Introduction to Super Hi-Res in cc65

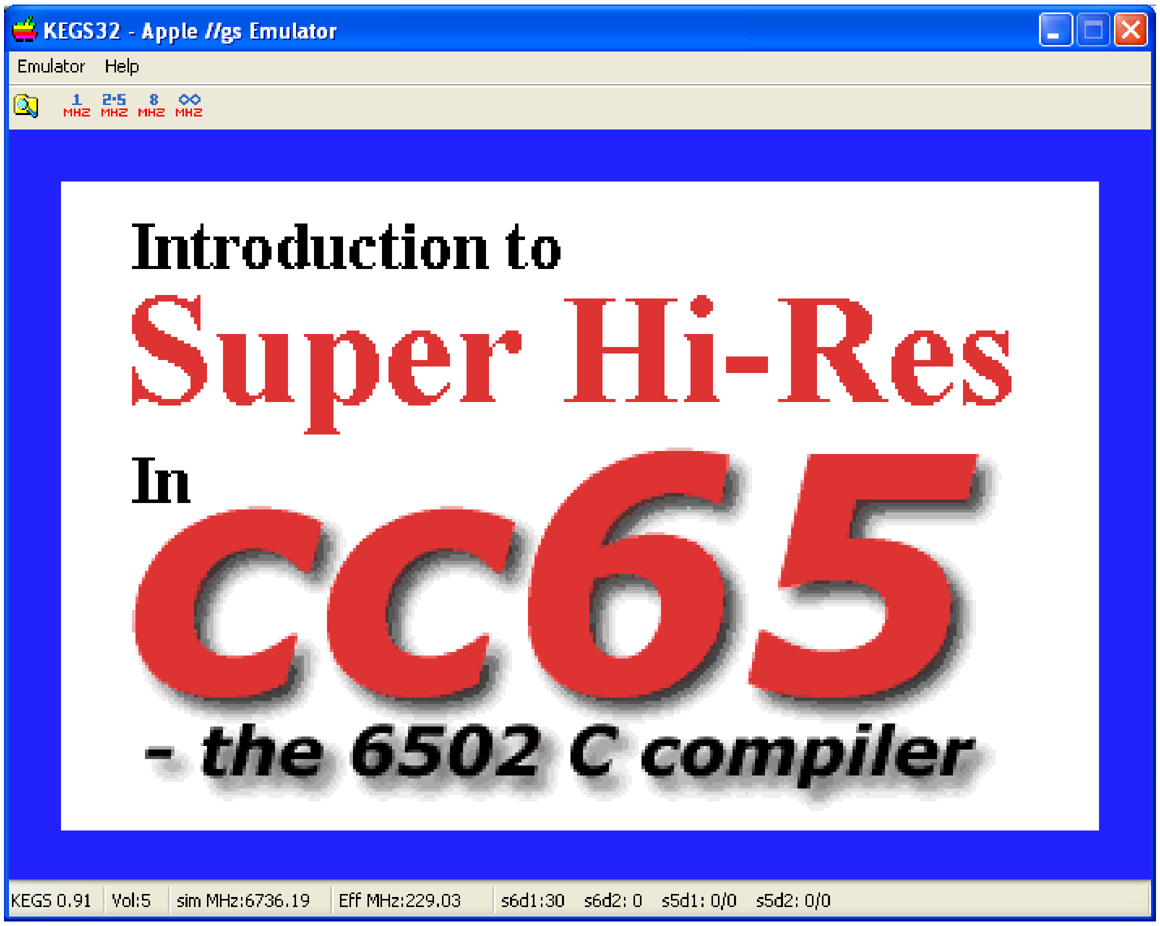
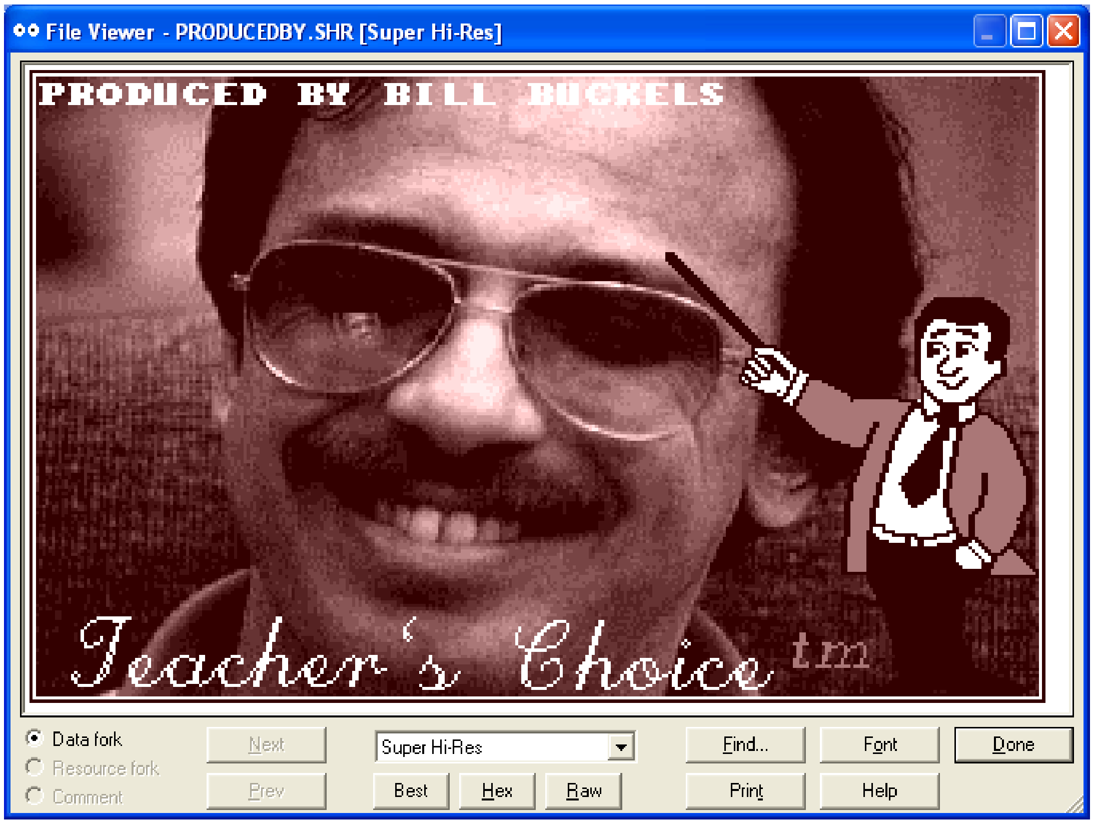

KEGS32 - Apple //gs Emulator
Emulator Help
1 MHz 2.5 MHz 8 MHz ∞ MHz
Introduction to
Super Hi-Res
In
cc65
- the 6502 C compiler
KEGS 0.91 Vol:5 sim MHz:6736.19 Eff MHz:229.03 s6d1:30 s6d2: 0 s5d1: 0/0 s5d2: 0/0

Table of Contents



File Viewer - PRODUCEDBY.SHR [Super Hi-Res]
PRODUCED BY BILL BUCKELS
Teacher's Choice tm
Data fork
Resource fork
Comment
Next
Prev
Super Hi-Res
Best
Hex
Raw
Find...
Font
Done
Print
Help

Forward

If you are wondering why anyone would write a pixel graphics program for the Apple IIgs Super Hi-Res (SHR) display using the cc65 6502 8-bit C compiler, when the Apple IIgs already has built-in routines and 16-bit compilers that will do pixel graphics on the SHR display… Famed New Zealander Sir Edmund Percival Hillary's first words to lifelong friend George Lowe on returning from Everest's summit probably say it best;

"Well George, we knocked the bastard off!"

Years earlier, another George, George Mallory, who disappeared during his 3rd attempt to reach the same summit, provides the reason why a cc65 SHR pixel graphics "knock-off" wasn't done until now. Before he disappeared, when Mallory was asked why he wanted to climb Mount Everest, he answered;

"Because it's there!"

Mahomet also made the people believe that he would call a hill to him…

Cc65 is a capable and modern C 6502 8 bit cross-compiler for writing Apple II programs, but a programmer wishing to write Apple II SHR graphics programs with only the routines that come with cc65.will share the programmatic equivalent of Mallory's fate.

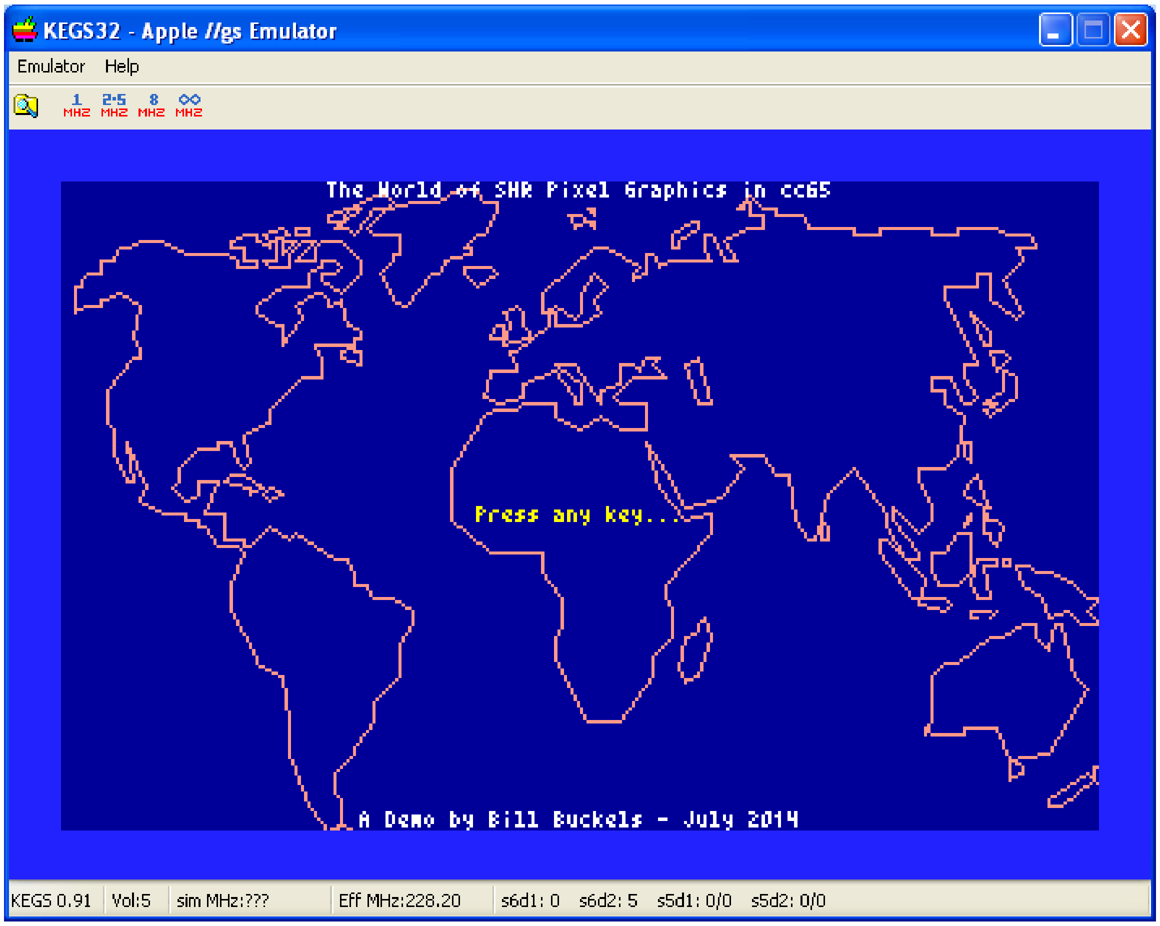
The SHR pixel graphics routines in this document are part of a larger Apple II cc65 project that I am in the process of "knocking off"; no less than providing methodology, routines and instructions missing from cc65 to do all there is to do on the Apple II. SHR pixel graphics have been out there a long time too (longer than cc65), but without a commitment to results, and to making cc65 work with SHR by climbing the hill from the bottom instead of attempting a far view from the top (which is the wrong way around), the fact of extending cc65's capability to SHR pixel graphics would still not be here or there even after I disappeared like Mallory, mumbling in csa2 and on facebook about some apparition of "vapour ware" SHR pixel graphics routines here and there.

On a Clear Disk you can Seek Forever

So if you are the type of "all-weather" C programmer that doesn't depend on a built-in toolbox to get the job done, and you enjoy doing "knock-off" applications in low-level C with a limited smattering of 6502 assembler, and are not motivated to some GUI-bound predecessor of what passes for C on today's desktops, ready yourself for a brief scale up cc65's SHR pixel graphics "molehill" (this is not a mountain nor "rocket science").

Simply knowing the C programming language and the cc65 compiler will be of limited use to you without knowledge of the Apple II and graphics programming. This document is only intended to give you a starting point. Working with these routines and the source code provided is highly recommended. Working harder is also recommended.

