HIGH-RESOLUTION PRINTER GRAPHICS

by Mark Bridger and Mark Goresky

You can address the individual dots used to generate dot-matrix characters.

ONE OF THE GREATEST frustrations in doing graphics on a microcomputer is the rather low resolution of the usual microcomputer monitor. The standard IBM Personal Computer color-graphics adapter and monitor display a maximum screen size of 640 by 200 pixels (picture elements); other computers and configurations do somewhat better, perhaps as much as 720 by 350 pixels. It is difficult to draw horizontal lines fast enough to keep the image from flickering. And there are limits to the amount of screen memory available on standard graphics boards.

Many dot-matrix printers are capable of printing individual dots at a much higher resolution than the typical CRT (cathode-ray tube) screen can display them. The Epson FX-80 and the IBM graphics printer are capable, for example, of printing 240 dots per inch horizontally (1920 dots per line) and 216 dots per inch vertically—the latter by printing a line of graphics, advancing the paper one-third of a dot, printing another "interlaced" line of graphics, etc. Other printers can perform similar feats. To use this capability you need to figure out how to "fire the pins" and you need enough extra memory to record where all the dots are to go. This article will show you how to draw some lines and curves on your resolution of up to 1600 by 640 dots.

SETTING UP THE PRINTERSCREEN

The first problem is memory. If you think of a dot as being either on or off, to use an analogy with the screen display, then encoding 1600 by 640 dots, or 1,024,000 points, requires that many bits of information. If you divide by 8 to convert bits to bytes, then the process requires 128,000 bytes, or nearly 129 kilobytes of memory. Somehow you must get aside that much memory to record this image. Unfortunately, this is not easily done in BASIC, so we must look elsewhere.

The most widely used microcomputer language that allows enticing off this much memory is Pascal, and because Turbo Pascal lets you point to nearly all available memory without having to give explicit addresses, it is the easiest language to use.

Let's set up two 64K-byte memory areas that represent the even lines and the odd lines of a picture. Each of these areas is represented by the following Pascal data type:

```
    type data_type = array[0..1599, 0..39] of char;
```

This type of variable is a doubly indexed 1600 by 40 array of characters: since one byte represents each character, this multiplies to about 64K bytes.

Now let's declare the variables that are to reserve this space:

```
    var Evenmap, Oddmap:
    data_type;
```

(continued)

Mark Bridger and Mark Goresky are associate professors of mathematics at Northeastern University. Mark Bridger has a Ph.D. from Brandeis University. Mark Goresky holds a Ph.D. from Brown University. Mark Bridger can be reached at Bridge Software, 31 Chapman St., Newton Upper Falls, MA 02164. Mark Goresky can be reached at the Mathematics Dept., Northeastern University, 360 Huntington Ave., Boston, MA 02115.

November 1985 • Byte • 119
The “-” defines a pointer. When you actually create these variables during program execution, using the command New, the computer sets aside two blocks of free memory and automatically reserves them for your use. Each of the variables Evenmap and Oddmap “points” to the beginning of one of these blocks, and you need never concern yourself with exactly where in memory these blocks reside.

**How A Dot-Matrix Printer Draws Dots**
The print head of a dot-matrix printer normally has seven or more wires, arranged vertically: the most common number is nine. (Eight are used to draw most of the characters, while the ninth is used to draw the bottoms of the g and y characters and to underlining.) When typing letters, the printer receives the ASCII code of the character—a number between 0 and 255. As the print head moves across the page, it extends certain wires, depending on the pattern stored in the printer’s memory for that character, and the head strikes them against the paper. Usually from 9 to 12 such columns of dots are needed to make a character.

You want to be able to tell the printer directly which wires to fire. In other words, you want to bypass that part of the printer’s memory that stores the patterns for the printing of usual characters (letters, numbers, etc.)—you want to do 8-buaped graphics. Most printers support this: it is usually called graphics mode. Let’s try to address a particular dot on the page.

First, since the wires on the print head are not that close together, you can make use of tiny partial lines to double the number of vertical dots. Table 1 contains a diagram of how it works. The characters represent dot positions on the page: the 1s represent the dots that you actually want to draw and the 0s represent the dot positions you want to skip. To get maximum resolution, you want the dots to be as close to each other as possible, both horizontally and vertically. Getting them close horizontally is accomplished by means of a simple printer command. To get them close vertically, you must divide the picture into the even rows (0, 2, 4, etc.) and the odd rows (1, 3, 5, etc.), as shown in Table 2.

