Small pictures can be “typeset” on raster devices in a way that simulates the screens used to print fine books on photography. The purpose of this note is to discuss some experiments in which METAFONT has created fonts from which halftones can be generated easily on laser printers. High levels of quality are not possible at low resolution, and large pictures will overflow \TeX’s memory at high resolution; yet these fonts have proved to be useful in several applications, and their design involves a number of interesting issues.

I began this investigation several years ago when about a dozen of Stanford’s grad students were working on a project to create “high-tech self-portraits” [see Ramsey Haddad and Donald E. Knuth, “A programming and problem-solving seminar.” Stanford Computer Science Report 1055 (Stanford, California, June 1985), pp. 88–103]. The students were manipulating digitized graphic images in many ingenious ways, but Stanford had no output devices by which the computed images could be converted to hardcopy. Therefore I decided to create a font by which halftones could be produced using \TeX.

Such a font is necessarily device-dependent. For example, a laser printer with 300 pixels per inch cannot mimic the behavior of another with 240 pixels per inch, if we are trying to control the patterns of pixels. I decided to use our 300-per-inch Imagen laserprinter because it gave better control over pixel quality than any other machine we had.

It seemed best at first to design a font whose “characters” were tiny \(8 \times 8\) squares of pixels. The idea was to have 65 characters for 65 different levels of brightness: For \(0 \leq k \leq 64\) there would be one character with exactly \(k\) black pixels and \(64 - k\) white pixels.

Indeed, it seemed best to find some permutation \(p\) of the 64 pixels in an \(8 \times 8\) square so that the black pixels of character \(k\) would be \(p_0, p_1, \ldots, p_{k-1}\). My first instinct was to try to keep positions \(p_0, p_1, p_2, \ldots\) as far apart from each other as possible. So my first \TeX program painted pixels black by ordering the positions as follows:

\[
\begin{array}{ccccccccc}
45 & 29 & 34 & 18 & 46 & 30 & 53 & 17 \\
13 & 61 & 2 & 50 & 14 & 62 & 1 & 49 \\
39 & 23 & 40 & 24 & 36 & 20 & 43 & 27 \\
7 & 55 & 8 & 56 & 4 & 52 & 11 & 59 \\
47 & 31 & 13 & 2 & 16 & 44 & 28 & 35 & 19 \\
15 & 3 & 63 & 0 & 48 & 12 & 60 & 3 & 51 \\
37 & 1 & 21 & 42 & 26 & 38 & 22 & 41 & 25 \\
5 & 53 & 10 & 58 & 6 & 54 & 9 & 57
\end{array}
\]

[This is essentially the “ordered dither” matrix of B. E. Bayer; see the survey paper by Jarvis, Judice, and Ninke in Computer Graphics and Image Processing 5 (1976), 22–27.]

It turns out to be easy to create such a font with METAFONT:

\begin{verbatim}
% halftone font with 65 levels of gray, characters "0" (white) to "p" (black)
pair p[]; % the pixels in order (first p0 becomes black, then p1, etc.)
pair d[]; d[0]=(0,0); d[1]=(1,1); d[2]=(0,1); d[3]=(1,0); % dither control
def wrap(expr z)=(xpart z mod 8, ypart z mod 8) enddef;
for i=0 upto 3: for j=0 upto 3: for k=0 upto 3:
  p[16i+4j+k]=wrap(4d[k]+2d[j]+d[i]+(2,2)); endfor endfor endfor
w#: =8/pt; % that's 8 pixels
font_quad:=w#; designsize:=8w#;
picture prevchar; prevchar=nullpicture; % the pixels blackened so far
for i=0 upto 64:
  beginchar(i+ASCII"O",w#,w#,0); currentpicture:=prevchar;
  if i>0: addto currentpicture also unitpixel shifted p[i-1]; fi
  prevchar:=currentpicture; endchar;
endfor
\end{verbatim}
This file was called dt.mf; I used it to make a font called ‘dt300’ by applying \texttt{METAFONT} in the usual way to the following file dt300.mf:

\begin{verbatim}
% Halftone font for Imagen, dithered
mode_setup;
if (pixels_per_inch<>300) or (mag<>1):
  errmessage "Sorry, this font is only for resolution 300";
  errmessage "Abort the run now or you'll clobber the TFM file";
forever: endfor
else: input dt fi
end.
\end{verbatim}

(The purpose of dt300.mf is to enforce the device-dependence of this font.)

It’s fairly easy to typeset pictures with dt300 if you input the following macro file hf65.tex in a \TeX{} document:

\begin{verbatim}
\font\halftone=dt300 % for halftones on the Imagen 300
\chardef\other=12
\def\beginhalftone{\vbox\bgroup\offinterlineskip\halftone
 \catcode`\\=\other \catcode`\-=\other \catcode`\-=\other
 \catcode`\-=\active \starthalftone}
{\catcode`\-=\active \catcode`\/=0 \catcode`\-=\other
 /gdef/starthalftone#l/endhalftone{/let.=/endhalftoneline
 /beginhalftoneline#l/endhalftone}}
\def\beginhalftoneline{\hbox\bgroup\ignorespaces}
\def\endhalftoneline{\egroup\beginhalftoneline}
\def\endhalftone{\egroup\setbox0=\lastbox\unskip\egroup}
% Example of use:
% \beginhalftone
% \chardef \other = 12
% \beginhalftoneline
% \chardef \other = 12
% \beginhalftoneline
% \beginhalftone
% chars for top line of picture.
% chars for second line of picture.
% ...
% chars for bottom line of picture.
% \endhalftone
\end{verbatim}

(These macros are a bit tricky because ‘\’ is one of the legal characters in \texttt{dt300}; we must make backslashes revert temporarily to the status of ordinary symbols.)

Unfortunately, the results with \texttt{dt300} weren't very good. For example, here are three typical pictures, shown full size as they came off the machine:

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{image1}
\includegraphics[width=0.3\textwidth]{image2}
\includegraphics[width=0.3\textwidth]{image3}
\caption{The squareness of the pixels is much too prominent.}
\end{figure}

* Asterisks are used throughout this paper to denote places where output from the 300-pixels-per-inch Imagen printer has been pasted in. Elsewhere, the typesetting is by an APS Micro-5, which has a resolution of about 723 pixels per inch.
Moreover, the laser printer does strange things when it is given pixel patterns like those in \texttt{dt300}:*

\begin{verbatim}
  62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\end{verbatim}

Although character $k$ has more black pixels than character $k-1$, the characters do not increase their darkness monotonically! Character 6 seems darker than character 7; this is an optical illusion. Character 32 is darker than many of the characters that follow, and in this case the effect is not illusory: Examination with a magnifying glass shows that the machine deposits its toner in a very curious fashion.

Another defect of this approach is that most of the characters are quite dark; 50\% density is reached already at about character number 16. Hence \texttt{dt300} overemphasizes light tones.

My next attempt was to look at halftone pictures in books and newspapers, in order to discover the secret of their success. Aha! These were done by making bigger and bigger black dots; in other words, the order of pixels $p_0, p_1, \ldots$ was designed to keep black pixels \textit{close together} instead of far apart. Also, the dots usually appear in a grid that has been rotated 45\%, since human eyes don't notice the dottiness at this angle as much as they do when a grid is rectilinear. Therefore I decided to blacken pixels in the following order:

\begin{verbatim}
  62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\end{verbatim}

Here I decided not to stick to an 8 \times 8 square; this nonsquare set of pixel positions still "tiles" the plane in Escher-like fashion, if we replicate it at 8-pixel intervals. The characters are considered to be 8 pixels wide and 8 pixels tall, as before, but they are no longer confined to an 8 \times 8 bounding box. The reference point is the lower left corner of position 24.

