CHAPTER 5

Introduction to Computer Graphics

In this chapter, we consider how graphics is done on a computer. We begin by considering the hardware involved; monitors, printers, and so on. Then we consider various graphics primitives (drawing lines and writing text, for example) and then how to design fairly sophisticated graphs such as two and three dimensional plots. Finally, we show how to use widely available programming tools (MS Windows, X-Windows, and Postscript) to create full purpose graphics packages.

Computer Monitors

Most computer screens are composed of a rectangular grid of dots called picture elements (popularly called "pixels" or "pels"). These dots are such that if they are all turned on in the same color in a region of the screen, then that region looks as though it were solidly drawn in that color. Typically, a computer will have a special set of high speed memory chips which contain a description of what color each pixel is on the screen (a bitmap) and a special processing unit that constantly redraws (refreshes) the screen. The more pixels a screen has, the better geometric shapes look on the screen. For example, If one wants to draw a line staring with one pixel and ending with another, the line will not look solid because of the discreteness of the pixels. This effect is commonly called the "jaggies" and can be visualized by thinking of a page of graph paper and drawing a line from one square to another by filling in those squares and "the ones in between."

The inadaquacy of lines on a pixel device is of course less pronounced if the pixels are very small than if they are large, that is, a device having small pixels is said to have "higher resolution." For example, a standard VGA monitor on a PC has 640 rows of pixels with each row having 480 pixels.

There are two prices to pay for high resolution. First, they can be much more expensive to produce, and second, the amount of storage required to represent what is to be displayed on the device is much more. For example, if one only wanted to use black and white on a VGA screen, the first eight pixels (white or black) can be represented by the eight bits (1 or 0) of the first byte of a region in memory, the next byte could be the next eight pixels, and so on. Thus it would take 80 bytes of memory to store each row of pixels or a total of $80 \times 480 = 38,400$ bytes to represent the screen. To have $256 = 2^8$ colors for each pixel would require an entire byte for each pixel, that is a total of 307,200 bytes.

Printers

Many printers work in much the same way as a screen except of course there need be no refreshing. The storage requirement for a printer can be much worse, due to its very high resolution. For an HP laserjet, there are 300 dots per inch in both the horizontal and vertical directions. Thus an entire page (8 by 10.5 inches say) would require 3,150 rows of dots with each row having 2,400 dots. A row would require 300 bytes for a total of 945,000 bytes for a bitmap ofthe entire page. Thus to do full page graphics on a Laserjet requires the printer to have at least one MByte of memory.

Some printers do not require bitmaps be sent to the printer. The most notable example is a Postscript printer. To draw a figure made up of a series of lines, one need only send a series of line drawing instructions (each requiring only a few bytes) and the printer can understand and execute the instructions. Thus the information sent to a postscript printer is typically much smaller than the bitmap that has to be sent to other printers. Further, postscript instructions can be easily modified so that the resulting figure can be of almost any size, that is, scaling is very easily done as opposed to bitmaps which can seldom be scaled.

Compressing Bitmaps

One solution to the problem of needing very large files to store bitmaps is to store them in compressed form. In many cases this is done using the idea of run length encoding (RLE). While there are many forms of this idea, most of them consist of using the fact that most mathematical and statistical graphic figures have a large amount of blank space.

The simplest form of RLE is of the following form. Suppose we have a black and white image on the screen that we want to compress. The process starts by noting the color of the pixel at the left of the topmost line of pixels and works across and then down the lines (the process need not go in this order but for convenience we look at it this way), counting the length of each "run" of pixels before the color changes (only the color of the upper left corner pixel need be recorded). Thus if we were compressing a standard $X - Y$ plot of a curve, many rows of pixels would all be the same color and a huge compression would be achieved.

Drawing Text on a Graphics Screen

Each letter on a screen is actually represented by a bitmap telling the colors of all the pixels in the rectangular area occupied by the character (typically, telling whether each pixel is in the "foreground" or "background" color). For example, in the high resolution graphics mode on an original IBM PC, each character occupies an 8 by 8 pixel region. The letter A is represented by

```
. . 1 1 . . . .    48
. 1 1 1 1 . . .   128
1 1 . . 1 1 . .   204
1 1 . . 1 1 . .   204
1 1 1 1 1 1 . .   252
1 1 . . 1 1 . .   204
1 1 . . 1 1 . .   204
. . . . . . . .     0
```

where a '.' represents a 0 in the representation, and we have listed the decimal number corresponding to the binary representation given by each row of pixels. PC's have a special place in memory where the bitmaps for the first 128 ASCII characters are stored.

