

CHAPTER 3. More Drawing Tools.

Computers are useless.
They can only give you answers.

Pablo Picasso

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

Will Rogers

Goals of the Chapter

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

Preview.

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

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

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

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

3.1. Introduction.

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

Vincent Van Gogh

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

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

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

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

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

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

3.2. World Windows and Viewports.

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

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

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

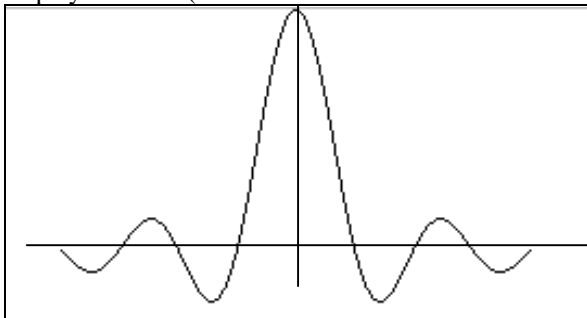


Figure 3.1. A plot of the “sinc” function.

```
void myDisplay(void)
```

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

```

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

```

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

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

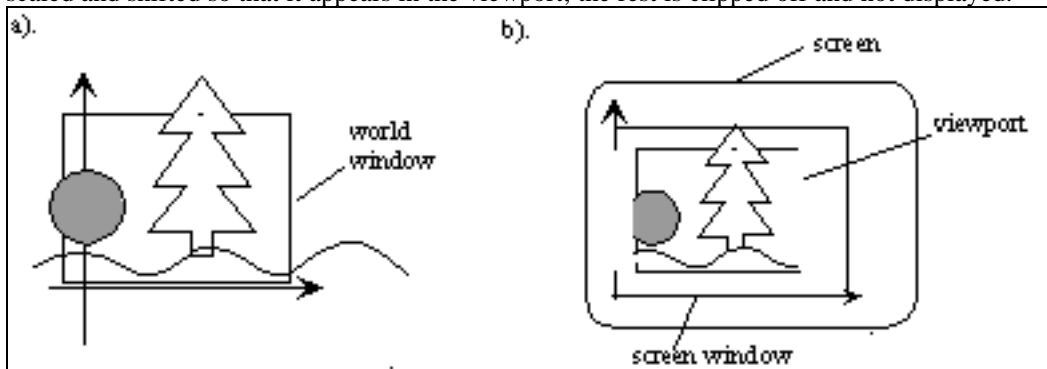


Figure 3.2. A world window and a viewport.

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

3.2.1. The mapping from the window to the viewport.

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

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

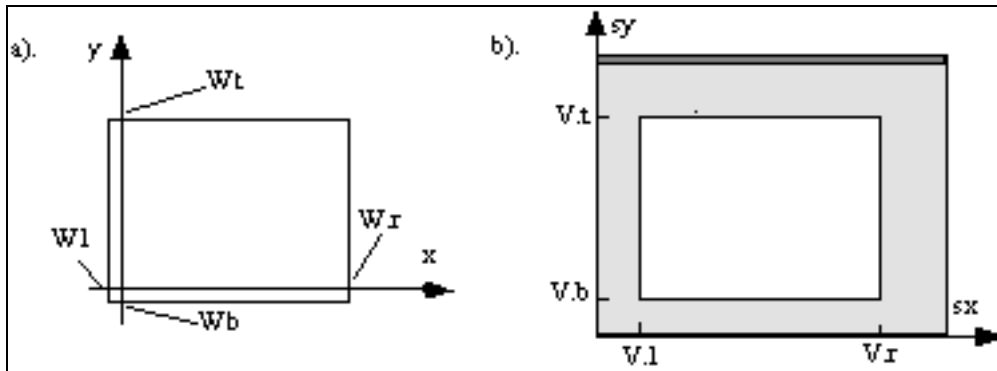


Figure 3.3. Specifying the window and viewport.

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

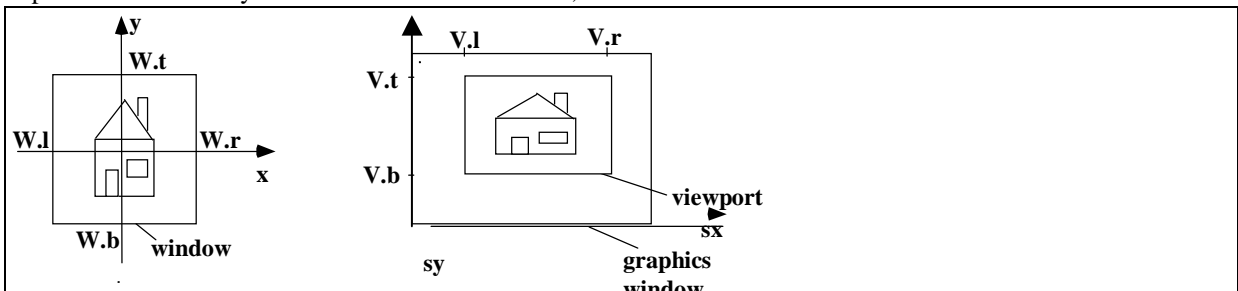


Figure 3.4. A picture mapped from a window to a viewport. Here some distortion is produced.

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

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

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

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

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

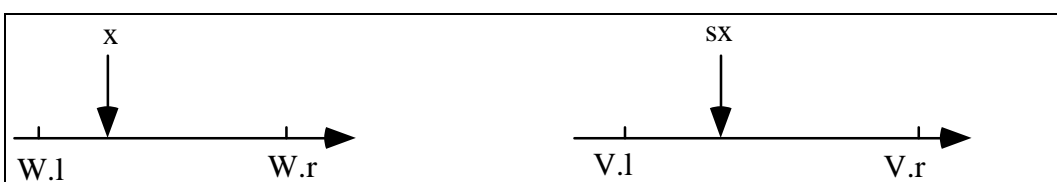


Figure 3.5. Proportionality in mapping x to sx .

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

or

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

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

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

Similarly, proportionality in y dictates that

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

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

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

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

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

with

(3.3)

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

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

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

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

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

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

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

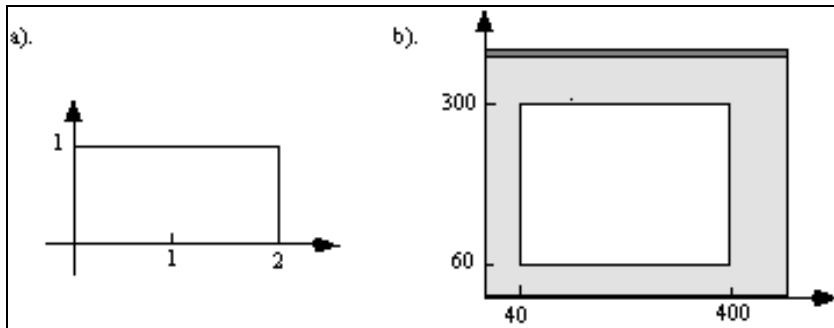


Figure 3.6. An example of a window and viewport.

Using the formulas in Equation 3.2.2 we obtain

$$\begin{aligned} A &= 180, C = 40, \\ B &= 240, D = 60 \end{aligned}$$

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

$$\begin{aligned} sx &= 180x + 40 \\ sy &= 240y + 60 \end{aligned}$$

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

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

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

Doing it in OpenGL.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1). in `main()`:

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

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

2). in `myInit()`:

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

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

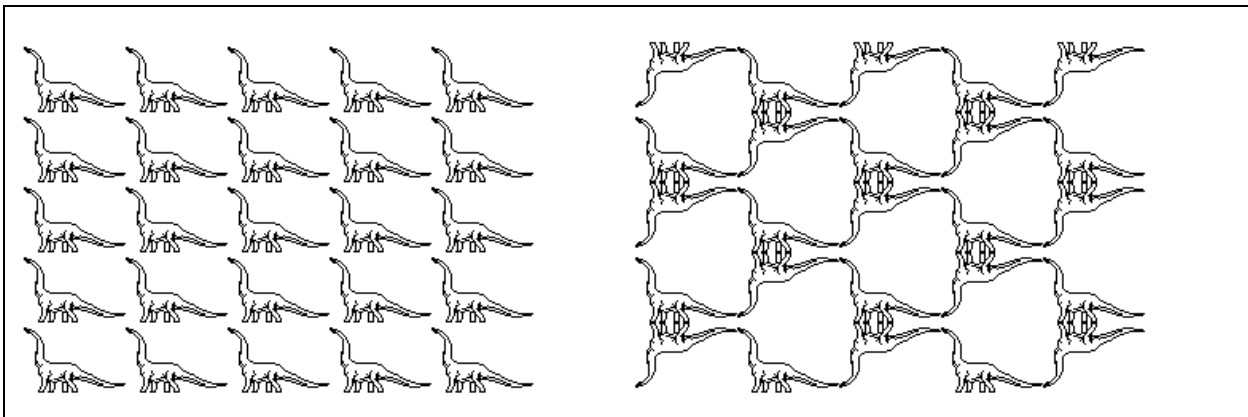


Figure 3.10. Tiling the display with copies of the dinosaur.

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

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

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

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

Example 3.2.5. Clipping parts of a figure.

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

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

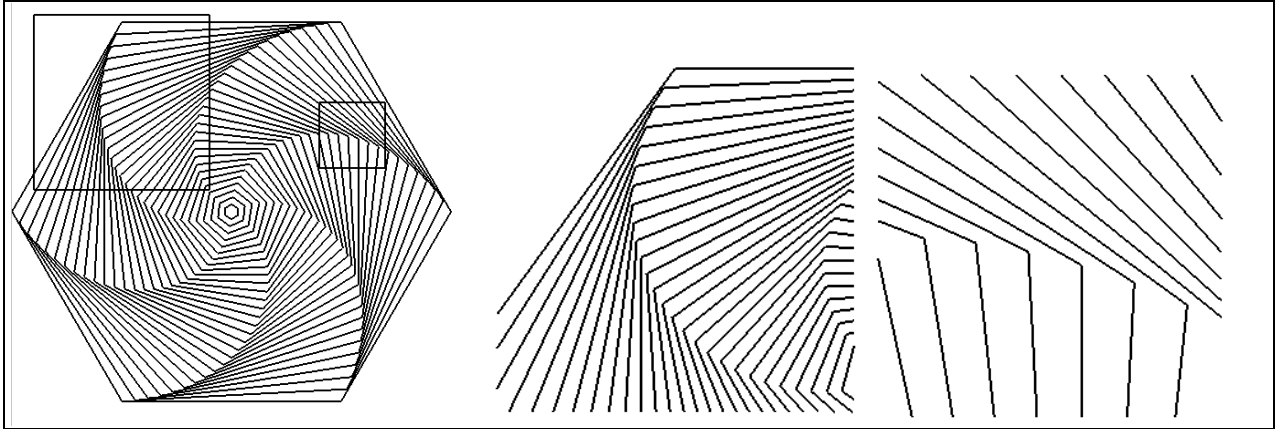


Figure 3.11. Using the window to clip parts of a figure.

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

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

Zooming and roaming.

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

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

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

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

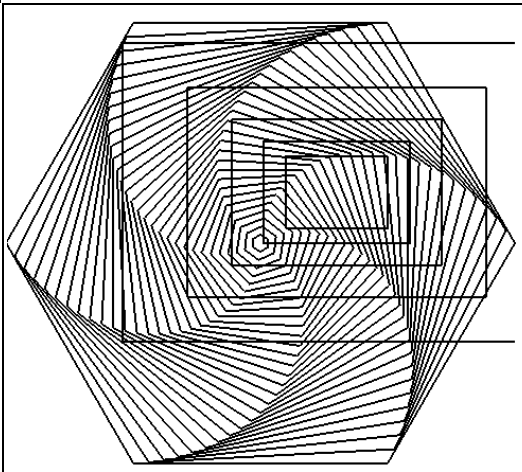


Figure 3.12. Zooming in on the swirl of hexagons. (file: fig3.12.bmp)

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

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

Figure 3.13. Making an animation.

Achieving a Smooth Animation.

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

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

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

- a). A steady display of the current figure;
- b). Instantaneous replacement of the current figure by the *finished* new figure;

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

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

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

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

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

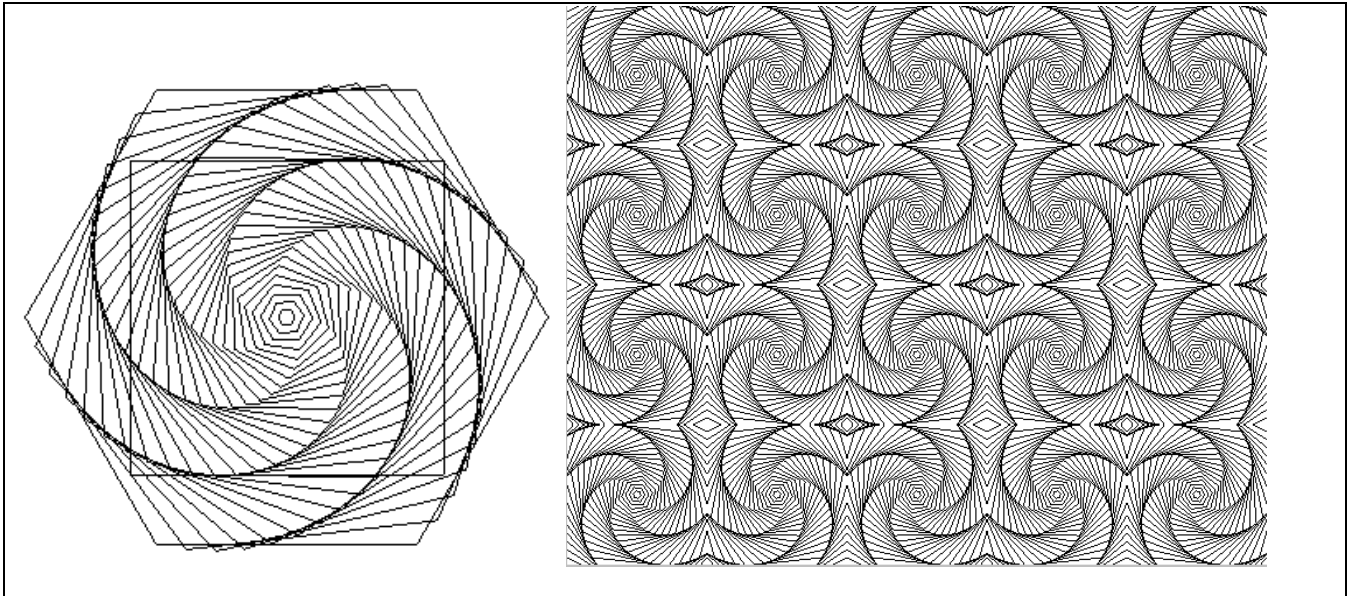


Figure 3.14. a). Whirling hexagons in a fixed window. b). A tiling formed using many viewports.