When the printer is in graphics mode, the printer prints, for each byte you send it, any pattern of eight vertical dots you specify. The strategy in Table 2 is to do the following:
1. Send the printer the 10 bytes that specify the 10 columns represented by the even rows.
2. Instruct the printer to do a carriage return plus a linefeed of one-half a vertical dot.
3. Send the printer the 10 bytes that specify the 10 columns represented by the odd rows.
4. Instruct the printer to do a carriage return plus a linefeed of 7/8 vertical dots, preparing it to draw more sets of even and odd rows if there are any.

In more ambitious applications you can have as many as 1600 columns across instead of just these 10. The array pointers Evenmap and Oddmap store this information for the printer. Each represents 600 columns; each column is 40 bytes or 320 dots high. Looked at another way, there are 320

---

### Table 1: This table shows the dot positions on the page. The 1s represent dots that you actually want to draw; the 0s, dot positions you want to skip over.  

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 2: This table shows the distribution of the various print dot positions between even and odd rows.  

<table>
<thead>
<tr>
<th>Even</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Odd</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

220 BYTE • NOVEMBER 1985
even rows and 320 odd rows. Each row is 1600 dots wide, and the printer will print eight even or eight odd rows in each pass. Note that these rows form a natural unit totaling 16 rows; let's call such a unit a printer line.

HOW TO LOCATE A DOT ON THE PAGE
Let's write a procedure—Part(y, color)—that draws a point of coor-
dinates x and y in the proper place in one of the two arrays. The coor-
dinates x and y denote the point's column and row (measured from the upper left-hand comer, respectively). The variable color can be equal to either 0 or 1: 0 means erase any point existing at that location; 1 means in-
ser a point there. [Ballator's note: All pro-
grams shown here are available for down-
loading on BIT/Enet Listings. Before Noverber 1 call (617) 861-9774. Al-
ternatively, call (617) 861-9764.] See listing 1 for the procedure Part. Start at the line that reads color =
color mod 2. First the procedure en-
sures that color is in the correct range
by applying a mod 2 to it. (When K
and N are whole numbers, K mod N
finds the remainder you get when you
divide K by N.) While inside by 2, you
get a remainder of only 0 or 1,
depending on whether K is even or odd, respectively. Next, you deter-
mine which printer line you're in by
dividing the row number by 16 (y div
16). When you know this line number,
you can determine which vertical dot
within that line you're at is the key.
Finally, y mod 2 tells you whether your
dot is in an even or an odd row.
For example, suppose you want to
print a dot in column 1173, row 554.
Then x equals 1173 and y equals 554.
554 div 16 equals 34, so you are in
the 34th printer line. 554 mod 16 is 16
and 1173 div 2 is 5. so the height of
the dot within the printer line is 5;
since 554 is even, you are in the ar-
ray pointed to by Evenmap. The pro-
gram now calls on the procedure
Change to insert this point into the
correct position in memory.

The problem now and the reason
Change is so complicated is that
(continues)
For the Prompter, these two last lines should be replaced by:

1: old = old OR MR (7 = height).  [See text for details.]
2: old = old AND RM (T = height)  

end;  

char_byte = char_and
end;  

[Change procedure found in file.  [Writes the dot at position (x,y) into memory arrays]]

var x,line,height : integer.

begin;  

plot(x + 2 div 5, y + 5 div 16, white);  

(draw dot on screen)  

    (This multiplies x by the ratio of screen width to printer width, 
    multiplies y by the ratio of screen height to printer height.
    For the Prompter, this last line should be replaced by:
    Plot(x div 2, y div 5 div 16, white);
    )  

    color = color mod 2;  
    height = y mod 16 div 2;  
    [vertical position of pixel consists of a line]
    [number between 0 and down; and a height]  
    [between 0 and 15, divided into]
    [even-odd groups]  
    if y mod 2 = 0 then change_oddmap = x line.color.height  
    else change_oddmap = x line.color.height  

end;  

[Plot]

Listing 2: Printout procedure for Prompter.
procedure Printout;  
var x,y;  
packed array [1:4] of char;  
a,b,i,j,k : integer.
begin;  

Printout;  

// write x,y // write packed array [1:4] of char,  
// a,b,i,j,k // integer.  

for Prompter

write (pt1:  );  

(write printer buffer // = 50 bytes)

written (pt1:  );  

[ = carriage return]

a = across + 1;  