Turning on Pixels

All graphics operations on a monitor (and any bitmap printer) consist ultimately of the operation of "turning on a pixel," that is, making a specified pixel (or printing dot) a specified color. As we saw above this includes drawing text. In the rest of this chapter we discuss how turning on pixels can be used to draw lines and ultimately very complicated graphs, including 3D and contour plots.

Some computers have special hardware for drawing lines or circles or polygons, but we will only consider basic algorithms for these operations.

Bresenham's Line Drawing Algorithm

Bresenham's line drawing algorithm is a method for determining which pixels between the end points of a line should be made the color of the line. The algorithm is remarkable in that it only requires arithmetic operations on integers, not on floating point numbers.

Before describing the algorithm in detail, there are a number of things to notice.

1. Unless the line happens to be horizontal, vertical, or falls on a diagonal of the grid of pixels, the line will not appear perfectly straight on the screen.
2. The line should be "connected", that is, there should be no discernible gaps in the set of pixels that make up the line. This means that if the line is "steep", that is, it travels more rows of pixels than it does columns, then a pixel in each row between the starting and ending row must be "turned on". On the other hand, a pixel in each column of a shallow line must be turned on. Thus steep and shallow lines are handled separately, and one loops over rows for steep lines and over columns for shallow lines. The
3. For a fixed row or column in a loop, one has a choice of two pixels that can be turned on. For a steep line with a positive slope for example, when going up from one row to the next in the loop, one has to choose which of two columns contains the pixel to be turned on, either the same column as the one for the previous row or the column to the right of the one for the previous row. These two choices are referred to as a "nondiagonal move" or a "diagonal move" and one must make a choice between such moves for each of four possible cases: 1) steep line with positive slope, 2) steep line with negative slope, 3) shallow line with positive slope, and 4) shallow line with negative slope.

The Algorithm

We will consider the pixel in the lower left hand corner of the screen to be in column zero and row zero and use the notation (c, r) to represent the pixel in column c and row r. Suppose we want to draw a line from pixel (c_0, r_0) to pixel (c_1, r_1), where $c_0 < c_1$, that is, (c_0, r_0) is to the left of (c_1, r_1). If we consider these pixels to be points in Euclidean space, then the true line connecting these two points is the set of (x, y) satisyfing

$$f(x, y) = (y - r_0)\delta_c - (x - c_0)\delta_r = 0,$$

where $\delta_c = c_1 - c_0$ and $\delta_r = r_1 - r_0$. Notice that $\delta_c > 0$ and that δ_r has the same sign as the slope of the line. Further, (x, y) is a point above (below) the line if and only if $f(x, y) > 0$ ($f(x, y) < 0$), while the line is steep (shallow) if $\delta_r > \delta_c$ ($\delta_r < \delta_c$).

In the shallow line case, we will loop over the columns from c_0 to c_1 (that is, from left to right), at each step determining if the row stays the same or increases (in the positive slope case) or decreases (in the negative slope case) by one. In the steep line case, we loop over the rows (from r_0 up to r_1 for positive slope or from r_0 down to r_1 for negative slope), at each step determining if the column stays the same or increases by one.

The Algorithm for a Shallow Line with Positive Slope

We consider how Bresenham's algorithm decides if a diagonal or nondiagonal move is to be made in the shallow line with positive slope case. The other three cases are done similarly and are summarized later. Suppose at the end of one step, the pixel (x_i, y_i) was turned on. Then the next one to be turned on is either $p_N = (x_i + 1, y_i)$, a nondiagonal move, or $p_D = (x_i + 1, y_i + 1)$, a diagonal move. Consider the midpoint m_i of these two pixels, that is $m_i = (x_i + 1, y_i + 1/2)$. Then the true line is closer to p_N (p_D) if it is below (above) the point m_i, that is, if $f(m_i) > 0$ ($f(m_i) < 0$). Thus deciding which move is to be made at a step requires only knowing whether $f(m_i)$ is positive or negative.