The matrix above is actually better than the one I first came up with, but I've forgotten what that one was. John Hobby took a look at mine and suggested this alternative, because he wanted the pattern of \textit{black} pixels in character $k$ to be essentially the same as the pattern of \textit{white} pixels in character $64-k$. (Commercial halftone schemes start with small black dots on a white background; then the dots grow until they form a checkerboard of black and white; then the white dots begin to shrink into their black background.) The matrix above has this symmetry property, because the sum of the entries in positions $(i,j)$ and $(i,j+4)$ is 63 for all $i$ and $j$, if you consider "wraparound" by computing indices modulo 8.

John and I used this new ordering of pixel positions to make a font called \texttt{dot300}, analogous to \texttt{dt300}. It has the following gray levels:*

\begin{verbatim}
  62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\end{verbatim}

Now we have a pleasantly uniform gradation, except for an inevitable anomaly between characters 62 and 63. The density reaches 50\% somewhere around character number 45, and we can compensate for this by preprocessing the data to be printed.
The three images that were displayed with dt300 above look like this when dot300 is used:

My students were able to use dot300 successfully, so I stopped working on halftones and resumed my normal activities.

However, I realized later that dot300 can easily be improved, because each of its characters is made up of two dots that are about the same size. There's no reason why the dots of a halftone image need to be paired up in such a way. With just a bit more work, we can typeset each dot independently!

Thus, I made a font hf300 with just 33 characters (not 65 as before), using the matrix

<table>
<thead>
<tr>
<th>31</th>
<th>19</th>
<th>23</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>26</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>21</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>22</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>20</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

to control the order in which pixels are blackened. (This matrix corresponds to just one of the two dots in the larger matrix above.) The characters are still regarded as 8 pixels wide, but they are now only 4 pixels tall. When a picture is typeset, the odd-numbered rows are to be offset horizontally by 4 pixels.

Here is the METAFONT file hf.mf that was used to generate the single-dot font:

```plaintext
% halftone font with 33 levels of gray, characters "0" (white) to "P" (black)
pair p[]; % the pixels in order (first p0 becomes black, then p1, etc.)
p0=(1,1); p4=(2,0); p8=(1,0); p12=(0,0);
p16=(3,-1); p20=(2,-1); p24=(1,-1); p28=(2,-2);
transform r; r=identity rotatedaround ((1.5,1.5),90);
for i=0 step 4 until 28:
p[i+1]=p[i] transformed r;
p[i+3]=p[i+1] transformed r;
p[i+2]=p[i+3] transformed r;
endfor
w#:=8/pt; % that's 8 pixels
font_quad:=w#; designsize:=8w#;
picture prevchar; prevchar=nullpicture; % the pixels blackened so far
for i=0 upto 32:
beginchar(i+ASCII"0",w#, .5w#, 0); currentpicture:=prevchar;
if i>0: addto currentpicture also unitpixel shifted p[i-1]; fi
prevchar:=currentpicture; endchar;
endfor
```
There's also a file hf300.mf, analogous to the file dt300.mf above. Here's how the three example images look when they're rendered by font hf300:

They are somewhat blurry because they were generated second-hand from data intended for square pixels; sharper results are possible if the data is expressly prepared for a 45° grid. For example, here is a sharper Mona Lisa, and an image whose dots were computed directly by mathematical formulas:

The \TeX macros hf65.tex shown above must be replaced by another set hf33.tex when independent dots are used:

```latex
\font\halftone=hf300 % for halftones on the Imagen 300, each dot independent
\chardef\other=12
\newif\ifshifted
\def\shift\{\makebox{\vbox{\offinterlineskip\halftone
  \catcode'\=.=\active\shiftedtrue\shift\hbox{\group}}
\catcode'\.=\active \gdef{\egroup
  \ifshifted\shiftedfalse \else\shiftedtrue \fi\hbox{\group}\ignorespaces})
\def\endhalftone{\egroup\setbox0=\lastbox\egroup}

% Example of use:
% \beginhalftone
% \other\shift\ch for top halfline of picture. (shifted right 4 pixels)
% \other\shift\ch for second halfline of picture. (not shifted right)
% \other\shift\ch for third halfline of picture. (shifted right 4 pixels)
% ...% \other\shift\ch for bottom halfline of picture. (possibly shifted right)
% \endhalftone
```

These macros are much simpler than those of hf65, because the 33 ASCII characters "0" to "P" have no special meaning to plain \TeX.
We can also create an analogous font \texttt{hf723} for the high-resolution APS, in which case the pictures come out looking like this:

The same \TeX macros were used, but font \texttt{\textbackslash halftone} was defined to be \texttt{hf723} instead of \texttt{hf300}. Now the pictures are smaller, because the font characters are still 8 pixels wide, and the pixels have gotten smaller. At this resolution the halftones look “real,” except that they are too dark. This problem can be fixed by adjusting the densities in a preprocessing program. Also, small deficiencies in the APS’s analog-to-digital conversion hardware become apparent when such tiny characters are typeset.

What resolution is needed? It is traditional to measure the quality of a halftone screen by counting the number of dots per inch in the corresponding unrotated grid, and it’s easy to do this with a magnifying glass. The photographs in a newspaper like the \textit{International Herald Tribune} use a 72-line screen, rotated 45°; this is approximately the resolution $50\sqrt{2}$ that we would obtain with the \texttt{hf400} font on a laser printer with 400 dots per inch. (The 300-per-inch font \texttt{hf300} gives a rotated screen with only $37.5\sqrt{2} \approx 53$ dots per inch.) The photographs on the book jacket of \textit{Computers & Typesetting} have a 133-line screen, again rotated 45°; this is almost identical to the resolution of \texttt{hf723}. But this is not the upper limit: A book that reproduces photographs with exceptionally high quality, such as \textit{Portraits of Success} by Carolyn Caddes (Portola Valley: Tioga Press, 1986), has a screen of about 270 lines per inch, in this case rotated 30°.

Let’s turn now to another problem: Suppose we have an image for which we want to obtain the best possible representation on a laser printer of medium resolution, because we will be using that image many times—for example, in a letterhead. In such cases it is clearly desirable to create a special font for that image alone; instead of using a general-purpose font for halftones, we’ll want to control every pixel. The desired image can then be typeset from a special-purpose font of “characters” that represent rectangular subsections of the whole.

The examples above were produced on an Imagen printer as 64 lines of 55 columns per line, with 8 pixels in each line and each column. To get an equivalent picture with every pixel selected individually, we can make a font that has, say, 80 characters, each 64 pixels tall and 44 pixels wide. By typesetting eight rows of ten characters each, we’ll have the desired image. For example, the following picture was done in that way:

\begin{verbatim}
\font\mona=mona300[hf,dek]
\newcount\n\newcount\m
\def\monalisa\vbox{\mona \offinterlineskip \n=0
   \loop \hbox{\m=0 \loop \char\n \global\advance\n by 1
      \advance\m by 1 \ifnum\m<10 \repeat}
   \ifnum\n<80 \repeat}\
\end{verbatim}
And once we have the individual pieces, we can combine them to get unusual effects.*

The font mona300 shown above was generated from a file mona.mf that began like this:

```
row(1); cols(1,5,9,13,15,17,21,24,30,32,39,46,56,62,70,
    78,86,95,103,110,118,120,127,135,142,151,159,167,175,183,
    311,318,328,334,342,350,358,366,375,382,383,390,392,398,
    400,405,408,416,421,424,430,432,439);
row(2);
```

... and so on, until 512 rows had been specified. The parameter file mona300.mf was

```
% Mona Lisa for Imagen 300
mode_setup;
if (pixels_per_inch<>300) or (mag<>1): ... (error messages as before)
else: input picfont
    width:=44; height:=64; m:=8; n:=10; filename:="mona";
do_it; fi
end.
```

and the driver file picfont.mf was

```
def do_it=
    for j=0 upto n-1: jj:=width*j; jjj:=jj+width; jjjj:=j;
    scantokens("input \&filename"); endfor enddef;
string filename;
def row(expr x) =
    cc:=(x-1)div height; rr:=height-1-((x-1)mod height);
    if rr=height-1: beginchar(cc*n+jjjj,width/pt,height/pt,0); fi enddef;
def cols(text t) =
    for tt:=t: exitif tt>=jjj; if tt>=jj:
        addto currentpicture also unitpixel shifted (tt,rr); fi endfor
    if rr=0: xoffset:=-jj; endchar; fi enddef;
```

This is not very efficient, but it's interesting and it seems to work.
Ken Knowlton and Leon Harmon have shown that surprising effects are possible once a picture has been digitized [see Computer Graphics and Image Processing 1 (1972), 1–20]. Continuing this tradition, I found that it’s fun to combine the \TeX macros above with new fonts that frankly acknowledge their digital nature. One needn’t always try to compete with commercial halftone screens!

