November 24, 2020 at 13:23

1. Introduction. This program prepares a METAFONT file for a special-purpose font that will approximate a given picture. The input file (*stdin*) is assumed to be an EPS file output by Adobe Photoshop™ on a Macintosh with the binary EPS option, containing m rows of n columns each; in Photoshop terminology the image is m pixels high and n pixels wide, in grayscale mode, with a resolution of 72 pixels per inch. The output file (*stdout*) will be a sequence of m lines like

```
row 10; data "d01...53";
```

this means that the pixel data for row 10 is the string of n bits 110100000001...01010011 encoded as a hexadecimal string of length $n/4$.

For simplicity, this program assumes that $m = 512$ and $n = 440$.

```
#define m 512   /* this many rows */
#define n 440   /* this many columns */
#include <stdio.h>
  float a[m + 2][n + 2];   /* darknesses: 0.0 is white, 1.0 is black */
  ⟨ Global variables 4 ⟩
  ⟨ Subroutines 12 ⟩
  main(argc, argv)
      int argc;
      char *argv[];
  {
    register int i, j, k, l, ii, jj, w;
    register float err;
    float zeta = 0.2, sharpening = 0.9;
    ⟨ Check for nonstandard zeta and sharpening factors 2 ⟩;
    ⟨ Check the beginning lines of the input file 3 ⟩;
    ⟨ Input the graphic data 5 ⟩;
    ⟨ Translate input to output 15 ⟩;
    ⟨ Spew out the answers 17 ⟩;
  }
```

2. Optional command-line arguments allow the user to change the *zeta* and/or *sharpening* parameters discussed below.

```
⟨ Check for nonstandard zeta and sharpening factors 2 ⟩ ≡
  if (argc > 1 ∧ sscanf(argv[1], "%g", &zeta) ≡ 1) {
    fprintf(stderr, "Using␣zeta␣=␣%g\n", zeta);
    if (argc > 2 ∧ sscanf(argv[2], "%g", &sharpening) ≡ 1)
      fprintf(stderr, "␣␣and␣sharpening␣factor␣%g\n", sharpening);
  }
```

This code is used in section 1.

3. Macintosh conventions indicate the end of a line by the ASCII ⟨ carriage return ⟩ character (i.e., control-M, aka `\r`), but the C library is set up to work best with newlines (i.e., control-J, aka `\n`). We aren't worried about efficiency, so we simply input one character at a time. This program assumes Macintosh conventions.

The job here is to look for the sequence `Box:` in the input, followed by 0, 0, the number of columns, and the number of rows.

```
#define panic(s)
        {
          fprintf(stderr, s); exit(−1);
        }
⟨ Check the beginning lines of the input file 3 ⟩ ≡
  k = 0;
scan:
  if (k++ > 1000) panic("Couldn't␣find␣the␣bounding␣box␣info!\n");
  if (getchar() ≠ 'B') goto scan;
  if (getchar() ≠ 'o') goto scan;
  if (getchar() ≠ 'x') goto scan;
  if (getchar() ≠ ':') goto scan;
  if (scanf("%d␣%d␣%d␣%d", &llx, &lly, &urx, &ury) ≠ 4 ∨ llx ≠ 0 ∨ lly ≠ 0)
    panic("Bad␣bounding␣box␣data!\n");
  if (urx ≠ n ∨ ury ≠ m) panic("The␣image␣doesn't␣have␣the␣correct␣width␣and␣height!\n");
```

This code is used in section 1.

4. ⟨ Global variables 4 ⟩ ≡

```
  int llx, lly, urx, ury;    /* bounding box parameters */
```

See also sections 8 and 11.

This code is used in section 1.

5. After we've seen the bounding box, we look for `beginimage\r`; this will be followed by the pixel data, one character per byte.

```
⟨ Input the graphic data 5 ⟩ ≡
  k = 0;
skan:
  if (k++ > 10000) panic("Couldn't␣find␣the␣pixel␣data!\n");
  if (getchar() ≠ 'b') goto skan;
  if (getchar() ≠ 'e') goto skan;
  if (getchar() ≠ 'g') goto skan;
  if (getchar() ≠ 'i') goto skan;
  if (getchar() ≠ 'n') goto skan;
  if (getchar() ≠ 'i') goto skan;
  if (getchar() ≠ 'm') goto skan;
  if (getchar() ≠ 'a') goto skan;
  if (getchar() ≠ 'g') goto skan;
  if (getchar() ≠ 'e') goto skan;
  if (getchar() ≠ '\r') goto skan;
  ⟨ Input rectangular pixel data 6 ⟩;
  if (getchar() ≠ '\r') panic("Wrong␣amount␣of␣pixel␣data!\n");
```

This code is used in section 1.

6. Photoshop follows the conventions of photographers who consider 0 to be black and 1 to be white; but we follow the conventions of computer scientists who tend to regard 0 as devoid of ink (white) and 1 as full of ink (black).