The first key idea in Bresenham's algorithm is the fact that $f(m_i)$ can be calculated recursively, that is

$$\begin{aligned} f(m_i) &= f(x_i + 1, y_i + 1/2) = (y_i + 1/2 - r_0)\delta_c - (x_i + 1 - c_0)\delta_r \\ &= \begin{cases} (y_{i-1} + 1 + 1/2 - r_0)\delta_c - (x_{i-1} + 1 + 1 - c_0)\delta_r, & \text{if last step diag} \\ (y_{i-1} + 1/2 - r_0)\delta_c - (x_{i-1} + 1 + 1 - c_0)\delta_r, & \text{if last step nondiag} \end{cases} \\ &= \begin{cases} f(m_{i-1}) + (\delta_c - \delta_r), & \text{if last step diagonal} \\ f(m_{i-1}) - \delta_r, & \text{if last step nondiagonal} \end{cases}. \end{aligned}$$

Since all we care about is the sign of $f(m_i)$, we can just as well keep track of $d_i = 2f(m_i)$ which has the advantage that it is guaranteed to be integer valued. Thus we have

$$d_i = 2f(m_i) = \begin{cases} d_{i-1} + 2(\delta_c - \delta_r), & \text{if last step diagonal} \\ d_{i-1} - 2\delta_r, & \text{if last step nondiagonal} \end{cases}.$$

Another key fact is that the terms $2(\delta_c - \delta_r)$ and $-2\delta_r$, referred to as the diagonal and nondiagonal increments respectively, are the same at every step. Thus they can be determined before starting the loop, and whenever a move is determined, the value of d_i can be updated for the next step. The final part of the recursion is to notice that the starting value d_0 is given by

$$d_0 = 2f(c_0 + 1, r_0 + 1/2) = \delta_c - 2\delta_r.$$

Summary for all Four Cases

Inspection of the four cases (remembering to always work from left to right) shows that a nondiagonal move is chosen when $f(m_i) > 0$ for the shallow line with positive slope and steep line with negative slope cases and when $f(m_i) < 0$ for the other two cases. So that we will always be checking for positivity of whatever we're keeping track of in the recursion, we will keep track of

$$a_i = \begin{cases} 2f(m_i), & \text{in the shallow positive and steep negative cases} \\ -2f(m_i), & \text{in the shallow negative and steep positive cases} \end{cases}.$$

If we work through the algebra for all four cases, we find that a_0 and the diagonal and nondiagonal increments (call them δ_D and δ_N respectively) are given by

$$\begin{aligned} a_0 &= \begin{cases} \delta_c - 2|\delta_r|, & \text{for shallow lines} \\ |\delta_r| - 2\delta_c, & \text{for steep lines} \end{cases} \\ \delta_D &= \begin{cases} 2(\delta_c - |\delta_r|), & \text{for shallow lines} \\ 2(|\delta_r| - \delta_c), & \text{for steep lines} \end{cases} \\ \delta_N &= \begin{cases} -2|\delta_r|, & \text{for shallow lines} \\ -2\delta_c, & \text{for steep lines} \end{cases} \end{aligned}.$$

In the next section we give a Fortran implementation of this general algorithm.

A Fortran Implementation of the Algorithm

```
      subroutine line(icol1,irow1,icol2,irow2,kolor)
c-----------------------------------------------------
c
c   Draw a line connecting pixel (icol1,irow1) and
c   (icol2,irow2) in color kolor.
c
c-----------------------------------------------------
c
      integer delc,delr,rinc,dinc,a
c
c   Get left point in (ic0,ir0), right point in (ic1,ir1):
c
      ic0=min0(icol1,icol2)
      ic1=max0(icol1,icol2)
      ir0=irow1
      ir1=irow2
      if(icol1.gt.icol2) then
             ir0=irow2
             ir1=irow1
      endif
c
c   Horizontal, vertical (diagonal is handled in shallow
c   case below):
c
      if(ir0.eq.ir1) then
             do 10 i=ic0,ic1
  10         call wrdota(i,ir0,kolor)
             return
      endif
      if(ic0.eq.ic1) then
             do 20 i=min0(ir0,ir1),max0(ir0,ir1)
  20         call wrdota(ic0,i,kolor)
             return
      endif
c
c   Other cases:
c
      delc=ic1-ic0
      delr=ir1-ir0
      idelc=iabs(delc)
      idelr=iabs(delr)
      rinc=idelr/delr
      ix=ic0
      iy=ir0
c
c   Shallow:
c
      if(delc.ge.idelr) then
             dinc=2*(delc-idelr)
             ndinc=-2*idelr
             call wrdota(ic0,ir0,kolor)
             a=delc-2*idelr
             do 40 ix=ic0+1,ic1
                    if(a.gt.0) then
                           a=a+ndinc
                    else
                           a=a+dinc
                           iy=iy+rinc
                    endif
                    call wrdota(ix,iy,kolor)
  40         continue
             return
c
c   Steep:
c
      else
             dinc=2*(idelr-delc)
             ndinc=-2*delc
             call wrdota(ic0,ir0,kolor)
             a=idelr-2*delc
             do 50 iy=ir0+rinc,ir1,rinc
                    if(a.gt.0) then
                           a=a+ndinc
                    else
                           ix=ix+1
                           a=a+dinc
                    endif
```

```
                    call wrdota(ix,iy,kolor)
   50           continue
        endif
        return
        end
```

