Displaying Apple II Double Lo-Res Pixel Graphics in a cc65 C Program

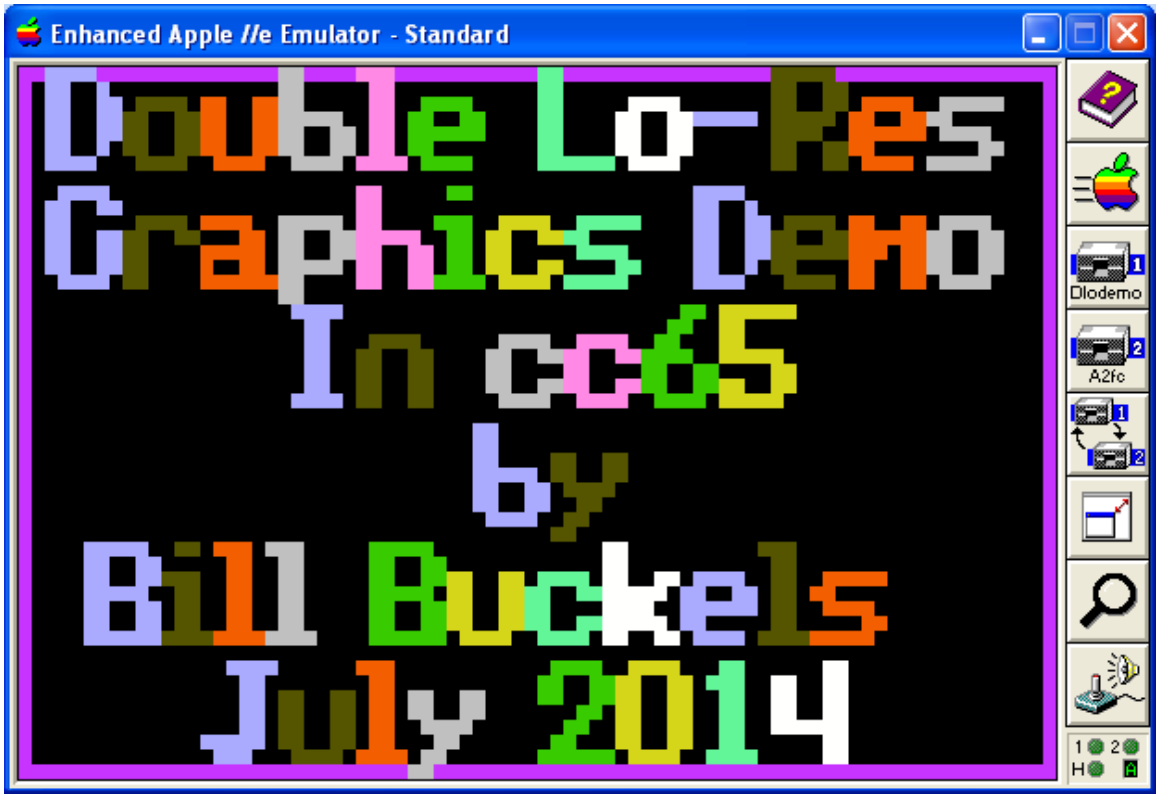


Table of Contents

Introduction

This document is about a Demo Program (called "dlodemo") written in C (and compiled using cc65) that plots Double Lo-Res (DLGR) Pixel Graphics on the Apple //e. An earlier version of dlodemo was originally distributed in 2013 as an Aztec C65 demo.

Double Resolution Graphics modes were not available until after 1983 when the second run of the Apple IIe came along, and even then, neither of these modes was directly supported by AppleSoft BASIC. Until then, only 2 graphics modes (Lo-Res and Hi-Res) could be used. Lo-Res Graphics Mode for the Apple II is such a coarse resolution that it is unsuitable for much more than a medium for very primitive graphics demos or similar programs. When DLGR came-along, doubling-up the horizontal resolution to 80 pixels x 48 rasters made it (barely) possible to display more recognizable Lo-Res type graphics.

Like AppleSoft BASIC, cc65 does not provide "built-in" support for double resolution graphics mode (through cc65's tgi drivers), but also like in AppleSoft BASIC or in 6502 Assembly code, DLGR can be used programmatically in a C program compiled using cc65. In 2010, Oliver Schmidt kindly ported some of my Lo-Res core routines from Aztec C65 to cc65, and, after some minor changes, these provide the basis for this demo.

Download dlodemo here: http://www.appleoldies.ca/cc65/programs/dlgr/dlodemo.zip

DLODEMO Program Summary

When the dlodemo program starts, a title screen is drawn then the program waits for a key press. A series of graphics primitives follows, then line drawing routines, and finally a kaleidoscope is displayed, until ESC is pressed. Then dlodemo exits to ProDOS. But "under the hood", a little more is happening.

DLGR Core Routines

More than a Demo

The dlodemo program is more than a demo; it is a prototype for the ongoing porting of Aztec C65's graphics routines, and their refinement and development for extending cc65's support for the Apple II Graphics Modes that are not currently directly supported by cc65. Since cc65 directly supports only two Apple II Graphics modes (LGR and HGR) through cc65 tgi drivers, and since cc65's direct support for Apple II bit-mapped graphics does not exist, this overall project is quite large and extensive.

The following documents discuss what I have done so far towards the above-stated goals and the accompanying demos provide cc65 source code and working disk images:

DLGR	**Bit-Mapped Graphics**
Demo	**http://www.appleoldies.ca/cc65/programs/dlgr/dloshow.zip**
Doc	**http://www.appleoldies.ca/cc65/docs/dlgr/dloshow.pdf**
DLGR	**Pixel Graphics**
Demo	**http://www.appleoldies.ca/cc65/programs/dlgr/dlodemo.zip**
Doc	**http://www.appleoldies.ca/cc65/docs/dlgr/dlodemo.pdf**
DHGR	**Bit-Mapped Graphics**
Demo	**http://www.appleoldies.ca/cc65/programs/dhgr/dhishow.zip**
Doc	**http://www.appleoldies.ca/cc65/docs/dhgr/dhishow.pdf**
DHGR	**Pixel Graphics**
Demo	**http://www.appleoldies.ca/cc65/programs/dhgr/dhiworld.zip**
Doc	**http://www.appleoldies.ca/cc65/docs/dhgr/dhiworld.pdf**

Building DLGR cc65 SYS Programs

There is nothing really special about building a DLGR ProDOS 8 SYS program for DLGR since the load address of a ProDOS SYS program is at $2000, well above the text screen display memory area which is used by DLGR (and LGR) for both graphics, and mixed-text and graphics display. It builds "out of the box". No special linker configuration is required. As in any of the demos listed above, a build environment complete with a gcc compatible MAKEFILE is provided so no need to document it here. Simply review the MAKEFILE and the other baggage that comes with dlodemo.

I should also note that these demos are provided with monolithic source code with many of the routines in header files, and do not instead link to a library other than the libraries provided with the current cc65 snapshot. This is because all of this is still under development and will not be put into any library or tgi driver until the end of this project. Since we are only at the start of this project, I will continue to use this monolithic format for distributing and improving everything and making it available as I go.

So now we get to the "crux of the biscuit"; the DLGR core routines, followed by the demo routines, followed by the dlodemo main program, so get ready for a long read.

DLGR Itself

DLGR is well documented. The ideal reader of this document should have a working knowledge of DLGR and LGR on the Apple II. While a great deal of detail about DLGR is included in this document, by the time you become interested in writing DLGR programs in cc65, I am assuming that you know how to write C programs, and you know the Apple II. This is not to say that you need to know what I know, and pretty-much any Apple II developer will likely learn something about DLGR by reading this document. I am simply saying that you shouldn't stop with my notes. Experience and additional DLGR research of your own is both required and fun! ☺

Double-Res Mode

Whether we are writing for DHGR, or DLGR, many commonalities exist since the Language Card and Auxiliary Memory are used by both of the Apple II's double-res modes. Some commonalities also exist between double-res mode on the Apple IIe and SHR (Super Hi-Res) graphics modes on the Apple IIgs (and the Apple II VOC – Apple's Video Overlay Card) in that Auxiliary Memory is also used in SHR, although somewhat differently.

Apple II Graphics in General

Whether we are writing for single-res (LGR and HGR) or double-res (DLGR or DHGR) many commonalities also exist, and many methods are re-usable, such as mixed-text and graphics routines for 80-column mode in DLGR, HGR, and DHGR. But you don't need to be a C language programmer to know this; whether you program the Apple II in BASIC, or some other language it is likely you already know a good deal about the material in this document. So the bits and pieces that are needed to write double-res graphics programs in cc65 and detailed herein can be viewed by you as "glue".

Graphics Programming in General

If you have not done much graphics programming do not fear, but keep your calculator and your Internet Search Engine both handy and active as you read along. The nasty thing about writing any graphics program is that the experience always leaves more questions asked than answered. This is also like Apple II programming, C programming, and any

programming. So the only way to get through graphics programming is to wade through it without drowning, dissecting each detail as you go, keeping in mind that no matter how humble a pixel on a display might seem, at the end of the day, it is that same pixel (or graphics image) on the display that builds the greatest of graphics applications.

And it's mostly just arithmetic and logic, and not advanced mathematics that is at the core of the greatest of graphics applications, and lots of humble pixels and work of course.

Getting Started in DLGR Pixel Graphics

We need to start with the DLGR Core Routines and "drill-down" from there. At the very heart of DLGR (and DHGR) is the interleaving of auxiliary display memory with main memory. In 16-color DHGR the pixels are packed entirely differently in memory than in 6-color HGR. But since LGR already had 16 colors to begin with, only the color index values differ between auxiliary and main memory. Double-Resolution is truly achieved in DLGR; for every LGR pixel, DLGR has two pixels; the first pixel in a pixel pair is in auxiliary memory and the second pixel in main memory. By comparison, DHGR is really not double the resolution of HGR… it is actually only half the resolution of HGR except in monochrome mode.

As far as comparing DLGR to DHGR, DLGR has no such thing as monochrome mode. DHGR is also not even double the horizontal resolution of DLGR, nor is it proportionally the same; only the colors are the same.

Auxiliary and Main Memory

The Apple II's double resolution modes, DLGR and DHGR, and the Apple II's 80-column text mode, split their screens across main and auxiliary memory.

```
#define  dhraux  ((unsigned char*)0xC055)
#define  dhrmain ((unsigned char*)0xC054)
```

Switching between auxiliary and main screen memory is as easy as writing a byte to the above soft switch addresses. These soft-switches are used throughout this demo:

```
dhraux[0] = 0;  /* select auxiliary memory */
dhrmain[0] = 0; /* reset to main memory */
```

Or alternately by storing whatever is in the accumulator to the soft switch:

```
asm("sta $c055"); /* AUX MEM */
asm("sta $c054"); /* MAIN MEM */
```

Setting Video Modes

The following functions are used to set the video modes used in this demo. Keep in mind that since cc65 uses the language card memory for other purposes in its linker configuration file, we must use cc65's routine (or an equivalent) to safely set to 80 column mode.

```
/* 80 column mode must be set before calling */
#pragma optimize (push,off)
void dloreson(void)
{
    /* page 1 double lores */
    asm("sta $c050"); /* GRAPHICS                 */
    asm("sta $c052"); /* GRAPHICS ONLY, NOT MIXED */
    asm("sta $c054"); /* PAGE ONE                 */
    asm("sta $c056"); /* LO-RES                   */
    asm("sta $c05e"); /* TURN ON DOUBLE RES       */
}
#pragma optimize (pop)
```

The above is very similar to setting double hi-res mode when using double hi-res page one. Some additional video mode routines are also used in this demo. All of these are exactly the same as the double hi-res equivalent routines:

```
/* the following two routines work with all Apple IIe graphics
modes */
#pragma optimize (push,off)
void mixedtexton(void)
{
    asm("sta $c053"); /* MIXED TEXT/GRAPHICS */
}
#pragma optimize (pop)

#pragma optimize (push,off)
void mixedtextoff(void)
{
    asm("sta  $c052"); /* GRAPHICS ONLY, NOT MIXED */
}
#pragma optimize (pop)

/* 80 column mode must be set to on after calling */
#pragma optimize (push,off)
void dloresoff(void)
{
    asm("sta $c051"); /* TEXT - HIDE GRAPHICS */
    asm("sta $c05f"); /* TURN OFF DOUBLE RES  */
    asm("sta $c054"); /* PAGE ONE             */
}
#pragma optimize (pop)
```

Lo-Res "Screen Holes" and Screen Memory

The Lo-Res and Double Lo-Res graphics modes use text screen memory which is interleaved and not linear (and not contiguous in DLGR), with small blocks of 8 bytes which are actively used by motherboard firmware and expansion slot devices, including disk controller firmware to store important information. These forbidden areas must be left alone in main memory.

