Fonts for Digital Halftones

Donald E. Knuth
Stanford University

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 8×8 squares of pixels. The idea was to have 65 characters for 65 different levels of brightness: For $0 \le k \le 64$ 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 8×8 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 METAFONT program painted pixels black by ordering the positions as follows:

45	29	34	18	46	30	33	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	32	16	44	28	35	19
15	63	0	48	12	60	3	51
37	21	42	26	38	22	41	25
5	53	10	58	6	54	9	57

[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:

```
% 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"0",w#,w#,0); currentpicture:=prevchar;
 if i>0: addto currentpicture also unitpixel shifted p[i-1]; fi
 prevchar:=currentpicture; endchar;
 endfor
```

This file was called dt.mf; I used it to make a font called 'dt300' by applying METAFONT in the usual way to the following file dt300.mf:

```
% 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.
```

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

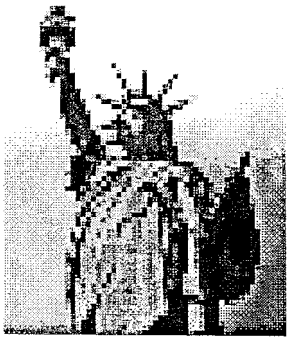
It's fairly easy to typeset pictures with dt300 if you input the following macro file hf65.tex in a TEX document:

```
\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#1\endhalftone{/let.=/endhalftoneline
  /beginhalftoneline#1/endhalftone}}
\def\beginhalftoneline{\hbox\bgroup\ignorespaces}
\def\endhalftoneline{\egroup\beginhalftoneline}
\def\endhalftone{\egroup\setbox0=\lastbox\unskip\egroup}
% Example of use:
% \beginhalftone
% chars for top line of picture.
% chars for second line of picture.
% ...
% chars for bottom line of picture.
% \endhalftone
```

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

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

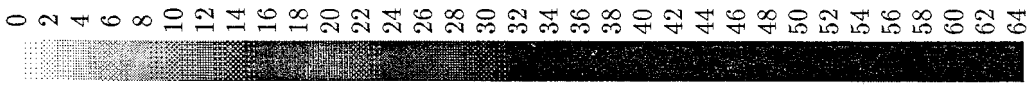




The squareness of the pixels is much too prominent.

* 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 `dt300`:*



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 `dt300` overemphasizes light tones.

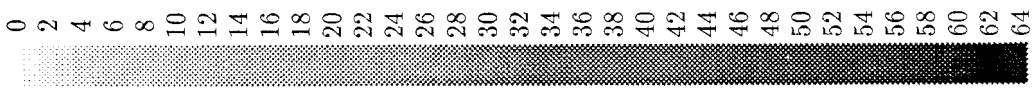
My next attempt was to look at halftone pictures in books and newspapers, in order to discover the secret of their success. Aha! These were done by making bigger and bigger black dots; in other words, the order of pixels p_0, p_1, ... was designed to keep black pixels *close together* instead of far 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:

				63			
				39	47	55	
				28	15	23	31	35	
			52	20	4	7	11	43	59
		62	60	44	12	1	3	19	51
	38	46	54	36	25	17	9	27	
	29	14	22	30	34	49	41	33	
53	21	5	6	10	42	58	57		
61	45	13	0	2	18	50			
	37	24	16	8	26				
			48	40	32				
				56					

Here I decided not to stick to an 8×8 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 8×8 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 *black* pixels in character k to be essentially the same as the pattern of *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.

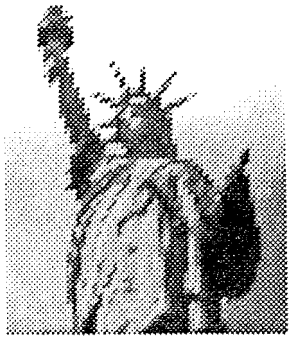
John and I used this new ordering of pixel positions to make a font called `dot300`, analogous to `dt300`. It has the following gray levels:*



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

			31					
		19	23	27				
		14	7	11	15	17		
	26	10	2	3	5	21	29	
30	22	6	0	1	9	25		
	18	12	8	4	13			
			24	20	16			
				28				

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.

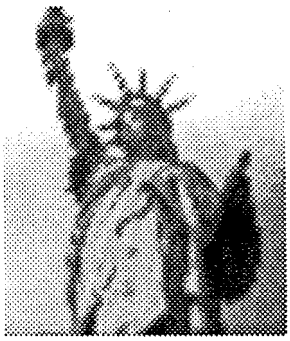
Here is the METAFONT file `hf.mf` that was used to generate the single-dot font:

```
% 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
```

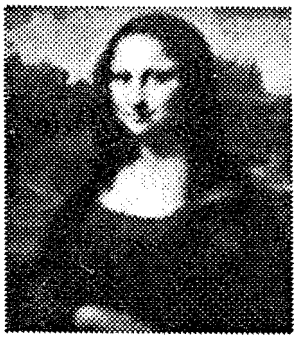
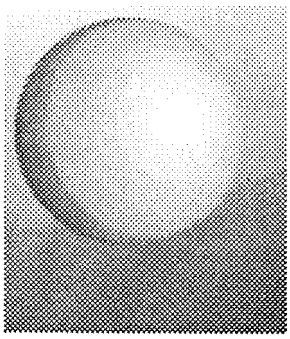
(There's also a file `hf300.mf`, analogous to the file `dt300.mf` above.)

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:

```
\font\halftone=hf300  % for halftones on the Imagen 300, each dot independent
\chardef\other=12

\newif\ifshifted
\def\shift{\moveright.5em}
\def\beginhalftone{\vbox\bgroup\offinterlineskip\halftone
 \catcode`\.=\active\shiftedtrue\shift\hbox\bgroup}
{\catcode`\.=\active \gdef.{\egroup
 \ifshifted\shiftedfalse\else\shiftedtrue\shift\fi\hbox\bgroup\ignorespaces}}
\def\endhalftone{\egroup\setbox0=\lastbox\egroup}

% Example of use:
% \beginhalftone
% chars for top halfline of picture. (shifted right 4 pixels)
% chars for second halfline of picture. (not shifted right)
% chars for third halfline of picture. (shifted right 4 pixels)
% ...
% chars 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 `hf723` for the high-resolution APS, in which case the pictures come out looking like this:

The same TEX macros were used, but font `\halftone` was defined to be `hf723` instead of `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 *International Herald Tribune* use a 72-line screen, rotated 45°; this is approximately the resolution $50\sqrt{2}$ that we would obtain with the `hf400` font on a laser printer with 400 dots per inch. (The 300-per-inch font `hf300` gives a rotated screen with only $37.5\sqrt{2} \approx 53$ dots per inch.) The photographs on the book jacket of *Computers & Typesetting* have a 133-line screen, again rotated 45°; this is almost identical to the resolution of `hf723`. But this is not the upper limit: A book that reproduces photographs with exceptionally high quality, such as *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°.

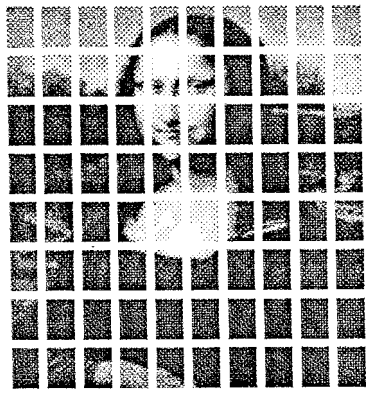
Let's turn now to another problem: Suppose we have an image for which we want to obtain the best possible representation on a laser printer of medium resolution, because we will be using that image many times—for example, in a letterhead. In such cases it is clearly desirable to create a special font 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.

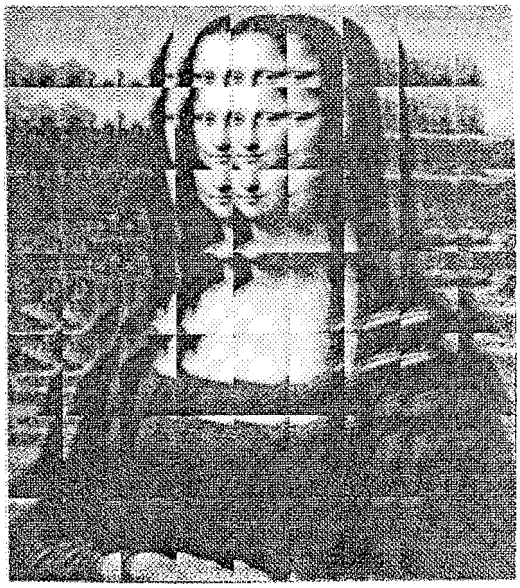
The examples above were produced on an Imagen printer as 64 lines of 55 columns per line, with 8 pixels in each line and each column. To get an equivalent picture with every pixel selected individually, we can make a font that has, say, 80 characters, each 64 pixels tall and 44 pixels wide. By typesetting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

TEX will typeset such an image if we say `\monalisa` after making the following definitions:

```
\font\mona=mona300[hf,dek]
\newcount\m \newcount\n
\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}}
```

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,
              191,198,207,215,223,230,238,246,254,263,271,279,287,295,302,
              311,318,328,334,342,350,358,366,367,375,382,383,390,392,398,
              400,405,408,414,416,421,424,430,432,439);
row(2); cols(4,7,12,20,23,28,30,37,38,40,45,48,53,61,64,
```

... 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.
```

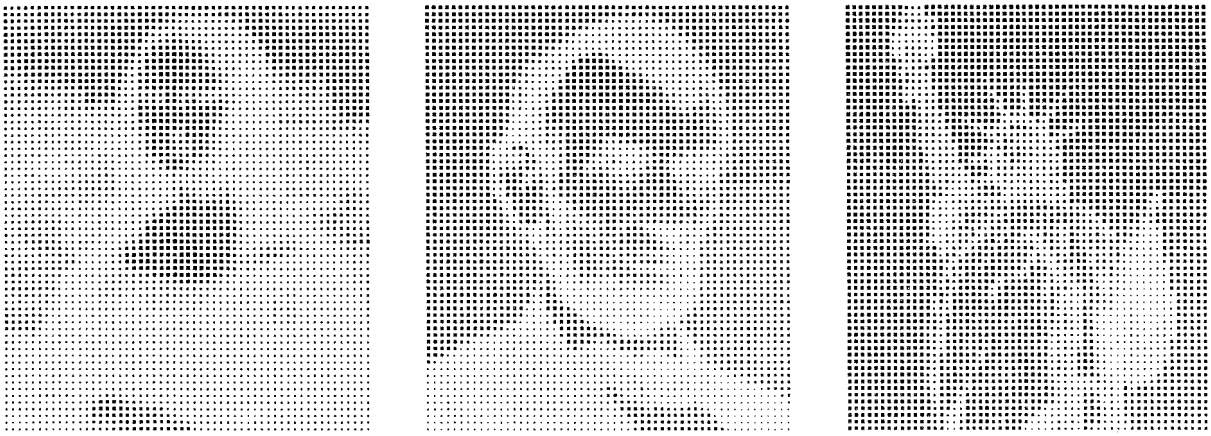
and the driver file picfont.mf was

```
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:

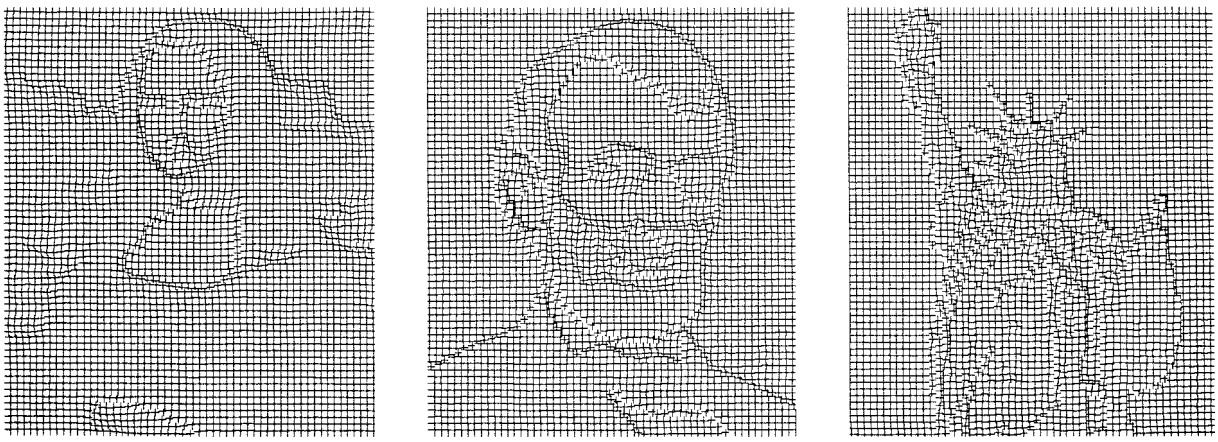


The METAFONT file `negdot.mf` that generated this font is quite simple:

```
% 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.



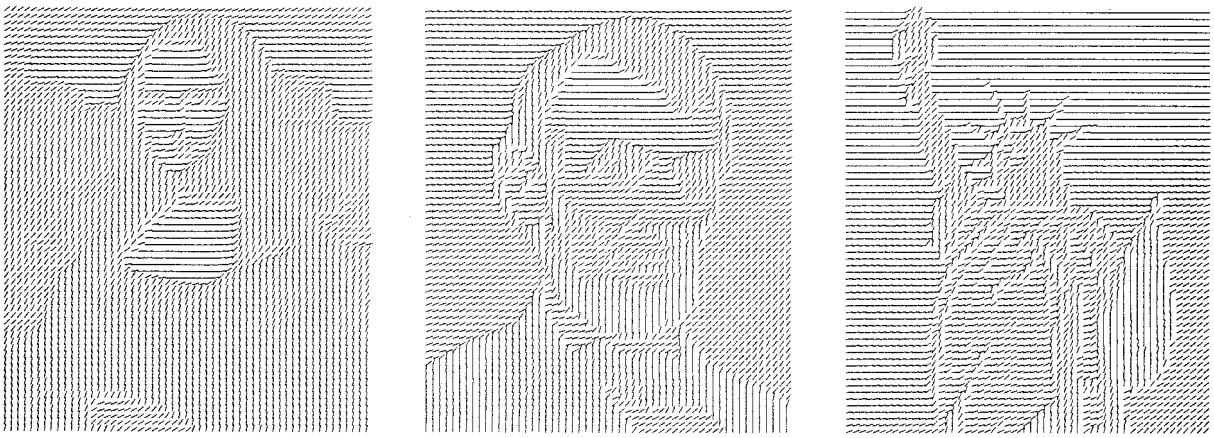
The METAFONT file `lines.mf` for this device-independent font is:

```
% pseudo-halftone font with 65 lines that move right and up
mode_setup; q:=savepen;
w#:=2.5pt#; 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#:=2.5pt#; 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:



(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 `hf33.tex` instead of `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#:=7.5pt#; 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

```
(Lisa)
FFHHIJKKJJJKLKKLLLKLLLLLMLLLLMMMLLLLLLMLKJJJJJJIIIHHGHH.
IIIIJKKKKJJJKLMLMMMLMNNOMLJHGFGIKMMMMMMLKKJJKJJIJIJJHJJ.
JJIIJKLLLKJKMMMMLNNNOOOOHCA988999DJNONONMLKKLLKKKKKKJLK.
IJJJLMMMMLLLMMMNNOOOPPLC9AA88888788EOPQONMMMNNMMMMMMLMM.
KJKLMNNNNMMMMMMNOOPQQKA89CA988877777BORRQONNNONPOOONMNN.
MLLMMMNNNNNNONNNOPQQOA9DHLIFC98777776CQRQPOOOOOPPQQOOMN.
MLLMNMNOOOOOOOOOPQRQFALSTUTRNG97666666GSSSRQQQQQRRPPPNN.
LKMMOOPPPOPPOOPPQRRL9JTVVVVUTNE976666680SSSRSRSSRQRQQOO.
LLMOPPQQQQQPPQPORSQBAPUVVVVVTQH97665666FSRRRSSSSRRSSRPP.
LKKOPQRSSRQPQRPPRSK8BRUVVVVUTPJC86655558NMPQOQQRRRSSSRR.
IFGNPRQORRRQQRQQSOB6GSUVVVVUTROJE9655556DGLKKOMORSTSSSR.
DEEHLMKJKQRRSSSTSJ77LTUVVVVUTTRPI9555555ACFEIKKIQTTTSSS.
BDEFGGHGGGJOSSTTQD66JSQRUUQJKLLHC855555589CGGFIEMSTTTTS.
BCCFDCDBCDHFQSSSM967BEGFQSECHD989965555579ABEFEBHQTTTTS.
CBCCBCC9BCGEPPKNL866HJIFQRCLODDIMB655555789ABBAAEPTTTSR.
BABA9AA8ABDBIEBCH857QSQPTRHQROOROB554455689ABA9ACMTTSRQ.
99988978989789A8A856PUVUURNTUVUQI85555556999A9889FQQQPP.
87778988777777777666LUVUURNTUUTLC65555556CCEDDDDDEHKKJI.
766779988777666665666GTUSUOJRUTOF955444455667889 9BDEFEBA.
77666678777766655555ANSSLA9OTQJC8555445556666776778889A9.
67666667667765555555 8JOSQDEJKMHB855444445677666666667899.
776667665666655555556HOMOHCEMOHA7554444456777777777787987.
766666555566656655559LSPHFHKIC865544444456877777777667.
666655555677777865555569NUTOJFB765554444456878986677677 7.
656655555567777965555559OQKD876555544444568889877777777.
65566555556778AC845445557A876665555444455587787666676 77.
56566777789ACBBB9444455557866666676544445577788666666 67.
5556788899BBBBBA9645444559GA8778A964444445778988777788 8.
6678899999ABBB9986444455558NJEBBBED8444444467999ABA9CCDA.
7789AB977779989765444455 6AMOKHHIIHB544454445555876DHDB9.
7798998779566665544444458FOPQOOOPNMF644554445555666BHC9A.
767898788666657654444 6DKQTTSRRSSRPJ85554444567AEFGGFA99.
668999ABA766657654445FPTUUUUTTUTSPI655544444457FJHCDCBA.
7799A9BC876675764444 8OTUUUUUUUUTSOJ754544444456ADFGFFEF.
979CDCDB977777754456FSUUVVUUUUUTSOG6545445654457ABEHCHF.
878DGGEDB8798644459CLUUVVVUUUUUTROF655556886544579DHJKN.
76889DHFC988644447HMTUVVVVVVVVUUTSOE7569CEED964444569BBC.
978889BECDB744445BPTUUVUUUVVVUUURNC879CHGB7544444456777A.
879A99BBBEC444446BQUVVUUUUVVVVUUSNF9ADA6544444444457778.
99BCCDEFFB54444578FQUUTUUVVVVVUURIC864444444444444446888.
BDCBBBBA84444446899AEJLSUVVVVUTPEA744444444444444445 99A.
BCB9BAA8544444579878 98ABEHKMNNIB875444445444444444445689.
ACDCDDEA444444677767 79A9AA999988564334455444444444445578.
8ABBBBB8444444666655689899876775443444555666555444444589.
889889964333446555555788766666544444444467765555444457B.
99999974443334554555566666556554444444456776554444544 57.
CDCCDC643333345445445555555554444444434567665444445545 7.
EFFEFC5333333444444444444555444444334345676543344455457.
CCCDC8333333334444444444444444444334334555544443344455556.
778864333333334444334444444444433334444433334444455545.
665533333333333333333444444444444333344443333334444444 5.
664333333433333333333344444444443333334433333333444444444.
5433434444433333333333444444444433333333333333333444444444.
433444444433333333333334444444433333333333333333344444444.
333344433333333333333443344443333333333333333444444444444.
333344333333333333333333333443333333333333333444444444444.
333455544443333333333333333333333333333333333344444444444.
333455444443333333333333333333333333333333333334444.
333445764655545997533333333333333333333333333334444.
3334555454566BLNLJFB744333333333333333333333333334444.
333354453444 8IMNMLJIFED9433333333333222333333333333334.
3223334333348HJKJKKJHHHFA643333333322233333333333333334.
3222333333346DEFFHIIIIIHGFB74333333223333333333333334.
32222322222347 88ADFHIIIJHHIG7433443334544433333333334.
```

```
(Lincoln)
RRRQRRQRQQQQRQQPRQPQQPPQPQQQ
RRRRRQQRRSRQRRQQQRQQQQQQQQRR
RSRRSRRRRSRQPRQQQQQQQQQRPNHC
QRRRSSSSSSSQQQRQRRRRQQOMD555
QQRSSSSSSTRRSRSRSSTSQMD55555
RSSSSSSSTTSUTTTSSSSSKE555555
RRRSSRSSTSTTSUSTSPKD76454555
RRRSSSSSSRSSSTRTNIA66549EKK9
SSRRSSSSQRRRRSSNDA7654DVVVVT
SSRRRRRRRRQRRRQ9846977VVVVVV
TSRRSRRRQRRSRSI953454RVVVVVV
SSSRRRRSRSRSRQ9A5443GVVVVVVV
SSSSSRRRRRRQQGE52435SVVVVVVV
TSSSRQRRQQQRPAE4285KUVVVVVVV
SRRRQRQRQRQRN7A2259PSUUUUVVV
SQRQQQQQQQRRN49238GOQTTTTUUU
RQQQPQQQQQRPK4213ANPOUUTTTTT
RQQQQQPPQQPQK22238NPPTTTTTSS
RPPQRQPPQPPQ911357IJKSSTSSST
QQQQQRPPPQPL201246HFHQRSSRRR
RRQPPQPQPPM6122247CBFOQRQQLB
RQQPPPPQPQ424KO74357EOQPRNDA
RQPPPPPPPL3753906335AOQQODD6
RPPOOOOOPN33659JK333ANRQI74B
QQPPPOPOON5B9FLME633FOPL89EB
RQQPPOOOOM7ILLL69C58MQNKFJMI
QQPOOONNMLADPPH5NN79OONONNNJ
PPOOONNMMN53QQ9PQO49NNNOPONM
PPOONMNMMO72KOLPOL73GMMNOONN
OONMNMMMLNN62MOOLJ824FKMNNNN
ONNMMMMLMLLB2NOMMGB125GKLMNN
NNNMMMMMLMLJ78ONMHG2287CGIJM
NNMMMMNMLLLKC7HNL2K31AB6ADFK
NMMMLLMLKKLKFF8621G659D9A9CG
NMLMLKKKKKKKIGB612BB9CFEF9GH
NMMLLKJKKJJJJJHD24B79AJGHBHA
MMLKKKKJJIJKLJIE93A67CJIIFIJ
MLKKKKJKKJJKJKJIH56A78GJKFIC
LLKKKJKJKJJKJJJII94485GIKEFB
MLKLKKKJJKJKJJKJIF71369JKHCH
MLKLJKLKJJJKIIIIIGD1244HEDBB
MLLLLKKJKKJJJIJII7H622369332
MLKLKKLKJJJKKJJJFCLB12223223
LLLKKKLKJKKJKJJK57NF42221111
LLLKKJKJJKKJJJJH23NMB2111111
LLLKKJKKJKJKKKK222KNMD211111
LLLLLKKJJJJKKKC2223NMMI41011
LLMLKKKKKJKLJF22223HMMML8110
LLLLLLLKLKJC32222232NMMMMD62
LLLKLLLJFA8133222223DNMMMLLK
LLMMMJA22222333333232MNMH85F
MMNLD1112122333333436J73322
MMG3111111223334343333234432
G5111111122234445434433335543
1111112112224434543443333343
1211111112223444554443343333
1112111222224445544433434344
1122111122222345544443334 36J
1222212122122445444443334 33D
1223222111122444434433333434
1223222111112444433333333333
2023232110112222222323222332
1011112100011111121222122222
0001011000001001111211112112
```

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.

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:;<=>?@ABCDEFGHIJKLMNOPQRSTUV