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

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

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

3.2.2. Setting the Window and Viewport Automatically.

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

Setting of the Window.

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

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

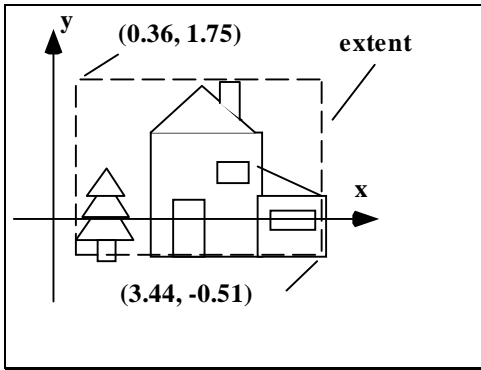


Figure 3.15. Using the Extent as the Window.

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

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

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

Pass 2: Execute the drawing routine again. Do the actual drawing.

Automatic setting of the viewport to Preserve Aspect Ratio.

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

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

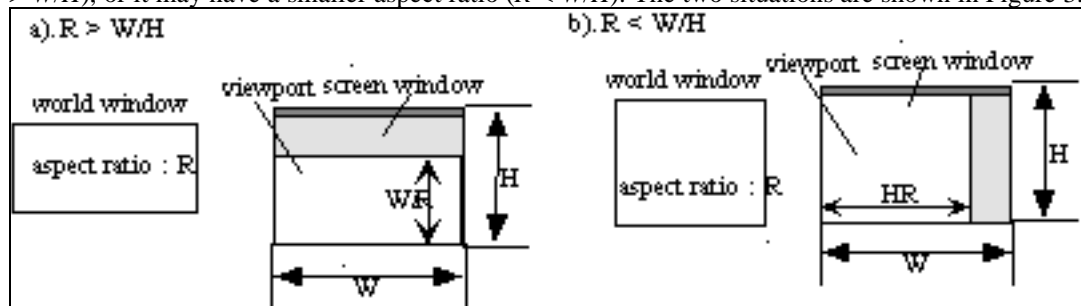


Figure 3.16. Possible aspect ratios for the world and screen windows.

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

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

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

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

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

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

Resizing the screen window, and the resize event.

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

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

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

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

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

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

Making a matched viewport.

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

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

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

Practice Exercises.

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

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

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

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

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

3.3. Clipping Lines.

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

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

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

3.3.1. Clipping a Line.

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

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

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

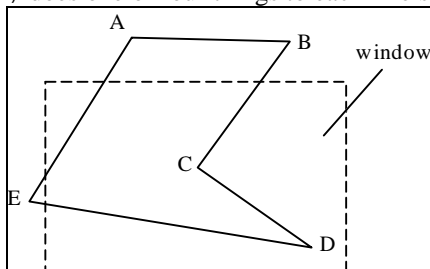


Figure 3.18. Clipping Lines at window boundaries.

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

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

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

3.3.2. The Cohen-Sutherland Clipping Algorithm

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

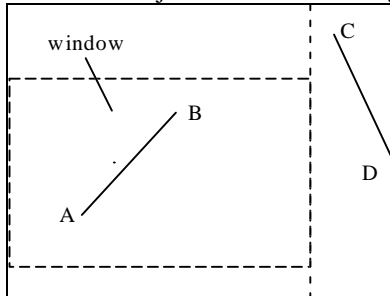


Figure 3.19. Trivial acceptance or rejection of a line segment.

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

Testing for a trivial accept or trivial reject.

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

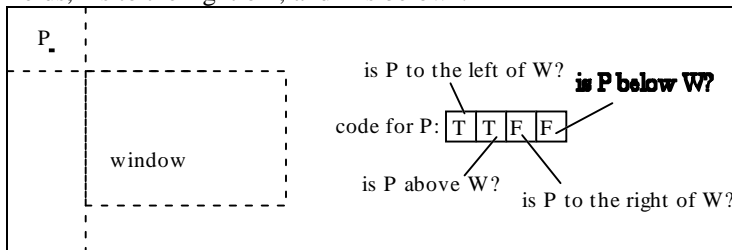


Figure 3.20. Encoding how point P is disposed with respect to the window.

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


TTF	FTF	FTF
TFF		FFT
TFT	FFT	FFT

Figure 3.21. Inside-outside codes for a point.

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

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

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

Chopping when there is neither trivial accept nor reject.

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

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

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

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

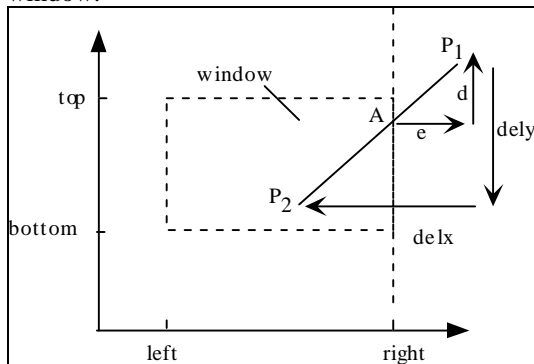


Figure 3.22. Clipping a segment against an edge.

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

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

where e is $p1.x - w.\text{right}$ and:

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

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

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

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

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

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

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

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

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

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

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

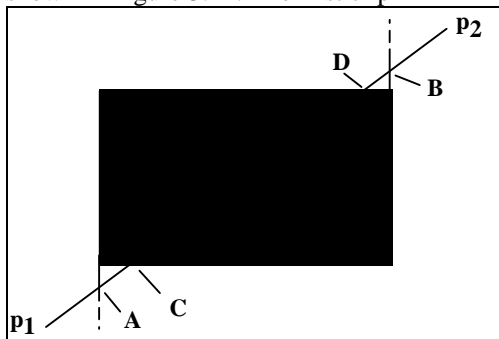


Figure 3.24. A segment that requires four clips.

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

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

3.3.2. Hand Simulation of `clipSegment()`.

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

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

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

3.4. Developing the Canvas Class.

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

Gustave Flaubert

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

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

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

3.4.1. Some useful Supporting Classes.

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

class Point2: A point having real coordinates.

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

```
class Point2
{
```

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

```

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

private:
    float x, y;
};

```

Note that values of `x` and `y` are cast to the type `GLfloat` when `glVertex2f()` is called. This is not likely unnecessary since the type `GLfloat` is defined on most systems as `float` anyway.

class IntRect: An aligned rectangle with integer coordinates.

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

```

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

```

class RealRect: An aligned rectangle with real coordinates.

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

```

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

```

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

3.4.2. Declaration of Class Canvas.

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

```

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

```


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

3.4.3. Implementation of Class Canvas.

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

[illegible]Figure 3.27. The constructor for *Canvas* – OpenGL version.

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

[illegible]

```

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

```

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

Practice Exercises.

3.4.2. Flesh out each of the member functions:

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

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

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

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

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

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

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

3.5. Relative Drawing.

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

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

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

```

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

```

```

}

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

```

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

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

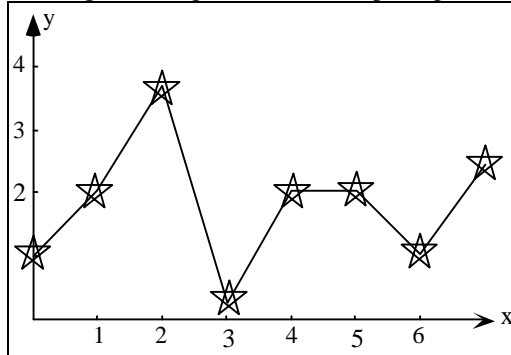


Figure 3.30. Placing markers for emphasis.

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

```

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

```

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

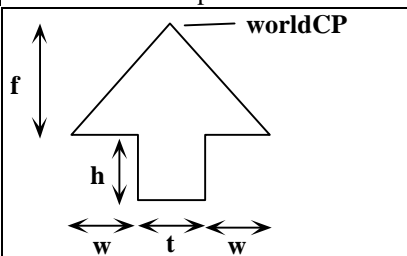


Figure 3.31. Model of an arrow.

```

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

```



```

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

```

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

3.5.2. Turtle Graphics.

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

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

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

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

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

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

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

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

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

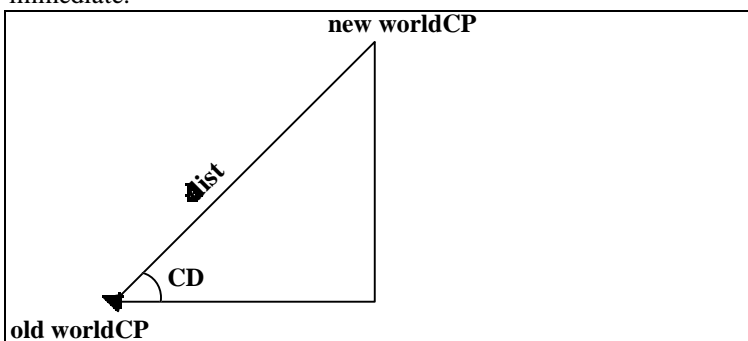


Figure 3.33. Effect of the `forward()` routine.

```

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

```

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

```

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

```

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

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

```

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

```

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

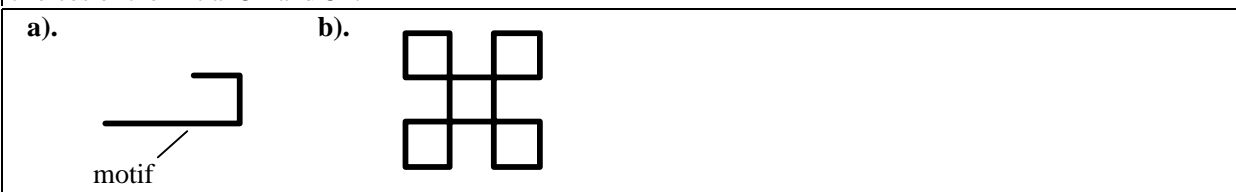


Figure 3.34. Building a figure out of several turtle motions.

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

```

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

```

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

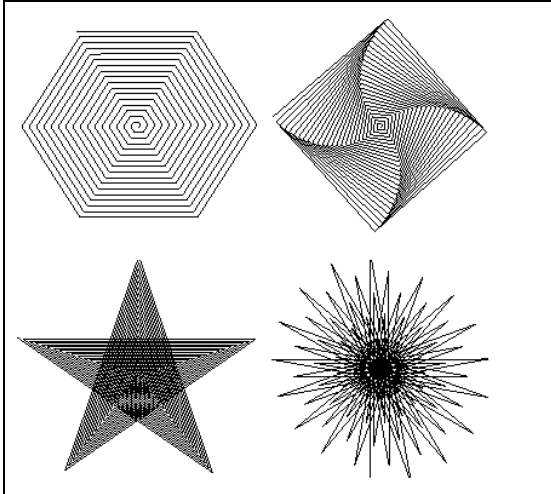


Figure 3.35. Examples of polypspirals. Angles are: a). 60, b). 89.5, c). -144, d). 170.

Practice Exercises.

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

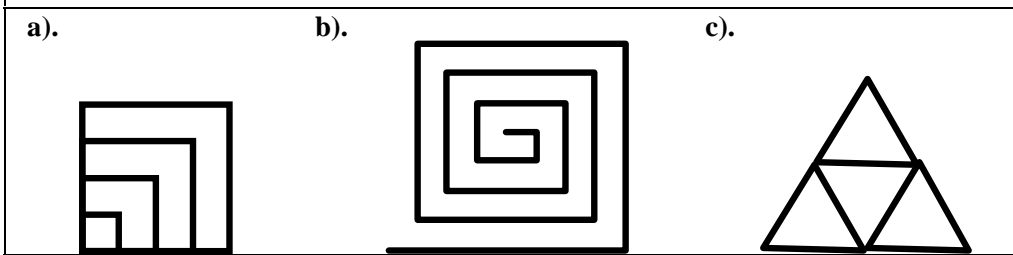


Figure 3.36. Other Simple Turtle Figures.

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

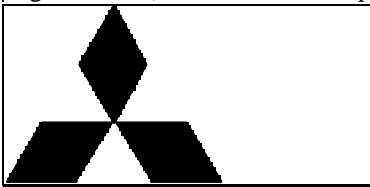


Figure 3.37. A famous logo.

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

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

What does the following sequence of commands produce?

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

3.5.4. Drawing Meanders. A **meander**⁷ is a pattern like that in Figure 3.38a, often made up of a continuous line meandering along some path. One frequently sees meanders on Greek vases, Chinese plates, or floor tilings from various countries. The motif for the meander here is shown in Figure 3.38b. After each motif is drawn the turtle is turned (how much?) to prepare it for drawing the next motif.

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



Figure 3.38. Example of a meander.

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

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

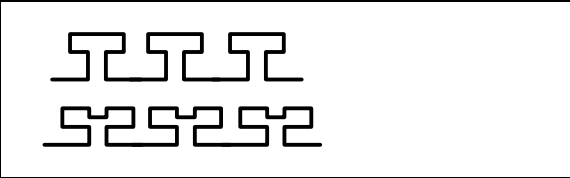


Figure 3.39. Additional figures for meanders.

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

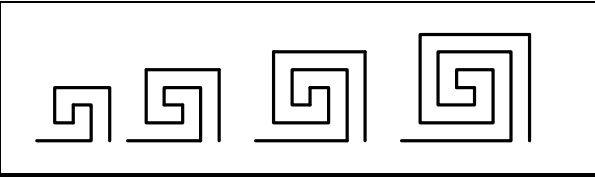


Figure 3.40. Hierarchy of meander motifs.

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

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

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

3.6. Figures based on Regular Polygons.

To generalize is to be an idiot.
William Blake

“Bees...by virtue of certain geometrical forethought...know that the hexagon is greater than the square and triangle, and will hold more honey for the same expenditure of material.”
Pappus of Alexandria

The regular polygons form a large and important family of shapes, often encountered in computer graphics. We need efficient ways to draw them. In this section we examine how to do this, and how to create a number of figures that are variations of the regular polygon.

3.6.1. The Regular Polygons.

First recall the definition of a regular polygon:

Definition: A polygon is **regular** if it is simple, if all its sides have equal lengths, and if adjacent sides meet at equal interior angles.

As discussed in Chapter 1, a polygon is **simple** if no two of its edges cross each other (more precisely: only adjacent edges can touch, and only at their shared endpoint). We give the name **n-gon** to a regular polygon having n sides. Familiar examples are the 4-gon (a square), a 5-gon (a regular pentagon), 8-gon (a regular octagon), and so on. A 3-gon is an equilateral triangle. Figure 3.41 shows various examples. If the number of sides of an n -gon is large the polygon approximates a circle in appearance. In fact this is used later as one way to implement the drawing of a circle.