Let's take a look at some examples that show how the dlodemo program uses text screen base address tables to avoid these problem areas when clearing the DLGR graphics screen, or clearing the bottom 4 lines of the text screen when using mixed-text and graphics mode…

```
/* clear the bottom 4 lines of the text screen in mixed mode
double res */
void dloclear_bottom(void)
{
      char *crt;
      int row, col;
      char c = 32 + 128;

      for (row = 0; row < 4; row++) {
        crt = (char *)(dlotextbase[row]);
       for (col = 0; col < 40; col++) {
              dhraux[0] = 0; /* select auxiliary memory */
              crt[col] = c;
              dhrmain[0] = 0; /* reset to main memory */
              crt[col] = c;
        }
    }
}

#pragma optimize (push,off)
void dloresclear()
{
    int idx;
    for (idx = 0; idx < 24; idx++) {
        asm("sta  $c055"); /* AUX MEM */
        memset ((char *)textbase[idx], 0, 40);
        asm("sta  $c054"); /* MAIN MEM */
        memset ((char *)textbase[idx], 0, 40);
    }

}
#pragma optimize (pop)
```

The screen holes are avoided the same way when clearing the DLGR screen to a color…

```
void dloresflood(unsigned char color)
{
    int idx;
    unsigned char maincolor, auxcolor;

    maincolor = (color << 4 | color);
    color = dloauxcolor[color];
    auxcolor = (color << 4 | color);

    for (idx = 0; idx < 24; idx++) {
        dhraux[0] = 0; /* select auxiliary memory */
        memset ((char *)textbase[idx], auxcolor, 40);
        dhrmain[0] = 0; /* reset to main memory */
        memset ((char *)textbase[idx], maincolor, 40);
    }

}
```

Base Address Tables

In the code above, base address tables are used to reference the starting memory of scan-lines and text rows.

There has been considerable discussion in the csa2 newsgroup about the pros and cons of using base address tables. For the average programmer a table of scan-line addresses is the most readable and logical method to reference a scan-line origin, just as indexable tables in general lend themselves to more readable and manageable C code.

.

Text Screen Base Address Table

Part of this demo used the Apple II's Mixed Text and Graphics feature to print directly to the bottom 4 lines of text screen memory in 80 column mode. The routines that do so are supported by an 8 byte table with the starting address of each of these 4 lines:

```
/* base addresses for last 4 lines of text screen memory page 1
for mixed text and graphics mode */
unsigned dlotextbase[4]={
    0x0650,
    0x06D0,
    0x0750,
    0x07D0};
```

The above table can be used in any of the Apple II and Apple IIe Mixed Text and Graphics modes, whether in 40 column or 80 column mode.

DLGR Graphics Screen Base Address Table

```
/* base addresses for text screen memory page 1 */
/* also the base addresses for the 48 scanline pairs for lores
graphics mode 40 x 48 x 16 colors */
unsigned textbase[24]={
    0x0400,
    0x0480,
    0x0500,
    0x0580,
    0x0600,
    0x0680,
    0x0700,
    0x0780,
    0x0428,
    0x04A8,
    0x0528,
    0x05A8,
    0x0628,
    0x06A8,
    0x0728,
    0x07A8,
    0x0450,
    0x04D0,
    0x0550,
    0x05D0,
    0x0650,
    0x06D0,
    0x0750,
    0x07D0};
```

DLGR Colors

Remapping Main Colors to Auxiliary Colors

In the **dloresflood()** code shown previously, you can see that the colors are different for auxiliary memory than they are for main memory. To facilitate remapping auxiliary colors a small table is used:

```
/* bank 0 color remapping to bank 1 color */
unsigned char dloauxcolor[16] = {
     0,8,1,9,2,10,3,11,4,12,5,13,6,14,7,15};
```

Using this table avoids the right rotate no carry (right circular shift) of the 4-bit DLGR color number that would otherwise be needed. Like scan-line base address tables (also mentioned above) the goal here is readability. You can also see that even though the DLGR display has a resolution of 80 x 48, in the **dloresflood()** code above only 24 lines are set to a color.

16 Colors – 4 Bits Per Pixel

```
/* lo-res colors and color order */
#define LOBLACK     0
#define LORED       1
#define LODKBLUE    2
#define LOPURPLE    3
#define LODKGREEN   4
#define LOGRAY      5
#define LOMEDBLUE   6
#define LOLTBLUE    7
#define LOBROWN     8
#define LOORANGE    9
#define LOGREY      10
#define LOPINK      11
#define LOLTGREEN   12
#define LOYELLOW    13
#define LOAQUA      14
#define LOWHITE     15
```

There are 16 – Colors on the Apple II DLGR screen. These are the same 16 colors as LGR and DHGR. The color values shown above are used thoughout the program for the color argument for all the functions. The **`dloauxcolor[16]`** table discussed previously uses these color values as an index map.

As shown above, each of DLGR's 16 colors has its own 4-bit value which plots alternately across Auxiliary and Main Memory (Banks 1 and 0) in increments of 1 pixel, with even pixels in bank 1 and odd pixels in bank 0.

2 Vertical Pixels Per Byte

Two lines of color plot on the DLGR display at the same time when we put a byte into screen memory. This because 2 colors (4-bit pixels) fit into one byte and you can't "split" a byte across 2 different addresses like bank0 and bank1, both because a byte is the lowest common denominator of memory address storage, and, due to "bank-switching" on the Apple II, each even or odd byte shares the same address. Also the DLGR memory address is the same as the 80 column text screen which of course also inherently works in bytes, and must work co-operatively with DLGR and vice-versa.

Screen Resolution – 80 x 48 (Full Graphics) or 80 x 40 (Mixed Mode)

The amount of "space" for pixel pairs (bytes) on DLGR is the storage size of an 80 column text screen… each byte is 2 colors (pixels) high and each line is 80 colors (columns, pixels) across.

This provides DLGR with a vertical resolution of 48 "pixel lines" in height in "full graphics" mode and 40 "pixel lines" in "mixed text and graphics" mode. Unlike in the

HGR display memory which can hold “full screen” graphics while we merrily switch “mixed” and “full” on and off. This is due to the “split personality” of the DLGR display memory (because the text and graphics in DLGR (and LGR) share exactly the same memory), so graphics in LO-RES “mixed” modes (DLGR and LGR) are a maximum of only 40 “pixel lines” (20 bytes) in height.

Mixed Text and Graphics

In the discussion of “Lo-Res” Screen Holes earlier in this document the **`dloclear_bottom ()`** function was already listed:

```
/* clear the bottom 4 lines of the text screen in mixed mode
double res */
void dloclear_bottom(void)
{
      char *crt, c = 32 + 128;
      int row, col;
      for (row = 0; row < 4; row++) {
        crt = (char *)(dlotextbase[row]);
       for (col = 0; col < 40; col++) {
              dhraux[0] = 0; /* select auxiliary memory */
              crt[col] = c;
              dhrmain[0] = 0; /* reset to main memory */
              crt[col] = c;
        }
```

```
    }
}
```

Dlodemo uses a second function to print rows of text to the bottom of the DLGR screen when in mixed-mode (The idea here is to go right to text screen memory in 80 column mode. Remember to use cc65 to set to 80 column mode.):

```
/* print string directly to text screen memory in mixed mode
double res */
/* row settings = 0,0 to 3,79 in mixed text and graphics mode */
void dloprint_bottom(char *str,int row,int col)
{
     char *crt, c;
     int x, aux = 1, idx, jdx = 0;

     x = col / 2;
     if (col % 2) aux = 0;

     crt = (char *)(dlotextbase[row]+x);
     idx = 0;
     for (;;) {
       c = str[idx]; idx++;
       if (c == 0)break;
       c+=128;
       if (aux == 1) {
         dhraux[0] = 0; /* select auxiliary memory */
         crt[jdx] = c;
         aux = 0;
       }
       else {
         dhrmain[0] = 0; /* reset to main memory */
         crt[jdx] = c;
         jdx++;
         aux = 1;
       }
   }
   /* safety play */
   dhrmain[0] = 0; /* reset to main memory */
}
```

The above is a simple function and does not print mouse-text characters or reverse video.

Font Routines

The font routines in this demo come in two variations:

- The “Tom Thumb” Font
- The Extended Aztec C65 Font

The dlodemo Title Screen uses the Extended Aztec C65 Font.

The Extended Aztec C65 Bitmapped Font

This font is really a monochrome font based on an Apple II bit-mapped font for HGR mode and also looks fine on the DHGR monochrome display where it plots quickly in mono to 80 columns. In this demo it is used as a bitwise DLGR “pixel map”:

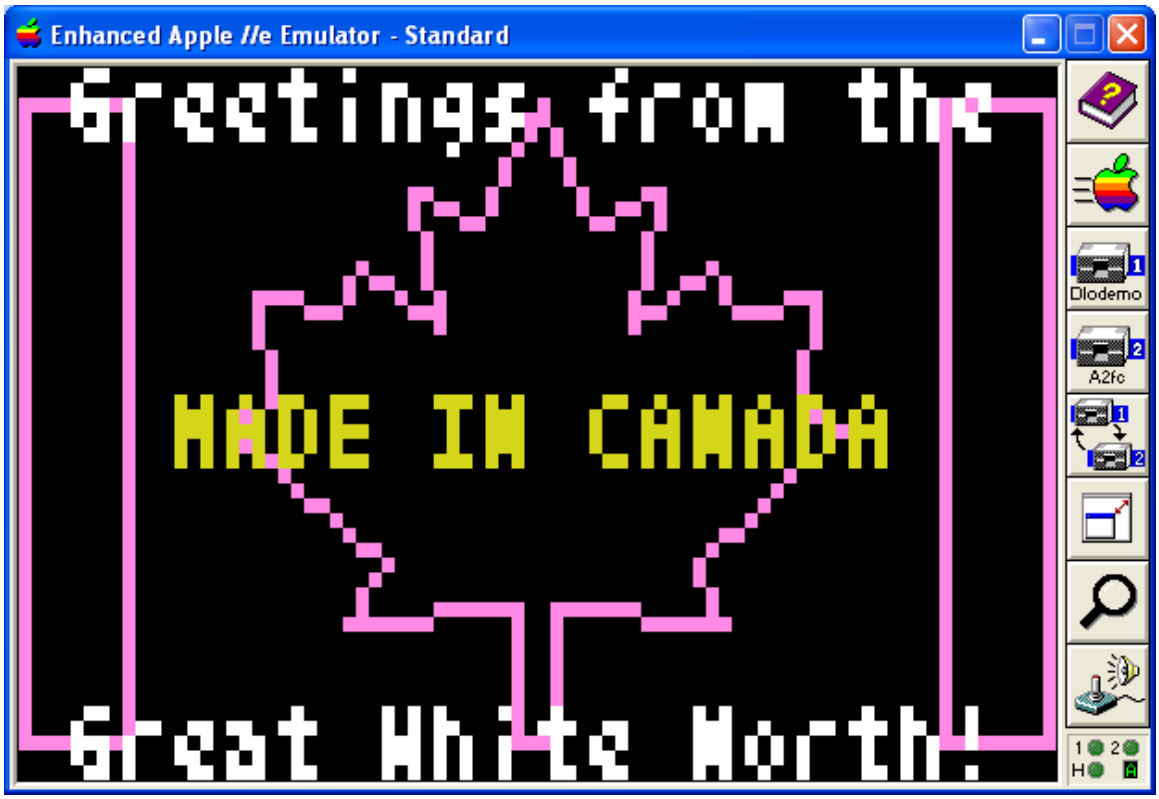
```
/* Apple II 7 x 8 bitmapped font - can be used as-is in HGR mode
or as a character map for plotting a pixel-graphics font. This is
an extended and modified version of the __chr[760] array
originally shipped with the graphics library for the Aztec C65
CII compiler for DOS 3.3 from Manx Software Systems, 1983. I
modified and extended it in 1989-1990 in its present form (below)
to contain Ascii Values from 32-168 which includes multilingual
characters and symbols based on the IBM-PC extended character set
for MS-DOS. I further extended it to include all the IBM extended
characters in my AWINDLO utility in 2013. */

unsigned char __chr[1096] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x00, 0x0C, 0x00,
...
0x3C, 0x66, 0x3C, 0x00, 0x7E, 0x00, 0x00, 0x00,
0x18, 0x00, 0x18, 0x0C, 0x06, 0x66, 0x3C, 0x00};

int msk7[]={0x1,0x2,0x4,0x8,0x10,0x20,0x40};

/* Double Lo-Res 7 x 8 x 16 color font routine */
/* uses scanline by scanline plotting rather than character by
character plotting */
void dlofont(char *str,int row,int col,int fg,int bg,int scale)
{
  int target, scanline, offset, r, r2, c, d, byte, nibble, x,
color;
  unsigned char ch;

  if (scale > 2)scale = 1;
  target = strlen(str);
  for(scanline=0;scanline<8;scanline++)
  {
```

```
    /* set values for vertical term */
    /* expand x scale in the vertical direction */
    r = (scanline * scale) + row; /* max 16 high */
    if (r > 47)break;
    r2 = r + 1;
    /* run the string 8 times
       if scale == 2 then print a double line
       each time which gives us a font of 16 high */
    for (byte=0;byte<target;byte++)
     {
      /* calculate the starting column for each run
         in the width of 7 pixels */
      c = (byte * 7) + col;
      if (c > 79)continue;
      d = str[byte]&0x7f;
      if (d < 32)d = 32;
      if (d == 32 && bg < 0)continue;
      offset = ((d-32) * 8) + scanline;
      ch = __chr[offset];
      for (nibble=0;nibble<7;nibble++)
       {
        x = c+nibble;
        if (x > 79)break;
        if (ch & msk7[nibble]){
          color = fg;
        }
        else {
          if (bg < 0)continue;
          color = bg;
        }
        dloplot((unsigned char)x,(unsigned char)r,
                (unsigned char)color);
        if (scale > 1)dloplot((unsigned char)x,
                      (unsigned char)r2,(unsigned char)color);

       }
    }
  }
}
```

As you can see, this font routine provides the following features:

- double scaling in the vertical direction
- background color
- foreground color

The "Tom Thumb" Font



In this demo, I am introducing the "Tom Thumb" Font to the Apple II. The "Tom Thumb" Font (shown above and below) was originally developed as a Palm Pilot font by developer Brian Swetland, and "fine-tuned" as a derivative work by Robey Pointer, robeypointer@gmail.com :

Brian Swetland's Copyright and Conditions of Use for the original font are in the dlodemo source code header file "tomthumb.h". If you decide to use Robey's "Tom Thumb" Font in your own programs, you must leave Brian's Copyright and Conditions in the source code. More can be read about this font at the following link:

http://robey.lag.net/2010/01/23/tiny-monospace-font.html

Mentioning the use of the "Tom Thumb" Font herein is neither an endorsement, nor a promotion by either of these two individuals; merely an attribution to them. The implementation of this font in the dlodemo was entirely hand-built by me in my

programmer's editor by simply looking at the image above and counting pixels. The same method of building a fixed pitch font can easily be done for any font.

This 4 x 6 fixed-pitch font provides 96 reasonably legible ASCII characters and can be pixel-mapped to provide the various displays for the Apple II with a color (or Monochrome) font:

Display Mode Color	**Columns**	**Rows**	**Characters**
LGR	10	8	80
DLGR	20	8	160
HGR	35	32	1120
DHGR	35	32	1120
SHR 320	80	33	2640
SHR 640	160	33	5280
Monochrome			
HGR	70	32	2240
DHGR	140	32	4480

For the Apple II's higher resolution modes display modes a tiny 4 x 6 font would be a novelty more than anything else, or might not appear proportionate. For the lower resolutions, other alternatives like Mixed-Text and Graphics mode are also available, especially for the plain Lo-Res (LGR) display. For HGR Color Mode or HGR and DHGR Monochrome Modes a Monochrome Font might be just as good for your purposes or alternately the "Tom Thumb" Font might be more effective.

But for DLGR, the "Tom Thumb" Font provides a reasonable mechanism for rendering short text strings on the graphics display:

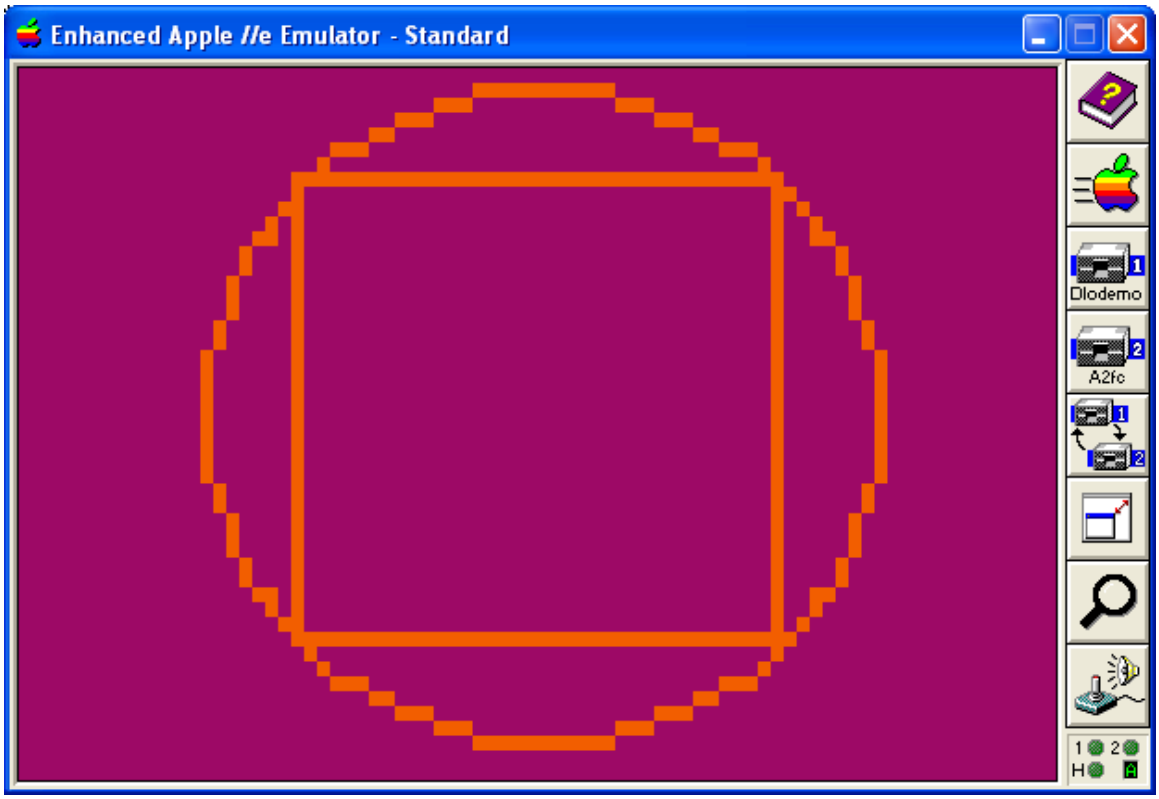
```
unsigned char tomthumb[1728] = {…};

/* mono-spaced "tom thumb" 4 x 6 font */
/* using a byte map to gain a little speed at the expense of
memory */
/* a bitmap could have been encoded into nibbles of 3 bytes per
character rather than the 18 bytes per character that I am using
but the trade-off in the speed in unmasking would have slowed
this down */
void plotthumb(unsigned char ch,
               unsigned char x, unsigned char y,
               unsigned char fg, unsigned char bg)
{
    unsigned offset;
    unsigned char x1, x2=x+3, y2=y+6, byte;

    if (ch < 33 || ch > 127) ch = 0;
    else ch -=32;
    if (ch == 0 && bg > 15) return;
```

```
    /* each of the 96 characters is encoded into 18 bytes */
    offset = (18 * ch);
    while (y < y2) {
        for (x1 = x; x1 < x2; x1++) {
           if (x1 > 79) {
               offset++;
               continue;
           }
           byte = tomthumb[offset++];
           if (byte == 0) {
               if (bg > 15) continue;
               dloplot(x1,y,bg);
           }
           else {
               if (fg > 15) continue;
               dloplot(x1,y,fg);
           }
        }
        /* if background color is being used then
           a trailing pixel is required
           between characters */
        if (bg < 16 && x2 < 79)dloplot(x2,y,bg);
        if (y++ > 47) break;
    }
}

/* normally spaced 4 x 6 font */
/* using character plotting function plotthumb() (above) */
void dlothumb(char *str,unsigned char x, unsigned char y,
              unsigned char fg,unsigned char bg,
              unsigned char justify)
{
  int target;
  unsigned char ch;

  if (justify == 'M' || justify == 'm') {
     target = strlen(str);
     x-= ((target * 4) /2);
  }
  while ((ch = *str++) != 0) {
     plotthumb(ch,x,y,fg,bg);
     x+=4;
  }
}
```

As you can see, this font routine provides the following features:

- left or middle justification of horizontal datum point
- background color
- foreground color

