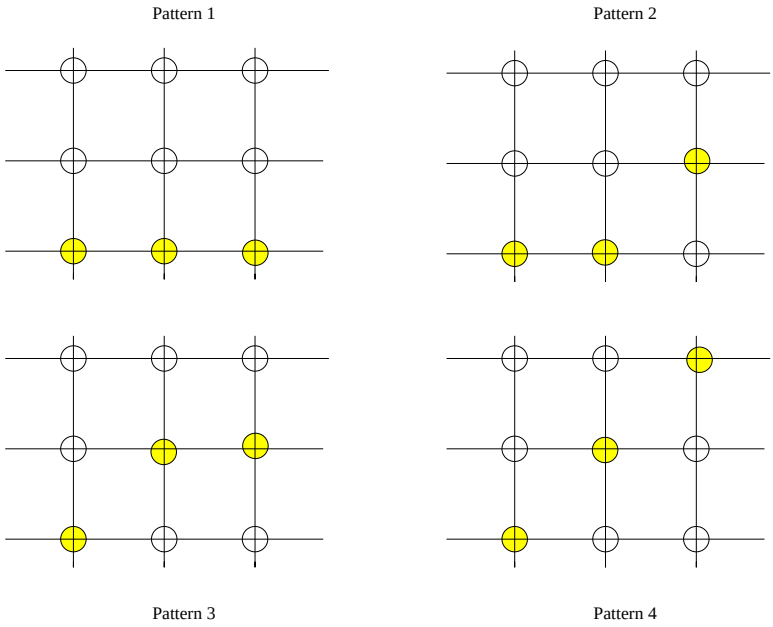


Figure 2.2: Double step midpoint line algorithm patterns.

possible pixel patterns. The four pixel patterns that can occur for a line with $0 \leq m \leq 1$ can be seen in Figure 2.2.

Wu showed that pattern 1 and pattern 4 cannot occur on the same line, and if the slope is less than $\frac{1}{2}$, patten 4 cannot occur (similarly for a slope greater than $\frac{1}{2}$, pattern 1 cannot occur). These properties mean that one of three patterns can be drawn for a given line, either 1,2,3 or 2,3,4. Assuming that the slope is between 0 and $\frac{1}{2}$, only patterns 1,2,3 need to be considered. It can be seen that pattern 1 is chosen if $C < 0$, and pattern 2 or 3 otherwise. Moreover, pattern 2 is chosen if $C < 2\Delta y$. Also,

$$\begin{array}{rcll} C_a & = & 4\Delta y - \Delta x & \\ C_{p+1} & = & C_p + 4\Delta y & \text{pattern 1 is chosen} \\ C_{p+1} & = & C_p + 4\Delta y - 2\Delta x & \text{either of pattern 2 or 3 are chosen} \end{array}$$

```
Procedure DoubleStep (xa, ya, xb, yb: Integer);

  Var
    Dx, Dy, x, y, current_x: Integer;
    C, inc_1, inc_2, cond: Integer;

  Begin

    Dx := xb - xa;
    Dy := yb - ya;
    C := Dy * 4 - Dx;
    inc_1  := Dy * 4;
    inc_2 := ((2 * Dy) - Dx) * 2;
    cond := 2 * Dy;
```

```
    current_x = xa;

    while (current_x < xb) do
        begin
            if C <= 0 then
                begin
                    draw_pixel_pattern(PATTERN_1, current_x);
                    C = C + inc_1;
                end;
            else
                begin
                    if (C < cond)
                        draw_pixel_pattern(PATTERN_2, current_x);
                    else draw_pixel_pattern(PATTERN_3, current_x);
                    C = C + inc_2;
                end;
            current_x = current_x + 2;
        end;

End;
```

The `draw_pixel_pattern()` routine needs to know the x position, because otherwise the line might be extended by a pixel. We note that all computations are done using integer additions. A simple form of antialiasing can be obtained, if instead of choosing between pattern 2 or 3 both are drawn at half-intensities. Wyvill (1990) demonstrates an algorithm that takes advantage of the symmetry about the midpoint to draw the line from two endpoints simultaneously, effectively doubling the algorithms speed.

2.8 Some points worth considering

Intensity of a line depending on the slope

It can be noted that lines with slopes near to ± 1 appear to be *fainter* than those with slopes near to 0 or ∞. This discrepancy can be solved by varying the intensity of the pixels according to the slope.

Antialiasing

Due to the discrete nature of the pixels, diagonal lines often appear *jaggied*. This *staircasing* effect is called **aliasing**, and techniques which try to solve, or at least minimize this problem are called **antialiasing**. For example, a line can be treated as a one-pixel thick rectangle and the intensity of the pixels is set according to a function of the area the rectangle covers over that particular pixel.

Endpoint Order

It must be ensured that a line drawn from (x_a, y_a) to (x_b, y_b) must be identical to the line drawn from (x_b, y_b) to (x_a, y_a). In the midpoint/Bresenham algorithm, the point **E** is chosen

when the selection criterion $C = 0$. (i.e. when the line passes through the midpoint of **NE** and **E**.) Thus it must be ensured that when the line is plotted from right to left, the point **SW** (not **W**) is selected when the criterion does not specify an obvious choice.

Clipping Lines

Suppose that the line segment $((x_a, y_a) \rightarrow (x_b, y_b))$ is clipped into $((x_c, y_c) \rightarrow (x_d, y_d))$. It must be ensured that the clipped line segment is identical to the relevant line segment of the otherwise unclipped line. This can be achieved by calculating the initial selection criterion $C(x_c, y_c)$ relative to the initial point (x_a, y_a) instead of considering (x_c, y_c) as the initial point.

Drawing Polylines

While drawing polylines, it must be ensured that the shared vertices are plotted only once.

Parallel Line Drawing Algorithms

On multiple processing systems, a line can be drawn concurrently by assigning a line segment to each processor.

2.9 Exercises

1. The line drawing routines discussed in this lecture make the assumption that the gradient m of the line being drawn lies in the range $0 \leq m < 1$. How would you generalize these algorithms to work for the cases where this assumption does not hold?

2. Implement the various line drawing algorithms described in this chapter using any graphics library of your choice.

Chapter 3

Graphics Primitives: Circle Drawing Algorithms

3.1 The circle equation

- The Cartesian equation of a circle with centre (x_c, y_c) and radius r is:

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

or,

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$

- It can be noted that at the 4 points $(x_c \pm r/\sqrt{2}, y_c \pm r/\sqrt{2})$, the gradient of the circle is ± 1.
- For $x_c - r/\sqrt{2} < x < x_c + r/\sqrt{2}$, the absolute value of the gradient is between 0 and 1.
- For $y_c - r/\sqrt{2} < y < y_c + r/\sqrt{2}$, the absolute value of the gradient is more than 1.

3.2 The Symmetry of a Circle

- Due to the eight way symmetry of a circle, it is necessary to calculate the positions of the pixels of only one octant.
- Up to eight points can thus be plotted simultaneously:

```
Procedure PlotCirclePoints (x, y: Integer);

  Begin

    PlotPixel (xc + x, yc + y);
    PlotPixel (xc + x, yc - y);
    If x <> 0 then
      begin
        PlotPixel (xc - x, yc + y);
```

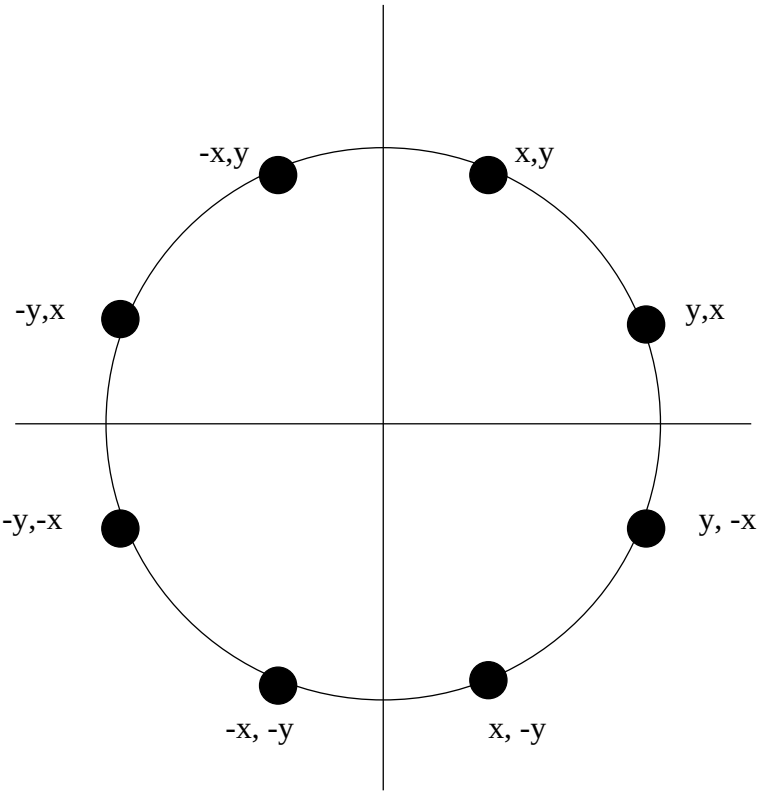


Figure 3.1: The 8-way symmetry of a circle.

```
      PlotPixel (xc - x, yc - y)
    end;
  If x <> y then
    begin
      PlotPixel (xc + y, yc + x);
      PlotPixel (xc - y, yc + x);
      If x <> 0 then
        begin
          PlotPixel (xc + y, yc - x);
          PlotPixel (xc - y, yc - x)
        end
    end

End;
```

- Note that care is taken so that no duplicate pixels are plotted.

3.3 A Straightforward Circle Drawing Algorithm

- The Cartesian circle equation can be used to implement a straightforward but very inefficient algorithm for scan converting circles.

```
For x := 0 to Round (r / Sqrt (2)) do
```

```
begin
  y := Round (Sqrt (r*r - x*x));
  PlotCirclePixels (x, y)
end;
```

- This algorithm is extremely inefficient due to excessive floating point computation.

3.4 Use of parametric polar form equation

The equation of a circle may be expressed in parametric polar form as follows:

$$\begin{aligned} x &= x_c + r\cos\theta \\ y &= y_c + r\sin\theta \end{aligned}$$

where r is the radius of the circle.

Additionally, the length of an arc δa corresponding to an angular displacement $\delta\theta$ is given by:

$$\delta a = r\delta\theta$$

To avoid gaps in our circle, we should use an increment value $\delta a = 1$, which corresponds to an increment value $\delta\theta = (1/r)$.

```
dtheta = 1/r;
theta = 0;
while (theta < PI / 4) do
  begin
    x := round(xc + r * cos(theta));
    y := round(yc + r * sin(theta))
    PlotCirclePixels (x, y)
    theta = theta + dtheta;
  end;
```

- This algorithm still makes excessive use of floating point arithmetic, and may still cause points to be plotted more than once.
- A value for $\delta\theta > (1/r)$ could be used to generate dotted circles.

3.5 The Midpoint Circle Drawing Algorithm

We have shown that we can devise a fast line drawing algorithm by selecting the pixels which are nearest to the real valued equation of the line. The same approach can be used to develop an algorithm for drawing circles.

Basically, given that the circle equation is:

$$f(x) = \sqrt{r^2 - x^2}$$

and that (x_p, y_p) is the previous pixel plotted, then we choose from the two candidate pixels **E** and **SE** by considering whether the boundary of the circle passes above or below the midpoint **M**. Where **E** is the point $(x_p + 1, y_p)$, and **SE** is the point $(x_p + 1, y_p - 1)$.

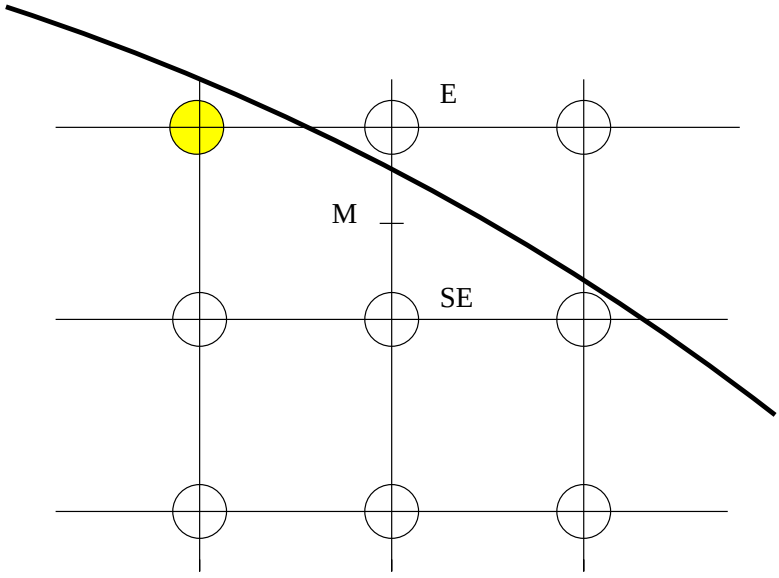


Figure 3.2: The midpoint circle drawing algorithm.

The pixel **E** is selected if **M** lies below the circle, that is, if:

$$\begin{aligned} M_y &< f(x_{p+1}) \\ y_p - \frac{1}{2} &< \sqrt{r^2 - (x_{p+1})^2} \\ (y_p - \frac{1}{2})^2 &< r^2 - (x_p + 1)^2 \\ (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - r^2 &< 0 \end{aligned}$$

Now, defining:

$$C(x_p, y_p) = (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - r^2$$

For all p, the point

- **E** $= (x_p + 1, y_p)$ is selected if $C(x_p, y_p) < 0$, or else
- **SE** $= (x_p + 1, y_p - 1)$ is selected if $C(x_p, y_p) \geq 0$.

