DAVID BRACKEEN

256-Color VGA Programming in C

A five-part tutorial on VGA programming for the DOS operating system.

Quick Start

1. Download the DOS emulator, DOSBox.
2. Download the DOS C/C++ compiler, DJGPP 2.0.
3. Start the tutorial. For help, see the Troubleshooting page.

About this tutorial

David Brackeen wrote this tutorial for a Technical Writing course in 1996. Although the subject of VGA programming is out of date, this tutorial is still useful for teaching computer graphics, programming old-school DOS games, and developing hobbyist operating systems.

Disclaimer from the author: this material is more than ten years old and is not my best work. Some of the text could be worded differently for clarity and accuracy. Also, the code samples are not high quality, and the diagrams are often muddy or confusing.

Good luck, and have fun!

DAVID BRACKEEN

256-Color VGA Programming in C

Introduction

Contents in this section:

Who this tutorial is for

This tutorial covers many topics in VGA programming in the C programming language. Users of this tutorial should have a comprehensive understanding of C and should also have a familiarity with DOS and BIOS function calls and interrupts.

A general knowledge of trigonometry and/or geometry would be helpful in the second section, Primitive Shapes & Lines

As with any sort of programming, an understanding of the hexadecimal number system would be helpful.

Materials needed

All the code in this tutorial was complied using Borland C/C++ 3.1 and DJGPP 2.0. The code was made to be as portable as possible, sticking close to the ANSI C standard, except for the DOS function calls and direct memory access. The programs should compile without a problem with the DOS 16-bit compilers Microsoft C (not Visual C++), Turbo C, and Borland C. For other compilers, see the Troubleshooting page.

The programs require DOS running on a 286 or better computer with a VGA or better video card. You can emulate this environment on almost any type of hardware by using the DOSBox emualtor.

Document Syntax

Text in `monospace type` is a program, a program segment, a DOS file name, or refers to a variable in a program or program segment.

Text in *italics* is a variable, usually in a formula.

Hexadecimal numbers will have an “0x” prepended to them, like 0x3CF.

Next: VGA Basics »

DAVID BRACKEEN

256-Color VGA Programming in C

VGA Basics

Contents in this section:

What is VGA?

VGA stands for Video Graphics Array, sometimes referred to as Video Graphics Adapter. It is a video card, which is an interface between a computer and its corresponding monitor. The VGA card is the most common video card — nearly every video card has VGA compatability — and it is fairly easy to program. It offers many different video modes, from 2 color to 256 color, and resolutions from 320x200 to 640x480. This tutorial pays close attention to the VGA's only 256-color mode, known as mode 0x13.

Structure of mode 0x13

In mode 0x13, the screen dimensions are 320 pixels in width and 200 pixels in height. This is mapped 0 to 319 on the *x* axis and 0 to 199 on the *y* axis, with the origin (0,0) at the top-left corner (Figure 1). Since this is a 256-color mode, each pixel represents 8 bits (2^8=256) or one byte, so the memory needed is 320*200 or 64,000 bytes.

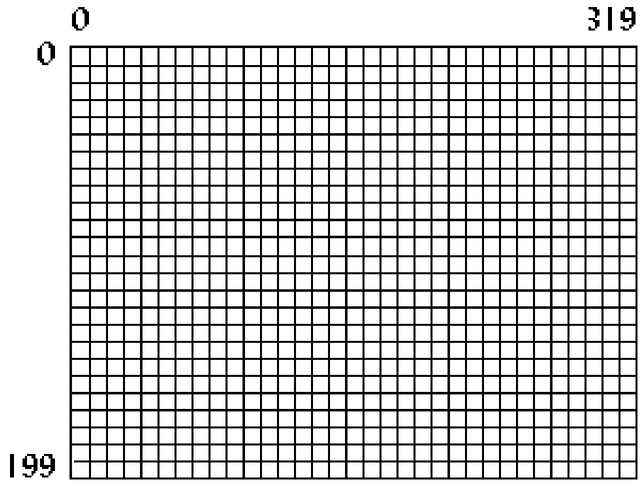

Figure 1. Structure of Mode 0x13.

Setting the video mode

To set the video mode, call interrupt 0x10 (BIOS video functions) with 0 (zero) in the AH register and the desired mode number in the AL register. For mode 0x13, the code (Borland C) would be as follows:

```
union REGS regs;

regs.h.ah = 0x00;  /* function 00h = mode set */
regs.h.al = 0x13;  /* 256-color */
int86(0x10,&regs,&regs); /* do it! */
```

To return to text mode after the program finishes, simply set the mode number to 3.

```
union REGS regs;

regs.h.ah = 0x00;
regs.h.al = 0x03; /* text mode is mode 3 */
int86(0x10,&regs,&regs);
```

Plotting a pixel

An easy way to plot a pixel is by using function 0x0C under BIOS interrupt 0x10. For this function, set CX and DX to the pixel *x* and *y* location. The color displayed depends on the value in AL. See Table I for a list of common colors.

```
union REGS regs;

regs.h.ah = 0x0C;   /* function 0Ch = pixel plot */
regs.h.al = color;
regs.x.cx = x;      /* x location, from 0..319  */
regs.x.dx = y;      /* y location, from 0..199  */
int86(0x10,&regs,&regs);
```

This pixel-plotting method is easy, but it is also very slow. BIOS will do certain checks to make sure that the input is valid, and then it will test to see if the (*x*,*y*) coordinates are within the screen boundaries, and finally it will calculate the offset to video memory. A faster way to plot a pixel is to write directly to video memory.

Mode 0x13 memory

As mentioned before, the memory needed for mode 0x13 is 64,000 bytes. This memory is located at segment 0xA000 in the computer's memory. Simply writing to that area in memory will also write to the screen. The color displayed depends on the byte value written to memory.

Value	**Color**
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	Light Gray
8	Dark Gray
9	Light Blue
10	Light Green
11	Light Cyan
12	Light Red
13	Light Magenta
14	Yellow
15	White

Table I. The first 16 VGA colors.

Since memory is linear (unlike the computer screen, which has both an *x* and a *y* dimension), the offset into computer memory must be calculated to plot a pixel. To do this the *y* value is multiplied by the width of the screen, or 320, and the *x* value is added to that. Thus to plot a pixel at location (256,8), first calculate 256+8*320=2816 or 0xB00, then write to segment 0xA000, offset 0xB00. The following program segment creates a pointer to address 0xA000:0000, computes the offset from two variables, and then writes to the calculated memory location.

Plotting a pixel quickly

```
typedef unsigned char byte;
byte far *VGA = (byte far*)0xA0000000L;
unsigned short offset;
...
offset = 320*y + x;
VGA[offset] = color;
```

The previous code has the following characteristics:

- The variable `offset` must be an `unsigned short` data type (16 bits with a range from 0 to 65,535) because the size of memory needed for mode 0x13 is 64,000 bytes. Using an `unsigned short` data type helps insure that we won't accidently write to an area of memory that isn't part of the video memory, which might cause our program to crash.
- If `y` were 5 and `x` were 340, the pixel would be displayed at (20,6), since video memory in mode 0x13 is linear, and the width of the screen is only 320. The BIOS function would not display a pixel on the screen for (340,5) since it clips to the screen boundaries.
- The pointer to the VGA memory segment must be `far` when compiled in the smaller memory modules. If the memory module used is COMPACT, LARGE, or HUGE, then the `far` keyword can be removed.
- In most DOS extenders, the 32-bit protected mode pointer to video memory would be `0xA0000` instead of the segment-offset pointer `0xA0000000L`.

Program: pixel.c

The following program demonstrates how much faster writing directly to video memory is. It plots 50,000 pixels using BIOS, then does the same by writing directly to video memory, and then displays the results.

DJGPP 2.0
    View pixel.c
    Download pixel.zip (Contains pixel.c, pixel.exe)
Borland C, Turbo C, etc.
    View pixel.c
    Download pixel.zip (Contains pixel.c, pixel.exe)

Having trouble compiling or running the program? See the Troubleshooting page.

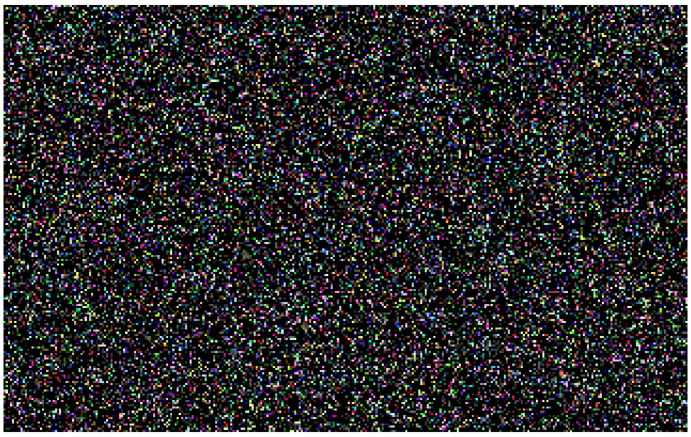
Figure 2. Screenshot of `pixel.exe`.

This program, along with all other programs in this tutorial, ran on a 486dx 33Mhz with 8MB of memory, 128KB cache, and a 16-bit ISA SVGA card. The results from `pixel.exe` on this computer were as follows:

```
Slow pixel plotting took 3.846154 seconds.
Fast pixel plotting took 0.989011 seconds.
Fast pixel plotting was 3.888889 times faster.
```

Shifting

A way to further speed up pixel plotting is to use shifting instead of multiplication when calculating the offset. Shifting a number to the left means to move all the bits in the number to the left, which produces the effect of multiplying by two (Figure 3).

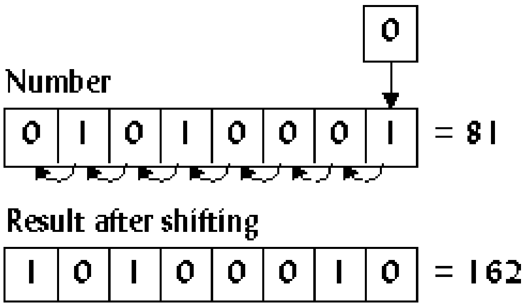

Figure 3. Shifting a number to the left.

If a number *n* was shifted to the left three times, the result would be 2^3n or 8*n*. In C, this is done by using the << operator:

```
a = 6<<3; /* same as 6*8 */
```

To multiply by 320, which is not a power of two, break the number down into powers of two: 256 and 64. For example,

$320y = 256y + 64y$,

so calculate the offset as follows:

```
offset = (y<<8) + (y<<6) + x;
```

The next section, which deals with basic drawing elements, uses this technique frequently.

« Previous: Introduction
Next: Primitive Shapes & Lines »

DAVID BRACKEEN

256-Color VGA Programming in C > VGA Basics

pixel.c

View as plain text

```
/**************************************************************************
 * pixel.c                                                                *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * Tab stops are set to 2.                                                *
 * This program compiles with DJGPP! (www.delorie.com)                    *
 * To compile in DJGPP: gcc pixel.c -o pixel.exe                          *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA, or compatible video adapter.                            *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates how much faster writing directly *
 * to video memory is.                                                    *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <sys/nearptr.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define WRITE_DOT           0x0C      /* BIOS func to plot a pixel. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

typedef unsigned char  byte;
typedef unsigned short word;


byte *VGA = (byte *)0xA0000;          /* this points to video memory. */
word *my_clock = (word *)0x046C;      /* this points to the 18.2hz system
                                         clock. */

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  plot_pixel_slow                                                       *
 *    Plot a pixel by using BIOS function 0x0C (Write Dot).               *
 **************************************************************************/

void plot_pixel_slow(int x,int y,byte color)
{
  union REGS regs;

  regs.h.ah = WRITE_DOT;
  regs.h.al = color;
  regs.x.cx = x;
  regs.x.dx = y;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  plot_pixel_fast                                                       *
 *    Plot a pixel by directly writing to video memory.                   *
 **************************************************************************/

void plot_pixel_fast(int x,int y,byte color)
{
  VGA[y*SCREEN_WIDTH+x]=color;
}

/**************************************************************************
 *  Main                                                                  *
 *    Plots 50000 pixels two different ways: using the BIOS and by        *
 *    directly writing to video memory.                                   *
 **************************************************************************/

void main()
{
  int x,y,color;
  float t1,t2;
  word i,start;

  if (__djgpp_nearptr_enable() == 0)
  {
    printf("Could get access to first 640K of memory.\n");
    exit(-1);
  }

  VGA+=__djgpp_conventional_base;
  my_clock = (void *)my_clock + __djgpp_conventional_base;
  srand(*my_clock);                   /* seed the number generator. */
  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<50000L;i++)               /* randomly plot 50000 pixels. */
  {
    x=rand()%SCREEN_WIDTH;
    y=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    plot_pixel_slow(x,y,color);
  }

  t1=(*my_clock-start)/18.2;          /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                         to clear the screen. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<50000L;i++)               /* randomly plot 50000 pixels. */
  {
    x=rand()%SCREEN_WIDTH;
    y=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    plot_pixel_fast(x,y,color);
  }

  t2=(*my_clock-start)/18.2;          /* calculate how long it took. */
  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */

  /* output the results... */
  printf("Slow pixel plotting took %f seconds.\n",t1);
  printf("Fast pixel plotting took %f seconds.\n",t2);
  if (t2 != 0) printf("Fast pixel plotting was %f times faster.\n",t1/t2);

  __djgpp_nearptr_disable();

  return;
}
```

« Back to VGA Basics

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

256-Color VGA Programming in C > VGA Basics

pixel.c

View as plain text

```
/**************************************************************************
 * pixel.c                                                                *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * This is a 16-bit program.                                              *
 * Tab stops are set to 2.                                                *
 * Remember to compile in the LARGE memory model!                         *
 * To compile in Borland C: bcc -ml pixel.c                               *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA or compatible video adapter.                             *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates how much faster writing directly *
 * to video memory is.                                                    *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define WRITE_DOT           0x0C      /* BIOS func to plot a pixel. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

typedef unsigned char  byte;
typedef unsigned short word;


byte *VGA=(byte *)0xA0000000L;        /* this points to video memory. */
word *my_clock=(word *)0x0000046C;    /* this points to the 18.2hz system
                                         clock. */

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  plot_pixel_slow                                                       *
 *    Plot a pixel by using BIOS function 0x0C (Write Dot).               *
 **************************************************************************/

void plot_pixel_slow(int x,int y,byte color)
{
  union REGS regs;

  regs.h.ah = WRITE_DOT;
  regs.h.al = color;
  regs.x.cx = x;
  regs.x.dx = y;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  plot_pixel_fast                                                       *
 *    Plot a pixel by directly writing to video memory.                   *
 **************************************************************************/

void plot_pixel_fast(int x,int y,byte color)
{
  VGA[y*SCREEN_WIDTH+x]=color;
}

/**************************************************************************
 *  Main                                                                  *
 *    Plots 50000 pixels two different ways: using the BIOS and by        *
 *    directly writing to video memory.                                   *
 **************************************************************************/

void main()
{
  int x,y,color;
  float t1,t2;
  word i,start;

  srand(*my_clock);                   /* seed the number generator. */
  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<50000L;i++)               /* randomly plot 50000 pixels. */
  {
    x=rand()%SCREEN_WIDTH;
    y=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    plot_pixel_slow(x,y,color);
  }

  t1=(*my_clock-start)/18.2;          /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                         to clear the screen. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<50000L;i++)               /* randomly plot 50000 pixels. */
  {
    x=rand()%SCREEN_WIDTH;
    y=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    plot_pixel_fast(x,y,color);
  }

  t2=(*my_clock-start)/18.2;          /* calculate how long it took. */
  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */

  /* output the results... */
  printf("Slow pixel plotting took %f seconds.\n",t1);
  printf("Fast pixel plotting took %f seconds.\n",t2);
  if (t2 != 0) printf("Fast pixel plotting was %f times faster.\n",t1/t2);

  return;
}
```

« Back to VGA Basics

DAVID BRACKEEN

256-Color VGA Programming in C

Primitive Shapes & Lines

Contents in this section:

Why shapes and lines?

This section covers three basic drawing elements: lines, rectangles, and circles. Some of the programming techniques in this chapter may not appear to have any clear use, but the information here is a valuable resource and provides a good foundation.

Drawing lines

One problem with line drawing is that the screen is arranged in a grid of horizontal and vertical lines. A line drawn on the screen will cross both horizontal *and* vertical grid lines unless the line itself is either horizontal or vertical (Figure 4).

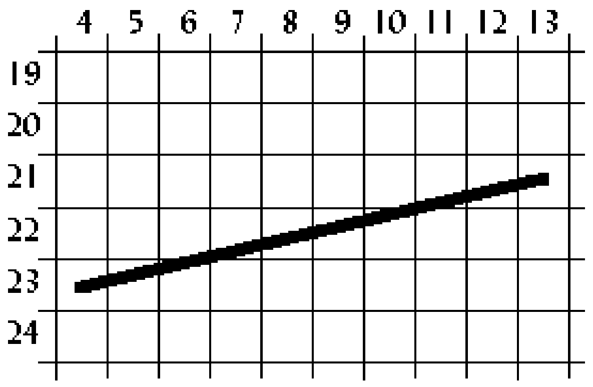

Figure 4. Drawing a line.

Therefore, the line drawn on the screen will not be exact; it will only be a representation of the line. The shaded areas in Figure 5 show the line drawn on a grid.

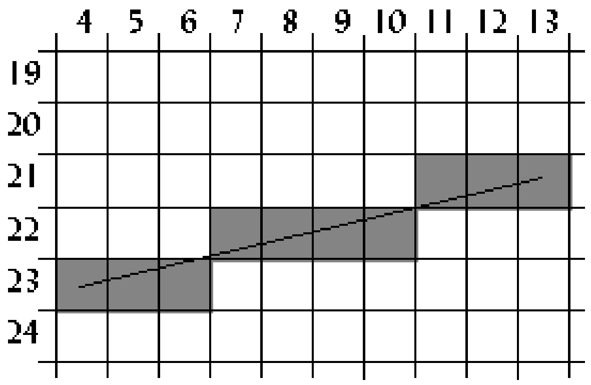

Figure 5. How the line appears on a grid.

One way to draw a line is to first calculate the slope of the line, then plot a pixel at each specified step along the major axis. To do this, we use a form of the point-slope equation of a line, which is

$$y = slope(x - x_1) + y_1 .$$

In the example in Figure 5, the major axis is the *x* axis because the line is more horizontal than vertical. The formula for the slope of a line is

$$slope = \frac{y_2 - y_1}{x_2 - x_1} .$$

Using (4,23) as (x_1, y_1) and (13,21) as (x_2, y_2),

$$slope = \frac{21 - 23}{13 - 4} = -\frac{2}{9} .$$

Plot a pixel for each *x* 4 through 13 inclusive using the point-slope equation of a line. The first pixel plotted in this example is

$$\left(4, -\frac{2}{9} \cdot (4 - 4) + 23\right) ,$$

or (4,23). The second, third and fourth pixels plotted is (5, 22.778), (6,22.556), and (7,22.333), or (5,23), (6,23), (7,22). See Table II for all the *x* and *y* values computed for this line.

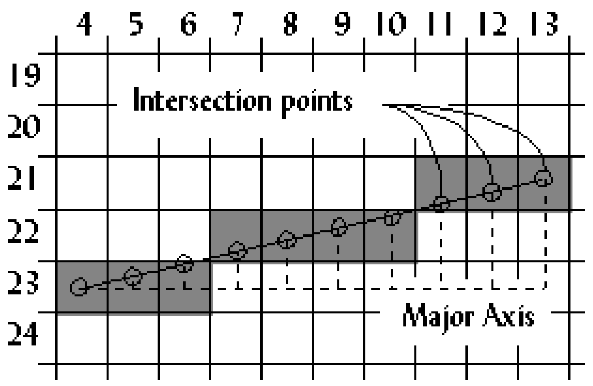

Figure 6. Calculating the pixels plotted.

x	$y = slope(x\text{-}x_1)+y_1$
4	-2/9(4-4) +23 = 23
5	-2/9(5-4) +23 = 22.778 = 23
6	-2/9(6-4) +23 = 22.556 = 23
7	-2/9(7-4) +23 = 22.333 = 22
8	-2/9(8-4) +23 = 22.111 = 22
9	-2/9(9-4) +23 = 21.889 = 22
10	-2/9(10-4) +23 = 21.667 = 22
11	-2/9(11-4) +23 = 21.444 = 21
12	-2/9(12-4) +23 = 21.222 = 21
13	-2/9(13-4) +23 = 21

Table II. Example calculations from the line in Figure 6.

Bresenham’s algorithm

Another way to draw a line is to use Bresenham’s line-drawing algorithm. The previous algorithm derives it, but uses a technique called incremental multiplication and division, which means the algorithm involves no *actual* multiplication or division, only additions or subtractions. An example of incremental multiplication would be computing 5+5+5 instead of 5*3. For incremental division, add the denominator to itself as long as the sum it is less than or equal to the numerator. For $\frac{8}{2}$, 2+2+2+2=8, and since 2 was added to itself 4 times, the answer is 4. For $\frac{11}{4}$, 4+4<11, so the answer is 2 with a remainder of 11-8 or 3.

The following program, which draws 5,000 lines on the screen, gives the complete code of Bresenham's line-drawing algorithm.