For example, we can use hf65.tex with a ‘negdot’ font that makes negative images out of square dots:

The METAFFONT file negdot.mf that generated this font is quite simple:

```latex
\% negative pseudo-halftone font with 65 sizes of square dots
mode_setup;
w#:={2.5pt#}; font_quad:=w#; designsize:=8w#;
for i=0 upto 64:
  beginchar(i+ASCII"0",w#,w#,0);
  r#:={sqrt(.9w###(1-i/80))}; define_pixels(r);
  fill unitsquare scaled r shifted(.5w,.5h);
  endchar;
endfor
end.
```

Unlike the previous fonts we have considered, this one is device-independent.

It’s even possible to perceive images when each character of the halftone font has exactly the same number of black pixels. Here, for example, is what happens when the three images above are typeset with a font in which each character consists of a vertical line and a horizontal line; the lines move up and to the right as the pixel gets darker, but they retain a uniform thickness. We perceive lighter and darker features only because adjacent lines get closer together or further apart.
The METAFONT file lines.mf for this device-independent font is:

```
% pseudo-halftone font with 65 lines that move right and up
mode_setup; q:=savepen;
w#:=.5pt#; font_quad:=w#; designsize:=8w#
for i=0 upto 64: beginchar(i+ASCII"0",w#,w#,0); pickup q;
draw (0,h*i/64)-(w,h*i/64); draw(w*i/64,0)-(w*i/64,h); endchar;
endfor end.
```

Yet another possibility is the font produced by angles.mf; here each character is a single line of radius 2.5 pt that rotates from horizontal to vertical as the density increases:

```
% pseudo-halftone font with 65 radii that move counterclockwise
mode_setup; q:=savepen;
w#:=.5pt#; font_quad:=w#; designsize:=8w#
for i=0 upto 64: beginchar(i+ASCII"0",w#,w#,0); pickup q;
draw ((0,0)--(w,0)) rotated (90*i/64); endchar;
endfor end.
```

The images are still amazingly easy to identify:

(The We can think of a large array of dials whose hands record the local light levels.) It is amusing to view these images by tilting the page up until your eyes are almost parallel to the paper.

As a final example, let’s consider a 33-character font that’s designed to be used with $\texttt{hf33.tex}$ instead of $\texttt{hf65.tex}$. Readers who like puzzles are invited to try to guess what this METAFONT code will do, before looking at the image of Mona Lisa that was typeset with the corresponding font. [Hint: The name of the METAFONT file is hex.mf.]

```
% pseudo-halftone font with 33 more-or-less hexagonal patterns
mode_setup; q:=savepen;
w#:=.5pt#; font_quad:=w#; designsize:=8w#
for i=0 upto 32: beginchar(i+ASCII"0",w#,.5w#,0); pickup q;
alpha:=.5-i/72; z0=(.5w,.5h);
z1=alpha[(5/6w,.5h),z0]; z2=alpha[(2/3w,-.5h),z0];
z0=.5[z2,z5]=.5[z3,z6]=.5[z1,z4]; x2=x6; y5=y6;
draw z1--z2; draw .5[z1,z2]--z0;
draw z3--z4; draw .5[z3,z4]--z0;
draw z5--z6; draw .5[z5,z6]--z0; endchar;
endfor end.
```

The answer to this puzzle can be seen in the illustration at the very end of this paper (following the appendices).
Appendix 1: Source data for the examples

The examples in this paper are mostly derived from the basic pixel values shown below. This data uses a convention taken from the book *Digital Image Processing* by Rafael C. Gonzalez and Paul Wintz (Addison-Wesley, 1977): The 32 characters 0123456789ABCDEFGHIJKLMNOPQRSTUV represent densities from 1.0 down to 0.0.

(Lisa)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

(Lincoln)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

Source data for the examples

The examples in this paper are mostly derived from the basic pixel values shown below. This data uses a convention taken from the book *Digital Image Processing* by Rafael C. Gonzalez and Paul Wintz (Addison-Wesley, 1977): The 32 characters 0123456789ABCDEFGHIJKLMNOPQRSTUV represent densities from 1.0 down to 0.0.
to 0.0 (i.e., '0' is black and 'V' is white). The Lisa data was digitized by a TV camera in Stanford's robotics lab. The Lincoln and Liberty data come mostly from Appendix B in the Gonzales-Wintz book, although I decided to change several dozen of the pixel values found there.

QQQQQQQQQPPPPOPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.
QQQQQQQQQQQPPPPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.
QQQQQQQQQQQPPPPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.
QQQQQQQQQQQPPPPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.
QQQQQQQQQQQPPPPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.
QQQQQQQQQQQPPPPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.
QQQQQQQQQQQPPPPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.
QQQQQQQQQQQPPPPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.
QQQQQQQQQQQPPPPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.
QQQQQQQQQQQPPPPOPO. VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV.

VUGboat, Volume 8 (1987), No. 2
Appendix 2: 65-level data for typesetting

The 32-level data in Appendix 1 was converted to 65-level data by a computer program similar to (but simpler than) the program in Appendix 3. In this case the representation is different: ‘0’ means white and ‘p’ means black. The 65 codes follow standard ASCII order: 0123456789 :<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ

Lincoln

(TUGboat, Volume 8 (1987), No. 2)
The density of character $k$ was assumed to be approximately $k/36$ for $0 \leq k \leq 8$, and approximately $k/72 + 1/9$ for $8 \leq k \leq 64$. 

\begin{verbatim}
WXYZ[ ]`- abcdefgijklmnop.
\end{verbatim}
Appendix 3: Transforming the pixel data
The following WEB program illustrates how to convert data like that of Appendix 1 into the form required by the fonts and macros described earlier.

1. Introduction. This program prepares 33-level halftone images for use in \TeX files. The input is assumed to be a sequence of pictures expressed in the form

\[
m \ n \\
\vdots \\
(\text{first line of pixel data, } n \ \text{characters long}) \\
\ldots \\
(\text{mth line of pixel data, } n \ \text{characters long})
\]
terminated by a line that says simply '0'. The pixel data consists of the characters "0" to "9" and "A" to "V", representing 32 levels of darkness from black to white. [See Appendix 1.]

The output is the same set of pictures, expressed in a simple format used for 33-level halftones, with ASCII characters "0" to "P" representing darkness levels from white to black. The levels are adjusted to compensate for the idiosyncrasies of Canon LBP-CX laser-printing engines. Two dots are typeset for each pixel of input; hence there are \(2m\) "halflines" of \(n\)-character data in the output.

2. Here’s an outline of the entire Pascal program:

```pascal
program halftones(input, output);
  label (Labels in the outer block 4)
  const (Constants in the outer block 3)
  type (Types in the outer block 5)
  var (Global variables 6)
procedure initialize; { this procedure gets things started properly }
  var (Local variables for initialization 8)
  begin (Set initial values 7)
  end;
begin initialize; { The main program 23};
end.
```

3. Each picture in the input data must contain fewer than \(max.m\) rows and \(max.n\) columns.

(Labels in the outer block 4) \(\equiv\)

\[
\begin{align*}
\text{max.m} & = 200; \quad \{m \text{ should be less than this}\} \\
\text{max.n} & = 200; \quad \{n \text{ should be less than this}\}
\end{align*}
\]
This code is used in section 2.