Now, if **E** is chosen, the new value of C is:

$$\begin{aligned} C(x_{p+1}, y_{p+1}) &= C(x_p + 1, y_p) \\ &= (x_p + 1 + 1)^2 + (y_p - \frac{1}{2})^2 - r^2 \\ &= (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - r^2 + 2(x_p + 1) + 1 \\ &= C(x_p, y_p) + 2x_{p+1} + 1 \end{aligned}$$

And if **SE** is chosen,

$$C(x_{p+1}, y_{p+1}) = C(x_p + 1, y_p - 1)$$

$$
\begin{aligned}
&= (x_p + 1 + 1)^2 + (y_p - \frac{1}{2} - 1)^2 - r^2 \\
&= (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - r^2 + 2(x_p + 1) + 1 - 2(y_p - \frac{1}{2}) + 1 \\
&= C(x_p, y_p) + 2(x_{p+1} - y_{p+1}) + 1
\end{aligned}
$$

The initial value of C is:

$$
\begin{aligned}
C(0, r) &= (1)^2 + (r - \frac{1}{2})^2 - r^2 \\
&= 1 + r^2 - r + \frac{1}{4} - r^2 \\
&= \frac{5}{4} - r \\
&= (1 - r) + \frac{1}{4}
\end{aligned}
$$

The initial value of C is $\frac{1}{4}$ greater than an integer. In each step, C is added to an integer value, thus the value of C is always $\frac{1}{4}$ greater than that of an integer. However, if we define $C' = C - \frac{1}{4}$, so that C' has always an integer value, it can be seen that $C < 0 \Leftrightarrow C' < 0$. Therefore, we can use the criterion C' (which requires only integer arithmetic to compute) instead of C.

```
Procedure MidPointCircle (xc, yc, r: Integer);

  Procedure PlotCirclePoints (x, y: Integer);
    Begin
      (* Refer to previous algorithm *)
    End;

  Var
    x, y, C: Integer;

  Begin

    x := 0;
    y := r;
    C := 1 - r;
    PlotCirclePoints (x, y);

    While (x < y) do
      begin
        if C < 0 then
          begin
            Inc (x);
            C := C + x * 2 + 1
          end
        else
          begin
            Inc (x);
```

```
          Dec (y);
          C := C + (x - y) * 2 + 1
        end;
      PlotCirclePoints (x, y)
    end

End;
```

- The midpoint algorithm uses only integer addition, subtraction and multiplication.
- The difference in the selection criterion depends on the coordinates of the previous pixel.
- However, since the first difference of the selection criterion is a linear function, it can be reduced to a constant if second order differences are applied.

3.5.1 Second Order Differences

It is noted that the first order differences of C depend on the current pixel coordinates:

$$\begin{aligned} incE(p) &= \Delta_E C(p) \\ &= C(x_{p+1}, y_{p+1}) - C(x_p, y_p) \\ &= ((x_p + 1 + 1)^2 + (y_p - \frac{1}{2})^2 - r^2) - (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - r^2) \\ &= 2x_p + 3 \end{aligned}$$

$$\begin{aligned} incSE(p) &= \Delta_{SE} C(p) \\ &= C(x_{p+1}, y_{p+1}) - C(x_p, y_p) \\ &= ((x_p + 1 + 1)^2 - (y_p - 1 - \frac{1}{2})^2 - r^2) - ((x_p + 1)^2 + (y_p - \frac{1}{2})^2 - r^2) \\ &= 2(x_p - y_p) + 5 \end{aligned}$$

Where Δ_E is the first difference given that pixel **E** is chosen, and Δ_{SE} is the first difference given that pixel **SE** is chosen.

Now, the second order differences are:

$$\begin{aligned} \Delta_E incE(p) &= incE(p+1) - incE(p) \\ &= (2(x_p + 1) + 3) - (2x_p + 3) \\ &= 2 \end{aligned}$$

$$\begin{aligned} \Delta_E incSE(p) &= incSE(p+1) - incSE(p) \\ &= (2(x_{p+1} - y_{p+1}) + 5) - (2(x_p - y_p) + 5) \\ &= (2(x_p + 1 - y_p) + 5) - (2(x_p - y_p) + 5) \\ &= 2 \end{aligned}$$

So, if **E** is chosen, 2 is added to $incE(p)$ and 2 is added to $incSE(p)$ to obtain the new values of the first differences.

$$\Delta_{SE} incE(p) = incE(p+1) - incE(p)$$

$$
\begin{aligned}
&= (2(x_p + 1) + 3) - (2x_p - 3) \\
&= 2
\end{aligned}
$$

$$
\begin{aligned}
\Delta_{SE} incSE(p) &= incSE(p+1) - incSE(p) \\
&= (2(x_{p+1} - y_{p+1}) + 5) - (2(x_p - y_p) + 5) \\
&= (2(x_p + 1 - y_p + 1) + 5) - (2(x_p - y_p) + 5) \\
&= 4
\end{aligned}
$$

So, if **SE** is chosen, 2 is added to $incE(p)$ and 4 is added to $incSE(p)$ to obtain the new values of the first differences.

Now, the initial values of $incE$ and $incSE$ are:

$$
\begin{aligned}
incE(0) &= 2x_0 + 3 \\
&= 3
\end{aligned}
$$

$$
\begin{aligned}
incSE(0) &= 2(x_0 - y_0) + 5 \\
&= 5 - 2r
\end{aligned}
$$

The optimized midpoint circle scan conversion algorithm thus follows:

```
Procedure MidPointCircle (xc, yc, r: Integer);

  Procedure PlotCirclePoints (x, y: Integer);
    Begin
      (* Refer to previous algorithm *)
    End;

  Var
    x, y, C: Integer;
    incE, incSE: Integer;

  Begin

    x := 0;
    y := r;
    C := 1 - r;
    incE := 3;
    incSE := 5 - r * 2;
    PlotCirclePoints (x, y);

    While (x < y) do
      begin
        if (C < 0) then
          begin
            C := C + incE;
            incE := incE + 2;
            incSE := incSE + 2;
```

```
          Inc (x)
        end
      else
        begin
          C := C + incSE;
          incE := incE + 2;
          incSE := incSE + 4;
          Inc (x);
          Dec (y)
        end;
    PlotCirclePoints (x, y)
  end

End;
```

- This algorithm is faster than the original midpoint algorithm because it avoids integer multiplication.

3.6 Ellipses

The cartesian equation of an ellipse with centre (x_c, y_c), semimajor axis r_1, and semiminor axis r_2 is:

$$\left(\frac{x - x_c}{r_1}\right)^2 + \left(\frac{y - y_c}{r_2}\right)^2 = 1$$

or,

$$y^2 = {r_2}^2\left(1 - \frac{x^2}{{r_1}^2}\right)$$

or in polar parametric form,

$$\begin{aligned} x &= x_c + r_1 \cos\theta \\ y &= y_c + r_2 \sin\theta \end{aligned}$$

- Circle drawing techniques can be adapted to draw ellipses. Note, however, that ellipses in general do not possess 8-way symmetry, but only 4-way symmetry.
- When using the mid-point and Bresenham techniques, it cannot be assumed that the same two choices for the next point are valid throughout; in previous cases, these choices were only adopted after taking the gradient into consideration. In the case of an ellipse quadrant, the approach has to be changed as the gradient crosses the $m = 1$ boundary.

3.7 Exercises

1. Implement the various circle algorithms.
2. Derive a mid-point algorithm suitable for drawing ellipses.

Chapter 4

Graphics primitives: Filled Areas

4.1 Filling polygons

4.1.1 Inside-outside test

An important issue that arises when filling polygons is that of deciding whether a particular point is interior or exterior to a polygon. A rule called the **odd-parity** (or the odd-even rule) is usually applied to test whether a point is interior or not. A half-line starting from the particular point and extending to infinity is drawn in any direction such that no polygon vertex intersects with the line. The point is considered to be interior if the number of intersections between the line and the polygon edges is odd.

4.2 The scan-line polygon filling algorithm

The scan-line polygon filling algorithm involves the horizontal scanning of the polygon from its lowermost to its topmost vertex, identifying which edges intersect the scan-line, and finally drawing the interior horizontal lines. The algorithm is specified by the following 3 steps:

For each horizontal scan-line:

1. List all the points that intersect with the horizontal scan-line.
2. Sort the intersection points in ascending order of the x coordinate.

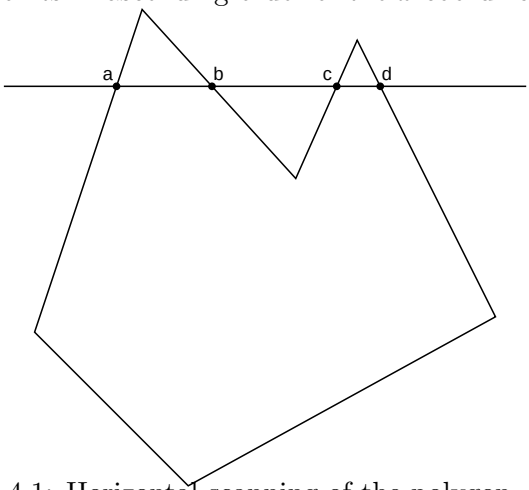


Figure 4.1: Horizontal scanning of the polygon.

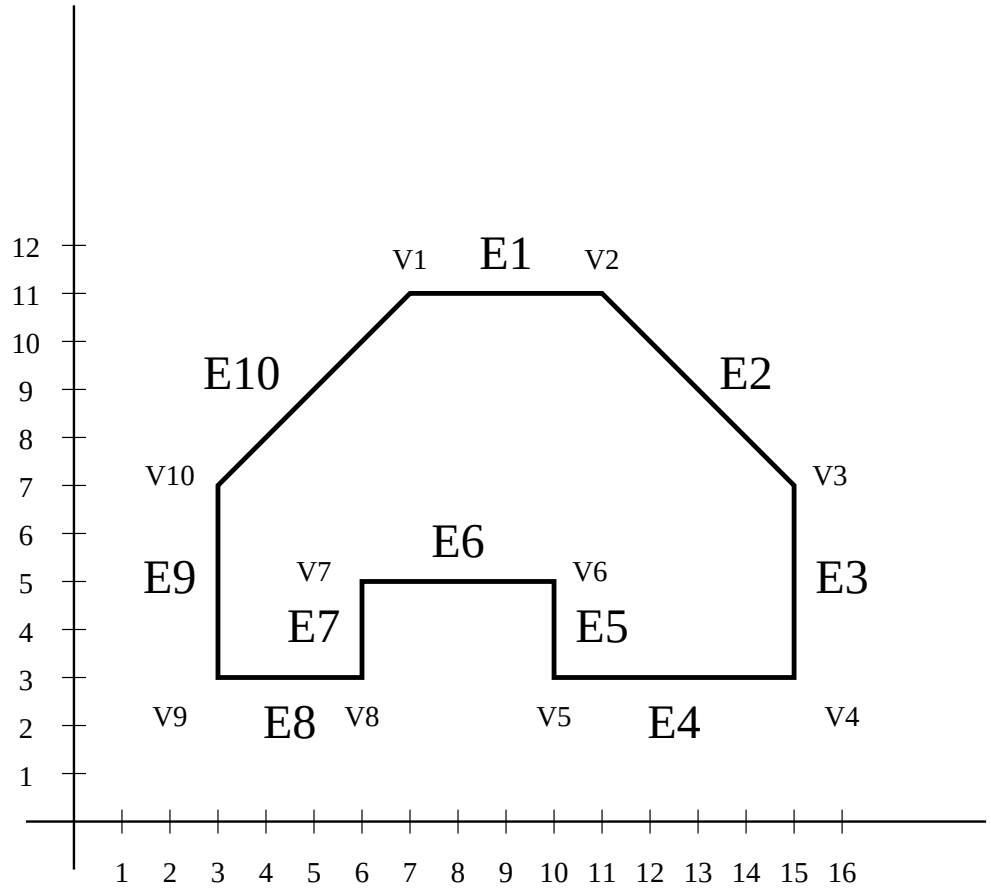


Figure 4.2: A non-simple polygon.

3. Fill in all the interior pixels between pairs of successive intersections.

The third step accepts a sorted list of points and connects them according to the odd-parity rule. For example, given the list $[p_1; p_2; p_3; p_4; \ldots; p_{2n-1}; p_{2n}]$, it draws the lines $p_1 \rightarrow p_2; p_3 \rightarrow p_4; \ldots; p_{2n-1} \rightarrow p_{2n}$. A decision must be taken as to whether the edges should be displayed or not: given that $p_1 = (x_1, y)$ and $p_2 = (x_2, y)$, should we display the line (x_1, y, x_2, y) or just the interior points $(x_1 + 1, y, x_2 - 1, y)$?

- Step 1 can be optimized by making use of a sorted edge table. Entries in the edge table are sorted on the basis of their lower y value. Next, edges sharing the same low y value are sorted on the basis of their higher y value. A pair of markers are used to denote the range of 'active' edges in the table that need to be considered for a particular scan-line. This range starts at the top of the table, and moves progressively downwards as higher scan-lines are processed.

- Given a scan-line $y = s$, and a non-horizontal edge with end-points $(x_1, y_1), (x_2, y_2), y_1 < y_2$, an intersection between the two exists if $y_1 \leq y \leq y_2$. The point of intersection is $(\frac{s-c}{m}, s)$ where $m = \frac{y_2 - y_1}{x_2 - x_1}$ and $c = y_1 - mx_1$.

4.2.1 An example

Consider the polygon in Figure 4.2. The edge table for such a polygon would be:

Edge	Y_{min}	Y_{max}	X of Y_{min}	X of Y_{max}	$\frac{1}{m}$
E1	11	11	6	11	0
E2	7	11	15	11	-1
E3	3	7	15	15	0
E4	3	3	10	15	-
E5	3	5	10	10	0
E6	5	5	10	6	-
E7	3	5	6	6	0
E8	3	3	3	6	-
E9	3	7	3	3	0
E10	7	11	3	7	1

The edge list for such a polygon, for each of the scan-lines from 3 to 11 is:

Scan-line	Edge number
11	-
10	-
9	-
8	-
7	2, 10
6	-
5	-
4	-
3	3, 5, 7, 9

Note that in the above table the horizontal lines are not added to the edge list. The reason for this is discussed below. The active edges for scan-line 3 would be 3, 5, 7, 9, these are sorted in order of their x values, in this case 9, 7, 5, 3. The polygon fill routine would proceed to fill the intersections between (3,3) (E9) and (6,3) (E7) and (10,3) (E5) to (15,3) (E3). The next scan-line (4) is calculated in the same manner. In this the values of x do not change (since the line is vertical; it is incremented by 0). The active edge at scan-line 7 are 10 and 2 (correct order).