Chapter Three – Super Hi-Res Pixel Graphics in cc65


KEGS32 - Apple //gs Emulator
Emulator Help
1 MHz 2.5 MHz 8 MHz ∞ MHz
The World of SHR Pixel Graphics in cc65
Press any key...
A Demo by Bill Buckels - July 2014
KEGS 0.91 Vol:5 sim MHz:??? Eff MHz:228.20 s6d1: 0 s6d2: 5 s5d1: 0/0 s5d2: 0/0

Introducing Shrworld

The shrworld graphics demo is my first Apple IIgs Super Hi-Res (SHR) pixel graphics program. While it is pretty "lame" as graphics demos go, using only the cc65 compiler and writing "from the ground-up" it took just a few days to finish.. It could probably be tweaked in some (perhaps many) places to run more quickly. Download the program here: http://www.appleoldies.ca/cc65/programs/shr/shrworld.zip

SHR graphics mode on the "stock" Apple IIgs provides two resolutions; 320 x 200 (mode320) and 640 x 200 (mode640). The shrworld program uses mode320 exclusively. SHR mode320 allows 16 colors per display line. The SHR display is "palettized" but is not limited to a palette of "fixed colors" like the earlier Apple II graphics modes. Since SHR is in 12 bit color, any 16 of 4096 possible colors can be used per line. However, without doing "clever programming", only 16 palettes can be shared between the 200 SHR display lines. For this demo only one palette is used for all the display lines so "clever programming" is not needed. ("clever programming" is an official Apple Computer term for "constantly rotating palettes during SHR screen updates".)

Program Summary

The shrworld program advances on keypresses. A series of SHR screens is displayed including graphics primitives, fonts, and line drawing routines, and finally shrworld ends with a kaleidoscope until ESC is pressed. Then shrworld exits.

Building the Program

The shrworld program is a cc65 binary program with a starting address at $4000. It is launched using Oliver Schmidt's LOADER.SYSTEM ProDOS 8 SYS program. For consistency it comes with the same linker configuration and build environment as my other cc65 SHR demo programs. The build environment is complete with a gcc compatible MAKEFILE. There is no need to document it here. For more info, review the MAKEFILE and the other baggage that comes with shrworld .

These SHR demos are provided with monolithic source code with many of the routines in header files; they *DO NOT* link to any special library. They only use 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 if at all. 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.

Core Routines

Auxiliary Memory Routines

AUXMOVE is generally a handy routine for any Apple IIe 6502 program (including a cc65 program like shrworld) that stores and retrieves data in auxiliary memory. AUXMOVE must be called with 80Store off. The carry flag determines the direction of the memory move:

```
/* move a block of data from main to auxiliary memory */
#pragma optimize (push,off)
void maintoaux(unsigned src0, unsigned src1, unsigned dest0)
{

    unsigned *src  = (unsigned *)0x3c;
    unsigned *dest = (unsigned *)0x42;

    src[0] = src0;
    src[1] = src1;
    dest[0] = dest0;

    asm("sec");
    asm("jsr $c311");

}
#pragma optimize (pop)
```

```
/* move a block of data from auxiliary to main memory */
#pragma optimize (push,off)
void auxtomain(unsigned src0, unsigned src1, unsigned dest0)
{

    unsigned *src  = (unsigned *)0x3c;
    unsigned *dest = (unsigned *)0x42;

    src[0] = src0;
    src[1] = src1;
    dest[0] = dest0;

    asm("clc");
    asm("jsr $c311");

}
#pragma optimize (pop)
```

When the SHR display is active, AUXMOVE is not only handy, but a necessity, and the only way to move data between the SHR display in Auxiliary Memory and program memory in a cc65 8-bit 6502 C program without writing 65816 subroutines outside the cc65 programming environment .

On a stock Apple IIe there are only 2 - 64K banks of memory; Bank $00 on the main board and Bank $01 on the language card. On the Apple IIgs there are 256 – 64K banks of memory, Bank $00 to Bank $FF.

The SHR screen memory locates itself in Bank $E1 in the 32,768 bytes (32K) starting at memory address $2000. This 32K of memory is physically located on the IIgs Video Graphics Controller (VGC). When engineer Larry Thompson designed the VGC he combined two separate 16K blocks of memory to make it appear to the programmer as a single continuous 32K block of memory.

When SHR is turned-on (see the **shgron()** function below), Auxiliary Memory is shadowed into the real screen memory (at $E12000) by clearing bit 3 of the IIgs shadow register at $C035.

Don't worry too much if you don't understand some of this. The shrworld code works fine as far as I know and you can "muck with it" at your leisure. But first read further and things should become clearer although I can't promise an "epiphany"☺

Setting-up the SHR Display

The following functions are used to set-up the SHR display:

SHR Soft Switches

```
#define  gswitch ((unsigned char*)0xC029)
#define  shadow  ((unsigned char*)0xC035)
/* graphics - save previous setting of gswitch */
unsigned char gsave = 0xff;
/* shadow memory - save previous setting of shadow switch */
unsigned char ssave = 0xff;

/* turn-on SHR */
void shgron(void)
{
    gsave = gswitch[0];       /* save previous gswitch settings */
    gswitch[0]= gsave | 0xc0;/* shgr on  - set bits 6 and 7 */
    ssave = shadow[0];
    /* Bank $01 is shadowed into $E1 by clearing bit
       3 of the Shadow register at $C035 */
    shadow[0] = ssave & 0xf7;

}

/* turn-off SHR */
void shgroff()
{
    gswitch[0]=gsave; /* shgr off - restore previous setting */
    shadow[0] =ssave;
}
```

SHR Initialization

There is more to setting-up the SHR display than just setting some "soft-switches". Since SHR Video Memory is divided into 3 parts in Auxiliary Memory starting at $2000, with the scanline control bytes (scb's) and the palettes following 32000 bytes of image data, all three of these areas need to be initialized before using the SHR display:

```
/* buffers for 16 palettes and 200 scanline control bytes */
unsigned char palbuf[512], scbbuf[256];

void clearpalette()
{
    unsigned src1 = (unsigned)&palbuf[0];
    unsigned src2 = src1 + 511;
    /* zero palette in auxiliary memory */
    maintoaux(src1,src2,0x9e00);
}
```

```
void clearscbs()
{
    unsigned src1 = (unsigned)&scbbuf[0];
    unsigned src2 = src1 + 199;

    /* zero scanline control bytes in auxiliary memory */
    maintoaux(src1,src2,0x9d00);

}

void initbuffers(void)
{
    memset((char *)&palbuf[0],0,512);
    memset((char *)&scbbuf[0],0,256);

}
```

Initializing and using the three SHR display memory areas requires additional information which will be provided throughout this document. But you can see in the code above that the palette and scanline control bytes are buffered in program memory and are initially cleared, then moved to the SHR display in Auxiliary Memory. Doing so has the same effect as clearing the SHR display to black, because effectively an all-black palette has been created, and all 200 lines of the scanline control bytes have also been cleared. So they initially point all 200 scanlines to palette "zero"; the first of 16 "blank" palettes in the SHR palette memory.

With the palettes cleared, we don't see what was in the SHR display when we started. And we can put whatever we want in the SHR's image data memory area without it being displayed, until we finally set-up the scanline control bytes to point our scanlines to specific palettes other than the first palette if we need to, and then finally when we put the color values into our palette(s), whatever is in the image data area will be displayed.

In my experience this is fundamentally how some other "classic" palletized display hardware like the IBM-PC's VGA display also works.

<u>Text Mode Initialization</u>

```
/* turn-on text mode */
void texton(void)
{
      POKE(0xc051,0);  /* text */
      POKE(0xc054,0);  /* page 1 */
}
```

The function above is self-explanatory. It may not be necessary to explicitly set to text mode in this demo. But because it is simple enough to do I have done so anyway.

SHR Initialization Sequence

Let’s take a brief look at the start of the shrworld main program right now to see how this is done in practice. In the code below our first step is to clear the palette and the scb buffers using the **`initbuffers()`** function previously discussed. But first, because we will be leaving text mode shortly, we prompt the user to select a 16 color palette for the demo.

Setting SHR palettes will be discussed later in the document, so for now just try to get some idea of what is going-on “behind the scenes”, and then run the start of the demo to see how the SHR initialization sequence works in-practice.

```
/* initialize empty palette and scb's */
initbuffers();
puts("Select a 16-color palette:");
puts("1 - Kegs32");
puts("2 - CiderPress");
puts("3 - Old AppleWin");
puts("4 - New AppleWin");

ch = cgetc();
clrscr();

/* turn shr on */
shgron();
/* now that the shr display is in auxiliary memory */
/* clear the palette and point all the scbs to the
   first palette */
clearpalette();
clearscbs();
/* clear the shr screen to blue before setting the palette...
   it will clear to black initially */
clear320((unsigned char)LODKBLUE);
/* set the palette that the user has selected
   and the screen will change to blue */
switch(ch) {
    case '1': ptr = (unsigned char *)&rgbkegs32[0];break;
    case '2': ptr = (unsigned char *)&rgbciderpress[0];break;
    case '3': ptr = (unsigned char *)&rgbawinold[0];break;
    default:  ptr = (unsigned char *)&rgbawinnew[0];break;
}
setpalette(ptr,0);
```

Plotting an SHR Pixel

The **`putpixel320()`** routine below is the core of shrworld's plotting functions. It plots a pixel on the SHR mode320 display in one of 16 colors; it gets a byte of image data from SHR display memory, writes a pixel in the specified color to the byte, then puts the updated byte back in SHR display memory, at the pixel position specified by the x,y co-ordinates:

```
#pragma optimize (push,off)
void putpixel320(unsigned x,unsigned y, unsigned char color)
{
    unsigned *src  = (unsigned *)0x3c;
    unsigned *dest = (unsigned *)0x42;
    unsigned pixel, buf = (unsigned) &crt[0];

    /* return if out of range */
    if (x > 319 || y > 199) return;

    pixel = (unsigned) (0x2000 + (y * 160) + (x/2));

    /* move pixel to buffer */
    src[0] = src[1] = pixel;
    dest[0] = buf;
    asm("clc");
    asm("jsr $c311");

    if (x%2 == 1) {
        /* odd pixels in low nibble */
        /* mask value to preserve high nibble */
        crt[0] &= (unsigned char) 240;
        crt[0] |= (unsigned char) color;
    }
    else {
        /* even pixels in high nibble */
        /* mask value to preserve low nibble */
        crt[0] &= (unsigned char) 15;
        crt[0] |= (unsigned char)(color << 4);
    }

    /* move buffer to pixel */
    src[0] = src[1] = buf;
    dest[0] = pixel;
    asm("sec");
    asm("jsr $c311");
}
#pragma optimize (pop)
```

Since the SHR display memory cannot be accessed directly (SHR is in auxiliary memory), a 1 byte buffer in main memory is used by **`putpixel320()`** as a temporary update area. Another routine called AUXMOVE ($C311) located in the Apple II's 80 column firmware is called by **`putpixel320()`** to move image data between SHR display memory and this 1 byte buffer in program memory.

The SHR mode320 display uses a “packed pixel” of 4 bits, which allows 2 pixel indices to be stored in the same byte. A 320 pixel scan-line is stored in 160 bytes. While this “scheme” is efficient “memory-wise”, it can be drastically inefficient when writing a single pixel to the SHR display since adjacent pixels must be preserved. This “packed pixel” is why **`putpixel320()`** needs to temporarily “double-buffer” the SHR display memory by doing a “get pixel” and an “update pixel” before doing a “put pixel”.

Plotting an SHR Pixel Pair

When writing a “pixel pair” or a line of “pixel pairs”, single-buffering can be used to gain some speed since adjacent pixels do not need to be preserved. For drawing vertical lines, **`putpixel320()`** and “double-buffering” are used in shrworld, but for drawing horizontal lines a combination of single pixels and pixel pairs can be used:

```
#pragma optimize (push,off)
void putpairs320(unsigned x, unsigned y, unsigned pairs, unsigned
char color)
{
    unsigned *src  = (unsigned *)0x3c;
    unsigned *dest = (unsigned *)0x42;
    unsigned buf = (unsigned) &scanline[0];

    if (y>199) return;

    /* set pair color */
    if (color < 16)memset((char *)&scanline[0],
                          (char)(color << 4 | color),pairs);

    /* move buffer to shr display */
    src[0] = buf;
    src[1] = buf + pairs - 1;
    dest[0] = (unsigned) (0x2000 + (y * 160) + (x/2));
    asm("sec");
    asm("jsr $c311");
}
#pragma optimize (pop)
```

Pixel pairs (2 pixel color bytes) can also be used for drawing colored blocks.