[a = number of graphics bytes]

repeat

b = a mod 10;  

[get next digit (d)]  

[insert character in string: w0r]  

l = 1;  

until l = 0;  

[word = digits of across]  

written (pt1, ch(27),' ');  

[set pitch for proportionate spacing —
  the highest horizontal density]  

for J = 0 to down do  

begin

write (pt1, ch(27), 'S', ' ');  

[enter graphics mode]

for l = 0 to across do  

write (pt1, 'w', 'e', 'r', 'e', 'm', 'a', 'p', ' ');  

[print even rows]

write (pt1, ch(27), 'T', ' 0');  

[start next line 1/2 dot down]

write (pt1, ch(27), 'T', ' 1');  

[graphics mode again]

for l = 2 to across do  

write (pt1, 'd', 'o', 'd', 'd', 'a', 'p', 'p', ' ');  

[print odd rows]

write (pt1, ch(27), 'T', ' 15');  

[start next line 7/9 dots down]  

end;  

[Printout]

turning on a point involves changing a single bit within a byte. Computers
are generally not equipped to do this easily. Remember that each byte con-
trols eight vertical dots, and you want to change only one of them. This
is most easily done with bit masks and the logical operations AND and OR.
See the see text box on bit manipulation.

"RES ANDOR Pieces" on page 225.  [Note that in the Pixel Masks pro-
cedure, the leftmost bit in a byte is called the zeroth bit, while the right-
most bit is the seventh.]  

If you want to insert a 1 in the third bit, you use the mask M[height],
where height equals 3 with the logical OR operation. The code that inserts
this 1 into the byte Old is simply:

Old = Old OR M[height]

M[height] is a byte made up of all zeros except for a 1 in bit height. If
Old is 0000000 and height is 3, then Old becomes the byte 00000010 OR
00000000 = 00000010.  

If you want to insert a 0 into this same byte, you use the mask
M[height] together with the logical AND operation:

Old = Old AND M[height]

Here, P[height] is a byte made up of 1s everywhere, except for 0 in bit height. If Old is
0111110 and height is 7, then Old becomes the byte 0111110 AND
11111110 = 01111100.  

Note that you write to printers using Pascal's Write and Write procedures,
and these procedures expect to be given a character. This is why you should set up your arrays as character
arrays and why the last command in the Change procedure converts the byte into a character.

Some Printer Differences

The eight vertically arranged print pins on most printers correspond to the eight bits in a byte. In a Epson
FX-80 and many other printers the high-order bits—those in the left half of
the byte—correspond to the upper pins; the low-order, or rightmost, bits
correspond to the lower pins Thus, the byte 10000010 causes the top pin
and the next-to-the-bottom pin to
make dots on the paper. On the other hand, for the Prowriter and several other printers, the exact opposite is true—the highest bit causes the lowest pin to fire. Thus, if you want to insert a 1 in the third bit for a Prowriter, you OR with M[7:Height] where Height equals 3. To avoid confusion, we have indicated the corrections necessary to handle the Prowriter properly (see listing 2). If you have a different printer, you should check your manual for the correct pin assignments. (The Prism printer, for example, uses only seven pins.)

Another important difference between printers is in how close you are allowed to print the dots horizontally and vertically. In the Epson quad-

ruple-density graphics mode, available only on the FX, RX, and IBM models, the printer prints 240 dots per inch or 1920 dots across an 8-inch page. Because of restrictions on the size of arrays (64K-byte maximum), the examples in this article draw only 1600 dots. (We can draw more, but at the expense of some vertical rows.) The older Epson MX prints only 960 dots across the page. For the Pro-

writer, the highest density possible is provided in proportional mode, where you can print 160 dots per inch or 1280 per line—we use 120 in our examples. Each dot on a dot-matrix printer is approximately 1/2 inch in diameter. The Epson FX-80 permits linefeeds of 1/10 dot, which results in a theoretical vertical density of 216 dots per inch. The Prowriter allows in-dot linefeeds, or a vertical density of 144 dots per inch. In the examples in this article, we use the Epson 1/10-dot linefeeds as if they were 1/10 dot; this works fine, un-

doubtedly due to the inherent inac-

curacy of paper advance.

Once again, you must consult your printer manual if you have a different printer. The Prism printer does not seem to support fractional linefeeds at all, while the Mannesmann Tally achieves them by raising or lowering the actual print head 1/10 dot.