Graphics Primitives in DLGR



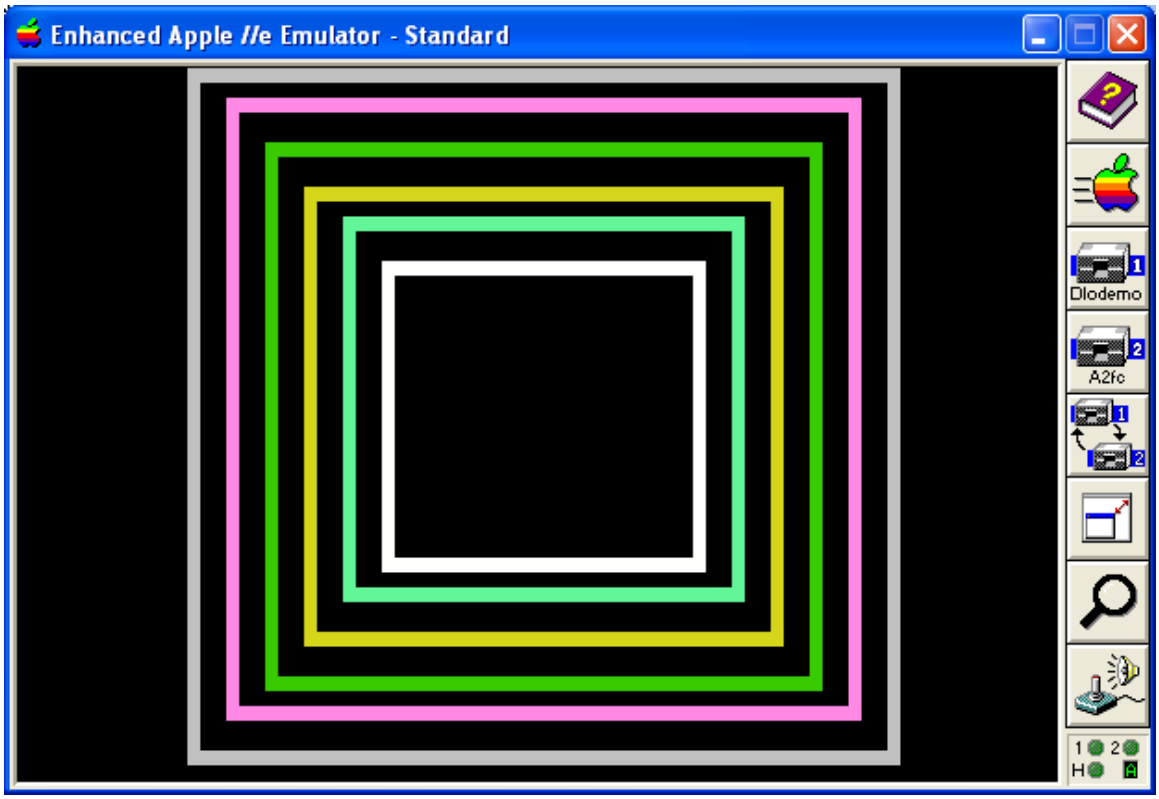
Does Pixel Size Matter?

Calculation of pixel graphics co-ordinates in any graphics display must always be as precise as possible. DLGR is a very coarse and "jaggy" resolution. You might think that because DLGR is coarse, that the appearance of a pixel that is out of position by a single digit might not matter, but a graphics primitive will appear out of balance even more so when pixels are large, than in a higher resolution mode like HGR which has smaller and more forgiving pixels.

Calculating Pixel Position

Proportion and Aspect Ratio

For proper appearance, especially when drawing a centric primitive like a circle or an equilateral triangle, screen resolution must be considered, and pixel placement must rotate proportionally and evenly around the axis point. As in any graphics display, an aspect ratio adjustment is required to make these primitives appear proportional on the Apple II DLGR display. Other primitives like squares and polygons must also observe the same proportional consideration, as must all other graphics, like characters in font sets, and bit-mapped graphics images and "pseudo-sprites" (image fragments).



To achieve proportional appearance, this demo uses a DLGR aspect ratio adjustment of 1.16 rounded down to the nearest integer to compress the vertical aspect from 48 logical scan-lines to 41 physical scan-lines when it calculates pixel plotting co-ordinates.

Precision Constraints

1. The Apple II display, like other pixel graphics displays, uses 2 integers to describe the horizontal and vertical co-ordinates to plot pixels, so eventually any real numbers must be rounded to integer co-ordinates for display, with some inherent precision loss.

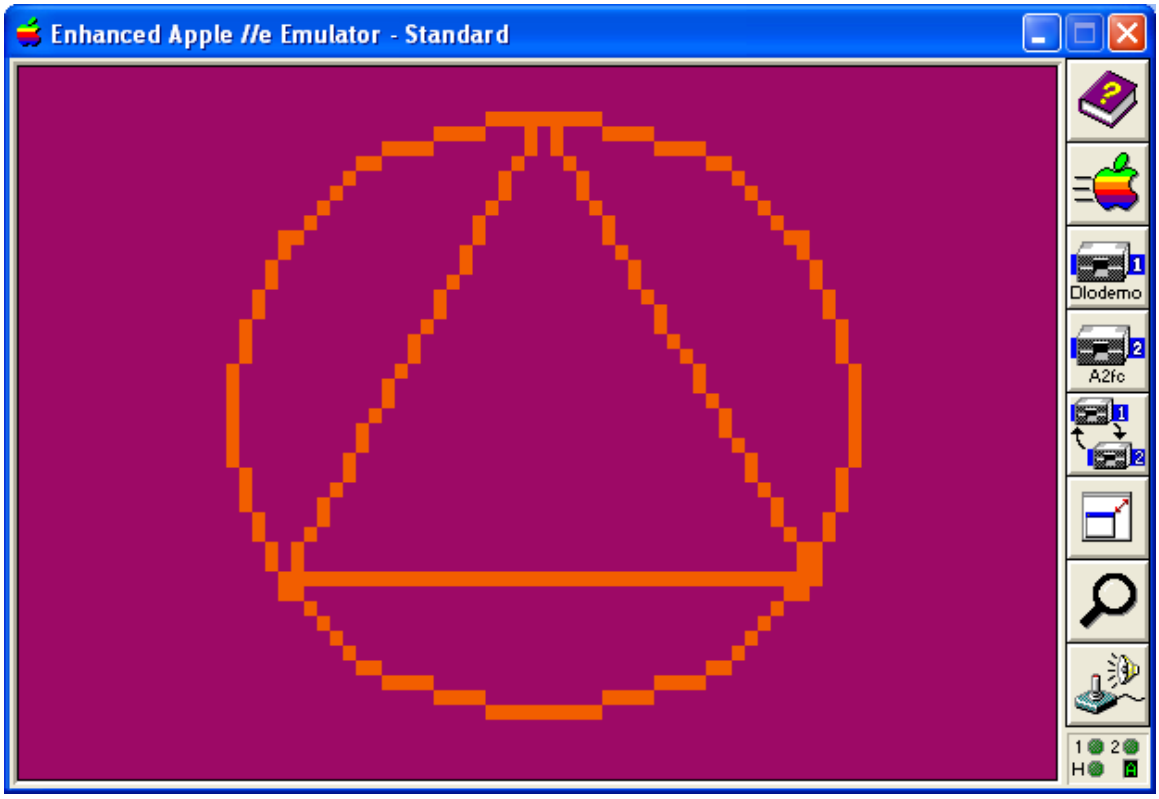
2. For calculations involving algebra, integer math is only a (usually) close approximation. Since the Apple II does not provide a floating point processor and the cc65 compiler does not provide "soft" floating point calculations, integer math must be used to calculate the shape co-ordinates of any primitive object like a circle or a triangle, resulting in precision loss from rounding.

3. Adjusting co-ordinates for DLGR's aspect ratio to achieve proportional appearance also adds some additional precision loss from rounding. To achieve the best possible results, aspect adjustment should be applied to final calculated co-ordinate values that are as precise as possible.

So consequently, the dlodemo program's graphics primitives are hardly perfect, but dlodemo provides some simple and reasonably effective techniques for using integers

instead of real numbers to draw graphics primitives on the DLGR display. These could be fine tuned with rounding error adjustments and all the rest of it, but since I am hurrying hard I simply didn't bother to take the time to take this further. The Internet is rich with C language source code for doing vector graphics and it is easily adapted to writing cc65 programs. So with all due respect, and with this working well enough, additions and refinements have been left as an optional exercise for the reader.

Achieving a Reasonable Compromise



Consider the primitive shape shown above. The equilateral triangle balances from side to side; all the pixels match from side to side. The circle is also balanced and the pixels match in all 8 segments. But the right edge of the triangle is 1 pixel closer to the right edge of the circle than the left edge.

While I have taken very little time to provide much more than some "lame" primitives for DLGR simply because the whole point of this demo is to provide cc65 with support for DLGR and not to impress you with elegant pixel graphics.

At the same time I won't avoid the discussion altogether, so let's talk a little more about the triangle in the circle shown above; here's the code that makes it happen:

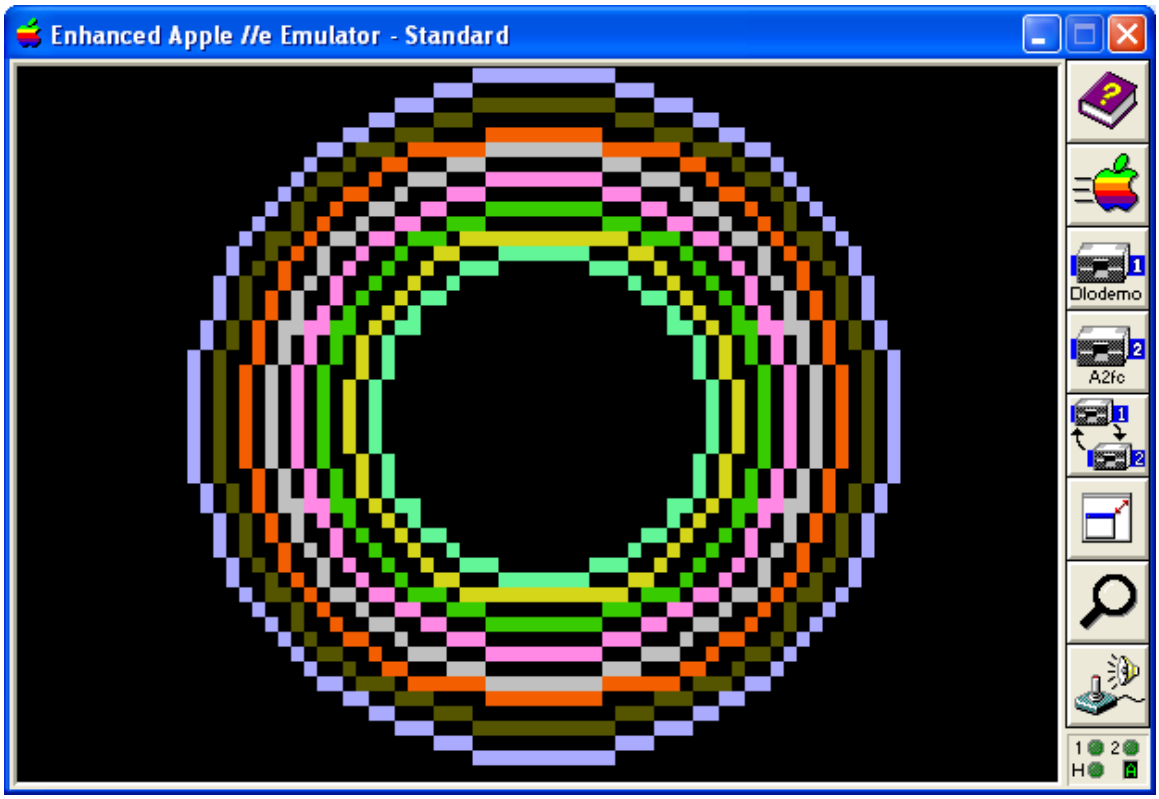
```
void dlotricircle(int x,int y,int base,unsigned char color)
{
    int x1 = x - (base/2);
    int x2 = x1 + base;
    int height = (base * 100) / 130;
    int y1 = y - (height/2);
    int y2 = y1 + height;

    int cy, a, b, r;

    dloline(x2,y2,x,y1,color);
    dloline(x,y1,x1,y2,color);
    dloline(x1,y2,x2,y2,color);

    cy = y1 + (height * 2) /3;
    a = (base / 2);
    b = (base / 3);
    r = ihypot(a,b) + 1;

    dlocircle(x,cy,r,color);

}
```

The code above is an aggregate graphics primitive that I "dreamed-up" based on the circumscribed circle of an equilateral triangle. Wikipedia has a good article on equilateral triangles, so if you are interested in the math behind what I have done here, visit the following link:

http://en.wikipedia.org/wiki/Equilateral_triangle

In the code above, I am also doing some proportional scaling based on DLGR's 1.16 vertical aspect ratio adjustment of 48 to 41 scan-lines and some integer math, as previously discussed.