Plotting Pixels in a Buffer

It is more efficient to create an entire graphics object by buffering a “chunk” of the SHR screen followed by plotting pixels in a buffer, then moving the updated “chunk” back to the SHR screen, than to continually use AUXMOVE to transfer a pixel at a time. However due to memory constraints, for large objects this is not always possible without doing complex “paging” of screen chunks. So when writing shrworld, I limited paging to one of the two font routines because I did not want to get too “fancy” For the font routines it makes good sense to use “paging”, since they are “pixel intensive” and generally require the screen background to be preserved. Since the font in shrworld is only 6 lines high, only 6 lines of the SHR screen need to be buffered to plot a line of text, and so it was easier to find enough memory for a relatively small 6 line “page” buffer:

```
/* plots pixel in buffer */
void bufferpixel320(unsigned x,unsigned y, unsigned char color)
{
    unsigned offset = (unsigned) ((y * 160) + (x/2));

    /* return if out of range */
    if (x > 319 || y > 6) return;

    if (x%2 == 1) {
        /* odd pixels in low nibble */
        /* mask value to preserve high nibble */
        scanline[offset] &= (unsigned char) 240;
        scanline[offset] |= (unsigned char) color;
    }
    else {
        /* even pixels in high nibble */
        /* mask value to preserve low nibble */
        scanline[offset] &= (unsigned char) 15;
        scanline[offset] |= (unsigned char)(color << 4);
    }
}
```

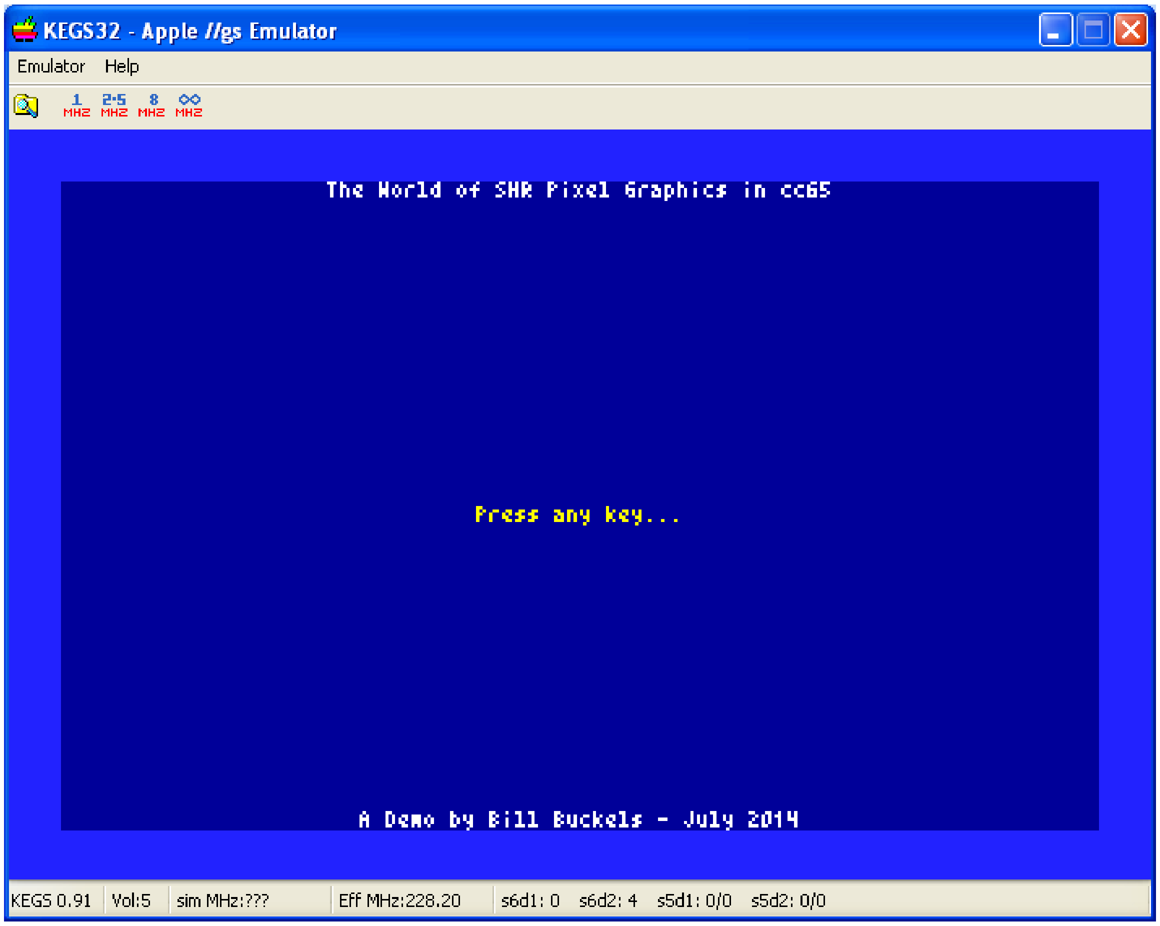
An additional advantage to plotting a text line as an “off-screen” chunk is that cc65 can optimize the pixel plotting, since assembly is not mixed with the C code. One other obvious efficiency advantage to plotting pixels in a buffered “chunk” of SHR screen data is that less code and fewer instructions are needed. This can be visually verified by comparing the code from **`putpixel320()`** with the code from **`bufferpixel320()`**.

Clearing the SHR Screen

Clearing SHR’s image data area (the SHR screen) is done by transferring a buffer of “colored” bytes of “pixel pairs” from main memory to the SHR display in auxiliary memory. The SHR display starts at $2000 in auxiliary memory and unlike the earlier Apple II graphics (and text) modes, it is linear and non-interleaved, and has no “screen holes”, which makes screen line memory address calculations arguably simpler. In the **`putpixel320()`** code above, you saw how the SHR display address of a byte of “pixel pairs” is calculated:

```
pixel = (unsigned) (0x2000 + (y * 160) + (x/2));
```

Since clearing the screen in shrworld is accomplished by doing a "wipe down" with a colored scan-line, and loops through all 200 of the SHR scan-lines, a similar calculation is done to find the memory location of the start of each scanline:



The World of SHR Pixel Graphics in cc65

Press any key...

A Demo by Bill Buckels - July 2014

KEGS 0.91 Vol:5 sim MHz:??? Eff MHz:228.20 s6d1: 0 s6d2: 4 s5d1: 0/0 s5d2: 0/0

```
void clear320(unsigned char color)
{
unsigned char paircolor = (unsigned char) (color << 4 | color);
unsigned y;
unsigned src1 = (unsigned) &scanline[0], src2 = src1+159;

    /* put pixel pair array on SHR display */
    /* fill scanline buffer with color pairs */
    memset((char *)&scanline[0],paircolor,160);
    /* wipe-down 200 scan lines with color */
    for (y=0;y<200;y++) {
        maintoaux(src1,src2,(unsigned) (0x2000 + (y * 160)));

    }
}
```

To prevent cc65 from destructively optimizing functions that use inline assembly the **`#pragma optimize`** "wrapper" is used throughout shrworld.

When using inline assembly, the practice of calling a simpler proxy function to do the assembly call means that cc65 can optimize more complicated code in the calling function. The `putpixel320()` code uses inline assembly to call AUXMOVE directly and is "wrapped" by `#pragma optimize` so it will not be optimized. But `clear320()` does not use inline assembly to call AUXMOVE directly, and instead uses the `maintoaux()` proxy function to indirectly call AUXMOVE, so cc65 can optimize the `clear320()` code. Let's look at the `maintoaux()` function again before we move on:

```
/* move a block of data from main to auxiliary memory */
#pragma optimize (push,off)
void maintoaux(unsigned src0, unsigned src1, unsigned dest0)
{

    unsigned *src  = (unsigned *)0x3c;
    unsigned *dest = (unsigned *)0x42;

    src[0] = src0;
    src[1] = src1;
    dest[0] = dest0;

    asm("sec");
    asm("jsr $c311");

}
#pragma optimize (pop)
```

__**SHR Palettes**__

In the `clear320()` code above, you saw the creation of a colored "pixel pair":

```
paircolor = (unsigned char) (color << 4 | color);
```

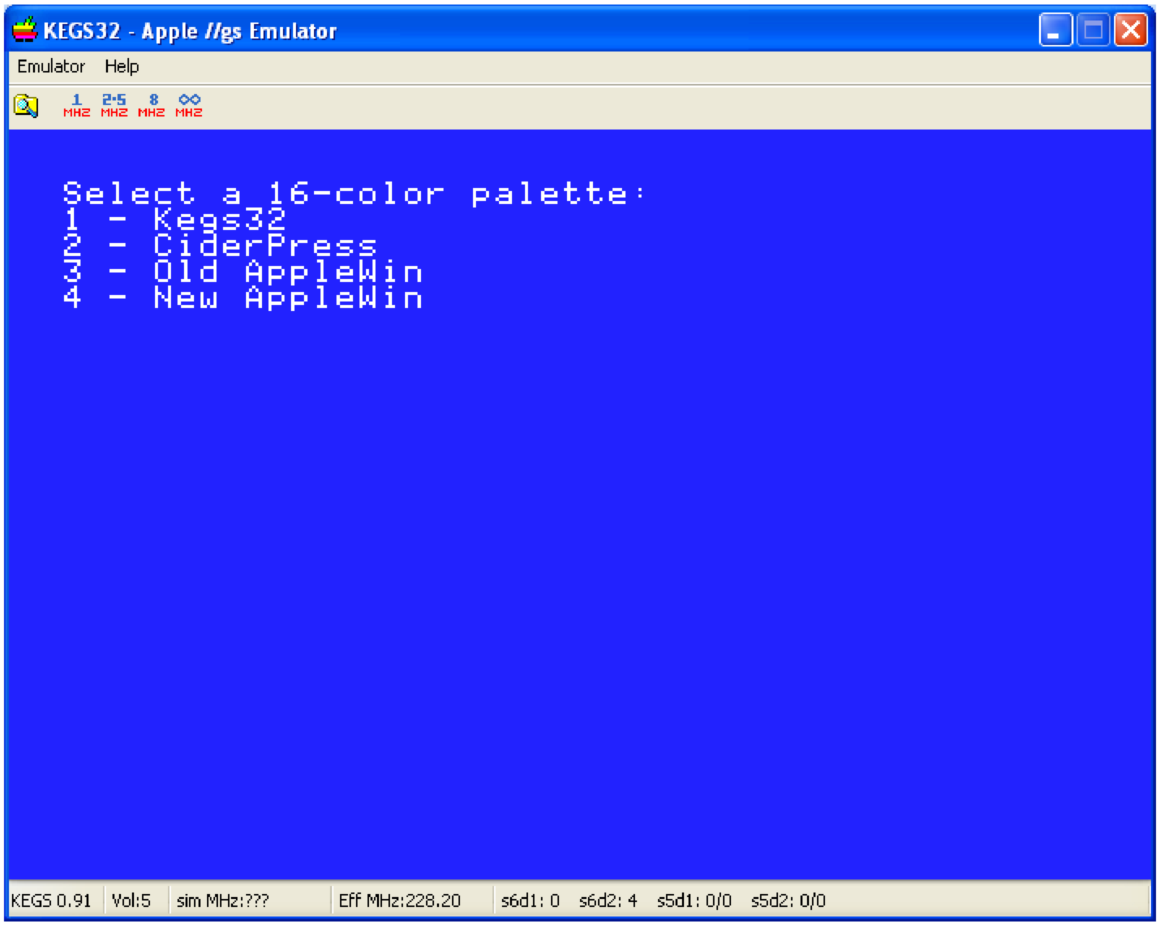
I have also mentioned palettes and palette indices, so with all this colorful talk, it is now time to review where the colors that shrworld uses come from.

SHR mode320 and SHR mode640 use two different methods to store colors in a palette. For this demo, and this document I am only covering mode320 palettes and since shrworld's routines use a fixed palette of 16 colors, I am not concerning myself with explaining the complexities of how to map multiple palettes for mode256 and mode3200 SHR display. That explanation will follow in some other chapter about techniques for developing optimal palettes for creating and displaying bit-mapped graphics in SHR.

However, SHR pixel graphics can be adversely affected when combined with bit-mapped graphics unless a palette contains both the colors in a background image and the colors needed to draw pixels consistently on the image, and "locks" the pixel colors that are needed in the palettes of the bit-map. But since shrworld is exclusively a pixel graphics program with only 16 colors from a single palette, this is not an issue with this demo.

Selecting 16 Colors

Shrworld provides a choice of four 16 color palettes when the program starts. These palettes are based on Windows programs widely used by today's Apple II users.