The Basic $X - Y$ Plot

Statistics is filled with $X-Y$ plots, that is, graphs that have horizontal and vertical axes with tic marks and numbers labeling the values on the tic marks, a graph of some kind (barplot, scatterplot, function plot, and so on), character strings labeling the meaning of the horizontal and vertical axes, and a character string above the plot providing a caption for the plot, and possibly a variety of other features. Typically, one constructs a plotting routine in such a way that features such as points, lines, text, arrows, and so on can be added to the plot later.

In this section we consider the basic numerical tools one needs to construct such plots.

From Real Coordinates to Pixels

In most graphics, the user thinks of the screen or the printer in real coordinates, that is, as a portion of the two dimensional Cartesian plane. Thus a graphics routine must eventually convert real coordinates to pixel coordinates. In most cases this consists of finding the pixel (c, r) that corresponds to a real coordinate (x, y) for a portion of the screen whose lower left hand corner has real coordinate (x_0, y_0) and pixel coordinate (c_0, r_0) and upper left hand corner having real coordinate (x_1, y_1) and pixel coordinate (c_1, r_1).

We consider finding c from c_0, c_1, x_0, and x_1 and note that finding r is done in the same way. We divide the interval $[x_0, x_1]$ into $m = c_1 - c_0 + 1$ intervals (each interval corresponding to a pixel) of length $h = (x_1 - x_0)/m$, where the ith interval is given by

$$[x_0 + (i-1)h, x_0 + ih), \qquad i = 1, \ldots,$$

and transform x to pixel $c = c_0 + (i-1)$ if x falls in the ith interval, that is, if $x \in [x_0 + (i-1)h, x_0 + ih)$, or equivalently, if $[(x - x_0)/h] = i - 1$.

If $x \geq x_1$, then x will be converted to a pixel greater than c_1, while if $x < x_0$ it will be converted to a pixel less than c_0. Thus one typical conversion method is given by

$$c = c_0 + \max(0,\ \min(\ \mathrm{int}((x - x_0)/h),\ m-1)).$$

Finding Nice Numbers for Axis Tic Mark Labels

In most $X - Y$ plots, one desires tic mark labels that are "nice" (S-Plus calls them "pretty"), that is, labels that are not something like 1.459 followed by 1.645, and so on. Suppose we want to construct a scatterplot of n pairs $(x_1, y_1), \ldots, (x_n, y_n)$ of numbers and we'd like to do this by calling a routine such as `scatter(x,y)`, that is, without having to specify the number of tic marks and the values at the tic marks. The simplest way to do this would be to have the ends of the axes be the minimum and maximum values of the x's and y's and then have the tic mark labels be the equally spaced values between the minima and maxima. This of course could easily lead to "ugly" labels.

Given values x_0 and x_1 of the smallest and largest elements of an array and a desired number N of intervals (and thus $N + 1$ is the desired number of tic marks), we describe an algorithm for finding a new minimum X_0 and maximum X_1 so that the range $[X_0, X_1]$ is the smallest range which contains the interval $[x_0, x_1]$ and simultaneously result in approximately N intervals each of length h where h is the product of 1, 2, or 5 and an integer power of 10 and X_0 and X_1 are integer multiples of h.