ECHOING ON THE SCREEN

We now have the complete setup for drawing a dot in “printer” memory.

Returning to listing 1, note the call to the procedure Plot. Plot is a Turbo Pascal procedure that draws a dot on the actual screen for each point you draw in memory. However, the scale for the printer is different from the scale for the screen: 1600 by 640 dots for the printer (1250 by 640 dots for the Prowriter) versus 640 by 200 dots (pixels) for the screen. For the Epson FX-80 you rescale by multiplying the column by 2, and rows by 2, by 640/1600 (2.5) and 200/640 (0.5/16). For the Prowriter you multiply by 640/1250 (0.568) and 640/640 (1/6). Since the actual number of multipli-

cation is time-consuming (unless you have an 8087 chip) and since Plot re-

quires integer parameters anyway, you (continued)

BITS AND/OR PIECES

Suppose you have two bytes, each represented as eight binary bits:

Byte 1 = 10110100 and Byte 2 = 00100000. To make calculation simpler later on, let’s call the first bit on the left of each byte the zeroth bit; the next is the first, then the second, etc. Thus, the zeroth bit of byte 1 is 1, the first is 0, and the seventh, or rightmost, bit is 0. When you OR Byte 1 and Byte 2 together, you produce a new byte, Byte 3. If either of the corresponding bits, for example the zeroth bits of Byte 1 and Byte 2, is a 1, then you make the corresponding bit of Byte 3 a 1; otherwise, it is 0. Thus, the zeroth bit of Byte 3 is a 1 since Byte 1 has a 1 in the zeroth position. The first bit of Byte 3 is 0 since neither Byte 1 nor Byte 2 has a 1 in that position.

Byte 1 = 10110100
Byte 2 = 00100000
Byte 3 = OR Byte 1 and Byte 2 = 10110100

If you perform an AND on the two

bytes, the process is similar except that you put a 1 in Byte 3 only if both corre-

sponding bits are 1. Let’s say Byte 4 = Byte 1 AND Byte 2.

Byte 1 = 10110100
Byte 2 = 00100000
Byte 4 = 00100000

Byte 4 = Byte 1 AND Byte 2 = 00100000

Suppose now that you have a byte B = (10010001) and you want to change the second bit from a 0 to a 1. If you have a byte that is all 0’s except for a 1 in this second position (i.e., third place from the left), then you can execute

B OR M2 = 10010100 OR 00000000 = 10010100

This accomplishes your purpose. You need eight different masks of this type to handle each possible bit position. Note that M2 = 00000000 (binary) = 20 (hexadecmal) = 32 (decimal), and M3 = 216 = 7 (binary) = (00000000) = 16 (hexadecimal) = 27 (decimal). Use these masks to AND and OR with a byte. The procedure Plasmask also constructs the array R of eight different reverse masks. The relation between the masks of the two types is easy to see. For ex-

ample, consider M3 = 00000000 and M2 = 11111111 (binary) = 255 (decimal). The procedure Plasmask constructs the array R of eight different reverse masks. The relation between the masks of the two types is easy to see. For ex-

ample, consider M3 = 00000000 and M2 = 11111111 (binary) = 255 (decimal).

Thus, you get the reverse pixel masks from the normal pixel masks by sub-

tracting the normal ones from 255.

On some graphics devices, you may need a second set of masks that do two things at a time. For example, the Apple monitors can display four colors at a time.
can do this quite neatly using integer multiplication and div:
Plot(x - 2 div 5, y + 5 div 16, white).

For the Prowriter:
Plot(x div 2, y + 5 div 16, white).

This is still somewhat wasteful since it draws some dots on top of others, but it is sufficient for this example.

HOW TO PRINT THE DOTS

In theory all we have to do is send these bytes to the printer. However, many printers are fussy and don’t like to be in graphics mode—in fact, they’ll only stay there for one line at a time. Furthermore, each time you invoke graphics mode you have to tell them how many graphics bytes to expect on that line; if you send them more, they start printing regular characters.