Program: lines.c

DJGPP 2.0
    View lines.c
    Download lines.zip (Contains lines.c, lines.exe)
Borland C, Turbo C, etc.
    View lines.c
    Download lines.zip (Contains lines.c, lines.exe)

Having trouble compiling or running the program? See the Troubleshooting page.

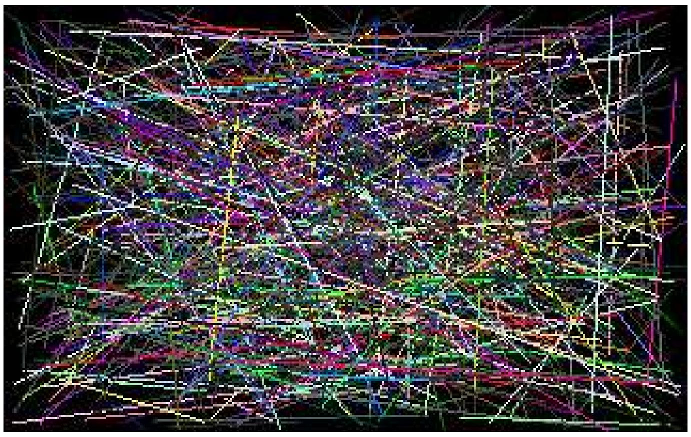
Figure 7. Screenshot of `lines.exe`.

The results from `lines.exe` were as follows:

```
Slow line drawing took 4.285714 seconds.
Fast line drawing took 1.758242 seconds.
Fast line drawing was 2.437500 times faster.
```

The reason Bresenham's line drawing algorithm is faster is that it uses no multiplication or division. Multiplication and division are slow on a computer, even on a computer with a math coprocessor.

Drawing polygons

Using the line-drawing function from the `lines.c` program, a polygon function can easily be created. The following code segment demonstrates this.

```
void polygon(int num_vertices,
             int *vertices,
             byte color)
{
  int i;
  for(i=0;i<num_vertices-1;i++)
  {
    line(vertices[(i<<1)+0],
         vertices[(i<<1)+1],
         vertices[(i<<1)+2],
         vertices[(i<<1)+3],
         color);
  }
  line(vertices[0],
       vertices[1],
       vertices[(num_vertices<<1)-2],
       vertices[(num_vertices<<1)-1],
       color);
}
```

The `polygon` function could be used to draw a triangle as follows:

```
int num_vertices=3;
int vertices[6]={5,0,    /* (x1,y1) */
                 7,5,    /* (x2,y2) */
                 1,4};   /* (x3,y3) */
polygon(3,vertices,15);
```

Drawing rectangles

Although this function is very flexible, it is not suitable for simple shapes like rectangles because rectangles are drawn with horizontal and vertical lines. The line-drawing function is not optimized for drawing those types of lines. Vertical and horizontal line drawing is as simple as plotting a pixel and incrementing the pointer to video memory. The program `rect.c` shows the difference between drawing rectangles using a previously created function and drawing rectangles from scratch. It also illustrates drawing solid rectangles.

Program: rect.c

DJGPP 2.0
    View rect.c
    Download rect.zip (Contains rect.c, rect.exe)
Borland C, Turbo C, etc.
    View rect.c
    Download rect.zip (Contains rect.c, rect.exe)

Having trouble compiling or running the program? See the Troubleshooting page.

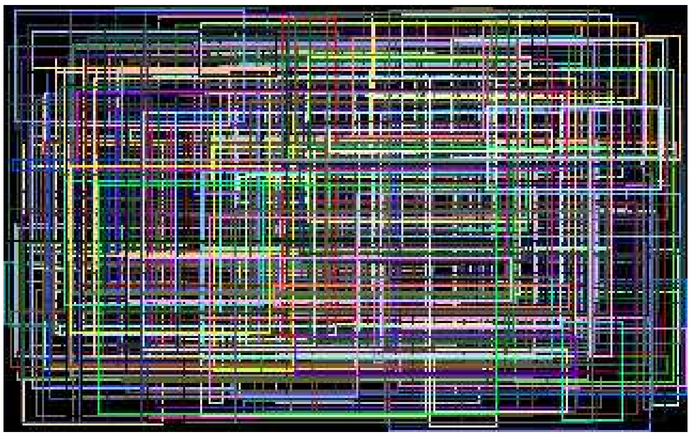
Figure 8. Screenshot of `rect.exe`.

```
Slow rectangle drawing took 4.230769 seconds.
Fast rectangle drawing took 1.153846 seconds.
Fast rectangle drawing was 3.666667 times faster.
```

Rectangle fills are one of more useful things when programming a graphical user interface. Circles, on the other hand, are not as common, but are described in the following sections to help the reader understand some important programming techniques that are used in many different applications.

Using tables to speed up calculations

In some applications, like drawing curves or animation, certain math functions like cosine and sine are used. On problem with these functions is that they are slow because of the time it takes the computer to calculate them. Not only that, but certain angles may be called more than once, like $\sin \frac{\pi}{2}$, so the computer has to calculate them multiple times.

To overcome this problem, tables can be used. When the program starts up, the sine and cosine of every angle is stored in an array:

```
#include <math.h>
...
float COS_TABLE[360], SIN_TABLE[360];
float pi=3.14159;
...
for(i=0;i<360;i++)
{
  COS_TABLE[i]=cos((float)i/180 * pi);
  SIN_TABLE[i]=sin((float)i/180 * pi);
}
```

In this example, the angles are mapped from zero to 359, but they could be mapped in any way. A common mapping is zero to 255 because it fits in one byte. Also, tables are not limited to just sine and cosine functions. The table used in the program `circle.c` is more complex than sine or cosine, and is mapped from 0 to 1023.

Fixed-point math

In many situations, like the one in the previous example, floating point numbers are used. Floating point numbers are very accurate on computers, but are very slow when multiplication or other math functions are used on them.

An alternative to floating point numbers are fixed-point numbers. Fixed point numbers are faster than floating point, but are not as accurate. The accuracy is suitable for most applications involving VGA graphics, however.

Fixed-point numbers are integers with an imaginary decimal point somewhere in the middle of the number. Fixed-point numbers are referenced by the number of bits in the whole part, *w*, and the number of bits in the fraction part, *f*. Thus, a 6:2 fixed point number would have six bits in the whole part and two bits in the fraction part (Figure 9).

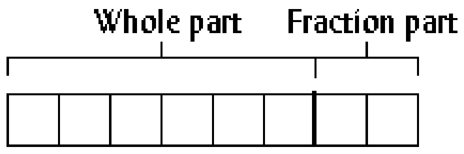

Figure 9. A 6:2 fixed-point number.

To assign floating point values to fixed point numbers, multiply the floating point number by 2^f:

```
unsigned char a,b,c;
...
whole_part=6;
fraction_part=2;
a=14.75 * (1<<fraction_part);
b=32.5  * (1<<fraction_part);
```

To display a fixed point number, divide the fixed point number by 2^f.

```
printf("a=%f",(float)a / (1<<fraction_part));
```

Adding or subtracting fixed point numbers is the same as adding or subtracting two integers. For example, 001110.11 + 100000.10 = 101111.01, or 14.75 + 32.5 = 47.25.

```
c=a+b;
printf("a*b=%f\n",(float)c/4);
```

Multiplying two fixed point numbers is different: both the whole part and the fraction part will double in length. For example, multiplying two 6:2 fixed-point numbers generates a 12:4 number. The solution to this problem is to shift the number right by *f* bits, and ignore the upper *w* bits of the product.

```
c=(a*b) >> fraction_part;
```

The program `circle.c` uses 16:16 fixed-point numbers to increase the speed of drawing circles.

Drawing circles

A simple way to draw a circle is to divide it into octants (Figure 10). First calculate only one octant of the circle; the rest of the circle is “mirrored” from the first octant.

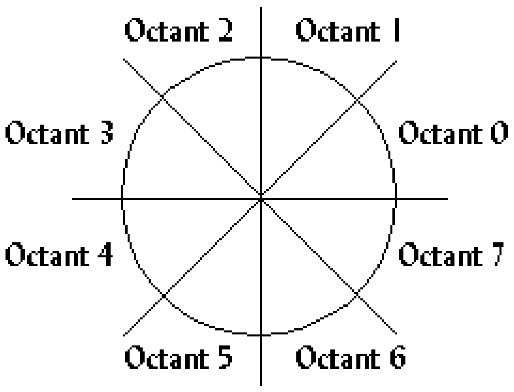

Figure 10. Dividing a circle into octants to reduce computation.

A formula for finding points along a common radius, like that of a circle, is

$(x, y) = (r \cos\ , r \sin\ )$,

where r is the radius of the circle and is the angle at which to plot the point. This formula is changed to

$$\theta = \arccos\frac{x}{r}\ , \quad \theta = \arcsin\frac{y}{r}\ ,$$

which is reduced to

$$y = r \sin\left(\arccos\frac{x}{r}\right).$$

This formula is used to find the y value for an x value and a radius. If the first octant calculated is octant 1, the x value starts at zero and increments, calculating y for every x. The loop finishes when $x>y$ (Figure 11). The rest of the circle is mirrored from octant 1.

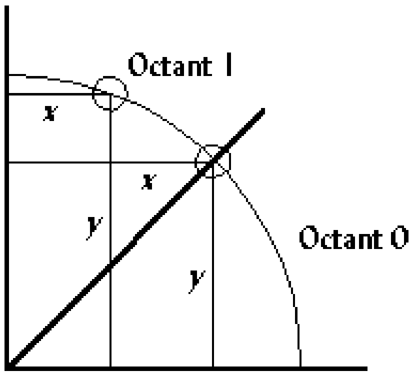

Figure 11. The arc is drawn in Octant 1 for all 0<=*x*<= *y*.

Program: circle.c

This is not the fastest algorithm for drawing circles, but is used in the following program to demonstrate using tables and fixed-point numbers. `circle.c` also demonstrates drawing filled circles.

DJGPP 2.0
  View circle.c
  Download circle.zip (Contains circle.c, circle.exe)
Borland C, Turbo C, etc.
  View circle.c
  Download circle.zip (Contains circle.c, circle.exe)

Having trouble compiling or running the program? See the Troubleshooting page.

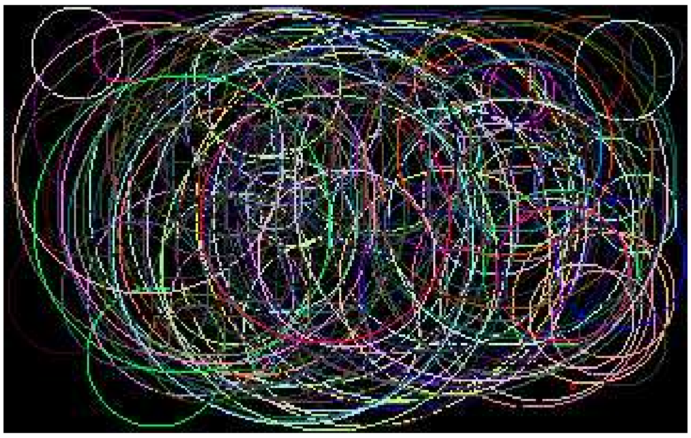

Figure 12. Screenshot of `circle.exe.`

```
Slow circle drawing took 6.868132 seconds.
Fast circle drawing took 1.098901 seconds.
Fast circle drawing was 6.249999 times faster.
```

Something noticeable about the output from `circle.exe` is that the circles do not appear as circles, they appear as ellipses. This is because of the odd aspect ratio of mode 0x13. Instead of a 4:3 aspect ratio, it has an 8:5 aspect ratio, which looks distorted on a screen. To overcome this, a circle's width must be 1.2 times longer than its height. This is also something to consider when drawing bitmaps; the next section covers bitmaps.

« Previous: VGA Basics
Next: Bitmaps & Palette Manipulation »

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

lines.c

View as plain text

```
/**************************************************************************
 * lines.c                                                                *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * Tab stops are set to 2.                                                *
 * This program compiles with DJGPP! (www.delorie.com)                    *
 * To compile in DJGPP: gcc lines.c -o lines.exe                          *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA, or compatible video adapter.                            *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates drawing how much faster it is to *
 * draw lines without using multiplication or division.                   *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <sys/nearptr.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

#define sgn(x) ((x<0)?-1:((x>0)?1:0)) /* macro to return the sign of a
                                         number */
typedef unsigned char  byte;
typedef unsigned short word;

byte *VGA = (byte *)0xA0000;          /* this points to video memory. */
word *my_clock = (word *)0x046C;      /* this points to the 18.2hz system
                                         clock. */

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  plot_pixel                                                            *
 *    Plot a pixel by directly writing to video memory, with no           *
 *    multiplication.                                                     *
 **************************************************************************/

void plot_pixel(int x,int y,byte color)
{
  /*  y*320 = y*256 + y*64 = y*2^8 + y*2^6   */
  VGA[(y<<8)+(y<<6)+x]=color;
}

/**************************************************************************
 *  line_slow                                                             *
 *    draws a line using multiplication and division.                     *
 **************************************************************************/

void line_slow(int x1, int y1, int x2, int y2, byte color)
{
  int dx,dy,sdx,sdy,px,py,dxabs,dyabs,i;
  float slope;

  dx=x2-x1;      /* the horizontal distance of the line */
  dy=y2-y1;      /* the vertical distance of the line */
  dxabs=abs(dx);
  dyabs=abs(dy);
  sdx=sgn(dx);
  sdy=sgn(dy);
  if (dxabs>=dyabs) /* the line is more horizontal than vertical */
  {
    slope=(float)dy / (float)dx;
    for(i=0;i!=dx;i+=sdx)
    {
      px=i+x1;
      py=slope*i+y1;
      plot_pixel(px,py,color);
    }
  }
  else /* the line is more vertical than horizontal */
  {
    slope=(float)dx / (float)dy;
    for(i=0;i!=dy;i+=sdy)
    {
      px=slope*i+x1;
      py=i+y1;
      plot_pixel(px,py,color);
    }
  }
}

/**************************************************************************
 *  line_fast                                                             *
 *    draws a line using Bresenham’s line-drawing algorithm, which uses   *
 *    no multiplication or division.                                      *
 **************************************************************************/

void line_fast(int x1, int y1, int x2, int y2, byte color)
{
  int i,dx,dy,sdx,sdy,dxabs,dyabs,x,y,px,py;

  dx=x2-x1;      /* the horizontal distance of the line */
  dy=y2-y1;      /* the vertical distance of the line */
  dxabs=abs(dx);
  dyabs=abs(dy);
  sdx=sgn(dx);
  sdy=sgn(dy);
  x=dyabs>>1;
  y=dxabs>>1;
  px=x1;
  py=y1;

  VGA[(py<<8)+(py<<6)+px]=color;

  if (dxabs>=dyabs) /* the line is more horizontal than vertical */
  {
    for(i=0;i<dxabs;i++)
    {
      y+=dyabs;
      if (y>=dxabs)
      {
        y-=dxabs;
        py+=sdy;
      }
      px+=sdx;
      plot_pixel(px,py,color);
    }
  }
  else /* the line is more vertical than horizontal */
  {
    for(i=0;i<dyabs;i++)
    {
      x+=dxabs;
      if (x>=dyabs)
      {
        x-=dyabs;
        px+=sdx;
      }
      py+=sdy;
      plot_pixel(px,py,color);
    }
  }
}
```

```
/**************************************************************************
 *  Main                                                                  *
 *    Draws 5000 lines                                                    *
 **************************************************************************/

void main()
{
  int x1,y1,x2,y2,color;
  float t1,t2;
  word i,start;

  if (__djgpp_nearptr_enable() == 0)
  {
    printf("Could get access to first 640K of memory.\n");
    exit(-1);
  }

  VGA+=__djgpp_conventional_base;
  my_clock = (void *)my_clock + __djgpp_conventional_base;

  srand(*my_clock);                   /* seed the number generator. */
  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<5000;i++)                 /* randomly draw 5000 lines. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    line_slow(x1,y1,x2,y2,color);
  }

  t1=(*my_clock-start)/18.2;          /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                         to clear the screen. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<5000;i++)                 /* randomly draw 5000 lines. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    line_fast(x1,y1,x2,y2,color);
  }

  t2=(*my_clock-start)/18.2;          /* calculate how long it took. */
  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */

  /* output the results... */
  printf("Slow line drawing took %f seconds.\n",t1);
  printf("Fast line drawing took %f seconds.\n",t2);
  if (t2 != 0) printf("Fast line drawing was %f times faster.\n",t1/t2);

  __djgpp_nearptr_disable();

  return;
}
```

« Back to Primitive Shapes & Lines

---

lines.c

View as plain text

```
/**************************************************************************
 * lines.c                                                                *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * This is a 16-bit program.                                              *
 * Tab stops are set to 2.                                                *
 * Remember to compile in the LARGE memory model!                         *
 * To compile in Borland C: bcc -ml lines.c                               *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA or compatible video adapter.                             *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates drawing how much faster it is to *
 * draw lines without using multiplication or division.                   *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

#define sgn(x) ((x<0)?-1:((x>0)?1:0)) /* macro to return the sign of a
                                         number */
typedef unsigned char  byte;
typedef unsigned short word;

byte *VGA=(byte *)0xA0000000L;        /* this points to video memory. */
word *my_clock=(word *)0x0000046C;    /* this points to the 18.2hz system
                                         clock. */

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  plot_pixel                                                            *
 *    Plot a pixel by directly writing to video memory, with no           *
 *    multiplication.                                                     *
 **************************************************************************/

void plot_pixel(int x,int y,byte color)
{
  /*  y*320 = y*256 + y*64 = y*2^8 + y*2^6   */
  VGA[(y<<8)+(y<<6)+x]=color;
}

/**************************************************************************
 *  line_slow                                                             *
 *    draws a line using multiplication and division.                     *
 **************************************************************************/

void line_slow(int x1, int y1, int x2, int y2, byte color)
{
  int dx,dy,sdx,sdy,px,py,dxabs,dyabs,i;
  float slope;

  dx=x2-x1;      /* the horizontal distance of the line */
  dy=y2-y1;      /* the vertical distance of the line */
  dxabs=abs(dx);
  dyabs=abs(dy);
  sdx=sgn(dx);
  sdy=sgn(dy);
  if (dxabs>=dyabs) /* the line is more horizontal than vertical */
  {
    slope=(float)dy / (float)dx;
    for(i=0;i!=dx;i+=sdx)
    {
      px=i+x1;
      py=slope*i+y1;
      plot_pixel(px,py,color);
    }
  }
  else /* the line is more vertical than horizontal */
  {
    slope=(float)dx / (float)dy;
    for(i=0;i!=dy;i+=sdy)
    {
      px=slope*i+x1;
      py=i+y1;
      plot_pixel(px,py,color);
    }
  }
}

/**************************************************************************
 *  line_fast                                                             *
 *    draws a line using Bresenham’s line-drawing algorithm, which uses   *
 *    no multiplication or division.                                      *
 **************************************************************************/

void line_fast(int x1, int y1, int x2, int y2, byte color)
{
  int i,dx,dy,sdx,sdy,dxabs,dyabs,x,y,px,py;

  dx=x2-x1;      /* the horizontal distance of the line */
  dy=y2-y1;      /* the vertical distance of the line */
  dxabs=abs(dx);
  dyabs=abs(dy);
  sdx=sgn(dx);
  sdy=sgn(dy);
  x=dyabs>>1;
  y=dxabs>>1;
  px=x1;
  py=y1;

  VGA[(py<<8)+(py<<6)+px]=color;

  if (dxabs>=dyabs) /* the line is more horizontal than vertical */
  {
    for(i=0;i<dxabs;i++)
    {
      y+=dyabs;
      if (y>=dxabs)
      {
        y-=dxabs;
        py+=sdy;
      }
      px+=sdx;
      plot_pixel(px,py,color);
    }
  }
  else /* the line is more vertical than horizontal */
  {
    for(i=0;i<dyabs;i++)
    {
      x+=dxabs;
      if (x>=dyabs)
      {
        x-=dyabs;
        px+=sdx;
      }
      py+=sdy;
      plot_pixel(px,py,color);
    }
  }
}
```

```
/**************************************************************************
 *  Main                                                                  *
 *    Draws 5000 lines                                                    *
 **************************************************************************/

void main()
{
  int x1,y1,x2,y2,color;
  float t1,t2;
  word i,start;

  srand(*my_clock);                   /* seed the number generator. */
  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<5000;i++)                 /* randomly draw 5000 lines. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    line_slow(x1,y1,x2,y2,color);
  }

  t1=(*my_clock-start)/18.2;          /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                         to clear the screen. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<5000;i++)                 /* randomly draw 5000 lines. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    line_fast(x1,y1,x2,y2,color);
  }

  t2=(*my_clock-start)/18.2;          /* calculate how long it took. */
  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */

  /* output the results... */
  printf("Slow line drawing took %f seconds.\n",t1);
  printf("Fast line drawing took %f seconds.\n",t2);
  if (t2 != 0) printf("Fast line drawing was %f times faster.\n",t1/t2);

  return;
}
```