We use the fact that global arrays are initially zero to assume that there are all-white rows of 0s above, below, and to the left and right of the input data.

$\langle$ Input rectangular pixel data 6 $\rangle \equiv$
  for $(i = 1;\ i \le ury;\ i{+}{+})$
    for $(j = 1;\ j \le urx;\ j{+}{+})\ a[i][j] = 1.0 - getchar(\,)/255.0;$

This code is used in section 5.

7. Dot diffusion. Our job is to go from eight-bit pixels to one-bit pixels; that is, from 256 shades of gray to an approximation that uses only black and white. The method used here is called *dot diffusion* (see [D. E. Knuth, "Digital halftones by dot diffusion," *ACM Transactions on Graphics* **6** (1987), 245–273]); it works as follows: The pixels are divided into 64 classes, numbered from 0 to 63. We convert the pixel values to 0s and 1s by assigning values 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 segment.

```
⟨ Choose pixel values and diffuse the errors in the buffer 7 ⟩ ≡
  for (k = 0; k < 64; k++)
    for (i = class_row[k]; i ≤ m; i += 8)
      for (j = class_col[k]; j ≤ n; j += 8) {
        ⟨ Decide the color of pixel [i, j] and the resulting err 9 ⟩;
        for (l = start[k]; l < start[k + 1]; l++) a[i + del_i[l]][j + del_j[l]] += err * alpha[l];
      }
```

This code is used in section 15.

8. We will use the following model for estimating the effect of a given bit pattern in the output: If a pixel is black, its darkness is 1.0; if it is white but at least one of its four neighbors is black, its darkness is *zeta*; if it is white and has four white neighbors, its darkness is 0.0. Laserprinters of the 1980s tended to spatter toner in a way that could be approximated roughly by taking $zeta = 0.2$ in this model. The value of *zeta* should be between -0.25 and $+1.0$.

An auxiliary array *aa* holds code values *white*, *gray*, or *black* to facilitate computations in this model. All cells are initially *white*; but when we decide to make a pixel *black*, we change its *white* neighbors (if any) to *gray*.

```
#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 */
```

```
⟨ Global variables 4 ⟩ +≡
  char aa[m + 2][n + 2];   /* white, gray, or black status of final pixels */
```

9. In this step the current pixel's final one-bit value is determined. It presently is either *white* or *gray*; we either leave it as is, or we blacken it and gray its white neighbors, whichever minimizes the magnitude of the error.

Potentially *gray* values near the newly chosen pixel make this calculation slightly tricky. Notice, for example, that the very first black pixel to be created will increase the local darkness of the image by $1 + 4zeta$. Suppose the original image is entirely black, so that $a[i][j]$ is 1.0 for $1 \le i \le m$ and $1 \le j \le n$. If a pixel of class 0 is set to *white*, the error (i.e., the darkness that needs to be diffused to its upperclass neighbors) is 1.0; but if it is set to *black*, the error is $-4zeta$. The algorithm will choose *black* unless $zeta \ge .25$.

```
⟨Decide the color of pixel [i, j] and the resulting err 9⟩ ≡
  if (aa[i][j] ≡ white) err = a[i][j] − 1.0 − 4 ∗ zeta;
  else {    /∗ aa[i][j] ≡ gray ∗/
    err = a[i][j] − 1.0 + zeta;
    if (aa[i − 1][j] ≡ white) err −= zeta;
    if (aa[i + 1][j] ≡ white) err −= zeta;
    if (aa[i][j − 1] ≡ white) err −= zeta;
    if (aa[i][j + 1] ≡ white) err −= zeta;
  }
  if (err + a[i][j] > 0) {    /∗ black is better ∗/
    aa[i][j] = black;
    if (aa[i − 1][j] ≡ white) aa[i − 1][j] = gray;
    if (aa[i + 1][j] ≡ white) aa[i + 1][j] = gray;
    if (aa[i][j − 1] ≡ white) aa[i][j − 1] = gray;
    if (aa[i][j + 1] ≡ white) aa[i][j + 1] = gray;
  }
  else err = a[i][j];    /∗ keep it white or gray ∗/
```

This code is used in section 7.

10. 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 reveals some of the mysterious underlying structure.

⟨ Initialize the diffusion tables 10 ⟩ ≡
  ⟨ Initialize the class number matrix 13 ⟩;
  ⟨ Compile "instructions" for the diffusion operations 14 ⟩;

This code is used in section 15.

11. ⟨ Global variables 4 ⟩ +≡

```
char class_row[64], class_col[64];    /* first row and column for a given class */
char class_number[10][10];    /* the number of a given position */
int kk = 0;    /* how many classes have been done so far */
int start[65];    /* the first instruction for a given class */
int del_i[256], del_j[256];    /* relative location of a neighbor */
float alpha[256];    /* diffusion coefficient for a neighbor */
```

12. 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, it is precisely the pattern used in the font `ddith300`, discussed in the author's paper "Fonts for Digital Halftones" [Chapter 21 of *Selected Papers on Digital Typography*].