4.2.2 Dealing with vertices

Dealing with vertices is not trivial since the odd-parity rule assumes that the lines used to check whether a point is interior does not intersect with a vertex. Should a vertex be considered as a single edge, two separate edges or none at all?

Consider the polygon in figure 4.3. It seems obvious that vertices `C` and `G` should be considered as two separate intersections (or none at all). However, vertices `A` and `E` should be considered as a single one!

The property that distinguishes `C` and `G` from `A` and `E` is that, in the former case, the lines sharing the edge are on the same side of the half-plane defined by the scan line, while in the case of `A` and `E`, the lines are at opposite sides. This criterion can be checked before considering whether each particular vertex is to be treated as one intersection or as two.

A more efficient solution is to adjust vertices so that they do not overlap whenever they should be treated as one intersection. One possible scheme involves traversing the edges in a clockwise fashion and increasing or decreasing the y value of an end-point by 1 (depending on whether y is decreasing or increasing), and calculating the corresponding x value (see figure 4.4).

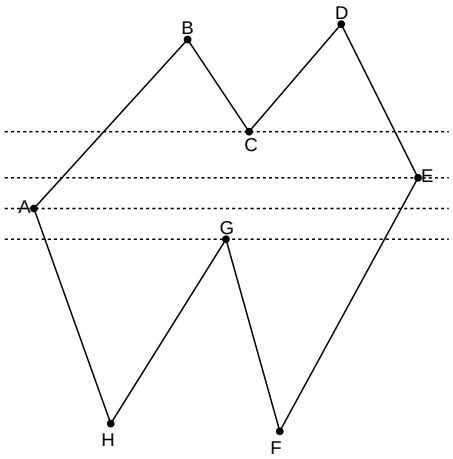


Figure 4.3: Considering polygon vertices.

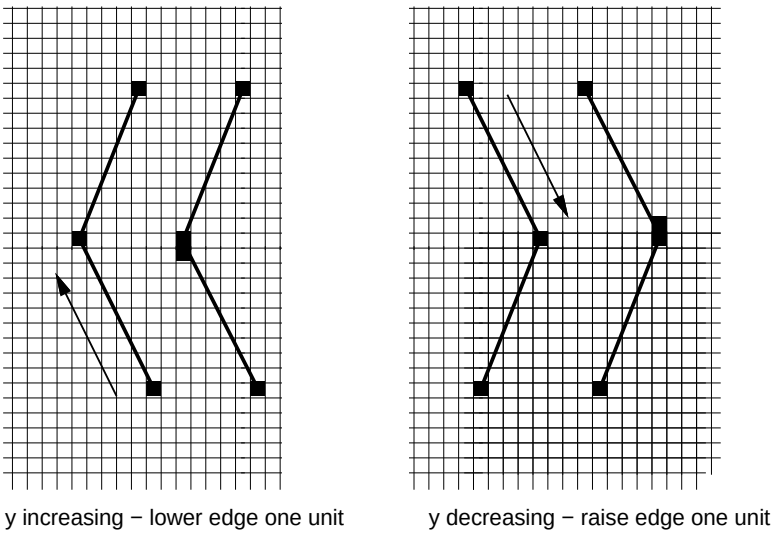


Figure 4.4: Shortening edges.

4.2.3 Horizontal edges

If the polygon boundary is to be included in the fill, horizontal edges can be handled simply by leaving them out of the edge table. It is important, however, that the same checks applied at individual vertices are applied to the end-points at each end of a horizontal line:

- if the edges on either side of the horizontal edge are in the same half plane defined by the horizontal line, then no additional action is required;
- otherwise one of the neighbouring edges must be shortened by one pixel. Determining which edge to shorten is tricky. Two possible approaches are:
  - choose any edge for shortening: this may leave the edge undrawn, so all such horizontal edges should be (re-)drawn seperately;
  - follow round the edges in a clockwise manner, and shorten that edge which first meets the horizontal line: this ensures that all edges are drawn whilst filling. If the other edge is chosen for shortening, then horizontal lines will not be drawn.

Choosing which edge to shorten when dealing with horizontal lineshorchoice

4.3 General-purpose filling algorithms

4.3.1 Boundary-fill algorithm

This makes use of coherence properties of the boundary of a primitive/figure: given a point inside the region the algorithm recursively plots the surrounding pixels until the primitive boundary is reached.

Given the `FillColour`, the `BoundaryColour` and a point inside the boundary, the following algorithm recursively sets the four adjacent pixels (2 horizontal and 2 vertical) to the `FillColour`.

```
Procedure BoundaryFill (x, y: Integer;
    FillColour, BoundaryColour: ColourType);

  Procedure BFill (x, y: Integer);
    Var
      CurrentColour: ColourType;
    Begin
      CurrentColour := GetPixel (x, y);
      If (CurrentColour <> FillColour) and
         (CurrentColour <> BoundaryColour)
      then
          SetPixel (x, y, FillColour);
          BFill (x + 1, y);
          BFill (x - 1, y);
          BFill (x, y + 1);
          BFill (x, y - 1)
    End;

  Begin
    BFill (x, y)
  End;
```

- Regions which can be completely filled with this algorithm are called 4-connected regions.
- Some regions cannot be filled using this algorithm. Such regions are called 8-connected and algorithms filling such areas consider the four diagonally adjacent pixels as well as the horizontal and vertical ones.

```
Procedure BFill8 (x, y: Integer);
  Var
    CurrentColour: ColourType;
  Begin
    CurrentColour := GetPixel (x, y);
    If (CurrentColour <> FillColour) and
       (CurrentColour <> BoundaryColour)
    then
        SetPixel (x, y, FillColour);
        BFill8 (x + 1, y);
        BFill8 (x - 1, y);
        BFill8 (x, y + 1);
        BFill8 (x, y - 1);
        BFill8 (x + 1, y + 1);
        BFill8 (x + 1, y - 1);
        BFill8 (x - 1, y + 1);
        BFill8 (x - 1, y - 1)
  End;
```

- Care must be taken to ensure that the boundary does not contain holes, which will cause the fill to 'leak'. The 8-connected algorithm is particularly vulnerable.

4.3.2 Flood-Fill Algorithm

The flood-fill algorithm is used to fill a region which has the same colour and whose boundary may have more than one colour.

```
Procedure FloodFill (x, y: Integer;
    FillColour, OldColour: ColourType);

  Procedure FFill4 (x, y): Integer;

    Begin
      if GetPixel (x, y) = OldColour then
          SetPixel (x, y, FillColour);
          FFill4 (x + 1, y);
          FFill4 (x - 1, y);
          FFill4 (x, y + 1);
          FFill4 (x, y - 1)
    End;

  Begin
    FFill4 (x, y)
  End;
```

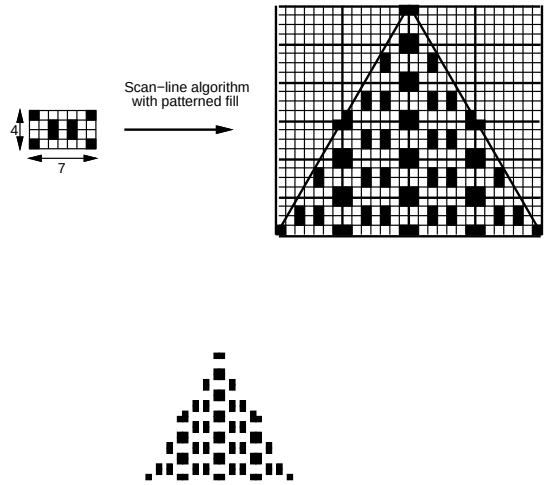


Figure 4.5: Scan-line algorithm with pattern.

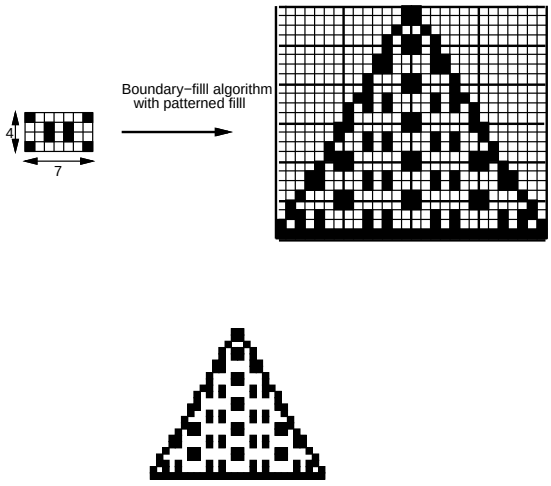


Figure 4.6: Bounary-fill (4-connect) algorithm with pattern.

4.4 Pattern Filling

Any of the filling algorithms discussed thus far can be modified to use a given pattern when filling. Figures 4.5 and 4.6 show the effects of using this with the scan-line and boundary fill algorithms respectively. Patterned fills can be achieved by changing `PlotPixel (x, y)` statements into `SetPixel (x, y, pixmap[x mod m, y mod n])`, where `pixmap` an m by n matrix that defines the fill pattern.

4.5 Exercises

1. Derive an algorithm for filling in circles and ellipses.
2. Implement the scan-line polygon filling algorithms for both simple and non-simple polygons.
3. Adapt the scan-line polygon filling algorithm from 1 and optimise it to cater for simple polygons only.
4. Implement the 4-way and 8-way boundary filling algorithms.

Chapter 5

Clipping in 2D

Usually only a specified region of a picture needs to be displayed. This region is called the **clip window**. An algorithm which displays only those primitives (or part of the primitives) which lie inside a clip window is referred to as a **clipping algorithm**. This lecture describes a few clipping algorithms for particular primitives.

5.1 Clipping Points

- A point (x, y) lies inside the rectangular clip window $(x_{min}, y_{min}, x_{max}, y_{max})$ if and only if the following inequalities hold:

$$\begin{array}{rcrcl} x_{min} & \leq & x & \leq & x_{max} \\ y_{min} & \leq & y & \leq & y_{max} \end{array}$$

5.2 The Cohen-Sutherland algorithm for line clipping

Clipping a straight line against a rectangular clip window results in either:

1. a line segment whose endpoints may be different from the original ones, or
2. not displaying any part of the line. This occurs if the line lies completely outside the clip window.

The Cohen-Sutherland line clipping algorithm considers each endpoint at a time and truncates the line with the clip window's edges until the line can be **trivially accepted** or **trivially rejected**. A line is trivially rejected if both endpoints lie on the same outside half-plane of some clipping edge.

The xy plane is partitioned into nine segments by extending the clip window's edges (figure 5.1). Each segment is assigned a 4-bit code according to where it lies with respect to the clip window:

Bit	Side	Inequality
1	N	$y > y_{max}$
2	S	$y < y_{min}$
3	E	$x > x_{max}$
4	W	$x < x_{min}$

For example bit 2 is set if and only if the region lies to the south of (below) the clip window.

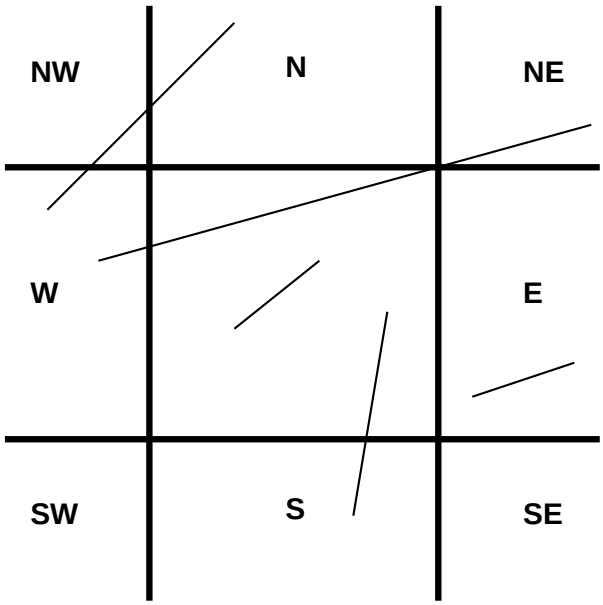


Figure 5.1: Partition of plane into 9 segments.

The Cohen-Sutherland algorithm starts by assigning an outcode to the two line endpoints (say c_1 and c_2).