« Back to Primitive Shapes & Lines

DAVID BRACKEEN

256-Color VGA Programming in C > Primitive Shapes & Lines

rect.c

View as plain text

```
/**************************************************************************
 * rect.c                                                                 *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * Tab stops are set to 2.                                                *
 * This program compiles with DJGPP! (www.delorie.com)                    *
 * To compile in DJGPP: gcc rect.c -o rect.exe                            *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA, or compatible video adapter.                            *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates drawing how much faster it is to *
 * draw rectangles without using previously created functions.            *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <sys/nearptr.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

#define sgn(x) ((x<0)?-1:((x>0)?1:0)) /* macro to return the sign of a
                                         number */
typedef unsigned char  byte;
typedef unsigned short word;

byte *VGA = (byte *)0xA0000;          /* this points to video memory. */
word *my_clock = (word *)0x046C;      /* this points to the 18.2hz system
                                         clock. */

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  plot_pixel                                                            *
 *    Plot a pixel by directly writing to video memory, with no           *
 *    multiplication.                                                     *
 **************************************************************************/

void plot_pixel(int x,int y,byte color)
{
  /*  y*320 = y*256 + y*64 = y*2^8 + y*2^6   */
  VGA[(y<<8)+(y<<6)+x]=color;
}

/**************************************************************************
 *  line                                                                  *
 *    draws a line using Bresenham’s line-drawing algorithm, which uses   *
 *    no multiplication or division.                                      *
 **************************************************************************/

void line(int x1, int y1, int x2, int y2, byte color)
{
  int i,dx,dy,sdx,sdy,dxabs,dyabs,x,y,px,py;

  dx=x2-x1;      /* the horizontal distance of the line */
  dy=y2-y1;      /* the vertical distance of the line */
  dxabs=abs(dx);
  dyabs=abs(dy);
  sdx=sgn(dx);
  sdy=sgn(dy);
  x=dyabs>>1;
  y=dxabs>>1;
  px=x1;
  py=y1;

  VGA[(py<<8)+(py<<6)+px]=color;

  if (dxabs>=dyabs) /* the line is more horizontal than vertical */
  {
    for(i=0;i<dxabs;i++)
    {
      y+=dyabs;
      if (y>=dxabs)
      {
        y-=dxabs;
        py+=sdy;
      }
      px+=sdx;
      plot_pixel(px,py,color);
    }
  }
  else /* the line is more vertical than horizontal */
  {
    for(i=0;i<dyabs;i++)
    {
      x+=dxabs;
      if (x>=dyabs)
      {
        x-=dyabs;
        px+=sdx;
      }
      py+=sdy;
      plot_pixel(px,py,color);
    }
  }
}

/**************************************************************************
 *  rect_slow                                                             *
 *    Draws a rectangle by calling the line function four times.          *
 **************************************************************************/

void rect_slow(int left,int top, int right, int bottom, byte color)
{
  line(left,top,right,top,color);
  line(left,top,left,bottom,color);
  line(right,top,right,bottom,color);
  line(left,bottom,right,bottom,color);
}

/**************************************************************************
 *  rect_fast                                                             *
 *    Draws a rectangle by drawing all lines by itself.                   *
 **************************************************************************/

void rect_fast(int left,int top, int right, int bottom, byte color)
{
  word top_offset,bottom_offset,i,temp;

  if (top>bottom)
  {
    temp=top;
    top=bottom;
    bottom=temp;
  }
  if (left>right)
  {
    temp=left;
    left=right;
    right=temp;
  }

  top_offset=(top<<8)+(top<<6);
  bottom_offset=(bottom<<8)+(bottom<<6);
  for(i=left;i<=right;i++)
  {
    VGA[top_offset+i]=color;
    VGA[bottom_offset+i]=color;
  }
  for(i=top_offset;i<=bottom_offset;i+=SCREEN_WIDTH)
  {
    VGA[left+i]=color;
    VGA[right+i]=color;
  }
}

/**************************************************************************
 *  rect_fill                                                             *
 *    Draws and fills a rectangle.                                        *
 **************************************************************************/

void rect_fill(int left,int top, int right, int bottom, byte color)
{
  word top_offset,bottom_offset,i,temp,width;

  if (top>bottom)
  {
    temp=top;
    top=bottom;
    bottom=temp;
  }
  if (left>right)
  {
    temp=left;
    left=right;
    right=temp;
  }

  top_offset=(top<<8)+(top<<6)+left;
  bottom_offset=(bottom<<8)+(bottom<<6)+left;
  width=right-left+1;

  for(i=top_offset;i<=bottom_offset;i+=SCREEN_WIDTH)
  {
    memset(&VGA[i],color,width);
  }
}

/**************************************************************************
 *  Main                                                                  *
 *    Draws 5000 rectangles                                               *
 **************************************************************************/

void main()
{
  int x1,y1,x2,y2,color;
  float t1,t2;
  word i,start;

  if (__djgpp_nearptr_enable() == 0)
  {
    printf("Could get access to first 640K of memory.\n");
    exit(-1);
  }

  VGA+=__djgpp_conventional_base;
  my_clock = (void *)my_clock + __djgpp_conventional_base;

  srand(*my_clock);                    /* seed the number generator. */
  set_mode(VGA_256_COLOR_MODE);        /* set the video mode. */

  start=*my_clock;                     /* record the starting time. */
  for(i=0;i<5000;i++)                  /* randomly draw 5000 rectangles. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    rect_slow(x1,y1,x2,y2,color);
  }

  t1=(*my_clock-start)/18.2;           /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);        /* set the video mode again in order
                                          to clear the screen. */

  start=*my_clock;                     /* record the starting time. */
  for(i=0;i<5000;i++)                  /* randomly draw 5000 rectangles. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    rect_fast(x1,y1,x2,y2,color);
  }

  t2=(*my_clock-start)/18.2;           /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);        /* set the video mode again in order
                                          to clear the screen. */

  for(i=0;i<1000;i++)                  /* randomly draw 1000 filled rects. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    rect_fill(x1,y1,x2,y2,color);
  }

  set_mode(TEXT_MODE);                 /* set the video mode back to
                                          text mode. */

  /* output the results... */
  printf("Slow rectangle drawing took %f seconds.\n",t1);
  printf("Fast rectangle drawing took %f seconds.\n",t2);
  if (t2 != 0) printf("Fast rectangle drawing was %f times faster.\n",t1/t2);

  __djgpp_nearptr_disable();

  return;
}
```

« Back to Primitive Shapes & Lines

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

256-Color VGA Programming in C > Primitive Shapes & Lines

rect.c

View as plain text

```
/**************************************************************************
 * rect.c                                                                 *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * This is a 16-bit program.                                              *
 * Tab stops are set to 2.                                                *
 * Remember to compile in the LARGE memory model!                         *
 * To compile in Borland C: bcc -ml rect.c                                *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA or compatible video adapter.                             *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates drawing how much faster it is to *
 * draw rectangles without using previously created functions.            *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <mem.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

#define sgn(x) ((x<0)?-1:((x>0)?1:0)) /* macro to return the sign of a
                                         number */
typedef unsigned char  byte;
typedef unsigned short word;

byte *VGA=(byte *)0xA0000000L;        /* this points to video memory. */
word *my_clock=(word *)0x0000046C;    /* this points to the 18.2hz system
                                         clock. */

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                                *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  plot_pixel                                                             *
 *    Plot a pixel by directly writing to video memory, with no           *
 *    multiplication.                                                      *
 **************************************************************************/

void plot_pixel(int x,int y,byte color)
{
  /*  y*320 = y*256 + y*64 = y*2^8 + y*2^6   */
  VGA[(y<<8)+(y<<6)+x]=color;
}

/**************************************************************************
 *  line                                                                   *
 *    draws a line using Bresenham's line-drawing algorithm, which uses   *
 *    no multiplication or division.                                       *
 **************************************************************************/

void line(int x1, int y1, int x2, int y2, byte color)
{
  int i,dx,dy,sdx,sdy,dxabs,dyabs,x,y,px,py;

  dx=x2-x1;      /* the horizontal distance of the line */
  dy=y2-y1;      /* the vertical distance of the line */
  dxabs=abs(dx);
  dyabs=abs(dy);
  sdx=sgn(dx);
  sdy=sgn(dy);
  x=dyabs>>1;
  y=dxabs>>1;
  px=x1;
  py=y1;

  VGA[(py<<8)+(py<<6)+px]=color;

  if (dxabs>=dyabs) /* the line is more horizontal than vertical */
  {
    for(i=0;i<dxabs;i++)
    {
      y+=dyabs;
      if (y>=dxabs)
      {
        y-=dxabs;
        py+=sdy;
      }
      px+=sdx;
      plot_pixel(px,py,color);
    }
  }
  else /* the line is more vertical than horizontal */
  {
    for(i=0;i<dyabs;i++)
    {
      x+=dxabs;
      if (x>=dyabs)
      {
        x-=dyabs;
        px+=sdx;
      }
      py+=sdy;
      plot_pixel(px,py,color);
    }
  }
}

/**************************************************************************
 *  rect_slow                                                             *
 *    Draws a rectangle by calling the line function four times.          *
 **************************************************************************/

void rect_slow(int left,int top, int right, int bottom, byte color)
{
  line(left,top,right,top,color);
  line(left,top,left,bottom,color);
  line(right,top,right,bottom,color);
  line(left,bottom,right,bottom,color);
}

/**************************************************************************
 *  rect_fast                                                             *
 *    Draws a rectangle by drawing all lines by itself.                   *
 **************************************************************************/

void rect_fast(int left,int top, int right, int bottom, byte color)
{
  word top_offset,bottom_offset,i,temp;

  if (top>bottom)
  {
    temp=top;
    top=bottom;
    bottom=temp;
  }
  if (left>right)
  {
    temp=left;
    left=right;
    right=temp;
  }

  top_offset=(top<<8)+(top<<6);
  bottom_offset=(bottom<<8)+(bottom<<6);
  for(i=left;i<=right;i++)
  {
    VGA[top_offset+i]=color;
    VGA[bottom_offset+i]=color;
  }
  for(i=top_offset;i<=bottom_offset;i+=SCREEN_WIDTH)
  {
    VGA[left+i]=color;
    VGA[right+i]=color;
  }
}

/**************************************************************************
 *  rect_fill                                                             *
 *    Draws and fills a rectangle.                                        *
 **************************************************************************/

void rect_fill(int left,int top, int right, int bottom, byte color)
{
  word top_offset,bottom_offset,i,temp,width;

  if (top>bottom)
  {
    temp=top;
    top=bottom;
    bottom=temp;
  }
  if (left>right)
  {
    temp=left;
    left=right;
    right=temp;
  }

  top_offset=(top<<8)+(top<<6)+left;
  bottom_offset=(bottom<<8)+(bottom<<6)+left;
  width=right-left+1;

  for(i=top_offset;i<=bottom_offset;i+=SCREEN_WIDTH)
  {
    memset(&VGA[i],color,width);
  }
}

/**************************************************************************
 *  Main                                                                  *
 *    Draws 5000 rectangles                                               *
 **************************************************************************/

void main()
{
  int x1,y1,x2,y2,color;
  float t1,t2;
  word i,start;

  srand(*my_clock);                   /* seed the number generator. */
  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<5000;i++)                 /* randomly draw 5000 rectangles. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    rect_slow(x1,y1,x2,y2,color);
  }

  t1=(*my_clock-start)/18.2;          /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                         to clear the screen. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<5000;i++)                 /* randomly draw 5000 rectangles. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    rect_fast(x1,y1,x2,y2,color);
  }

  t2=(*my_clock-start)/18.2;          /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                         to clear the screen. */

  for(i=0;i<1000;i++)                 /* randomly draw 1000 filled rects. */
  {
    x1=rand()%SCREEN_WIDTH;
    y1=rand()%SCREEN_HEIGHT;
    x2=rand()%SCREEN_WIDTH;
    y2=rand()%SCREEN_HEIGHT;
    color=rand()%NUM_COLORS;
    rect_fill(x1,y1,x2,y2,color);
  }

  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */

  /* output the results... */
  printf("Slow rectangle drawing took %f seconds.\n",t1);
  printf("Fast rectangle drawing took %f seconds.\n",t2);
  if (t2 != 0) printf("Fast rectangle drawing was %f times faster.\n",t1/t2);

  return;
}
```

« Back to Primitive Shapes & Lines

DAVID BRACKEEN

256-Color VGA Programming in C > Primitive Shapes & Lines

circle.c

View as plain text

```
/**************************************************************************
 * circle.c                                                               *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * Tab stops are set to 2.                                                *
 * This program compiles with DJGPP! (www.delorie.com)                    *
 * To compile in DJGPP: gcc circle.c -o circle.exe                        *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA, or compatible video adapter.                            *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates drawing how much faster it is to *
 * draw circles using tables rather than math functions.                  *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <math.h>
#include <sys/nearptr.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

typedef unsigned char  byte;
typedef unsigned short word;
typedef long           fixed16_16;

fixed16_16 SIN_ACOS[1024];

byte *VGA = (byte *)0xA0000;          /* this points to video memory. */
word *my_clock = (word *)0x046C;      /* this points to the 18.2hz system
                                         clock. */

/**************************************************************************
```

```
 *  set_mode                                                               *
 *     Sets the video mode.                                                *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  circle_slow                                                            *
 *    Draws a circle by using floating point numbers and math fuctions.   *
 **************************************************************************/

void circle_slow(int x,int y, int radius, byte color)
{
  float n=0,invradius=1/(float)radius;
  int dx=0,dy=radius-1;
  word dxoffset,dyoffset,offset=(y<<8)+(y<<6)+x;

  while (dx<=dy)
  {
    dxoffset = (dx<<8) + (dx<<6);
    dyoffset = (dy<<8) + (dy<<6);
    VGA[offset+dy-dxoffset] = color;  /* octant 0 */
    VGA[offset+dx-dyoffset] = color;  /* octant 1 */
    VGA[offset-dx-dyoffset] = color;  /* octant 2 */
    VGA[offset-dy-dxoffset] = color;  /* octant 3 */
    VGA[offset-dy+dxoffset] = color;  /* octant 4 */
    VGA[offset-dx+dyoffset] = color;  /* octant 5 */
    VGA[offset+dx+dyoffset] = color;  /* octant 6 */
    VGA[offset+dy+dxoffset] = color;  /* octant 7 */
    dx++;
    n+=invradius;
    dy=radius * sin(acos(n));
  }
}

/**************************************************************************
 *  circle_fast                                                            *
 *    Draws a circle by using fixed point numbers and a trigonometry      *
 *    table.                                                               *
 **************************************************************************/

void circle_fast(int x,int y, int radius, byte color)
{
  fixed16_16 n=0,invradius=(1/(float)radius)*0x10000L;
  int dx=0,dy=radius-1;
  word dxoffset,dyoffset,offset = (y<<8)+(y<<6)+x;

  while (dx<=dy)
  {
    dxoffset = (dx<<8) + (dx<<6);
    dyoffset = (dy<<8) + (dy<<6);
    VGA[offset+dy-dxoffset] = color;  /* octant 0 */
    VGA[offset+dx-dyoffset] = color;  /* octant 1 */
    VGA[offset-dx-dyoffset] = color;  /* octant 2 */
    VGA[offset-dy-dxoffset] = color;  /* octant 3 */
    VGA[offset-dy+dxoffset] = color;  /* octant 4 */
    VGA[offset-dx+dyoffset] = color;  /* octant 5 */
    VGA[offset+dx+dyoffset] = color;  /* octant 6 */
    VGA[offset+dy+dxoffset] = color;  /* octant 7 */
    dx++;
    n+=invradius;
    dy = (int)((radius * SIN_ACOS[(int)(n>>6)]) >> 16);
  }
}

/**************************************************************************
 *  circle_fill                                                           *
 *    Draws and fills a circle.                                           *
 **************************************************************************/

void circle_fill(int x,int y, int radius, byte color)
{
  fixed16_16 n=0,invradius=(1/(float)radius)*0x10000L;
  int dx=0,dy=radius-1,i;
  word dxoffset,dyoffset,offset = (y<<8)+(y<<6)+x;

  while (dx<=dy)
  {
    dxoffset = (dx<<8) + (dx<<6);
    dyoffset = (dy<<8) + (dy<<6);
    for(i=dy;i>=dx;i--,dyoffset-=SCREEN_WIDTH)
    {
      VGA[offset+i -dxoffset] = color;  /* octant 0 */
      VGA[offset+dx-dyoffset] = color;  /* octant 1 */
      VGA[offset-dx-dyoffset] = color;  /* octant 2 */
      VGA[offset-i -dxoffset] = color;  /* octant 3 */
      VGA[offset-i +dxoffset] = color;  /* octant 4 */
      VGA[offset-dx+dyoffset] = color;  /* octant 5 */
      VGA[offset+dx+dyoffset] = color;  /* octant 6 */
      VGA[offset+i +dxoffset] = color;  /* octant 7 */
    }
    dx++;
    n+=invradius;
    dy = (int)((radius * SIN_ACOS[(int)(n>>6)]) >> 16);
  }
}

/**************************************************************************
 *  Main                                                                  *
 *    Draws 5000 circles                                                  *
 **************************************************************************/

void main()
{
  int x,y,radius,color;
  float t1,t2;
  word i,start;
```

```
if (__djgpp_nearptr_enable() == 0)
{
  printf("Could get access to first 640K of memory.\n");
  exit(-1);
}

VGA+=__djgpp_conventional_base;
my_clock = (void *)my_clock + __djgpp_conventional_base;


for(i=0;i<1024;i++)                 /* create the sin(arccos(x)) table. */
{
  SIN_ACOS[i]=sin(acos((float)i/1024))*0x10000L;
}

srand(*my_clock);                   /* seed the number generator. */
set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

start=*my_clock;                    /* record the starting time. */
for(i=0;i<5000;i++)                 /* randomly draw 5000 circles. */
{
  radius=rand()%90+1;
  x=rand()%(SCREEN_WIDTH-radius*2)+radius;
  y=rand()%(SCREEN_HEIGHT-radius*2)+radius;
  color=rand()%NUM_COLORS;
  circle_slow(x,y,radius,color);
}

t1=(*my_clock-start)/18.2;          /* calculate how long it took. */

set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                       to clear the screen. */

start=*my_clock;                    /* record the starting time. */
for(i=0;i<5000;i++)                 /* randomly draw 5000 circles. */
{
  radius=rand()%90+1;
  x=rand()%(SCREEN_WIDTH-radius*2)+radius;
  y=rand()%(SCREEN_HEIGHT-radius*2)+radius;
  color=rand()%NUM_COLORS;
  circle_fast(x,y,radius,color);
}

t2=(*my_clock-start)/18.2;          /* calculate how long it took. */

set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                       to clear the screen. */

for(i=0;i<1000;i++)                 /* draw 1000 filled circles. */
{
  radius=rand()%90+1;
  x=rand()%(SCREEN_WIDTH-radius*2)+radius;
  y=rand()%(SCREEN_HEIGHT-radius*2)+radius;
  color=rand()%NUM_COLORS;
  circle_fill(x,y,radius,color);
}
```

```
  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */

  /* output the results... */
  printf("Slow circle drawing took %f seconds.\n",t1);
  printf("Fast circle drawing took %f seconds.\n",t2);
  if (t2 != 0) printf("Fast circle drawing was %f times faster.\n",t1/t2);

  __djgpp_nearptr_disable();

  return;
}
```