Drawing Circles in DLGR



So let's also take a look at the DLGR circle routine that I have provided:

```
/* Aztec C65 circle algorithm, taken from the HGR mode Aztec C65
3.2b version, and originally written circa 1982-83, then modified
for DLGR */
void dlocircle(int cx,int cy,int r, unsigned char color)
{
    int x,y,a,b,c,d,f;

    x=r; y=0; b=1; f=0;
    a=(-2)*x+1;

point:

    c=   (int) x;
    d=   (int) y;

    /* 1.16 aspect ratio reduction in y axis for DLGR */
    dloplot((unsigned char)c+cx,
(unsigned char)((y*100)/116)+cy, color);
    dloplot((unsigned char)d+cx,
(unsigned char)((x*100)/116)+cy, color);
    dloplot((unsigned char)-d+cx,
```

```
(unsigned char)((x*100)/116)+cy, color);
    dloplot((unsigned char)-c+cx,
(unsigned char)((y*100)/116)+cy, color);
    dloplot((unsigned char)-c+cx,
(unsigned char)((-y*100)/116)+cy,color);
    dloplot((unsigned char)-d+cx,
(unsigned char)((-x*100)/116)+cy,color);
    dloplot((unsigned char)d+cx,
(unsigned char)((-x*100)/116)+cy,color);
    dloplot((unsigned char)c+cx,
(unsigned char)((-y*100)/116)+cy,color);

    if(b>= -a)
        goto fin;

    y+=1; f+=b;
    b+=2;

    if(f>r)
    {
        f+=a;
        a+=2;
        x-=1;
    }
    goto point;

fin:;

}
```

In the code above and as previously discussed, DLGR's horizontal aspect ratio is 1.16 of the vertical, so to compensate and make the circle appear round instead of elongated, the y axis (vertical axis) is compressed by 86%, from 48 to 41 scan-lines, during the plotting process.

Integer Algebra

You should also note in the **`dlotricircle()`** code shown above that a function called **`ihypot()`** is called to provide the radius to draw the circle:

```
/* based on Integer Square Roots by Jack W. Crenshaw
http://www.embedded.com/electronics-blogs/programmer-s-
toolbox/4219659/Integer-Square-Roots

copied from (with minor changes)
http://www.codecodex.com/wiki/Calculate_an_integer_square_root */
```

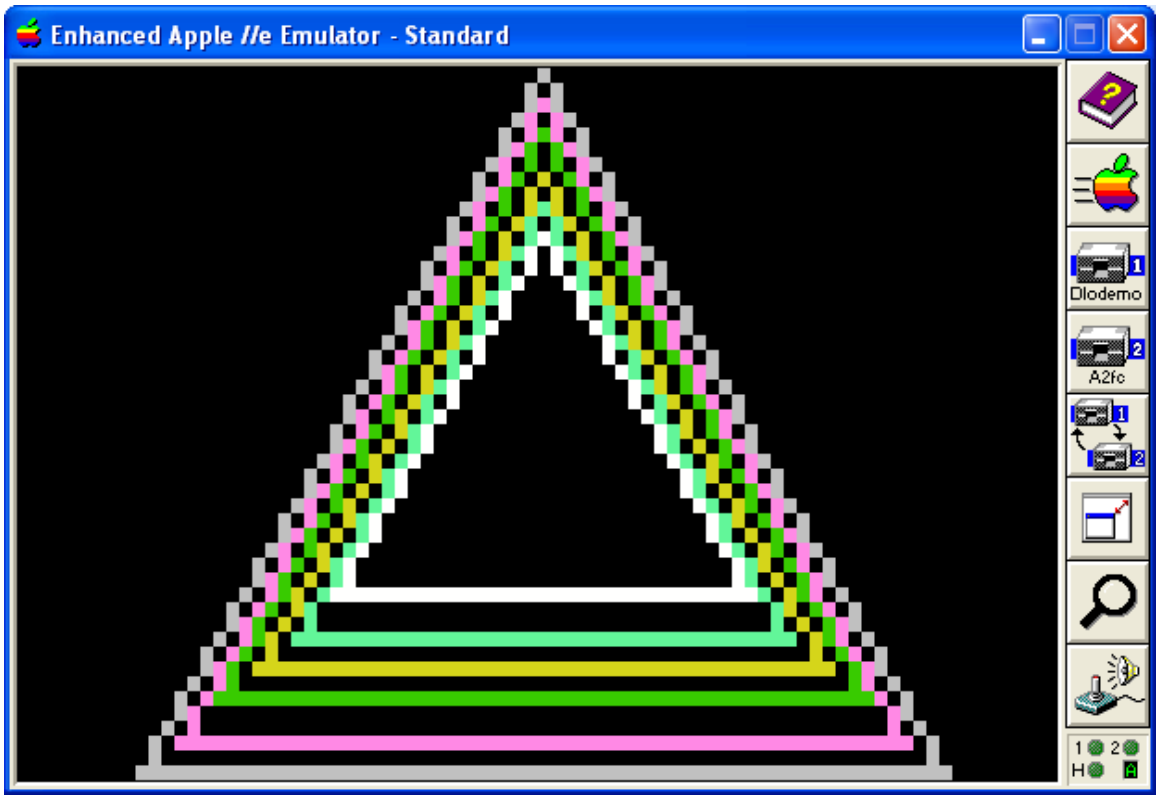
```
unsigned isqrt(unsigned n)
{

    unsigned root = 0, remainder = n, place = 0x4000;

    while (place > remainder)
        place = place >> 2;

    while (place)
    {
        if (remainder >= root + place)
        {
            remainder = remainder - root - place;
            root = root + (place << 1);
        }
        root = root >> 1;
        place = place >> 2;
    }
    return root;
}

/* returns the integer value of the hypoteneuse of a right angle
triangle. may require a rounding area adjustment of an additional
pixel */
unsigned ihypot(unsigned x, unsigned y)
{
    return isqrt ((x * x) + (y *y));
}
```

Those of you who know Jack Crenshaw also likely know that he programmed the Apollo Space Mission and is again programming the trajectory analysis with a Google Ranger Team. Jack has been on this planet a long time… around 80 years or so, and knows a lot about programming, and the code he has cut from that time to this is pretty darned stable whether for landing on the moon or more common and less critical purposes, so for the purposes of this demo, and following Crenshaw's first law (KISS), I kept it simple, Simon, and just used Jack's code instead of literally re-inventing the wheel.

As previously noted, I'll leave it to the reader to explore this particular piece of the demo a little further. Suffice to say that apparently everything seems to work as expected when it comes to calculating the correct integer value for the radius of the bounding circle for the circumscribed equilateral triangle from the hypotenuse algorithm made possible by Crenshaw's integer square root function.

Crenshaw has other integer-based routines rattling around the Internet which may also be adaptable to your particular needs, whether for DLGR or just any old integer math.

Drawing Equilateral Triangles in DLGR



As far as the triangle itself goes, the code below used in this demo pretty-much sums-it up:

```
void dloequilateral(int x,int y,int base,unsigned char color)
{
    int x1 = x - (base/2);
    int x2 = x1 + base;
    int height = (base * 100) / 130;
    int y1 = y - (height/2);
    int y2 = y1 + height;

    dloline(x2,y2,x,y1,color);
    dloline(x,y1,x1,y2,color);
    dloline(x1,y2,x2,y2,color);
}
```

Drawing Lines in DLGR

You can see in the code above that the **`dloline()`** function is called. This is an "almost standard" line drawing routine:

```
/* Bresenham Algorithm line drawing routine for double lo-res */
void dloline(int x1, int y1, int x2, int y2, unsigned char color)
{

    int dx, dy, sx, sy, err, err2;

    /* single pixel */
    if (x1 == x2 && y1 == y2) {
        dloplot((unsigned char)x1,(unsigned char)y1,color);
        return;
    }

    /* vertical line */
    if (x1 == x2) {
        if (y1 < y2) dlovline((unsigned char)x1,
           (unsigned char)y1, (unsigned char)y2, color);
        else dlovline((unsigned char)x1,
           (unsigned char)y2, (unsigned char)y1, color);
        return;

    }

    /* horizontal line */
    if (y1 == y2) {
        if (x1 < x2) dlohline((unsigned char) y1,
           (unsigned char)x1, (unsigned char)x2, color);
        else dlohline((unsigned char) y1,
           (unsigned char)x2, (unsigned char)x1, color);
        return;
    }
    /* vector */
    if(x1 < x2) {
        dx = x2 - x1;
        sx = 1;
    }
    else {
        sx = -1;
        dx = x1 - x2;
    }
    if(y1 < y2) {
        sy = 1;
        dy = y2 - y1;
    }
    else {
        sy = -1;
```

```
        dy = y1 - y2;
    }
    err = dx-dy;
    for (;;) {
        dloplot((unsigned char)x1,(unsigned char)y1,color);
        if(x1 == x2 && y1 == y2)break;
        err2 = err*2;
        if(err2 > (0-dy)) {
             err = err - dy;
             x1 = x1 + sx;
        }
        if(err2 <  dx) {
             err = err + dx;
             y1 = y1 + sy;
        }
    }
}
```

Note that I said this line drawing routine is "almost standard"… at the beginning of the line drawing routine, if the line is a horizontal line, I call the **`dlohline()`** function and if the line is a vertical line, I call the **`dlovline()`** function, and for everything else, the **`dloplot()`** function is called. Let's take a look at those functions now:

Plotting a Pixel in DLGR

```
#pragma optimize (push,off)
void dloplot(unsigned char x, unsigned char y,
             unsigned char color)
{
    unsigned char w, z = x;
/* Double Lo-Res works the same way 80-column text does:
   columns 0, 2, 4, ...78 are stored in Auxiliary Memory,
   and columns 1, 3, 5, ...79 are stored in Main Memory. */
   x = z / 2;
   w = x * 2;
   if (z==w)  {
       setlocolor(dloauxcolor[color]);
       dhraux[0] = 0; /* select auxiliary memory */
   }
   else {
       setlocolor(color);
       dhrmain[0] = 0; /* select main memory */
   }

  asm("LDY #%o", y);
  asm("LDA (sp),Y");
  asm("PHA");

  asm("LDY #%o", x);
  asm("LDA (sp),Y");
  asm("TAY");  // Lo-Res Plot X (Horizontal) Coordinate (0-39)
```

```
  asm("PLA");  // Lo-Res Plot Y (Vertical) Coordinate (0-39)
  SWITCH_ROM;
  asm("JSR $F800");
  SWITCH_LC2;
  /* safety play */
  if (z==w) dhrmain[0] = 0; /* reset to main memory */
}
#pragma optimize (pop)
```

Setting the DLGR Pixel Color

In the code above a call is made to the **setlocolor()** function. Whether we are using DLGR or LGR, this same call is used. But with DLGR (as previously mentioned) we need to remap the colors to a different value for auxiliary memory (as shown above).

```
#pragma optimize (push,off)
void setlocolor(unsigned char value)
{
  asm("LDY #%o", value);
  asm("LDA (sp),Y");// Sets the plotting color to N, 0 <= N <= 15

  SWITCH_ROM;
  asm("JSR $F864");
  SWITCH_LC2;
}
#pragma optimize (pop)
```

The code above is from Oliver Schmidt, and was provided as part of a csa2 thread from a few years back. It is a cc65 equivalent of my Aztec C65 LGR routines.