- If both outcodes are 0 (c_1 OR $c_2 = 0$) the line lies entirely in the clip window and is thus trivially accepted.
- If the two outcodes have at least one bit in common (c_1 AND $c_2 \neq 0$), then they lie on the same side of the window and the line is trivially rejected.
- If a line cannot be accepted or rejected, it is then truncated by an intersecting clip edge and the previous steps are repeated.

```
Function CSClip (var x1, y1, x2, y2: Real; xmin, ymin, xmax, ymax:
Real): Boolean;
    Const
        North = 8; South = 4; East = 2; West = 1;

    Function OutCode (x, y: Real): Integer;
        Var
            c: Integer;
        Begin
            c := 0;
            if y > ymax then
                c := North
            else if y < ymin then
                c := South;
            if x > xmax then
                c := c OR East
            else if x < xmin then
```

```
            c := c OR West;
        OutCode := c
    End;

Var
    Accept, Done: Boolean;
    c1, c2, c: Integer;
    x, y: Real;
Begin
    Accept := False; Done := False;
    c1 := OutCode (x1, y1); c2 := OutCode (x2, y2);
    repeat
        if (c1 OR c2) = 0 then
            Accept := True; Done := True
        else if (c1 AND c2) <> 0 then
            Done := True
        else
            if c1 <> 0 then
                c := c1; x := x1; y := y1;
            else
                c := c2; x := x2; y := y2

            if (c AND North) <> 0 then
                x := x1 + (x2 - x1) * (ymax - y1) / (y2 - y1);
                y := ymax
            else if (c AND South) <> 0 then
                x := x1 + (x2 - x1) * (ymin - y1) / (y2 - y1);
                y := ymin
            else if (c AND East) <> 0 then
                x := xmax;
                y := y1 + (y2 - y1) * (xmax - x1) / (x2 - x1)
            else (* West *)
                x := xmin;
                y := y1 + (y2 - y1) * (xmin - x1) / (x2 - x1)

            if c = c1 then
                c1 := OutCode (x, y); x1 := x; y1 := y
            else
                c2 := OutCode (x, y); x2 := x; y2 := y
    Until Done;
    CSClip := Accept;
End;
```

5.3 Sutherland-Hodgman polygon clipping algorithm

A polygon can be clipped against a rectangular clip window by considering each clip edge at a time. For filled polygons, a new polygon boundary must be computed (figure 5.2). At each clipping stage, a new polygon is created by removing the outside vertices and inserting

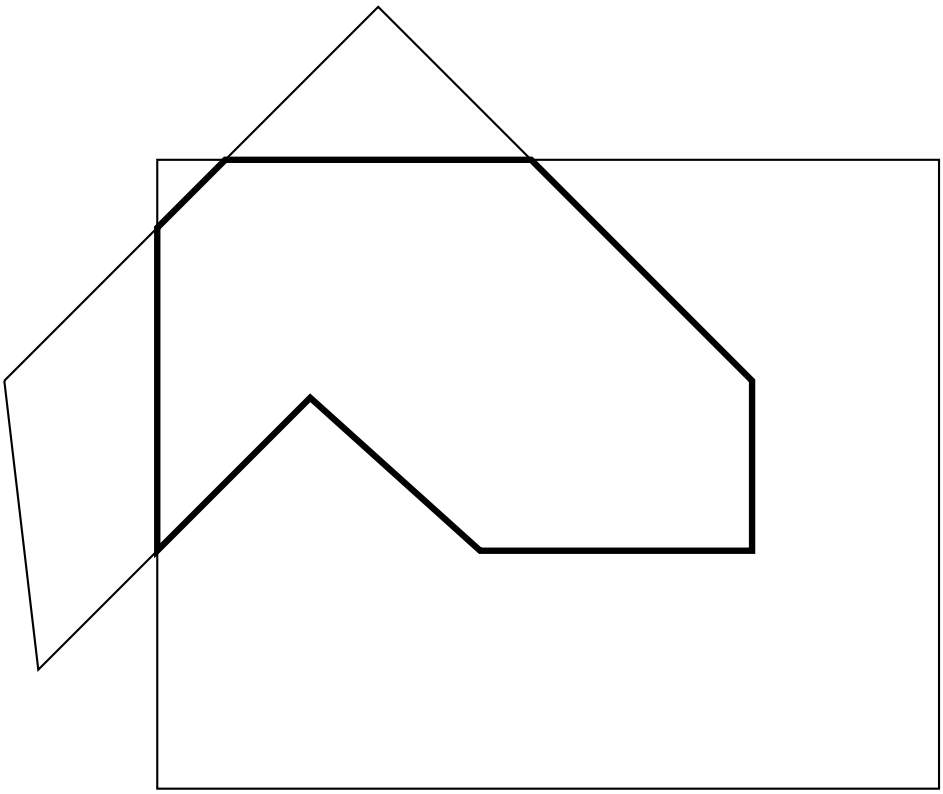


Figure 5.2: Polygon clipping.

the intersections with the clip boundary.

5.3.1 Clipping against an edge

Given the polygon $[v_1; v_2; \ldots; v_n]$, we can create a new polygon $[w_1; w_2; \ldots; w_m]$ by considering the vertices in sequence and deciding which ones (and any intersections with the edge) have to be inserted into the clipped polygon vertex list.

Suppose that v_i has been processed in the previous step. There are four possible cases which need to be considered while processing the next vertex v_{i+1}.

1. If v_i is outside the window and v_{i+1} is inside, then the intersection of the line $v_i \rightarrow v_{i+1}$ with the clipping edge, and the vertex v_{i+1} have to be inserted into the new polygon list.

2. If v_i and v_{i+1} are both outside the clip window then no point need be inserted.

3. If both v_i and v_{i+1} are inside the window, v_{i+1} is inserted into the list, and finally

4. If v_i is inside the window and v_{i+1} is outside, only the intersection of the line $v_i \rightarrow v_{i+1}$ with the clip boundary is inserted.

```
Type
    EdgeType = (Top, Left, Bottom, Right);
    VertexType = Record x, y: Real; End;
```

```
    PolygonType = Record
                      Vertnum: Integer;
                      Vertex: Array[1..MAX] of VertexType;
                  End;

Procedure CHPolyClip (Pin: PolygonType; Var Pout: PolygonType;
xmin, ymin, xmax, ymax: Integer);
    Function Inside (v: VertexType; e: EdgeType): Boolean;
        Begin
            Case e of
                Top:      Inside := v.y <= ymax;
                Left:     Inside := v.x >= xmin;
                Bottom:   Inside := v.y >= ymin;
                Right:    Inside := v.x <= xmax
        End;

    Procedure Intersect (v1, v2: VertexType; e: EdgeType; var v: VertexType);
        Begin
            Case e of
                Top:     v.x := v1.x + (ymax - v1.y) *
                            (v2.x - v1.x) / (v2.y - v1.y);
                         v.y := ymax
                Left:    v.x := xmin;
                         v.y := v1.y + (xmin - v1.x) *
                            (v2.y - v1.y) / (v2.x - v1.x)
                Bottom:  v.x := v1.x + (ymin - v1.y) *
                            (v2.x - v1.x) / (v2.y - v1.y);
                         v.y := ymin
                Right:   v.x := xmax;
                         v.y := v1.y + (xmax - v1.x) *
                            (v2.y - v1.y) / (v2.x - v1.x)
        End;

    Procedure ClipEdge (P1: PolygonType; Var P2: PolygonType; e: EdgeType);
        Procedure Insert (v: VertexType);
            Begin
                inc (P2.Vertnum); P2.Vertex [P2.Vertnum] := v
            End
        Var
            v1, v2, i: VertexType; n: Integer;
        Begin
            P2.VertNum := 0;
            if P1.VertNum <> 0 then
                v1 := P1.Vertex[P1.VertNum];
                For n := 1 to P1.VertNum do
                    v2 := P1.Vertex[n];
                    if Inside (v2, e) then
                        if Inside (v1, e) then
```

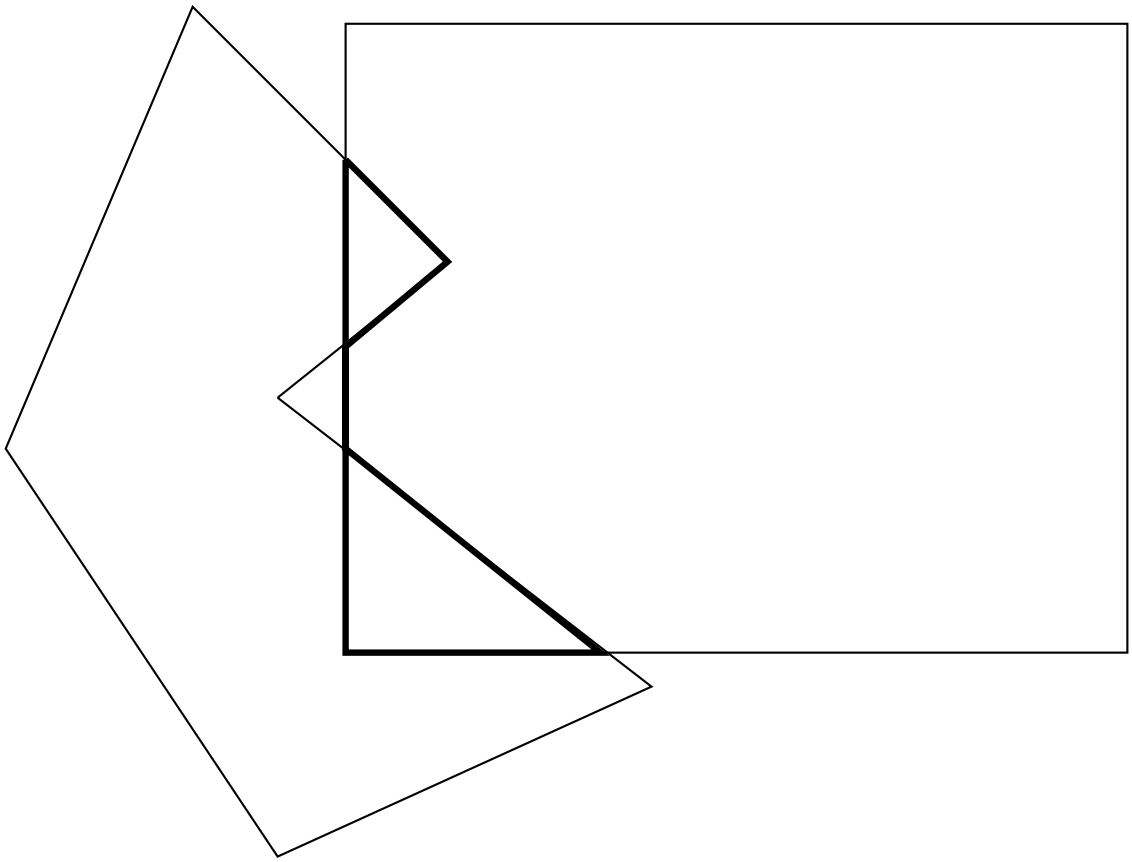


Figure 5.3: Problems with clipping a concave shape.

```
                Insert (v2)
            else
                Intersect (v1, v2, e, i);
                Insert (i);
                Insert (v2)
        else if Inside (v1, e) then
            Intersect (v1, v2, e, i);
            Insert (i)
        v1 := v2
    End;
Var
    e: EdgeType;
Begin
    For e := Top to Right do ClipEdge (Pin, Pin, e);
    Pout := Pin
End;
```

- The Sutherland-Hodgman algorithm fails on certain concave polygons. In such cases the polygon must be split into two convex ones. See figure 5.3.

Chapter 6

Geometrical transformations

6.1 Preliminaries

6.1.1 Vectors

A *vector* is an n-tuple of real numbers, where $n = 2$ for 2D space, $n = 3$ for 3D space etc. We will denote vectors using small italicized letters, and represent them as columns e.g. $v = \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix}$.

Vectors may be added:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x + x' \\ y + y' \\ z + z' \end{bmatrix}$$

Vectors may be multipled by real numbers (scalar multiplication):

$$s. \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} sx \\ sy \\ sz \end{bmatrix}$$

A *linear combination* of the vectors $v_1, v_2, \ldots, v_n$, is any vector of the form $\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$.

Dot product and applications

Dot product:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} . \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = xx' + yy' + zz'$$

Properties of dot product:

- symmetric: $v.w = w.v$

- nondegenerate: $v.v = 0$ iff $v = 0$
- bilinear: $v.(u + \alpha w) = v.u + \alpha(v.w)$

Uses of dot product:

- *magnitude* (length) of a vector: $\|v\| = \sqrt{v.v}$
- *normalization* and *unit vector*: we can normalize a vector v to obtain a unit vector $\bar{v}$ as follows: $\bar{v} = v/\|v\|$. A unit vector is one whose magnitude is 1.
- *angle between two vectors*: given two vectors, v and w, the angle between them θ is given by $\theta = cos^{-1}\left(\frac{v.w)}{\|v\|.\|w\|}\right)$. If v and w are unit vectors then the division is unnecessary.
- *projection*: given a unit vector $\bar{v}$ and a vector w, the projection u of w in the direction of $\bar{v}$ is given by $\bar{v}.w$.

6.1.2 Matrices

A matrix is a rectangular array of real numbers which can be used to represent operations (called *transformations*) on vectors. A vector in n-space is effectively an $nx1$ matrix, or a *column matrix*. Non-column matrices will be denoted using capitalized italics e.g.

$$A = \begin{bmatrix} 3 & 0 & 1 \\ 2 & 0 & 1 \\ 0 & 3 & 1 \end{bmatrix}$$

Multiplication: if A is an nxm matrix with entries a_{ij} and B is an mxp matrix with entries b_{ij}, then AB is defined as an nxp matrix with entries c_{ij} where each $c_{ij} = \sum_{s=1}^{m} a_{is}b_{sj}$.

Identity matrix:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Determinant (of a square matrix)

$$detA = \sum_{i=1}^{n}(-1)^{1+i}A_{1i}$$

where A_{ij} denotes the determinant of the $(n-l)x(n-1)$ matrix obtained by removing the ith row and jth column of A.

Cross product of two vectors

Given $v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}, w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$ then

$$vXw = det\begin{bmatrix} i & j & k \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{bmatrix}$$

$$
\begin{aligned}
&= (v_2w_3 - v_3w_2)i + (v_3w_1 - v_1w_3)j \\
&\quad +(v_1w_2 - v_2w_1)k
\end{aligned}
$$

where i, j, k represent unit vectors directed along the three coordinate axes i.e.

$$
vXw = \begin{bmatrix} v_2w_3 - v_3w_2 \\ v_3w_1 - v_1w_3 \\ v_1w_2 - v_2w_1 \end{bmatrix}
$$

The cross product uXv of the two vectors u and v is a vector with the following properties:

- it is perpendicular to the plane containing u and v,
- $\|uXv\| = \|u\|.\|v\|.|sin\theta|$, where θ is the angle between u and v.

Transpose

The transpose of an mxn matrix is an nxm matrix obtained by flipping the matrix along its diagonal. Denote the tranpose of a matrix A by A^T.

