Floyd-Steinberg Dithering

STEPHEN M. OMOHUNDRO

International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, California 94704
Phone: 415-643-9153
Internet: om@icsi.berkeley.edu
October 25, 1990

The problem of converting a color or grayscale image to an image with a smaller number of colors or intensity levels arises in many settings. We present an efficient implemention technique for dithering by error diffusion.

We focus on the common case of converting an 8 bit grayscale image to a bitmap but mention the more general setting at the end. Many output devices, such as dot-matrix and laser-printers, bitmap displays and many printing processes are only capable of generating two intensity levels. This leads to the common problem of displaying images with 256 intensity levels on such output devices. An obvious approach is to assign a rectangular patch of pixels in the output bitmap to each pixel of the original image and to fill it with a pattern of pixels whose density is proportional to the gray level to be represented. For example, the gray levels 0-255 might be represented by patterns in 32 by 32 squares which proceed from an empty square to an almost filled square. Unfortunately, this approach requires the output image be larger than the original image and the square dithering regions tend to be very visible and annoying.

An approach which doesn't suffer from these problems was suggested by Floyd and Steinberg in 1976. The idea of the so-called "error diffusion" approach to dithering is to produce a pattern of pixels such that the average intensity over regions in the output bitmap is approximately the same as the average over the same region in the orginal image. The idea is to keep track of the error made in replacing each original pixel by a single bit and to add this error to the intensity of neighboring pixels in such a way that the average

remains close to the orignal. It is convenient to compute the output pixels in scanline order from upper left to lower right. Each pixel's intensity is compared with the intermediate value 128. If it is less than this, a black pixel is output (value 0), otherwise a white pixel is output (value 255). The difference between what we actually output and what we should have output is the error. Because we don't want to alter the already computed pixels, we spread this error intensity only to the pixels on the right, the lower-right, the bottom, and the lower left. The amount of error which is spread to each neighbor may be varied, but sending 3/8 of the error to the right and lower pixels and 1/8 to the two diagonal neighbors gives good results.

Figure 1

The one-dimensional analog of this process is similar to Bresenham's line drawing algorithm. In this case we are outputting a row of dots whose running average intensity should be close to the true average. The error in this case is only spread to the neighboring pixel on the right. If you were to plot the accumulated intensity along the line, then the slope of this graph is supposed to be the given gray scale intensity. If this were constant, then we would have the problem of approximating a straight line of given slope by discrete steps. Bresenham's approach to line drawing keeps track of an accumulated error which causes a shift when it reaches a threshold at which point the error is decremented exactly analogously to error diffusion.

In practice, the bytes of the original 8-bit image are usually packed tightly in memory and the entire range of 0-255 is used for representing intensity levels. The accumulating error at a pixel can easily be larger than this range or negative. If the dithering error is directly diffused in the orginal image, many pixel values will overflow or underflow leading to severe problems in the resulting bitmap. This might lead one to think that the error diffusion process will require first allocating a two-dimensional array of 32 bit integers of the same size as the original image and copying the intensity levels over. For large images this can be prohibitive in space and expensive in time. A better solution is apparent when one considers the pixels whose values are altered at any point during the process. As shown in figure 2, only the row beneath the current row up to the current pixel, the current pixel, and the current row until the end have values which are different from the original image. We may declare an array of integers which is sufficiently large to hold the accumulated error for these pixels (the size of a row plus 1). As the process proceeds the correspondence between the error pixel and the orginal pixel changes but no error values are lost.

Figure 2

Using a separate array of integers to represent the errors of the modified pixels, we may code the algorithm quite simply:

```
procedure dither
(
im_width, im_height,                        -- size of image
get_pixel(x,y), set_output_pixel(x,y)       -- access to pixels
)

error_arr:array[0..im_width] of integer;    -- holds the errors
error:integer;                              -- error for current pixel

y:integer <- 0;
until y=im_height do
   x:integer <- 0;
   until x=im_width-1 loop                  -- don't go all the way to edge
      val<-get_pixel(x,y)+error_arr[x+1];   -- pixel value with error
      if val>128                            -- above threshold
      then set_output_pixel(x,y);  error <- val-255; -- output white
      else error <- val;                    -- leave output black
      if x/=0 then                          -- don't spread left at edge
         error_arr[x-1] <- error_arr[x-1]+error/8; -- lower left pixel
      error_arr[x] <- error_arr[x]+3*error/8; -- pixel below
      error_arr[x+1] <- error/8;            -- initial error for lower right
      error_arr[x+2] <- error_arr[x+2]+3*error/8; -- pixel to the right
      x<-x+1;
   endloop;
   y<-y+1;
endloop;
```

The result of applying the algorithm to a typical scene is shown in figure 3.