4. The main program has one statement label, namely cleanup_and_terminate.

\[
\begin{align*}
\text{define cleanup_and_terminate} & = 9998 \\
\text{define finish} & \equiv \text{goto cleanup_and_terminate} \quad \{\text{do this when all the pictures have been output}\}
\end{align*}
\]
(Labels in the outer block 4) \(\equiv\)

\[
\begin{align*}
\text{cleanup_and_terminate;}
\end{align*}
\]
This code is used in section 2.

5. The character set. We need translation tables between ASCII and the actual character set, in order to make this program portable. The standard conventions of \TeX: The Program are copied here, essentially verbatim.

\[
\begin{align*}
\text{define text.char} & \equiv \text{char} \quad \{\text{the data type of characters in text files}\} \\
\text{define first.text.char} & = 0 \quad \{\text{ordinal number of the smallest element of text.char}\} \\
\text{define last.text.char} & = 127 \quad \{\text{ordinal number of the largest element of text.char}\}
\end{align*}
\]
(Labels in the outer block 5) \(\equiv\)

\[
\begin{align*}
\text{ASCII.code} & = 0 .. 127; \quad \{\text{seven-bit numbers}\}
\end{align*}
\]
This code is used in section 2.
6. (Global variables $e$) \equiv
\begin{itemize}
\item $xord$: array [text_char] of ASCII\_code; \{ specifies conversion of input characters \}
\item $zchr$: array [ASCII\_code] of text_char; \{ specifies conversion of output characters \}
\end{itemize}
See also sections 10, 13, 14, and 22.

This code is used in section 2.

7. (Set initial values $\tau \equiv$
\begin{verbatim}
xchr['40'] ← 'A';  xchr['41'] ← 'B';  xchr['42'] ← 'C';  xchr['43'] ← 'D';  xchr['44'] ← 'E';  xchr['45'] ← 'F';  xchr['46'] ← 'G';  xchr['47'] ← 'H';  xchr['50'] ← 'I';  xchr['51'] ← 'J';  xchr['52'] ← 'K';  xchr['53'] ← 'L';  xchr['54'] ← 'M';  xchr['55'] ← 'N';  xchr['56'] ← 'O';  xchr['57'] ← 'P';  xchr['58'] ← 'Q';  xchr['59'] ← 'R';  xchr['60'] ← 'S';  xchr['61'] ← 'T';  xchr['62'] ← 'U';  xchr['63'] ← 'V';  xchr['64'] ← 'W';  xchr['65'] ← 'X';  xchr['66'] ← 'Y';  xchr['67'] ← 'Z';  xchr['70'] ← 'a';  xchr['71'] ← 'b';  xchr['72'] ← 'c';  xchr['73'] ← 'd';  xchr['74'] ← 'e';  xchr['75'] ← 'f';  xchr['76'] ← 'g';  xchr['77'] ← 'h';  xchr['78'] ← 'i';  xchr['79'] ← 'j';  xchr['80'] ← 'k';  xchr['81'] ← 'l';  xchr['82'] ← 'm';  xchr['83'] ← 'n';  xchr['84'] ← 'o';  xchr['85'] ← 'p';  xchr['86'] ← 'q';  xchr['87'] ← 'r';  xchr['88'] ← 's';  xchr['89'] ← 't';  xchr['90'] ← 'u';  xchr['91'] ← 'v';  xchr['92'] ← 'w';  xchr['93'] ← 'x';  xchr['94'] ← 'y';  xchr['95'] ← 'z';
\end{verbatim}

See also sections 9, 11, 15, and 17.

This code is used in section 2.

8. (Local variables for initialization $s$) \equiv
\begin{itemize}
\item $i$: 0 .. last\_text\_char;
\end{itemize}
This code is used in section 2.

9. (Set initial values $\tau \equiv$
\begin{verbatim}
for $i ← 1$ to 37 do xchr[$i$] ← 'a';
for $i ← 1$ to 176 do xord[chr[$i$]] ← 177;
\end{verbatim}

10. **Inputting the data.** We keep the pixel values in a big global array called $v$. The variables $m$ and $n$ keep track of the current number of rows and columns in use.

The $dd$ table contains density values assumed for the input, indexed by single-character codes.

(Global variables $e$) \equiv
\begin{itemize}
\item $v$: array [0 .. max.m \times 0 .. max.n] of real; \{ pixel darknesses, from 0.0 to 1.0 \}
\item $m$: integer; \{ rows 0 \ldots m + 1 of $v$ should contain relevant data \}
\item $n$: integer; \{ columns 0 \ldots n + 1 of $v$ should contain relevant data \}
\item $dd$: array [text_char] of real;
\end{itemize}
11. All input codes give zero density, except "0" to "9" and "A" to "V".

(set initial values 7)

\[
\text{for } i \leftarrow \text{first text char to last text char do } dd[\text{chr}(i)] \leftarrow 0.0;
\]
\[
\text{for } i \leftarrow "0" \text{ to } "9" \text{ do } dd[\text{chr}(i)] \leftarrow 1.0 - (i - "0") / 31.0;
\]
\[
\text{for } i \leftarrow "A" \text{ to } "V" \text{ do } dd[\text{chr}(i)] \leftarrow 1.0 - (i - "A" + 10) / 31.0;
\]

12. The process of inputting pixel values is quite simple. We terminate the program if anomalous values of 
\( m \) and \( n \) occur. Boundary values are added at the top, left, right, and bottom in order to provide “padding” 
that will be convenient in the pixel transformation process. Each boundary value is equal to one of its 
adjacent neighbors.

(Input a picture, or terminate the program 12)

\[
\text{if } (m \leq 0) \lor (n \geq \text{max } m) \text{ then finish;}
\]
\[
\text{if } (n \leq 0) \lor (n \geq \text{max } n) \text{ then finish;}
\]
\[
\text{for } i \leftarrow 1 \text{ to } m \text{ do }
\]
\[
\begin{align*}
\text{begin for } j \leftarrow 1 \text{ to } n \text{ do } \varepsilon[i,j] \leftarrow dd[c]; \\
\text{end:}
\end{align*}
\]
\[
\text{v}[i,0] \leftarrow v[i,1]; \ v[i,n+1] \leftarrow v[i,n];
\]
\[
\text{read } \ln (c); \text{ end:}
\]
\[
\text{for } j \leftarrow 0 \text{ to } n + 1 \text{ do }
\]
\[
\begin{align*}
\text{begin } v[0,j] \leftarrow v[1,j]; \ v[m + 1,j] \leftarrow v[m,j]; \\
\text{end}
\end{align*}
\]

This code is used in section 23.

13. The code just written makes use of three temporary registers that must be declared:

\[
\begin{align*}
\text{(Global variables } 6) & \equiv \\
i,j & : \text{integer}; \ \{ \text{current row and column} \} \\
c & : \text{char}; \ \{ \text{character read from input} \}
\end{align*}
\]

14. Pixel compensation. The 33-level output of this program is assumed to be printed by a font that 
contains 4 \times 8 characters, where each character has 0 to 32 black bits. Physical properties of output devices 
distortions, so that a character with \( k \) black bits does not have an apparent density of \( k/32 \). We 
therefore maintain a table of apparent density values.

\[
\text{define } \text{max } l = 32 \quad \{ \text{maximum output level} \}
\]
\[
\text{(Global variables } 6) \equiv \\
d & : \text{array } [0 \ldots \text{max } l] \text{ of } \text{real}; \ \{ \text{apparent densities, from 0.0 to 1.0} \}
\]

15. This table is based on some densitometer measurements that are not especially reliable. The amount 
of toner seems to vary between the top of a page and the bottom; also blocks of the character "W" seem to 
appear darker than blocks of the character "O", because of some property of xerography, although the "0" 
has one more bit turned on. Such anomalies have been smoothed out here, since the resulting values should 
prove good enough in practice.