Drawing Horizontal Lines

The following code draws a horizontal line in DLGR mode using the **dloplot()** function:

```
void dlohline(unsigned char y, unsigned char x1,
              unsigned char x2, unsigned char color)
{
    unsigned char x;
    /* swap horizontal co-ordinates if out of order */
    if (x1 > x2) {
       x = x2;
       x2 = x1;
       x1 = x;
    }
    x2++;
    for (x = x1; x < x2; x++) dloplot(x, y, color);
}
```

Drawing Vertical Lines

The following code draws a vertical line in DLGR mode:

```
/* Oliver's lovlin routine - used by dlovline */
#pragma optimize (push,off)
void lovlin(unsigned char x, unsigned char y1, unsigned char y2)
{
  /* Bottom Y Coordinate (0-47) */
  /* Store it at V2 Lo-res line end-point */
  *(unsigned char *)0x2d = y2;

  asm("LDY #%o", y1);
  asm("LDA (sp),Y");
  asm("PHA");

  asm("LDY #%o", x);
  asm("LDA (sp),Y");
  asm("TAY");  // X Coordinate (0-39)

  asm("PLA");  // Top Y Coordinate (0-47)
  SWITCH_ROM;
  asm("JSR $F828");
  SWITCH_LC2;
}
#pragma optimize (pop)

/* dlovline calls Oliver's lovlin routine above */
void dlovline(unsigned char x, unsigned char y1,
              unsigned char y2, unsigned char color)
{
    unsigned char y, z=x;

    /* swap co-ordinates if out of order */
    if (y1 > y2) {
       y = y2;
       y2 = y1;
       y1 = y;
    }

/* Double Lo-Res works the same way 80-column text does:
   columns 0, 2, 4, ...78 are stored in Auxiliary Memory,
   and columns 1, 3, 5, ...79 are stored in Main Memory. */

   x = z/2;
   y = x * 2;

   if (y==z) {
       setlocolor(dloauxcolor[color]);
       dhraux[0] = 0; /* select auxiliary memory */
   }
```

```
    else {
        setlocolor(color);
        dhrmain[0] = 0; /* select main memory */
    }

    /* just calling Oliver's routine directly */
    lovlin(x,y1,y2);
    if (y==z) dhrmain[0] = 0; /* reset main memory */

}
```

Plotting a Pixel in DLGR – Alternate Method

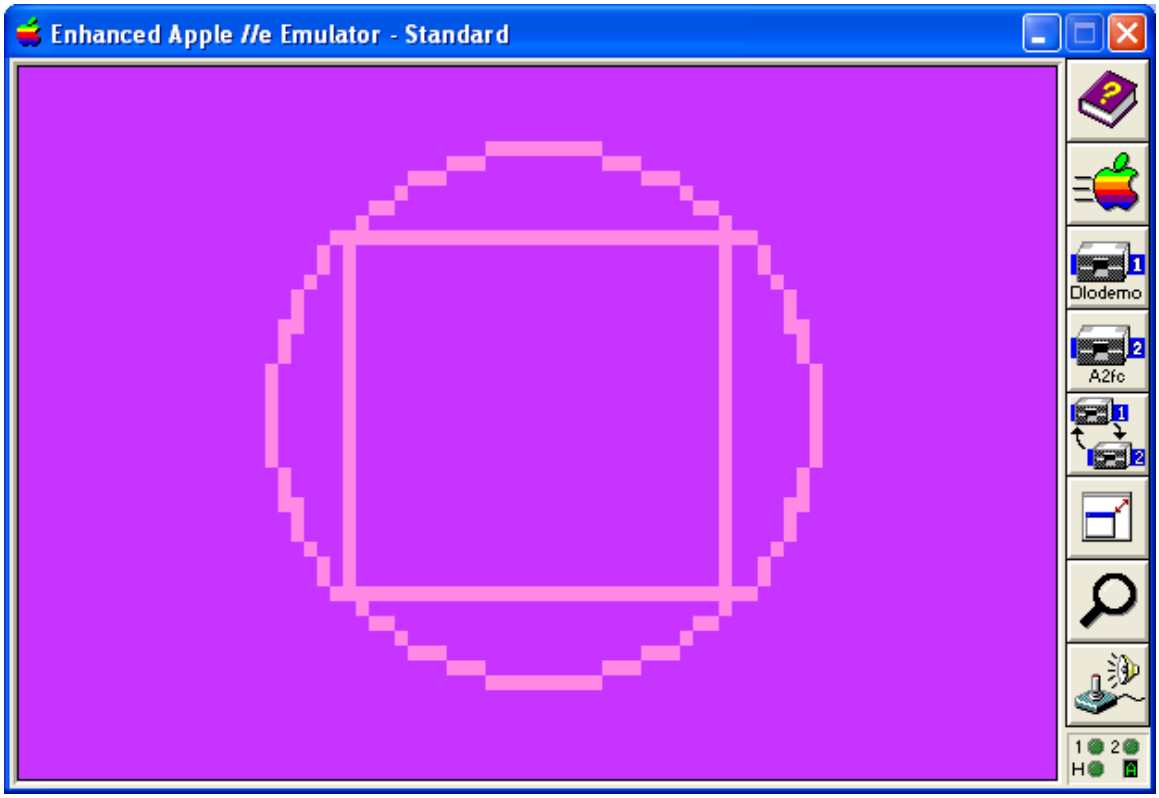
Rather than calling the Apple II's built-in LGR routines using inline assembly, an alternate method of plotting a pixel in DLGR is to address screen memory directly:

```
/* an equivalent plotting routine that does not use the
   Apple II's built-in LGR plotting routine
   but uses the textbase array and goes directly to memory */
void dloplot(unsigned char x,unsigned char y,
             unsigned char color)
{
    unsigned char *crt = (unsigned char *)(textbase[y/2]+x/2);

    if (x%2 == 0) {
       color = dloauxcolor[color];
       dhraux[0] = 0; /* select auxiliary memory */
    }
    else {
       dhrmain[0] = 0; /* select main memory */
    }

     if (y%2 == 0) {
         /* even rows in low nibble */
         /* mask value to preserve high nibble */
         crt[0] &= 240;
         crt[0] |= color;

     }
     else {
         /* odd rows in high nibble */
         /* mask value to preserve low nibble */
         crt[0] &= 15;
         crt[0] |= (color << 4);
     }

     if (x%2 == 0) dhrmain[0] = 0; /* reset to main memory */
}
```

Drawing Squares in DLGR

Drawing a square in DLGR uses the same 1.16 vertical aspect compression technique as drawing a circle to ensure that a square is roughly square:

```
/* square algorithm */
void dlosquare(int x,int y,int base,unsigned char color)
{
     int x1 = x - (base/2);
     int x2 = (x1 + base)-1;
     int height = (base * 100) / 116;
     int y1 = y - (height/2);
     int y2 = (y1 + height)-1;

     dloline(x1,y1,x2,y1,color);
     dloline(x1,y1+1,x1,y2-1,color);
     dloline(x2,y1+1,x2,y2-1,color);
     dloline(x1,y2,x2,y2,color);

}
```



Drawing a square circumscribed within a circle is somewhat similar to drawing a circumscribed equilateral triangle:

```
void dlosquarecircle(int x,int y,int base,unsigned char color)
{

     int x1 = x - (base/2);
     int x2 = (x1 + base) - 1;
     int height = (base * 100) / 116;
     int y1 = y - (height/2);
     int y2 = (y1 + height) - 1;
     int r = ihypot(base/2,height/2)+2;

     dloline(x1,y1,x2,y1,color);
     dloline(x1,y1+1,x1,y2-1,color);
     dloline(x2,y1+1,x2,y2-1,color);
     dloline(x1,y2,x2,y2,color);

     dlocircle(x,y,r,color);

}
```

Drawing Boxes in DLGR

From the view of not doing much math, a box is the most primitive of any of the graphics primitives composed of line elements and points:

```
void dlobox(unsigned char x1, unsigned char y1,
            unsigned char x2, unsigned char y2,
            unsigned char color)
{
    unsigned char x, y;
    /* swap horizontal co-ordinates if out of order */
    if (x1 > x2) {
       x = x2;
       x2 = x1;
       x1 = x;
    }
    /* swap vertical co-ordinates if out of order */
    if (y1 > y2) {
       y = y2;
       y2 = y1;
       y1 = y;
    }
    y = x2 + 1;
    for (x = x1; x < y; x++) dloplot(x, y1,color);
    y1++;
    y2--;
    dlovline(x1, y1, y2, color);
    dlovline(x2, y1, y2, color);
    y2++;
    for (x = x1; x < y; x++) dloplot(x, y2, color);

}
```

Unlike triangles, squares and circles, boxes do not concern themselves with proportion; the math to draw a box is very simple and self-explanatory. While more advanced primitives can be fun to diddle in DLGR, results are pretty lame and less than stellar, and barely better than LGR. But besides that, at the end of the day, how often do you even need a triangle or a square or even a circle? Some might even argue that spending any time at all to draw anything besides a box in DLGR is a waste of time.

Line Drawing Scripts in DLGR

Line drawing scripts in DLGR can be fun, in the same way that Apple II shape tables can be fun. They are calculation intensive and really serve no useful purpose on their own. They do however provide an alternate method to bitmapped graphics, of rendering a graphics image on the display:

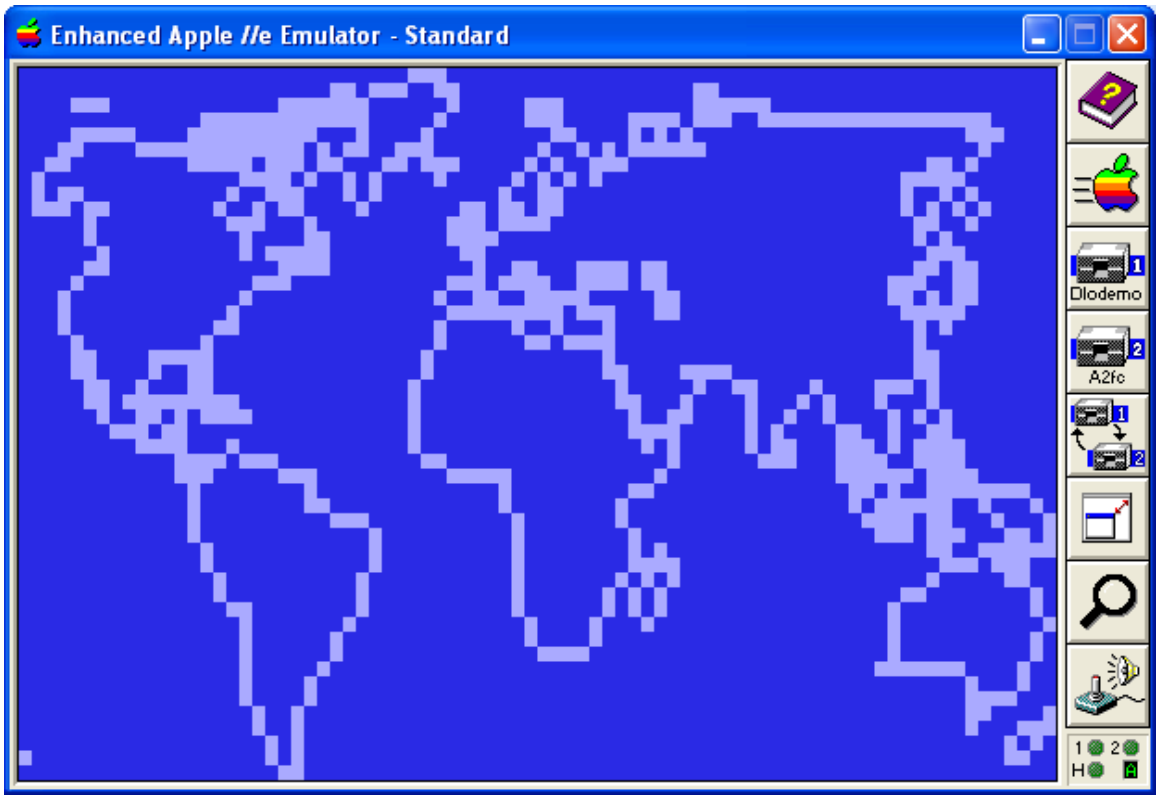

```
#define WORLDMAP 0
#define USAMAP 1
#define MAPLELEAF 2

/* the coordinates to draw the whole world */
int WORLD[] = {… -1, -2};
/* the coordinates to draw the USA */
int USA[] = {… -1, -2};
/* the coordinates to draw the canadian flag */
int MAPLE[] = {… -1, -2};
```