Figure 3

The algorithm as presented tends to produce dot patterns which have too much structure at low intensity levels. As discussed in Ulichney, 1987, a simple way to improve the output is to vary the threshold randomly from 128. Because all the error is still accounted for, the intensity averages will still be close to the correct value, but patterns will be broken up. There is no need to compute random values for each pixel, a small table of random thresholds is sufficient.

Exactly the same ideas may be applied to situations in which the output pixels have a range of intensities instead of just white and black. The algorithm should choose the best output representative of the input (say by choosing the one closest in intensity) and spread the error. For color images, the nearest entry in the color map is chosen and the error is accumulated in each color channel. Many of the artifacts of representing a 24 bit color image with an 8 bit color map can be eliminated with this kind of dithering. Suprisingly good color images can be achieved with only two bits per pixel. The color map assigns the four possible values to black, saturated red, saturated green, and saturated blue. A three dimensional color error is diffused exactly as the monochrome case. In this way one may pack 4 high quality full color images into a single 8 bit frame buffer. By cycling the color map appropriately, one may achieve 4 frames of color animation on even inexpensive systems.

References

Floyd, R. W. and L. Steinberg (1976). "An adaptive algorithm for spatial greyscale", Proc. SID, vol. 17/2, pp. 75-77.

Ulichney, Robert (1987). Digital Halftoning. MIT Press, Cambridge, Massachusetts.

C code for Sun workstations

```
'/* This program will display dithered versions of color rasterfiles in
   a window.
   Compile it as:
   cc -O simpdither.c -o simpdither -lsuntool -lsunwindow -lpixrect
*/

#include <stdio.h>
#include <sys/file.h>
#include <suntool/tool_hs.h>

structgfxsubwindow {
  intgfx_windowfd;
  intgfx_flags;
#defineGFX_RESTART0x01
#defineGFX_DAMAGED0x02
  intgfx_reps;
  structpixwin *gfx_pixwin;
  structrect gfx_rect;
  caddr_tgfx_takeoverdata;
};

externstruct gfxsubwindow *gfxsw_init();
struct gfxsubwindow *picwin;

main(argc, argv)
     int argc;
     char *argv[];
{
  int ixsize, iysize, mxbytes, sxbytes;
  struct pixrect *source_pixrect, *pr_load(), *mem_pixrect;
  unsigned char *malias, *salias;
  FILE *imagefp, *fopen();
  colormap_t colormap;
  short line[1500];/* holds temp dither error values */
  unsigned char gmap[256];/* holds gray equivalents for color values */

  if (argc!=2)
    {fprintf(stderr,"Usage: %s rasterfile\n", argv[0]); exit(1);}

  if ((imagefp = fopen(argv[1],"r")) == NULL)
    {fprintf(stderr, "%s: can't open %s\n", *argv, argv[1]); exit(2);}

  if ((picwin = gfxsw_init(0,(char**)0)) == (struct gfxsubwindow *) 0)
    exit(1);

  if (picwin->gfx_flags&GFX_DAMAGED)
    {
      gfxsw_handlesigwinch(picwin);
      pw_write(picwin->gfx_pixwin,0,0,1280,1280,PIX_SRC,0,0,0);
    }
  if (picwin->gfx_flags&GFX_RESTART)
    picwin->gfx_flags &= ~GFX_RESTART;

  /* set up colormap */
  colormap.length = 256;
  colormap.map[0] = (unsigned char *)malloc(colormap.length);
  colormap.map[1] = (unsigned char *)malloc(colormap.length);
  colormap.map[2] = (unsigned char *)malloc(colormap.length);
```

```
  /* load in the image file */
  source_pixrect = pr_load(imagefp, colormap);

  ixsize = source_pixrect->pr_size.x; /* size of image */
  iysize = source_pixrect->pr_size.y;
  if (source_pixrect->pr_depth != 8)
    (fprintf(stderr, "%s: Must use an 8 bit image\n", *argv);exit(3);)

  /* Make the gray map gmap */
  for(i=0; i<256; i++)
    gmap[i] = (colormap.map[0][i] + colormap.map[1][i] + colormap.map[2][i])/3;

  mem_pixrect = mem_create(ixsize, iysize, 1);/* output pixrect */
  mxbytes = mpr_d(mem_pixrect)->md_linebytes;
  malias = (unsigned char *) mpr_d(mem_pixrect)->md_image;
  sxbytes = mpr_d(source_pixrect)->md_linebytes;
  salias = (unsigned char *) mpr_d(source_pixrect)->md_image;

  /* Do the actual dithering. */
  {
    int lpt;/* where we are in line */
    short error, val, fer, eer, teer, lr, x, y;
    for(y = 0; y < iysize; y++, salias+=sxbytes, malias+=mxbytes)
      {
  lr = 0;/* error for lower right */
  for(x=1, lpt = 1; x<ixsize; x++, lpt++)
    {
      if ((val = (line[lpt] + *(salias + x))) < 128)
        {
    *(malias + ((lpt-1)>>3))|=1<<(7-((lpt-1)%8)); /* set bit */
    error = val;
        }
      else
        error = val - 255;
      fer = error/4;/* a fourth of the error */
      eer = fer/2;/* an eighth of the error */
      teer = fer + eer;/* three eighths of the error */
      line[lpt-1] += eer;/* represents lower left */
      line[lpt] = teer + lr; /* represents lower pixel now */
      lr = eer;/* lower right */
      line[lpt+1] += teer; /* right neigbor pixel */
    }
  if (y%8 == 0)
    pw_write(picwin->gfx_pixwin, 0, 0, ixsize, iysize, PIX_SRC,
       mem_pixrect, 0, 0);/* put image up on screen */
      }
  }

  pw_write(picwin->gfx_pixwin, 0, 0, ixsize, iysize, PIX_SRC,
     mem_pixrect, 0, 0);/* put image up on screen */

  pr_destroy(source_pixrect);/* get rid of pixrects */
  pr_destroy(mem_pixrect);

  while(1)/* loop forever to keep image on screen */
    {
    }
}
```