Inverse

A multiplicative inverse matrix is defined only for square matrices whose determinants are non-zero. The inverse of a matrix A is denoted by A^{-1}, and has the property $AA^{-1} = I$. The inverse of a matrix may be computed using a technique known as *Gaussian elimination.*

6.2 2D transformations

6.2.1 Translation

The point (x, y) is translated to the point (x', y') by adding the **translation distances** (t_x, t_y):

$$
\begin{aligned}
x' &= x + t_x \\
y' &= y + t_y
\end{aligned}
$$

The above equations can be expressed in matrix form by:

$$
\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}
$$

So, the translation of the two dimensional vector P by T into P' is given by:

$$
P' = P + T
$$

where

$$
\begin{aligned}
P' &= \begin{bmatrix} x' \\ y' \end{bmatrix} \\
P &= \begin{bmatrix} x \\ y \end{bmatrix} \\
T &= \begin{bmatrix} t_x \\ t_y \end{bmatrix}
\end{aligned}
$$

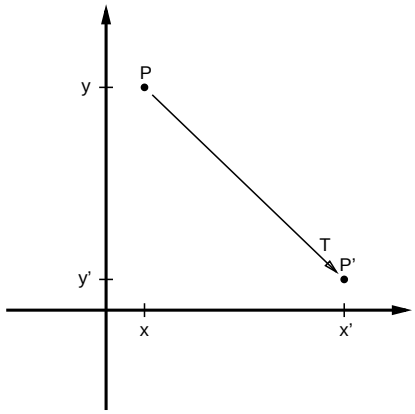


Figure 6.1: Two Dimensional Translation

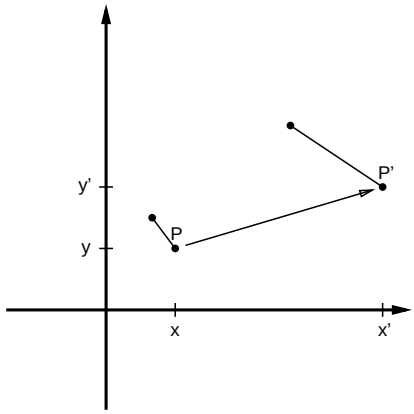


Figure 6.2: Two Dimensional Scaling

6.2.2 Scaling relative to origin

The vertex (x, y) is scaled into the vertex (x', y') by multiplying it with the **scaling factors** s_x and s_y:

$$\begin{aligned} x' &= s_x x \\ y' &= s_y y \end{aligned}$$

This can be expressed in matrix form by:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

or

$$P' = S \cdot P$$

where

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

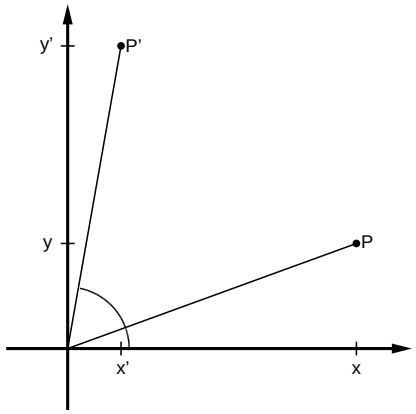


Figure 6.3: Two Dimensional Rotation

- If $s_x = s_y$ the transformation is called a **uniform scaling**.
- If $s_x \neq s_y$ it is called a **differential scaling**.

6.2.3 Rotation relative to origin

The point (x, y), or (r, ϕ) in polar coordinates, is rotated anticlockwise about the origin by θ into the point (x', y'), or $(r, \phi + \theta)$. So

$$\begin{aligned} x' &= r\cos(\phi + \theta) \\ &= r(\cos\phi\cos\theta - \sin\phi\sin\theta) \end{aligned}$$

$$\begin{aligned} y' &= r\sin(\phi + \theta) \\ &= r(\cos\phi\sin\theta + \sin\phi\cos\theta) \end{aligned}$$

Now

$$\begin{aligned} x &= r\cos\phi \\ y &= r\sin\phi \end{aligned}$$

Therefore

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}$$

This can be expressed by:

$$P' = R \cdot P$$

where

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

6.3 Homogeneous coordinates

A combination of translations, rotations and scaling can be expressed as:

$$\begin{aligned} P' &= S \cdot R \cdot (P + T) \\ &= (S \cdot R) \cdot P + (S \cdot R \cdot T) \\ &= M \cdot P + A \end{aligned}$$

Because scalings and rotations are expressed as matrix multiplication but translation is expressed as matrix addition, it is not, in general, possible to combine a set of operations into a single matrix operation. Composition of transformations is often desirable if the same set of operations have to be applied to a list of position vectors.

We can solve this problem by representing points in *homogenous coordinates*. In homogenous coordinates we add a third coordinate to a point i.e. instead of representing a point by a pair (x, y), we represent it as (x, y, W). Two sets of homogenous coordinates represent the same point if one is a multiple of the other e.g. $(2, 3, 6)$ and $(4, 6, 12)$ represent the same point.

Given homogenous coordinates (x, y, W), the cartesian coordinates (x', y') correspond to $(x/W, y/W)$ i.e. the homogenous coordinates at which $W = 1$. Points with $W = 0$ are called points at infinity.

Representing points as 3-dimensional vectors, we can re-write our transformation matrices as follows:

Translation

A translation by (t_x, t_y) is represented by:

$$P' = T(t_x, t_y) \cdot P$$

where

$$T(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling

Scaling by the factors s_x and s_y relative to the origin is given by:

$$P' = S(s_x, s_y) \cdot P$$

where

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation

Rotation about the origin by θ is given by:

$$P' = R(\theta) \cdot P$$

where

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

6.4 Composition of 2D Transformations

Using the associative property of matrix multiplication, we can combine several transformations into a single matrix.

It can easily be shown that:

$$\begin{aligned} T(t_{x_2}, t_{y_2}) \cdot T(t_{x_1}, t_{y_1}) &= T(t_{x_1} + t_{x_2}, t_{y_1} + t_{y_2}) \\ S(s_{x_2}, s_{y_2}) \cdot S(s_{x_1}, s_{y_1}) &= S(s_{x_1} s_{x_2}, s_{y_1} s_{y_2}) \\ R(\theta_2) \cdot R(\theta_1) &= R(\theta_1 + \theta 2) \end{aligned}$$

6.4.1 General fixed-point Scaling

Scaling with factors s_x and s_y with respect to the **fixed point** (x, y) can be achieved by the following list of operations:

1. translate by $(-x, -y)$ so that the fixed point is moved to the origin;
2. scale by s_x and s_y with respect to the origin;
3. translate by (x, y) to return the fixed point back to its original position.

So, the general fixed-point scaling transformation matrix is:

$$\begin{aligned} S_{x,y}(s_x, s_y) &= T(x, y) \cdot S(s_x, s_y) \cdot T(-x, -y) \\ &= \begin{bmatrix} s_x & 0 & x(1 - s_x) \\ 0 & s_y & y(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

6.4.2 General pivot-point rotation

A rotation about the **rotation-point** or **pivot-point** (x, y) by θ is achieved as follows:

1. translate by $(-x, -y)$;
2. rotate by θ;
3. translate by (x, y).

The general pivot-point rotation matrix is therefore given by:

$$\begin{aligned} R_{x,y}(\theta) &= T(x, y) \cdot R(\theta) \cdot T(-x, -y) \\ &= \begin{bmatrix} \cos\theta & -\sin\theta & x(1 - \cos\theta) + y\sin\theta \\ \sin\theta & \cos\theta & y(1 - \cos\theta) - x\sin\theta \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

6.4.3 General direction scaling

Scaling an object by the factors s_x and s_y along a line making angle θ with the x-axis can be achieved as follows:

1. rotate by $-\theta$;
2. scale by s_x and s_y;
3. rotate by θ.

This transformation can be expressed by the matrix:

$$S_\theta(s_x, s_y) = R(\theta) \cdot S(s_x, s_y) \cdot R(-\theta)$$

6.5 Inverse transformations

The matrix of an inverse transformation is the inverse matrix of the transformation: if

$$P' = M \cdot P$$

then

$$P = M^{-1} \cdot P'$$

It can easily be shown that:

$$\begin{aligned} T(t_x, t_y)^{-1} &= T(-t_x, -t_y) \\ S(s_x, s_y)^{-1} &= S(\frac{1}{s_x}, \frac{1}{s_y}) \\ R(\theta)^{-1} &= R(-\theta) \end{aligned}$$

6.6 Other 2D Transformations

6.6.1 Reflection

Reflection about the y-axis can be accomplished by multiplying the x coordinate by -1 and leaving the y coordinate unaltered:

$$\begin{aligned} x' &= -x \\ y' &= y \end{aligned}$$

This can represented by:

$$P' = F_y \cdot P$$

where

$$F_y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

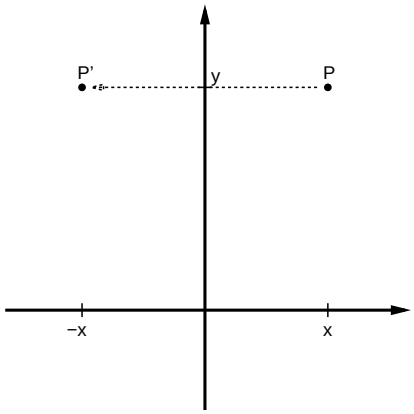


Figure 6.4: Reflection about the y-axis

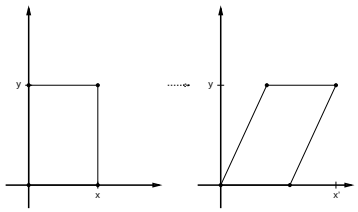


Figure 6.5: Shearing in the x direction

Similarly, reflection about the x-axis is represented by:

$$P' = F_x \cdot P$$

where

$$F_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Reflection can be represented by negative unit scaling:

$$F_y = S(-1, 1)$$
$$F_x = S(1, -1)$$

6.6.2 Shearing

A shearing in the x direction relative to the x-axis unalters the y coordinate while shifts the x coordinate by a value directly proportional to the y coordinate.

An x direction shearing relative to the x-axis with **shearing parameter** h_x is given by:

$$P' = H_x(h_x) \cdot P$$

where

$$H_x(h_x) = \begin{bmatrix} 1 & h_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

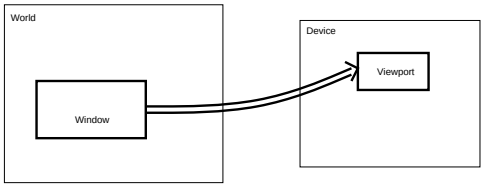


Figure 6.6: Window to viewport mapping

Similarly, a y direction shearing relative to the y-axis with shearing parameter h_y is represented by:

$$P' = H_y(h_y) \cdot P$$

where

$$H_y(h_y) = \begin{bmatrix} 1 & 0 & 0 \\ h_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

6.7 Applying transformations

Transformations can be applied to polygons simply by transforming the polygon vertices, and then joining them with lines in the usual way. However, with the exception of translation and reflection, transformation of circles, ellipses, and bitmap images is not so straightforward. For example:

- when a circle or ellipse is sheared, its symmetrical properties are lost;
- if a circle or bitmap is scaled by applying the scaling transformation to each plotted pixel, this could result in gaps (if the scale factor > 1) or the same pixel being plotted several times (if the scale factor < 1), possibly leaving the wrong colour in the case of bitmaps;

In general, algorithms for transforming bitmap images must ensure that no holes appear in the image, and that where the image is scaled down, the colour chosen for each pixel represents the best choice for that particular area of the picture.

To simplify transformation of circles and ellipses, these are often approximated using polygons with a large number of sides.

6.8 The 2D viewing pipeline

Viewing involves transforming an object specified in a *world coordinates* frame of reference into a *device coordinates* frame of reference. A region in the world coordinate system which is selected for display is called a *window*. The region of a display device into which a window is mapped is called a *viewport*.

The steps required for transforming world coordinates (or modelling coordinates) into device coordinates are referred to as the *viewing pipeline*. Steps involved in the 2D viewing pipeline:

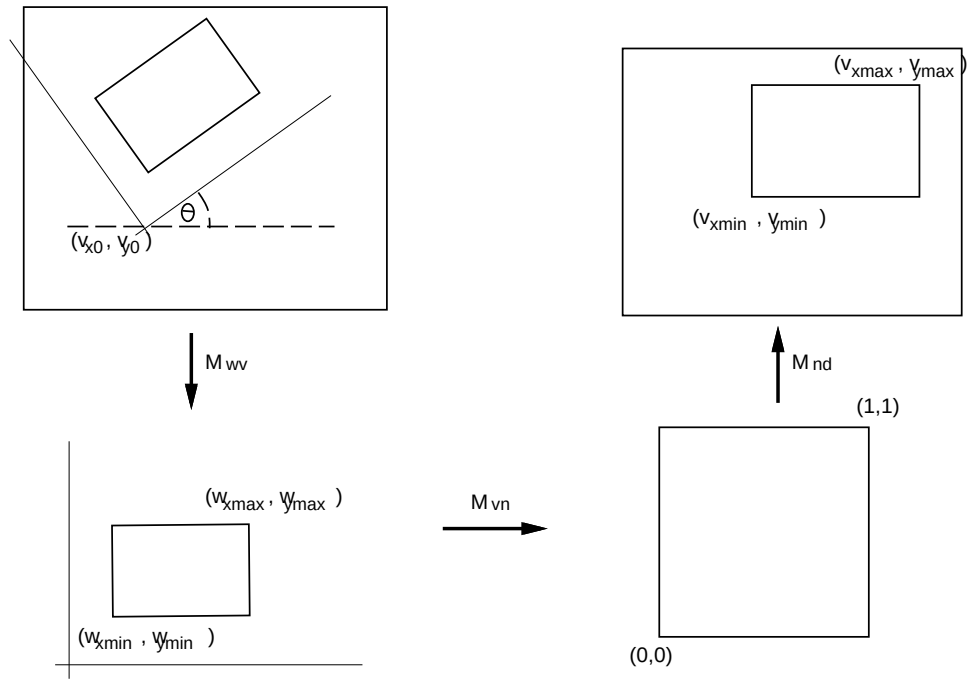


Figure 6.7: Example of viewing pipeline

1. Transforming world coordinates into *viewing coordinates*, usually called the viewing transformation.
2. Normalising the viewing coordinates.
3. Transforming the normalised viewing coordinates into device coordinates.