(Lisa)

```
MMIHFDBADDDA?BA???B?????=????===??????=?BDDDDDDFFFIHKIH.
FFFFDBABADDDB?=?===?=;;8=?DIKMKFB======?BADDADDFDFDDHDD.
DDFFDB???BDA====?;:;8888ITY[]^[[[RD;8;8;=?BA??BABBABD?B.
FDDD?====???===;:89877?T[YY]^]]^`]]P8768:===;;======?==.
BDA?=;:;:======:98766AY^[TY[]^]````_W85468;:;8:7888;=:;.
=??===;:;;;:;8;:;87668Y[RH?FMT[^_`_``bT656788889776688=;.
=?@=:=;8888888887646MY?33125;K[`bbbbbbK33456666645777:;.
?B==8877787798776555?[D2000012;O[`bbbbb]9433535335646688.
??=8776666677678536VY710000026I[`bbdbbbM355443435534477.
?BA8764435676477538]V510000137DT]bbeded^;=7686654534355.
FMK;756854556566639VbK3100001258DO[bdeddbRK?AB8=85324344.
RPOI?=ADA655334424C``?31000012257F[eddedeXTMPFABF6232434.
VROMKKHKKKC843226RbbD365116DB??IT^dededd^[TKKMFO=422224.
WTTMRTRVTRIM6443=[b`VPKM640TIR[][[bdeded`[YVPMPVI623233.
TVTTWTT[WTKO77A;?^bbIDFM64T?8RRF=Wbddded`][YVVYYO722245.
WYWY[YY]YVRVFOWTH]e_636725I658858Vedggddb^[YWY[YT=23356.
[[[]][`][^[`^[Y]Y^db710115:21016F]edededb[[[Y[]][M66677.
]```][^]```_`````bbb?10115:3112?TbdeddedbTTPQRRRRPHAADF.
`bb``[[^]`_`bbbbdbbbK21418D5129M[edgggggdebb`]^[[VRPMPVY.
``bbbb`]````bbbdededY;43@Y[826DT^dedgfdedbbbb``b`]]^[Y[.
b`bbbbb`bb`_bdeddede]D846QPDB=IV]edggggfeb``bbbbbbb`][[.
`_bbb`bbdbbbbededdddbI8=8HTO=8HY`ddgfgggdb`````````][^`.
`bbbbbdedebbbdbbedeed[?37IMIAFT^bdeggfgfeb]`_`_`_```bb`.
bbbbededdb`````]bdddeb[;129DMW`bdedgfgggdb]`][^bb`_b``_.
bdbbdedede`_```[bededde[96AR]`bdeddgggfgdb^]^[]`````_``.
bedbbdddedb``]YT]gdggddd`Y]`bbbddedgfggded]``]`bbbb`b``.
dbebb````][YTVWV[ggggdedd`]bbbbbb`bdggggde```]^bbbbbbb`.
eddb`]^][[WVWVWY[bgdgggde[KY]```^Y[bggfgfgd`_][]]````]]].
bb`]^[[[[[YVVW[[]bgfgfded];DPVVVPR]gggggggb`[[[YVY[TTRY.
``][YV[`````[[]`bdgggddbY=8AIIFFHWdgfgdfggdede]`bRHRV[.
``[^[[^`_[dbbbbdeggfggd^M8768887;=MbggddggfeddebbbVIT[Y.
`b`][]`]^]bbbe`bdgfggbRA6224553447D]ddeggfgdb`YOMKKMY[[.
bb][[[YVY`bbbd`bdgggdM721111221247Fbeddgggggggd`MDITRTWY.
``[[Y[WT]`bb`d`bggfg]8212111112248D`dgdgfgfggdbYRMKMMOM.
[`[TRTRV[`````dggdbM4110012111239Kbdgdggdbdggd`YVOITIM.
]`]RKKORW]`[]bggfd[T?1100011111358Mbdeddb^]bdggd`[RHDA;.
`b]^[RIMT[]^bgfgg`I=210000000112480`db[TPOR[bggfgdb[WVT.
[`]]][VPTRV`gfggdV721201110001215:T^`[TIKV`egfgggdb```Y.
]`[Y[[WVVPTggggfbW610011100000113;M[YRYbdggfggfgggd``_^.
[[WTTROMMWdgfgfe`]M611311000000115FT]bgfgggfggggfgggb]^].
WRTVWVWY]gggggggb][[YPD?4100001370Y`ggggggfgggfgfggfgd[[Y.
VTW[VYY]efgfgfd`[^`][]YVPIB=;;FV^`dgfgfgdgfgggggfggdb^[.
YTRTRROYgggggggb``_b``[Y[YX[[[[]^dbgiiggdeggfgfgfggggd`].
]YVWVWV^fgfgfgbbbbdeb][][[^`b``dggfigfgdddbbdedggfgfe][.
^][]^[[bgiiiggbdededd`^]`bbbbbdggfgggggggb``beddefgggd`V.
[[[[[[`gfgiiifedgddedbbbbdebdegfgggfgfgdb``bdeggggdggd`.
TRTTRTbgiiiiigdggdggdeddeddefgggfgggigdb`bbdgfgfgddgd`.
PMMPMTdiiiiiiggfgggfgggfgdedggfgggiifigdb`bdgiigggdegd`.
TTTRT^iiiiiiiiggfgggggfgggggfgggfgifiigdeddgggiigfgdddeb.
``]^bgiiiiiiiigfggiifggfgggfgfggfiiiigggggiiiigfgggdefd.
bbddiiiiiiiiiiiiiiiiiiiggfgggggfggiiiigfgfiiiiiiggfggggd.
bbgiiiiiigiiiiiiiiiiiifggfgfgfgiiiiiiggiiiiiiigfgggfggg.
dgiigigggfgiiiiiiiiiiiggggggggiiiiiiiiiiiiiiiiggfggfgf.
giiggfgfggiiiiiiiiiiiiiifgfgfgfiiiiiiiiiiiiiiiiigfggfggg.
iiiiggiiiiiiiiiiiiiiiiggiiggggiiiiiiiiiiiiiiiggfgggggfg.
iiiigfiiiiiiiiiiiiiiiiiiiiiifgiiiiiiiiiiiiiiiiiggggfgfggg.
iiigdedgggggiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiggggfgf.
iiigddggfgfiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiggggg.
iiiggd`bgbdedgd[[`diiiiiiiiiiiiiiiiiiiiiiiiiiiiiiifgfg.
iiigddefdgdbbW?;?DMW`ggiiiiiiiiiiiiiiiiiiiiiiiiiiiiggggg.
iiiidggdiggg]F=;=?DFMOR[giiiiiiiiiiiilkiiiiiiiiiiiiiiiif.
ikliiigiiiig]IDADBADIHIMYbgiiiiiiiiikkliiiiiiiiiiiiiiiig.
ikkkiiiiiiigbROMMIFFFFFIKMV`giiiiiilkiiiiiiiiiiiiiiiig.
ilklkiklkkkig`]^YRMIFFFDHIFK`giiggiiigdgggiiiiiiiiiiig.
```

(Lincoln)

```
554655646666566756766776766 6
455455655355546655665666664 5
535535545356756666656664 7:IT
654534343436655554556 58=Rddd
665343434254453534236=Rdeded
434343432241222434344APdedded
554435342422414247AR`bgdgded
5453434344434252:FYbbdg[PAB[
345534346555443;RY`bdgR00002
435455454555546[]gb[``000000
244535546555353G[eigdg5000000
34355453535356[YdgfiK0000000
4343445554566JPdkgid30000000
2434465466647YOgk^dB10000000
355465656465:`Ykke[741111000
464666566645:g[li]K862323111
466676665657BfkniY;781122232
566656776676Alkki];772323243
577656776776[nnie`FDA4324342
65665577767?kpmkgbHMI6534455
5567767677<bnklkg`TWM86566?V
4667777576fkgA8`gid`O8674;RY
567777777?i`di[8biidY8668RQb
577888887;iibe[DBiiiY:56F`gV
657778788;dW[M?=0biiM87?][OW
566778898=`F???b[Te]=6:BMD=F
667888;:=?YR77Id;;`[88;8;:;D
77888;;==;di66[768g[;;:878;=
7788;=:==8`kA8?78¿iK==;88;:
88;=:===?;:bl=88?D]kgMA=;;:;
8;:====?=??Wk;8==KWnkdKB?=;:
;:;=====?=??D`]8;=HKkk^`TKFD=
;:====:=???BT`I:?lAinYVbYRMA
;===??=?BB?BMM]blmKbd[R[Y[TK
:=?=?BBABABAFKWbnkWV[TMPM[KH
;==??BDABDDDDDHRkgV`[YDKHWHY
==?BABADDFDA?DFP[iYb`TDFFMFD
=?BABADBADDBDBDFHebY`]KDBMFT
??BABDADBDDADDDFF[gf^dKFAPMW
=?B?BABDDADBDDADFM`nib[DBHTH
=?B?DB?ADDDAFFFFFKRmkggHPRWV
=????ABDABDDDFDFG`Hblkib[iik
=?B?BB?BDDDABDDDMT?Wmlkkikli
???BAB?ADABDADDAe`:Mgklknnmn
?@?BADBDDABDDDDHki;=Vlmnnnnn
???BADABDBDABABklkB:=Rknmnmn
?????BADDDDABATklki;==Fgnpnm
?@=?BBABADA?DMlkkliH===?]nmp
???????B?BDTikklkkik;====Rbl
??@B???DMYiniilklkliR:===??A
??===DYkkkkkiiiiiiikk=;=H^dM
==;?Rnnnknlkiiiiiiigib D`iikl
==Kimnmnnnkliiigigiiiikiggik
Kdnnnnnnmnkkkigggdgiggiiiddgi
nnmnmnknnklkggifefigfiiiiigi
nknnmnnnmnkklifggdegfgiigiiii
nmnknmnkklkkkgggddgggiigiggg
nnkknnnnkklklifdegfgfiiigibD
nkklknknklmlkggdgggggiiigiiR
nklikkknnnnkkgggfifgiiiiigig
nkkiklknmnmnkgfggiiiiiiiiiii
kpliknknnpnnkklkkkliklkiik
npnnnnknpppmnnnnnnknklknkklkl
pppmpnnpppppnppmnnmlmnnnknnk
```

WXYZ[\]^_‘abcdefghijklmnop. The density of character k was assumed to be approximately $k/36$ for $0 \le k \le 8$, and approximately $k/72 + 1/9$ for $8 \le k \le 64$.