Line drawing scripts of x,y co-ordinates in dlodemo are stored as integer arrays of multiple line segments. Each line segment is terminated with x,y co-ordinates of -1,-1 and the script itself is terminated with -1,-2. In the USA map displayed above, line elements of several colors are plotted. This is a feature of the drawing script routine and these aren't just some random colors, but come from (yet another) table:

```
int colors[] = {
LOLTBLUE,LOPINK,LOLTGREEN,LOYELLOW,LOAQUA,LOWHITE,-1};
```

As you can see, the table above is used with the **drawmap()** function below when the alternating color option is selected by providing an out of range **drawcolor**:

```
void drawmap(int mapidx,int drawcolor)
{
  int *ptr, xidx=0,yidx=1,newx,oldx,newy,oldy,cidx=0;
  unsigned char color;
  /* select vector map or vector object */
  switch(mapidx)
  {
      case MAPLELEAF:   ptr = (int *)&MAPLE[0];
                        break;
      case USAMAP:      ptr = (int *)&USA[0];
                        break;
      case WORLDMAP:
      default:          ptr = (int *)&WORLD[0];
  }
  /* draw selected map or object */
  for (;;) {
      /* alternating color option */
      if (drawcolor < 0 || drawcolor > 15) {
            if (colors[cidx] < 0)cidx = 1;
            color = (unsigned char)colors[cidx];
            cidx++;
      }
      else {
          color = (unsigned char)drawcolor;
      }
      oldx= ptr[xidx];
      oldy= ptr[yidx];
      for(;;)
      {
          xidx +=2;
          yidx +=2;
          newx= ptr[xidx];
          newy= ptr[yidx];
          if (newx == -1) break;
          /* draw next line */
          dloline(oldx,oldy,newx,newy,color);

          oldx=newx;
          oldy=newy;
       }
       if (newy == -2)break;
       xidx +=2;
       yidx +=2;
   }
}
```



Dlodemo Program Organization

The dlodemo main program and its "helper" functions call the core routines previously listed. You can write your own DLGR cc65 pixel graphics demo with these self-same core routines or maybe even something a tad more useful, like a DLGR paint program or whatever you wish. But for now you may wish to look at how the dlodemo program pulls all this DLGR pixel graphics stuff together.

Helper Functions

The following prints the titling on the title screen at the start of the dlodemo program. This uses the bit-mapped font modified from the old Aztec C65 HGR font and not the "Tom Thumb" font. (Both fonts were discussed previously in this document.)

```
/* font layout - each character is 7 pixels wide x 8 pixels high
   DLGR screen resolution is 80 x 48... which means we can write
   Only 11 characters across x 6 rows down... this isn't much
   Space but enough for a title of some sort... the font routine
   Below takes this somewhat further cramming each character
   Together into a six pixel cell width, with lowercase L's and
   I's and such into 5.

   This gives us 13 characters to muck with...
   01234567890123
*/
```

```
char *title[] = {
   "Double Lo-Res",
   "Graphics Demo",
   "In cc65",
   "by",
   "Bill Buckels",
   "July 2014"};

#define CELL_WIDTH 6
#define CELL_HEIGHT 8
#define SCREEN_WIDTH 80
#define MAX_ROWS 6

void rainbowfont(char *str, int row, int col)
{
    int idx, color = 7;
    char buf[2], ch;

    buf[1]=0;
    for (idx=0;str[idx]!=0;idx++) {
         ch = buf[0] = str[idx];
         if (ch == 'l' || ch == 'i' || ch == '1' || ch == 32 ||
             ch == '.' || ch == '-' || ch == '!')col -=1;
         dlofont(buf,row,col,color,-1,1);
         color++;
         if (color > 15)color = 7;
         if (ch == 'l' || ch == 'i' || ch == '1' || ch == 32 ||
             ch == '.' || ch == '-' || ch == '!')col+=5;
         else col+=6;
    }
}
```

Helper Function dloboxes()

The **rainbowfont()** function above is called in the routine below. The demo is logically divided into 3 parts. The routine below is the first part. It draws a title screen and some geometrical shapes and then vanishes into the night:

```
void dloboxes(void)
{
    int b, r, x1,y1,x2,y2,idx,len,color;

    /* splash screen */
    /* draw a box around the screen */
    dlobox(0,0,79,47,LOPURPLE);
    /* loop for all 6 available lines */
    y1 = 0;
    for (idx = 0; idx < MAX_ROWS; idx++) {
        /* centre title horizontally */
        len = strlen(title[idx]) * CELL_WIDTH;
```

```
    x1 = (SCREEN_WIDTH - len) /2;
    rainbowfont(title[idx],y1,x1);
    y1+=CELL_HEIGHT;
}
cgetc();

/* square in circle demo */
dloresflood(LODKBLUE);
dlosquarecircle(40, 23, 26, (unsigned char)LOLTBLUE);
cgetc();
dloresflood(LOPURPLE);
dlosquarecircle(40, 23, 30, (unsigned char)LOPINK);
cgetc();
dloresflood(LODKGREEN);
dlosquarecircle(40, 23, 34, (unsigned char)LOLTGREEN);
cgetc();
dloresflood(LORED);
dlosquarecircle(40, 23, 38, (unsigned char)LOORANGE);
cgetc();

/* square demo */
dloresclear();
/* 1.16 base ratio in y axis */
b = (48 * 116) / 100;
for (color = 0; color < 6; color++) {
    dlosquare(40, 23, b-(color*6), (unsigned char)color+10);
}
cgetc();

/* triangle in circle demo */
dloresflood(LODKBLUE);
dlotricircle(40, 18, 23, (unsigned char)LOLTBLUE);
cgetc();
dloresflood(LOPURPLE);
dlotricircle(40, 18, 29, (unsigned char)LOPINK);
cgetc();
dloresflood(LODKGREEN);
dlotricircle(40, 18, 35, (unsigned char)LOLTGREEN);
cgetc();
dloresflood(LORED);
dlotricircle(40, 18, 41, (unsigned char)LOORANGE);
cgetc();

/* triangle demo */
dloresclear();
/* 1.30 base ratio in y axis */
b = (48 * 130) / 100;
for (color = 0; color < 6; color++) {
    dloequilateral(40, 23, b-(color*6),
                   (unsigned char)color+10);
}
cgetc();
```

```
    /* circle demo */
    dloresclear();
    /* 1.16 radius aspect ratio in y axis */
    r = (24 * 116) / 100;
    color = 7;
    for (b = 0; b < 8; b++) {
        dlocircle(40, 23, r-(b*2), (unsigned char)color);
        color++;
    }
    cgetc();

    /* box demo */
    dloresclear();
    x1 = 0;
    x2 = 79;
    y1 = 0;
    y2 = 39;
    for (color = 1; color < 16; color++) {
        dlobox(x1,y1,x2,y2, color);
        y1++;
        y2--;
        x1++;
        x2--;
    }
    dlothumb("Press a key",40,17,LOWHITE,255,'M');
    cgetc();
    dloclear_bottom();
    mixedtexton();
    dloprint_bottom(
    "Tired of looking at a bunch of boxes and stuff?",1,0);
    dloprint_bottom("Press any Key to see the USA!",3,0);
    cgetc();
}
```

Helper Function dloworld()

The function below is the second of the 3 parts of the dlodemo program. It implements the line drawing scripts previously discussed in this document. First we draw the map of the USA and wait for a keypress:

```
void dloworld(void)
{
    mixedtextoff();
    dloresclear();
    drawmap(USAMAP,-1);
    dlothumb("Press a key to C",40,0,LOWHITE,255,'M');
    dlothumb("The World of DLGR ",40,43,LOWHITE,255,'M');
    cgetc();
```

Then we draw the map of the world and wait for a keypress:

```
    dloresflood(LODKBLUE);
    drawmap(WORLDMAP,LOLTBLUE);
    cgetc();
```

Then we draw the outline of the Canadian Flag and wait for a keypress:

```
    dloresclear();
    drawmap(MAPLELEAF,LOPINK);
    dlothumb("Greetings from the",40,0,LOWHITE,255,'M');
    dlothumb("Great White North!",40,43,LOWHITE,255,'M');
    dlothumb("MADE IN CANADA",40,22,LOYELLOW,255,'M');
    cgetc();
```

Finally we switch to mixed-mode, and when we get the last keypress we are outa' here:

```
    dloclear_bottom();
    mixedtexton();
    dloprint_bottom("How's Your Eyeballs? Tired of looking at a
bunch of jaggies?",1,0);
    dloprint_bottom("Prolly... So Press any Key to treat them to
a bit of a massage.",3,0);
    cgetc();

}
```

Readability Considerations

"If the mountain won't come to Muhammad then Muhammad must go to the mountain."

I continue to review lots of C code that works well but fails to be readable, and therefore is not easy to share. Some notion that C is self-commenting doesn't appeal to me. Comments should always be in C code.

If you want to do Apple II programming, there's lots of low-level stuff to be mixed-in with your C code that can make reading it even more confusing. Add comments.

Some other notion that Assembly and Low-Level Code should always be in separate modules is too abstract. For one thing, it splits your C program into bits and pieces that are too hard for most people to think of in "English" in tandem with the low-level code and math and stuff that is necessary to accomplish a finished task.

All C programs should be documented and readable by your peers, so despite the fact that many will disagree with what I have already said, I consider my peers to be people who are average programmers and not the "hardcore" writers of compilers or accomplished mathematicians or hardware engineers.

Simply put, the average programmer that takes time to learn the Apple II or any programming platform or programming venue like Graphics, needs to be able to tie

thoughts and deeds to the task at hand, so nothing is more annoying or non-productive than to need to open 20 or 30 books to discover how one line of code works.

Elegance and Efficiency are important too, and so is separation for portability in cross-platform compilers like cc65, but at some point, readability, understanding, and application level maintenance need to take priority over cross-platform reuse.

Just as transparency is important in a free democracy, readability is important in the freedom of sharing knowledge, including the sharing of program code. In the real world, I never make my code so hard to read that only an expert can read it. I provide extensive documentation that explains fully what I am doing in both English and whatever other programming language that I am working-in. I fully acknowledge that this "average" approach won't appeal to some scientists and other experts. But they are not my peers.
.
You will note in the code above that I split the listing by adding some additional notations. I haven't done that much throughout this document because in most cases notation badly damages the readability of a code listing. I also initially said that I expect you to be able to (eventually) read C code to follow along. There's actually a little more to-it than that; ideally you should be thinking in "English" (or whatever language you normally think-in) while writing in C. If you don't already do that, put it on your list of things to learn… It would be pointless for me to make such a bold remark without an example of what I mean, so I offer the notations in the code above as said example.