Figures

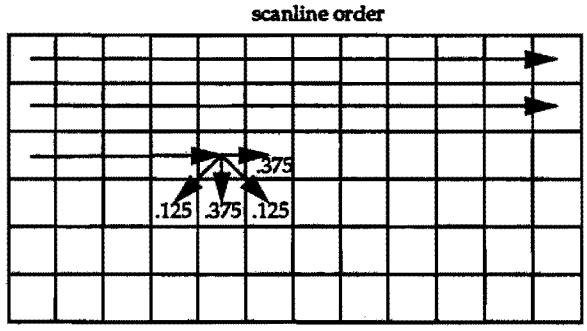

Figure 1. How the error is spread from a pixel.

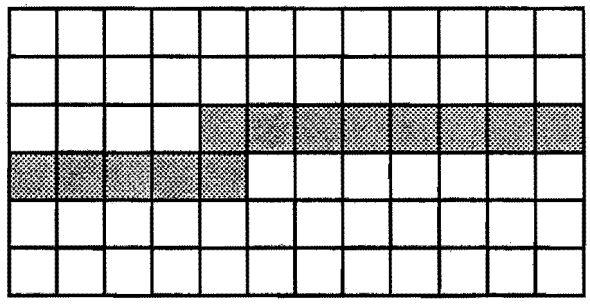

Figure 2. The pixels represented by the error array.

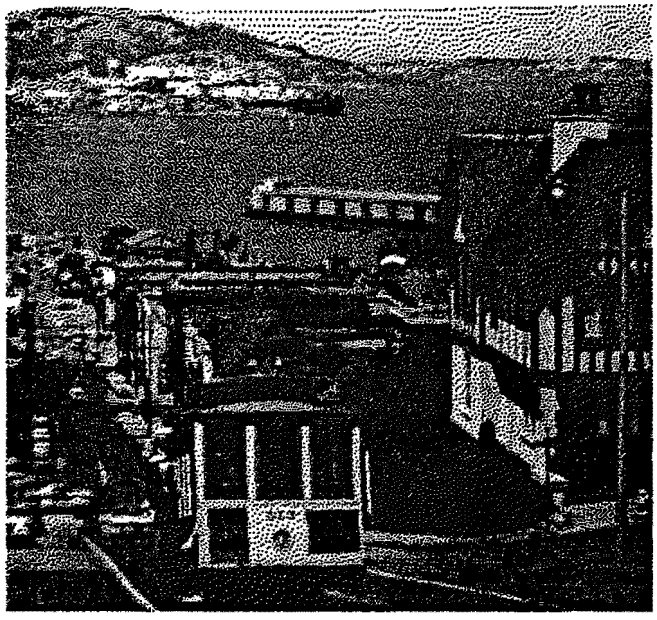

Figure 3. A typical dithered scene.