« Back to Primitive Shapes & Lines

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

256-Color VGA Programming in C > Primitive Shapes & Lines

circle.c

View as plain text

```
/**************************************************************************
 * circle.c                                                               *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * This is a 16-bit program.                                              *
 * Tab stops are set to 2.                                                *
 * Remember to compile in the LARGE memory model!                         *
 * To compile in Borland C: bcc -ml circle.c                              *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA or compatible video adapter.                             *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates drawing how much faster it is to *
 * draw circles using tables rather than math functions.                  *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <math.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

typedef unsigned char  byte;
typedef unsigned short word;
typedef long           fixed16_16;

fixed16_16 SIN_ACOS[1024];
byte *VGA=(byte *)0xA0000000L;        /* this points to video memory. */
word *my_clock=(word *)0x0000046C;    /* this points to the 18.2hz system
                                         clock. */

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                                 *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  circle_slow                                                           *
 *    Draws a circle by using floating point numbers and math fuctions.   *
 **************************************************************************/

void circle_slow(int x,int y, int radius, byte color)
{
  float n=0,invradius=1/(float)radius;
  int dx=0,dy=radius-1;
  word dxoffset,dyoffset,offset=(y<<8)+(y<<6)+x;

  while (dx<=dy)
  {
    dxoffset = (dx<<8) + (dx<<6);
    dyoffset = (dy<<8) + (dy<<6);
    VGA[offset+dy-dxoffset] = color;  /* octant 0 */
    VGA[offset+dx-dyoffset] = color;  /* octant 1 */
    VGA[offset-dx-dyoffset] = color;  /* octant 2 */
    VGA[offset-dy-dxoffset] = color;  /* octant 3 */
    VGA[offset-dy+dxoffset] = color;  /* octant 4 */
    VGA[offset-dx+dyoffset] = color;  /* octant 5 */
    VGA[offset+dx+dyoffset] = color;  /* octant 6 */
    VGA[offset+dy+dxoffset] = color;  /* octant 7 */
    dx++;
    n+=invradius;
    dy=radius * sin(acos(n));
  }
}

/**************************************************************************
 *  circle_fast                                                           *
 *    Draws a circle by using fixed point numbers and a trigonometry      *
 *    table.                                                              *
 **************************************************************************/

void circle_fast(int x,int y, int radius, byte color)
{
  fixed16_16 n=0,invradius=(1/(float)radius)*0x10000L;
  int dx=0,dy=radius-1;
  word dxoffset,dyoffset,offset = (y<<8)+(y<<6)+x;

  while (dx<=dy)
  {
    dxoffset = (dx<<8) + (dx<<6);
    dyoffset = (dy<<8) + (dy<<6);
    VGA[offset+dy-dxoffset] = color;  /* octant 0 */
    VGA[offset+dx-dyoffset] = color;  /* octant 1 */
    VGA[offset-dx-dyoffset] = color;  /* octant 2 */
    VGA[offset-dy-dxoffset] = color;  /* octant 3 */
    VGA[offset-dy+dxoffset] = color;  /* octant 4 */
    VGA[offset-dx+dyoffset] = color;  /* octant 5 */
    VGA[offset+dx+dyoffset] = color;  /* octant 6 */
    VGA[offset+dy+dxoffset] = color;  /* octant 7 */
    dx++;
    n+=invradius;
    dy = (int)((radius * SIN_ACOS[(int)(n>>6)]) >> 16);
  }
}

/**************************************************************************
 *  circle_fill                                                           *
 *    Draws and fills a circle.                                           *
 **************************************************************************/

void circle_fill(int x,int y, int radius, byte color)
{
  fixed16_16 n=0,invradius=(1/(float)radius)*0x10000L;
  int dx=0,dy=radius-1,i;
  word dxoffset,dyoffset,offset = (y<<8)+(y<<6)+x;

  while (dx<=dy)
  {
    dxoffset = (dx<<8) + (dx<<6);
    dyoffset = (dy<<8) + (dy<<6);
    for(i=dy;i>=dx;i--,dyoffset-=SCREEN_WIDTH)
    {
      VGA[offset+i -dxoffset] = color;  /* octant 0 */
      VGA[offset+dx-dyoffset] = color;  /* octant 1 */
      VGA[offset-dx-dyoffset] = color;  /* octant 2 */
      VGA[offset-i -dxoffset] = color;  /* octant 3 */
      VGA[offset-i +dxoffset] = color;  /* octant 4 */
      VGA[offset-dx+dyoffset] = color;  /* octant 5 */
      VGA[offset+dx+dyoffset] = color;  /* octant 6 */
      VGA[offset+i +dxoffset] = color;  /* octant 7 */
    }
    dx++;
    n+=invradius;
    dy = (int)((radius * SIN_ACOS[(int)(n>>6)]) >> 16);
  }
}

/**************************************************************************
 *  Main                                                                  *
 *    Draws 5000 circles                                                  *
 **************************************************************************/

void main()
{
  int x,y,radius,color;
  float t1,t2;
  word i,start;
  for(i=0;i<1024;i++)                 /* create the sin(arccos(x)) table. */
  {
    SIN_ACOS[i]=sin(acos((float)i/1024))*0x10000L;
  }

  srand(*my_clock);                   /* seed the number generator. */
  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<5000;i++)                 /* randomly draw 5000 circles. */
  {
    radius=rand()%90+1;
    x=rand()%(SCREEN_WIDTH-radius*2)+radius;
    y=rand()%(SCREEN_HEIGHT-radius*2)+radius;
    color=rand()%NUM_COLORS;
    circle_slow(x,y,radius,color);
  }

  t1=(*my_clock-start)/18.2;          /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                         to clear the screen. */

  start=*my_clock;                    /* record the starting time. */
  for(i=0;i<5000;i++)                 /* randomly draw 5000 circles. */
  {
    radius=rand()%90+1;
    x=rand()%(SCREEN_WIDTH-radius*2)+radius;
    y=rand()%(SCREEN_HEIGHT-radius*2)+radius;
    color=rand()%NUM_COLORS;
    circle_fast(x,y,radius,color);
  }

  t2=(*my_clock-start)/18.2;          /* calculate how long it took. */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode again in order
                                         to clear the screen. */

  for(i=0;i<1000;i++)                 /* draw 1000 filled circles. */
  {
    radius=rand()%90+1;
    x=rand()%(SCREEN_WIDTH-radius*2)+radius;
    y=rand()%(SCREEN_HEIGHT-radius*2)+radius;
    color=rand()%NUM_COLORS;
    circle_fill(x,y,radius,color);
  }
  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */

  /* output the results... */
  printf("Slow circle drawing took %f seconds.\n",t1);
  printf("Fast circle drawing took %f seconds.\n",t2);
  if (t2 != 0) printf("Fast circle drawing was %f times faster.\n",t1/t2);

  return;
}
```

« Back to Primitive Shapes & Lines

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

256-Color VGA Programming in C

Bitmaps & Palette Manipulation

Contents in this section:

What is a bitmap?

One of the most important things in creating a user-friendly interface is the use of bitmaps. Without bitmaps, there would be no icons, no fancy buttons, and mouse pointers would have to be made of lines.

The term *bitmap* is a throwback from when monitors could only display one other color besides black. For two-color data files that store an image, each bit in the data file represents one pixel; a 1 meant the pixel was on, a 0 meant the pixel was off (Figure 13). Therefore, a two-color image is a map of bits.

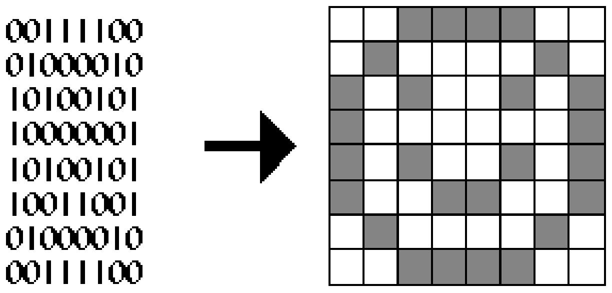

Figure 13. A black & white bitmap in memory and on the screen.

The BMP file format

There are many file formats for storing bitmaps, such as RLE, JPEG, TIFF, TGA, PCX, BMP, PNG, PCD and GIF. The bitmaps studied in this section will be 256-color bitmaps, where eight bits represents one pixel.. One of the easiest 256-color bitmap file format is Windows' BMP. This file format can be stored uncompressed, so reading BMP files is fairly simple; most other graphics formats are compressed, and some, like GIF, are difficult to decompress. To learn about other graphics file formats, visit x2ftp.

There are a few different sub-types of the BMP file format. The one studied here is Windows' RGB-encoded BMP format. For 256-color bitmaps, it has a 54-byte header (Table III) followed by a 1024-byte palette table. After that is the actual bitmap, which starts at the lower-left hand corner.

Data	**Description**
`WORD Type;`	File type. Set to "BM".
`DWORD Size;`	Size in BYTES of the file.
`DWORD Reserved;`	Reserved. Set to zero.
`DWORD Offset;`	Offset to the data.
`DWORD headerSize;`	Size of rest of header. Set to 40.
`DWORD Width;`	Width of bitmap in pixels.
`DWORD Height;`	Height of bitmap in pixels.
`WORD Planes;`	Number of Planes. Set to 1.
`WORD BitsPerPixel;`	Number of bits per pixel.
`DWORD Compression;`	Compression. Usually set to 0.
`DWORD SizeImage;`	Size in bytes of the bitmap.
`DWORD XPixelsPerMeter;`	Horizontal pixels per meter.
`DWORD YPixelsPerMeter;`	Vertical pixels per meter.
`DWORD ColorsUsed;`	Number of colors used.
`DWORD ColorsImportant;`	Number of "important" colors.

Table III. Windows' BMP file format header.

Drawing bitmaps

Once read, displaying the bitmap is relatively easy, and involves only a few memory copies to display memory. The following is the code to display a 32x64 image stored in an array `bitmap`:

```
for(y=0;y<64;y++)
  for(x=0;x<32;x++)
    VGA [x+y*320]=bitmap [x+y*32];
```

Something interesting to note about the BMP file format is that each scan line is padded to the nearest 4-byte boundry. So, if the image read has a width that is not divisible by four, say, 21 bytes, there would be 3 bytes of padding at the end of every scan line. The program `bitmap.exe` does not account for this; it assumes the bitmap's width *is* divisible by four.

There are many techniques to implement transparency. One way is to assign one of the 256 colors to be transparent in the program. When drawing the image, a byte with the transparency value is not written to video memory. The following implements this using zero as the transparency value:

```
for(y=0;y<64;y++)
  for(x=0;x<32;x++)
  {
    data=bitmap [x+y*32];
    if (data!=0) VGA [x+y*320]=data;
  }
```

The following program `bitmap.c` reads a bitmap file `rocket.bmp` (Figure 14) and draws it to the screen in a tiled fashion, using both opaque and transparent bitmap drawing functions.

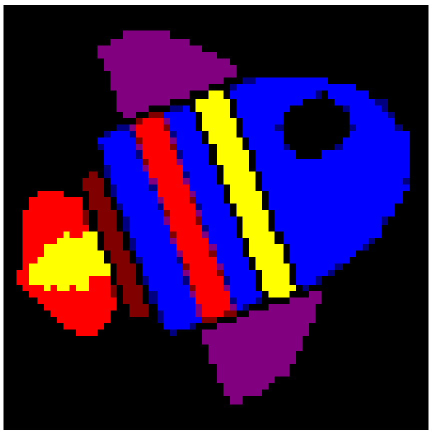

Figure 14. Bitmap `rocket.bmp`.

Program: bitmap.c

DJGPP 2.0
- View bitmap.c
- Download bitmap.zip (Contains bitmap.c, bitmap.exe, rocket.bmp)

Borland C, Turbo C, etc.
- View bitmap.c
- Download bitmap.zip (Contains bitmap.c, bitmap.exe, rocket.bmp)

Having trouble compiling or running the program? See the Troubleshooting page.

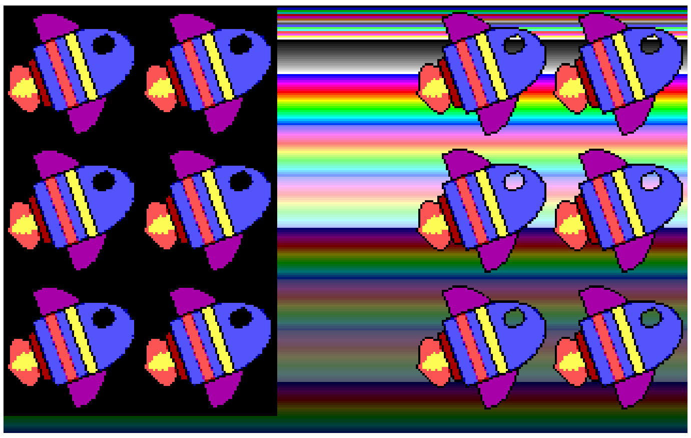

Figure 15. Output from `bitmap.exe`.

Palette manipulation

The background in the output of `bitmap.exe` is a representation of the VGA's 256-color palette. Fortunately, the palette is programmable to other colors, so bitmaps are not forced onto an odd palette. Unfortunately, the VGA only gives us 6 bits per color channel, so the best you can get is 18-bit color (but you can only pick 256 of those colors, of course). Palette information is stored after the header in the BMP file format. Four bytes define each color: one byte each for blue, green, red, and one reserved byte. The VGA understands color values in the order red, green, blue, (reverse of the BMP format) plus the program needs to change the palette data form 24-bit to 18-bit (divide each color by four, or right-shift by two).

To set one color in the palette, write the color index to port 0x3C8 and then write the red, green, and blue values, in order, to port 0x3C9

```
outp(0x03c8, index);
outp(0x03c9, red);
outp(0x03c9, green);
outp(0x03c9, blue);
```

To set all 256 colors in the palette, write zero to port 0x3C8 and then write all 768 bytes of the palette to port 0x3C9.

```
outp(0x03c8, 0);
for(i=0;i<256;i++)
{
  outp(0x03c9, palette_red[i]);
  outp(0x03c9, palette_green[i]);
  outp(0x03c9, palette_blue[i];
}
```

Note that the palette cannot be set until the 256-color video mode has been set.

The program `palette.c` reads in an image, displays it, and then cycles through all the colors by repeatedly changing the palette.

Program: palette.c

DJGPP 2.0
  View palette.c
  Download palette.zip (Contains palette.c, palette.exe, mset.bmp)
Borland C, Turbo C, etc.
  View palette.c
  Download palette.zip (Contains palette.c, palette.exe, mset.bmp)

Having trouble compiling or running the program? See the Troubleshooting page.

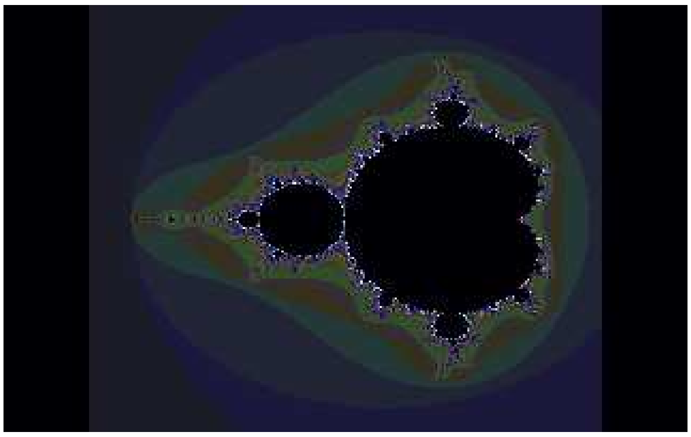

Figure 16. Output from `palette.exe`.

Vertical retrace

Something to note about the program is the function `wait_for_retrace`:

```
void wait_for_retrace(void)
{
  /* wait until done with vertical retrace */
  while  ((inp(INPUT_STATUS) & VRETRACE));
  /* wait until done refreshing */
  while (!(inp(INPUT_STATUS) & VRETRACE));
}
```

If the `while` loops in this function were commented out and the program was run, two things would happen: the palette would cycle very, very quickly, and the image would appear to have shearing effect as the palette cycled. The reason has to do with the VGA’s vertical refresh rate.

The VGA refreshes the screen 70 times a second, or 70hz. An electron gun goes across the screen from left to right, top to bottom. When it gets to the bottom of the screen, (i.e., it finished refreshing), the electron gun has to travel from the bottom right corner of the screen to the upper left corner of the screen. That time, called the retrace period, is the ideal time to change the palette. If the program did not wait for the retrace, the palette would sometimes be changed in the middle of a refresh, resulting in different colors on the top portion of the screen for a split second. This is how the shearing effect happens. To eliminate this, `palette.c` uses the `wait_for_retrace` function.

The other effect is that the function acts as a timer, which, since the function is called twice, makes the palette cycle at 35 times a second. This is very useful when animating bitmaps, which is a primary focus in the next section.

« Previous: Primitive Shapes & Lines
Next: Mouse Support & Animation »

DAVID BRACKEEN

256-Color VGA Programming in C > Bitmaps & Palette Manipulation

bitmap.c

View as plain text

```
/**************************************************************************
 * bitmap.c                                                               *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * Tab stops are set to 2.                                                *
 * This program compiles with DJGPP! (www.delorie.com)                    *
 * To compile in DJGPP: gcc bitmap.c -o bitmap.exe                        *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA, or compatible video adapter.                            *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates drawing bitmaps, including       *
 * transparent bitmaps.                                                   *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <sys/nearptr.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;

byte *VGA = (byte *)0xA0000;          /* this points to video memory. */
word *my_clock = (word *)0x046C;      /* this points to the 18.2hz system
                                         clock. */

typedef struct tagBITMAP              /* the structure for a bitmap. */
{
  word width;
  word height;
  byte *data;
} BITMAP;

/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory.                                    *
 **************************************************************************/

void load_bmp(char *file,BITMAP *b)
{
  FILE *fp;
  long index;
  word num_colors;
  int x;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }

  /* read in the width and height of the image, and the
```

```
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;


  /* try to allocate memory */
  if ((b->data = (byte *) malloc((word)(b->width*b->height))) == NULL)
  {
    fclose(fp);
    printf("Error allocating memory for file %s.\n",file);
    exit(1);
  }

  /* Ignore the palette information for now.
     See palette.c for code to read the palette info. */
  fskip(fp,num_colors*4);

  /* read the bitmap */
  for(index=(b->height-1)*b->width;index>=0;index-=b->width)
    for(x=0;x<b->width;x++)
      b->data[(word)index+x]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  draw_bitmap                                                           *
 *    Draws a bitmap.                                                     *
 **************************************************************************/

void draw_bitmap(BITMAP *bmp,int x,int y)
{
  int j;
  word screen_offset = (y<<8)+(y<<6)+x;
  word bitmap_offset = 0;

  for(j=0;j<bmp->height;j++)
  {
    memcpy(&VGA[screen_offset],&bmp->data[bitmap_offset],bmp->width);

    bitmap_offset+=bmp->width;
    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  draw_transparent_bitmap                                               *
 *    Draws a transparent bitmap.                                         *
 **************************************************************************/
```

```
void draw_transparent_bitmap(BITMAP *bmp,int x,int y)
{
  int i,j;
  word screen_offset = (y<<8)+(y<<6);
  word bitmap_offset = 0;
  byte data;

  for(j=0;j<bmp->height;j++)
  {
    for(i=0;i<bmp->width;i++,bitmap_offset++)
    {
      data = bmp->data[bitmap_offset];
      if (data) VGA[screen_offset+x+i] = data;
    }
    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  wait                                                                  *
 *    Wait for a specified number of clock ticks (18hz).                  *
 **************************************************************************/

void wait(int ticks)
{
  word start;

  start=*my_clock;

  while (*my_clock-start<ticks)
  {
    *my_clock=*my_clock;              /* this line is for some compilers
                                         that would otherwise ignore this
                                         loop */
  }
}

/**************************************************************************
 *  Main                                                                  *
 *    Draws opaque and transparent bitmaps                                *
 **************************************************************************/

void main()
{
  int i,x,y;
  BITMAP bmp;

  if (__djgpp_nearptr_enable() == 0)
  {
    printf("Could get access to first 640K of memory.\n");
    exit(-1);
  }

  VGA+=__djgpp_conventional_base;
  my_clock = (void *)my_clock + __djgpp_conventional_base;
  load_bmp("rocket.bmp",&bmp);        /* open the file */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  /* draw the background */
  for(i=0;i<200;i++)
    memset(&VGA[320*i],i,SCREEN_WIDTH);

  wait(25);

  /* draw a tiled bitmap pattern on the left */
  for(y=0;y<=SCREEN_HEIGHT-bmp.height;y+=bmp.height)
    for(x=0;x<=(SCREEN_WIDTH)/2-bmp.width;x+=bmp.width)
      draw_bitmap(&bmp,x,y);

  wait(25);

  /* draw a tiled transparent bitmap pattern on the right */
  for(y=0;y<=SCREEN_HEIGHT-bmp.height;y+=bmp.height)
    for(x=SCREEN_WIDTH-bmp.width;x>=SCREEN_WIDTH/2;x-=bmp.width)
      draw_transparent_bitmap(&bmp,x,y);

  wait(100);

  free(bmp.data);                     /* free up memory used */

  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */
  __djgpp_nearptr_disable();

  return;
}
```

« Back to Bitmaps & Palette Manipulation

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

256-Color VGA Programming in C > Bitmaps & Palette Manipulation

bitmap.c

View as plain text

```
/**************************************************************************
 * bitmap.c                                                               *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * This is a 16-bit program.                                              *
 * Tab stops are set to 2.                                                *
 * Remember to compile in the LARGE memory model!                         *
 * To compile in Borland C: bcc -ml bitmap.c                              *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA or compatible video adapter.                             *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates drawing bitmaps, including       *
 * transparent bitmaps.                                                   *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <mem.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;

byte *VGA=(byte *)0xA0000000L;        /* this points to video memory. */
word *my_clock=(word *)0x0000046C;    /* this points to the 18.2hz system
                                         clock. */

typedef struct tagBITMAP              /* the structure for a bitmap. */
{
  word width;
  word height;
  byte *data;
} BITMAP;

/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory.                                    *
 **************************************************************************/

void load_bmp(char *file,BITMAP *b)
{
  FILE *fp;
  long index;
  word num_colors;
  int x;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }
  /* read in the width and height of the image, and the
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;


  /* try to allocate memory */
  if ((b->data = (byte *) malloc((word)(b->width*b->height))) == NULL)
  {
    fclose(fp);
    printf("Error allocating memory for file %s.\n",file);
    exit(1);
  }

  /* Ignore the palette information for now.
     See palette.c for code to read the palette info. */
  fskip(fp,num_colors*4);

  /* read the bitmap */
  for(index=(b->height-1)*b->width;index>=0;index-=b->width)
    for(x=0;x<b->width;x++)
      b->data[(word)index+x]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  draw_bitmap                                                           *
 *    Draws a bitmap.                                                     *
 **************************************************************************/

void draw_bitmap(BITMAP *bmp,int x,int y)
{
  int j;
  word screen_offset = (y<<8)+(y<<6)+x;
  word bitmap_offset = 0;

  for(j=0;j<bmp->height;j++)
  {
    memcpy(&VGA[screen_offset],&bmp->data[bitmap_offset],bmp->width);

    bitmap_offset+=bmp->width;
    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  draw_transparent_bitmap                                               *
 *    Draws a transparent bitmap.                                         *
 **************************************************************************/

void draw_transparent_bitmap(BITMAP *bmp,int x,int y)
{
  int i,j;
  word screen_offset = (y<<8)+(y<<6);
  word bitmap_offset = 0;
  byte data;

  for(j=0;j<bmp->height;j++)
  {
    for(i=0;i<bmp->width;i++,bitmap_offset++)
    {
      data = bmp->data[bitmap_offset];
      if (data) VGA[screen_offset+x+i] = data;
    }
    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  wait                                                                  *
 *    Wait for a specified number of clock ticks (18hz).                  *
 **************************************************************************/

void wait(int ticks)
{
  word start;

  start=*my_clock;

  while (*my_clock-start<ticks)
  {
    *my_clock=*my_clock;              /* this line is for some compilers
                                         that would otherwise ignore this
                                         loop */
  }
}

/**************************************************************************
 *  Main                                                                  *
 *    Draws opaque and transparent bitmaps                                *
 **************************************************************************/

void main()
{
  int i,x,y;
  BITMAP bmp;

  load_bmp("rocket.bmp",&bmp);        /* open the file */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  /* draw the background */
  for(i=0;i<200;i++)
    memset(&VGA[320*i],i,SCREEN_WIDTH);
  wait(25);

  /* draw a tiled bitmap pattern on the left */
  for(y=0;y<=SCREEN_HEIGHT-bmp.height;y+=bmp.height)
    for(x=0;x<=(SCREEN_WIDTH)/2-bmp.width;x+=bmp.width)
      draw_bitmap(&bmp,x,y);

  wait(25);

  /* draw a tiled transparent bitmap pattern on the right */
  for(y=0;y<=SCREEN_HEIGHT-bmp.height;y+=bmp.height)
    for(x=SCREEN_WIDTH-bmp.width;x>=SCREEN_WIDTH/2;x-=bmp.width)
      draw_transparent_bitmap(&bmp,x,y);

  wait(100);

  free(bmp.data);                    /* free up memory used */

  set_mode(TEXT_MODE);               /* set the video mode back to
                                        text mode. */

  return;
}
```