(set initial values 7)

\[
\begin{align*}
d[0] & = 0.0; \ d[1] = 0.06; \ d[2] = 0.095; \ d[3] = 0.125; \ d[4] = 0.155; \\
d[5] & = 0.175; \ d[6] = 0.215; \ d[7] = 0.245; \ d[8] = 0.27; \ d[9] = 0.29; \\
d[10] & = 0.3; \ d[11] = 0.31; \ d[12] = 0.32; \ d[13] = 0.33; \ d[14] = 0.34; \\
d[15] & = 0.35; \ d[16] = 0.36; \ d[17] = 0.37; \ d[18] = 0.38; \ d[19] = 0.4; \\
d[20] & = 0.42; \ d[21] = 0.44; \ d[22] = 0.47; \ d[23] = 0.5; \ d[24] = 0.53; \\
d[25] & = 0.57; \ d[26] = 0.61; \ d[27] = 0.66; \ d[28] = 0.72; \ d[29] = 0.8; \\
d[30] & = 0.88; \ d[31] = 0.96; \ d[32] = 1.0;
\end{align*}
\]
16. We convert the pixel values by using a variant of the Floyd-Steinberg algorithm for adaptive grayscale
[Society for Information Display, SID 75 Digest. 36-37]. The idea is to find the best available density, then
to diffuse the error into adjacent pixels that haven’t yet been processed.

The following code assumes that \( x \) is the desired density value in column \( j \) of the current halfline. It
outputs one 33-level density, then updates \( x \) and \( j \) in preparation for the next column. Adjustments to the
densities in the two next halflines are accumulated in auxiliary arrays \( \text{next1} \) and \( \text{next2} \); this will compensate
for errors in the current halfline.

We assume that \( \text{next1}[j], \text{next1}[j+1], \) and \( \text{next2}[j] \) correspond to the dots that are adjacent to \( \text{current}[j] \).

16. This code is used in sections 18 and 18.

17. The constants \( \alpha \ldots \delta \) control the distribution of errors to adjacent dot positions.

\[
\begin{align*}
\text{alpha} & \leftarrow 7/16: \{ \text{error diffusion to SW neighbor} \} \\
\text{beta} & \leftarrow 1/16: \{ \text{error diffusion to S neighbor} \} \\
\text{gamma} & \leftarrow 5/16: \{ \text{error diffusion to SE neighbor} \} \\
\text{delta} & \leftarrow 3/16: \{ \text{error diffusion to E neighbor} \}
\end{align*}
\]

18. Here is the overall control of the process. Every halfline of the picture being output is a sequence of
ASCII characters from "0" to "P", terminated by ".".

\[
\begin{align*}
\text{for } j & \leftarrow 1 \text{ to } n+1 \text{ do} \\
& \text{begin} \quad \text{next1}[j] \leftarrow 0.0; \text{next2}[j] \leftarrow 0.0; \\
& \quad \text{end}; \\
\text{for } i & \leftarrow 1 \text{ to } m \text{ do} \\
& \text{begin} \quad \text{(Set the current halfline data for the upper row of dots in line } i \text{ 19)}; \\
& \quad j \leftarrow 1; x \leftarrow \text{current}[1]; \\
& \quad \text{repeat} \quad \text{(Output one value and move to the next column 16)}; \\
& \quad \text{until } j > n; \\
& \quad \text{write ln(".")}; \quad \text{(Set the current halfline data for the lower row of dots in line } i \text{ 20)}; \\
& \quad j \leftarrow 1; x \leftarrow \text{current}[1]; \\
& \quad \text{repeat} \quad \text{(Output one value and move to the next column 16)}; \\
& \quad \text{until } j > n; \\
& \quad \text{write ln(".")}; \\
& \quad \text{end}
\end{align*}
\]

19. The density value for dot \( j \) in the upper halfline of line \( i \) is obtained as a weighted average of the input
values in rows \( i - 1 \) and \( i \), columns \( j \) and \( j + 1 \). The upper halfline is skewed to the right, so we must shift
\( \text{next1} \) and \( \text{next2} \) appropriately.

\[
\begin{align*}
\text{for } j & \leftarrow 1 \text{ to } n \text{ do} \\
& \text{begin} \quad \text{current}[j] \leftarrow (9 \times v[i,j] + 3 \times v[i,j+1] + 3 \times v[i-1,j] + v[i-1,j+1])/16 + \text{next1}[j+1]; \\
& \quad \text{next1}[j] \leftarrow \text{next2}[j]; \\
& \quad \text{end}; \\
& \quad \text{next1}[n+1] \leftarrow 0.0
\end{align*}
\]

This code is used in section 18.
20. The lower halfline is similar, but in this case there is leftward skew; we use rows \( i \) and \( i + 1 \), columns \( j - 1 \) and \( j \).

\[
\begin{align*}
\text{Set the current halfline data for the lower row of dots in line } i & \ 20) \\
\text{for } j & \ = 1 \ \text{to } n \ \text{do} \\
\text{begin current}[j] & \ = (9 \times v[i, j] + 3 \times v[i, j - 1] + 3 \times v[i + 1, j] + v[i + 1, j - 1])/16 + \text{nextl} [j]; \\
\text{nextl} [j + 1] & \ = \text{nextl} [j]; \\
\text{end}; \\
\text{nextl} [1] & \ = 0.0
\end{align*}
\]

This code is used in section 18.

21. The algorithm is now complete except for the part that chooses the closest possible dot size. A straightforward binary search works well for this purpose:

\[
\begin{align*}
\text{Find } l \text{ so that } d[l] & \text{ is as close as possible to } x \ 21) \\
\text{if } x & \leq 0.0 \ \text{then } l \ = 0 \\
\text{else if } x & > 1.0 \ \text{then } l \ = \ \text{max}.l \\
\text{else begin low}.l & \ = 0; \ \text{high}.l \ = \ \text{max}.l; \ \{ \text{we have } d[\text{low}.l] \leq x < d[\text{high}.l] \} \\
\text{while } \text{high}.l & \ = \ \text{low}.l > 1 \ \text{do} \\
\text{begin mid}.l & \ = (\text{low}.l + \text{high}.l) \ \text{div} 2; \\
\text{if } x & > d[\text{mid}.l] \ \text{then } \ \text{low}.l \ = \ \text{mid}.l \\
\text{else } \ \text{high}.l & \ = \ \text{mid}.l; \\
\text{end}; \\
\text{if } x & - d[\text{low}.l] \leq d[\text{high}.l] \ - x \ \text{then } l \ = \ \text{low}.l \ \text{else } l \ = \ \text{high}.l
\end{align*}
\]

This code is used in section 16.

22. We had better declare the variables we’ve been using.

\[
\text{Global variables } 6) \ + \ = \\
x: \ \text{real}; \ \{ \text{current pixel density} \} \\
\text{err}: \ \text{real}; \ \{ \text{difference between } x \text{ and the best we can achieve} \} \\
\text{current}: \ \text{array } [0... \ \text{max}.n] \ \text{of } \ \text{real}; \ \{ \text{desired densities in current halfline} \} \\
\text{nextl, next2}: \ \text{array } [0... \ \text{max}.n] \ \text{of } \ \text{real}; \ \{ \text{corrections to subsequent densities} \} \\
\alpha, \beta, \gamma, \delta: \ \text{real}; \ \{ \text{constants of error diffusion} \} \\
l, \ \text{low}.l, \ \text{mid}.l, \ \text{high}.l: \ 0... \ \text{max}.l; \ \{ \text{trial density levels} \}
\]

23. The main program. Now we’re ready to put all the pieces together.

\[
\begin{align*}
\text{(The main program 23) } & \ = \\
\text{write}.ln(\text{`}\text{input}.hf33`); \ \text{write}.ln; \\
\text{while } \text{true} \ \text{do} \\
\text{begin} \ \text{(Input a picture, or terminate the program 12)}; \\
\text{write}.ln(\text{`}\text{beginhalftone}`); \ \{ \text{Output the picture 18} \}; \\
\text{write}.ln(\text{`}\text{endhalftone}`); \ \text{write}.ln; \\
\text{end}; \\
\text{cleanup_and_terminate};
\end{align*}
\]

This code is used in section 2.
Appendix 4: Pixel optimization
Here is another short WEB program. It was used to generate the special font for Mona Lisa.