KEGS32 - Apple //gs Emulator
Emulator Help
1 MHz 2.5 MHz 8 MHz ∞ MHz
KEGS 0.91 Vol:5 sim MHz:??? Eff MHz:228.20 s6d1: 0 s6d2: 4 s5d1: 0/0 s5d2: 0/0

```
Select a 16-color palette:
1 - Kegs32
2 - CiderPress
3 - Old AppleWin
4 - New AppleWin
```

```
/* 4 - palette options */
/* these are the rgb values of the lo-res colors from the
AppleWin and Kegs32 Emulators and from the CiderPress File
Viewer. two sets of AppleWin Colors are provided: one from an
older version and one from a newer version. the routines in this
demo are based on a fixed palette so it was convenient to use
known colors and color order.*/

unsigned char rgbawinold[48] = {…};
unsigned char rgbawinnew[48] = {…};
unsigned char rgbkegs32[48] = {…};
unsigned char rgbciderpress[48] = {…};
```

One of the reasons that I stayed with LORES colors and color order is to be consistent; I have used these in my other cc65 graphics programs and documentation. Of course the palettes in your own SHR pixel graphics programs can be anything you wish.

Setting an SHR Palette

```
/* sets a 12 bit color palette line from an array of 24 bit color
values */
/* rgbpal is a 48 byte character array of 16 - r,g,b values */
/* palidx is the SHR palette number in the range of 0-15 */
void setpalette(unsigned char *rgbpal,int palidx)
{
    unsigned char r,g,b;
    int i,j,k;
    unsigned src1 = (unsigned)&shrpal[0], src2 = src1 + 31;

    /* build 12 bit palette line of $0RGB color entries
       from 24 bit r,g,b values */
    for (i=0,j=0,k=0;i<16;i++,j+=3,k+=2) {
        r = rgbpal[j]   >> 4;
        g = rgbpal[j+1] >> 4;
        b = rgbpal[j+2] >> 4;
        shrpal[k] = (unsigned char)((g << 4) | b);
        shrpal[k+1] = r;
    }

    /* move palette line to palette */
    maintoaux(src1, src2,(unsigned)(0x9e00 + (palidx * 32)));

}
```

The code above is self-explanatory and can be used in your own SHR pixel graphics programs to set the palettes of your choice.

Plotting Lines in SHR

The shrworld program has a variety of line plotting functions.

Plotting Horizontal Lines

```
/* potentially faster hline using a combination of pixels and
pairs */
void hline320(unsigned y,unsigned x1, unsigned x2,
              unsigned char color)
{
    unsigned temp, pairs;
    /* swap co-ordinates if out of order */
    if (x1>x2) {
        temp = x1;
        x1   = x2;
        x2   = temp;
    }
    if (y > 199 || x1 > 319) return;
    if (x2 > 319) x2=319;
    if (x1 == x2) {
        putpixel320(x1,y,color);
        return;
    }
    if ((x1%2) == 1) {
        putpixel320(x1,y,color);
        x1++;
    }
    if ((x2%2) == 0) {
        putpixel320(x2,y,color);
        x2--;
    }

    pairs = ((x2 + 1) - x1) / 2;
    if (pairs > 0) {
        putpairs320(x1,y,pairs,color);
    }
}
```

As you can see, both **`putpixel320()`** and **`putpairs320()`** are called by **`hline320()`**. As previously noted, putting byte-aligned pairs of pixels on the SHR screen is faster than plotting individual pixels. Also as noted previously, and shown above, calling functions that use inline assembly to plot pixels or pixel-pairs allows cc65 to optimize the calling code (in this case **`hline320()`** can be optimized).

<u>Plotting Vertical Lines</u>

Unlike **`hline320()`**, **`vline320()`** is "pixel-intensive". Pixel pairs can't be used:

```
/* simple vertical line - must set individual pixels */
void vline320(unsigned x,unsigned y1, unsigned y2,
              unsigned char color)
{
    unsigned temp;
    /* swap co-ordinates if out of order */
    if (y1>y2) {
        temp = y1;
        y1   = y2;
        y2   = temp;
    }
    y2++;
    while (y1 < y2) {
        putpixel320(x,y1,color);
        y1++;
    }

}
```

Plotting Vectors

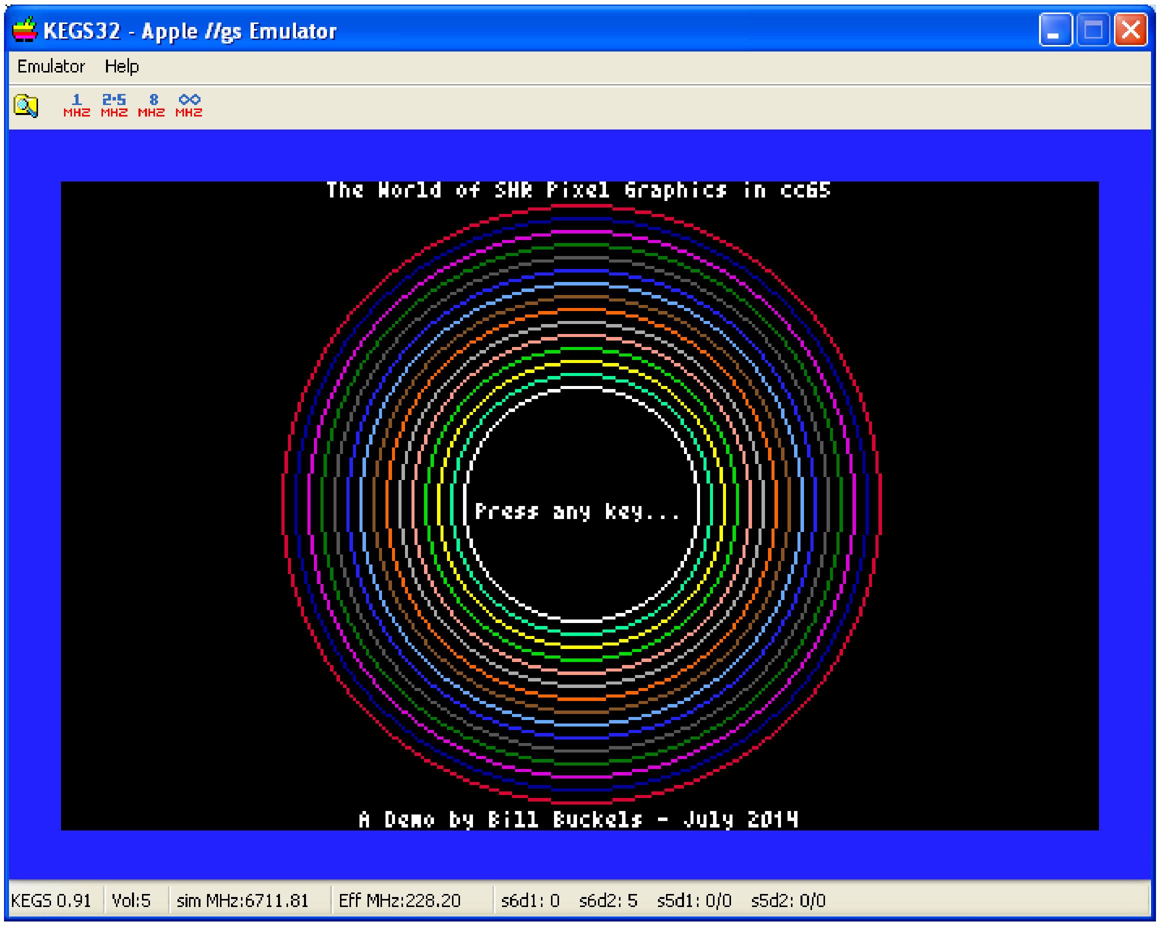
The code below is pretty self-explanatory:

```
/* Bresenham Algorithm line drawing routine for SHR mode320 */
void line320(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) {
        putpixel320((unsigned)x1,(unsigned)y1,color);
        return;
    }
    /* vertical line */
    if (x1 == x2) {
        vline320((unsigned)x1, (unsigned)y1, (unsigned)y2,
                 color);
        return;
    }
    /* horizontal line */
    if (y1 == y2) {
        hline320((unsigned) y1, (unsigned)x1, (unsigned)x2,
                 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 (;;) {
        putpixel320((unsigned)x1,(unsigned)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;
        }
    }
}
```

Graphics Primitives in SHR

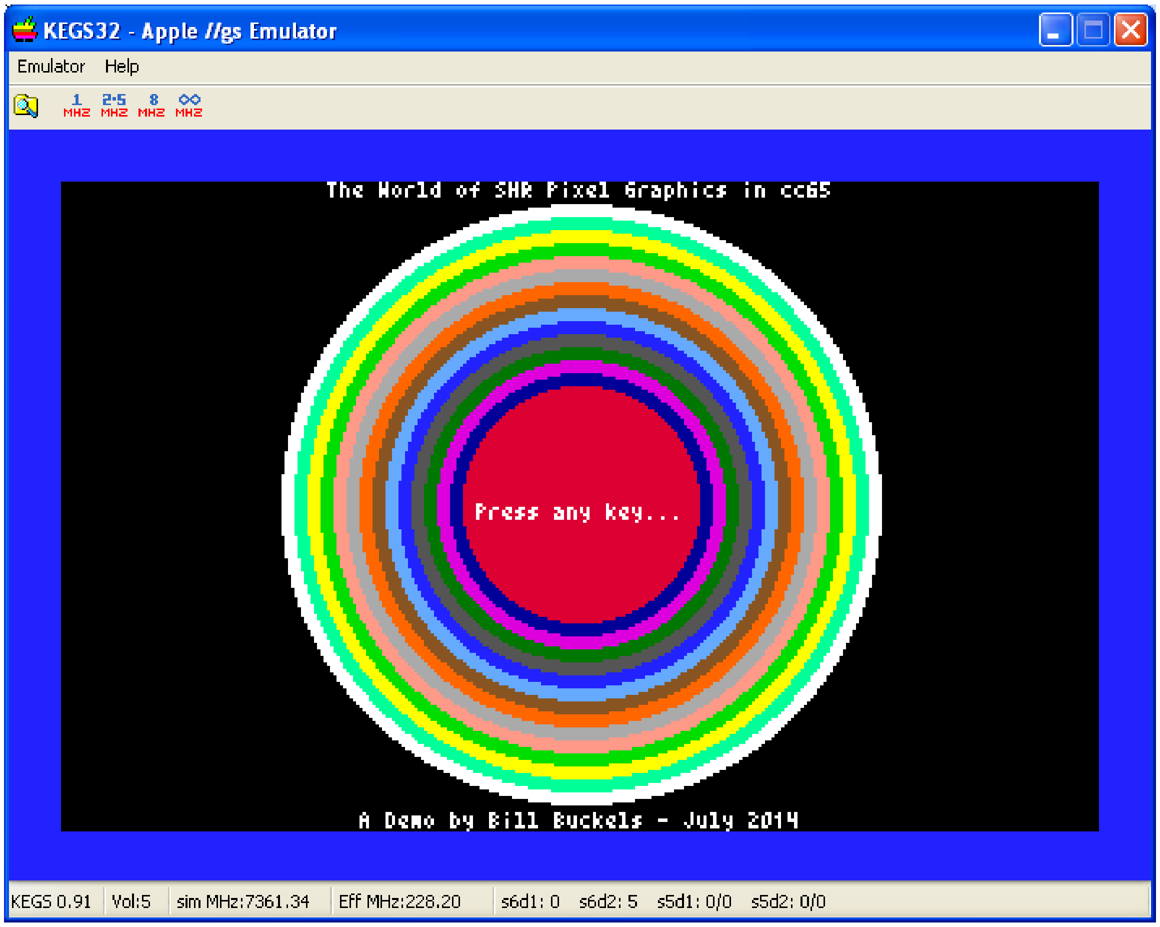
The SHR display is relatively proportional, so graphics primitives like squares and circles don’t require additional calculations for scaling to appear correct.

