PHILIPS

PHILIPS INTERACTIVE MEDIA

Technical Note #83

A Compression Algorithm for Monochrome Images

Graham Trott, BEPL

October 10, 1992

A compression algorithm was developed during solution of the problem of putting up a graphical error screen, which needed to be kept in memory at all times. The compression method discussed herein results in a considerable reduction of the image storage requirements for these error screens.

A Compression Algorithm for Monochrome Images

The minimal error handling strategy recommended by PIMA Technical Note #54, *Error Strategy for CD-I Final Product Deliverable,* requires an information screen to appear when a fatal error occurs. Where the title makes no use of font modules, the error screen has to be implemented as a graphic. Since it must be present at all times , size is a major consideration. (There's not much point in keeping disc error messages on the disc.)

A typical message comprises six lines of text (18-point Helvetica bold). The example used by the author compresses to 7500 bytes using RL3 coding; however, by using the technique described herein, the image storage requirements are reduced to 4200 bytes.

The title in which this was used has two message screens: one for disc errors and the other for anything else reported by Balboa (overruns, memory full, etc.). The two screens and ten numeric digits (see below) take up a total of 9100 bytes. This is a tolerable memory size to lose.

The technique is one of run-length compression, with an additional saving made by implementing a "repeat last line N times" feature. Only two colors are catered for, which is the chief reason that the result is nearly twice as efficient as RL3. The first listing (**CompressImage**) is the encoder routine. This assumes that a CLUT7 IFF file is already open (**fileRef**), as is an output data file (**outFileRef**). The offset to the start of the image data (in the input file) is passed as **filePos**; the width and height (in pixels) of the image are passed as **picW** and **picH**.

The coding uses bit 7 of a byte to indicate the color (foreground or background) and the remaining 7 bits for the run length of that color. Two special cases are:

1. When the run length is 127, it is interpreted as "continue to end of line." This implies that only run lengths of 1-126 are otherwise valid.
2. When the entire byte is zero, the following byte is taken to be the number of lines to be repeated (up to 255).

The second listing (**ExpandImage**) is the corresponding run-time decoder. You must have previously set up a CLUT7 PICTURE with a CLUT having at least two entries; the choice of colors is up to you. You pass the function a pointer to the PICTURE, the coordinates and size of the graphic (which may be zero and the size of the PICTURE), and a pointer to the compressed data. Drawing is done using the Balboa **blit_write** function. A couple of noteworthy points follow:

1. As a general programming hint, I find it most effective to hide function calls inside macros, as in the encoder listing. All the error reporting is hidden from the program itself, and I never succumb to the temptation to leave it out "just while I'm developing this piece of code."
2. Concerning error reporting, there's always a problem in identifying the cause of a crash in CD-I. Although many errors can be caught on an emulator, at some point, the prototypes have to be tested on real players without diagnostic ports. Since the sessions

Sample Program

```
/* Macros used in the encoder to improve readability */

#define ReadFile(stream,buffer,count) \
   if (fread((void *)buffer,count,1,stream)==0) \
   { Error("fread failed"); }

#define WriteFile(stream,buffer,count) \
   if (fwrite((void *)buffer,count,1,stream)==0) \
   { Error("fwrite failed"); }


/* The image encoder */

void CompressImage(fileRef, outFileRef, filePos, picW, picH)
FILE *fileRef, *outFileRef;
long filePos;
int picW, picH;
{
   int repeats, row, col, length, run, thisRun;
   unsigned char thisRow[400], lastRow[400];

   repeats=0;
   for (row=0; row<picH; filePos+=picW, row++)
   {
      memcpy(lastRow,thisRow,picW);          /* save the last row */
      fseek(fileRef,filePos,SEEK_SET);
      ReadFile(fileRef,thisRow,picW);        /* read a row */
      if (row!=0)
      {
         for (n=0; n<picW; n++) if (lastRow[n]!=thisRow[n]) break;
         if (n==picW)
         {
            repeats++;                      /* it's the same */
            if (row==picH-1 || repeats==255)
            {
               code[0]=0;                   /* "repeat last row" */
                 code[1]=repeats;            /* # of rows to repeat */
               WriteFile(outFileRef,code,2);  /* write code */
               repeats=0;
            }
            continue;                       /* go check next row */
         }
         if (repeats)
         {
            code[0]=0;                      /* "repeat last row" */
            code[1]=repeats;                 /* # of rows to repeat */
            WriteFile(outFileRef,code,2);   /* write code */
            repeats=0;                      /* reset the count */
         }
      }
      col=0;
      while (col<picW)
      {
         for (length=1; thisRow[col]==thisRow[col+length]; length++)
         {
            if (col+length>=picW) break;
         }
         if (col+length>=picW)               /* run to end of line */
         {
            code[0]=((thisRow[col]<<7)|127);
            WriteFile(outFileRef,code,1);    /* write code */
            col=picW;
         }
         else
         {
            run=length;                      /* got the run length */
            while (run>0)
            {
               thisRun=(run>126)?126:run;
               code[0]=(thisRow[col]<<7)|thisRun;
               WriteFile(outFileRef,code,1);    /* write code */
               run-=thisRun;
            }
            col+=length;
         }
      }
   }
}


/* The image decoder; part of the run-time application */

void ExpandImage(Pic, left, top, width, height, runptr)
PICTURE *Pic;
int left, top, width, height;
unsigned char *runptr;
{
   int columns, rows, col, row, length, color;
   unsigned char thisRow[400];

   columns=width>>1;
   rows=height>>1;
   row=0;
```

```
loop:
   while (row<rows)
   {
      col=0;
      while (col<columns)
      {
         if (*runptr==0)                          /* repeat last line */
         {
            for (n=runptr[1]; n!=0; row++, n--)
            {
               blit_write(thisRow,Pic,left,top+(row<<1),width,2);
            }
            runptr+=2;
            goto loop;
         }
         color=*runptr>>7;
         length=*runptr&0x7f;
         if (length==127) length=columns-col;   /* remainder of line */
         memset(&thisRow[col],color,length);
         col+=length;
         runptr++;
      }
      blit_write(thisRow,Pic,left,top+(row<<1),width,2);
      row++;
   }
}
```