The Fortran program given below contains the method and produces the following output:

```
     xmin      xmax     n     xmin1     xmax1         h    n1
-------------------------------------------------------------
     0.53      0.79     7      0.50      0.80      0.05     6
```

10.00	47.00	11	10.00	50.00	5.00	8
-0.60	-0.40	5	-0.60	-0.40	0.05	4
119.00	9875.00	10	0.00	10000.00	1000.00	10
2.00	19.00	6	2.00	20.00	2.00	9

The Nice Numbers Program

```
      dimension xmin(5),xmax(5),n(5)

      data xmin/ .53, 10., -.6,  119.,  2./
      data xmax/ .79, 47., -.4, 9875., 19./
      data n/      7, 11,    5,   10,   6/

      write(*,10)
 10   format(1x,5x,4hxmin,5x,4hxmax,4x,1hn,4x,5hxmin1,
    1        4x,5hxmax1,8x,1hh,3x,2hn1/,1x,55(1h-))

      do 20 i=1,5
      call nice(xmin(i),xmax(i),n(i),xmin1,xmax1,n1,h)
 20   write(*,30) xmin(i),xmax(i),n(i),xmin1,xmax1,h,n1
 30   format(1x,2f9.2,i5,3f9.2,i5)

      stop
      end
      subroutine nice(xmin,xmax,n,xmin1,xmax1,n1,dist)
c-------------------------------------------------------------
c
c   Given the minimum (xmin) and maximum (xmax) of an array,
c   and a desired number of intervals (n), this subroutine
c   finds a number of intervals (n1) and a new minimum (xmin1)
c   and a new maximum (xmax1) such that [xmin1,xmax1] contains
c   [xmin,xmax] and the length of the intervals (dist) is the
c   product of 1, 2, or 5 with an integer power of 10 and xmin1
c   and xmax1 are multiples of dist.
c
c-------------------------------------------------------------

      dimension vint(4),sqr(3)

      data vint/1.,2.,5.,10./
      data sqr/1.414214,3.162278,7.071068/

      del=.00002

c   Find interval width dist:

      a=(xmax-xmin)/n
      al=log10(a)
      nal=al
      if(a.lt.1.) nal=nal-1
      b=a/(10.**nal)

      do 10 i=1,3
 10   if(b.lt.sqr(i)) go to 30
      i=4

 30   continue

      dist=vint(i)*10.**nal

c   Get xmin1 = largest multiple of dist .le. xmin:

      fm1=xmin/dist
      m1=fm1
      if(fm1.lt.0.0) m1=m1-1
      if(abs(m1+1.0-fm1).lt.del) m1=m1+1
      xmin1=dist*m1

c   Get xmax1 = smallest multiple of dist .ge. xmax:

      fm1=xmax/dist
      m1=fm1
      m1=m1+1
      if(fm1.lt.-1.0) m1=m1-1
      if(abs(fm1+1.0-m1).lt.del) m1=m1-1
      xmax1=dist*m1
```

```

      if(xmin1.gt.xmin) xmin1=xmin
      if(xmax1.lt.xmax) xmax1=xmax

c   Get new number of intervals:

      n1=((xmax1-xmin1)/dist)+0.05

      return
      end
```

Contour Plots

One popular plot is the contour plot of a function $f(x, y)$ of two variables x and y. Typically, we are given values of the function at a grid of (x, y) points, where the grid is made up of all combinations of $x_1 < \cdots < x_{n_x}$ and $y_1 < \cdots, y_{n_y}$. Usually, the values of f are stored in an $(n_x \times n_y)$ matrix Z where $Z_{ij} = f(x_i, y_j)$.

A contour plot is made up of several contours, where a contour having level c traces a curve where the horizontal plane having height c intersects the three dimensional solid object determined by the function f, that is, the set of (x, y) such that $f(x, y) = c$. Since we only have the values of f at a grid of points, we have no way of knowing the behavior of f within a rectangle in the grid.