« Back to Bitmaps & Palette Manipulation

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

palette.c

View as plain text

```
/**************************************************************************
 * palette.c                                                              *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * Tab stops are set to 2.                                                *
 * This program compiles with DJGPP! (www.delorie.com)                    *
 * To compile in DJGPP: gcc palette.c -o palette.exe                      *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA, or compatible video adapter.                            *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates palette manipulation and         *
 * vertical retrace sychronization.                                       *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <sys/nearptr.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define PALETTE_INDEX       0x03c8
#define PALETTE_DATA        0x03c9
#define INPUT_STATUS        0x03da
#define VRETRACE            0x08

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;

byte *VGA = (byte *)0xA0000;          /* this points to video memory. */
word *my_clock = (word *)0x046C;      /* this points to the 18.2hz system
                                         clock. */
```

```
typedef struct tagBITMAP              /* the structure for a bitmap. */
{
  word width;
  word height;
  byte palette[256*3];
  byte *data;
} BITMAP;


/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory.                                    *
 **************************************************************************/

void load_bmp(char *file,BITMAP *b)
{
  FILE *fp;
  long index;
  word num_colors;
  int x;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }

  /* read in the width and height of the image, and the
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;

  /* try to allocate memory */
  if ((b->data = (byte *) malloc((word)(b->width*b->height))) == NULL)
  {
    fclose(fp);
    printf("Error allocating memory for file %s.\n",file);
    exit(1);
  }

  /* read the palette information */
  for(index=0;index<num_colors;index++)
  {
    b->palette[(int)(index*3+2)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+1)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+0)] = fgetc(fp) >> 2;
    x=fgetc(fp);
  }

  /* read the bitmap */
  for(index=(b->height-1)*b->width;index>=0;index-=b->width)
    for(x=0;x<b->width;x++)
      b->data[(word)(index+x)]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  draw_bitmap                                                           *
 *    Draws a bitmap.                                                     *
 **************************************************************************/

void draw_bitmap(BITMAP *bmp,int x,int y)
{
  int j;
  word screen_offset = (y<<8)+(y<<6)+x;
  word bitmap_offset = 0;

  for(j=0;j<bmp->height;j++)
  {
    memcpy(&VGA[screen_offset],&bmp->data[bitmap_offset],bmp->width);

    bitmap_offset+=bmp->width;
    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  set_palette                                                           *
 *    Sets all 256 colors of the palette.                                 *
 **************************************************************************/

void set_palette(byte *palette)
{
  int i;

  outp(PALETTE_INDEX,0);              /* tell the VGA that palette data
                                         is coming. */
  for(i=0;i<256*3;i++)
    outp(PALETTE_DATA,palette[i]);    /* write the data */
}

/**************************************************************************
 *  rotate_palette                                                        *
 *    Rotates the colors of the palette.                                  *
 **************************************************************************/

void rotate_palette(byte *palette)
{
  int i,red,green,blue;

  red  = palette[3];
  green= palette[4];
  blue = palette[5];

  for(i=3;i<256*3-3;i++)
    palette[i]=palette[i+3];

  palette[256*3-3]=red;
  palette[256*3-2]=green;
  palette[256*3-1]=blue;

  set_palette(palette);
}

/**************************************************************************
 *  wait_for_retrace                                                      *
 *    Wait until the *beginning* of a vertical retrace cycle (60hz).      *
 **************************************************************************/

void wait_for_retrace(void)
{
    /* wait until done with vertical retrace */
    while  ((inp(INPUT_STATUS) & VRETRACE)) {};
    /* wait until done refreshing */
    while (!(inp(INPUT_STATUS) & VRETRACE)) {};
}
```

```
/**************************************************************************
 *  wait                                                                  *
 *    Wait for a specified number of clock ticks (18hz).                  *
 **************************************************************************/

void wait(int ticks)
{
  word start;

  start=*my_clock;

  while (*my_clock-start<ticks)
  {
    *my_clock=*my_clock;              /* this line is for some compilers
                                         that would otherwise ignore this
                                         loop */
  }
}

/**************************************************************************
 *  Main                                                                  *
 *    Draws a bitmap and then rotates the palette.                        *
 **************************************************************************/

void main()
{
  BITMAP bmp;
  int i;

  if (__djgpp_nearptr_enable() == 0)
  {
    printf("Could get access to first 640K of memory.\n");
    exit(-1);
  }

  VGA+=__djgpp_conventional_base;
  my_clock = (void *)my_clock + __djgpp_conventional_base;

  load_bmp("mset.bmp",&bmp);          /* open the file */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  set_palette(bmp.palette);           /* set the palette */

  draw_bitmap(&bmp,                   /* draw the bitmap centered */
    (SCREEN_WIDTH-bmp.width) >>1,
    (SCREEN_HEIGHT-bmp.height) >>1);

  wait(25);

  for(i=0;i<510;i++)                  /* rotate the palette at 30hz */
  {
    wait_for_retrace();
    wait_for_retrace();
    rotate_palette(bmp.palette);
  }
  wait(25);

  free(bmp.data);                     /* free up memory used */

  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */
  __djgpp_nearptr_disable();

  return;
}
```

« Back to Bitmaps & Palette Manipulation

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

palette.c

View as plain text

```
/**************************************************************************
 * palette.c                                                              *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * This is a 16-bit program.                                              *
 * Tab stops are set to 2.                                                *
 * Remember to compile in the LARGE memory model!                         *
 * To compile in Borland C: bcc -ml palette.c                             *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA or compatible video adapter.                             *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates palette manipulation and         *
 * vertical retrace sychronization.                                       *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <mem.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define PALETTE_INDEX       0x03c8
#define PALETTE_DATA        0x03c9
#define INPUT_STATUS        0x03da
#define VRETRACE            0x08

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;

byte *VGA=(byte *)0xA0000000L;        /* this points to video memory. */
word *my_clock=(word *)0x0000046C;    /* this points to the 18.2hz system
```

```
                                         clock. */

typedef struct tagBITMAP              /* the structure for a bitmap. */
{
  word width;
  word height;
  byte palette[256*3];
  byte *data;
} BITMAP;


/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory.                                    *
 **************************************************************************/

void load_bmp(char *file,BITMAP *b)
{
  FILE *fp;
  long index;
  word num_colors;
  int x;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }

  /* read in the width and height of the image, and the
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;

  /* try to allocate memory */
  if ((b->data = (byte *) malloc((word)(b->width*b->height))) == NULL)
  {
    fclose(fp);
    printf("Error allocating memory for file %s.\n",file);
    exit(1);
  }

  /* read the palette information */
  for(index=0;index<num_colors;index++)
  {
    b->palette[(int)(index*3+2)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+1)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+0)] = fgetc(fp) >> 2;
    x=fgetc(fp);
  }

  /* read the bitmap */
  for(index=(b->height-1)*b->width;index>=0;index-=b->width)
    for(x=0;x<b->width;x++)
      b->data[(word)(index+x)]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  draw_bitmap                                                           *
 *    Draws a bitmap.                                                     *
 **************************************************************************/

void draw_bitmap(BITMAP *bmp,int x,int y)
{
  int j;
  word screen_offset = (y<<8)+(y<<6)+x;
  word bitmap_offset = 0;

  for(j=0;j<bmp->height;j++)
  {
    memcpy(&VGA[screen_offset],&bmp->data[bitmap_offset],bmp->width);

    bitmap_offset+=bmp->width;
    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  set_palette                                                           *
 *    Sets all 256 colors of the palette.                                 *
 **************************************************************************/

void set_palette(byte *palette)
{
  int i;

  outp(PALETTE_INDEX,0);              /* tell the VGA that palette data
                                         is coming. */
  for(i=0;i<256*3;i++)
    outp(PALETTE_DATA,palette[i]);    /* write the data */
}

/**************************************************************************
 *  rotate_palette                                                        *
 *    Rotates the colors of the palette.                                  *
 **************************************************************************/

void rotate_palette(byte *palette)
{
  int i,red,green,blue;

  red  = palette[3];
  green= palette[4];
  blue = palette[5];

  for(i=3;i<256*3-3;i++)
    palette[i]=palette[i+3];

  palette[256*3-3]=red;
  palette[256*3-2]=green;
  palette[256*3-1]=blue;

  set_palette(palette);
}

/**************************************************************************
 *  wait_for_retrace                                                      *
 *    Wait until the *beginning* of a vertical retrace cycle (60hz).      *
 **************************************************************************/

void wait_for_retrace(void)
{
    /* wait until done with vertical retrace */
    while  ((inp(INPUT_STATUS) & VRETRACE)) {};
    /* wait until done refreshing */
    while (!(inp(INPUT_STATUS) & VRETRACE)) {};
```

```
}

/**************************************************************************
 *  wait                                                                  *
 *    Wait for a specified number of clock ticks (18hz).                  *
 **************************************************************************/

void wait(int ticks)
{
  word start;

  start=*my_clock;

  while (*my_clock-start<ticks)
  {
    *my_clock=*my_clock;              /* this line is for some compilers
                                         that would otherwise ignore this
                                         loop */
  }
}

/**************************************************************************
 *  Main                                                                  *
 *    Draws a bitmap and then rotates the palette.                        *
 **************************************************************************/

void main()
{
  BITMAP bmp;
  int i;

  load_bmp("mset.bmp",&bmp);          /* open the file */

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */

  set_palette(bmp.palette);           /* set the palette */

  draw_bitmap(&bmp,                   /* draw the bitmap centered */
    (SCREEN_WIDTH-bmp.width) >>1,
    (SCREEN_HEIGHT-bmp.height) >>1);

  wait(25);

  for(i=0;i<510;i++)                  /* rotate the palette at 30hz */
  {
    wait_for_retrace();
    wait_for_retrace();
    rotate_palette(bmp.palette);
  }

  wait(25);

  free(bmp.data);                     /* free up memory used */

  set_mode(TEXT_MODE);                /* set the video mode back to
```

```
                                        text mode. */

  return;
}
```

« Back to Bitmaps & Palette Manipulation

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

256-Color VGA Programming in C

Mouse Support & Animation

Contents in this section:

Why a mouse?

If a program that is meant to have an easy-to-use interface does not have mouse support, some would say it would not (and could not) be an easy-to-use interface. Programming the mouse is fairly simple on the low level, but there is a lot to deal with when considering buttons and animated mouse pointers, all of which will be covered in this section.

There are two ways to communicate with the mouse:

- with the serial port itself, or
- with the installed mouse driver through interrupt 0x33.

Reading the serial port can be cumbersome because the mouse must be detected, not to mention there are usually two or more serial ports on a computer, each of which could be connected to the mouse. Not to mention some mice use a PS/2 port or USB port.

The mouse driver, if installed on the user's machine, provides an easy way to detect the mouse's existence and poll messages like when a mouse button has been pressed and when the mouse has moved. Using the mouse driver also ensures the code will work no matter what type of mouse the user has. The driver also provides mouse pointer support, but it is limited to two-color mouse pointers. The program `mouse.c` at the end of this section uses all of its own mouse pointers instead of the driver's.

Initializing the mouse

Initializing the mouse is as easy as setting `AX` to zero and calling interrupt 0x33. If the mouse is installed, `AX` is set to FFFFh on return. The `BX` register returns the number of mouse buttons.

```
union REGS regs;

regs.x.ax = 0;
int86(0x33, &regs, &regs);
mouse_on = regs.x.ax;
num_buttons = regs.x.bx;
```

This also sets the mouse driver's internal mouse position to the center of the screen. The center of the screen, according to the mouse driver, is not (160,100), it is (320,100). This is because the mouse driver maps the *x* position of the mouse from 0 to 639 and the *y* position from 0 to 199, no matter what video mode is currently active.

Mouse status

To get the mouse's current status, set AX to 3 and call interrupt 0x33. The *x* value is returned in CX and the *y* value is returned in DX. BX contains the status of the mouse buttons (Figure 17).

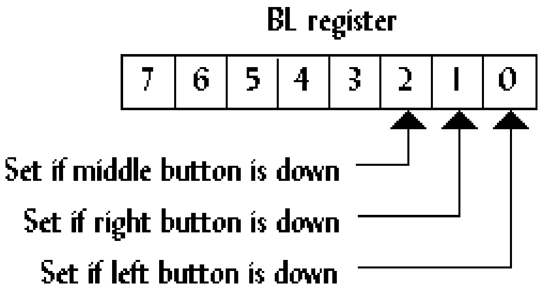
BL register
7
6
5
4
3
2
1
0
Set if middle button is down
Set if right button is down
Set if left button is down

Figure 17. The status of the mouse buttons returned in BL after calling function 3.

The following program segment is a basic mouse handler than uses a pixel as the mouse pointer. The program exits when a mouse button is pressed.

```
union REGS regs;

regs.x.ax = 0;
int86(0x33, &regs, &regs);
mouse_on = regs.x.ax;
if (!mouse_on)
{
  printf("Mouse not found.\n");
  exit(1);
}

buttons=0;
while (!buttons)
{
  regs.x.ax=3;
  int86(0x33, &regs, &regs);
  cx = regs.x.cx;
  dx = regs.x.dx;
  offset = (cx>>1)+(dx<<8)+(dx<<6);
  VGA[offset] = 15;
  if (regs.x.bx) buttons=1;
}
```

Mouse Motion

Using the mouse driver's mouse position is easy, but it is not very portable to other video modes, like 320x240 or 640x480. An alternate way to keep track of the mouse position is to let the program do it, using function 0xB to get the motion that the mouse has moved. This function returns the motion the mouse has moved horizontally in CX, and the motion the mouse has moved vertically in DX. The following program segment uses function 0xB keep track of the mouse pointer. The loop exits when the mouse is placed in the upper-left corner.

```
union REGS reg;

x=160;
y=100;
while (x>0 || y>0)
{
  /*... display mouse here ...*/

  reg.x.ax=0x0B;
  int86(0x33,&regs,&regs);
  x += (int)reg.x.cx;
  y += (int)reg.x.dx;
  if (x<0) x=0;
  if (y<0) y=0;
  if (x>319) x=319;
  if (y>199) y=199;
}
```

Mouse buttons

Sometimes it does not matter if a mouse button is up or down, only if it was just *pressed*. With functions 5 and 6, instead of reading when a button is *down*, they read when a button is first *pressed* and finally *released*. To do this, set BX to the button (0=left, 1=right, 2=middle) and call interrupt 0x33. The function returns the number of presses or releases that have occurred in BX. The following code displays the status of the left button. The loop exits when the right button is released.

```
printf("Press right button to quit\n");
do
{
  regs.x.ax=5;
  regs.x.bx=0;
  int86(0x33,&regs,&regs);
  if (regs.x.bx)
    printf("Left button pressed.\n");
  regs.x.ax=6;
  regs.x.bx=0;
  int86(0x33,&regs,&regs);
  if (regs.x.bx)
```

```
    printf("Left button released.\n");
  regs.x.ax=6;
  regs.x.bx=1;
  int86(0x33,&regs,&regs);
} while (!regs.x.bx)
```

Animation

In a user-friendly interface, when the user selects a command that takes a while to execute, the mouse pointer might become an animated clock that lets the user know the computer is working. This technique is demonstrated in `mouse.c`. The mouse bitmap is stored in a structure that has a pointer to another mouse pointer, which is the next bitmap in the animation.

The vertical retrace is monitored to eliminate flickering of the mouse pointer as well.

All the button bitmaps and mouse pointers are stored in a single file, `images.bmp` (Figure 18). The BMP file is read, then the separate icons are extracted from it.

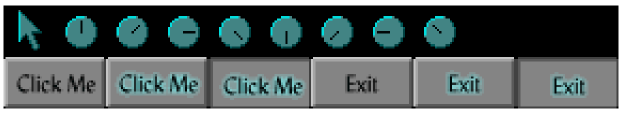

Figure 18. Bitmap `images.bmp`.

Program: mouse.c

DJGPP 2.0
    View mouse.c
    Download mouse.zip (Contains mouse.c, mouse.exe, images.bmp)
Borland C, Turbo C, etc.
    View mouse.c
    Download mouse.zip (Contains mouse.c, mouse.exe, images.bmp)

Having trouble compiling or running the program? See the Troubleshooting page.

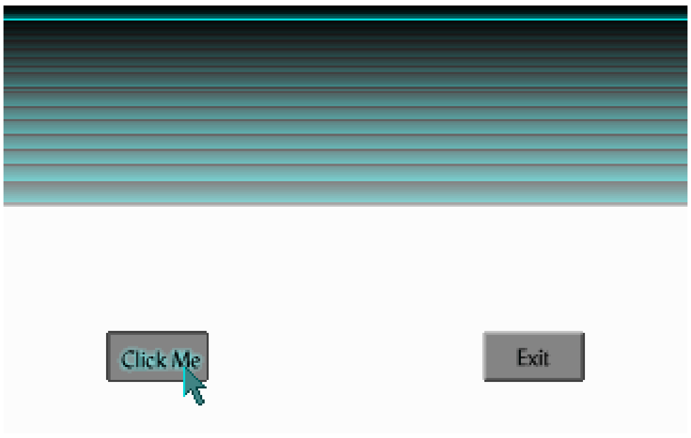

Figure 19. Output from `mouse.exe`.

The previous program is the most substantial program in this tutorial. It covers everything from this section and the previous section, and actually demonstrates a solid user interface, although the program does not do anything useful. With the ideas from these past three programs, a pull-down menu system could be created.

However, the scheme used to reduce flickering in this program is not very accurate at times. To create flicker-free programs, one of two techniques should be used: double buffering or page flipping. Both of these techniques are discussed in the next section.