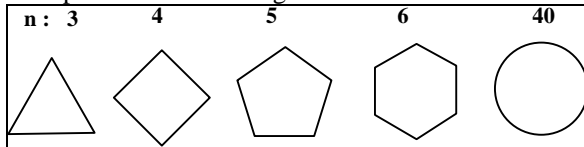


Figure 3.41. Examples of n -gons.

The vertices of an n -gon lie on a circle, the so-called “parent circle” of the n -gon, and their locations are easily calculated. The case of the hexagon is shown in Figure 3.42 where the vertices lie equispaced every 60° around the circle. The parent circle of radius R (not shown) is centered at the origin, and the first vertex P_0 has been placed on the positive x -axis. The other vertices follow accordingly, as $P_i = (R \cos(i \cdot a), R \sin(i \cdot a))$, for $i = 1, \dots, 5$, where a is $2\pi/6$ radians. Similarly, the vertices of the general n -gon lie at:

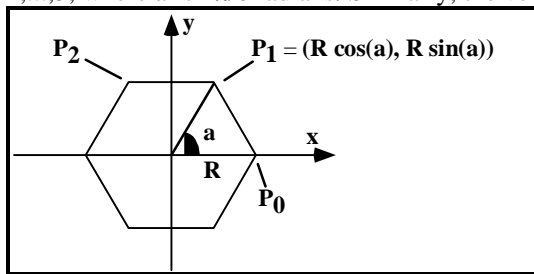


Figure 3.42. Finding the vertices of an 6-gon.

$$P_i = (R \cos(2\pi i / n), R \sin(2\pi i / n)), \text{ for } i = 0, \dots, n-1 \quad (3.6)$$

It's easy to modify this n -gon. To center it at position (cx, cy) we need only add cx and cy to the x - and y -coordinates, respectively. To scale it by factor S we need only multiply R by S . To rotate through angle A we need only add A to the arguments of $\cos()$ and $\sin()$. More general methods for performing geometrical transformations are discussed in Chapter 6.

It is simple to implement a routine that draws an n -gon, as shown in Figure 3.43. The n -gon is drawn centered at (cx, cy) , with radius `radius`, and is rotated through `rotAngle` degrees.

```
void ngon(int n, float cx, float cy, float radius, float rotAngle)
{
    // assumes global Canvas object, cvs
    if(n < 3) return; // bad number of sides
    double angle = rotAngle * 3.14159265 / 180; // initial angle
    double angleInc = 2 * 3.14159265 / n; // angle increment
    cvs.moveTo(radius + cx, cy);
    for(int k = 0; k < n; k++) // repeat n times
    {
        angle += angleInc;
        cvs.lineTo(radius * cos(angle) + cx, radius * sin(angle) + cy);
    }
}
```

Figure 3.43. Building an n -gon in memory.

Example 3.6.1: A Turtle-driven n -gon. It is also simple to draw an n -gon using turtlegraphics. Figure 3.44 shows how to draw a regular hexagon. The initial position and direction of the turtle is indicated by the small triangle. The turtle simply goes forward six times, making a CCW turn of 60 degrees between each move:

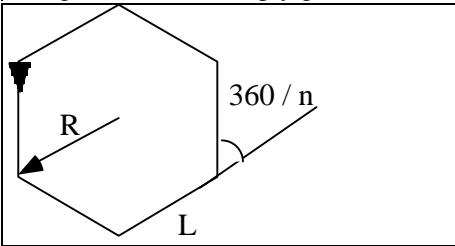


Figure 3.44. Drawing a hexagon.

```
for (i = 0; i < 6; i++)
{
    cvs.forward(L, 1);
    cvs.turn(60);
}
```

One vertex is situated at the initial CP , and both CP and CD are left unchanged by the process. Drawing the general n -gon, and some variations of it, is discussed in the exercises.

3.6.2. Variations on n -gons.

Interesting variations based on the vertices of an n -gon can also be drawn. The n -gon vertices may be connected in various ways to produce a variety of figures, as suggested in Figure 3.45. The standard n -gon is drawn in Figure 3.45a by connecting adjacent vertices, but Figure 3.45b shows a **stellation** (or star-like figure) formed by connecting every other vertex. And Figure 3.45c shows the interesting **rosette**, formed by connecting each vertex to every other vertex. We discuss the rosette next. Other figures are described in the exercises.

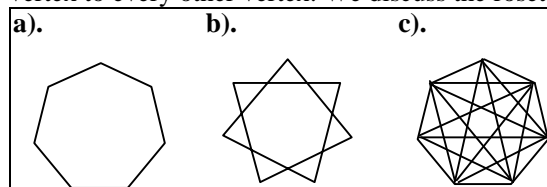


Figure 3.45. A 7-gon and its offspring. a). the 7-gon, b). a stellation, c). a “7-rosette”.

Example 3.6.2. The rosette, and the Golden 5-rosette.

The **rosette** is an n -gon with each vertex joined to every other vertex. Figure 3.46 shows 5-, 11-, and 17-rosettes. A rosette is sometimes used as a test pattern for computer graphics devices. Its orderly shape readily reveals any distortions, and the resolution of the device can be determined by noting the amount of “crowding” and blurring exhibited by the bundle of lines that meet at each vertex.

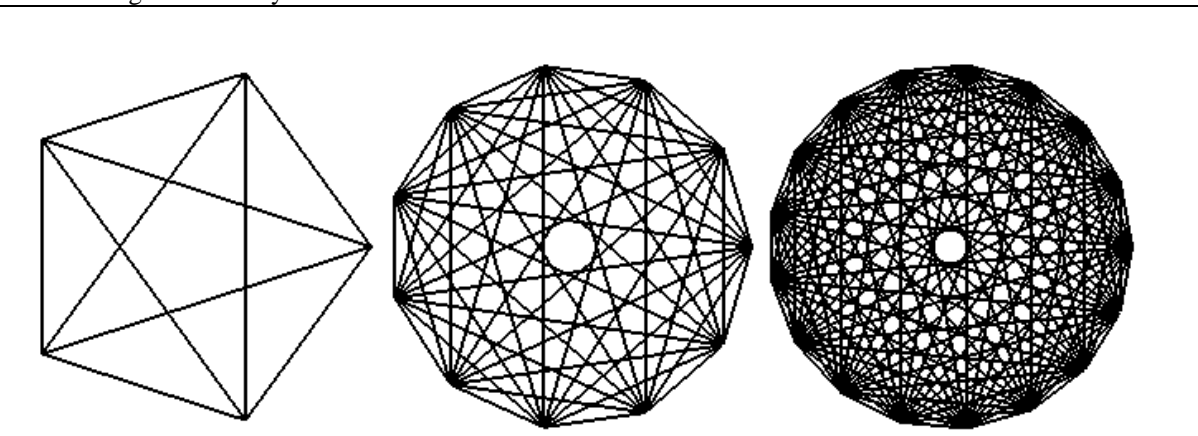


Figure 3.46. The 5-, 11-, and 17-rosettes.

Rosettes are easy to draw: simply connect every vertex to every other. In pseudocode this looks like

```
void Rosette(int N, float radius)
{
    Point2 pt[big enough value for largest rosette];
    generate the vertices pt[0], . . . , pt[N-1], as in Figure 3.43
    for(int i = 0; i < N - 1; i++)
        for(int j = i + 1; j < N ; j++)
        {
            cvs.moveTo(pt[i]); // connect all the vertices
            cvs.lineTo(pt[j]);
        }
}
```

The 5-rosette is particularly interesting because it embodies many instances of the golden ratio ϕ (recall Chapter 2). Figure 3.47a shows a 5-rosette, which is made up of an outer pentagon and an inner pentagram. The Greeks saw a mystical significance in this figure. Its segments have an interesting relationship: Each segment is ϕ times longer than the next smaller one (see the exercises). Also, because the edges of the star pentagram form an inner pentagon, an infinite regression of pentagrams is possible, as shown in Figure 3.47b.

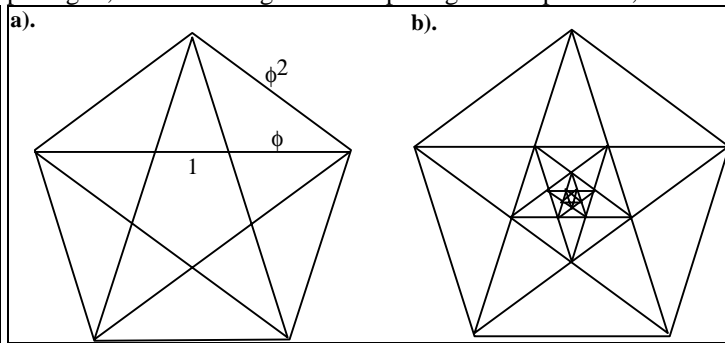


Figure 3.47. 5-rosette and Infinite regressions - pentagons and pentagrams.

Example 3.6.3. Figures based on two concentric n-gons.

Figures 3.48 shows some shapes built upon two concentric parent circles, the outer of radius R , and the inner of radius fR for some fraction f . Each figure uses a variation of an n -gon whose radius alternates between the inner and outer radii. Parts a) and b) show familiar company logos based on 6-gons and 10-gons. Part c) is based on the 14-gon, and part d) shows the inner circle explicitly.

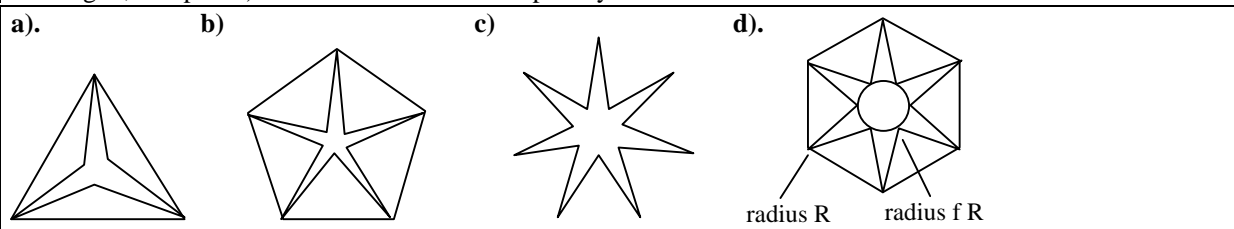


Figure 3.48. A family of Famous Logos.

Practice Exercises.

3.6.1. Stellations and rosettes. The pentagram is drawn by connecting “every other” point as one traverses around a 5-gon. Extend this to an arbitrary odd-valued n -gon and develop a routine that draws this so-called “stellated” polygon. Can it be done with a single initial `moveTo()` followed only by `lineTo()`’s (that is, without “lifting the pen”)? What happens if n is even?

3.6.2. How Many Edges in an N -rosette? Show that a rosette based on an N -gon, an N -rosette, has $N(N - 1) / 2$ edges. This is the same as the number of “clinks” one hears when N people are seated around a table and everybody clinks glasses with everyone else.

3.6.3. Prime Rosettes. If a rosette has a prime number N of sides, it can be drawn without “lifting the pen,” that is, by using only `lineTo()`. Start at vertex v_0 and draw to each of the others in turn: v_1, v_2, v_3, \dots until v_0 is again reached and the polygon is drawn. Then go around again drawing lines, but skip a vertex each time – that is, increment the index by 2 – thereby drawing to v_2, v_4, \dots, v_0 . This will require going around twice to arrive back at v_0 . (A *modulo* operation is performed on the indices so that their values remain between 0 and $N-1$.)

Then repeat this, incrementing by 3: $v_3, v_6, v_0, \dots, v_0$. Each repeat draws exactly N lines. Because there are $N(N-1)/2$ lines in all, the process repeats $(N-1)/2$ times. Because the number of vertices is a prime, no pattern is ever repeated until the drawing is complete. Develop and test a routine that draws prime rosettes in this way.

3.6.4. Rosettes with an odd number of sides. If n is prime we know the n -rosette can be drawn as a single polyline without “lifting the pen”. It can also be drawn as a single polyline for any *odd* value of n . Devise a method that does this.

3.6.5. The Geometry of the Star Pentagram. Show that the length of each segment in the 5-rosette stands in the golden ratio to that of the next smaller one. One way to tackle this is to show that the triangles of the star pentagram are “golden triangles” with an inner angle of $\pi/5$ radians. Show that $2 * \cos(\pi/5) = \phi$ and $2 * \cos(2\pi/5) = 1/\phi$. Another approach uses only two families of similar triangles in the pentagram and the relation $\phi^3 = 2\phi + 1$ satisfied by ϕ .

3.6.6. Erecting Triangles on n -gon legs. Write a routine that draws figures like the logo in part a of Figure 3.48 for any value of f , positive or negative. What is a reasonable geometric interpretation of negative f ?

3.6.7. Drawing the Star with Relative Moves and Draws. Write a routine to draw a pentagram that uses only relative moves and draws, centering the star at the *CP*.

3.6.8. Draw a pattern of stars. Write a routine to draw the pattern of 21 stars shown in Figure 3.49. The small stars are positioned at the vertices of an n -gon.

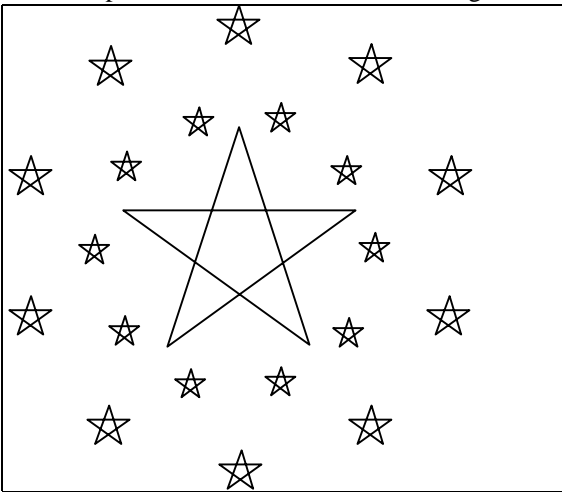


Figure 3.49. A star pattern.

3.6.9. New points on the “7-gram”. Figure 3.50 shows a figure formed from the 7 points of a 7-gon, centered at the origin. The first point lies at $(R, 0)$. Instead of connecting consecutive points around the 7-gon, two intermediary points are skipped. (This is a form of “stellation” of an n -gon.) Find the coordinates of point P , where two of the edges intersect.

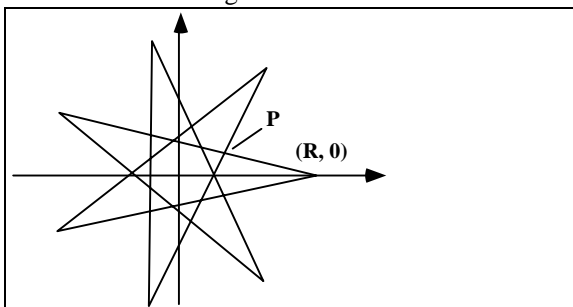


Figure 3.50. A “7-gram”.