And so to conclude my rant, my view of cc65 is that it is only a tool to write programs for the Apple II. If it needs to be "hacked" a little to make my programs work the way I want them to work, and to be read by an "average" programmer, the code and documentation I provide will always put readability ahead of other loftiness like portability and optimizations that can be added at the program level.

The notations in the code above are simply provided as a demonstration of how I expect you to (eventually) view a C program in the context of how I write one. If it works for you then great!

Helper Function doublerod()

After all my talk above about comments and readability I now offer a piece of code without comments that requires you to count on your fingers and your toes to understand. In my view the average programmer will be able to understand the following third and final helper function in the dlodemo program with some help from their fingers and toes and a scribble-pad (grab a piece of paper from your printer if you don't have a pad handy and if you need to go to the kitchen and grab a pen to jot down some notes).

Rod is a well used and well known kaleidoscope algorithm. I'll leave it to the reader to visit the following link and find-out a little more:

http://www.appleoldies.ca/azgraphics33/rod33.htm

As I have said, I will also leave it to the reader to decipher the math behind what is happening below. The core functions in dlodemo (that appeared earlier in this document) certainly provide sufficient information to do so.

The only clue that I am going to give you is that Rod's Color Pattern is twice as wide in DLGR as it is in LGR where in 1978, Randy Wigginton originally conceived this "simple but eloquent program. It generates a continuous flow of colored mosaic-like patterns in a 40 high by 40 wide block matrix. Many of the patterns generated by this program are pleasing to the eye and will dazzle the mind for minutes at a time."

```
/* rod's color pattern in dlogr */
int doublerod ()
{
    int i, j, k, w, fmi, fmk, color;
    unsigned char c=0;

    for (w = 3; w < 51; w++) {
        for (i = 1; i < 20; i++) {
            for (j = 0; j < 20; j++) {
                k = i + j;
                color = (j * 3) / (i + 3) + i * w / 12;
                fmi = 40 - i;
                fmk = 40 - k;

                dloplot((unsigned char)i,
(unsigned char)k,(unsigned char)color);
                dloplot((unsigned char)i+39,
(unsigned char)k,(unsigned char)color);
                dloplot((unsigned char)k,
(unsigned char)i,(unsigned char)color);
                dloplot((unsigned char)k+39,
(unsigned char)i,(unsigned char)color);

                dloplot((unsigned char)fmi,
(unsigned char)fmk,(unsigned char)color);
                dloplot((unsigned char)fmi+39,
(unsigned char)fmk,(unsigned char)color);
                dloplot((unsigned char)fmk,
(unsigned char)fmi,(unsigned char)color);
                dloplot((unsigned char)fmk+39,
(unsigned char)fmi,(unsigned char)color);

                dloplot((unsigned char)k,
(unsigned char)fmi,(unsigned char)color);
                dloplot((unsigned char)k+39,
(unsigned char)fmi,(unsigned char)color);
                dloplot((unsigned char)fmi,
(unsigned char)k,(unsigned char)color);
                dloplot((unsigned char)fmi+39,
(unsigned char)k,(unsigned char)color);
```

```
                dloplot((unsigned char)i,
(unsigned char)fmk,(unsigned char)color);
                dloplot((unsigned char)i+39,
(unsigned char)fmk,(unsigned char)color);
                dloplot((unsigned char)fmk,
(unsigned char)i,(unsigned char)color);
                dloplot((unsigned char)fmk+39,
(unsigned char)i,(unsigned char)color);

                if (kbhit()) {
                    c = cgetc();
                    return (int)c;
                }
            }
        }
    }

    if (kbhit()) {
        c = cgetc();
    }
    return (int)c;
}
```

This concludes the 3 helper functions used in dlodemo.

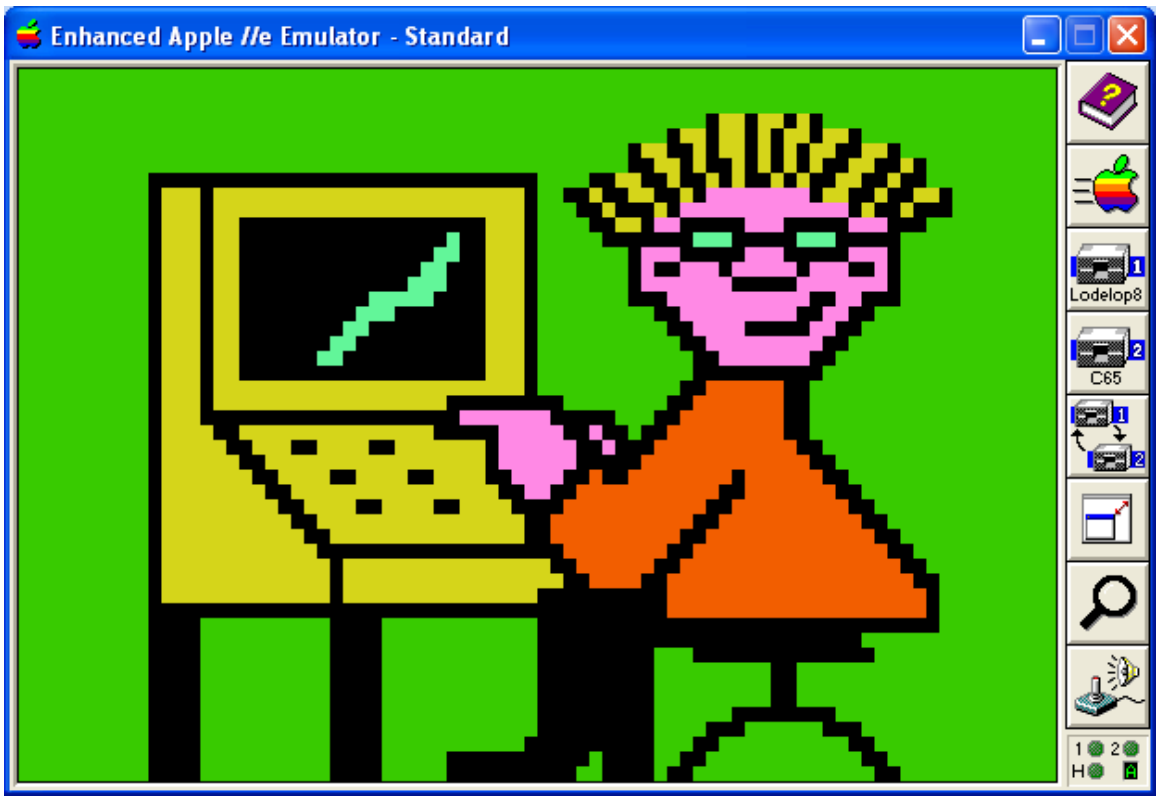
Main Program

```
int main()
{
    /* initialize 80 column card */
    /* this inializes the language card safely for cc65's
       standard memory configuration */
    videomode(VIDEOMODE_80COL);
    clrscr();

    /* turn-on double hi-res */
    dloreson();
    dloresclear();

    dloboxes();

    dloworld();

    for (;;) {
        mixedtextoff();
        dloresclear();
        dloclear_bottom();
        mixedtexton();
        dloprint_bottom("Double Rod - Rod's Color Pattern in
Duplex and Glorious Double Lo-Res...",0,0);
```

```
        dloprint_bottom("Rod's Color Pattern by Randy Wigginton
originally written in BASIC appeared on",1,0);
        dloprint_bottom("page 55 of the Red Book distributed by
Apple Computer, Inc. circa 1978.",2,0);
        dloprint_bottom("Press Escape to End or Any Other Key to
Refresh...",3,0);
        if (doublerod()==27)break;

    }

    mixedtextoff();
    dloresclear();
    dloresoff();

    /* set to 80 column again so cc65 can exit cleanly */
    videomode(VIDEOMODE_80COL);
    clrscr();

    return 0;
}
```

Well, dlodemo's main program certainly looks simple.

Additional Notes



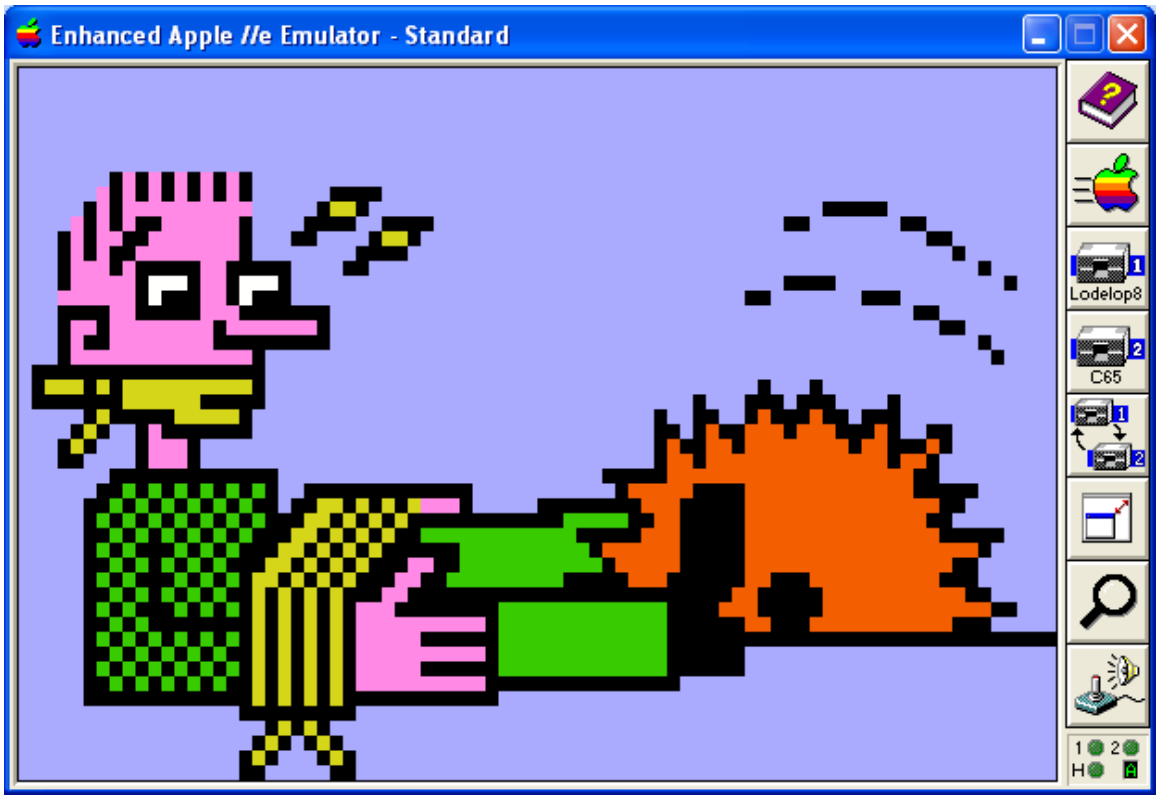
So there you have it! cc65 now supports DLGR pixel graphics, but then it always did, just not directly through its tgi driver scheme. That's a project for a different day and perhaps for a different programmer.

Hopefully you gained something from all of this, and please remember not to clobber the text screen holes if you decide to do something in DLGR.

As I have said before about DHGR demos, bitmapped DLGR demos are sexier than pixel graphics demos like this one. There is no doubt about that.

Bill Buckels
bbuckels@mts.net

PS – Let me know if I forgot to mention something ☺



The Importance of a well placed C Saw cannot be overstated.