⟨ Subroutines 12 ⟩ ≡

```
void store(i, j)
    int i, j;
{
  if (i < 1) i += 8; else if (i > 8) i -= 8;
  if (j < 1) j += 8; else if (j > 8) j -= 8;
  class_number[i][j] = kk;
  class_row[kk] = i; class_col[kk] = j;
  kk++;
}
void store_eight(i, j)
    int i, j;
{
  store(i, j); store(i - 4, j + 4); store(1 - j, i - 4); store(5 - j, i);
  store(j, 5 - i); store(4 + j, 1 - i); store(5 - i, 5 - j); store(1 - i, 1 - j);
}
```

This code is used in section 1.

13. ⟨ Initialize the class number matrix 13 ⟩ ≡

```
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; i <= 8; i++) class_number[i][0] = class_number[i][8], class_number[i][9] = class_number[i][1];
for (j = 0; j <= 9; j++) class_number[0][j] = class_number[8][j], class_number[9][j] = class_number[1][j];
```

This code is used in section 10.

14. The "compilation" in this step simulates going through the diffusion process the slow way, recording the actions it performs. Then those actions can all be done at high speed later.

⟨ Compile "instructions" for the diffusion operations 14 ⟩ ≡

```
for (k = 0, l = 0; k < 64; k++) {
  start[k] = l;    /* l is the number of instructions compiled so far */
  i = class_row[k]; j = class_col[k]; w = 0;
  for (ii = i − 1; ii ≤ i + 1; ii++)
    for (jj = j − 1; jj ≤ j + 1; jj++)
      if (class_number[ii][jj] > k) {
        del_i[l] = ii − i; del_j[l] = jj − j; l++;
        if (ii ≠ i ∧ jj ≠ j) w++;    /* diagonal neighbors get weight 1 */
        else w += 2;    /* orthogonal neighbors get weight 2 */
      }
  for (jj = start[k]; jj < l; jj++)
    if (del_i[jj] ≠ 0 ∧ del_j[jj] ≠ 0) alpha[jj] = 1.0/w;
    else alpha[jj] = 2.0/w;
}
start[64] = l;    /* at this point l will be 256 */
```

This code is used in section 10.

15. Synthesis. Now we're ready to put the pieces together.

⟨ Translate input to output 15 ⟩ ≡
```
⟨ Initialize the diffusion tables 10 ⟩;
if (sharpening) ⟨ Sharpen the input 16 ⟩;
⟨ Choose pixel values and diffuse the errors in the buffer 7 ⟩;
```
This code is used in section 1.

16. Experience shows that dot diffusion often does a better job if we apply a filtering operation that exaggerates the differences between the intensities of a pixel and its neighbors:

$$a_{ij} \leftarrow \frac{a_{ij} - \alpha \bar{a}_{ij}}{1 - \alpha},$$

where $\bar{a}_{ij} = \frac{1}{9}\sum_{\delta=-1}^{+1}\sum_{\epsilon=-1}^{+1} a_{(i+\delta)(j+\epsilon)}$ is the average value of a_{ij} and its eight neighbors. (See the discussion in the *Transactions on Graphics* paper cited earlier. The parameter α is the *sharpening* value, which must obviously be less than 1.0.)

We could use a buffering scheme to apply this transformation in place, but it's easier to store the new value of a_{ij} in $a_{(i-1)(j-1)}$ and then shift everything back into position afterwards. The values of a_{i0} and a_{0j} don't have to be restored to zero after this step, because they will not be examined again.

⟨ Sharpen the input 16 ⟩ ≡
```
{
  for (i = 1; i ≤ m; i++)
    for (j = 1; j ≤ n; j++) { float abar;
      abar = (a[i − 1][j − 1] + a[i − 1][j] + a[i − 1][j + 1] + a[i][j − 1] +
          a[i][j] + a[i][j + 1] + a[i + 1][j − 1] + a[i + 1][j] + a[i + 1][j + 1])/9.0;
      a[i − 1][j − 1] = (a[i][j] − sharpening ∗ abar)/(1.0 − sharpening);
    }
  for (i = m; i > 0; i−−)
    for (j = n; j > 0; j−−) a[i][j] = (a[i − 1][j − 1] ≤ 0.0 ? 0.0 : a[i − 1][j − 1] ≥ 1.0 ? 1.0 : a[i − 1][j − 1]);
}
```
This code is used in section 15.

17. Here I'm assuming that n is a multiple of 4.

⟨ Spew out the answers 17 ⟩ ≡
```
for (i = 1; i ≤ m; i++) {
  printf("row␣%d;␣data␣\"", i);
  for (j = 1; j ≤ n; j += 4) {
    for (k = 0, w = 0; k < 4; k++) w = w + w + (aa[i][j + k] ≡ black ? 1 : 0);
    printf("%x", w);
  }
  printf("\";\n");
}
```
This code is used in section 1.

18. Index.

DOT-DIFF