3.6.10. Turtle drawings of the n -gon. Write `turtleNgon(int numSides, float length)` that uses `turtlegraphics` to draw an n -gon with `numSides` sides and a side of length `length`.

3.6.11. Polygons sharing an edge. Write a routine that draws n -gon's, for $n = 3, \dots, 12$, on a common edge, as in Figure 3.51.

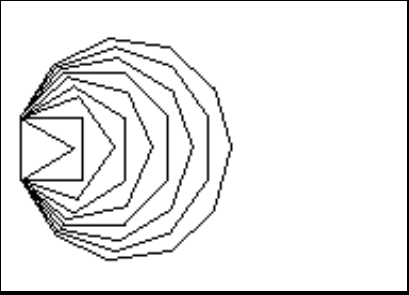


Figure 3.51. N -gons sharing a common edge.

3.6.12. A more elaborate figure. Write a routine that draws the shape in Figure 3.52 by drawing repeated hexagons rotated relative to one another.

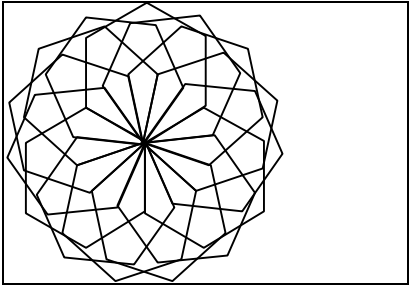


Figure 3.52. Repeated use of turtle commands.

3.6.13. Drawing a famous logo. The esteemed logo shown in Figure 3.53 consists of three instances of a motif, rotated a certain amount with respect to each other. Show a routine that draws this shape using `turtlegraphics`.

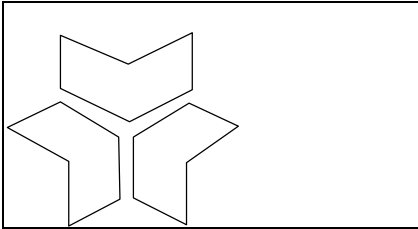


Figure 3.53. Logo of the University of Massachusetts.

3.6.14. Rotating Pentagons: animation. Figure 3.54 shows a pentagram oriented with some angle of rotation within a pentagon, with corresponding vertices joined together. Write a program that “animates” this figure. The configuration is drawn using some initial angle A of rotation for the pentagram. After a short pause it is erased and then redrawn but with a slightly larger angle A . This process repeats until a key is pressed.

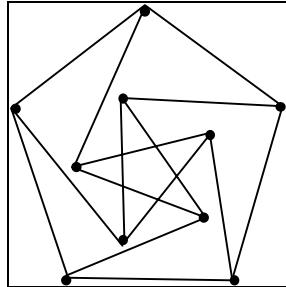


Figure 3.54. Rotating penta-things.

3.7. Drawing Circles and Arcs.

Drawing a circle is equivalent to drawing an n -gon that has a large number of vertices. The n -gon resembles a circle (unless it is scrutinized too closely). The routine `drawCircle()` shown in Figure 3.55 draws a 50-sided n -gon, by simply passing its parameters on to `ngon()`. It would be more efficient to write `drawCircle()` from scratch, basing it on the code of Figure 3.43.

```
void drawCircle(Point2 center, float radius)
{
    const int numVerts = 50;    // use larger for a better circle
    ngon(numVerts, center.getX(), center.getY(), radius, 0);
}
```

Figure 3.55. Drawing a circle based on an 50-gon.

3.7.1. Drawing Arcs.

Many figures in art, architecture, and science involve arcs of circles placed in pleasing or significant arrangements. An arc is conveniently described by the position of the center, c , and radius, R , of its “parent” circle, along with its beginning angle a and the angle b through which it “sweeps”. Figure 3.56 shows such an arc. We assume that if b is positive the arc sweeps in a CCW direction from a . If b is negative it sweeps in a CW fashion. A circle is a special case of an arc, with a sweep of 360° .

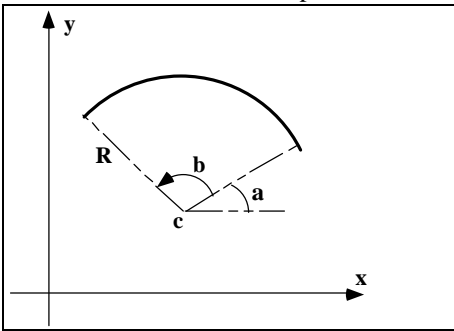


Figure 3.56. Defining an arc.

We want a routine, `drawArc()`, that draws an arc of a circle. The function shown in Figure 3.57 approximates the arc by part of an n -gon, using `moveTo()` and `lineTo()`. Successive points along the arc are found by computing a $\cos()$ and $\sin()$ term each time through the main loop. If sweep is negative the angle automatically decreases each time through.

```
void drawArc(Point2 center, float radius, float startAngle, float sweep)
{
    // startAngle and sweep are in degrees
    const int n = 30; // number of intermediate segments in arc
    float angle = startAngle * 3.14159265 / 180; // initial angle in radians
    float angleInc = sweep * 3.14159265 / (180 * n); // angle increment
    float cx = center.getX(), cy = center.getY();
    cvs.moveTo(cx + radius * cos(angle), cy + radius * sin(angle));
    for(int k = 1; k < n; k++, angle += angleInc)
        cvs.lineTo(cx + radius * cos(angle), cy + radius * sin(angle));
}
```

Figure 3.57. Drawing an arc of a circle.

The *CP* is left at the last point on the arc. (In some cases one may wish to omit the initial `moveTo()` to the first point on the arc, so that the arc is connected to whatever shape was being drawn when `drawArc()` is called.)

A much faster arc drawing routine is developed in Chapter 5 that avoids the repetitive calculation of so many $\sin()$ and $\cos()$ functions. It may be used freely in place of the procedure here.

With `drawArc()` in hand it is a simple matter to build the routine `drawCircle(Point2 center, float radius)` that draws an entire circle (how?).

The routine `drawCircle()` is called by specifying a center and radius, but there are other ways to describe a circle, which have important applications in interactive graphics and computer-aided design. Two familiar ones are:

- 1). **The center is given, along with a point on the circle.** Here `drawCircle()` can be used as soon as the radius is known. If c is the center and p is the given point on the circle, the radius is simply the distance from c to p , found using the usual Pythagorean Theorem.
- 2). **Three points are given through which the circle must pass.** It is known that a unique circle passes through any three points that don't lie in a straight line. Finding the center and radius of this circle is discussed in Chapter 4.

Example 3.7.1. Blending Arcs together. More complex shapes can be obtained by using parts of two circles that are tangent to one another. Figure 3.58 illustrates the underlying principle. The two circles are tangent at point A , where they share tangent line L . Because of this the two arcs shown by the thick curve “blend” together seamlessly at A with no visible break or corner. Similarly the arc of a circle blends smoothly with any tangent line, as at point B .

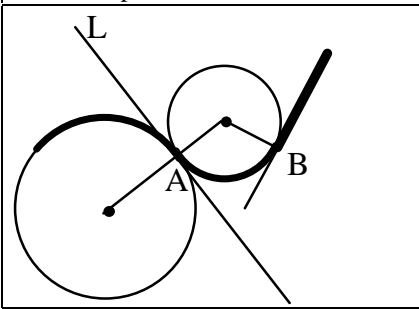


Figure 3.58. Blending arcs using tangent circles.

Practice Exercises.

3.7.1. Circle Figures in Philosophy. In Chinese philosophy and religion the two principles of yin and yang interact to influence all creatures' destinies. Figure 3.59 shows the exquisite yin–yang symbol. The dark portion, yin, represents the feminine aspect, and the light portion, yang, represents the masculine. Describe in detail the geometry of this symbol, supposing it is centered in some coordinate system.

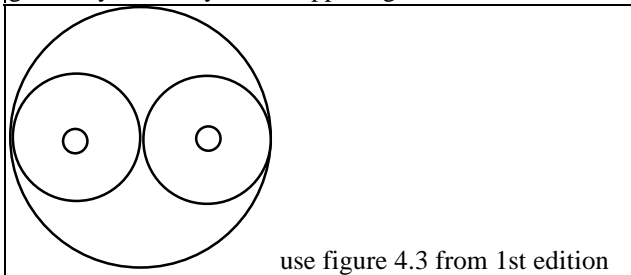


Figure 3.59. The yin-yang symbol.

3.7.2. The Seven Pennies. Describe the configuration shown in Figure 3.60 in which six pennies fit snugly around a center penny. Use symmetry arguments to explain why the fit is exact; that is, why each of the outer pennies *exactly* touches its three neighbors.

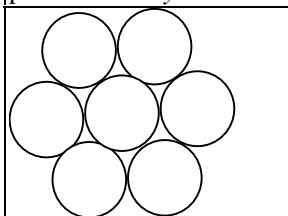


Figure 3.60. The seven circles.

3.7.3. A famous logo. Figure 3.61 shows a well-known automobile logo. It is formed by erecting triangles inside an equilateral triangle, but the outer triangle is replaced by two concentric circles. After determining the “proper” positions for the three inner points, write a routine to draw this logo.

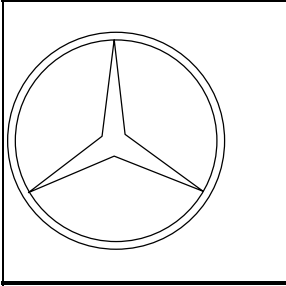


Figure 3.61. A famous logo.

3.7.4. Drawing clocks and such. Circles and lines may be made tangent in a variety of ways to create pleasing smooth curves, as in Figure 3.62a. Figure 3.62b shows the underlying lines and circles. Write a routine that draws this basic clock shape

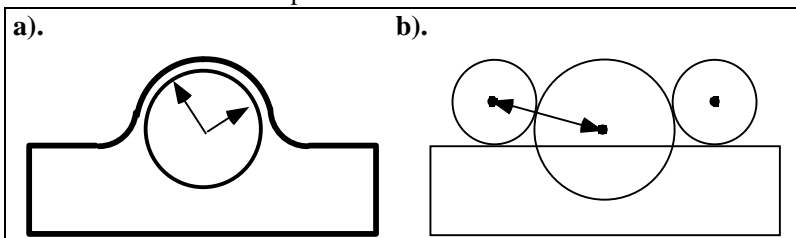


Figure 3.62. Blending arcs to form smooth curves.

3.7.5. Drawing rounded rectangles. Figure 3.63 shows an aligned rectangle with rounded corners. The rectangle has width W and aspect ratio R , and each corner is described by a quarter-circle of radius $r = g W$ for some fraction g . Write a routine `drawRoundRect(float W, float R, float g)` that draws this rectangle centered at the CP . The CP should be left at the center when the routine exits.

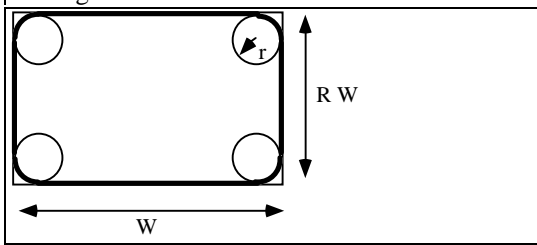


Figure 3.63. A rounded rectangle.

3.7.6. Shapes involving arcs. Figure 3.64 shows two interesting shapes that involve circles or arcs. One is similar to the Atomic Energy Commission symbol (How does it differ from the standard symbol?). Write and test two routines that draw these figures.

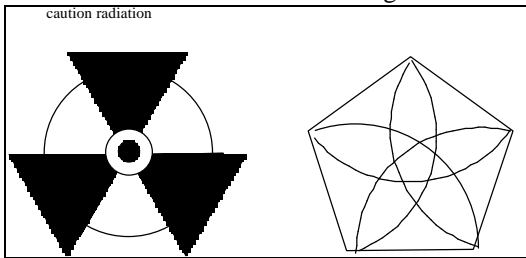


Figure 3.64. Shapes based on arcs.

3.7.7. A tear drop. A “tear drop” shape that is used in many ornamental figures is shown in Figure 3.65a. As shown in part b) it consists of a circle of given radius R snugged down into an angle ϕ . What are the coordinates of the circle’s center C for a given R and ϕ ? What are the initial angle of the arc, and its sweep? Develop a routine to draw a tear drop at any position and in any orientation.

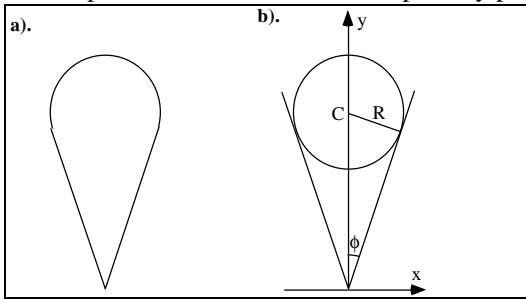


Figure 3.65. The tear drop and its construction.

3.7.8. Drawing Patterns of Tear Drops. Figure 3.66 show some uses of the tear drop. Write a routine that draws each of them.

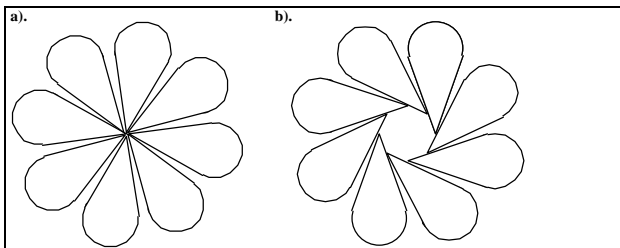


Figure 3.66. Some figures based on the tear drop.

3.7.9. Pie Charts. A sector is closely related to an arc: each end of the arc is connected to the center of the circle. The familiar pie chart is formed by drawing a number of sectors. A typical example is shown in Figure 3.67. Pie charts are used to illustrate how a whole is divided into parts, as when a pie is split up and distributed. The eye quickly grasps how big each “slice” is relative to the others. Often one or more of the slices is “exploded” away from the pack as well, as shown in the figure. Sectors that are exploded are simply shifted slightly away from the center of the pie chart in the proper direction

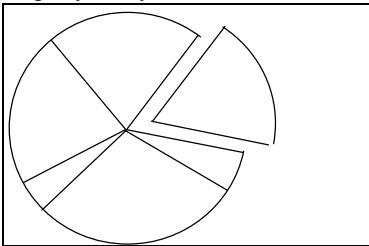


Figure 3.67. A pie chart.