Circles and Disks


KEGS32 - Apple //gs Emulator
Emulator Help
1 MHz 2.5 MHz 8 MHz ∞ MHz
The World of SHR Pixel Graphics in cc65
Press any key...
A Demo by Bill Buckels - July 2014
KEGS 0.91 Vol:5 sim MHz:6711.81 Eff MHz:228.20 s6d1: 0 s6d2: 5 s5d1: 0/0 s5d2: 0/0

```
/* Aztec C65 circle algorithm, taken from the HGR mode Aztec C65
3.2b version, and originally written circa 1982-83, then modified
for SHR */
void circle320(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;

    putpixel320((unsigned)(c+cx), (unsigned)(y+cy), color);
    putpixel320((unsigned)(d+cx), (unsigned)(x+cy), color);
    putpixel320((unsigned)(-d+cx),(unsigned)(x+cy), color);
    putpixel320((unsigned)(-c+cx),(unsigned)(y+cy), color);
    putpixel320((unsigned)(-c+cx),(unsigned)(-y+cy),color);
    putpixel320((unsigned)(-d+cx),(unsigned)(-x+cy),color);
    putpixel320((unsigned)(d+cx), (unsigned)(-x+cy),color);
    putpixel320((unsigned)(c+cx), (unsigned)(-y+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:;

}
```

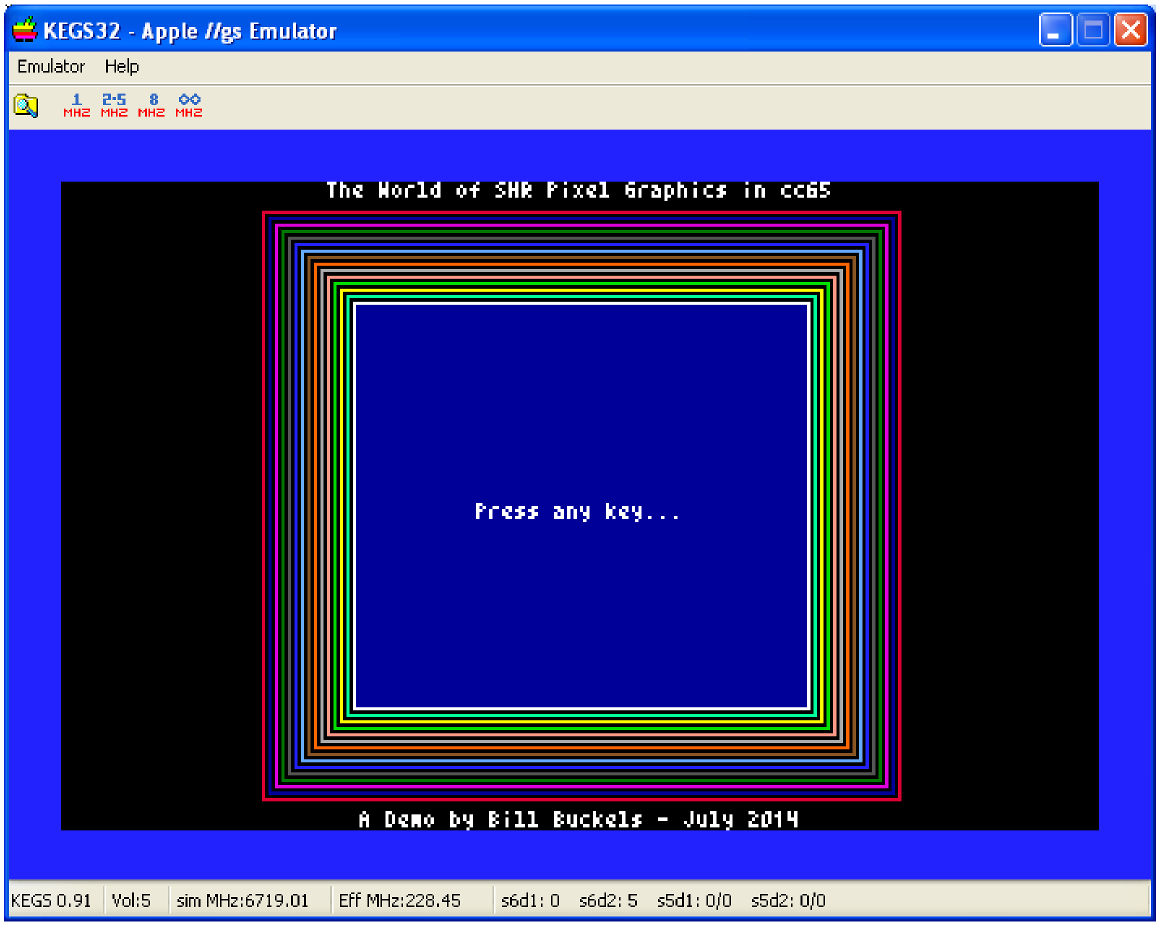
Drawing the outline of a circle is slow because the opportunity for optimization is limited. It is almost as quick to draw a filled disk because unlike drawing a circle outline, double-buffering is not needed for the areas of the disk that can be filled using pixel pairs.


KEGS32 - Apple //gs Emulator
Emulator Help
1 MHz 2.5 MHz 8 MHz ∞ MHz
The World of SHR Pixel Graphics in cc65
Press any key...
A Demo by Bill Buckels - July 2014
KEGS 0.91 Vol:5 sim MHz:7361.34 Eff MHz:228.20 s6d1: 0 s6d2: 5 s5d1: 0/0 s5d2: 0/0

```
/* filled disk */
void disk320(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;

    hline320((unsigned)(y+cy),(unsigned)(-c+cx),
             (unsigned)(c+cx),color);
    hline320((unsigned)(x+cy),(unsigned)(-d+cx),
             (unsigned)(d+cx),color);
    hline320((unsigned)(-y+cy),(unsigned)(-c+cx),
             (unsigned)(c+cx),color);
    hline320((unsigned)(-x+cy),(unsigned)(-d+cx),
             (unsigned)(d+cx),color);

    if(b>= -a)
```

```
            goto fin;

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

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

fin:;

}
```

Boxes

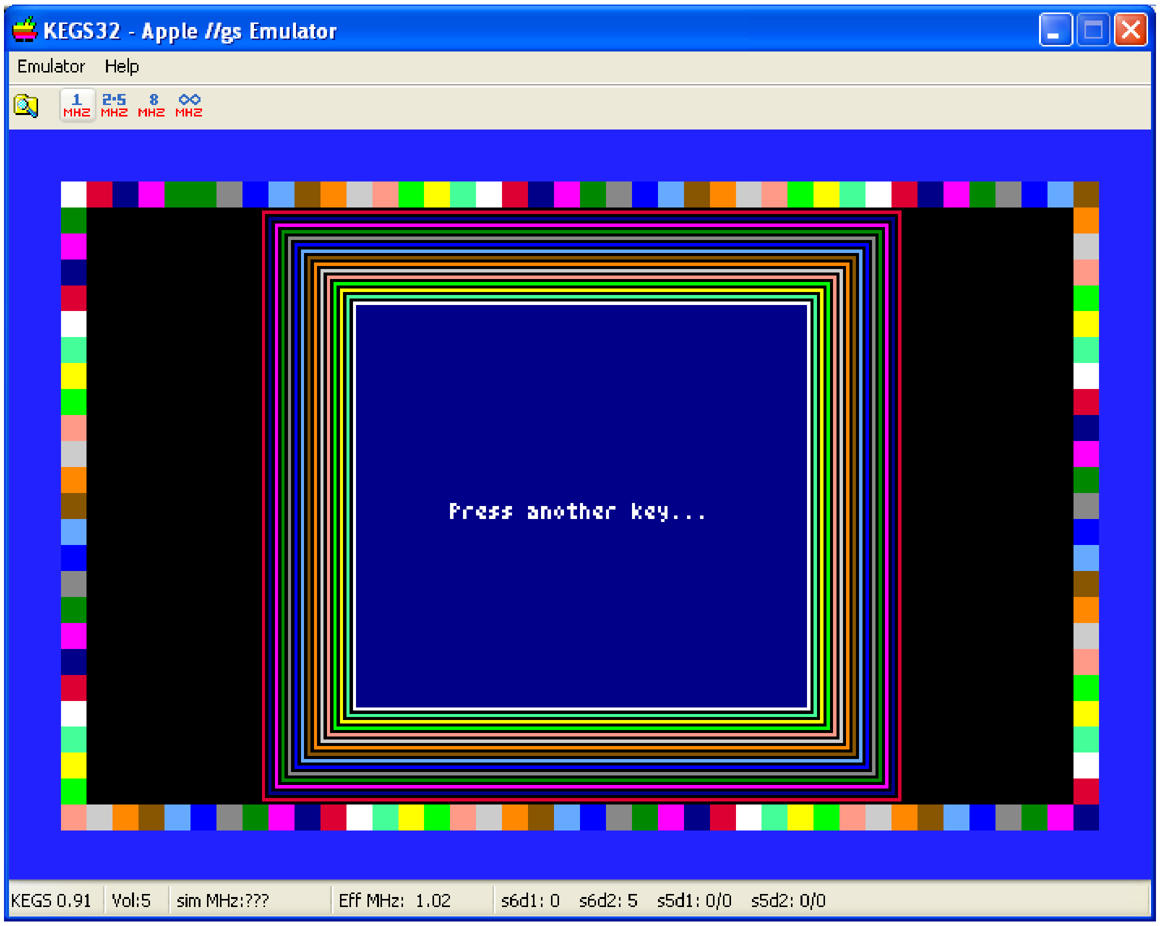

KEGS32 - Apple //gs Emulator
Emulator Help
1 MHz 2.5 MHz 8 MHz ∞ MHz
The World of SHR Pixel Graphics in cc65
Press any key...
A Demo by Bill Buckels - July 2014
KEGS 0.91 Vol:5 sim MHz:6719.01 Eff MHz:228.45 s6d1: 0 s6d2: 5 s5d1: 0/0 s5d2: 0/0

The box drawing routine in shrworld draws both filled and unfilled boxes. Filled boxes can have a different colored fill than the outline color:

```
/* filled or unfilled boxes */
void box320(int x1, int y1, int x2, int y2, unsigned char color,
unsigned char fillcolor)
{
    hline320((unsigned)y1,(unsigned)x1,(unsigned)x2,color);
    y1++;
    while (y1 < y2) {
        putpixel320((unsigned)x1,(unsigned)y1,color);
        if (fillcolor < 16)
            hline320((unsigned)y1,(unsigned)x1+1,
                     (unsigned)x2-1,fillcolor);
        putpixel320((unsigned)x2,(unsigned)y1,color);
        y1++;
    }
    hline320((unsigned)y2,(unsigned)x1,(unsigned)x2,color);
}
```

As graphics primitives go, nothing composed of line elements and points is as simple as a box from the point of view of not doing much math. But a box composed of color blocks can be a little more complicated (see below).

Box Pseudo-Animation


KEGS32 - Apple //gs Emulator
Emulator Help
1 MHz 2.5 MHz 8 MHz ∞ MHz
KEGS 0.91 Vol:5 sim MHz:??? Eff MHz: 1.02 s6d1: 0 s6d2: 5 s5d1: 0/0 s5d2: 0/0

Press another key...

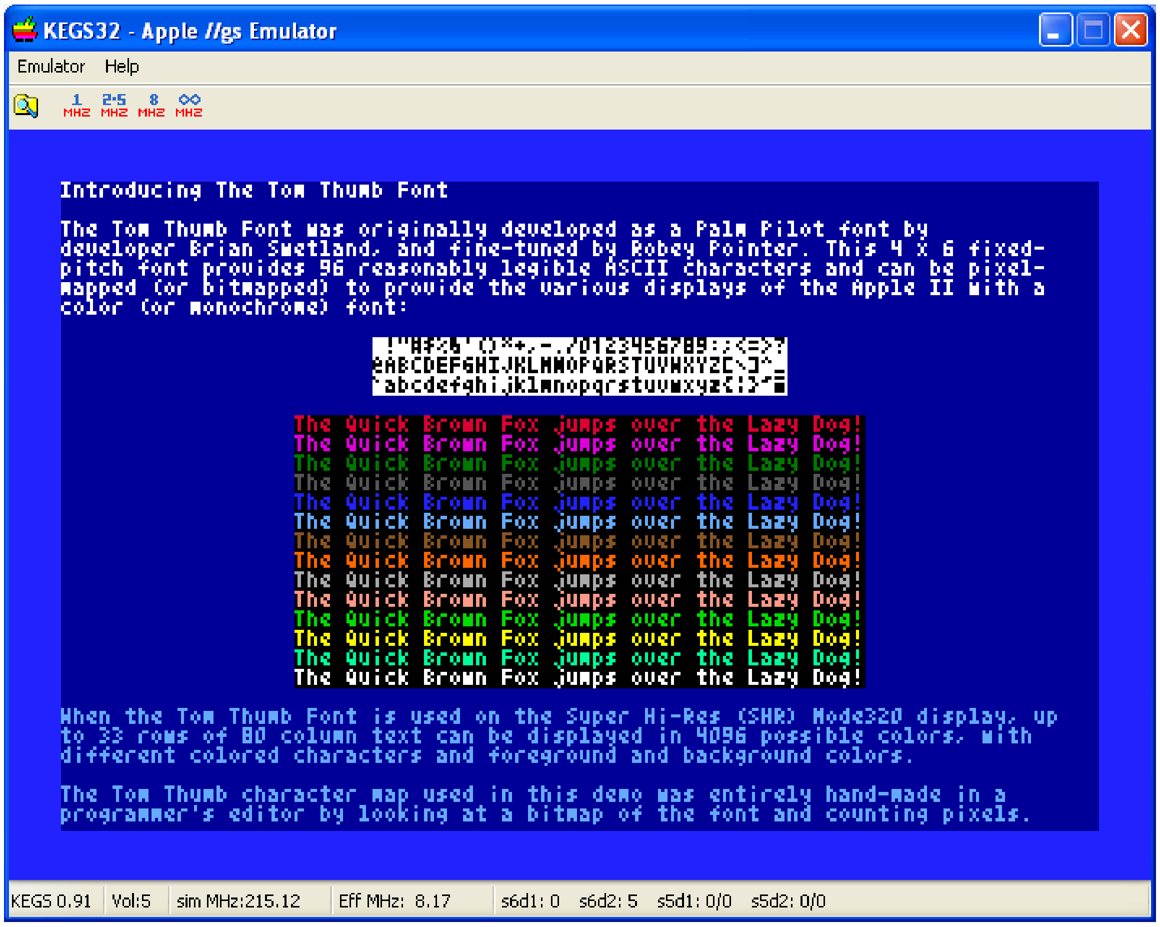
```
unsigned char candyframe320(unsigned char color)
{
    /* uses an 8 x 8 cell - 320 x 200 pixels = 40 x 25 cells*/
    unsigned x,y,i,j;
    unsigned char ch = color;

    for (i = 0,x = 0;i<40;i++,x+=8) {
        for (y = 0;y<8;y++) putpairs320(x,y,4,color);
        color++;
        if (color > 15)color = 1;
    }
    for (i = 1, y=8; i < 24;i++,y+=8) {
        for (j = 0;j<8;j++) putpairs320(312,y+j,4,color);
        color++;
        if (color > 15)color = 1;
    }
    for (i = 0,x = 312;i<40;i++,x-=8) {
        for (y = 0;y<8;y++) putpairs320(x,192+y,4,color);
        color++;
        if (color > 15)color = 1;
    }
    for (i = 23, y=184; i > 0;i--,y-=8) {
        for (j = 0;j<8;j++) putpairs320(0,y+j,4,color);
        color++;
        if (color > 15)color = 1;
    }
    ch++;
    if (ch > 15)ch = 1;
    return ch;
}
```

The code above draws a box frame around the SHR screen starting at the left corner in 8 x 8 repeating blocks of 15 colors; all of shrworld's colors (with the exception of black) from dark red to white in ascending order. When called in a loop in the main program it also increments the starting color by one, creating the effect of a reverse motion of color blocks, which is a little hard to explain (so needs to be seen), but fairly effective at 2.5 MHZ and better at 8 MHZ.