- Given 2D objects represented in world coordinates, a *window* is specified in terms of a viewing coordinate system defined relative to the world coordinate system.
- The object's world coordinates are transformed into viewing coordinates and usually normalised so that the extents of the window fit in the rectangle fit in the rectangle with lower left corner at $(0, 0)$ and the upper right corner at $(1, 1)$.
- The normalised viewing coordinates are then transformed into device coordinates to fit into the viewport.

6.8.1 Example of a Window to Viewport Transformation

Consider a viewing coordinate system defined as a Cartesian coordinate system with the origin having world coordinates (u_0, v_0). The axis of the viewing coordinate system makes an angle of θ with the world coordinate system x-axis. The transformation M_{wv} transforms a point with world coordinates P_w into viewing coordinates P_v. This transformation can be achieved by first translating by $(-u_0, -v_0)$ and then rotating by $-\theta$. So

$$P_v \quad = \quad M_{wv} \cdot P_w$$

where

$$M_{wv} = R(-\theta) \cdot T(-u_0, -v_0)$$

The window is defined as the rectangle having bottom-left corner at (x_l, y_b), the opposite corner at (x_r, y_t), and edges parallel to the viewing coordinate axes.

The normalising transformation M_{vn} is obtained by first translating by $(-x_l, -y_b)$ and then scaling with factors $\frac{1}{x_r - x_l}$ and $\frac{1}{y_t - y_b}$, so that the lower left and upper right vertices of the window have normalised viewing coordinate $(0, 0)$ and $(1, 1)$ respectively.

So a point with viewing coordinates P_v has normalised viewing coordinates:

$$P_n = M_{vn} \cdot P_v$$

where

$$M_{vn} = S(\frac{1}{x_r - x_l}, \frac{1}{y_t - y_b}) \cdot T(-x_l, -y_b)$$

Transforming into the device coordinates is accomplished by the matrix M_{nd}. If the viewport is a rectangle with the lower left and upper right corners having device coordinates (u_l, v_b) and (u_r, v_t) respectively, then M_{nd} is achieved by first scaling with factors $(u_r - u_l)$ and $(v_t - v_b)$ and then translating by (u_l, v_b):

$$\begin{aligned} P_d &= M_{nd} \cdot P_n \\ M_{nd} &= T(u_l, v_b) \cdot S(u_r - u_l, v_t - v_b) \end{aligned}$$

Thus, the whole viewing pipeline can be achieved by concatenating the three matrices:

$$\begin{aligned} M_{wd} &= M_{nd} \cdot M_{vn} \cdot M_{wv} \\ &= T(u_l, v_t) \cdot S(u_r - u_l, v_t - v_b) \\ &\quad \cdot S(\frac{1}{x_r - x_l}, \frac{1}{x_r - x_l}) \cdot R(-\theta) \cdot T(-u_0, -v_0) \\ &= T(u_l, v_t) \cdot S(\frac{u_r - u_l}{x_r - x_l}, \frac{v_t - v_b}{x_r - x_l}) \\ &\quad \cdot R(-\theta) \cdot T(-u_0, -v_0) \end{aligned}$$

Defining

$$\begin{aligned} s_x &= \frac{u_r - u_l}{x_r - x_l} \\ s_y &= \frac{v_t - v_b}{y_t - y_b} \end{aligned}$$

If $s_x \neq s_y$, M_{wd} scales objects differentially; however if we want objects to be scaled uniformly during the viewing process and that all objects in the window are mapped into the viewport, then we define

$$s' = \min(s_x, s_y)$$

and scale using $S(s', s')$ instead of $S(s_x, s_y)$.

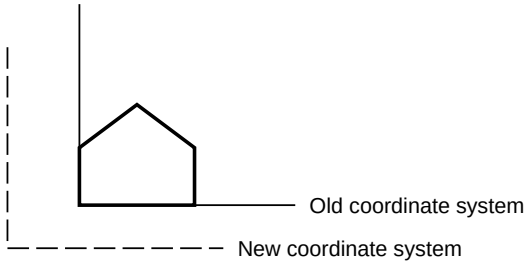


Figure 6.8: Change in coordinate system using $T(5, 3)$

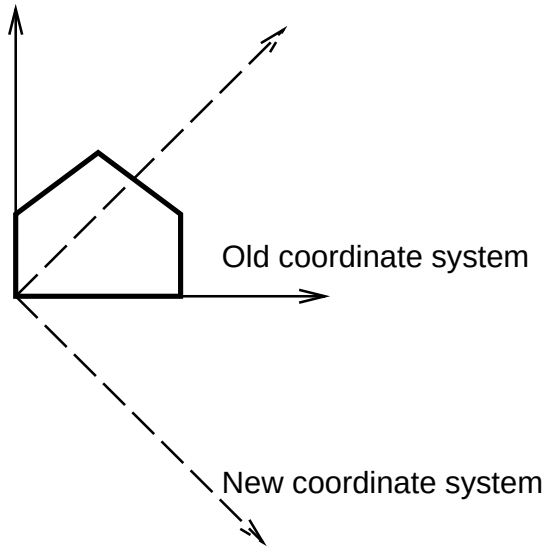


Figure 6.9: Change in coordinate system using $R(45)$

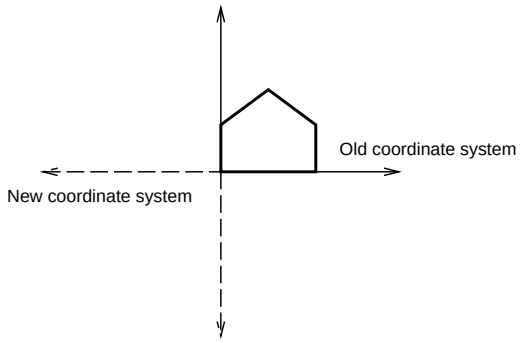


Figure 6.10: Change in coordinate system using $S(-1)$

6.9 Transformation as a change in coordinate system

Rather than thinking of transformations as operations which manipulate graphic objects, it is sometimes useful to think of them as representing changes in coordinate system. This is especially useful when multiple objects, each defined relative to its own local coordinate system, are to be combined in a 'higher-level' object expressed in terms of a single global coordinate system (we will use this when we use SPHIGS).

Examples:

- the transformation $T(5,3)$ effects a change in coordinate system whereby the origin of the new coordinate system corresponds to the point $(-5,-3)$ in the old system (figure 6.8);
- the transformation $S(2,2)$ gives us a new coordinate system in which each unit value is half the size of that in the old system;
- the transformation $R(45)$ gives us a new coordinate system in which the x-axis makes an angle of $-45°$ with the old x-axis (figure 6.9);
- the transformation $S(-1,-1)$ gives us a new coordinate system in which x and y values respectively increase to the left and downwards instead of to the right and upwards (figure 6.10).

6.10 3D transformations

3D transformations are just an extension of 2D ones. Just like 2D transformations are represented by 3×3 matrices, 3D transformations are represented by 4×4 matrices.

Translation by (t_x, t_y, t_z) is represented by the matrix:

$$T(t_x,t_y,t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling by the factors s_x, s_y and s_z, along the co-ordinate axes is given by the matrix:

$$S(s_x,s_y,s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the x-axis:

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

Rotation about the y-axis:

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

Rotation about the z-axis:

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

Twisting is a purely 3D transformation (ie no 2D equivalent). For twisting along along the z-axis use:

$$\begin{aligned} x' &= x\cos(\alpha z) - y\sin(\alpha z) \\ y' &= x\sin(\alpha z) + y\cos(\alpha z) \\ z' &= z \end{aligned}$$

6.11 Exercises

1. Implement a matrix library.
2. Use the matrix library to create routines that scale, rotate, translate, shear and reflect 2D objects.
3. Create an application for displaying and animating a mechanical arm made up of two rectangles. Each rectangle represents a section of the arm. Add functions that rotate and move the entire arm, and a function to rotate the lower part of the arm only. Note that changes to the upper arm must be reflected in the forearm.

Chapter 7

Viewing in 3D

7.1 Preliminaries

7.1.1 3D co-ordinate systems

A point in 3D space can be referenced relative to an RHS or LHS cartesian coordinate system (figure 7.1) using three coordinate values. This can be represented as a 1×3 matrix, (or as a 1×4 matrix for homogenous transformations). We will normally be working with an RHS system.

The transformation $S(1, 1, -1)$ can be used to transform from a right-handed 3D Cartesian coordinate system to a left-handed one and vice-versa.

7.1.2 Representation of 3D objects

One way of representing a 3D object is as a list of 2D polygons in 3D space. For example, a pyramid consists of a square base and four triangles.

- Curved surfaces may be approximated using polygons, although there are techniques which represent curved surfaces more accurately, including generalisations of bezier and spline curves.
- If 3D objects are represented as lists of edges (as opposed to polygons), then this allows rendering of 'see-through' wireframe images. However, hidden-line removal for more

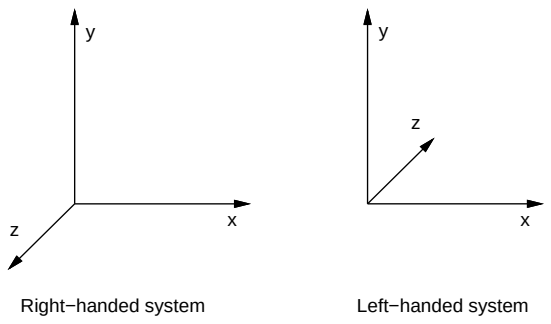


Figure 7.1: RHS and LHS coordinate systems

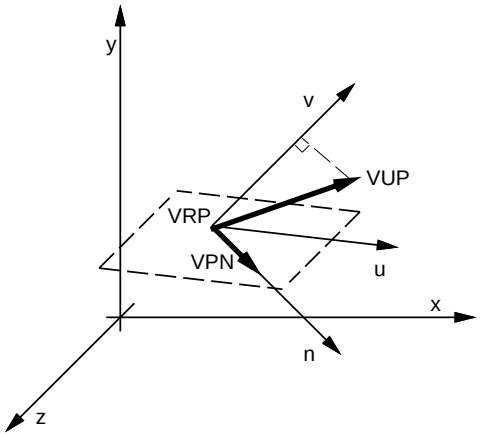


Figure 7.2: 3D view reference coordinate system

realistic images is not possible because this requires knowledge of the surfaces of the object.

7.2 The 3D viewing pipeline

Three dimensional viewing is more complex than 2D viewing because, while the world coordinate systems is three dimensional, the device on which they are displayed is usually two dimensional. This requires an additional transformation, namely the *projection transformation*, in the viewing pipeline, whereby 3D coordinates are transformed into 2D.

The basic steps involved in 3D viewing are:

1. transforming world coordinates into viewing coordinates;
2. transforming viewing coordinates into projection coordinates (the projection transformation);
3. transforming projection coordinates into device coordinates.

The first step effectively defines a position and orientation for a virtual camera, whereas the second step determines the kind of picture to be taken,

Note that whereas clipping in 2D is carried out before applying the viewing pipeline, in 3D it is often delayed to just before the third step. This is because in 3D clipping has to be performed against a *view volume* which may be of an awkward shape e.g. a frustrum in the initial stages. After carrying out the projection transformation, the view volume is a regular parallelepiped which is much easier to clip against.

7.2.1 The view reference coordinate system

The *viewing-reference coordinate system* (VRC) is defined in terms of a *view reference point* (VRP) which represents the origin, a *view-plane normal vector* (VPN) and a *view-up vector* (VUP). This is analogous to placing a camera (or an eye) at the VRP pointing in the direction given by the VPN. The camera is then rotated around the perpendicular to the plane that passes through it (the camera) so that the VUP is vertical.

The coordinate system uvn (figure 7.2) is set up so that n lies in the direction of the VPN, and v is in the direction of the vector which is in the same plane of the VPN and VUP, is perpendicular to the VNP and makes an acute angle with the VUP. u is chosen so that uvn is right-handed.

The orthogonal unit vectors u, v and n corresponding to the VRC can be constructed as follows:

$$\begin{aligned} n &= \frac{1}{|VPN|}VPN \\ u &= \frac{1}{|VUP \times VPN|}VUP \times VPN \\ v &= n \times u \end{aligned}$$

Let the VRP have world coordinates (x_0, y_0, z_0), and let

$$\begin{aligned} u &= (u_x, u_y, u_z) \\ v &= (v_x, v_y, v_z) \\ n &= (n_x, n_y, n_z) \end{aligned}$$

Transforming from world coordinates to viewing coordinates can be achieved by first translating by $(-x_0, -y_0, -z_0)$ and then aligning the coordinate axes by rotation. Thus if P is a point with world coordinates P_w, its viewing coordinates P_v can be obtained using:

$$P_v = M_{wv} \cdot P_w$$

where

$$\begin{aligned} M_{wv} &= R_{uvn} \cdot T(-x_0, -y_0, -z_0) \\ R_{uvn} &= \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

7.2.2 Projections

In general, a projection is a transformation from an n dimensional space to an m dimensional space, where $m < n$. In this section, we concentrate on projections from a 3D Cartesian coordinate system to a 2D one.

Consider a set of points (x_i, y_i, z_i) in viewing coordinates. To carry out a projection, a line (called a *projector*) is drawn from each point to a fixed point called the *centre of projection* (COP). The projectors intersect the view plane (which is perpendicular to z_v) at the points $(x'_i, y'_i, 0)$. Projections that can be represented as

$$J : (x_i, y_i, z_i) \longmapsto (x'_i, y'_i)$$

are called **planar geometric projections**.