To draw a pie chart we must know the relative sizes of the slices. Write and test a routine that accepts data from the user and draws the corresponding pie chart. The user enters the fraction of the pie each slice represents, along with an ‘e’ if the slice is to be drawn exploded, or an ‘n’ otherwise.

3.8. Using the Parametric form for a curve.

There are two principal ways to describe the shape of a curved line: implicitly and parametrically. The **implicit form** describes a curve by a function $F(x, y)$ that provides a relationship between the x and y coordinates: the point (x, y) lies on the curve if and only if it satisfies:

$$F(x, y) = 0 \quad \text{condition for } (x, y) \text{ to lie on the curve} \quad (3.7)$$

For example, the straight line through points A and B has implicit form:

$$F(x, y) = (y - A_y)(B_x - A_x) - (x - A_x)(B_y - A_y) \quad (3.8)$$

and the circle with radius R centered at the origin has implicit form:

$$F(x, y) = x^2 + y^2 - R^2 \quad (3.9)$$

A benefit of using the implicit form is that you can easily test whether a given point lies on the curve: simply evaluate $F(x, y)$ at the point in question. For certain classes of curves it is meaningful to speak of an inside and an outside of the curve, in which case $F(x, y)$ is also called the **inside-outside function**, with the understanding that

$$\begin{aligned} F(x, y) &= 0 && \text{for all } (x, y) \text{ on the curve} \\ F(x, y) &> 0 && \text{for all } (x, y) \text{ outside the curve} \\ F(x, y) &< 0 && \text{for all } (x, y) \text{ inside the curve} \end{aligned} \quad (3.10)$$

(Is $F(x, y)$ of Equation 3.9 a legitimate inside-outside function for the circle?)

Some curves are **single-valued** in x , in which case there is a function $g(\cdot)$ such that all points on the curve satisfy $y = g(x)$. For such curves the implicit form may be written $F(x, y) = y - g(x)$. (What is $g(\cdot)$ for the line of Equation 3.8?) Other curves are single-valued in y , (so there is a function $h(\cdot)$ such that points on the curve satisfy $x = h(y)$). And some curves are not single-valued at all: $F(x, y) = 0$ cannot be rearranged into either of the forms $y = g(x)$ nor $x = h(y)$. The circle, for instance, can be expressed as:

$$y = \pm\sqrt{R^2 - x^2} \quad (3.11)$$

but here there are two functions, not one.

3.8.1. Parametric Forms for Curves.

A parametric form for a curve produces different points on the curve based on the value of a parameter. Parametric forms can be developed for a wide variety of curves, and they have much to recommend them, particularly when one wants to draw or analyze the curve. A parametric form suggests the movement of a point through time, which we can translate into the motion of a pen as it sweeps out the curve. The path of the particle traveling along the curve is fixed by two functions, $x(\cdot)$ and $y(\cdot)$, and we speak of $(x(t), y(t))$ as the **position** of the particle at time t . The curve itself is the totality of points “visited” by the particle as t varies over some interval. For any curve, therefore, if we can dream up suitable functions $x(\cdot)$ and $y(\cdot)$ they will represent the curve concisely and precisely.

The familiar Etch-a-Sketch⁸ shown in Figure 3.68 provides a vivid analogy. As knobs are turned, a stylus hidden in the box scrapes a thin visible line across the screen. One knob controls the horizontal position, and the other directs the vertical position of the stylus. If the knobs are turned in accordance with $x(t)$ and $y(t)$, the parametric curve is swept out. (Complex curves require substantial manual dexterity.)

1st Ed. Figure 7.11

Figure 3.68. Etch-a-Sketch drawings of parametric curves. (Drawing by Suzanne Casiello.)

Examples: The line and the ellipse.

The straight line of Equation 3.8 passes through points A and B . We choose a parametric form that visits A at $t = 0$ and B at $t = 1$, obtaining:

$$\begin{aligned} x(t) &= A_x + (B_x - A_x)t \\ y(t) &= A_y + (B_y - A_y)t \end{aligned} \quad (3.12)$$

⁸Etch-a-Sketch is a trademark of Ohio Art.

Thus the point $P(t) = (x(t), y(t))$ “sweeps through” all of the points on the line between A and B as t varies from 0 to 1 (check this out).

Another classic example is the **ellipse**, a slight generalization of the circle. It is described parametrically by

$$\begin{aligned} x(t) &= W \cos(t) \\ y(t) &= H \sin(t) \end{aligned} \quad , \text{ for } 0 \leq t \leq 2\pi. \quad (3.13)$$

Here W is the “half-width”, and H the “half-height” of the ellipse. Some of the geometric properties of the ellipse are explored in the exercises. When W and H are equal the ellipse is a circle of radius W . Figure 3.69 shows this ellipse, along with the component functions $x(\cdot)$ and $y(\cdot)$.

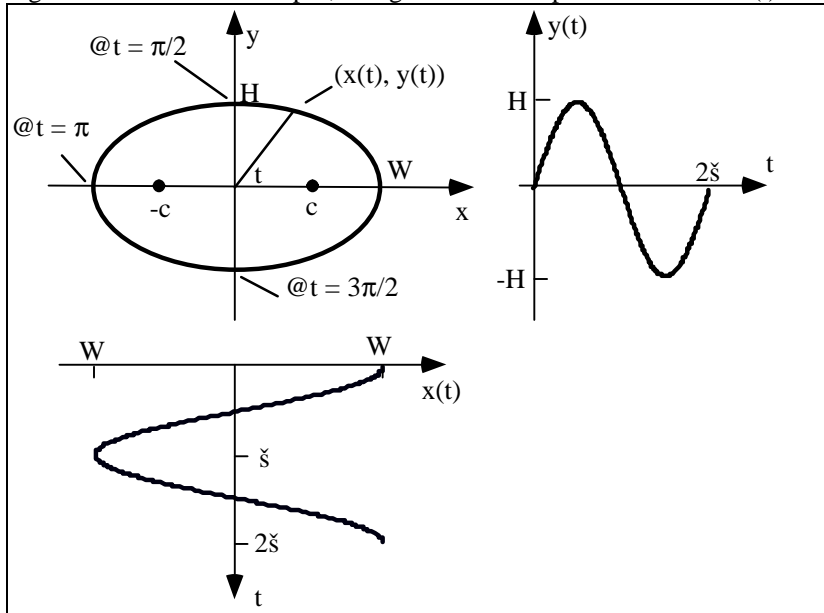


Figure 3.69. An ellipse described parametrically.

As t varies from 0 to 2π the point $P(t) = (x(t), y(t))$ moves once around the ellipse starting (and finishing) at $(W, 0)$. The figure shows where the point is located at various “times” t . It is useful to visualize drawing the ellipse on an Etch-a-Sketch. The knobs are turned back and forth in an undulating pattern, one mimicking $W \cos(t)$ and the other $H \sin(t)$. (This is surprisingly difficult to do manually.)

• Finding an implicit form from a parametric form - “implicitization”.

Suppose we want to check that the parametric form in Equation 3.13 truly represents an ellipse. How do we find the implicit form from the parametric form? The basic step is to combine the two equations for $x(t)$ and $y(t)$ to somehow eliminate the variable t . This provides a relationship that must hold for *all* t . It isn’t always easy to see how to do this — there are no simple guidelines that apply for all parametric forms. For the ellipse, however, square both x/W and y/H and use the well-known fact $\cos(t)^2 + \sin(t)^2 = 1$ to obtain the familiar equation for an ellipse:

$$\left(\frac{x}{W}\right)^2 + \left(\frac{y}{H}\right)^2 = 1 \quad (3.14)$$

The following exercises explore properties of the ellipse and other “classical curves”. They develop useful facts about the **conic sections**, which will be used later. Read them over, even if you don’t stop to solve each one.

| Practice Exercises

3.8.1. On the geometry of the Ellipse. An ellipse is the set of all points for which the sum of the distances to two foci is constant. The point $(c, 0)$ shown in Figure 3.69 forms one “focus”, and $(-c, 0)$ forms the other. Show that H , W , and c are related by: $W^2 = H^2 + c^2$.

3.8.2. How eccentric. The **eccentricity**, $e = c / W$, of an ellipse is a measure of how non circular the ellipse is, being 0 for a true circle. As interesting examples, the planets in our solar system have very nearly circular orbits, with e ranging from 1/143 (Venus) to 1/4 (Pluto). Earth’s orbit exhibits $e = 1/60$. As the eccentricity of an ellipse approaches 1, the ellipse flattens into a straight line. But e has to get very close to 1 before this happens. What is the ratio H / W of height to width for an ellipse that has $e = 0.99$?

3.8..3. The other Conic Sections.

The ellipse is one of the three conic sections, which are curves formed by cutting (“sectioning”) a circular cone with a plane, as shown in Figure 3.70. The conic sections are:

- ellipse: if the plane cuts one “nappe” of the cone;
- hyperbola: if the plane cuts both nappes
- parabola: if the plane is parallel to the side of the cone;

Figure 3.70. The classical conic sections.

The parabola and hyperbola have interesting and useful geometric properties. Both of them have simple implicit and parametric representations.

Show that the following parametric representations are consistent with the implicit forms given:

• **Parabola:** Implicit form: $y^2 - 4ax = 0$

$$\begin{aligned} x(t) &= at^2 \\ y(t) &= 2at \end{aligned} \quad (3.15)$$

• **Hyperbola:** Implicit form: $(x/a)^2 - (y/b)^2 = 1$

$$\begin{aligned} x(t) &= a \sec(t) \\ y(t) &= b \tan(t) \end{aligned} \quad (3.16)$$

What range in the parameter t is used to sweep out this hyperbola? Note: A hyperbola is defined as the locus of all points for which the *difference* in its distances from two fixed foci is a constant. If the foci here are at $(-c, 0)$ and $(+c, 0)$, show that a and b must be related by $c^2 = a^2 + b^2$.

3.8.2. Drawing curves represented parametrically.

It is straightforward to draw a curve when its parametric representation is available. This is a major advantage of the parametric form over the implicit form. Suppose a curve C has the parametric representation $P(t) = (x(t), y(t))$ as t varies from 0 to T (see Figure 3.71a). We want to draw a good approximation to it, using only straight lines. Just take **samples** of $P(t)$ at closely spaced “instants”. A sequence $\{t_i\}$ of times are chosen, and for each t_i the position $P_i = P(t_i) = (x(t_i), y(t_i))$ of the curve is found. The curve $P(t)$ is then approximated by the polyline based on this sequence of points P_i , as shown in Figure 3.71b.

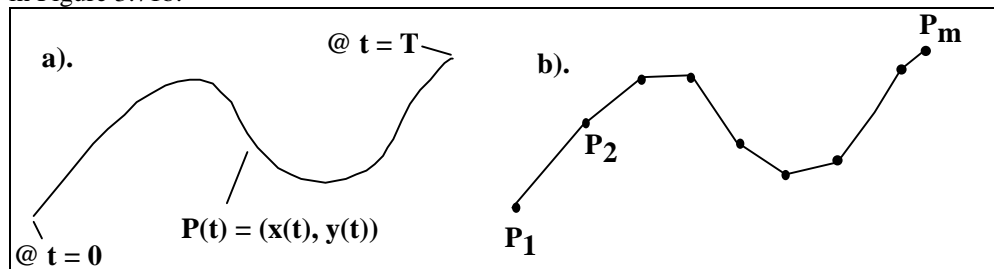


Figure 3.71. Approximating a curve by a polyline.

Figure 3.72 shows a code fragment that draws the curve $(x(t), y(t))$ when the desired array of sample times $t[i]$ is available.


```
// draw the curve (x(t), y(t)) using
// the array t[0],...,t[n-1] of "sample-times"

glBegin(GL_LINES);
    for(int i = 0; i < n; i++)
        glVertex2f((x(t[i]), y(t[i]));
glEnd();
```

Figure 3.72. Drawing the ellipse using points equispaced in t .

If the samples are spaced sufficiently close together, the eye will naturally blend together the line segments and will see a smooth curve. Samples must be closely spaced in t -intervals where the curve is “wiggling” rapidly, but may be placed less densely where the curve is undulating slowly. The required “closeness” or “quality” of the approximation depends on the situation.

Code can often be simplified if it is needed only for a specific curve. The ellipse in Equation 3.13 can be drawn using n equispaced values of t with:

```
#define TWOPI 2 * 3.14159265
glBegin(GL_LINES);
    for(double t = 0; t <= TWOPI; t += TWOPI/n)
        glVertex2f(W * cos(t), H * sin(t));
glEnd();
```

For drawing purposes, parametric forms circumvent all of the difficulties of implicit and explicit forms. Curves can be multi-valued, and they can self-intersect any number of times. Verticality presents no special problem: $x(t)$ simply becomes constant over some interval in t . Later we see that drawing curves that lie in 3D space is just as straightforward: three functions of t are used, and the point at t on the curve is $(x(t), y(t), z(t))$.

Practice Exercises.

3.8.4. An Example Curve. Compute and plot by hand the points that would be drawn by the fragment above for $W = 2$, $H = 1$, at the 5 values of $t = 2\pi i/9$, for $i = 0, 1, \dots, 4$.

3.8.5. Drawing a logo. A well-known logo consists of concentric circles and ellipses, as shown in Figure 3.73. Suppose you have a drawing tool: `drawEllipse(W, H, color)` that draws the ellipse of Equation 3.13 filled with color `color`. (Assume that as each color is drawn it completely obscures any previously drawn color.) Choose suitable dimensions for the ellipses in the logo and give the sequence of commands required to draw it.

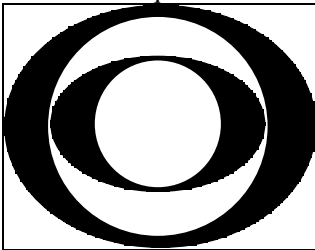


Figure 3.73. A familiar “eye” made of circles and ellipses.

Some specific examples of curves used in computer graphics will help to cement the ideas.

3.8.3. Superellipses

An excellent variation of the ellipse is the **superellipse**, a family of ellipse-like shapes that can produce good effects in many drawing situations. The implicit formula for the superellipse is

$$\left(\frac{x}{W}\right)^n + \left(\frac{y}{H}\right)^n = 1 \quad (3.17)$$

where n is a parameter called the *bulge*. Looking at the corresponding formula for the ellipse in Equation 3.14, the superellipse is seen to become an ellipse when $n = 2$. The superellipse has the following parametric representation:

$$\begin{aligned} x(t) &= W \cos(t) |\cos(t)|^{2/n-1} \\ y(t) &= H \sin(t) |\sin(t)|^{2/n-1} \end{aligned} \quad (3.18)$$

for $0 \leq t \leq 2\pi$. The exponent on the $\sin()$ and $\cos()$ is really $2/n$, but the peculiar form as shown is used to avoid trying to raise a negative number to a fractional power. A more precise version avoids this. Check that this form reduces nicely to the equation for the ellipse when $n = 2$. Also check that the parametric form for the superellipse is consistent with the implicit equation.