1. **Introduction.** This program prepares a METAFONT program for a special-purpose font that will approximate a given picture. The input is assumed to be a binary file that contains one byte of density information per pixel. The output will be a sequence of lines like

   row(10); cols(3, 15, 16, 17);

this means that bits 3, 15, 16, and 17 of the character for row 10 should be black.

2. Here's an outline of the entire Pascal program:

   ```pascal
   program picfont(bytes_in, output);
   type
     eight_bits = 0..255;  { unsigned one-byte quantity }
   var
     bytes_in: byte_file;  { files that contain binary data }
   begin
     The main program
   end.
   ```

3. The picture in the input data is assumed to contain \( mm \) rows and \( nn \) columns.

   ```pascal
   define mm = 512  { this many rows }
   define nn = 440  { this many columns }
   ```

4. It's convenient to declare a macro for incrementation.

   ```pascal
   define incr(#) := # + 1
   ```

5. **Inputting and outputting the data.** The input appears in a file of 8-bit bytes, with 00 representing black and FF representing white. There are \( mm \times nn \) bytes; they appear in order from top to bottom and left to right just as we normally read a page of text.

   ```pascal
   define eight_bits = 0..255;  { unsigned one-byte quantity }
   define byte_file = packed file of eight_bits;  { files that contain binary data }
   ```

   This code is used in section 2.

6. (Global variables 6)

   ```pascal
   bytes_in: byte_file;
   ```

   See also sections 9, 14, 16, 22, and 25.

   This code is used in section 2.

7. Different Pascal systems have different ways of dealing with binary files. Here is one common way.

   ```pascal
   define white = 0  { code for a white pixel with all white neighbors }
   define gray = 1  { code for a white pixel with 1, 2, 3, or 4 black neighbors }
   define black = 2  { code for a black pixel }
   define zeta = 0.2  { assumed darkness of white pixel with a black neighbor }
   ```

   This code is used in section 26.

8. We shall use the following model for estimating the effect of a given bit pattern: If a pixel is black, the darkness is 1.0; if it is white but at least one of its four neighbors is black, the darkness is \( \zeta \); if it is white and has four white neighbors, the darkness is zero.
9. There isn’t room to store all the input bytes in memory at once, but it suffices to keep buffers for about a dozen rows near the current area being computed.

(Global variables 6) \oplus

\( ii: \text{integer}; \) \{ the buffer holds rows \( 8ii - 7 \) through \( 8ii + 4 \) \}

\( \text{buffer: array [-2..9,0..nn + 1] of real}; \) \{ densities in twelve current rows \}

\( \text{darker: array [-3..9,0..nn + 1] of white .. black}; \) \{ darks in buffer rows \}

\( \text{new_row: array [0..nn + 1] of real}; \) \{ densities in row being input \}

10. The \text{get_in} procedure computes the densities in a specified row and puts them in \text{new_row}. This procedure is called successively for \( i = 1, 2, \ldots \).

(Basic procedures 10) \oplus

\textbf{procedure get_in(i: integer);} \{ \text{byte of input} \}

\begin{verbatim}
var j: integer; t: byte; \hspace{1em} \text{byte of input}
begin
new_row[0] := 0.0;
if i > mm then
  for j := 0 to nn do new_row[j] := 0.0
else for j := 0 to nn do
  read(bytes_in,t); new_row[j] := (255.5 - t)/256.0;
end;
new_row[nn + 1] := 0.0;
end:
\end{verbatim}

See also sections 11 and 20.
This code is used in section 2.

11. Here is a procedure that “rolls” the buffer down eight lines:

(Basic procedures 10) \oplus

\textbf{procedure roll;} \{ \text{eight_bits} \}

\begin{verbatim}
var j: 0..nn + 1; i: 2..9; k: integer;
begin
for i := 6 to 9 do
  for j := 0 to nn + 1 do
    begin
      buffer[i - 8,j] := buffer[i,j];
      darkness[i - 8,j] := darkness[i,j];
    end;
for j := 0 to nn + 1 do
darkness[-3,j] := darkness[5,j];
incr(ii);
for i := 2 to 9 do
  begin
    get_in(8 * ii + i - 5);
    for j := 0 to nn + 1 do
      begin
        buffer[i,j] := new_row[j];
        darkness[i,j] := white;
      end;
  end;
end:
\end{verbatim}

12. It’s tedious but not difficult to get everything started. We put zeros above the top lines in the picture.

(Initialize the buffers 12) \oplus

\( ii := 0; \)

\begin{verbatim}
for i := 6 to 9 do
  begin
    get_in(i - 5);
    for j := 0 to nn + 1 do
      begin
        buffer[i,j] := new_row[j];
        darkness[i,j] := white;
      end;
  end;
for i := -2 to 5 do
  for j := 0 to nn + 1 do
    begin
      buffer[i,j] := 0.0; darkness[i,j] := white;
    end;
for j := 0 to nn + 1 do
darkness[-3,j] := white
\end{verbatim}

This code is used in section 26.
13. It's easy to output the current darkness values. Here we output eight consecutive rows.

\[
\text{for } i \leftarrow -2 \text{ to } 5 \text{ do}
\begin{align*}
\text{begin} & \quad \text{write('row', 8 * i + 5 + i : 1, ';'); cols(');} \quad \text{cols.out} \leftarrow 0; \\
\text{for } j \leftarrow 1 \text{ to } \text{nn} \text{ do} & \\
\text{if } \text{darkness}[i, j] = \text{black} \text{ then} & \\
\text{begin} & \quad \text{if } \text{cols.out} > 15 \text{ then write(';', ')');} \\
& \quad \text{incr(cols.out);} \\
& \quad \text{end;} \\
\text{else begin} & \quad \text{write_in(';', ' ');} \quad \text{write('............'); cols.out} \leftarrow 1; \\
& \quad \text{end;} \\
& \quad \text{write}(j : 1); \\
& \quad \text{end;} \\
& \quad \text{write_in(''); ')'} \\
\text{end}
\end{align*}
\]

This code is used in section 26.

14. (Global variables 6) \[ \equiv \]
\[
cols.out : 0 \ldots 15; \quad \text{the number of columns output so far on this line}
\]

15. Dot diffusion. The pixels are divided into 64 classes, numbered from 0 to 63. We convert the pixel values to darknesses by using a method called “dot diffusion.” Values are assigned first to all the pixels of class 0, then to all the pixels of class 1, etc.; the error incurred at each step is distributed to the neighbors whose class numbers are higher. This is done by means of precomputed tables \text{class.row}, \text{class.col}, \text{start}, \text{del.i}, \text{del.j}, \text{and} \alpha \text{ whose function is easy to deduce from the following code:}

\[
\text{for } k \leftarrow 0 \text{ to } 63 \text{ do}
\begin{align*}
\text{begin } i \leftarrow \text{class.row}[k]; & \quad j \leftarrow \text{class.col}[k]; \\
\text{while } j \leq \text{nn} \text{ do} & \\
\text{begin} & \quad \text{(Decide the color of pixel } [i, j] \text{ and the resulting err 17);} \\
& \quad \text{for } l \leftarrow \text{start}[k] \text{ to start}[k + 1] - 1 \text{ do} \\
& \quad \begin{align*}
& \quad u \leftarrow i + \text{del.i}[l]; \quad v \leftarrow j + \text{del.j}[l]; \\
& \quad \text{buffer}[u, v] \leftarrow \text{buffer}[u, v] + \text{err} \ast \alpha[l]; \\
& \quad \text{end;} \\
& \quad j \leftarrow j + 8; \\
& \quad \text{end;}
\end{align*}
\end{align*}
\]

This code is used in section 26.