	(Liberty)
665456775666666666777778788==.	000000000DYV000000000000000000000000000000000000000000.
6AD8;=76466566567677778988;.	0000000007V‘b8000000000000000000000000000000000000000.
[bb[][OB7666665665776788;;:.	00000000KIORYP000000000000000000000000000000000000000.
bbb][^‘_Y<7466456777788;=:=.	0000000B?[‘YR2000000000000000000000000000000000000000.
bbb[[[[][^H946556776888;8==.	0000000TddbdbM000000000000000000000000000000000000000.
bbb‘^][^]‘]W6545666787887;=.	0000000ggd‘‘be000000000000000000000000000000000000000.
bbb‘b[[[^]^_K554678877898:=.	0000000idTW[‘f000000000000000000000000000000000000000.
ebb‘bb‘‘]^‘‘]F5677877788;8?.	0000000ggDFVde000000000000000000000000000000000000000.
Mbb‘b‘‘‘b_‘‘‘[F668878;8889=.	00000006HPRK]000000000000000000000000000000000000000.
0BY‘b‘_‘]‘‘_b‘[767788=====?.	000000000‘TI300000000000000000000000000000000000000.
002BT‘b‘bbb‘b‘_F77887;===:?.	00000000[?RRY000000000000000000000000000000000000000.
00016HR‘‘bb]‘]‘Y8787;8:;:=?.	00000000;=TTY00000000I000000000000000000000000000000.
0000015J‘‘‘]‘M]‘A8;:;:;?==?.	000000001[[YV200000000:0000=000000000000000000000000.
00000003YO‘‘‘OPbY88;;:=====.	00000000=]bYRT00000000[0000V00000F000000000000000000.
00000011IR[b]PA‘‘B8;==?????.	000000000e?]RR000000000Y000[0000K000000000000000000000.
101011322K][K?BVbF===??@??B.	00000000060[PO000000000]200[000D?000000000000000000011.
31111123347B:;?DbM=?A?D??BA.	000000000000VRPM000000004]=D[00;V000000000000000000011111.
22324434354688?I][;?BBDBAB?.	000000000000TOMW006H6000A‘^[‘]=]000016800000000001011131.
322333555333577?I[‘=ADAFBAB?.	000000000000RPRYF000K[?;‘V54@[‘Y004?T5000000000121112221.
62223354532568=H‘‘ADBDDDBDD.	000000000000RTRVV00000T]DVRTTRY‘60YF000000000001112232423.
[PI8556543353348Kb‘?DDADDDD?.	000000000000R[RYV=00000;Y7IOW[RX[020000000001111112434342.
bddeVA857378;CFP_bFBDBDADAB.	000000000000F[YVY^00001]?6YHRTVR^B00101011111112233444353.
degdbWA6?=K][W]‘b[FBA?AB?B?.	0000000000010YV[W]0027Ye[A;:MTWY[[0112322323232224343444.
Mbgb‘V=5F‘bebbdbbTDA=??BA?D.	0000011000110Y‘[VW39]WKdPY8DRWXVY]1324333432434435545554.
‘gfd^?86Fdd‘‘ded[WB??@??B?B.	001111111111?[YY[^7111dIMHMRVYWVW2435544343433454554665.
V[‘PF;?65[][‘ddeR?????????B.	1111213132323X^T[‘‘14DdTPVTY[[‘TT2445656545655456677766.
IOTO88866=VTY‘b‘??BBBBBBBAB.	1312422222233Y[D]bbDV[‘‘YYW[YY‘O‘?5676576777666667588877.
=DMB;88868YPVWY[DDDFDFDDDDF.	2232333444544800Wd‘[b;360][[VYV]W[Y6777787887777588989:8:.
??=;:;=:88H[Y]RYFFFKFFFDDFH.	23243434335346Kb‘b‘O=757g[YRYY]‘V‘657788887877Bi;888;;=.
==;:=?==;;DFDFK[KKKMKKKHHII.	2243435355545XddWT‘RD]DTdPOKRRV^V[877789898888di8;===?=.
=;:BD?F==:?F?DM‘MKMMMKKIHIH.	4355355545453‘bgdYM?R===FI=7=88?IYOYP?88;:8;8?gY;==?=?=.
==ARA;=?==BIBM][KMMPMKKKHKI.	3433545455556dbgdPA8;=RVPA7=:;IH=?BKMbB8;;;8FK[===?=?BA.
?FTDKKFMMKR]O^iPKMMOMMOMMKK.	435535356454FgggYY8?TTRFA;8B:TB8I?DFMPi?:;:=Kg[?==?BABD.
I]B=Riilkiikbi VKMOMPMMPPMMM.	5545555465657bgfR:?WD‘FB[8?=DBFAWYK?RMRi=;=RRdnd=???BAD.
MD??BHWiknibilMKMPMPOPOOPOP.	6466676645456iib‘FR?bDFFK8¿]M?M[]7OT[Fb];=Mbknn??BADDF.
IABABAD[dng]l_MKMOPOMPOPOPO.	565565666664VgfM^HDbP88W8;YO‘eHGY‘8DPKDRKDDdikik?BABFFF.
MTMKIHIHWgigkPMMPMMPROPOPRO.	46556664545666ibHIbI88R78?F?Kd‘HMb8=[MKYDT]^nkglA?BBDFF.
bMIKR[‘[kknkkTMMPOPORRRRRRO.	655665555455466fIFd??7?;7;A==e0iWM^A8?bVVM?inkgikFBDDFFI.
IMYRKKHM^ingkWPOPMMPORRQTRR.	546666455645 66=Fi=I88R37=B?Db?bkMFF7@gRTRMinnmniYDDFFIF.
DDK]iiiknmgiiOPOOMPORROR TTT.	656656646456466_MB86I648=?Bb=BKkgTR;?KOYF‘dpnnkikDDFFFF.
KDFFIHMWbdbklTMMPMOPORRRTTT.	55676666654656He=O77=66<DRVP8?P[iWD??DbWTbepppnnnDDFIHI.
gV^OMMPY]nknnVPOPOPOPORTTTT.	666766767656567=O87F68=]K?B8;=[KgXYF;Ain]bnpppppmBDFFFF.
id‘gb^dignnpmWOPOPORRRRTTTW.	55665665665557[D[78878gD7=??:=‘DVFed?FYkbdpppppppFDFFIF.
kgkkkinnmnmnnOPOPOPRRRRRTWV.	5546666546666A8M?6;3==]87?=7==]BVi]bMKTeblpppppppnmDFIFI.
nnknnnpknnppiOMPORRQRQTTTTT.	4565656664667V=‘88?5==K78==8;?T?bbinDK[[bpppppppnpDFFFH.
mnnnmnpppmpnVMMPOPRRRRTTTWV.	54646665656 56]M‘8966:;:89?;:;?K=^d‘i?TdbgppnpppmnFFFGH.
nmnmnnppppkTKMMPOOOPRRTTTTT.	5365664554568D[D;:3=====A?;:=?F?]T‘d;Rbpppppmpppp klFFIHK.
pnnnmpppdlMKMMPOPPRORTTTTTT.	556656545546=BY:8@5==???IM=;??KI‘P‘g:Mpppppgppppgk FFIKK.
npppppnbOYgKMMPOPQRRRTTTWWV.	4545566646666JB‘;:=6?=?;BD?88?9F[bRbd8Ikpppngpppngk DFIKK.
R‘fikdWHKKp]KMPMORORRRTTTTT.	445453546666RTd?;:=A=?;?=?89?8Bb‘]bg?Pgpppminpnmik FFHKH.
?=?BDBDFFIkpTMOPPORRRRYYTTW.	355645554665FWd=;:I;:I7A???;?;?g_^‘g?Kgppnnkipnii PIIKKI.
]FFIKQPHFFepnPMOPORRQV[VWVV.	445465456646=V^=:7D:;T7?@B6:?;?g][‘_;Agmnppkgdndi FFHIHK.
lkl[‘knnbMdppgMMPOPORWVYVYW.	345545645466D[O;76?8‘?8=AA7;=:?d^]‘B:Bgkpppipkkln FFIHKK.
klmnmnnmpnnppmWOMMMRRR[WVWV.	233554566666F[BA58?=T8=FD?7:;=DbT^]B:¿gppnnnnkk‘FIHKKK.
ikknnmpppppppnniYPOMOTVTWVW.	343545666666F]8I3F;TB8;?F=;7;?BbT]b?8¿dppknkknnFFIHKKK.
degiklnnnnmnnmkppn‘RPTTTTTR.	344566656566M];=5H;=:F?FNA;7:?=[‘bgFD?Ydnppkkpn YDFHIHKK.
iddbgkgkkiikknnnpppmbWTPRRR.	345466666766Q‘;:8BR:@?;??=68=?;TbdgkgbTbininkiVDFIHIKKK.
Vedb[gigiggiklmnmpppppgVRRR.	445665766777Hd;7==b;D=;??;87=D8Ibbbi[^gbbk?dKFDFFIHIHKM.
¿dgbgkikkklnmnnnnpppppgVR.	546477777787;‘:8B_I:?:;7;7:7?=8Dbbei?AYg‘kBDDDDFFIHIKKK.
KMR[‘iiklnnkgnpmnmpppppppp‘.	3555667877877V=8Rd7;;;=?:;;==;ATb‘bi??MkbkDDFFFFIHIKHKK.
fI?BH]FM[b[OM]kpnnpppppppppp.	6665766777777MBBd‘6=7:?;;=8?;Hilgb‘i?9Id‘iDDFFGHIKHIKKM.
iiFBDDBBDDFDDDKkpnpppppppppp.	6787878:88888IA‘iY6<7?=DAA?=HliifbdiB:IbgiMFFIHKKKKKMMM.
kigDDDFFFFFIHIHMnpppppppppp.	:;:;:;:;====?=ODdkR6?8?FP;??InknkbgbkB?HbdgbIHMMMMMOMMMP.
nknVHIHIKKKKKKKKVpppppppppp.	??????BDABABBHIdnR=D?DTA=BKnpnpibignK‘igbkkMMRPRRRRRRRR.