To see how a contour plot is drawn, consider a contour having height c and the (i, j)th box in the grid of (x, y) values and let

$$v_1 = (x_i, y_j), \quad v_2 = (x_i, y_{j+1}), \quad v_3 = (x_{i+1}, y_{j+1}), \quad v_4 = (x_{i+1}, y_j),$$

be the vertices of the box starting at the lower left corner and proceding clockwise around the box (see below). Let f_1, f_2, f_3, and f_4 be the corresponding values of f at the vertices. If c is greater than all four f's, then we will assume that the contour plane is entirely above the function in this box (and any contour plane above this one is also entirely above the function). If c is less than all four f's, then the contour plane is assumed below the function in this box (and so is any contour plane below this one).

```
v2 .------------. v3
   |            |
   |            |
   |            |
   |            |
v1 .------------. v4
```

If some of the four f's are above c and some are below, then we can assume that the contour plane 'cuts' the function inside the box. For example if f_1 and f_3 are below c while f_2 is above, then we will draw a line in this box representing the intersection of the contour plane and the plane drawn through the points f_1, f_2, and f_3. This will result in a line from the left side to the top of the box where the endpoints of the line are determined by interpolation. Thus if c is close to f_2, then the left endpoint of the line will be close to v_2. For the (i, j)th box, the x value of the endpoint on the top of the box is given by

$$x = x_i + \frac{f_2 - c}{f_2 - f_3}(x_{i+1} - x_i).$$

There are actually 16 combinations of $+$'s and $-$'s of the values of f at the four vertices. Each of the combinations leads to no lines, one line, or two lines being drawn. In the table below, we list each of the results.

```
      subroutine cntour(x,y,z,nx,ny,ndim,npx,npy,nx1,ny1,
     1                  nctrs,ctrs,ix,iy)
```