The animation loop in the main program is very simple:

```
/* erase text */
tinyplots("Press another key...",160,99,LOWHITE,LODKBLUE,'M');
/* candy frame demo */
ch = 1;
for (;;) {
    ch = candyframe320(ch);
    if (kbhit() > 0) break;
}
while (kbhit() > 0)cgetc();
```

The "Tom Thumb" Font



```
Introducing The Tom Thumb Font

The Tom Thumb Font was originally developed as a Palm Pilot font by
developer Brian Swetland, and fine-tuned by Robey Pointer. This 4 x 6 fixed-
pitch font provides 96 reasonably legible ASCII characters and can be pixel-
mapped (or bitmapped) to provide the various displays of the Apple II with a
color (or monochrome) font:

                   !"#$%&'()*+,-./0123456789:;<=>?
                  @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
                  `abcdefghijklmnopqrstuvwxyz{|}~█

            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!
            The Quick Brown Fox jumps over the Lazy Dog!

When the Tom Thumb Font is used on the Super Hi-Res (SHR) Mode320 display, up
to 33 rows of 80 column text can be displayed in 4096 possible colors, with
different colored characters and foreground and background colors.

The Tom Thumb character map used in this demo was entirely hand-made in a
programmer's editor by looking at a bitmap of the font and counting pixels.
```

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 :

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~█
```

Brian Swetland's Copyright and Conditions of Use for the original font are in the shrworld 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.

I mentioned earlier in this document that the **bufferpixel320()** function is used to speed-up the plotting of this font. There are actually two routines to plot fonts in shrworld. Both use the Tom Thumb Font. One of them is quicker and uses **bufferpixel320()** to put an entire font line on the screen at once (with a “putimage” effect) and the other uses **putpixel320()** and plots to the screen a pixel at a time, with a “teletype” animation effect which is a little more pleasing to watch and follow along-with even though it is slower:

```
/* 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 plotthumb320(unsigned char ch, unsigned x, unsigned y,
                  unsigned char fg, unsigned char bg)
{
    unsigned offset, x1, x2=x+3, y2=y+6;
    unsigned char 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 > 319) {
               offset++;
               continue;
           }
           byte = tomthumb[offset++];
           if (byte == 0) {
               if (bg > 15) continue;
               putpixel320(x1,y,bg);
           }
           else {
               if (fg > 15) continue;
               putpixel320(x1,y,fg);
           }
        }
        /* if background color is being used then a trailing
           pixel is required between characters */
        if (bg < 16 && x2 < 320)putpixel320(x2,y,bg);
        if (y++ > 199) break;
    }
}
```

```
/* normally spaced 4 x 6 font */
/* using character plotting function plotthumb() (above) */
void thumb320(char *str,unsigned x, unsigned 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) {
     plotthumb320(ch,x,y,fg,bg);
     x+=4;
  }
}

/* functions to plot a font line into buffered memory */
/* faster but not as "fluid" as the font routines above */
void tinychar(unsigned char ch, unsigned x,
              unsigned char fg, unsigned char bg)
{
    unsigned offset, x1, x2=x+3, y, y2=6;
    unsigned char 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);
    for(y = 0; y < 6; y++) {
        for (x1 = x; x1 < x2; x1++) {
           if (x1 > 319) {
               offset++;
               continue;
           }
           byte = tomthumb[offset++];
           if (byte == 0) {
               if (bg > 15) continue;
               bufferpixel320(x1,y,bg);
           }
           else {
               if (fg > 15) continue;
               bufferpixel320(x1,y,fg);
           }
        }
        /* if background color is being used then a trailing
           pixel is required between characters */
        if (bg < 16 && x2 < 320)bufferpixel320(x2,y,bg);
    }
}
```

```
void tinyplots(char *str,unsigned x, unsigned y,
               unsigned char fg,unsigned char bg,
               unsigned char justify)
{
  int target;
  unsigned char ch;
  unsigned src1, buf = (unsigned)&scanline[0];

  if (justify == 'M' || justify == 'm') {
     target = strlen(str);
     if (target == 0) return;
     x-= ((target * 4) /2);

  }
  if (x > 319) x = 0;
  if (y > 194) y = 0;
  /* copy 6 scanlines to scanline buffer */
  src1 = (unsigned) 0x2000 + (y * 160);
  auxtomain(src1,src1+959,buf);
  /* plot 1 font line in 6 buffered scanlines */
  while ((ch = *str++) != 0) {
     tinychar(ch,x,fg,bg);
     x+=4;
  }
  /* copy updated lines back to SHR memory */
  maintoaux(buf,buf+959,src1);
}
```

As you can see, these font routines provide the following features:

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

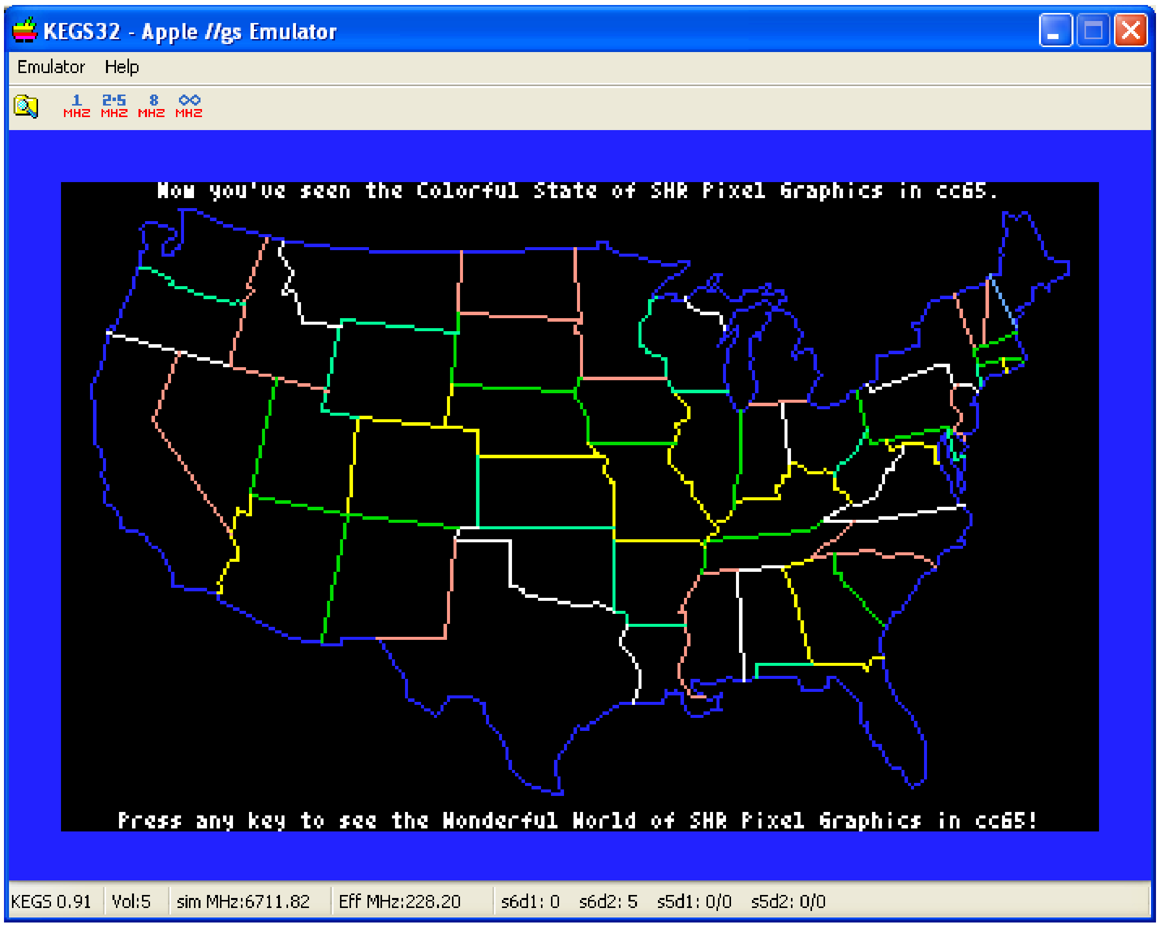
How these font routines are actually used in the demo is covered in the Helper Functions section later in this document.

Program Organization

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

Helper Functions

Line Drawing Scripts



```
Now you've seen the Colorful State of SHR Pixel Graphics in cc65.
Press any key to see the Wonderful World of SHR Pixel Graphics in cc65!
```

Line drawing scripts can be fun, in the same way that Apple II shape tables can be fun. They also provide an alternative image rendering method to bitmapped graphics.

```
#define WORLDMAP  0
#define USAMAP    1
#define USASTATES 2
#define MAPLELEAF 3
```

```
/* mode320 - outline of the USA */
int usa[] = {… -1, -2};
/* mode320 - states of the USA */
int states[] = {… -1, -2};
/* the coordinates to draw the whole world */
int world[] = {… -1, -2};
/* the coordinates to draw the canadian flag */
int MAPLE[] = {… -1, -2};
```

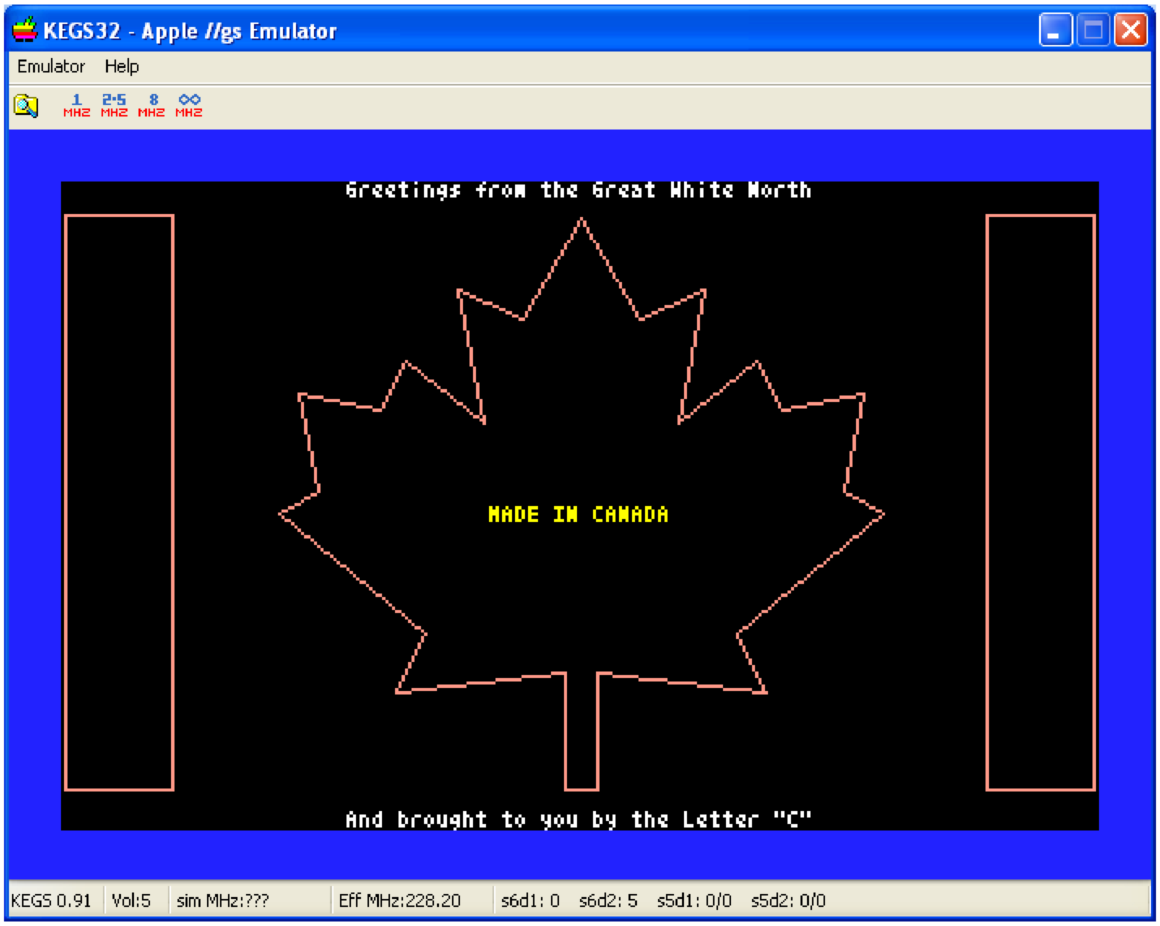
Line drawing scripts of x,y co-ordinates in shrworld are stored as integer arrays of multiple line segments. Each line segment is terminated with x,y co-ordinates of -1,-1 and the script itself is terminated with -1,-2. In the map of the USA displayed above, line elements of several colors are plotted.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