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
⟨ first line of pixel data, n characters long ⟩
...
⟨ mth line of pixel data, n 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:

```
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.

```
⟨ Constants in the outer block 3 ⟩ ≡
  max_m = 200;  { m should be less than this }
  max_n = 200;  { n should be less than this }
```

This code is used in section 2.

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

```
  define cleanup_and_terminate = 9998
  define finish ≡ goto cleanup_and_terminate  { do this when all the pictures have been output }
⟨ Labels in the outer block 4 ⟩ ≡
  cleanup_and_terminate;
```

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.

```
  define text_char ≡ char  { the data type of characters in text files }
  define first_text_char = 0  { ordinal number of the smallest element of text_char }
  define last_text_char = 127  { ordinal number of the largest element of text_char }
⟨ Types in the outer block 5 ⟩ ≡
  ASCII_code = 0 .. 127;  { seven-bit numbers }
```

This code is used in section 2.

6. ⟨ Global variables 6 ⟩ ≡
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [*ASCII_code*] **of** *text_char*; { specifies conversion of output characters }

See also sections 10, 13, 14, and 22.

This code is used in section 2.

7. ⟨ Set initial values 7 ⟩ ≡
xchr['40] ← ' '; *xchr*['41] ← '!'; *xchr*['42] ← '"'; *xchr*['43] ← '#'; *xchr*['44] ← '$'; *xchr*['45] ← '%';
xchr['46] ← '&'; *xchr*['47] ← '''';
xchr['50] ← '('; *xchr*['51] ← ')'; *xchr*['52] ← '*'; *xchr*['53] ← '+'; *xchr*['54] ← ','; *xchr*['55] ← '-';
xchr['56] ← '.'; *xchr*['57] ← '/';
xchr['60] ← '0'; *xchr*['61] ← '1'; *xchr*['62] ← '2'; *xchr*['63] ← '3'; *xchr*['64] ← '4'; *xchr*['65] ← '5';
xchr['66] ← '6'; *xchr*['67] ← '7';
xchr['70] ← '8'; *xchr*['71] ← '9'; *xchr*['72] ← ':'; *xchr*['73] ← ';'; *xchr*['74] ← '<'; *xchr*['75] ← '=';
xchr['76] ← '>'; *xchr*['77] ← '?';
xchr['100] ← '@'; *xchr*['101] ← 'A'; *xchr*['102] ← 'B'; *xchr*['103] ← 'C'; *xchr*['104] ← 'D';
xchr['105] ← 'E'; *xchr*['106] ← 'F'; *xchr*['107] ← 'G';
xchr['110] ← 'H'; *xchr*['111] ← 'I'; *xchr*['112] ← 'J'; *xchr*['113] ← 'K'; *xchr*['114] ← 'L';
xchr['115] ← 'M'; *xchr*['116] ← 'N'; *xchr*['117] ← 'O';
xchr['120] ← 'P'; *xchr*['121] ← 'Q'; *xchr*['122] ← 'R'; *xchr*['123] ← 'S'; *xchr*['124] ← 'T';
xchr['125] ← 'U'; *xchr*['126] ← 'V'; *xchr*['127] ← 'W';
xchr['130] ← 'X'; *xchr*['131] ← 'Y'; *xchr*['132] ← 'Z'; *xchr*['133] ← '['; *xchr*['134] ← '\';
xchr['135] ← ']'; *xchr*['136] ← '^'; *xchr*['137] ← '_';
xchr['140] ← '`'; *xchr*['141] ← 'a'; *xchr*['142] ← 'b'; *xchr*['143] ← 'c'; *xchr*['144] ← 'd';
xchr['145] ← 'e'; *xchr*['146] ← 'f'; *xchr*['147] ← 'g';
xchr['150] ← 'h'; *xchr*['151] ← 'i'; *xchr*['152] ← 'j'; *xchr*['153] ← 'k'; *xchr*['154] ← 'l';
xchr['155] ← 'm'; *xchr*['156] ← 'n'; *xchr*['157] ← 'o';
xchr['160] ← 'p'; *xchr*['161] ← 'q'; *xchr*['162] ← 'r'; *xchr*['163] ← 's'; *xchr*['164] ← 't';
xchr['165] ← 'u'; *xchr*['166] ← 'v'; *xchr*['167] ← 'w';
xchr['170] ← 'x'; *xchr*['171] ← 'y'; *xchr*['172] ← 'z'; *xchr*['173] ← '{'; *xchr*['174] ← '|';
xchr['175] ← '}'; *xchr*['176] ← '~';
xchr[0] ← ' '; *xchr*['177] ← ' '; { ASCII codes 0 and '177 do not appear in text }

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

This code is used in section 2.

8. ⟨ Local variables for initialization 8 ⟩ ≡
i: 0 .. *last_text_char*;

This code is used in section 2.

9. ⟨ Set initial values 7 ⟩ +≡
for *i* ← 1 **to** '37 **do** *xchr*[*i*] ← ' ';
for *i* ← *first_text_char* **to** *last_text_char* **do** *xord*[*chr*(*i*)] ← '177;
for *i* ← 1 **to** '176 **do** *xord*[*xchr*[*i*]] ← *i*;

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 6 ⟩ +≡
v: **array** [0 .. *max_m*, 0 .. *max_n*] **of** *real*; { pixel darknesses, from 0.0 to 1.0 }
m: *integer*; { rows 0 .. $m+1$ of v should contain relevant data }
n: *integer*; { columns 0 .. $n+1$ of v should contain relevant data }
dd: **array** [*text_char*] **of** *real*;

11. All input codes give zero density, except "0" to "9" and "A" to "V".

```
⟨ Set initial values 7 ⟩ +≡
  for i ← first_text_char to last_text_char do dd[chr(i)] ← 0.0;
  for i ← "0" to "9" do dd[chr(i)] ← 1.0 − (i − "0")/31.0;
  for i ← "A" to "V" do dd[chr(i)] ← 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 ⟩ ≡
  read(m); if (m ≤ 0) ∨ (m ≥ max_m) then finish;
  read_ln(n); if (n ≤ 0) ∨ (n ≥ max_n) then finish;
  for i ← 1 to m do
    begin for j ← 1 to n do
      begin read(c); v[i, j] ← dd[c];
      end;
    v[i, 0] ← v[i, 1]; v[i, n + 1] ← v[i, n];
    read_ln;
    end;
  for j ← 0 to n + 1 do
    begin v[0, j] ← v[1, j]; v[m + 1, j] ← v[m, j];
    end
```

This code is used in section 23.

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

```
⟨ Global variables 6 ⟩ +≡
i, j: integer;  { current row and column }
c: char;  { character read from input }
```

14. Pixel compensation. The 33-level output of this program is assumed to be printed by a font that contains 4×8 characters, where each character has 0 to 32 black bits. Physical properties of output devices cause 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.

```
  define max_l = 32  { maximum output level }
⟨ Global variables 6 ⟩ +≡
d: array [0 .. max_l] of real;  { 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 "N" seem to appear darker than blocks of the character "0", 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 ⟩ +≡
  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] ← 0.80;
  d[30] ← 0.88; d[31] ← 0.96; d[32] ← 1.0;
```

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 *next1* and *next2*; this will compensate for errors in the current halfline.

We assume that *next1*$[j]$, *next1*$[j+1]$, and *next2*$[j]$ correspond to the dots that are adjacent to *current*$[j]$.

```
⟨ Output one value and move to the next column 16 ⟩ ≡
  ⟨ Find l so that d[l] is as close as possible to x 21 ⟩;
  write(xchr["0" + l]); err ← x − d[l];
  next1[j] ← next1[j] + alpha * err;
  next2[j] ← beta * err;
  j ← j + 1;  { move right }
  next1[j] ← next1[j] + gamma * err;
  x ← current[j] + delta * err
```

This code is used in sections 18 and 18.

17. The constants *alpha* .. *delta* control the distribution of errors to adjacent dot positions.

```
⟨ Set initial values 7 ⟩ +≡
  alpha ← 7/16;  { error diffusion to SW neighbor }
  beta ← 1/16;  { error diffusion to S neighbor }
  gamma ← 5/16;  { error diffusion to SE neighbor }
  delta ← 3/16;  { error diffusion to E neighbor }
```

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 ".".