The following parameters must be supplied to fully describe a projection:

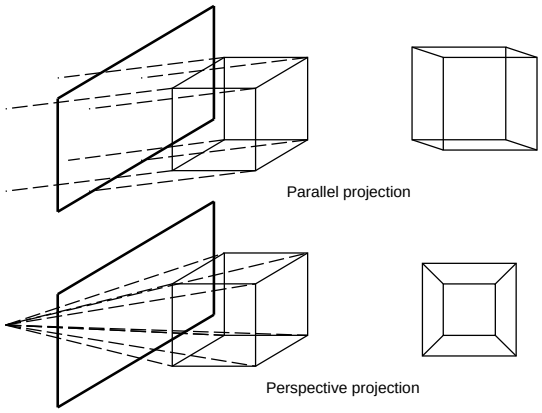


Figure 7.3: Parallel and perspective projections

- a window in the view plane that defines the boundaries of the picture to be taken—this is specified in VRC coordinates and is usually, though not necessarily, taken to be a rectangle with the VRP at the centre;
- a COP towards which all projectors converge;
- *front and back clipping planes*: the front and back clipping planes define cut-off points beyond which graphic objects are not considered for projection, effetively eliminating objects which are too close to (or behind) the camera, or too far away from it.

The position of the COP may tend to infinity; in such cases the projectors are parallel to each other and their direction is referred to as the *direction of projection*. A planar geometric projection constructed using parallel projectors is called a *parallel projection*. If the projectors intersect at the centre of projection, the projection is called a *perspective projection* (see figure 7.3).

Parallel projections preserve all parallel lines as well as the angles in any plane parallel to the plane of projection. Perspective projections preserve parallel lines and angles only in the planes parallel to the plane of projection. However, scenes produced using a perspective projection appear realistic because distant objects appear smaller than near ones. In a scene produced by parallel projection, distant objects have the same size as near ones.

Perspective projections of parallel lines are either parallel lines themselves (if they are parallel to the view plane) or intersect at a *vanishing point*.

7.3 Parallel projections

If the projectors are perpendicular to the plane of projection, the resulting parallel projection is called *orthographic*, otherwise it is called *oblique*.

7.3.1 Orthographic projections

A useful set of views produced by orthographic projections are the *front elevation*, *side elevation* and *plan view* of an object. Such views are produced when the plane of projection is parallel to one of the planes defined by any two principal axes of the world coordinate

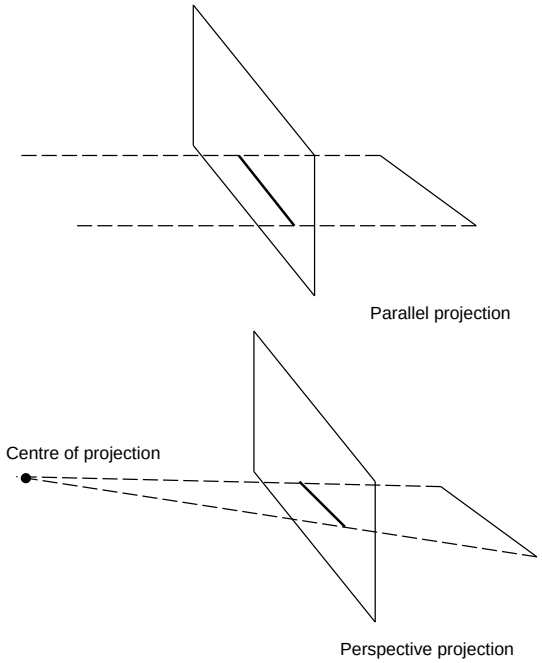


Figure 7.4: Effect of parallel and perspective projections

system, or equivalently, the plane of projection intersects with only one world coordinate principal axis.

The front elevation othographic parallel projection can be represented by the following transformation:

$$\begin{aligned} x_p &= x_v \\ y_p &= y_v \end{aligned}$$

or by the matrix:

$$J_{FrontElevation} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If the projection is orthographic and the plane of projection intersects with more than one world coordinate axis, then the resulting view is termed *axonometric*. One useful axonometric projection is the *isometric projection*, which is produced when the projection plane intersects each principal axis at the same distance from the origin. The projected edges of an isometric view of a cube aligned along the world co-ordinate axes have equal length.

7.3.2 Oblique projections

Oblique parallel projections are produced when the projectors are not perpendicular to the plane of projection. The *direction of projection* (DOP) may be given by 2 angles: α and ϕ. α is defined as the angle each projector makes with the plane of projection, and ϕ as the angle the horizontal makes with the line that joins the projection with the corresponding orthographic projection (figure 7.5). ϕ is usually chosen to be 30° or 45°.

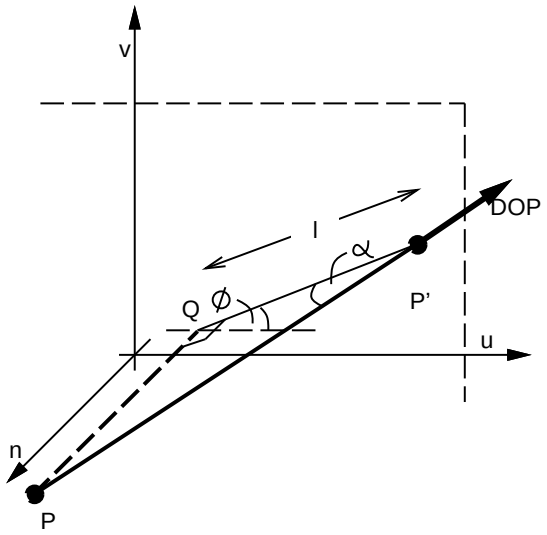


Figure 7.5: Oblique projection

7.3.3 Parallel projection transformation

Consider the point P with view coordinates (x_v, y_v, z_v) projected to the point P' with coordinates (x_p, y_p).

Let Q be the orthographic projection of P on the plane, and l be the length of line QP'. Then

$$l = \frac{z_v}{\tan\alpha}$$

and the coordinates of P' are:

$$\begin{aligned} x_p &= x_v + l\cos\phi \\ &= x_v + z_v\frac{\cos\phi}{\tan\alpha} \\ y_p &= y_v + l\sin\phi \\ &= y_v + z_v\frac{\sin\phi}{\tan\alpha} \end{aligned}$$

Oblique parallel projections can thus be obtained using the transformation matrix:

$$J_{Oblique} = \begin{bmatrix} 1 & 0 & \frac{\cos\phi}{\tan\alpha} & 0 \\ 0 & 1 & \frac{\sin\phi}{\tan\alpha} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In the case of an orthographic projection, we have $\phi = 0$ and $\alpha = 90$, giving us the transformation:

$$J_{Orthogonal} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

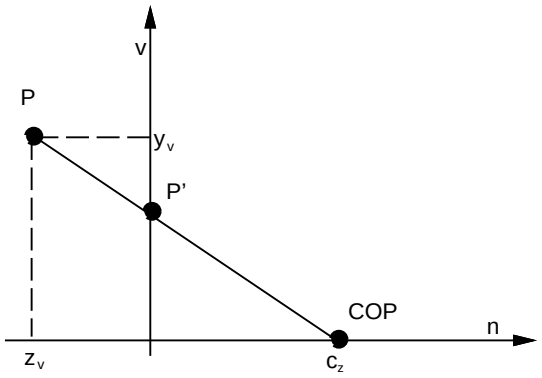


Figure 7.6: Perspective projections

Note that in the orthogonal case we simply discard the z coordinate.

Note also that when using the matrix representations of these transformations we still end up with a vector representing a point in 3D space, albeit with a z value of 0.

7.4 Perspective Projections

Let us consider a point P with viewing coordinates (x_v, y_v, z_v). The point is projected to P' with coordinates (x_p, y_p). We will consider the special case where the centre of projection lies on the z_v-axis at the point $(0, 0, c_z)$. The projection is given by the transformation:

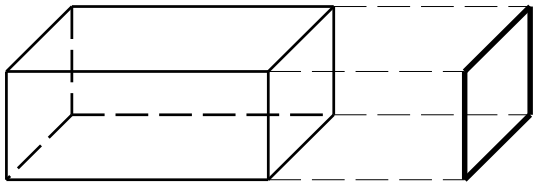
$$\begin{aligned} x_p &= x_v \frac{-c_z}{z_v - c_z} \\ &= x_v \frac{1}{-\frac{z_v}{c_z} + 1} \\ y_p &= y_v \frac{-c_z}{z_v - c_z} \\ &= y_v \frac{1}{-\frac{z_v}{c_z} + 1} \end{aligned}$$

Multiplying by a value dependent on z_v causes the size of objects to vary depending on their distance from the origin. Note that a value $z_v = c_z$, will cause problems as the projected point will be at infinity i.e. objects are infinitely large at the view plane. Normally we will clip points for which $z_v \leq c_z$.

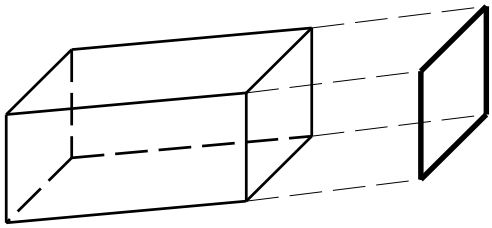
We can write the transformation in matrix form:

$$J_{Perspective} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{c_z} & 1 \end{bmatrix}$$

This gives us the coordinate $\begin{bmatrix} x_v & y_v & 0 & 1 - \frac{z_v}{c_z} \end{bmatrix}^T$, which which when homogenized gives us the point $\begin{bmatrix} \frac{x_v}{-\frac{z_v}{c_z}+1} & \frac{y_v}{-\frac{z_v}{c_z}+1} & 0 & 1 \end{bmatrix}^T$.



Orthogonal



Oblique

Figure 7.7: View volume for parallel projection

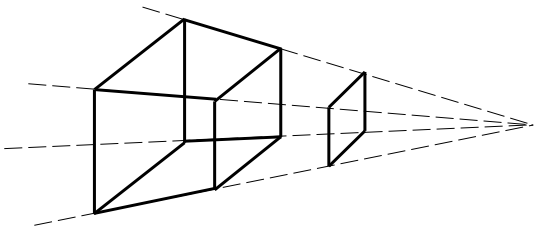


Figure 7.8: View volume for perspective projection

7.5 Clipping in 3D

The part of our world which is to be captured by our 2D image is determined by the projection window, the front and back clipping planes and the centre of projection. Together these define a *view volume* corresponding to that portion of the world to be displayed. This view volume and has the shape of a parallelepiped for parallel projection views and the shape of a frustrum for the case of perspective projections.

As in the case of 2D, objects lying (partially) outside the view volume must be clipped, leaving only those (parts of) objects that are relevant to the view under consideration. This clipping operation can be performed before or after applying the projection transformation.

For oblique and perspective projections the clipping and subsequent projection are made much simpler if the view volume is normalised, or transformed into a regular parallelepiped (ie cuboid) view volume. We will use a normalized view volume defined by the six planes $x = -1, x = 1, y = -1, y = 1, z = 0, z - 1$

We will use the following approach:

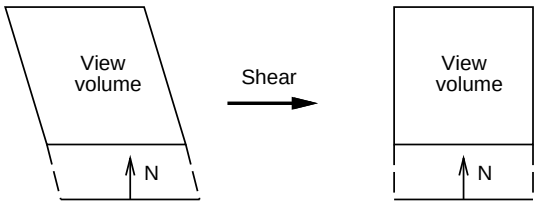


Figure 7.9:

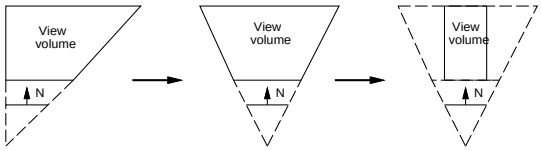


Figure 7.10:

- transform view volume into a normalized view volume;
- clip lines about the normalized volume, using a more general form of the Cohen-Sutherland algorithm.

Subsequently the normalized view volume may be mapped onto a 2D viewport using an orthogonal projection. This step may also involve other operations, including hidden surface removal, and colour shading.

Note that in the case of orthographic projections, the view volume is already a regular parallelepiped (although this is not normalized).

7.5.1 Normalisation of view volume

Three cases:

1. parallel orthographic: translate and scale to fit normalized view volume;
2. parallel oblique: shear view volume to align projection direction with normal to projection plane N, then translate and scale to fit normalized view volume;
3. perspective projection:
   (a) translate view volume to bring COP to the origin;
   (b) shear view volume to align centre line of view volume with n axis (figure 7.9);
   (c) scale view volume relative to COP, to obtain a canonical view volume bounded by the planes $x = z, x = -z, y = z, y = -z, z = z_{min}, z = -1$ where $-1 < z_{min} < 0$.
   (d) scale view volume to obtain normalized view volume, using a varying scaling factor which is inversely proportional to the distance from the COP (figure 7.10);

In order to deal with parallel and perspective projections in a uniform manner, it is useful to define the notion of a *projection reference point* (PRP). In the case of perspective projections, this is defined to be the centre of projection. In the case of parallel projections,

it is defined such that a line drawn from the centre of the window to the PRP is in the direction of the DOP.

The following derivations for oblique and perspective normalisation transformations assume

- a projection window W in the uv plane bounded by minimum and maximum u and v values u_l, v_b, u_r and v_t, with window centre $CW = \begin{bmatrix} \frac{u_l+u_r}{2} & \frac{v_b+v_t}{2} & 0 & 1 \end{bmatrix}^T$;
- a PRP defined as $\begin{bmatrix} prp_u & prp_v & prp_n & 1 \end{bmatrix}^T$;
- front and back clipping planes defined by $z = F$, $z = B$.