« Previous: Bitmaps & Palette Manipulation
Next: Double Buffering, Page Flipping, & Unchained Mode »

DAVID BRACKEEN

mouse.c

View as plain text

```
/**************************************************************************
 * mouse.c                                                                *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * Tab stops are set to 2.                                                *
 * This program compiles with DJGPP! (www.delorie.com)                    *
 * To compile in DJGPP: gcc mouse.c -o mouse.exe                          *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA, or compatible video adapter.                            *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates mouse functions.                 *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <sys/nearptr.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define PALETTE_INDEX       0x03c8
#define PALETTE_DATA        0x03c9
#define INPUT_STATUS        0x03da
#define VRETRACE            0x08

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

#define MOUSE_INT           0x33
#define MOUSE_RESET         0x00
#define MOUSE_GETPRESS      0x05
#define MOUSE_GETRELEASE    0x06
#define MOUSE_GETMOTION     0x0B
#define LEFT_BUTTON         0x00
#define RIGHT_BUTTON        0x01
#define MIDDLE_BUTTON       0x02
```

```
#define MOUSE_WIDTH         24
#define MOUSE_HEIGHT        24
#define MOUSE_SIZE          (MOUSE_HEIGHT*MOUSE_WIDTH)

#define BUTTON_WIDTH        48
#define BUTTON_HEIGHT       24
#define BUTTON_SIZE         (BUTTON_HEIGHT*BUTTON_WIDTH)
#define BUTTON_BITMAPS      3
#define STATE_NORM          0
#define STATE_ACTIVE        1
#define STATE_PRESSED       2
#define STATE_WAITING       3

#define NUM_BUTTONS         2
#define NUM_MOUSEBITMAPS    9

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;
typedef short sword;                  /* signed word */

byte *VGA = (byte *)0xA0000;          /* this points to video memory. */
word *my_clock = (word *)0x046C;      /* this points to the 18.2hz system
                                         clock. */

typedef struct                         /* the structure for a bitmap. */
{
  word width;
  word height;
  byte palette[256*3];
  byte *data;
} BITMAP;
                                      /* the structure for animated
                                         mouse pointers. */
typedef struct tagMOUSEBITMAP MOUSEBITMAP;
struct tagMOUSEBITMAP
{
  int hot_x;
  int hot_y;
  byte data[MOUSE_SIZE];
  MOUSEBITMAP *next;   /* points to the next mouse bitmap, if any */
};

typedef struct            /* the structure for a mouse. */
{
  byte on;
  byte button1;
  byte button2;
  byte button3;
  int num_buttons;
  sword x;
  sword y;
  byte under[MOUSE_SIZE];
  MOUSEBITMAP *bmp;

} MOUSE;
```

```
typedef struct              /* the structure for a button. */
{
  int x;
  int y;
  int state;
  byte bitmap[BUTTON_BITMAPS][BUTTON_SIZE];

} BUTTON;

/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory.                                    *
 **************************************************************************/

void load_bmp(char *file,BITMAP *b)
{
  FILE *fp;
  long index;
  word num_colors;
  int x;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }

  /* read in the width and height of the image, and the
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;

  /* try to allocate memory */
  if ((b->data = (byte *) malloc((word)(b->width*b->height))) == NULL)
  {
    fclose(fp);
    printf("Error allocating memory for file %s.\n",file);
    exit(1);
  }

  /* read the palette information */
  for(index=0;index<num_colors;index++)
  {
    b->palette[(int)(index*3+2)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+1)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+0)] = fgetc(fp) >> 2;
    x=fgetc(fp);
  }

  /* read the bitmap */
  for(index=(b->height-1)*b->width;index>=0;index-=b->width)
    for(x=0;x<b->width;x++)
      b->data[(word)(index+x)]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  set_palette                                                           *
 *    Sets all 256 colors of the palette.                                 *
 **************************************************************************/

void set_palette(byte *palette)
{
  int i;

  outp(PALETTE_INDEX,0);              /* tell the VGA that palette data
                                         is coming. */
  for(i=0;i<256*3;i++)
    outp(PALETTE_DATA,palette[i]);    /* write the data */
```

```
}

/**************************************************************************
 *  wait_for_retrace                                                      *
 *    Wait until the *beginning* of a vertical retrace cycle (60hz).      *
 **************************************************************************/

void wait_for_retrace(void)
{
    /* wait until done with vertical retrace */
    while  ((inp(INPUT_STATUS) & VRETRACE));
    /* wait until done refreshing */
    while (!(inp(INPUT_STATUS) & VRETRACE));
}

/**************************************************************************
 *  get_mouse_motion                                                      *
 *    Returns the distance the mouse has moved since it was lasted        *
 *    checked.                                                            *
 **************************************************************************/

void get_mouse_motion(sword *dx, sword *dy)
{
  union REGS regs;

  regs.x.ax = MOUSE_GETMOTION;
  int86(MOUSE_INT, &regs, &regs);
  *dx=regs.x.cx;
  *dy=regs.x.dx;
}

/**************************************************************************
 *  init_mouse                                                            *
 *    Resets the mouse.  Returns 0 if mouse not found.                    *
 **************************************************************************/

sword init_mouse(MOUSE *mouse)
{
  sword dx,dy;
  union REGS regs;

  regs.x.ax = MOUSE_RESET;
  int86(MOUSE_INT, &regs, &regs);
  mouse->on=regs.x.ax;
  mouse->num_buttons=regs.x.bx;
  mouse->button1=0;
  mouse->button2=0;
  mouse->button3=0;
  mouse->x=SCREEN_WIDTH/2;
  mouse->y=SCREEN_HEIGHT/2;
  get_mouse_motion(&dx,&dy);
  return mouse->on;
}

/**************************************************************************
 *  get_mouse_press                                                       *
 *    Returns 1 if a button has been pressed since it was last checked.   *
 **************************************************************************/

sword get_mouse_press(sword button)
{
  union REGS regs;

  regs.x.ax = MOUSE_GETPRESS;
  regs.x.bx = button;
  int86(MOUSE_INT, &regs, &regs);
  return regs.x.bx;
}

/**************************************************************************
 *  get_mouse_release                                                     *
 *    Returns 1 if a button has been released since it was last checked.  *
 **************************************************************************/

sword get_mouse_release(sword button)
{
  union REGS regs;

  regs.x.ax = MOUSE_GETRELEASE;
  regs.x.bx = button;
  int86(MOUSE_INT, &regs, &regs);
  return regs.x.bx;
}

/**************************************************************************
 *  show_mouse                                                            *
 *    Displays the mouse.  This code is not optimized.                    *
 **************************************************************************/

void show_mouse(MOUSE *mouse)
{
  int x, y;
  int mx = mouse->x - mouse->bmp->hot_x;
  int my = mouse->y - mouse->bmp->hot_y;
  long screen_offset = (my<<8)+(my<<6);
  word bitmap_offset = 0;
  byte data;

  for(y=0;y<MOUSE_HEIGHT;y++)
  {
    for(x=0;x<MOUSE_WIDTH;x++,bitmap_offset++)
    {
      mouse->under[bitmap_offset] = VGA[(word)(screen_offset+mx+x)];
      /* check for screen boundries */
      if (mx+x < SCREEN_WIDTH  && mx+x >= 0 &&
          my+y < SCREEN_HEIGHT && my+y >= 0)
      {
        data = mouse->bmp->data[bitmap_offset];
        if (data) VGA[(word)(screen_offset+mx+x)] = data;
      }
    }
    screen_offset+=SCREEN_WIDTH;
  }
}
```

```
/**************************************************************************
 *  hide_mouse                                                            *
 *    hides the mouse.  This code is not optimized.                       *
 **************************************************************************/

void hide_mouse(MOUSE *mouse)
{
  int x, y;
  int mx = mouse->x - mouse->bmp->hot_x;
  int my = mouse->y - mouse->bmp->hot_y;
  long screen_offset = (my<<8)+(my<<6);
  word bitmap_offset = 0;

  for(y=0;y<MOUSE_HEIGHT;y++)
  {
    for(x=0;x<MOUSE_WIDTH;x++,bitmap_offset++)
    {
      /* check for screen boundries */
      if (mx+x < SCREEN_WIDTH  && mx+x >= 0 &&
          my+y < SCREEN_HEIGHT && my+y >= 0)
      {

        VGA[(word)(screen_offset+mx+x)] = mouse->under[bitmap_offset];
      }
    }

    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  draw_button                                                           *
 *    Draws a button.                                                     *
 **************************************************************************/

void draw_button(BUTTON *button)
{
  int x, y;
  word screen_offset = (button->y<<8)+(button->y<<6);
  word bitmap_offset = 0;
  byte data;

  for(y=0;y<BUTTON_HEIGHT;y++)
  {
    for(x=0;x<BUTTON_WIDTH;x++,bitmap_offset++)
    {
      data = button->bitmap[button->state%BUTTON_BITMAPS][bitmap_offset];
      if (data) VGA[screen_offset+button->x+x] = data;
    }
    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  Main                                                                  *
 **************************************************************************/
```

```
void main()
{
  BITMAP bmp;
  MOUSE  mouse;
  MOUSEBITMAP *mb[NUM_MOUSEBITMAPS],
    *mouse_norm, *mouse_wait, *mouse_new=NULL;

  BUTTON *button[NUM_BUTTONS];
  word redraw;
  sword dx = 0, dy = 0, new_x, new_y;
  word press, release;
  int i,j, done = 0, x,y;
  word last_time;

  if (__djgpp_nearptr_enable() == 0)
  {
    printf("Could get access to first 640K of memory.\n");
    exit(-1);
  }

  VGA+=__djgpp_conventional_base;
  my_clock = (void *)my_clock + __djgpp_conventional_base;

  for (i=0; i<NUM_MOUSEBITMAPS; i++)
  {
    if ((mb[i] = (MOUSEBITMAP *) malloc(sizeof(MOUSEBITMAP))) == NULL)
    {
      printf("Error allocating memory for bitmap.\n");
      exit(1);
    }
  }

  for (i=0; i<NUM_BUTTONS; i++)
  {
    if ((button[i] = (BUTTON *) malloc(sizeof(BUTTON))) == NULL)
    {
      printf("Error allocating memory for bitmap.\n");
      exit(1);
    }
  }
  mouse_norm = mb[0];
  mouse_wait = mb[1];

  mouse.bmp = mouse_norm;

  button[0]->x     = 48;               /* set button states */
  button[0]->y     = 152;
  button[0]->state = STATE_NORM;

  button[1]->x     = 224;
  button[1]->y     = 152;
  button[1]->state = STATE_NORM;

  if (!init_mouse(&mouse))            /* init mouse */
  {
    printf("Mouse not found.\n");
    exit(1);
  }

  load_bmp("images.bmp",&bmp);        /* load icons */
  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */


  for(i=0;i<NUM_MOUSEBITMAPS;i++)     /* copy mouse pointers */
    for(y=0;y<MOUSE_HEIGHT;y++)
      for(x=0;x<MOUSE_WIDTH;x++)
      {
        mb[i]->data[x+y*MOUSE_WIDTH] = bmp.data[i*MOUSE_WIDTH+x+y*bmp.width];
        mb[i]->next = mb[i+1];
        mb[i]->hot_x = 12;
        mb[i]->hot_y = 12;
      }

  mb[0]->next  = mb[0];
  mb[8]->next  = mb[1];
  mb[0]->hot_x = 7;
  mb[0]->hot_y = 2;
                                      /* copy button bitmaps */
  for(i=0;i<NUM_BUTTONS;i++)
    for(j=0;j<BUTTON_BITMAPS;j++)
      for(y=0;y<BUTTON_HEIGHT;y++)
        for(x=0;x<BUTTON_WIDTH;x++)
        {
          button[i]->bitmap[j][x+y*BUTTON_WIDTH] =
            bmp.data[i*(bmp.width>>1) + j*BUTTON_WIDTH + x +
            (BUTTON_HEIGHT+y)*bmp.width];
        }

  free(bmp.data);                     /* free up memory used */

  set_palette(bmp.palette);

  for(y=0;y<SCREEN_HEIGHT;y++)        /* display a background */
    for(x=0;x<SCREEN_WIDTH;x++)
      VGA[(y<<8)+(y<<6)+x]=y;

  new_x=mouse.x;
  new_y=mouse.y;
  redraw=0xFFFF;
  show_mouse(&mouse);
  last_time=*my_clock;
  while (!done)                       /* start main loop */
  {
    if (redraw)                       /* draw the mouse only as needed */
    {
      wait_for_retrace();
      hide_mouse(&mouse);
      if (redraw>1)
      {
        for(i=0;i<NUM_BUTTONS;i++)
          if (redraw & (2<<i)) draw_button(button[i]);
      }
      if (mouse_new!=NULL) mouse.bmp=mouse_new;
      mouse.x=new_x;
      mouse.y=new_y;
      show_mouse(&mouse);
      last_time=*my_clock;
      redraw=0;
      mouse_new=NULL;
    }

    do {                              /* check mouse status */
      get_mouse_motion(&dx,&dy);
      press   = get_mouse_press(LEFT_BUTTON);
      release = get_mouse_release(LEFT_BUTTON);
    } while (dx==0 && dy==0 && press==0 && release==0 &&
      *my_clock==last_time);

    if (*my_clock!=last_time)         /* check animation clock */
    {
      if (mouse.bmp!=mouse.bmp->next)
      {
        redraw=1;
        mouse.bmp = mouse.bmp->next;
      }
      else
        last_time = *my_clock;
    }

    if (press) mouse.button1=1;
    if (release) mouse.button1=0;

    if (dx || dy)                     /* calculate movement */
    {
      new_x = mouse.x+dx;
      new_y = mouse.y+dy;
      if (new_x<0)   new_x=0;
      if (new_y<0)   new_y=0;
      if (new_x>319) new_x=319;
      if (new_y>199) new_y=199;
      redraw=1;
    }

    for(i=0;i<NUM_BUTTONS;i++)        /* check button status */
    {
      if (new_x >= button[i]->x && new_x < button[i]->x+48 &&
          new_y >= button[i]->y && new_y < button[i]->y+24)
      {
        if (release && button[i]->state==STATE_PRESSED)
        {
          button[i]->state=STATE_ACTIVE;
          redraw|=(2<<i);
          if (i==0)
          {
            if (mouse.bmp==mouse_norm)
              mouse_new=mouse_wait;
            else
              mouse_new=mouse_norm;
          }
          else if (i==1) done=1;
        }
        else if (press)
        {
          button[i]->state=STATE_PRESSED;
          redraw|=(2<<i);
        }
        else if (button[i]->state==STATE_NORM && mouse.button1==0)
        {
          button[i]->state=STATE_ACTIVE;
          redraw|=(2<<i);
        }
        else if (button[i]->state==STATE_WAITING)
        {
          if (mouse.button1)
          {
            button[i]->state=STATE_PRESSED;
          }
          else
          {
            button[i]->state=STATE_ACTIVE;
          }
          redraw|=(2<<i);
        }
      }
      else if (button[i]->state==STATE_ACTIVE)
      {
        button[i]->state=STATE_NORM;
        redraw|=(2<<i);
      }
      else if (button[i]->state==STATE_PRESSED && mouse.button1)
      {
        button[i]->state=STATE_WAITING;
        redraw|=(2<<i);
      }
      else if (button[i]->state==STATE_WAITING && release)
      {
        button[i]->state=STATE_NORM;
        redraw|=(2<<i);
      }

    }
  }                                   /* end while loop */

  for (i=0; i<NUM_MOUSEBITMAPS; i++)  /* free allocated memory */
  {
    free(mb[i]);
  }

  for (i=0; i<NUM_BUTTONS; i++)       /* free allocated memory */
  {
    free(button[i]);
  }

  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */
  __djgpp_nearptr_disable();
```

```
  return;
}
```