```
/* line drawings script demo routines */
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**:

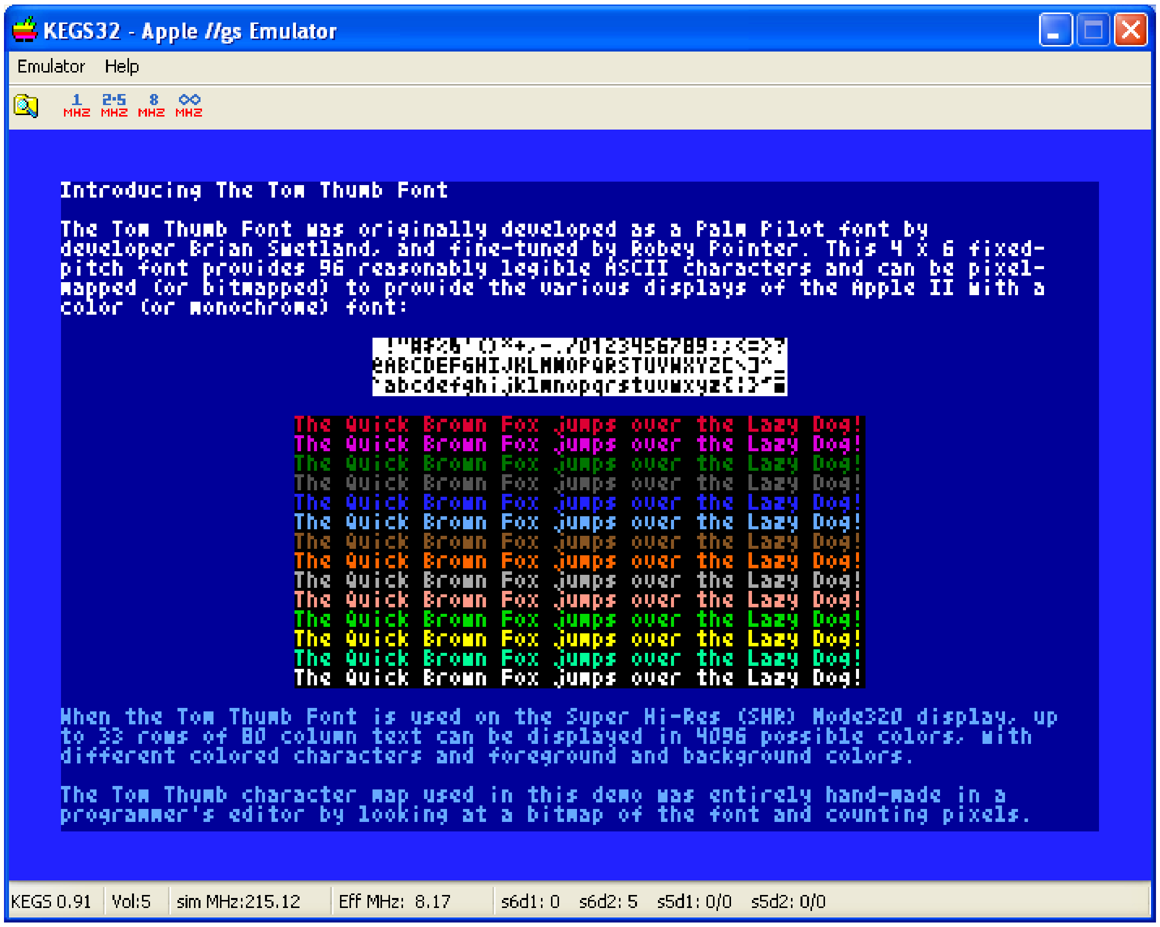
```
void drawmap(int mapidx,int drawcolor)
{
  int *ptr, xidx=0,yidx=1,newx,oldx,newy,oldy,color,cidx=0;

  /* select vector map or vector object */
  switch(mapidx)
  {
      case MAPLELEAF:   ptr = (int *)&maple[0];
                        break;
      case USAMAP:      ptr = (int *)&usa[0];
                        break;
      case USASTATES:   ptr = (int *)&states[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 = colors[cidx];
            cidx++;
      }
      else {
           color = 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 */
            line320(oldx,oldy,newx,newy,color);
            oldx=newx;
            oldy=newy;
         }
         if (newy == -2)break;
         xidx +=2;
         yidx +=2;

    }
}
```


KEGS32 - Apple //gs Emulator
Emulator Help
1 MHz 2.5 MHz 8 MHz ∞ MHz
Greetings from the Great White North
MADE IN CANADA
And brought to you by the Letter "C"
KEGS 0.91 Vol:5 sim MHz:??? Eff MHz:228.20 s6d1: 0 s6d2: 5 s5d1: 0/0 s5d2: 0/0

Tiny Font Demo

```
char *caption1[] = {…,NULL};
char *caption2[] = {…,NULL};
char *fox ="The Quick Brown Fox jumps over the Lazy Dog!";

void TinyFontDemo()
{
    int i;
    unsigned y=0;
    char buf[33];

    clear320((unsigned char)LODKBLUE);
    /* display top caption */
    for (i=0;caption1[i] != NULL;i++) {
        thumb320(caption1[i],0,y,(unsigned char)LOWHITE,255,'L');
        y+=6;
    }
    /* display font characters */
    buf[32] = 0;
    y+=6;
    for (i=0;i<32;i++)buf[i] = (char)i+32;
    tinyplots(buf,160,y,LOBLACK,LOWHITE,'M');
    y+=6;
    for (i=0;i<32;i++)buf[i] = (char)i+64;
    tinyplots(buf,160,y,LOBLACK,LOWHITE,'M');
    y+=6;
    for (i=0;i<32;i++)buf[i] = (char)i+96;
    tinyplots(buf,160,y,LOBLACK,LOWHITE,'M');
    y+=6;y+=6;
    for (i=LORED;i<MAXCOLORS;i++) {
        if (i == LODKBLUE) continue;
        tinyplots(fox,160,y,(unsigned char)i,LOBLACK,'M');
        y+=6;
    }
    /* display bottom caption */
    y+=6;
    for (i=0;caption2[i] != NULL;i++) {
       thumb320(caption2[i],0,y,(unsigned char)LOLTBLUE,255,'L');
       y+=6;
    }
    while (kbhit() > 0)cgetc();
    cgetc();
}
```



```
Introducing The Tom Thumb Font

The Tom Thumb Font was originally developed as a Palm Pilot font by
developer Brian Swetland, and fine-tuned by Robey Pointer. This 4 x 6 fixed-
pitch font provides 96 reasonably legible ASCII characters and can be pixel-
mapped (or bitmapped) to provide the various displays of the Apple II with a
color (or monochrome) font:

                      !"#$%&'()*+,-./0123456789:;<=>?
                     @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
                     `abcdefghijklmnopqrstuvwxyz{|}~

                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!
                  The Quick Brown Fox jumps over the Lazy Dog!

When the Tom Thumb Font is used on the Super Hi-Res (SHR) Mode320 display, up
to 33 rows of 80 column text can be displayed in 4096 possible colors, with
different colored characters and foreground and background colors.

The Tom Thumb character map used in this demo was entirely hand-made in a
programmer's editor by looking at a bitmap of the font and counting pixels.
```

```
KEGS 0.91 | Vol:5 | sim MHz:215.12 | Eff MHz:  8.17 | s6d1: 0   s6d2: 5   s5d1: 0/0   s5d2: 0/0
```

In the code above, two methods of displaying the font are used. These were discussed earlier in the document and you should run the demo to see the difference:

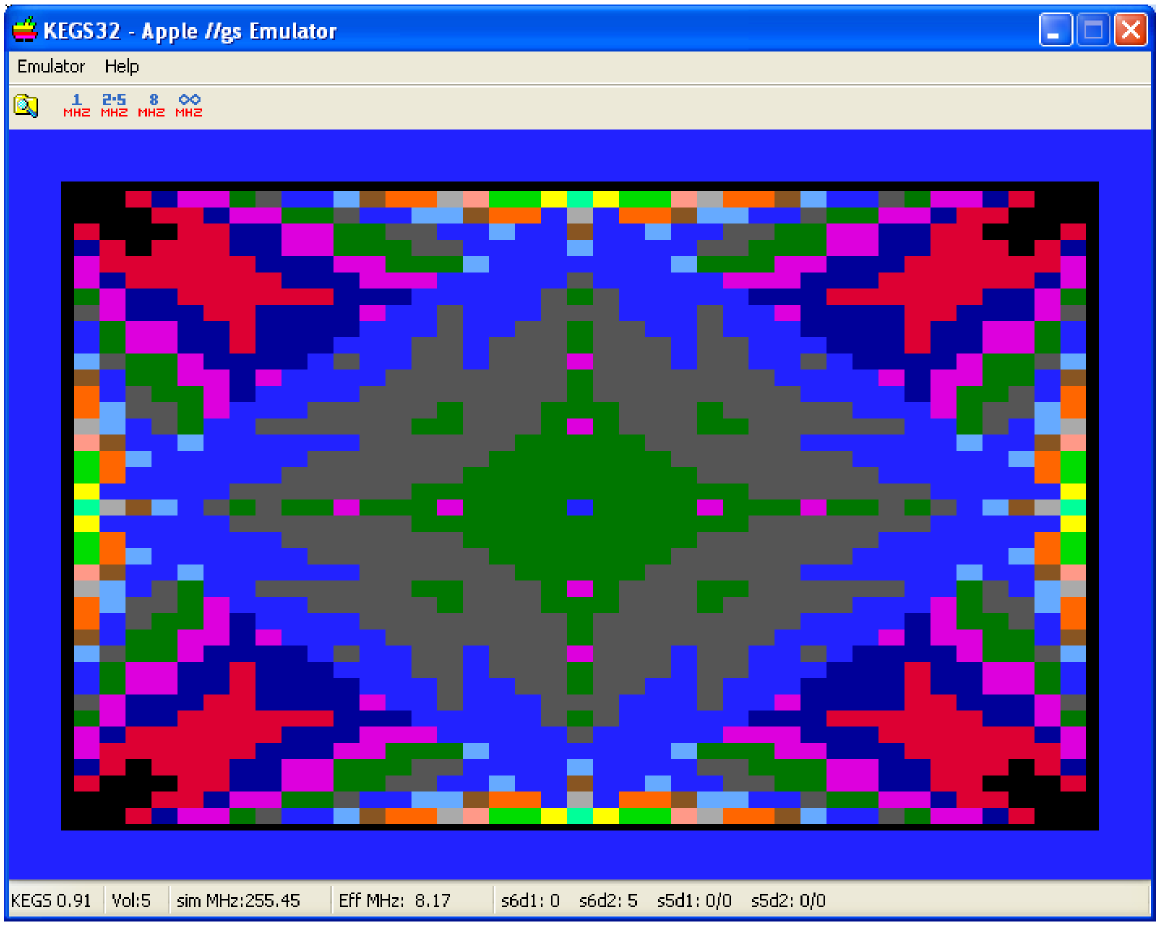
- `thumb320()` Pixel intensive "teletype" effect – good for following along.
- `tinyplots()` Fast Paging "putimage" method – quicker but still a little choppy.