16. (Global variables 6) \[ \equiv \]
\[
\text{class.row: array [0 .. 63] of -2 .. 8: \{ buffer row containing pixels of a given class\}} \\
\text{class.col: array [0 .. 63] of 1 .. 8: \{ first column containing pixels of a given class\}} \\
\text{class.number: array [-2 .. 9, 0 .. 9] of 0 .. 63: \{ number of a given position\}} \\
\text{err: real: \{ error introduced at the current position\}} \\
\text{err.black: real: \{ error introduced at the current position if black chosen\}} \\
\text{black.diff: real: \{ difference between err and err.black for gray pixel\}} \\
\text{l: 0 .. 256: \{ index into diffusion tables\}} \\
\text{start: array [0 .. 64] of 0 .. 256: \{ first entry of diffusion table for a given class\}} \\
\text{del.i, del.j: array [0 .. 256] of -1 .. 1: \{ neighboring location for diffusion\}} \\
\text{alpha: array [0 .. 256] of real: \{ constant of proportionality for diffusion\}}
\]
17. Here we choose white or black, whichever minimizes the magnitude of the error. Potentially gray values of this pixel and its neighbors make this calculation slightly tricky, as we must subtract \( \zeta \) when a gray pixel is created and add \( \zeta \) when it is destroyed.

\[
\text{(Decide the color of pixel } [i, j] \text{ and the resulting } err_{17}) \equiv
\]

\[
\text{if } \text{darkness}[i, j] = \text{gray} \text{ then}
\begin{align*}
\text{begin} & \quad \text{err} \leftarrow \text{buffer}[i, j] - \zeta; \quad \text{err}_{\text{black}} \leftarrow \text{err} - \text{black.diff}; \\
\text{end;}
\end{align*}
\]

\[
\text{else begin}
\begin{align*}
\text{err} & \leftarrow \text{buffer}[i, j]; \quad \text{err}_{\text{black}} \leftarrow \text{err} - 1.0;
\end{align*}
\]

\[
\text{end;}
\]

\[
\text{if } \text{darkness}[i - 1, j] = \text{white} \text{ then } \text{err}_{\text{black}} \leftarrow \text{err}_{\text{black}} - \zeta;
\]

\[
\text{if } \text{darkness}[i, j - 1] = \text{white} \text{ then } \text{err}_{\text{black}} \leftarrow \text{err}_{\text{black}} - \zeta;
\]

\[
\text{if } \text{darkness}[i, j + 1] = \text{white} \text{ then } \text{err}_{\text{black}} \leftarrow \text{err}_{\text{black}} - \zeta;
\]

\[
\text{if } \text{darkness}[i + 1, j] = \text{white} \text{ then } \text{err}_{\text{black}} \leftarrow \text{err}_{\text{black}} - \zeta;
\]

\[
\text{if } \text{err}_{\text{black}} + \text{err} > 0 \text{ then}
\begin{align*}
\text{begin} & \quad \text{err} \leftarrow \text{err}_{\text{black}}; \quad \text{darkness}[i, j] \leftarrow \text{black}; \\
\text{if } \text{darkness}[i - 1, j] = \text{white} \text{ then } \text{darkness}[i - 1, j] \leftarrow \text{gray}; \\
\text{if } \text{darkness}[i, j - 1] = \text{white} \text{ then } \text{darkness}[i, j - 1] \leftarrow \text{gray}; \\
\text{if } \text{darkness}[i, j + 1] = \text{white} \text{ then } \text{darkness}[i, j + 1] \leftarrow \text{gray}; \\
\text{if } \text{darkness}[i + 1, j] = \text{white} \text{ then } \text{darkness}[i + 1, j] \leftarrow \text{gray};
\end{align*}
\]

This code is used in section 15.

18. \( \text{(Initialize the diffusion tables 18)} \equiv \)

\[
\text{black.diff} \leftarrow 1.0 - 2.0 \times \zeta;
\]

See also section 19.

This code is used in section 26.

19. **Computing the diffusion tables.** The tables for dot diffusion could be specified by a large number of boring assignment statements, but it is more fun to compute them by a method that shows some of the mysterious underlying structure.

\( \text{(Initialize the diffusion tables 18)} + \equiv \)

\( \text{(Initialize the class number matrix 21)}; \)

\( \text{(Compile “instructions” for the diffusion operations 23)} \)

20. The order of classes used here is the order in which pixels might be blackened in a font for halftones based on dots in a 45° grid. In fact, this is precisely the pattern used in the \(-\text{dot300}-\) font that was described earlier.

\( \text{(Basic procedures 10)} + \equiv \)

\[
\text{procedure store}(i, j : \text{integer}); \quad \{ \text{establish new } \text{class.row}, \text{class.col} \}
\begin{align*}
\text{begin} & \quad \text{if } i < 1 \text{ then } i \leftarrow i + 8 \text{ else if } i > 8 \text{ then } i \leftarrow i - 8; \\
\text{if } j < 1 \text{ then } j \leftarrow j + 8 \text{ else if } j > 8 \text{ then } j \leftarrow j - 8; \\
\text{class.number}[i, j] \leftarrow k; \quad \text{class.row}[k] \leftarrow i; \quad \text{class.col}[k] \leftarrow j; \quad \text{incr}(k); \\
\text{end;}
\end{align*}
\]

\[
\text{procedure store.eight}(i, j : \text{integer}); \quad \{ \text{rotate and shift for eight classes} \}
\begin{align*}
\text{begin} & \quad \text{store}(i, j); \quad \text{store}(i - 4, j + 4); \quad \text{store}(5 - j, i); \quad \text{store}(1 - j, i - 4); \\
\text{store}(4 + j, 1 - i); \quad \text{store}(j, 5 - i); \quad \text{store}(5 - i, 5 - j); \quad \text{store}(1 - i, 1 - j); \\
\text{end;}
\end{align*}
\]
21. (Initialize the class number matrix)\textsuperscript{21} \equiv \\
k \leftarrow 0; \text{store-eight}(7,2); \text{store-eight}(8,3); \text{store-eight}(8,2); \text{store-eight}(8,1); \\
\text{store-eight}(1,4); \text{store-eight}(1,3); \text{store-eight}(1,2); \text{store-eight}(2,3); \\
\text{for } i \leftarrow 1 \text{ to } 8 \text{ do} \\
\hspace{1em} \begin{align*}
\text{begin class-number}[i,0] & \leftarrow \text{class-number}[i,8]; \text{class-number}[i,9] \leftarrow \text{class-number}[i,1]; \\
\text{end}; \\
\text{for } j \leftarrow 0 \text{ to } 9 \text{ do} \\
\hspace{1em} \begin{align*}
\text{begin class-number}[-2,j] & \leftarrow \text{class-number}[6,j]; \text{class-number}[-1,j] \leftarrow \text{class-number}[7,j]; \\
\text{class-number}[0,j] & \leftarrow \text{class-number}[8,j]; \text{class-number}[9,j] \leftarrow \text{class-number}[1,j]; \\
\text{end}
\end{align*}
\end{align*}
This code is used in section 19.

22. The tricky part of this process is the fact that some values near the bottom of the buffer aren’t ready for processing until errors have been diffused from the next bufferload. In such cases we go up eight rows to process a value that has been held over.

\text{Global variables} 6) +\equiv \\
\text{hold: array [0..9,0..9] of boolean;} \\
\hspace{1em} \{ \text{is this value too close to the bottom of the buffer to allow immediate processing?} \}

23. The “compilation” in this step simulates going through the diffusion process the slow way, and records the actions it performs (so that they can all be done at high speed later).