Let’s do a brief rundown on the Epson FX-80 graphics Printout procedure (see listing 1). LA1 is Turbo Pascal’s name for the printer. The Epson FX-80 instruction to enter quadruple-density graphics mode is Escape (chr(27)) followed by Z (on the MX, replace Z with L). Then the printer needs to receive the number of graphics bytes it should expect as a sequence of two characters, which are determined as follows:

\[
\text{Byte } #1 = \text{\texttt{y-n}_{10}} \quad \text{(of bytes mod 256)}
\]
\[
\text{Byte } #2 = \text{\texttt{n}_{10}} \quad \text{(of bytes div 256)}
\]

(This information should be easy to obtain from your printer manual under “Graphics Mode”)

Procedure PrintOut has two nested loops; the big one controls the printer lines, while the smaller sends out the character bytes within each printer line. Recall that a printer line consists of one even and one odd group of 1600 bytes. For each of these we must, as just mentioned, reenter graphics mode and give the byte count. The command write(5, chr(0)) is simply a carriage return.

The only other lines of interest are the paperfeeds. The Epson FX-80 won’t do a linefeed of ½ dot but rather works in multiples of ¾ dot. Since even Epson disclaims any great accuracy for such a tiny linefeed, we tried various combinations such as ¾ and ¾ ¾ and ¾, etc. The best image seemed to result from using ¾ and ¾ (227).

Now let’s take a look at the Pro-

writer graphics Printout procedure (see listing 2); since the Prowriter works a little differently you should clear out the 50-byte printer buffer by writing 50 blanks—we’ve never seen the necessity of this, but it is suggested as a precaution. Next, you should report the number of graphics bytes the printer is to expect (= across + 1) by sending a string whose characters are the decimal digits of this number. These are computed by the small loop from n = = across + 1 through until i = 0. The rest of the code is the same as the Epson FX-80’s except for the different printer instructions (escape sequences).

THE TESTCURVE PROGRAM

To demonstrate how these pro-

cedures work, listing 3 contains a driver program that sketches the simple parabola y = x*x (see figure 1).

The heart of this program is the pro-

cedure Plotcurve, which illustrates the scaling and coordinate manipulation necessary to draw “computer pic-
tures.” Since the origin is in the upper left-hand corner and the x-coordinate is measured downward, you are essentially plotting, y = 630 - (x - 25)*x should go from 0 to 50, since the width of the screen is across (1599 or 1249) you round across to the nearest 50 (width: = across - (across mod 50)) and let i go from 0 to width. The scale factor scaler is width/50 and x equals liscalar or (liscalar) = 50 Then, when i equals 0, x is 0; when i equals width, x is 50. Then you use Pset to graph your points:
Pset(i - width, y), liscalar or (liscalar) = 25 or (liscalar = 25).

Note that you must truncate (trunc) since Pset requires integer parameters.

CONNECTING THE DOTS

The procedure Plotcurve draws a curve by computing each point sepa-

(continued)
rately and then plotting it. Although this sufficed for a simple demonstration, it has two major shortcomings. First, it can skip points. For example, suppose \( y = 5 \) when \( x = 1 \) and \( y \) equals 10 when \( x = 2 \). Then there is a vertical gap of four dots between the points \((1.5)\) and \((2.10)\). This didn’t happen on the parabola graphic because \( x \) went from 1 to 5 in 1999 steps, so each step represented a change of about 0.003. Thus, even at the steepest part of the curve, \( y \) (continued)

Listing 4: Bresenham’s line drawing algorithm. (The Pascal implementation is courtesy of Professor Richard Rasias of Northeastern University)

```
Procedure PixelLine(x1,y1,x2,y2:integer);
var x, y, z, a, b, dx, dy, d, de1ta, delfaq, integer;
begin
  dx := abs(x2 - x1);
  dy := abs(y2 - y1);
  if dy <= dx then begin [Store <= 1] 
    begin
      x = x1; [initialize x]
      y = y1; [initialize y]
      z = dx; [set sentinel in x direction]
      [Now set x-incmement]
      if x1 < x2 then
        a = 1; [x increases]
      else
        a = -1; [x decreases]
      [Now set y-increment]
      if y1 < y2 then
        b = 1; [y increases]
      else
        b = -1; [y decreases]
      [Initialize decision function and its delta]
      deltay = dy + b * dy;
      d = deltay - dx;
      deltad = d;
      [Locate and plot pare]
      Pixel(x1,y1); [First point]
      while x < x2 do begin
        x := x + a;
        if d < 0 then
          d := d + deltad
        else
          d := d + deltax
        y := y + b
        end; [else]
      Pixel(x2,y2); [Last point]
      end; [white]
else begin
  [dx < dy] so view x as a function of y]
  begin
    y := y1; [initialize y]
    x := x1; [initialize x]
    z := dy; [set sentinel in y direction]
    [Now set y-increment]
    if y1 < y2 then
      a = 1; [y increases]
    else
      a = -1; [y decreases]
    [Initialize decision function and its delta]
    if x1 < x2 then
      b = 1; [x increases]
    else
      b = -1; [x decreases]
  end; [white]
end; [Case: if dx < dy]
```

(continued)
HI-RES PRINTER GRAPHICS

CopyWrite
BACKS UP
IBM PC SOFTWARE

Hundreds of the most popular copy-protected programs are copied readily. CopyWrite needs no complicated parameters. It needs an IBM Personal Computer, or an XT or an AT, 128K bytes of memory, and one diskette drive. CopyWrite will run faster with more memory or another drive.