In your own SHR programs, if you wish to present information while the reader follows along, the thumb320() function is a neat way to do so. This "Tom Thumb Font" is "the real thing"… it works great! And you can pack lots of letters on the screen combining them with bitmapped images as well as pixel graphics. SHR bitmapped graphics can be easily created in paint programs. *A little later in this project (but very soon) I will be providing SHR image fragment ("sprite") routines in cc65. These routines will come with a utility for making sprites with a fixed 16 color palette from Windows 24-bit bmp files.*

The technique of "blitting" the font onto the screen using the tinyfont() function is also a neat way to go for showing an "instant" example. Both methods can even be combined on the same line.

The last thing I will say about the font is that it could be used in a terminal emulator.

Kaleidoscope Animation


KEGS32 - Apple //gs Emulator
Emulator Help
1 MHz 2.5 MHz 8 MHz ∞ MHz
KEGS 0.91 Vol:5 sim MHz:255.45 Eff MHz: 8.17 s6d1: 0 s6d2: 5 s5d1: 0/0 s5d2: 0/0

Rod is a well known and well used kaleidoscope algorithm. This "simple but eloquent program" was written for the Apple II's Lo-Res graphics display by Randy Wigginton in 1978. "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." The version of Rod in shrworld is "full-screen", and uses blocks of color instead of pixels. Since shrworld uses the Lo-Res color palette, it was fairly simple to use the original algorithm without much change. You can visit the following link to find-out a little more about Rod:

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

```
/* kaleidoscope demo helper functions */
void rodgi_setcolor(unsigned color)
{
    /* set a buffer of pixel-pairs for block-drawing */
    memset((char *)&scanline[0],(unsigned char)(color << 4 |
color),4);
}
```

```
void rodgi_setcenter()
{
    int y;

    /* draw the center block for the demo */
    for (y = 0; y < 5; y++) putpairs320(156,98+y,4,255);
}

void rodgi_setpixel(unsigned x, unsigned y)
{
    unsigned y2;

    /* draw the blocks for the demo */
    /* the demo was orginally written for single pixels in lgr */
    /* convert to shr from lgr
    40 x 40 to 320 x 200 = 8 pixel x 5 line blocks of 4 bytes */
    x = x * 8;
    y = y * 5;
    y2 = y + 5;
    /* an offset adjustment of 4 pixels and 2 scanlines is
       needed to center the kaleidoscope */
    while (y < y2) {
        putpairs320(x-4,y-2,4,255);
        y++;
    }
}

/* kaleidoscope demo */
int shrod()
{
    unsigned i, j, k, w, fmi, fmk, color;

    clear320(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;
                rodgi_setcolor(color & 0x0f);
                fmi = 40 - i;  fmk = 40 - k;
              rodgi_setpixel(i, k);     rodgi_setpixel(k, i);
              rodgi_setpixel(fmi, fmk); rodgi_setpixel(fmk, fmi);
              rodgi_setpixel(k, fmi);   rodgi_setpixel(fmi, k);
              rodgi_setpixel(i, fmk);   rodgi_setpixel(fmk, i);
                if (kbhit() > 0) return (int) cgetc();
            }
        }
        rodgi_setcenter();
    }
    return 0;
}
```

Main Program

```
int main(void)
{
int idx,c,x1,x2,y1,y2;
unsigned char *ptr, ch;

/* initialize text mode. stay in 40 column mode. */
texton();
clrscr();

/* initialize empty palette and scb's */
initbuffers();
puts("Select a 16-color palette:");
puts("1 - Kegs32");
puts("2 - CiderPress");
puts("3 - Old AppleWin");
puts("4 - New AppleWin");
ch = cgetc();
clrscr();

/* turn shr on */
shgron();
/* now that the shr display is in auxiliary memory */
/* clear the palette and point all the scbs to the
first palette */
clearpalette();
clearscbs();
/* clear the shr screen to blue before setting the palette...
it will clear to black initially */
clear320((unsigned char)LODKBLUE);
/* set the palette that the user has selected
and the screen will change to blue */
switch(ch) {
    case '1': ptr = (unsigned char *)&rgbkegs32[0];break;
    case '2': ptr = (unsigned char *)&rgbciderpress[0];break;
    case '3': ptr = (unsigned char *)&rgbawinold[0];break;
    default:  ptr = (unsigned char *)&rgbawinnew[0];break;
}
setpalette(ptr,0);
thumb320("The World of SHR Pixel Graphics in cc65",
160,0,LOWHITE,255,'M');
thumb320("A Demo by Bill Buckels - July 2014",
160,194,LOWHITE,255,'M');
thumb320("Press any key...",
160,100,(unsigned char)LOYELLOW,255,'M');
while (kbhit() > 0)cgetc();
cgetc();

clear320(0);
```

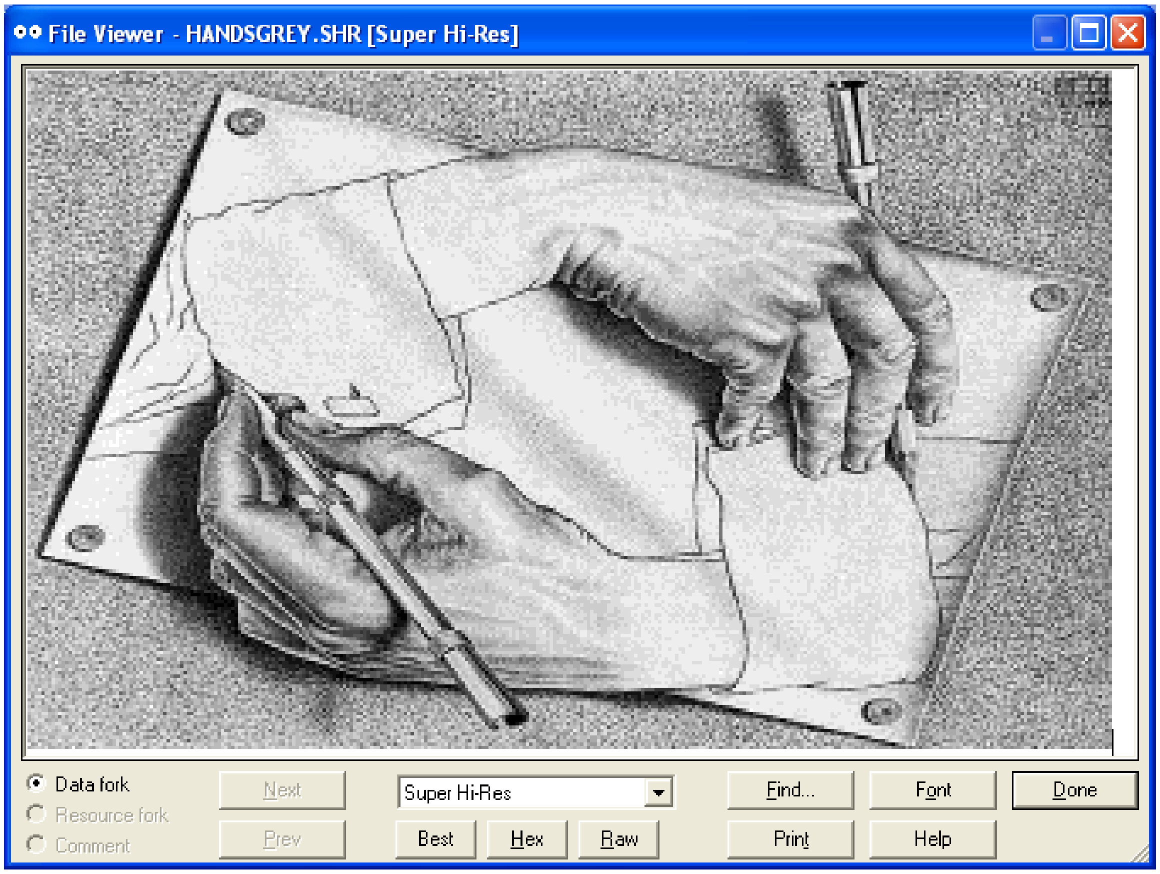
```
/* print demo title */
thumb320("The World of SHR Pixel Graphics in cc65",
160,0,LOWHITE,255,'M');
thumb320("A Demo by Bill Buckels - July 2014",
160,194,LOWHITE,255,'M');

/* circle demo */
for (idx = LORED; idx < MAXCOLORS; idx++) {
    c = (idx * 4);
    circle320(160,99, 96-c, (unsigned char)idx);
}
while (kbhit() > 0)cgetc();
thumb320("Press any key...",160,99,LOWHITE,255,'M');
cgetc();

clear320(0);
/* print demo title */
tinyplots("The World of SHR Pixel Graphics in cc65",
160,0,LOWHITE,255,'M');
tinyplots("A Demo by Bill Buckels - July 2014",
160,194,LOWHITE,255,'M');

/* filled circle demo */
for (idx = 1; idx < 16; idx++) {
    c = (idx * 4);
    disk320(160,99, 96-c, (unsigned char)(MAXCOLORS - idx));
}
while (kbhit() > 0)cgetc();
tinyplots("Press any key...",160,99,LOWHITE,255,'M');
cgetc();

clear320(0);
/* print demo title */
tinyplots("The World of SHR Pixel Graphics in cc65",
160,0,LOWHITE,255,'M');
tinyplots("A Demo by Bill Buckels - July 2014",
160,194,LOWHITE,255,'M');

/* box demo */
y1 = 7;
y2 = 192;
x1 = 60;
x2 = 260;
ch = 255;
for (idx = LORED; idx < MAXCOLORS; idx++) {
c = (idx * 2);
/* set fillcolor for white box */
if (idx == LOWHITE) ch = LODKBLUE;
box320(x1+c,y1+c,x2-c,y2-c,(unsigned char)idx,ch);
}
while (kbhit() > 0)cgetc();
tinyplots("Press any key...",160,99,LOWHITE,255,'M');
```

```
cgetc();

/* erase text */
tinyplots("Press another key...",160,99,LOWHITE,LODKBLUE,'M');
/* candy frame demo */
ch = 1;
for (;;) {
ch = candyframe320(ch);
if (kbhit() > 0) break;
}
while (kbhit() > 0)cgetc();

/* Tom Thumb Font screen */
TinyFontDemo();

/* first line script screen */
clear320(0);
drawmap(USAMAP,LOMEDBLUE);
drawmap(USASTATES,-1);
thumb320("Now you've seen the Colorful State of SHR Pixel
Graphics in cc65.",160,0,LOWHITE,255,'M');
thumb320("Press any key to see the Wonderful World of SHR Pixel
Graphics in cc65!",160,194,LOWHITE,255,'M');
while (kbhit() > 0)cgetc();
cgetc();

/* second line script screen */
clear320((unsigned char)LODKBLUE);
drawmap(WORLDMAP,LOPINK);
thumb320("The World of SHR Pixel Graphics in cc65",
160,0,LOWHITE,255,'M');
thumb320("A Demo by Bill Buckels - July 2014",
160,194,LOWHITE,255,'M');
thumb320("Press any key...",160,100,(unsigned
char)LOYELLOW,255,'M');
while (kbhit() > 0)cgetc();
cgetc();

/* third line script screen */
clear320(0);
drawmap(MAPLELEAF,LOPINK);
tinyplots("Greetings from the Great White North",
160,0,LOWHITE,255,'M');
tinyplots("MADE IN CANADA",160,100,LOYELLOW,255,'M');
tinyplots("And brought to you by the Letter \"C\"",
160,194,LOWHITE,255,'M');
while (kbhit() > 0)cgetc();
cgetc();

/* kaleidoscope demo */
while (shrod()!=27);
```

```
/* clear the palette and scbs then
turn shroff and re-initialize text mode */
clearpalette();
clearscbs();
shgroff();
texton();
clrscr();

/* and exit */
return 0;
}
```

Well, dlodemo’s main program certainly looks simple.

Additional Notes



So there you have it! cc65 now supports SHR pixel graphics, but then it always did, just not directly through its tgi driver scheme. Hopefully you gained something from all of this. But as I have said before, bitmapped demos are sexier than pixel graphics demos like this one. There is no doubt about that either.

Bill Buckels
bbuckels@mts.net

Links to my Apple II Demos in cc65

The shrworld program 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:

SHR	**Bit-Mapped Graphics**
Demo	**Image Loader: http://www.appleoldies.ca/cc65/programs/shr/piclode.zip**
Demo	**SlideShow: http://www.appleoldies.ca/cc65/programs/shr/picshow.zip**
SHR	**Pixel Graphics**
Demo **Doc**	**http://www.appleoldies.ca/cc65/programs/shr/shrworld.zip** **http://www.appleoldies.ca/cc65/docs/shr/shrworld.pdf**
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**
Other	**Miscellaneous (not Graphics)**
Demo	**No-Slot Clock:** **http://www.appleoldies.ca/cc65/programs/REALTIME.zip**

My Apple II Graphics Converters

http://www.appleoldies.ca/graphics/index.htm

These converters were written for the Aztec C65 compiler and not cc65. The bias in the documentation is therefore in favour of Aztec C65.