Parallel projection normalisation

The direction of projection (DOP) is given by $CW - PRP$. The shear transformation matrix required is of the form:

$$\begin{aligned} SH_{par} &= SH_{xy}(shx_{par})(shy_{par}) \\ &= \begin{bmatrix} 1 & 0 & shx_{par} & 0 \\ 0 & 1 & shy_{par} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

which leaves the n coordinate unaffected as desired.

We require the values for shx_{par} and shy_{par} such that:

$$\begin{aligned} SH_{par} \cdot DOP &= \begin{bmatrix} 0 \\ 0 \\ -prp_z \\ 0 \end{bmatrix} \\ &= DOP' \end{aligned}$$

where DOP' corresponds to direction of projection along the n axis. Solving this equation gives us:

$$\begin{aligned} shx_{par} &= \frac{prp_u - \frac{u_r+u_l}{2}}{prp_z} \\ shy_{par} &= \frac{prp_v - \frac{v_t+v_b}{2}}{prp_z} \end{aligned}$$

Finally we translate and scale to fit the normalized view volume (this is similar to the 2D case). The transformations required are:

$$\begin{aligned} T_{par} &= T\left(-\frac{u_r+u_l}{2}, -\frac{v_t+v_b}{2}, -F\right) \\ S_{par} &= S\left(\frac{2}{u_r-u_l}, \frac{2}{v_t-v_b}, \frac{1}{F-B}\right) \end{aligned}$$

The complete transformation for parallel projections (starting from world coordinates) is therefore given by:

$$N_{par} = S_{par} \cdot T_{par} \cdot SH_{par} \cdot R \cdot T(-VRP)$$

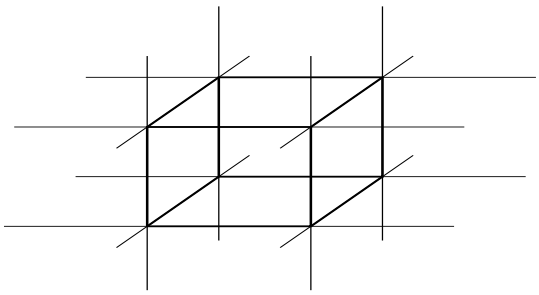


Figure 7.11:

Perspective projection normalisation

The first step involves translating the PRP to the origin using $T(-prp_u, -prp_v, -prp_n)$.

Next we need to shear the volume to align the line passing through the translated window centre and the origin with the n-axis i.e. the direction $CW - PRP$ and the n axis. This is exactly the same as for the parallel case, and can be achieved using SH_{par}.

After this shear, the situation is as follows:

- the window (and hence the view volume) is centred around the z axis;
- the VRP, previously at the origin has been translated and sheared to get $VRP' = SH_{par} \cdot T(-PRP) \cdot \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T$.

Next we need to scale the view volume to obtain a perspective canonical view volume. This can be done in two steps: first scale using $S(\frac{2 \cdot prp_n}{u_r + u_l}, \frac{2 \cdot prp_n}{v_t + v_b}, 1)$ to bring x and y coordinates between -1 and 1. Then scale by $S(\frac{1}{prp_n - B})$ to align the back clipping plane with $z = -1$.

At this stage it is possible to clip against this canonical volume. However we will perform one last step to obtain a parallel canonical view volume, thereby making it possible to use the same clipping and rendering techniques as for parallel projections. It can be shown that the transformation to do this is:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{min}} & \frac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note that this transformation produces coordinates which are NOT homogenized.

Thus for perspective projections, the complete transformation (starting from world coordinates) is given by:

$$N_{per} = M \cdot S_{per} \cdot SH_{par} \cdot T(-PRP) \cdot R \cdot T(-VRP)$$

7.5.2 3D version of Cohen-Sutherland clipping algorithm

The 3D version of the Cohen-Sutherland algorithm is simply a generalisation of the 2D version, using a 6-bit code instead of a 4-bit one (figure 7.11), to enable clipping against the

sides of a regular parallelepiped. Note that here lines are clipped against planes instead of lines. It is useful to make use of the parametric equation of a line:

$$\begin{aligned} x &= x_1 + (x_2 - x_1)u \\ y &= y_1 + (y_2 - y_1)u \\ z &= z_1 + (z_2 - z_1)u \\ & 0 \leq u \leq 1 \end{aligned}$$

Example: to clip against $z = z_m in$, substitute $u = \frac{z_m in - z_1}{z_2 - z_1}$ in the equations for x and y.

7.6 Mapping into a 2D viewport

Having constructed a normalised view volume in the shape of a regular parallelepiped, a 2D image can be constructed by projecting the volume orthographically. In the process of doing this, various other operations may be carried out: invisible edges and surfaces may be eliminated, and colouring and shading algorithms may be applied to produce a more realistic image. Techniques for *rendering* the final image are discussed in later lectures.

Chapter 8

Hidden surface and hidden line removal

Hidden surface (line) removal is the process of removing surfaces (lines) which are not visible from the chosen viewing position. Hidden surface/line algorithms are often classified as one of the following:

- object space: algorithms which work on object definitions directly;
- image space: algorithms which work on the projected image.

There is no one place in the viewing pipeline where these algorithms should be used—the ideal position in the pipeline is often specific to the particular algorithm. Some algorithms do not perform hidden surface/line-removal completely, but are designed to do some of the work early on in the pipeline, thereby reducing the amount of surfaces to be processed.

A word of caution: it does not always make sense to say that one surface is behind another! One surface may contain points both behind and in front of another surface (figure 8.1).

8.1 Back-face culling

Back-face culling is an object-space algorithm designed to be used early on in the pipeline just before the projection transformation is applied. It is useful when the objects in the scene consist of solid polyhedra i.e. polyhedra whose edges are all used in exactly two surfaces, so that the inside faces of the polyhedra are never visible.

In back-face culling, the direction of the surface normal is used to determine whether the surface should be kept or eliminated. The direction of the surface normal is such that it

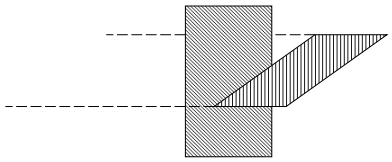


Figure 8.1: Alternately overlapping polygons

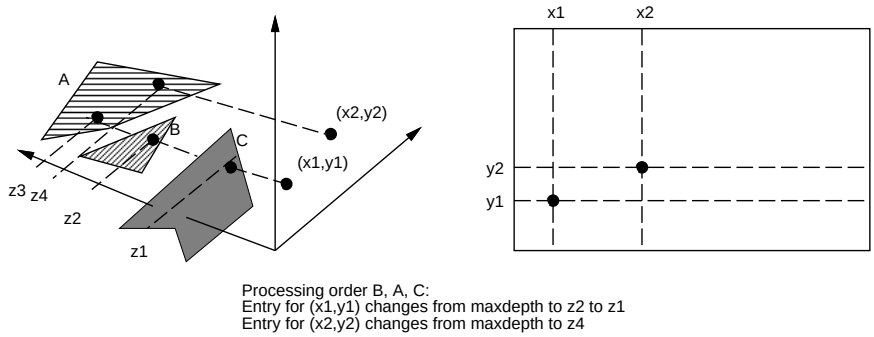


Figure 8.2: z-buffer algorithm

points towards a viewer from whose perspective the listed vertices appear in anti-clockwise order (this is the case with SPHIGS).

Back-face culling eliminates all those surfaces whose normals point away from the COP, in the knowledge that such surfaces would be completely hidden by other front-facing surfaces of the same polyhedron. This can be tested for by computing the dot product of the surface normal, and a vector from the COP to any point on the polygon. If this is > 0 then the surface is eliminated.

In case of polyhedra with missing or clipped front-faces, back-face culling leads to the creation of misleading images. One solution is to have two surfaces in opposite directions defined for each polygon. This is also useful for enabling the surface to have different colours on either of its two sides.

8.2 Depth-buffer (or z-buffer) method

This image space algorithm makes use of a *depth buffer* and a *refresh buffer*. The depth buffer stores a depth value for each pixel in the projection window, whereas the refresh buffer specifies the colour for each pixel in the projection window.

Assuming that the view volume has been converted to a regular parallelepiped with the z-coordinate varying from 0 to 1, the algorithm proceeds as follows:

1. Set all values in the depth buffer to 1 (ie furthermost depth).
2. Set all values in the refresh buffer to the background colour.
3. Loop through all the surfaces to be drawn, applying the following procedure to each projected point (x, y) (typically obtained through scan-line algorithm):
   (a) calculate z-value for (x, y) (using equation of plane of polygon);
   (b) if this z-value is less than the current value stored for (x, y) in the z-buffer, update the buffer to this new z-value and update the (x, y) slot in the refresh buffer to reflect the current polygon's colour (possibly adjusted to reflect the depth).
4. Display the refresh buffer.

Comments:

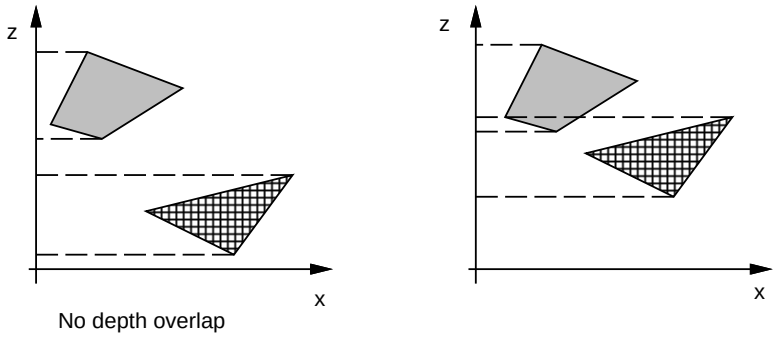


Figure 8.3: Testing for depth overlap

- Easy to implement
- Reasonably fast—no sorting required
- Requires additional depth buffer—memory hungry

Optimizations:

- Divide the projection window into horizontal segments and work on each, one at a time, to reduce the z-buffer size. This slightly complicates the algorithm because we cannot deal with each surface in one loop, but have to return to it for each window segment.
- Rather than calculate the value of z for each (x, y) from scratch, we can do this incrementally. If we know that the depth value at (x, y) for a polygon S is $z(S, x, y)$, and the equation of the plane of S is $Ax + By + Cz = D$, then using the following equations:

$$\begin{aligned} z(S,x,y) &= \frac{Ax - By - D}{C} \\ z(S,x+1,y) &= \frac{-A(x+1) - By - D}{C} \\ z(S,x+1,y-1) &= \frac{-Ax - B(y-1) - D}{C} \end{aligned}$$

we can derive:

$$\begin{aligned} z(S,x+1,y) &= z(S,x,y) - \frac{A}{C} \\ z(S,x,y-1) &= z(S,x,y) + \frac{B}{C} \end{aligned}$$

8.3 Scan-line method

This algorithm processes all polygons simultaneously, one scan-line at a time. Essentially this is an extension of scan-line filling for polygons: as each scan line is processed, all polygon surfaces intersecting with that line are examined to determine which surfaces are visible. At each position, the depth-value for each intersecting polygon is worked out, and the colour

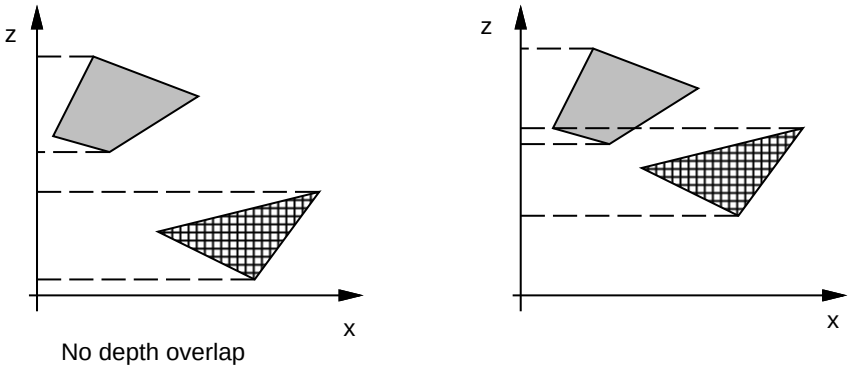


Figure 8.4: Testing for need to re-order

of the nearest polygon is chosen. Of course there is no need to calculate the depth value if only one polygon is active at a particular point.

As long as polygons do not intersect each other, this algorithm can be optimized significantly by only making depth calculations whenever a polygon edge is activated or deactivated, and the number of active polygons is greater than 1.

8.4 Depth-sorting method (painter's algorithm)

This algorithm makes use of both image-space and object-space techniques. The basic steps are:

1. Sort surfaces in order of decreasing greatest depth (object-space).
2. Scan-convert surfaces in order, starting with the surface of greatest depth (image-space).

Of course, problems occur when polygons overlap in depth. Before a polygon S is scan-converted, the next polygon in the list should be checked for depth overlap (figure 8.3). If there is none, the polygon surface can be processed, and the process can be applied to the next polygon in the list. If there is overlap however, some extra checks are required and re-ordering of polygons may be necessary. The following tests (figure 8.4) are carried out until one is found to succeed, in which case no re-ordering is necessary, or all of them fail, in which case additional processing is required.

1. The bounding rectangles in the xy plane for the two surfaces do not overlap.
2. Surface S is on the outside of the overlapping surface, relative to the view plane.
3. The overlapping surface is on the inside of surface S, relative to the view plane.
4. The projections of the two surfaces onto the view plane do not overlap (check for edge intersections).

Note that these tests are ordered so that they become progressively more computing intensive.

If all four tests fail, the two surfaces are swapped, and the process is re-started for the new current polygon. If two surfaces alternately obscure each other, the polygons should be flagged when swapped so as to avoid infinite re-shuffling of polygons. In such cases, the offending polygon can be split into two at the intersection of the two planes under comparison.

8.5 Hidden-line removal

Many hidden-surface removal techniques, including scan-line and depth-sorting, can be adapted to perform hidden-line removal in wire-frame graphics. Instead of colouring surfaces with the surface colour, the background colour is used instead, and only edges are drawn in some specified foreground colour.