« Back to Mouse Support & Animation

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

256-Color VGA Programming in C > Mouse Support & Animation

mouse.c

View as plain text

```
/**************************************************************************
 * mouse.c                                                                *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * This is a 16-bit program.                                              *
 * Tab stops are set to 2.                                                *
 * Remember to compile in the LARGE memory model!                         *
 * To compile in Borland C: bcc -ml mouse.c                               *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA or compatible video adapter.                             *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demostrates mouse functions.                 *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */

#define PALETTE_INDEX       0x03c8
#define PALETTE_DATA        0x03c9
#define INPUT_STATUS        0x03da
#define VRETRACE            0x08

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

#define MOUSE_INT           0x33
#define MOUSE_RESET         0x00
#define MOUSE_GETPRESS      0x05
#define MOUSE_GETRELEASE    0x06
#define MOUSE_GETMOTION     0x0B
#define LEFT_BUTTON         0x00
#define RIGHT_BUTTON        0x01
#define MIDDLE_BUTTON       0x02
```

```
#define MOUSE_WIDTH         24
#define MOUSE_HEIGHT        24
#define MOUSE_SIZE          (MOUSE_HEIGHT*MOUSE_WIDTH)

#define BUTTON_WIDTH        48
#define BUTTON_HEIGHT       24
#define BUTTON_SIZE         (BUTTON_HEIGHT*BUTTON_WIDTH)
#define BUTTON_BITMAPS      3
#define STATE_NORM          0
#define STATE_ACTIVE        1
#define STATE_PRESSED       2
#define STATE_WAITING       3

#define NUM_BUTTONS         2
#define NUM_MOUSEBITMAPS    9

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;
typedef short sword;                  /* signed word */

byte *VGA=(byte *)0xA0000000L;        /* this points to video memory. */
word *my_clock=(word *)0x0000046C;    /* this points to the 18.2hz system
                                         clock. */

typedef struct                         /* the structure for a bitmap. */
{
  word width;
  word height;
  byte palette[256*3];
  byte *data;
} BITMAP;
                                      /* the structure for animated
                                         mouse pointers. */
typedef struct tagMOUSEBITMAP MOUSEBITMAP;
struct tagMOUSEBITMAP
{
  int hot_x;
  int hot_y;
  byte data[MOUSE_SIZE];
  MOUSEBITMAP *next;   /* points to the next mouse bitmap, if any */
};

typedef struct            /* the structure for a mouse. */
{
  byte on;
  byte button1;
  byte button2;
  byte button3;
  int num_buttons;
  sword x;
  sword y;
  byte under[MOUSE_SIZE];
  MOUSEBITMAP *bmp;

} MOUSE;
```

```
typedef struct              /* the structure for a button. */
{
  int x;
  int y;
  int state;
  byte bitmap[BUTTON_BITMAPS][BUTTON_SIZE];

} BUTTON;

/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory.                                    *
 **************************************************************************/

void load_bmp(char *file,BITMAP *b)
{
  FILE *fp;
  long index;
  word num_colors;
  int x;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }

  /* read in the width and height of the image, and the
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;

  /* try to allocate memory */
  if ((b->data = (byte *) malloc((word)(b->width*b->height))) == NULL)
  {
    fclose(fp);
    printf("Error allocating memory for file %s.\n",file);
    exit(1);
  }

  /* read the palette information */
  for(index=0;index<num_colors;index++)
  {
    b->palette[(int)(index*3+2)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+1)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+0)] = fgetc(fp) >> 2;
    x=fgetc(fp);
  }

  /* read the bitmap */
  for(index=(b->height-1)*b->width;index>=0;index-=b->width)
    for(x=0;x<b->width;x++)
      b->data[(word)(index+x)]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  set_palette                                                           *
 *    Sets all 256 colors of the palette.                                 *
 **************************************************************************/

void set_palette(byte *palette)
{
  int i;

  outp(PALETTE_INDEX,0);              /* tell the VGA that palette data
                                         is coming. */
  for(i=0;i<256*3;i++)
    outp(PALETTE_DATA,palette[i]);    /* write the data */
```

```
}

/**************************************************************************
 *  wait_for_retrace                                                      *
 *    Wait until the *beginning* of a vertical retrace cycle (60hz).      *
 **************************************************************************/

void wait_for_retrace(void)
{
    /* wait until done with vertical retrace */
    while  ((inp(INPUT_STATUS) & VRETRACE));
    /* wait until done refreshing */
    while (!(inp(INPUT_STATUS) & VRETRACE));
}

/**************************************************************************
 *  get_mouse_motion                                                      *
 *    Returns the distance the mouse has moved since it was lasted        *
 *    checked.                                                            *
 **************************************************************************/

void get_mouse_motion(sword *dx, sword *dy)
{
  union REGS regs;

  regs.x.ax = MOUSE_GETMOTION;
  int86(MOUSE_INT, &regs, &regs);
  *dx=regs.x.cx;
  *dy=regs.x.dx;
}

/**************************************************************************
 *  init_mouse                                                            *
 *    Resets the mouse.  Returns 0 if mouse not found.                    *
 **************************************************************************/

sword init_mouse(MOUSE *mouse)
{
  sword dx,dy;
  union REGS regs;

  regs.x.ax = MOUSE_RESET;
  int86(MOUSE_INT, &regs, &regs);
  mouse->on=regs.x.ax;
  mouse->num_buttons=regs.x.bx;
  mouse->button1=0;
  mouse->button2=0;
  mouse->button3=0;
  mouse->x=SCREEN_WIDTH/2;
  mouse->y=SCREEN_HEIGHT/2;
  get_mouse_motion(&dx,&dy);
  return mouse->on;
}

/**************************************************************************
 *  get_mouse_press                                                       *
 *    Returns 1 if a button has been pressed since it was last checked.   *
 ***********************************************************************/

sword get_mouse_press(sword button)
{
  union REGS regs;

  regs.x.ax = MOUSE_GETPRESS;
  regs.x.bx = button;
  int86(MOUSE_INT, &regs, &regs);
  return regs.x.bx;
}

/***********************************************************************
 *  get_mouse_release                                                  *
 *    Returns 1 if a button has been released since it was last checked.  *
 ***********************************************************************/

sword get_mouse_release(sword button)
{
  union REGS regs;

  regs.x.ax = MOUSE_GETRELEASE;
  regs.x.bx = button;
  int86(MOUSE_INT, &regs, &regs);
  return regs.x.bx;
}

/***********************************************************************
 *  show_mouse                                                         *
 *    Displays the mouse.  This code is not optimized.                 *
 ***********************************************************************/

void show_mouse(MOUSE *mouse)
{
  int x, y;
  int mx = mouse->x - mouse->bmp->hot_x;
  int my = mouse->y - mouse->bmp->hot_y;
  long screen_offset = (my<<8)+(my<<6);
  word bitmap_offset = 0;
  byte data;

  for(y=0;y<MOUSE_HEIGHT;y++)
  {
    for(x=0;x<MOUSE_WIDTH;x++,bitmap_offset++)
    {
      mouse->under[bitmap_offset] = VGA[(word)(screen_offset+mx+x)];
      /* check for screen boundries */
      if (mx+x < SCREEN_WIDTH  && mx+x >= 0 &&
          my+y < SCREEN_HEIGHT && my+y >= 0)
      {
        data = mouse->bmp->data[bitmap_offset];
        if (data) VGA[(word)(screen_offset+mx+x)] = data;
      }
    }
    screen_offset+=SCREEN_WIDTH;
  }
}
```

```
/**************************************************************************
 *  hide_mouse                                                            *
 *    hides the mouse.  This code is not optimized.                       *
 **************************************************************************/

void hide_mouse(MOUSE *mouse)
{
  int x, y;
  int mx = mouse->x - mouse->bmp->hot_x;
  int my = mouse->y - mouse->bmp->hot_y;
  long screen_offset = (my<<8)+(my<<6);
  word bitmap_offset = 0;

  for(y=0;y<MOUSE_HEIGHT;y++)
  {
    for(x=0;x<MOUSE_WIDTH;x++,bitmap_offset++)
    {
      /* check for screen boundries */
      if (mx+x < SCREEN_WIDTH  && mx+x >= 0 &&
          my+y < SCREEN_HEIGHT && my+y >= 0)
      {

        VGA[(word)(screen_offset+mx+x)] = mouse->under[bitmap_offset];
      }
    }

    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  draw_button                                                           *
 *    Draws a button.                                                     *
 **************************************************************************/

void draw_button(BUTTON *button)
{
  int x, y;
  word screen_offset = (button->y<<8)+(button->y<<6);
  word bitmap_offset = 0;
  byte data;

  for(y=0;y<BUTTON_HEIGHT;y++)
  {
    for(x=0;x<BUTTON_WIDTH;x++,bitmap_offset++)
    {
      data = button->bitmap[button->state%BUTTON_BITMAPS][bitmap_offset];
      if (data) VGA[screen_offset+button->x+x] = data;
    }
    screen_offset+=SCREEN_WIDTH;
  }
}

/**************************************************************************
 *  Main                                                                  *
 **************************************************************************/
```

```
void main()
{
  BITMAP bmp;
  MOUSE  mouse;
  MOUSEBITMAP *mb[NUM_MOUSEBITMAPS],
    *mouse_norm, *mouse_wait, *mouse_new=NULL;

  BUTTON *button[NUM_BUTTONS];
  word redraw;
  sword dx = 0, dy = 0, new_x, new_y;
  word press, release;
  int i,j, done = 0, x,y;
  word last_time;

  for (i=0; i<NUM_MOUSEBITMAPS; i++)
  {
    if ((mb[i] = (MOUSEBITMAP *) malloc(sizeof(MOUSEBITMAP))) == NULL)
    {
      printf("Error allocating memory for bitmap.\n");
      exit(1);
    }
  }

  for (i=0; i<NUM_BUTTONS; i++)
  {
    if ((button[i] = (BUTTON *) malloc(sizeof(BUTTON))) == NULL)
    {
      printf("Error allocating memory for bitmap.\n");
      exit(1);
    }
  }
  mouse_norm = mb[0];
  mouse_wait = mb[1];

  mouse.bmp = mouse_norm;

  button[0]->x     = 48;               /* set button states */
  button[0]->y     = 152;
  button[0]->state = STATE_NORM;

  button[1]->x     = 224;
  button[1]->y     = 152;
  button[1]->state = STATE_NORM;

  if (!init_mouse(&mouse))            /* init mouse */
  {
    printf("Mouse not found.\n");
    exit(1);
  }

  load_bmp("images.bmp",&bmp);        /* load icons */
  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */


  for(i=0;i<NUM_MOUSEBITMAPS;i++)     /* copy mouse pointers */
    for(y=0;y<MOUSE_HEIGHT;y++)
      for(x=0;x<MOUSE_WIDTH;x++)
      {
        mb[i]->data[x+y*MOUSE_WIDTH] = bmp.data[i*MOUSE_WIDTH+x+y*bmp.width];
        mb[i]->next = mb[i+1];
        mb[i]->hot_x = 12;
        mb[i]->hot_y = 12;
      }

  mb[0]->next  = mb[0];
  mb[8]->next  = mb[1];
  mb[0]->hot_x = 7;
  mb[0]->hot_y = 2;
                                      /* copy button bitmaps */
  for(i=0;i<NUM_BUTTONS;i++)
    for(j=0;j<BUTTON_BITMAPS;j++)
      for(y=0;y<BUTTON_HEIGHT;y++)
        for(x=0;x<BUTTON_WIDTH;x++)
        {
          button[i]->bitmap[j][x+y*BUTTON_WIDTH] =
            bmp.data[i*(bmp.width>>1) + j*BUTTON_WIDTH + x +
            (BUTTON_HEIGHT+y)*bmp.width];
        }

  free(bmp.data);                     /* free up memory used */

  set_palette(bmp.palette);

  for(y=0;y<SCREEN_HEIGHT;y++)        /* display a background */
    for(x=0;x<SCREEN_WIDTH;x++)
      VGA[(y<<8)+(y<<6)+x]=y;

  new_x=mouse.x;
  new_y=mouse.y;
  redraw=0xFFFF;
  show_mouse(&mouse);
  last_time=*my_clock;
  while (!done)                       /* start main loop */
  {
    if (redraw)                       /* draw the mouse only as needed */
    {
      wait_for_retrace();
      hide_mouse(&mouse);
      if (redraw>1)
      {
        for(i=0;i<NUM_BUTTONS;i++)
          if (redraw & (2<<i)) draw_button(button[i]);
      }
      if (mouse_new!=NULL) mouse.bmp=mouse_new;
      mouse.x=new_x;
      mouse.y=new_y;
      show_mouse(&mouse);
      last_time=*my_clock;
      redraw=0;
      mouse_new=NULL;
    }

    do {                              /* check mouse status */
```

```
    get_mouse_motion(&dx,&dy);
    press   = get_mouse_press(LEFT_BUTTON);
    release = get_mouse_release(LEFT_BUTTON);
  } while (dx==0 && dy==0 && press==0 && release==0 &&
    *my_clock==last_time);

  if (*my_clock!=last_time)         /* check animation clock */
  {
    if (mouse.bmp!=mouse.bmp->next)
    {
      redraw=1;
      mouse.bmp = mouse.bmp->next;
    }
    else
      last_time = *my_clock;
  }

  if (press) mouse.button1=1;
  if (release) mouse.button1=0;

  if (dx || dy)                     /* calculate movement */
  {
    new_x = mouse.x+dx;
    new_y = mouse.y+dy;
    if (new_x<0)   new_x=0;
    if (new_y<0)   new_y=0;
    if (new_x>319) new_x=319;
    if (new_y>199) new_y=199;
    redraw=1;
  }

  for(i=0;i<NUM_BUTTONS;i++)        /* check button status */
  {
    if (new_x >= button[i]->x && new_x < button[i]->x+48 &&
        new_y >= button[i]->y && new_y < button[i]->y+24)
    {
      if (release && button[i]->state==STATE_PRESSED)
      {
        button[i]->state=STATE_ACTIVE;
        redraw|=(2<<i);
        if (i==0)
        {
          if (mouse.bmp==mouse_norm)
            mouse_new=mouse_wait;
          else
            mouse_new=mouse_norm;
        }
        else if (i==1) done=1;
      }
      else if (press)
      {
        button[i]->state=STATE_PRESSED;
        redraw|=(2<<i);
      }
      else if (button[i]->state==STATE_NORM && mouse.button1==0)
      {
        button[i]->state=STATE_ACTIVE;
```

```
          redraw|=(2<<i);
        }
        else if (button[i]->state==STATE_WAITING)
        {
          if (mouse.button1)
          {
            button[i]->state=STATE_PRESSED;
          }
          else
          {
            button[i]->state=STATE_ACTIVE;
          }
          redraw|=(2<<i);
        }
      }
      else if (button[i]->state==STATE_ACTIVE)
      {
        button[i]->state=STATE_NORM;
        redraw|=(2<<i);
      }
      else if (button[i]->state==STATE_PRESSED && mouse.button1)
      {
        button[i]->state=STATE_WAITING;
        redraw|=(2<<i);
      }
      else if (button[i]->state==STATE_WAITING && release)
      {
        button[i]->state=STATE_NORM;
        redraw|=(2<<i);
      }

    }
  }                                   /* end while loop */

  for (i=0; i<NUM_MOUSEBITMAPS; i++)  /* free allocated memory */
  {
    free(mb[i]);
  }

  for (i=0; i<NUM_BUTTONS; i++)       /* free allocated memory */
  {
    free(button[i]);
  }

  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */

  return;
}
```

« Back to Mouse Support & Animation

- Home
- Introduction
- VGA Basics

- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

256-Color VGA Programming in C

Double Buffering, Page Flipping, & Unchained Mode

Contents in this section:

Why double buffering and/or page flipping?

Two important concepts used in many games and multimedia applications are double buffering and page flipping. Programmers primarily use these techniques for two purposes:

- to keep the user from seeing objects being drawn onto the screen
- to eliminate flickering.

Double buffering

Double Buffering is a fairly simple concept to grasp. Instead of drawing directly to video memory, the program draws everything to a double buffer (Figure 20a). When finished, the program copies the double buffer to video memory all at once (Figure 20b). At that point the program clears the double buffer (if necessary) and the process starts over.

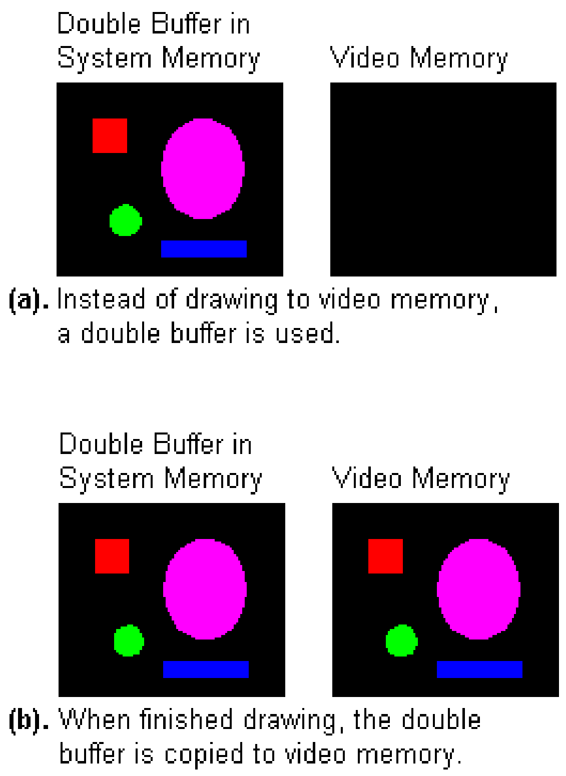
Double Buffer in System Memory
Video Memory
(a). Instead of drawing to video memory, a double buffer is used.
Double Buffer in System Memory
Video Memory
(b). When finished drawing, the double buffer is copied to video memory.

Figure 20. Double buffering concept.

Implementing a double buffer is fairly simple as well. The double buffer is generally the same size as the screen. In mode 0x13, the double buffer would be 64,000 bytes. When the program begins it allocates memory for the double buffer.

```
unsigned char *double_buffer;

...

double_buffer = (unsigned char *) malloc(320*200);
if (double_buffer==NULL)
{
  printf("Not enough memory for double buffer.\n");
  exit(1);
}
```

Then, instead of writing to video memory, the program writes to the double buffer.

```
/* plot a pixel in the double buffer */
double_buffer[(y<<8) + (y<<6) + x] = color;
```

When finished, the program copies the double buffer to video memory (with careful consideration of the vertical retrace to eliminate flickering).

```
while ((inp(INPUT_STATUS_1) & VRETRACE));
while (!(inp(INPUT_STATUS_1) & VRETRACE));
memcpy(VGA,double_buffer,320*200);
```

Using a double buffer would be faster if, instead of having to copy the information from the double buffer to video memory (address 0xA000:0000), the video card could be programmed to get video data directly from the double buffer rather than from its regular address (0xA000:0000). While this is not possible on the VGA, it is close to how page flipping works.

Page flipping

With page flipping, there must be enough video memory for two screens. So, if the screen size is 320x200 at 256 colors, 2*320*200 or 128,000 bytes of video memory must be available. Instead of drawing to the *visible* area in video memory, or *visible page*, the program draws to the *non-visible page* (Figure 21a). When finished, the program swaps the *visible page* pointer with the *non-visible page* pointer (Figure 21b). Now the program clears the newly placed *non-visible page* (if necessary) and the process starts over.

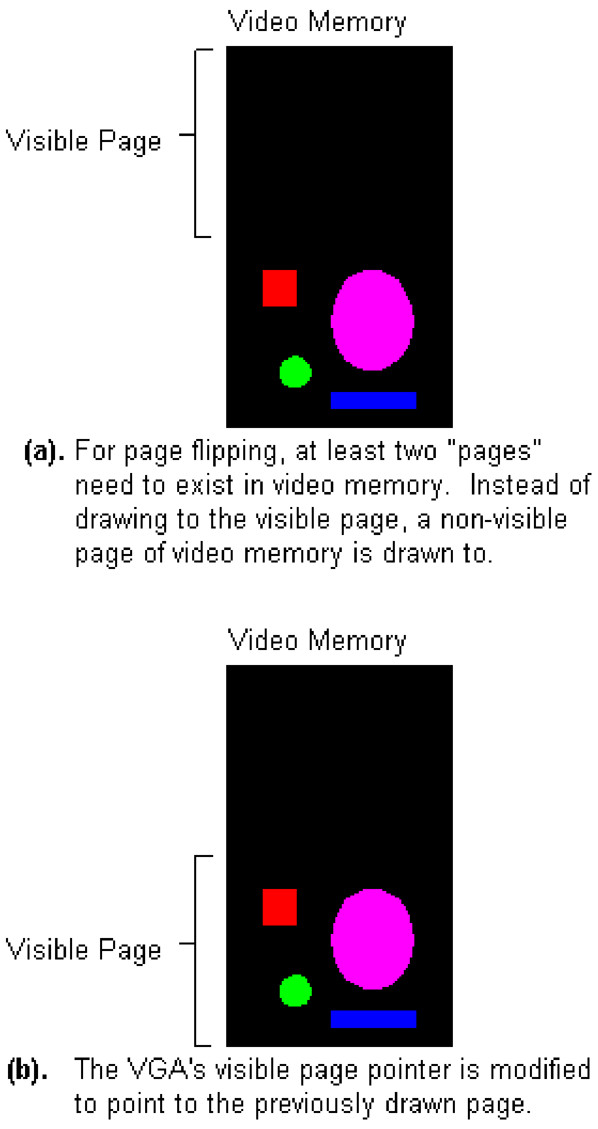
Video Memory
Visible Page
(a). For page flipping, at least two "pages" need to exist in video memory. Instead of drawing to the visible page, a non-visible page of video memory is drawn to.
Video Memory
Visible Page
(b). The VGA's visible page pointer is modified to point to the previously drawn page.

Figure 21. Page flipping concept.

One problem is this: in mode 0x13, only 64K of video memory is available, even if the video card has more memory on it. Even if it is a 4MB video card, mode 0x13 can only access 64K. There is a way, however, to tweak mode 0x13 into a 256-color mode that has a total of 256K of video memory, so that page flipping is possible. This undocumented mode is sometimes referred to as “mode-x,” or “unchained mode.”

Structure of unchained mode

The VGA card has 256K of memory. Many SVGA cards have much more, but even on those cards, VGA modes can only access the first 256K-except for mode 0x13, which can only access 64K. The reason is that mode 0x13 is a chain-4 mode, which basically means only every forth byte of video memory is used. The reason for this is because the linear structure of the video memory allowed fast and easy video memory access. Turning off chain-4 mode allows the program to access of all 256K of video memory, but involves more complicated programming.

In unchained mode, memory exists in four 64K *planes*. Each plane corresponds to a specific column of video memory: plane 0 contains pixels 0, 4, 8, etc.; plane 1 contains pixels 1, 5, 9, etc.; plane 2 contains columns 2, 6, 10, etc.; and plane 3 contains columns 3, 7, 11, etc. (Figure 22). So to plot a pixel at position (5,7), plane 1 is selected, and the offset is (320*7+5)/4 = 561.

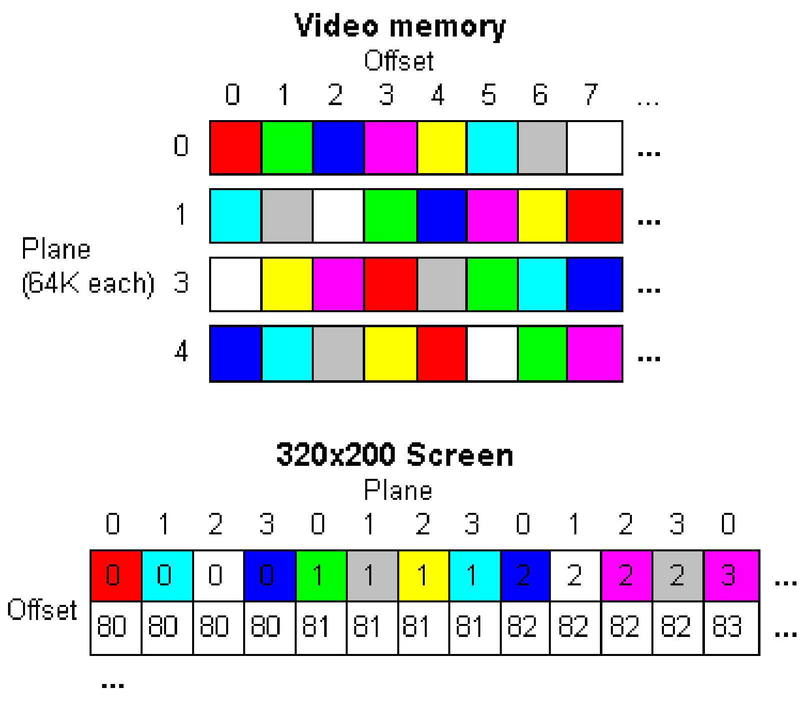

Figure 22. How video memory relates to the screen.

Tweaking mode 0x13

Since unchained mode is not a standard VGA mode, it cannot be set using a BIOS function call. Instead, certain VGA registers have to be tweaked. It involves two VGA controllers: the sequence controller (port 0x3C4) and the CRT controller (port 0x3D4).

```
/* VGA sequence controller */
#define SC_INDEX      0x03c4
#define SC_DATA       0x03c5

/* VGA CRT controller */
#define CRTC_INDEX    0x03d4
#define CRTC_DATA     0x03d5

#define MEMORY_MODE   0x04
#define UNDERLINE_LOC 0x14
#define MODE_CONTROL  0x17

...
```

```
/* turn off chain-4 mode */
outp(SC_INDEX, MEMORY_MODE);
outp(SC_DATA, 0x06);

/* TODO: Insert code to clear the screen here.
   (the BIOS only sets every fourth byte
   to zero -- the rest needs to be set to
   zero, too) */

/* turn off long mode */
outp(CRTC_INDEX, UNDERLINE_LOC);
outp(CRTC_DATA, 0x00);
/* turn on byte mode */
outp(CRTC_INDEX, MODE_CONTROL);
outp(CRTC_DATA, 0xe3);
```

The VGA registers can sometimes be fairly complex. For a complete list of the VGA registers, visit PC-GPE Online.

Plotting a pixel in unchained mode

Plotting a pixel in unchained mode is a tad bit more tedious than it is in mode 0x13, because the proper plane has to be selected. To select a plane, write 2^{plane} to the VGA Map Mask register, where *plane* is a value from 0 to 3 (Figure 23).

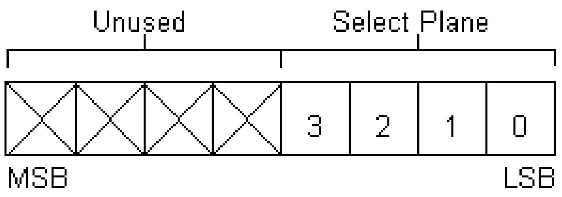

Figure 23. Selecting a plane with the Map Mask register.

The Map Mask register is located at index 2 of the Sequence Controller. To select the Map Mask register, write 2 to the Sequence Controller address at port 0x3C4. Then the Map Mask can be found at the Sequence Controller's data port at port 0x3C5.

```
plane = (x&3);
/* select the map mask register */
outp(0x3c4, 0x02);
/* write 2^plane */
outp(0x3c5, 1 << plane);
```

In mode 0x13, the offset is calculated as 320*y* + *x*. Since unchained mode memory is arranged in four planes, the offset in unchained mode is calculated as $\frac{320y+x}{4}$ (Figure 22).

```
VGA[(y<<6) + (y<<4) + (x>>2)] = color;
```

If a value other than a power of two was used to select a plane, multiple planes would be selected. For example, if 13 (binary 1101) were used, planes 0, 2, and 3 would be selected. That means every plane selected is written with the color value. One use for this is fast screen-clearing. If every plane is selected, only 16,000 bytes need to be written, instead of 64,000 like in mode 0x13.

```
/* set map mask to all 4 planes */
outpw(0x3c4, 0xff02);
memset(VGA,0, 16000);
```

Page flipping in unchained mode

First, set up two word-sized variables to keep track of the visible and non-visible pages. These are offsets to video memory.

```
unsigned int visible_page=0;
unsigned int non_visible_page=320*200/4;
```

Then do all the drawing to the non-visible page. For instance, if a pixel was to be plotted:

```
/* select plane */
outp(SC_INDEX, MAP_MASK);
outp(SC_DATA,  1 << (x&3) );

VGA[non_visible_page+(y<<6)+(y<<4)+(x>>2)]=color;
```

When all the drawing is finished, it is time to switch the pages. The new offset is set through two registers on the CRT controller. The first, 0x0C, sets the upper 8-bits of the offset, and the second, 0x0D, sets the lower 8-bits.

```
/* CRT controller registers */
#define HIGH_ADDRESS 0x0C
#define LOW_ADDRESS  0x0D

...

temp = visible_page;
visible_page = non_visible_page;
non_visible_page = temp;

high_addr=HIGH_ADDRESS | (visible_page & 0xff00);
low_addr =LOW_ADDRESS  | (visible_page << 8);

while ((inp(INPUT_STATUS_1) & VRETRACE));
outpw(CRTC_INDEX, high_addr);
outpw(CRTC_INDEX, low_addr);
while (!(inp(INPUT_STATUS_1) & VRETRACE));
```

Here are some things to consider when using page flipping:

- If the program was using interrupts, it would be advisable to disable interrupts before the page was flipped and re-enable them afterward. If an interrupt occurred at the wrong time, the screen could be temporarily distorted.
- When the offset registers are changed, the page flip does not occur until the end of the *next* vertical retrace. So after the page is flipped, the program should wait until the end of the vertical retrace before drawing to the *non-visible* page.