CopyWrite is revised monthly to keep up with the latest in copy-protection. You may get a new edition at any time for a $15 trade in fee.

CopyWrite makes back up copies to protect you against accidental loss of your software. It is not for producing copies for sale or trade, or for any other use that deprives the author of payment for his work.

To order CopyWrite, send a check for $50 U.S. or call us with your credit card. We will ship the software within a day.

Quaid Software Limited
45 Charles Street East
Third Floor
Toronto, Ontario M5Y 1Z2
(416) 961-8243

Ask about ZenDisk to see copy-protected software from a hard disk without floppy's.

changes only about 1.5 dots per change in x—hardly visible at over 200 dots per inch.)

Second, this point-by-point calculation takes time. Even when the curve is smooth or nearly straight, every point must be calculated. For curves from simple functions this doesn't produce too much overhead, but for complicated mathematical equations or for curves produced by rotating images, this "overcalculation" is unacceptably slow.

The solution to both of these problems is to compute fewer points and to join the points computed with simple, easy-to-calculate curves. For most purposes these simple curves can be taken to be straight lines. If you only compute every fifth point and you connect the points by lines, there is a considerable time savings if point computations are reasonably complex and the line-drawing algorithm is fast. Furthermore, this solves the problem of gaps, since, in the example above, the points (1,5) and (2,10) would be joined by a small line segment "filling in" the missing four points.

The problem, then, is finding a fast line-drawing algorithm. Trying to find the equation of the line joining two points and then plotting it requires a considerable amount of real-number (decimal) arithmetic. This kind of arithmetic, especially multiplication and division, is quite slow in comparison with whole-number manipulation. Furthermore, since the coordinates of points on the screen (or printer page) are always integers—column and row numbers—you would naturally hope for a whole-number algorithm. Fortunately, there is one, called the Bresenham Line Algorithm (named for its inventor, J. E. Bresenham). It not only computes the points on the line connecting any two screen points using whole-number arithmetic, but it accomplishes this feat without using either multiplication or division!

Listing 4 contains a Pascal implementation of this. The procedure call is Pixel_line(x1,y1,x2,y2,color) where x1,y1 and x2,y2 are the endpoints of the line. For an easy-to-read description of the theory behind Pixel_line, see Fundamentals of Interactive Computer Graphics by James D. Foley and Andries Van Dam (Addison-Wesley, 1982). Sometimes, when speed is even more important and points are very time-consuming to compute, you must cut down radically on the number of points calculated. Joining the points by straight lines will usually produce a figure that is too polygonal in appearance. In figure 1, the points are joined by curved pieces called splines, for which there are now very fast computational algorithms. There is some discussion of splines in Foley and Van Dam's book, but the

(diminished)
There are clever ways of getting even more speed out of the line drawing—especially for lines of small slope—by exploiting block moves of bytes.

Most efficient algorithms are to be found in the current technical computer science journals.

Further Applications and Extensions

Armed with procedures for drawing points and lines on the screen and on the printer, you can implement procedures for making very complex high-resolution pictures. It is possible, given enough memory, to set aside more pairs of arrays to increase further the image size you can print. This is the reason to use dynamic variables, the ones with the "".

It is also possible to print your picture sideways, but this requires a restructuring of the procedure Change so that it addresses the points correctly.

Finally, you can use pixel masks to draw points on the graphics screen as well as the printer. The point and line-drawing procedures included in BASIC and Turbo Pascal, for example, are implemented by combining color and monochrome pixel masks with some version of Breenham line drawing. There are clever ways of getting even more speed out of the line drawing—especially for lines of small slope—by exploiting block moves of bytes.

Figure 2 shows a surface plotted by an Epson FX-80 printer with a resolution of 1600 by 640 dots. It indicates the complexity of drawing possible with this method of printer addressing.