\text{Compile “instructions” for the diffusion operations} 23 \equiv \\
\text{for } j \leftarrow 0 \text{ to } 9 \text{ do} \text{hold}[9,j] \leftarrow \text{true}; \\
\text{for } i \leftarrow 0 \text{ to } 8 \text{ do} \\
\hspace{1em} \begin{align*}
\text{for } j \leftarrow 0 \text{ to } 9 \text{ do} \text{hold}[i,j] \leftarrow \text{false}; \\
\text{l} \leftarrow 0; \text{k} \leftarrow 0; \\
\text{repeat } \text{i} \leftarrow \text{class-row}[k]; \text{j} \leftarrow \text{class-col}[k]; \text{w} \leftarrow 0; \text{start}[k] \leftarrow l; \\
\text{for } u \leftarrow i-1 \text{ to } i+1 \text{ do} \\
\hspace{1em} \begin{align*}
\text{for } v \leftarrow j-1 \text{ to } j+1 \text{ do} \\
\hspace{2em} \begin{align*}
\text{if class-number}[u,v] > k \ & \begin{align*}
\text{begin del-i[l]} & \leftarrow u-i; \text{del-j[l]} \leftarrow v-j; \text{incr}(l); \\
\text{if } u = i \ & \begin{align*}
\text{w} \leftarrow w+2 \{ \text{neighbors in the same row get weight 2} \}
\end{align*}
\end{align*}
\text{else if } v = j \ & \begin{align*}
\text{w} \leftarrow w+2 \{ \text{neighbors in the same column get weight 2} \}
\end{align*}
\text{else incr}(w); \{ \text{diagonal neighbors get weight 1} \}
\end{align*}
\text{end}
\text{else if } \text{hold}[u,v] \ & \text{then hold}[i,j] \leftarrow \text{true}; \\
\text{if } \text{hold}[i,j] \ & \begin{align*}
\text{class-row}[k] \leftarrow i-8; \\
\text{begin del-i[l]} & \leftarrow 0 \text{ then alpha}[l] \leftarrow 2.0/w \\
\text{else if del-j[l]} & \leftarrow 0 \text{ then alpha}[l] \leftarrow 2.0/w \\
\text{else alpha}[l] \leftarrow 1.0/w; \\
\text{end}
\end{align*}
\text{end}
\end{align*}
\text{until } k = 64; \\
\text{start}[64] \leftarrow l
\end{align*}
\end{align*}
This code is used in section 19.

24. (Compute the alpha values for class \( k \), given the total weight \( w \) \textsuperscript{24}) \equiv \\
\text{for } l \leftarrow \text{start}[k] \text{ to } l-1 \text{ do} \\
\hspace{1em} \begin{align*}
\text{begin if del-i[l]} & = 0 \text{ then alpha}[l] \leftarrow 2.0/w \\
\text{else if del-j[l]} & = 0 \text{ then alpha}[l] \leftarrow 2.0/w \\
\text{else alpha}[l] \leftarrow 1.0/w; \\
\text{end}
\end{align*}
This code is used in section 23.

25. (Global variables) 6) +\equiv \\
\text{ll: 0..256}; \{ \text{loop index} \} \\
\text{u,v: integer}; \{ \text{neighbors of } i \text{ and } j \} \\
\text{w: integer}; \{ \text{the weighted number of high-class neighbors} \} \\
\text{i,j: integer}; \{ \text{the current pixel position being considered} \} \\
\text{k: 0..64}; \{ \text{the current class being considered} \}
26. The main program. Finally we’re ready to get it all together.

(The main program 26) ≜
   (Initialize the diffusion tables 18);
   (Open the input file 7);
   (Initialize the buffers 12);
   repeat (Choose pixel values and diffuse the errors in the buffer 15);
      if \( ii > 0 \) then (Output the pixel values for the top eight rows of the buffer 13);
      \( \text{roll} \);
   until \( 8 \cdot ii > \text{max} \)

This code is used in section 2.

Acknowledgements. The research described in this paper was supported in part by the System Development Foundation and in part by National Science Foundation grants IST-8201926, MCS-8300984, and DCR-8308109.
Addendum

Stop the presses! When I wrote the preceding pages (and had them typeset), I was unaware of a "well known" method that should have been included for comparison. So far this paper has considered (1) a halftone font with 65 levels of gray, in which each $8 \times 8$ character essentially contributes two dots to a picture; and (2) a halftone font with 33 levels of gray, in which each $4 \times 8$ character contributes one dot to a picture. It's also possible to construct (3) a halftone font with 17 levels of gray, in which each $4 \times 4$ character essentially contributes half of a dot (actually two quarter-dots) to a picture. This third method is based on an idea due to Robert L. Gard [Computer Graphics and Image Processing 5 (1976), 151–171].

The $k$th level of gray in the half-dot scheme is obtained by blackening cells $0$ to $k-1$ in the array

\begin{align*}
1 & 5 & 10 & 14 \\
3 & 7 & 8 & 12 \\
13 & 9 & 6 & 2 \\
15 & 11 & 4 & 0 \\
\end{align*}

or

\begin{align*}
14 & 10 & 5 & 1 \\
12 & 8 & 7 & 3 \\
2 & 6 & 9 & 15 \\
0 & 4 & 11 & 15 \\
\end{align*}

(We actually make two sets of characters, one the mirror image of the other, and alternate between them as a picture is typeset.) The following METAFONT file will generate such a font \texttt{hd300}, in essentially the same way that the other fonts \texttt{dot300} and \texttt{hf300} were generated earlier:

\begin{verbatim}
% halftone font with 17 levels of gray, characters "A" (white) to "Q" (black)
% includes also the mirror-reflected characters "a" (white) to "q" (black)
pair p[]; % the pixels in order (first p0 becomes black, then p1, etc.)
p0=(3,0); p4=(2,0); p8=(2,2); p12=(3,2);
transform r; r=identity rotatedaround ((1.5,1.5),180);
for i=0 step 4 until 12: p[i+1]=p[i] transformed r;
p[i+2]=p[i] shifted (0,1); p[i+3]=p[i+2] transformed r; endfor
w#:=4/pt; % that's 4 pixels
font_quad:=w#; designsize:=8w#;

r:=identity reflectedabout ((2,0),(2,3));
picture prevchar; prevchar=nullpicture; % the pixels blackened so far
for i=0 upto 16:
    beginchar(i+ASCII"A",w#,w#,0); currentpicture:=prevchar;
    if i>0: addto currentpicture also unitpixel shifted p[i-1]; fi
    prevchar:=currentpicture; endchar;
    beginchar(i+ASCII"a",w#,w#,0);
    currentpicture:=prevchar transformed r; endchar;
endfor
\end{verbatim}

Here are four pictures for comparison, showing also the result of the elaborate "dot diffusion" method discussed at the end of my paper:

![double dot](image1.png) ![single dot](image2.png) ![half dot](image3.png) ![dot diffusion](image4.png)
(Each of these was printed on a Canon laser printer with 300 pixels per inch.) Gard's half-dot method clearly improves the quality of single-dot pictures; it also has the advantage that its characters are square instead of diamond-shaped, hence the data is easier to compute. On the other hand, it does require twice as much data. Indeed, the double-dot picture shown here was typeset from 64 rows of 55 characters each; the single-dot picture was typeset from 128 rows of 55 characters each; and the half-dot picture was typeset from 128 rows of 110 characters each.

The upper left corner of the half-dot data for Mona Lisa looks like this:

\beginhalftone
\makeatletter
\let\make@halftone\halftone
\makeatother

Uppercase and lowercase letters alternate in a checkerboard fashion, so that the reflected characters will appear in the correct positions. The \beginhalftone macro is the same for half dots as for double dots: only the font name and the data encoding scheme are different.