```
⟨ Output the picture 18 ⟩ ≡
  for j ← 1 to n + 1 do
    begin next1[j] ← 0.0; next2[j] ← 0.0;
    end;
  for i ← 1 to m do
    begin ⟨ Set the current halfline data for the upper row of dots in line i 19 ⟩;
    j ← 1; x ← current[1];
    repeat ⟨ Output one value and move to the next column 16 ⟩;
    until j > n;
    write_ln('.');  ⟨ Set the current halfline data for the lower row of dots in line i 20 ⟩;
    j ← 1; x ← current[1];
    repeat ⟨ Output one value and move to the next column 16 ⟩;
    until j > n;
    write_ln('.');
    end
```

This code is used in section 23.

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 *next1* and *next2* appropriately.

```
⟨ Set the current halfline data for the upper row of dots in line i 19 ⟩ ≡
  for j ← 1 to n do
    begin current[j] ← (9 * v[i, j] + 3 * v[i, j + 1] + 3 * v[i − 1, j] + v[i − 1, j + 1])/16 + next1[j + 1];
    next1[j] ← next2[j];
    end;
  next1[n + 1] ← 0.0
```

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.

```
⟨ Set the current halfline data for the lower row of dots in line i 20 ⟩ ≡
  for j ← 1 to n do
    begin current[j] ← (9 * v[i,j] + 3 * v[i,j-1] + 3 * v[i+1,j] + v[i+1,j-1])/16 + next1[j];
    next1[j+1] ← next2[j];
    end;
  next1[1] ← 0.0
```

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:

```
⟨ Find l so that d[l] is as close as possible to x 21 ⟩ ≡
  if x ≤ 0.0 then l ← 0
  else if x ≥ 1.0 then l ← max_l
    else begin low_l ← 0; high_l ← max_l;  { we have d[low_l] ≤ x < d[high_l] }
      while high_l - low_l > 1 do
        begin mid_l ← (low_l + high_l) div 2;
        if x ≥ d[mid_l] then low_l ← mid_l
        else high_l ← mid_l;
        end;
      if x - d[low_l] ≤ d[high_l] - x then l ← low_l else l ← high_l;
      end
```

This code is used in section 16.

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

```
⟨ Global variables 6 ⟩ +≡
x: real;  { current pixel density }
err: real;  { difference between x and the best we can achieve }
current: array [0 .. max_n] of real;  { desired densities in current halfline }
next1, next2: array [0 .. max_n] of real;  { corrections to subsequent densities }
alpha, beta, gamma, delta: real;  { constants of error diffusion }
l, low_l, mid_l, high_l: 0 .. max_l;  { trial density levels }
```

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

```
⟨ The main program 23 ⟩ ≡
  write_ln('\input␣hf33'); write_ln;
  while true do
    begin ⟨ Input a picture, or terminate the program 12 ⟩;
    write_ln('\beginhalftone'); ⟨ Output the picture 18 ⟩;
    write_ln('\endhalftone'); write_ln;
    end;
cleanup_and_terminate:
```

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:

```
program picfont(bytes_in, output);
  type ⟨ Types in the outer block 5 ⟩
  var ⟨ Global variables 6 ⟩
    ⟨ Basic procedures 10 ⟩
    begin ⟨ The main program 26 ⟩;
    end.
```

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

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

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

```
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.

```
⟨ Types in the outer block 5 ⟩ ≡
  eight_bits = 0 .. 255;  { unsigned one-byte quantity }
  byte_file = packed file of eight_bits;  { files that contain binary data }
```

This code is used in section 2.

6. ⟨ Global variables 6 ⟩ ≡

```
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.

```
⟨ Open the input file 7 ⟩ ≡
  reset(bytes_in, '', '/B:8')
```

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.

```
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 }
```

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 ⟩ +≡
ii: integer;  { the buffer holds rows 8ii − 7 through 8ii + 4 }
buffer: array [−2 .. 9, 0 .. nn + 1] of real;  { densities in twelve current rows }
darkness: array [−3 .. 9, 0 .. nn + 1] of white .. black;  { darknesses in buffer rows }
new_row: array [0 .. nn + 1] of real;  { densities in row being input }
```

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

```
⟨ Basic procedures 10 ⟩ ≡
procedure get_in(i : integer);
  var j: integer; t: eight_bits;  { byte of input }
  begin new_row[0] ← 0.0;
  if i > mm then
    for j ← 1 to nn do new_row[j] ← 0.0
  else for j ← 1 to nn do
      begin read(bytes_in, t); new_row[j] ← (255.5 − t)/256.0;
      end;
  new_row[nn + 1] ← 0.0;
  end;
```

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 ⟩ +≡
procedure roll;
  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;
```

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 ⟩ ≡
  ii ← 0;
  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
```

This code is used in section 26.

13. It's easy to output the current darkness values. Here we output eight consecutive rows.

```
⟨ Output the pixel values for the top eight rows of the buffer 13 ⟩ ≡
  for i ← −2 to 5 do
    begin write('row(',8 * ii − 5 + i : 1,');␣cols('); cols_out ← 0;
    for j ← 1 to nn do
      if darkness[i, j] = black then
        begin if cols_out < 15 then
          begin if cols_out > 0 then write(',');
          incr(cols_out);
          end
        else begin write_ln(','); write('␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'); cols_out ← 1;
          end;
        write(j : 1);
        end;
    write_ln(');')
    end
```

This code is used in section 26.

14. ⟨ Global variables 6 ⟩ +≡

```
cols_out: 0 .. 15;  { 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 *class_row*, *class_col*, *start*, *del_i*, *del_j*, and *alpha* whose function is easy to deduce from the following code:

```
⟨ Choose pixel values and diffuse the errors in the buffer 15 ⟩ ≡
  for k ← 0 to 63 do
    begin i ← class_row[k]; j ← class_col[k];
    while j ≤ nn do
      begin ⟨ Decide the color of pixel [i, j] and the resulting err 17 ⟩;
      for l ← start[k] to start[k + 1] − 1 do
        begin u ← i + del_i[l]; v ← j + del_j[l]; buffer[u, v] ← buffer[u, v] + err * alpha[l];
        end;
      j ← j + 8;
      end;
    end
```

This code is used in section 26.

16. ⟨ Global variables 6 ⟩ +≡

```
class_row: array [0 .. 63] of −2 .. 8;  { buffer row containing pixels of a given class }
class_col: array [0 .. 63] of 1 .. 8;  { first column containing pixels of a given class }
class_number: array [−2 .. 9, 0 .. 9] of 0 .. 63;  { number of a given position }
err: real;  { error introduced at the current position }
err_black: real;  { error introduced at the current position if black chosen }
black_diff: real;  { difference between err and err_black for gray pixel }
l: 0 .. 256;  { index into diffusion tables }
start: array [0 .. 64] of 0 .. 256;  { first entry of diffusion table for a given class }
del_i, del_j: array [0 .. 256] of −1 .. 1;  { neighboring location for diffusion }
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.

```
⟨ Decide the color of pixel [i,j] and the resulting err 17 ⟩ ≡
  if darkness[i,j] = gray then
    begin err ← buffer[i,j] − zeta; err_black ← err − black_diff;
    end
  else begin err ← buffer[i,j]; err_black ← err − 1.0;
    end;
  if darkness[i − 1,j] = white then err_black ← err_black − zeta;
  if darkness[i,j − 1] = white then err_black ← err_black − zeta;
  if darkness[i,j + 1] = white then err_black ← err_black − zeta;
  if darkness[i + 1,j] = white then err_black ← err_black − zeta;
  if err_black + err > 0 then
    begin err ← err_black; darkness[i,j] ← black;
    if darkness[i − 1,j] = white then darkness[i − 1,j] ← gray;
    if darkness[i,j − 1] = white then darkness[i,j − 1] ← gray;
    if darkness[i,j + 1] = white then darkness[i,j + 1] ← gray;
    if darkness[i + 1,j] = white then darkness[i + 1,j] ← gray;
    end
```

This code is used in section 15.

18.
```
⟨ Initialize the diffusion tables 18 ⟩ ≡
  black_diff ← 1.0 − 2.0 * 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.