Figure 3.74a shows a family of **supercircles**, special cases of superellipses for which $W = H$. Figure 3.74b shows a scene composed entirely of superellipses, suggesting the range of shapes possible.

1st Ed. Figures 4.16 and 4.17 together

Figure 3.74. Family of supercircles. b). Scene composed of superellipses.

For $n > 1$ the bulge is outward, whereas for $n < 1$ it is inward. When $n = 1$, it becomes a square. (In Chapter 6 we shall look at three-dimensional “superquadrics,” surfaces that are sometimes used in CAD systems to model solid objects.)

Superellipses were first studied in 1818 by the French physicist Gabriel Lamé. More recently in 1959, the extraordinary inventor Piet Hein (best known as the originator of the Soma cube and the game Hex) was approached with the problem of designing a traffic circle in Stockholm. It had to fit inside a rectangle (with $W/H = 6/5$) determined by other roads, and had to permit smooth traffic flow as well as be pleasing to the eye. An ellipse proved to be too pointed at the ends for the best traffic patterns, and so Piet Hein sought a fatter curve with straighter sides and dreamed up the superellipse. He chose $n = 2.5$ as the most pleasing bulge. Stockholm quickly accepted the superellipse motif for its new center. The curves were “strangely satisfying, neither too rounded nor too orthogonal, a happy blend of elliptical and rectangular beauty” [Gardner75, p. 243]. Since that time, superellipse shapes have appeared in furniture, textile patterns, and even silverware. More can be found out about them in the references, especially in [Gardner75] and [Hill 79b].

The **superhyperbola** can also be defined [Barr81]. Just replace $\cos(t)$ by $\sec(t)$, and $\sin(t)$ by $\tan(t)$, in Equation 3.18. When $n = 2$, the familiar hyperbola is obtained. Figure 3.75 shows example superhyperbolas. As the bulge n increases beyond 2, the curve bulges out more and more, and as it decreases below 2, it bulges out less and less, becoming straight for $n = 1$ and pinching inward for $n < 1$.

1st Ed. Figure 9.14.

Figure 3.75. The superhyperbola family.

3.8.4. Polar Coordinate Shapes

Polar coordinates may be used to represent many interesting curves. As shown in Figure 3.76, each point on the curve is represented by an angle θ and a radial distance r . If r and θ are each made a function of t , then as t varies the curve $(r(t), \theta(t))$ is swept out. Of course this curve also has the Cartesian representation $(x(t), y(t))$ where:

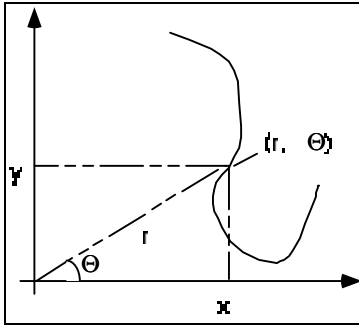


Figure 3.76. Polar coordinates.

$$\begin{aligned} x(t) &= r(t) \cos(\theta(t)) \\ y(t) &= r(t) \sin(\theta(t)). \end{aligned} \quad (3.19)$$

But a simplification is possible for a large number of appealing curves. In these instances the radius r is expressed directly as a function of θ , and the parameter that “sweeps” out the curve is θ itself. For each point (r, θ) the corresponding Cartesian point (x, y) is given by

$$\begin{aligned} x &= f(\theta) \cdot \cos(\theta) \\ y &= f(\theta) \cdot \sin(\theta) \end{aligned} \quad (3.20)$$

Curves given in polar coordinates can be generated and drawn as easily as any others: The parameter is θ , which is made to vary over an interval appropriate to the shape. The simplest example is a circle with radius K : $f(\theta) = K$. The form $f(\theta) = 2K \cos(\theta)$ is another simple curve (which one?). Figure 3.77 shows some shapes that have simple expressions in polar coordinates:

1st Ed. Figure 4.19

Figure 3.77. Examples of curves with simple polar forms..

- *Cardioid*: $f(\theta) = K (1 + \cos(\theta))$.
- *Rose curves*: $f(\theta) = K \cos(n \theta)$, where n specifies the number of petals in the rose. Two cases are shown.
- *Archimedian spiral*: $f(\theta) = K \cdot \theta$.

In each case, constant K gives the overall size of the curve. Because the cardioid is periodic, it can be drawn by varying θ from 0 to 2π . The rose curves are periodic when n is an integer, and the Archimedian spiral keeps growing forever as θ increases from 0. The shape of this spiral has found wide use as a cam to convert rotary motion to linear motion (see [Yates46] and [Seggern90]).

The **conic sections** (ellipse, parabola, and hyperbola) all share the following polar form:

$$f(\theta) = \frac{1}{1 \pm e \cdot \cos(\theta)} \quad (3.21)$$

where e is the eccentricity of the conic section. For $e = 1$ the shape is a parabola; for $0 \leq e < 1$ it is an ellipse; and for $e > 1$ it is a hyperbola.

• The Logarithmic Spiral

The **logarithmic spiral** (or “equiangular spiral”) $f(\theta) = Ke^{a\theta}$, shown in Figure 3.78a, is also of particular interest [Coxeter61]. This curve cuts all radial lines at a constant angle α , where $a = \cot(\alpha)$. This is the only spiral that has the same shape for any change of scale: Enlarge a photo of such a spiral any amount, and the enlarged

Figure 3.78. The logarithmic spiral and b). chambered nautilus

spiral will fit (after a rotation) exactly on top of the original. Similarly, rotate a picture of an equiangular spiral, and it will seem to grow larger or smaller [Steinhaus69]⁹. This preservation of shape seems to be used by some animals such as the mollusk inside a chambered nautilus (see Figure 3.78b). As the animal grows, its shell also grows along a logarithmic spiral in order to provide a home of constant shape [Gardner61].

Other families of curves are discussed in the exercises and Case Studies, and an exhaustive listing and characterization of interesting curves is given in [yates46, seggern90, shikin95].

3.8.5. 3D Curves.

Curves that meander through 3D space may also be represented parametrically, and will be discussed fully in later chapters. To create a parametric form for a 3D curve we invent three functions $x(\cdot)$, $y(\cdot)$, and $z(\cdot)$, and say the curve is “at” $P(t) = (x(t), y(t), z(t))$ at time t .

Some examples are:

The helix: The circular helix is given parametrically by:

$$\begin{aligned} x(t) &= \cos(t) \\ y(t) &= \sin(t) \\ z(t) &= bt \end{aligned} \tag{3.22}$$

for some constant b . It illustrated in Figure 3.79 as a stereo pair. See the Preface for viewing stereo pairs. If you find this unwieldy, just focus on one of the figures.

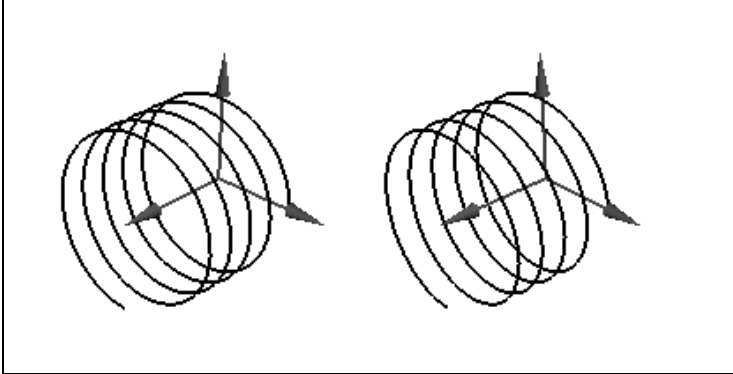


Figure 3.79. The helix, displayed as a stereo pair.

Many variations on the circular helix are possible, such as the elliptical helix $P(t) = (W \cos(t), H \sin(t), bt)$, and the conical helix $P(t) = (t \cos(t), t \sin(t), bt)$ (sketch these). Any 2D curve $(x(t), y(t))$ can of course be converted to a helix by appending $z(t) = bt$, or some other form for $z(t)$.

The toroidal spiral. A toroidal spiral, given by

$$\begin{aligned} x(t) &= (a \sin(ct) + b) \cos(t) \\ y(t) &= (a \sin(ct) + b) \sin(t) \\ z(t) &= a \cos(ct) \end{aligned} \tag{3.23}$$

⁹This curve was first described by Descartes in 1638. Jacob Bernoulli (1654---1705) was so taken by it that his tombstone in Basel, Switzerland, was engraved with it, along with the inscription *Eadem mutata resurgo*: “Though changed I shall arise the same.”

is formed by winding a string about a torus (doughnut). Figure 3.80 shows the case $c = 10$, so the string makes 10 loops around the torus. We examine tubes based on this spiral in Chapter 6.

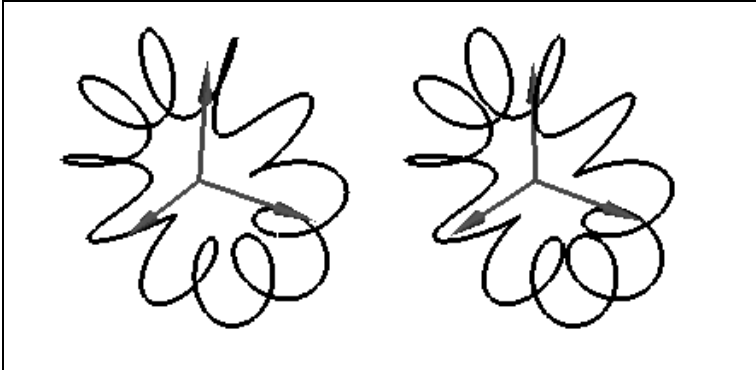


Figure 3.80. A toroidal spiral, displayed as a stereo pair.

Practice Exercises

3.8.6. Drawing superellipses. Write a routine `drawSuperEllipse(. . .)` that draws a superellipse. It takes as parameters c , the center of the superellipse, size parameters W and H , the bulge n , and m , the number of “samples” of the curve to use in fashioning the polyline approximation.

3.8.7. Drawing polar forms. Write routines to draw an n -petaled rose and an equiangular spiral.

3.8.8. Golden Cuts. Find the specific logarithmic spiral that makes “golden cuts” through the intersections of the infinite regression of golden rectangles, as shown in Figure 3.81 (also recall Chapter 2). How would a picture like this be drawn algorithmically?

1st Ed. Figure 4.22

Figure 3.81. The spiral and the golden rectangle.

3.8.9. A useful implicit form function. Define a suitable implicit form for the rose curve defined earlier in polar coordinate form: $f(\theta) = K \cos(n \theta)$.

3.8.10. Inside-outside functions for polar curves. Discuss whether there is a single method that will yield a suitable inside-outside function for *any* curve given in polar coordinate form as in Equation 3.20. Give examples or counter-examples.

3.9. Summary of the Chapter.

In this chapter we developed several tools that make it possible for the applications programmer to “think” and work directly in the most convenient “world” coordinate system for the problem at hand. Objects are defined (“modeled”) using high precision real coordinates, without concern for “where” or “how big” the picture of the object will be on the screen. These concerns are deferred to a later selection of a window and a viewport – either manually or automatically – that define both how much of the object is to be drawn, and how it is to appear on the display. This approach separates the modeling stage from the viewing stage, allowing the programmer or user to focus at each phase on the relevant issues, undistracted by details of the display device.

The use of windows makes it very easy to “zoom” in or out on a scene, or “roam” around to different parts of a scene. Such actions are familiar from everyday life with cameras. The use of viewports allows the programmer to place pictures or collections of pictures at the desired spots on the display in order to compose the final picture. We described techniques for insuring that the window and viewport have the same aspect ratio, in order to prevent distortion.

Clipping is a fundamental technique in graphics, and we developed a classical algorithm for clipping line segments against the world window. This allows the programmer to designate which portion of the picture will actually be rendered: parts outside the window are clipped off. OpenGL automatically performs this clipping, but in other environments a clipper must be incorporated explicitly.

We developed the *Canvas* class to encapsulate many underlying details, and provide the programmer with a single uniform tool for fashioning drawing programs. This class hides the OpenGL details in convenient

routines such as `setWindow()`, `setViewport()`, `moveTo()`, `lineTo()`, and `forward()`, and insures that all proper initializations are carried out. In a Case Study we implement *Canvas* for a more basic non-OpenGL environment, where explicit clipping and window-to-viewport mapping routines are required. Here the value of data-hiding within the class is even more apparent.

A number of additional tools were developed for performing relative drawing and turtle graphics, and for creating drawings that include regular polygons, arcs and circles. The parametric form for a curve was introduced, and shown to be a very natural description of a curve. It makes it simple to draw a curve, even those that are multi-valued, cross over themselves, or have regions where the curve moves vertically.

3.10. Case Studies.

One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.

Bertrand Russell

3.10.1. Case Study 3.1. Studying the Logistic Map and Simulation of Chaos.

(Level of Effort: II) Iterated function systems (IFS's) were discussed at the end of Chapter 2. Another IFS provides a fascinating look into the world of **chaos** (see [Gleick87, Hofs85]), and requires proper setting of a window and viewport. A sequence of values is generated by the repeated application of a function $f(\cdot)$, called the **logistic map**. It describes a parabola:

$$f(x) = 4\lambda x(1 - x) \quad (3.24)$$

where λ is some chosen constant between 0 and 1. Beginning at a given starting point, x_0 , between 0 and 1, function $f(\cdot)$ is applied iteratively to generate the **orbit** (recall its definition in Chapter 2):

$$x_k = f^{[k]}(x_0)$$

How does this sequence behave? A world of complexity lurks here. The action can be made most vivid by displaying it graphically in a certain fashion, as we now describe. Figure 3.82 shows the parabola $y = 4\lambda x(1 - x)$ for $\lambda = 0.7$ as x varies from 0 to 1.

1st Ed. Figure 3.28

Figure 3.82. The logistic map for $\lambda = 0.7$.

The starting point $x_0 = 0.1$ is chosen here, and at $x = 0.1$ a vertical line is drawn up to the parabola, showing the value $f(x_0) = 0.252$. Next we must apply the function to the new value $x_1 = 0.252$. This is shown visually by moving horizontally over to the line $y = x$, as illustrated in the figure. Then to evaluate $f(\cdot)$ at this new value a line is again drawn up vertically to the parabola. This process repeats forever as in other IFS's. From the previous position (x_{k-1}, x_k) a horizontal line is drawn to (x_k, x_k) from which a vertical line is drawn to (x_k, x_{k+1}) . The figure shows that for $\lambda = 0.7$, the values quickly converge to a stable "attractor," a fixed point so that $f(x) = x$. (What is its value for $\lambda = 0.7$?) This attractor does not depend on the starting point; the sequence always converges quickly to a final value.