Table of Configurations in Contour Plotting

```
Line(s)           Configuration(s)   Line(s)        Configuration(s)
-------------------------------------------------------------------
None:             +   +       -   -

                  +   +       -   -

Left -> Bottom    -   -       +   +  Left -> Top    +   -     -   +

                  +   -       -   +                 -   -     +   +

Left -> Right     +   +       -   -  Top -> Bottom  +   -     -   +

                  -   -       +   +                 +   -     -   +

Right ->Bottom    -   -       +   +  Right -> Top   -   +     +   -

                  -   +       +   -                 -   -     +   +

Left  -> Top      -   +              Left-> Bottom  +   -
    and                                  and
Right ->Bottom    +   -              Right -> Top   -   +
-------------------------------------------------------------------
```

```
c----------------------------------------------------------------
c
c   Subroutine to draw a contour plot on a raster device.
c
c   Input:
c       nx, ny  : Number of grid points in x and y directions
c       x, y    : Vectors containing x and y values of grid pts.
c                 Each vector must be in increasing order.
c       z       : Matrix whose (i,j)th element is value of the
c                 function at grid point (x(i),y(j)).
c       ndim    : Row dimension of z in calling program.
c       npx, npy: Size in pixels of plotting region.
c       nx1, ny1: Pixel column and row numbers of lower left hand
c                 corner of plotting region. Note that the lower
c                 left corner of the screen is column 0, row 0.
c       nctrs   : Number of contours.
c       ctrs    : Vector containing contour values in
c                 increasing order.
c
c   Output:
c       ix, iy  : Vectors defined by: ix(i) = pixel col of x(i).
c                                     iy(i) = pixel row of y(i).
c
c   Notes:
c       1) The outline of plotting region is drawn, but no grid.
c       2) The contours are drawn in the same color. To change
c          this, one need only modify the line subroutine.
c       3) The user is responsible for setting the graphics mode
c          before calling cntour and for making sure that the
c          values of npx, npy, nx1, ny1 are selected so that
c          the plotting region fits on the screen.
c       4) The only graphics routine called by cntour is line.
c
c----------------------------------------------------------------

        dimension x(nx),y(ny),z(ndim,1),ctrs(nctrs),ix(nx),iy(ny)
c
c   Draw lines outlining plotting region:
c
        call line(nx1,ny1,nx1,ny1+npy-1,1)
        call line(nx1,ny1+npy-1,nx1+npx-1,ny1+npy-1,1)
        call line(nx1+npx-1,ny1+npy-1,nx1+npx-1,ny1,1)
        call line(nx1+npx-1,ny1,nx1,ny1,1)
```

```
c
      rx=x(nx)-x(1)
      ry=y(ny)-y(1)
      xmin=x(1)
      ymin=y(1)
c
c  Find pixel coordinates of grid points:
c
      do 5 i=1,nx
  5   ix(i)=ifpix(x(i),npx,nx1,xmin,rx)
      do 6 i=1,ny
  6   iy(i)=ifpix(y(i),npy,ny1,ymin,ry)
c
c  Loop over i = column of boxes, j = row of boxes, k = which
c  contour
c
      do 20 i=1,nx-1
c
      xleft=x(i)
      xright=x(i+1)
      xdiff=xright-xleft
      ixl=ix(i)
      ixr=ix(i+1)
c
      do 20 j=1,ny-1
c
      ylow=y(j)
      yup=y(j+1)
      ydiff=yup-ylow
      iyl=iy(j)
      iyu=iy(j+1)
c
c     z2 .-----------. z3
c         |          |
c         |          |
c         |          |
c         |          |
c     z1 .-----------. z4
c
      z1=z(i,j)
      z2=z(i,j+1)
      z3=z(i+1,j+1)
      z4=z(i+1,j)
c
c
      do 10 k=1,nctrs
      cc=ctrs(k)
c
c  nzg = # of z's greater or equal to current contour
c
      nzg=0
      if(z1.ge.cc) nzg=nzg+1
      if(z2.ge.cc) nzg=nzg+1
      if(z3.ge.cc) nzg=nzg+1
      if(z4.ge.cc) nzg=nzg+1
c
c  If nzg = 0, then all later contours are above too so jump out
c  of contour loop:
c
      if(nzg.eq.0) go to 15
      if(nzg.eq.4) go to 10
c
c  ns = sum of vertex numbers of z's that are greater or
c  equal to cc
c
      ns=0
      if(z1.ge.cc) ns=ns+1
      if(z2.ge.cc) ns=ns+2
      if(z3.ge.cc) ns=ns+3
      if(z4.ge.cc) ns=ns+4
c
c  nzg = 3 and ns = j is the same as nzg = 1 and ns = 10 - j
c
      if(nzg.eq.3) then
              ns=10-ns
              nzg=1
      endif
c
      if(nzg.eq.1) then
c
```

```
                go to(11,12,13,14) ns
c
c   left to bottom:
c
  11            ixx=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                call line(ixl,iyy,ixx,iyl,1)
                go to 10
c
c   left to top:
c
  12            ixx=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                call line(ixl,iyy,ixx,iyu,1)
                go to 10
c
c   top to right:
c
  13            ixx=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixx,iyu,ixr,iyy,1)
                go to 10
c
c   right to bottom:
c
  14            ixx=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixx,iyl,ixr,iyy,1)
                go to 10
c
        endif
c
        if(nzg.eq.2) then
c
c   top to bottom:
c
                if(ns.eq.3.or.ns.eq.7) then
                ixx1=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                ixx2=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                call line(ixx1,iyu,ixx2,iyl,1)
                go to 10
                endif
c
c   left to right:
c
                if(ns.eq.5) then
                iyy1=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                iyy2=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixl,iyy1,ixr,iyy2,1)
                go to 10
                endif
c
c   two lines with negative slopes:
c
                if(ns.eq.4) then
                ixx=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                call line(ixl,iyy,ixx,iyl,1)
                ixx=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixx,iyu,ixr,iyy,1)
                go to 10
                endif
c
c   two lines with positive slopes:
c
                if(ns.eq.6) then
                ixx=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                call line(ixl,iyy,ixx,iyu,1)
                ixx=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixx,iyl,ixr,iyy,1)
                go to 10
                endif
        endif

c
  10    continue
  15    continue
```

```
 20    continue
c
c
       return
       end
       integer function intpix(x1,xd,c,z1,z2,npx,nx1,xmin,rx)
c-----------------------------------------------------------------
c
c   Interpolate and find pixel. x1 is lower x value, xd is
c   difference in x values.
c
c-----------------------------------------------------------------

       xx=x1+xd*(z1-c)/(z1-z2)
       intpix=ifpix(xx,npx,nx1,xmin,rx)
       return
       end
       integer function ifpix(x,npx,nx1,xmin,rx)
c-----------------------------------------------------------------
c
c   Convert real world coordinate x to pixel coordinate ifpix.
c
c-----------------------------------------------------------------

       ifpix=nx1+min(npx-1,max(0,nint(npx*(x-xmin)/rx)))
c
       return
       end
```