```
⟨ Initialize the diffusion tables 18 ⟩ +≡
  ⟨ Initialize the class number matrix 21 ⟩;
  ⟨ 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 —dot300— font that was described earlier.

```
⟨ Basic procedures 10 ⟩ +≡
procedure store(i,j : integer); { establish new class_row, class_col }
  begin if i < 1 then i ← i + 8 else if i > 8 then i ← i − 8;
  if j < 1 then j ← j + 8 else if j > 8 then j ← j − 8;
  class_number[i,j] ← k; class_row[k] ← i; class_col[k] ← j; incr(k);
  end;

procedure store_eight(i,j : integer); { rotate and shift for eight classes }
  begin store(i,j); store(i − 4,j + 4); store(5 − j,i); store(1 − j,i − 4);
  store(4 + j,1 − i); store(j,5 − i); store(5 − i,5 − j); store(1 − i,1 − j);
  end;
```

21. ⟨ Initialize the class number matrix 21 ⟩ ≡

```
k ← 0; store_eight(7,2); store_eight(8,3); store_eight(8,2); store_eight(8,1);
store_eight(1,4); store_eight(1,3); store_eight(1,2); store_eight(2,3);
for i ← 1 to 8 do
  begin class_number[i,0] ← class_number[i,8]; class_number[i,9] ← class_number[i,1];
  end;
for j ← 0 to 9 do
  begin class_number[−2,j] ← class_number[6,j]; class_number[−1,j] ← class_number[7,j];
  class_number[0,j] ← class_number[8,j]; class_number[9,j] ← class_number[1,j];
  end
```

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.

```
⟨ Global variables 6 ⟩ +≡
hold: array [0 .. 9, 0 .. 9] of boolean;
        { 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).

```
⟨ Compile "instructions" for the diffusion operations 23 ⟩ ≡
  for j ← 0 to 9 do hold[9,j] ← true;
  for i ← 0 to 8 do
    for j ← 0 to 9 do hold[i,j] ← false;
  l ← 0; k ← 0;
  repeat i ← class_row[k]; j ← class_col[k]; w ← 0; start[k] ← l;
    for u ← i − 1 to i + 1 do
      for v ← j − 1 to j + 1 do
        if class_number[u,v] > k then
          begin del_i[l] ← u − i; del_j[l] ← v − j; incr(l);
          if u = i then w ← w + 2   { neighbors in the same row get weight 2 }
          else if v = j then w ← w + 2   { neighbors in the same column get weight 2 }
            else incr(w);   { diagonal neighbors get weight 1 }
          end
        else if hold[u,v] then hold[i,j] ← true;
    if hold[i,j] then class_row[k] ← i − 8;
    ⟨ Compute the alpha values for class k, given the total weight w 24 ⟩;
    incr(k);
  until k = 64;
  start[64] ← l
```

This code is used in section 19.

24. ⟨ Compute the *alpha* values for class k, given the total weight w 24 ⟩ ≡

```
  for ll ← start[k] to l − 1 do
    begin if del_i[ll] = 0 then alpha[ll] ← 2.0/w
    else if del_j[ll] = 0 then alpha[ll] ← 2.0/w
      else alpha[ll] ← 1.0/w;
    end
```

This code is used in section 23.

25. ⟨ Global variables 6 ⟩ +≡

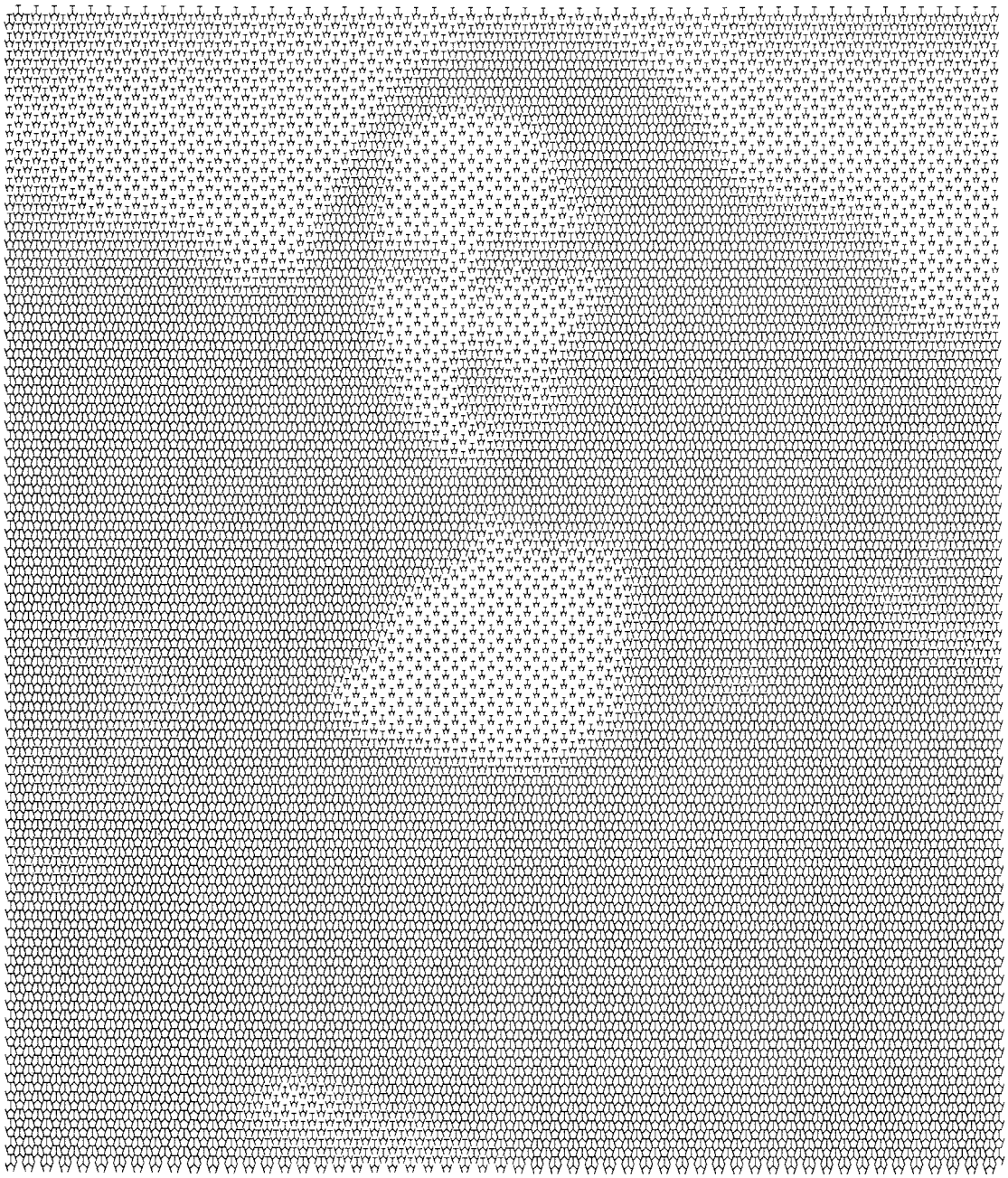
```
ll: 0 .. 256;   { loop index }
u, v: integer;   { neighbors of i and j }
w: integer;   { the weighted number of high-class neighbors }
i, j: integer;   { the current pixel position being considered }
k: 0 .. 64;   { the current class being considered }
```

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 ⟩;
    roll;
  until 8 * ii > mm
```

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 8×8 character essentially contributes two dots to a picture; and (2) a halftone font with 33 levels of gray, in which each 4×8 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 4×4 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 kth level of gray in the half-dot scheme is obtained by blackening cells 0 to $k-1$ in the array

1	5	10	14
3	7	8	12
13	9	6	2
15	11	4	0

or

14	10	5	1
12	8	7	3
2	6	9	13
0	4	11	15

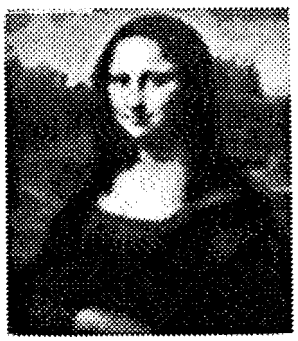
.

(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 `hd300`, in essentially the same way that the other fonts `dot300` and `hf300` were generated earlier:

```
% 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
```

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



single dot

half dot

dot diffusion

(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
mMmMmLlLkKjJiIiIjJjJjJiIgGiJjJiGgGgIiIgGgGgGgGgIgFgGgGgGgFfFeFfFeGgGgGgGgGgFfFgIi ...
LmMlLkKlLkJiIiHiIiJjJkJiGgIiIiGgFgGgGgGgGfFfGgFfEfFgFfFfGgEgGfEfGgGgGgGfFfFgGeGiH ...
kKkKkKkKkJiIiHiIiIjJkKkJiIgGgFgIgFfFfFgGfEeEeEeEfGeGgGgHiJjJjIgGiGfGgGgFeFeGgFgGg ...
JjJjKkKkKjIiGiIgGiJkJjJiIiIgGfGiGfEfEfGfFeEdEeDdEeFiKlMnNnNnNnMlKiGfFfFfFeFfFeFgG ...
jJkJkKkLkJiHgGgGgIiJjJjIgGgFeFeGgGfEeEfEeDdDdEdDeJlNnOnOoOoOoOmOoNmKgEeDeEeEeDeEf ...
```

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.