If λ is set to small values, the action will be even simpler: There is a single attractor at $x = 0$. But when the " λ -knob" is increased, something strange begins to happen. Figure 3.83a shows what results when $\lambda = 0.85$. The "orbit" that represents the sequence falls into an endless repetitive cycle, never settling down to a final value. There are several attractors here, one at each vertical line in the limit cycle shown in the figure. And when λ is increased beyond the critical value $\lambda = 0.892486418\dots$ the process becomes truly chaotic.

1st Ed. Figure 3.29



Figure 3.83. The logistic map for a). $\lambda = 0.85$ and b). $\lambda = 0.9$.

The case of $\lambda = 0.9$ is shown in Figure 3.83b. For most starting points the orbit is still periodic, but the number of orbits observed between the repeats is extremely large. Other starting points yield truly aperiodic motion, and very small changes in the starting point can lead to very different behavior. Before the truly remarkable character of this phenomenon was first recognized by Mitchell Feigenbaum in 1975, most researchers believed that very small adjustments to a system should produce correspondingly small changes in its behavior and that simple systems such as this could not exhibit arbitrarily complicated behavior. Feigenbaum's work spawned a new field of inquiry into the nature of complex nonlinear systems, known as chaos theory [Gleick87]. It is intriguing to experiment with this logistic map.

Write and exercise a program that permits the user to study the behavior of repeated iterations of the logistic map, as shown in Figure 3.83. Set up a suitable window and viewport so that the entire logistic map can be clearly seen. The user gives the values of x_0 and λ and the program draws the limit cycles produced by the system.

3.10.2. Case Study 3.2. Implementation of the Cohen Sutherland Clipper in C/C++.

(Level of Effort: II) The basic flow of the Cohen Sutherland algorithm was described in Section 3.3.2. Here we flesh out some details of its implementation in C or C++, exploiting for efficiency the low-level bit manipulations these languages provide.

We first need to form the “inside-outside” code words that report how a point P is positioned relative to the window (see Figure 3.20). A single 8-bit word code suffices: four of its bits are used to capture the four pieces of information. Point P is tested against each window boundary in turn; if it lies outside this boundary, the proper bit of code is set to 1 to represent TRUE. Figure 3.84 shows how this can be done. code is initialized to 0, and then its individual bits are set as appropriate using a bit-wise OR operation. The values 8, 4, 2, and 1 are simple masks. For instance, since 8 in binary is 00001000, bit-wise OR-ing a value with 8 sets the fourth bit from the right end to 1.

```
unsigned char code = 0;           // initially all bits are 0
...
if(P.x < window.l)               code |= 8;      // set bit 3
if(P.y > window.t)               code |= 4;      // set bit 2
if(P.x > window.r)               code |= 2;      // set bit 1
if(P.y < window.b)               code |= 1;      // set bit 0
```

Figure 3.84. Setting bits in the “inside-outside code word” for a point P .

In the clipper both endpoints P_1 and P_2 (see Figure 3.22) are tested against the window, and their code words `code1` and `code2` are formed. We then must test for “trivial accept” and “trivial reject”.

- **trivial accept:** Both endpoints are inside, so both codes `code1` and `code2` are identically 0. In C/C++ this is quickly determined using the bit-wise OR: a trivial accept occurs if $(\text{code1} \mid \text{code2})$ is 0.

- **trivial reject:** A trivial reject occurs if both endpoints lie outside the window *on the same side*: both to the left of the window, both above, both below, or both to the right. This is equivalent to their codes having at least one 1 in the *same* bit position. For instance if `code1` is 0110 and `code2` is 0100 then P_1 lies both above and to the right of the window, while P_2 lies above but neither to the left nor right. Since both points lie above, no part of the line can lie inside the window. So trivial rejection is easily tested using the bit-wise AND of `code1` and `code2`: if they have some 1 in the same position then `code1 & code2` does also, and $(\text{code1} \& \text{code2})$ will be nonzero.

Chopping when there is neither trivial accept nor reject.

Another implementation issue is efficient chopping of the portion of a line segment that lies outside the window, as in Figure 3.22. Suppose it is known that point P with code word `code` lies outside the window. The

individual bits of code can be tested to see on which side of the window P lies, and the chopping can be accomplished as in Equation 3.5. Figure 3.85 shows a chop routine that finds the new point (such as A in Figure 3.22) and replaces P with it. It uses the bit-wise AND of code with a mask to determine where P lies relative to the window.

```
ChopLine(Point2 &P, unsigned char code)
{
    if(code & 8){          // to the Left
        P.y += (window.l - P.x) * dely / delx;
        P.x = window.l;
    }
    else if(code & 2){      // to the Right
        P.y += (window.r - P.x) * dely / delx;
        P.x = window.r;
    }
    else if(code & 1){      // below
        P.x += (window.b - P.y) * delx / dely;
        P.y = window.b;
    }
    else if(code & 4){      // above
        P.x += (window.t - P.y) * delx / dely;
        P.y = window.t;
    }
}
```

Figure 3.85. Chopping the segment that lies outside the window.

Write a complete implementation of the Cohen Sutherland algorithm, putting together the pieces described here with those in Section 3.3.2. If you do this in the context of a *Canvas* class implementation as discussed in the next Case Study, consider how the routine should best access the private data members of the window and the points involved, and develop the code accordingly.

Test the algorithm by drawing a window and a large assortment of randomly chosen lines, showing the parts that lie inside the window in red, and those that lie outside in black.

Practice Exercises.

3.10.1. Why will a “divide by zero” never occur? Consider a *vertical* line segment such that delx is zero. Why is the code $\text{P.y} += (\text{window.l} - \text{P.x}) * \text{dely} / \text{delx}$ that would cause a divide by zero never reached? Similarly explain why each of the four statements that compute delx/dely or dely/delx are never reached if the denominator happens to be zero.

3.10.2. Do two chops in the same iteration? It would seem to improve performance if we replaced lines such “else if($\text{code} \& 2$)” with “if($\text{c} \& 2$)” and tried to do two line “chops” in succession. Show that this can lead to erroneous endpoints being computed, and hence to disaster.

3.10.3. Case Study 3.3. Implementing Canvas for Turbo C++.

(Level of Effort: III) It is interesting to develop a drawing class like *Canvas* in which all the details are worked out, to see how the many ingredients go together. Sometimes it is even necessary to do this, as when a supporting library like OpenGL is not available. We design Canvas here for a popular graphics platform that uses Borland’s Turbo C++.

We want an implementation of the *Canvas* class that has essentially the same interface as that in Figure 3.25. Figure 3.86 shows the version we develop here (omitting parts that are simple repeats of Figure 3.25). The constructor takes a desired width and height but no title, since Turbo C++ does not support titled screen windows. There are several new private data members that internally manage clipping and the window to viewport mapping.

```
class Canvas {
public:
    Canvas(int width, int height); // constructor
    setWindow(), setViewport(), lineTo(), etc .. as before
```


}

3). moveTo(), and lineTo() with clipping.

The routine `moveTo()` converts its point from world coordinates to screen coordinates, and calls the Turbo C++ specific `moveTo()` to update the internal current position maintained by Turbo C++. It also updates *Canvas*' world coordinate *CP*. Routine `lineTo()` works similarly, but it must first determine which part if any of the segment lies within the window. To do this it uses `clipSegment()` described in Section 3.3 and in Case Study 3.2, which returns the `first` and `second` endpoints of the inside portion. If so it moves to `first` and draws a line to `second`. It finishes with a `moveTo()` to insure that the *CP* will be current (both the *Canvas CP* and the internal Turbo C++ *CP*).

ChopLine and ClipSegment are same as in Case Study 3.2.

```
//<<<<<<<<<<<<<<<<<<< moveTo >>>>>>>>>>>>>>>>>>>
void Canvas::moveTo(float x, float y)
{
    int sx = (int)(mapA * x + mapC);
    int sy = (int)(mapB * y + mapD);
    moveto(sx, sy); // a Turbo C++ routine
    CP.set(x, y);
}
//<<<<<<<<<<<<<<<<<<<.lineTo >>>>>>>>>>>>>>>>>>>
void Canvas::lineTo(float x, float y)
{ // Draw a line from CP to (x,y), clipped to the window
    Point2 first = CP; // initial value of first
    Point2 second(x, y); // initial value of second
    if(clipSegment(first, second)) // any part inside?
    {
        moveTo(first.x, first.y); // to world CP
        int sx = (int)(mapA * second.x + mapC);
        int sy = (int)(mapB * second.y + mapD);
        lineto(sx,sy); // a Turbo C++ routine
    }
    moveTo(x, y); // update CP
}
```

Write a full implementation of the Canvas class for Turbo C++ (or a similar environment that requires you to implement clipping and mapping). Cope appropriately with setting the drawing and background colors (this is usually quite system-specific). Test your class by using it in an application that draws polyspirals as specified by the user.

3.10.4. Case Study 3.4. Drawing Arches.

(Level of Effort: II) Arches have been used throughout history in architectural compositions. Their structural strength and ornamental beauty make them very important elements in structural design, and a rich variety of shapes have been incorporated into cathedrals, bridges, doorways, etc.

Figure 3.87 shows two basic arch shapes. The arch in part a) is centered at the origin, and has a width of $2W$. The arch begins at height H above the base line. Its principal element is a half-circle with a radius $R = W$. The ratio H/W can be adjusted according to taste. For instance, H/W might be related to the golden ratio.

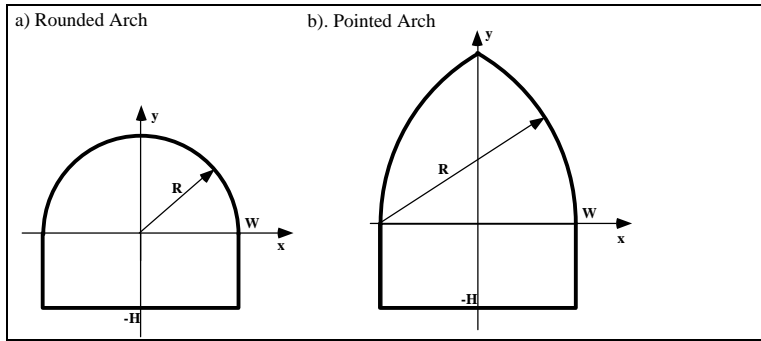


Figure 3.87. Two basic arch forms.

Figure 3.73b shows an idealized version of the second most famous arch shape, the **pointed** or “equilateral” arch, often seen in cathedrals¹⁰. Here two arcs of radius $R = 2W$ meet directly above the center. (Through what angle does each arc sweep?)

The **ogee**¹¹ (or “keel”) arch is shown in Figure 3.88. This arch was introduced about 1300 AD, and was popular in architectural structures throughout the late Middle Ages. Circles of radius fR rest on top of a rounded arch of radius R for some fraction f . This fixes the position of the two circles. (What are the coordinates of point C ?) On each side two arcs blend together to form a smooth pointed top. It is interesting to work out the parameters of the various arcs in terms of W and f .

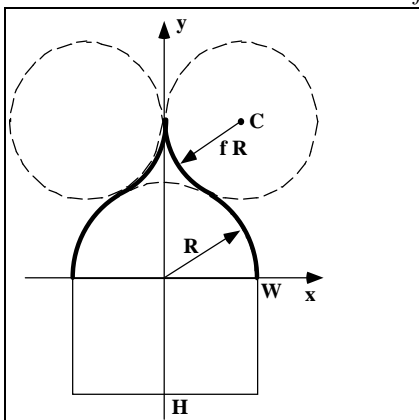


Figure 3.88. The Ogee arch.

Develop routines that can draw each of the arch types described above. Also write an application that draws an interesting collection of such arches in a castle, mosque, or bridge of your design.

3.10.5. Case Study 3.5. Some Figures used in Physics and Engineering.

(Level of Effort: II) This Case Study works with a collection of interesting pictures that arise in certain topics within physics and engineering. The first illustrates a physical principal of circles intersecting at right angles; the second creates a chart that can be used to study electromagnetic phenomena; the third develops symbols that are used in designing digital systems.

1). Electrostatic Fields. The pattern of circles shown in Figure 3.89 is studied in physics and electrical engineering, as the electrostatic field lines that surround electrically charged wires. It also appears in mathematics in connection with the analytic functions of a complex variable. In Chapter 5 these families also are found when we examine a fascinating set of transformations, “inversions in a circle.” Here we view them simply as an elegant array of circles and consider how to draw them.

¹⁰From J.Fleming, H. Honour, N. Pevsner: **Dictionary of Architecture**. Penguin Books, London 1980

¹¹From the old French *ogive* meaning an S-shaped curve.

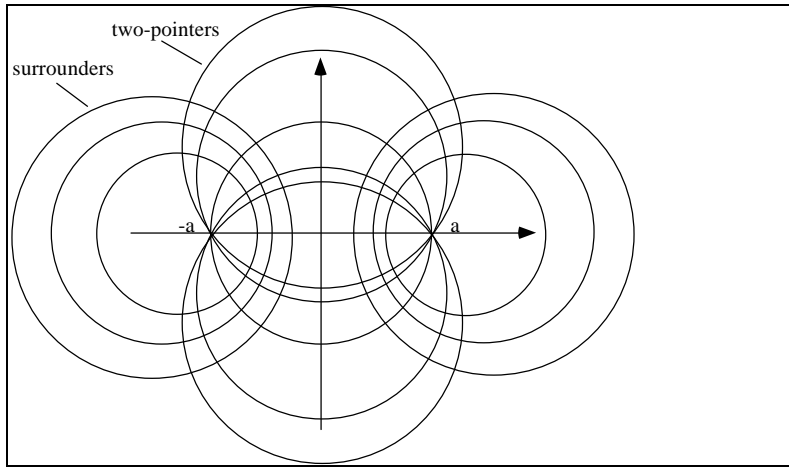


Figure 3.89. Families of orthogonal circles..

There are two families of circles, which we will call “two-pointers” and “surrounders”. The two-pointers family consists of circles that pass through two given points. Suppose the two points are $(-a, 0)$ and $(a, 0)$. The two-pointers can be distinguished by some parameter m , and for each value of m two different circles are generated (see Figure 3.75). The circles have centers and radii given by:

$$\text{center} = (0, \pm a\sqrt{m^2 - 1}) \quad \text{and} \quad \text{radius} = am$$

as m varies from 1 to infinity.

Circles in the surrounders family surround one of the points $(-a, 0)$ or $(a, 0)$. The centers and radii of the surrounders are also distinguished by a parameter n and have the values

$$\text{center} = (\pm an, 0) \quad \text{and} \quad \text{radius} = a\sqrt{n^2 - 1}$$

as n varies from 1 to infinity. The surrounder circles are also known as “circles of Apollonius,” and they arise in problems of pursuit [Ball & Coxeter]. The distances from any point on a circle of Apollonius to the points $(-a, 0)$ and $(a, 0)$ have a constant ratio. (What is this ratio in terms of a and n ?)

The “surrounder” family is intimately related to the two-pointer family: Every surrounder circle “cuts” through every two-pointer circle at a right angle. The families of circles are thus said to be **orthogonal** to one another.

Write and exercise a program that draws the two families of orthogonal circles. Choose sets of values of m and n so that the picture is well balanced and pleasing.

2). Smith Charts. Another pattern of circles is found in Smith charts, familiar in electrical engineering in connection with electromagnetic transmission lines. Figure 3.90 shows the two orthogonal families found in Smith charts. Here all members of the families pass through a common point $(1, 0)$. Circles in family A have centers at $(1 - m, 0)$ and radii m , and circles in family B have centers at $(1, \pm n)$ and radii n , where both m and n vary from 0 to π . Write and exercise a program that draws these families of circles.

1st Ed. Figure 4.31

Figure 3.90. The Smith Chart.

3). Logic Gates for Digital Circuits. Logic gates are familiar to scientists and engineers who study basic electronic circuits found in computers. Each type of gate is symbolized in a circuit diagram by a characteristic shape, several of which are based on arcs of circles. Figure 3.91a shows the shape of the so-called

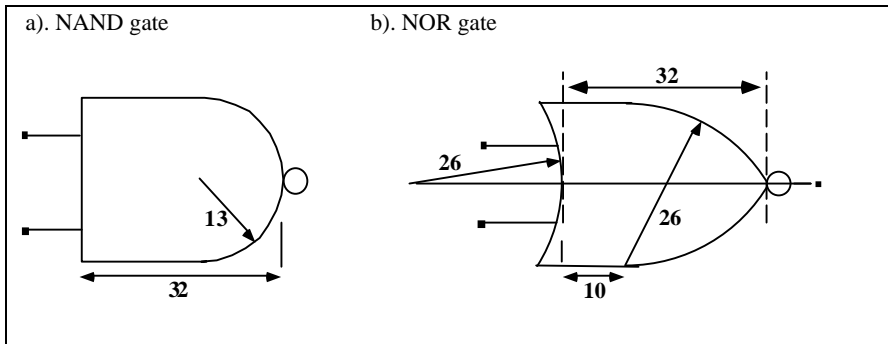


Figure 3.91. Standard Graphic Symbol for the Nand and Nor Gates.

NAND gate, according to a world-wide standard¹². The NAND gate is basically a rounded arch placed on its side. The arc has radius 13 units relative to the other elements, so the NAND gate must be 26 units in height.

Figure 3.91b shows the standard symbol for a NOR gate. It is similar to a pointed arch turned on its side. Three arcs are used, each having a radius of 26 units. (The published standard as shown has an error in it, that makes it impossible for certain elements to fit together. What is the error?)

Write a program that can draw both of these circuit types at any size and position in the world. (For the NOR gate find and implement a reasonable correction to the error in Figure 3.77b.) Also arrange matters so that your program can draw these gates rotated by 90° , 180° , or 270° .

3.10.6. Case Study 3.6. Tilings.

(Level of Effort: II) Computer graphics offers a powerful tool for creating pleasing pictures based on geometric objects. One of the most intriguing types of pictures are those that apparently repeat forever in all directions. They are called variously **tilings**, and **repeat patterns**. They are studied in greater detail in Chapter ???.

A). Basic Tilings. Figure 3.92 shows a basic tiling. A **motif**, in this case four quarter circles in a simple arrangement, is designed in a square region of the world. To draw a tiling over the plane based on this motif, a collection of viewports are created side by side that cover the display surface, and the motif is drawn once inside each viewport.

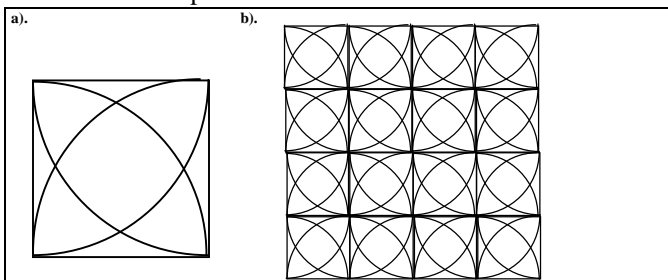


Figure 3.92. A motif and the resulting tiling.

Write a program that:

- chooses a square window in the world, and draws some interesting motif (possibly clipping portions of it, as in Figure 3.14);
- successively draws the picture in a set of viewports that abut one another and together cover the display surface.

Exercise your program with at least two motifs.

¹²The Institute of Electrical and Electronic Engineers (IEEE) publishes many things, including standard definitions of terminology and graphic shapes of circuit elements. These drawings are taken from the standard document: IEEE Std. 91-1984 .

B). Truchet Tiles. A slight variation of the method above selects successive motifs randomly from a “pool” of candidate motifs. Figure 3.93a shows the well-known Truchet tiles¹³, which are based on two quarter circles centered at opposite corners of a square. Tile 0 and tile 1 differ only by a 90° rotation.

artist sketches the two tiles here

Figure 3.93. Truchet Tiles. a). the two tiles. b). A truchet pattern

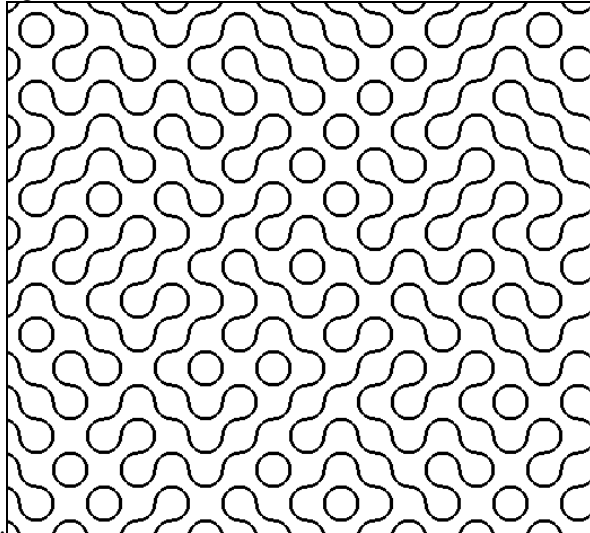


Figure 3.93.b.

Write an application that draws Truchet tiles over the entire viewport. Each successive tile uses tile 0 or tile 1, selected at random.

Curves other than arcs can be used as well, as suggested in Figure 3.94. What conditions should be placed on the angle with which each curve meets the edge of the tile in order to avoid sharp corners in the resulting curve? This notion can also be extended to include more than two tiles.

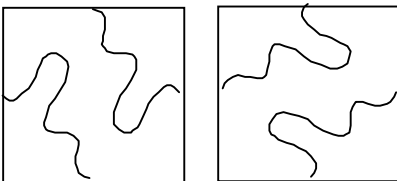


Figure 3.94. Extension of Truchet tiles.

Extend the program above so that it introduces random selections of two or more motifs, and exercise it on the motifs you have designed. Design motifs that “blend” together properly.

3.10.7. Case Study 3.7. Playful Variations on a Theme.

(Estimate of time required: four hours). In Section 3.8 we discussed how to draw a curve represented parametrically by $P(t)$: take a succession of instants $\{t_i\}$ and connect the successive “samples” $(x(t_i), y(t_i))$ by straight lines. A wide range of pictures can be created by varying the way in which the samples are taken. We suggest some possibilities here.

¹³Smith, C. “The Tiling Patterns of Sebastian Truchet and the topology of structural hierarchy.” *Leonardo*, 20:4, pp: 373-385, 1987. (refd in Pickover, p.386)

Write a program that draws each of the four shapes:

- a). an ellipse
- b). a hyperbola
- c). a logarithmic spiral
- d). a 5-petal rose curve

for each of the methods described below for obtaining t -samples.

1). Unevenly Spaced Values of t . Instead of using a constant increment between values of t when sampling the functions $x()$ and $y()$, use a varying increment. It is interesting to experiment with different choices to see what visual effects can be achieved. Some possibilities for a sequence of $(n+1)$ t -values between 0 and T (suitably chosen for the curve shape at hand) are

- $t_i = T\sqrt{i/n}$: The samples cluster closer and closer together as i increases.
- $t_i = T(i/n)^2$: The samples spread out as i increases.
- $t_i = T(i/n) + A\sin(ki/n)$ The samples cyclically cluster together or spread apart. Constants A and k are chosen to vary the amount and speed of the variation.

2). Randomly Selected t -Values. The t -values can be chosen randomly as in

$$t_i = \text{randChoose}(0, T)$$

Here $\text{randChoose}(0, T)$ is a function (devised by you) that returns a value randomly selected from the range 0 to T each time it is called. (See Appendix 3 for a basic random number generator.)

Figure 3.95 shows the polyline generated in this fashion for points on an ellipse. It is interesting to watch such a picture develop on a display. A flurry of seemingly unrelated lines first appears, but soon the eye detects some order in the chaos and “sees” an elliptical “envelope” emerging around the cloud of lines.

1st Ed. Figure 4.26

Figure 3.95. A random ellipse polyline.

Alternatively, a sequence of increasing t -values can be used, generated by

$$t_i = t_{i-1} + \text{randChoose}(0, r)$$

where r is some small positive value.

3). Connecting Vertices in Different Orders

In a popular children’s game, pins are driven into a board in some pattern, and a piece of thread is woven around the pins in some order. The t -values here define the positions of the pins in the board, and $\text{worldLineTo}()$ plays the role of the thread.

The samples of $P(t)$ are prestored in a suitable array $P[i]$, $i = 0, 1, \dots, n$. The polyline is drawn by sequencing in an interesting way through values of i . That is, the sequence i_0, i_1, \dots is generated from values between 0 and n , and for each index i_k a call to $\text{worldLineTo}(P[i_k])$ is made. Some possibilities are:

- “Random Deal”: the sequence i_0, i_1, \dots is a random permutation of the values $0, 1, \dots, n$, as in dealing a fixed set of cards from a shuffled deck.
- Every pair of points is connected by a straight line. So every pair of values in the range $0, 1, \dots, n$ appears in adjacent spots somewhere in the sequence i_0, i_1, \dots . The prime rosette of Chapter 5 gave one example, where lines were drawn connecting each point to every other.

• One can also draw “webs,” as suggested in Figure 3.96. Here the index values cycle many times through the possible values, skipping by some M each time. This is easily done by forming the next index from the previous one using $i = (i + M) \bmod (n+1)$.

1st Ed. Figure 4.27

Figure 3.96. Adding webs to a curve.

3.10.8. Case Study 3.8. Circles Rolling around Circles.

(Level of Effort: II) Another large family of interesting curves can be useful in graphics. Consider the path traced by a point rigidly attached to a circle as the circle rolls around another fixed circle [thomas53, Yates46]. These are called **trochoids**, and Figure 3.97 shows how they are generated. The tracing point is attached to the rolling circle (of radius b) at the end of a rod k units from the center. The fixed circle has radius a . There are two basic kinds: When the circle rolls externally (Figure 3.97a), an **epitrochoid** is generated, and when it rolls internally (Figure 3.97b), a **hypotrochoid** is generated. The children’s game Spirograph¹⁴ is a familiar tool for drawing trochoids, which have the following parametric forms:

1st Ed. Figure 4.23

Figure 3.97. Circles rolling around circles.

The epitrochoid:

$$\begin{aligned} x(t) &= (a + b) \cos(2\pi t) - k \cos(2\pi \frac{(a + b)t}{b}) \\ y(t) &= (a + b) \sin(2\pi t) - k \sin(2\pi \frac{(a + b)t}{b}) \end{aligned} \quad (3.24)$$

The hypotrochoid:

$$\begin{aligned} x(t) &= (a - b) \cos(2\pi t) + k \cos(2\pi \frac{(a - b)t}{b}) \\ y(t) &= (a - b) \sin(2\pi t) - k \sin(2\pi \frac{(a - b)t}{b}) \end{aligned} \quad (3.25)$$

An ellipse results from the hypotrochoid when $a = 2b$ for any k .

When the tracing point lies on the rolling circle ($k = b$) these shapes are called **cycloids**. Some familiar special cases of cycloids are

Epicycloids:

Cardioid: $b = a$
Nephroid: $2b = a$

Hypocycloids:¹⁵

Line segment: $2b = a$
Deltoid: $3b = a$
Astroid: $4b = a$.

Some of these are shown in Figure 3.98. Write a program that can draw both epitrochoids and hypotrochoids. The user can choose which family to draw, and can enter the required parameters. Exercise the program to draw each of the special cases listed above.

1st Ed. Figure 4.24.

Figure 3.98. Examples of cycloids: a) nephroid, b) $a/b = 10$, c) deltoid, d) astroid.

¹⁴A trademark of Kenner Products.

¹⁵Note that the astroid is also a superellipse! It has a bulge of $2/3$.

3.10.9. Case Study 3.9. Superellipses.

(Level of Effort: I) Write and exercise a program to draw superellipses. To draw each superellipse, the user indicates opposite corners of its bounding, and types a value for the bulge, whereupon the specified superellipse is drawn.

(Optional). Extend the program so that it can draw rotated superellipses. The user types an angle after typing the bulge.

3.11. For Further Reading.

When getting started with graphics it is very satisfying to write applications that produce fascinating curves and patterns. This leads you to explore the deep connection between mathematics and the visual arts. Many books are available that offer guidance and provide myriad examples. McGregor and Watt's *THE ART OF GRAPHICS FOR THE IBM PC*, [mcgregor86] offers many algorithms for creating interesting patterns. Some particularly noteworthy books on curves and geometry are Jay Kappraff's *CONNECTIONS* [kappraff91], Dewdney's *THE ARMCHAIR UNIVERSE* [dewdney88], Stan Ogilvy's *EXCURSIONS IN GEOMETRY* [ogilvy69], Pedoe's *GEOMETRY AND THE VISUAL ARTS* [pedoe76], Roger Sheperd's *MIND SIGHTS* [shep90], and the series of books on mathematical excursions by Martin Gardner, (such as *TIME TRAVEL* [gardner88] and *PENROSE TILES TO TRAPDOOR CIPHERS* [gardner89]). Coxeter has written elegant books on geometry, such as *INTRODUCTION TO GEOMETRY* [Coxeter69] and *MATHEMATICAL RECREATIONS AND ESSAYS* [ball74], and Hoggar's *MATHEMATICS FOR COMPUTER GRAPHICS* [hoggar92] discusses many features of iterated function systems.