In the following program, instead of referring to the pages as *visible* and *non-visible* refers to them as *visual* and *active*. It draws animated balls (Figure 24) around the screen using both double buffering and page flipping, and then outputs the results. It defaults to drawing eight balls; a unique number of balls can be drawn by specifying a number at the command prompt. In this example, 16 balls were drawn by using the command `unchain 16`.

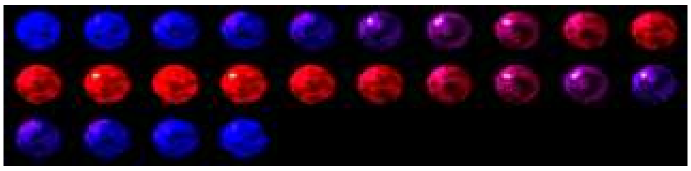

Figure 24. Bitmap `balls.bmp`.

Program: unchain.c

DJGPP 2.0
    View unchain.c
    Download unchain.zip (Contains unchain.c, unchain.exe, balls.bmp)
Borland C, Turbo C, etc.
    View unchain.c
    Download unchain.zip (Contains unchain.c, unchain.exe, balls.bmp)

Having trouble compiling or running the program? See the Troubleshooting page.

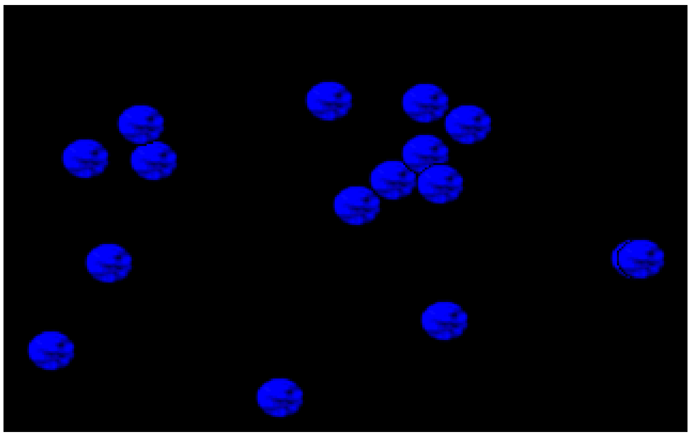

Figure 25. Output from `unchain.exe`.

```
Results with 16 objects:
  Mode 0x13 with double buffering:
    5.989011 seconds,
    23.376147 frames per second.
  Unchained mode with page flipping:
    4.065934 seconds,
    34.432431 frames per second.
  Unchained mode with page flipping was
  1.472973 times faster.
```

Although page flipping in unchained mode was faster than double buffering in mode 0x13 in this example, it is not always faster. This program was created to prove a point: depending on the number of pixels drawn and the number of `outp()`'s or `outpw()`'s used in unchained mode, mode 0x13 can still be faster. The program was tested (ignoring the vertical retrace) on various numbers of objects to show the relationship (Figure 26).

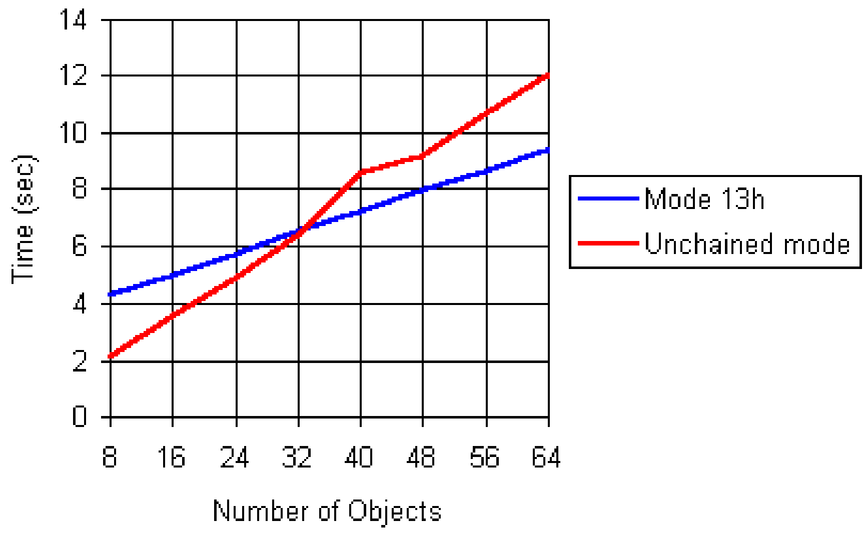

Figure 26. Unchained mode is not always faster.

One of the reasons mode 0x13 is sometimes faster than unchained mode is that for each frame, the selected plane is changed four times for each ball object. The program could have been created to select the plane only four times per frame, which would have increased performance, because `outp()`'s and `outpw()`'s are very slow statements. When designing a program for unchained mode, the number of `outp()`'s and `outpw()`'s used should be limited to as few as possible.

Other unchained modes

The code below will someday be included in another section of this site, but right know it's just here to show you how to program the different unchained modes, like 320x240 and 360x480.

Program: modes.c

This program demonstrates various unchained modes. It supports widths of 320 and 360, and heights of 200, 400, 240, and 480, so there are a total of eight combinations. Setting the mode you want is done like so:

```
set_unchained_mode(320,240);
```

The program also demonstrates planar bitmaps, which speeds things up a bit. Make sure you download `ghosts.bmp` to get the program to work.

DJGPP 2.0
  View modes.c
  Download modes.zip (Contains modes.c, modes.exe, ghosts.bmp)
Borland C, Turbo C, etc.
  View modes.c
  Download modes.zip (Contains modes.c, modes.exe, ghosts.bmp)

Having trouble compiling or running the program? See the Troubleshooting page.

« Previous: Mouse Support & Animation

- Home
- Introduction
- VGA Basics
- Primitive Shapes & Lines
- Bitmaps & Palette Manipulation
- Mouse Support & Animation
- Double Buffering, Page Flipping, & Unchained Mode

- Links and Other Resources
- Download
- FAQ and Troubleshooting

DAVID BRACKEEN

256-Color VGA Programming in C > Double Buffering, Page Flipping, & Unchained Mode

unchain.c

View as plain text

```
/**************************************************************************
 * unchain.c                                                              *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * Tab stops are set to 2.                                                *
 * This program compiles with DJGPP! (www.delorie.com)                    *
 * To compile in DJGPP: gcc unchain.c -o unchain.exe                      *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA, or compatible video adapter.                            *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demonstrates VGA’s unchained mode            *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <sys/nearptr.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */


#define SC_INDEX            0x03c4    /* VGA sequence controller */
#define SC_DATA             0x03c5
#define PALETTE_INDEX       0x03c8    /* VGA digital-to-analog converter */
#define PALETTE_DATA        0x03c9
#define GC_INDEX            0x03ce    /* VGA graphics controller */
#define GC_DATA             0x03cf
#define CRTC_INDEX          0x03d4    /* VGA CRT controller */
#define CRTC_DATA           0x03d5
#define INPUT_STATUS_1      0x03da

#define MAP_MASK            0x02      /* Sequence controller registers */
#define ALL_PLANES          0xff02
#define MEMORY_MODE         0x04

#define LATCHES_ON          0x0008    /* Graphics controller registers */
#define LATCHES_OFF         0xff08
```

```
#define HIGH_ADDRESS        0x0C      /* CRT controller registers */
#define LOW_ADDRESS         0x0D
#define UNDERLINE_LOCATION  0x14
#define MODE_CONTROL        0x17

#define DISPLAY_ENABLE      0x01      /* VGA input status bits */
#define VRETRACE            0x08

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define SCREEN_SIZE         (word)(SCREEN_WIDTH*SCREEN_HEIGHT)
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

#define BITMAP_WIDTH        32
#define BITMAP_HEIGHT       25
#define ANIMATION_FRAMES    24
#define TOTAL_FRAMES        140
#define VERTICAL_RETRACE              /* comment out this line for more
                                         accurate timing */
typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;

byte *VGA = (byte *)0xA0000;          /* this points to video memory. */
word *my_clock = (word *)0x046C;      /* this points to the 18.2hz system
                                         clock. */

typedef struct tagBITMAP              /* the structure for a bitmap. */
{
  word width;
  word height;
  byte palette[256*3];
  byte *data;
} BITMAP;

typedef struct tagOBJECT              /* the structure for a moving object
                                         in 2d space; used for animation */
{
  int x,y;
  int dx,dy;
  byte width,height;
} OBJECT;

/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}

/**************************************************************************
```

```
 *  set_mode                                                          *
 *     Sets the video mode.                                           *
 **********************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**********************************************************************
 *  set_unchained_mode                                                *
 *    resets VGA mode 0x13 to unchained mode to access all 256K of memory *
 **********************************************************************/

void set_unchained_mode(void)
{
  word i;
  dword *ptr=(dword *)VGA;            /* used for faster screen clearing */

  outp(SC_INDEX,  MEMORY_MODE);       /* turn off chain-4 mode */
  outp(SC_DATA,   0x06);

  outpw(SC_INDEX, ALL_PLANES);        /* set map mask to all 4 planes */

  for(i=0;i<0x4000;i++)               /* clear all 256K of memory */
    *ptr++ = 0;

  outp(CRTC_INDEX,UNDERLINE_LOCATION);/* turn off long mode */
  outp(CRTC_DATA, 0x00);

  outp(CRTC_INDEX,MODE_CONTROL);      /* turn on byte mode */
  outp(CRTC_DATA, 0xe3);
}

/**********************************************************************
 *  page_flip                                                         *
 *    switches the pages at the appropriate time and waits for the    *
 *    vertical retrace.                                               *
 **********************************************************************/

void page_flip(word *page1,word *page2)
{
  word high_address,low_address;
  word temp;

  temp=*page1;
  *page1=*page2;
  *page2=temp;

  high_address = HIGH_ADDRESS | (*page1 & 0xff00);
  low_address  = LOW_ADDRESS  | (*page1 << 8);

  #ifdef VERTICAL_RETRACE
```

```
    while ((inp(INPUT_STATUS_1) & DISPLAY_ENABLE));
  #endif
  outpw(CRTC_INDEX, high_address);
  outpw(CRTC_INDEX, low_address);
  #ifdef VERTICAL_RETRACE
    while (!(inp(INPUT_STATUS_1) & VRETRACE));
  #endif
}

/**************************************************************************
 *  show_buffer                                                           *
 *    displays a memory buffer on the screen                              *
 **************************************************************************/

void show_buffer(byte *buffer)
{
  #ifdef VERTICAL_RETRACE
    while ((inp(INPUT_STATUS_1) & VRETRACE));
    while (!(inp(INPUT_STATUS_1) & VRETRACE));
  #endif
  memcpy(VGA,buffer,SCREEN_SIZE);
}

/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory.                                    *
 **************************************************************************/

void load_bmp(char *file,BITMAP *b)
{
  FILE *fp;
  long index;
  word num_colors;
  int x;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }

  /* read in the width and height of the image, and the
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;

  /* try to allocate memory */
  if ((b->data = (byte *) malloc((word)(b->width*b->height))) == NULL)
  {
    fclose(fp);
    printf("Error allocating memory for file %s.\n",file);
    exit(1);
  }

  /* read the palette information */
  for(index=0;index<num_colors;index++)
  {
    b->palette[(int)(index*3+2)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+1)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+0)] = fgetc(fp) >> 2;
    x=fgetc(fp);
  }

  /* read the bitmap */
  for(index = (b->height-1)*b->width; index >= 0;index-=b->width)
    for(x = 0; x < b->width; x++)
      b->data[(int)(index+x)]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  set_palette                                                           *
 *    Sets all 256 colors of the palette.                                 *
 **************************************************************************/

void set_palette(byte *palette)
{
  int i;

  outp(PALETTE_INDEX,0);              /* tell the VGA that palette data
                                         is coming. */
  for(i=0;i<256*3;i++)
    outp(PALETTE_DATA,palette[i]);    /* write the data */
}

/**************************************************************************
 *  plot_pixel                                                            *
 *    Plots a pixel in unchained mode                                     *
 **************************************************************************/

void plot_pixel(int x,int y,byte color)
{
  outp(SC_INDEX, MAP_MASK);          /* select plane */
  outp(SC_DATA,  1 << (x&3) );

  VGA[(y<<6)+(y<<4)+(x>>2)]=color;
```

```
}

/**************************************************************************
 *  Main                                                                  *
 **************************************************************************/

void main(int argc, char *argv[])
{
  word bitmap_offset,screen_offset;
  word visual_page = 0;
  word active_page = SCREEN_SIZE/4;
  word start;
  float t1,t2;
  int i,repeat,plane,num_objects=0;
  word x,y;
  byte *double_buffer;
  BITMAP bmp;
  OBJECT *object;

  if (__djgpp_nearptr_enable() == 0)
  {
    printf("Could get access to first 640K of memory.\n");
    exit(-1);
  }

  VGA+=__djgpp_conventional_base;
  my_clock = (void *)my_clock + __djgpp_conventional_base;

  /* get command-line options */
  if (argc>0) num_objects=atoi(argv[1]);
  if (num_objects<=0) num_objects=8;

  /* allocate memory for double buffer and background image */
  if ((double_buffer = (byte *) malloc(SCREEN_SIZE)) == NULL)
  {
    printf("Not enough memory for double buffer.\n");
    exit(1);
  }
  /* allocate memory for objects */
  if ((object = (OBJECT *) malloc(sizeof(OBJECT)*num_objects)) == NULL)
  {
    printf("Not enough memory for objects.\n");
    free(double_buffer);
    exit(1);
  }

  /* load the images */
  load_bmp("balls.bmp",&bmp);

  /* set the object positions */
  srand(*my_clock);
  for(i=0;i<num_objects;i++)
  {
    object[i].width   = BITMAP_WIDTH;
    object[i].height  = BITMAP_HEIGHT;
    object[i].x       = rand() % (SCREEN_WIDTH - BITMAP_WIDTH );
    object[i].y       = rand() % (SCREEN_HEIGHT- BITMAP_HEIGHT);
    object[i].dx      = (rand()%5) - 2;
    object[i].dy      = (rand()%5) - 2;
  }

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */
  set_palette(bmp.palette);

  start=*my_clock;                    /* record the starting time. */
  for(repeat=0;repeat<TOTAL_FRAMES;repeat++)
  {
    if ((repeat%ANIMATION_FRAMES)==0) bitmap_offset=0;
    /* clear background */
    memset(double_buffer,0,SCREEN_SIZE);

    for(i=0;i<num_objects;i++)
    {
      screen_offset = (object[i].y<<8) + (object[i].y<<6) + object[i].x;
      /* draw the object. */
      for(y=0;y<BITMAP_HEIGHT*bmp.width;y+=bmp.width)
        for(x=0;x<BITMAP_WIDTH;x++)
          if (bmp.data[bitmap_offset+y+x]!=0)
            double_buffer[screen_offset+y+x]=bmp.data[bitmap_offset+y+x];
      /* check to see if the object is within boundries */
      if (object[i].x + object[i].dx < 0 ||
          object[i].x + object[i].dx > SCREEN_WIDTH-object[i].width-1)
            object[i].dx=-object[i].dx;
      if (object[i].y + object[i].dy < 0 ||
          object[i].y + object[i].dy > SCREEN_HEIGHT-object[i].height-1)
            object[i].dy=-object[i].dy;
      /* move the object */
      object[i].x+=object[i].dx;
      object[i].y+=object[i].dy;
    }

    /* point to the next image in the animation */
    bitmap_offset+=BITMAP_WIDTH;
    if ((bitmap_offset%bmp.width)==0)
      bitmap_offset+=bmp.width*(BITMAP_HEIGHT-1);

    /* show the buffer */
    show_buffer(double_buffer);
  }
  t1=(*my_clock-start)/18.2;          /* calculate how long it took. */

  free(double_buffer);                /* free up memory used */

  /************************************************************************/

  set_unchained_mode();               /* set unchained mode */

  start=*my_clock;                    /* record the starting time. */
  for(repeat=0;repeat<TOTAL_FRAMES;repeat++)
  {
    if ((repeat%ANIMATION_FRAMES)==0) bitmap_offset=0;
    /* clear background */
    outpw(SC_INDEX,ALL_PLANES);
    memset(&VGA[active_page],0,SCREEN_SIZE/4);
    outp(SC_INDEX, MAP_MASK);          /* select plane */
    for(i=0;i<num_objects;i++)
    {
      screen_offset = (object[i].y<<6) + (object[i].y<<4) + (object[i].x>>2);
      /* draw the object. */
      for(plane=0;plane<4;plane++)
      {
        /* select plane */
        outp(SC_DATA,  1 << ((plane+object[i].x)&3) );
        for(y=0;y<BITMAP_HEIGHT*bmp.width;y+=bmp.width)
          for(x=plane;x<BITMAP_WIDTH;x+=4)
            if (bmp.data[bitmap_offset+y+x]!=0)
              VGA[active_page+screen_offset+(y>>2)+((x+(object[i].x&3)) >> 2)]=
                bmp.data[bitmap_offset+y+x];
      }
      /* check to see if the object is within boundries */
      if (object[i].x + object[i].dx < 0 ||
          object[i].x + object[i].dx > SCREEN_WIDTH-object[i].width-1)
            object[i].dx=-object[i].dx;
      if (object[i].y + object[i].dy < 0 ||
          object[i].y + object[i].dy > SCREEN_HEIGHT-object[i].height-1)
            object[i].dy=-object[i].dy;
      /* move the object */
      object[i].x+=object[i].dx;
      object[i].y+=object[i].dy;
    }

    /* point to the next image in the animation */
    bitmap_offset+=BITMAP_WIDTH;
    if ((bitmap_offset%bmp.width)==0)
      bitmap_offset+=bmp.width*(BITMAP_HEIGHT-1);

    /* flip the pages */
    page_flip(&visual_page,&active_page);
  }
  t2=(*my_clock-start)/18.2;          /* calculate how long it took. */

  free(bmp.data);
  free(object);

  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */
  /* output the results... */

  printf("Results with %i objects",num_objects);
  #ifdef VERTICAL_RETRACE
    printf(":\n");
  #else
    printf(" (vertical retrace *ignored*):\n");
  #endif
  printf("  Mode  0x13 with double buffering:\n");
  printf("    %f seconds,\n",t1);
  printf("    %f frames per second.\n",(float)TOTAL_FRAMES/t1);
  printf("  Unchained mode with page flipping:\n");
  printf("    %f seconds,\n",t2);
  printf("    %f frames per second.\n",(float)TOTAL_FRAMES/t2);
  if (t2 != 0)
    printf("  Unchained mode with page flipping was %f times faster.\n",t1/t2);

  __djgpp_nearptr_disable();
  printf("%p\n",my_clock);

  return;
}
```

« Back to Double Buffering, Page Flipping, & Unchained Mode

DAVID BRACKEEN

unchain.c

View as plain text

```
/**************************************************************************
 * unchain.c                                                              *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * This is a 16-bit program.                                              *
 * Tab stops are set to 2.                                                *
 * Remember to compile in the LARGE memory model!                         *
 * To compile in Borland C: bcc -ml unchain.c                             *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA or compatible video adapter.                             *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demonstrates VGA’s unchained mode            *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <mem.h>


#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */


#define SC_INDEX            0x03c4    /* VGA sequence controller */
#define SC_DATA             0x03c5
#define PALETTE_INDEX       0x03c8    /* VGA digital-to-analog converter */
#define PALETTE_DATA        0x03c9
#define GC_INDEX            0x03ce    /* VGA graphics controller */
#define GC_DATA             0x03cf
#define CRTC_INDEX          0x03d4    /* VGA CRT controller */
#define CRTC_DATA           0x03d5
#define INPUT_STATUS_1      0x03da

#define MAP_MASK            0x02      /* Sequence controller registers */
#define ALL_PLANES          0xff02
#define MEMORY_MODE         0x04
```

```
#define LATCHES_ON          0x0008    /* Graphics controller registers */
#define LATCHES_OFF         0xff08

#define HIGH_ADDRESS        0x0C      /* CRT controller registers */
#define LOW_ADDRESS         0x0D
#define UNDERLINE_LOCATION  0x14
#define MODE_CONTROL        0x17

#define DISPLAY_ENABLE      0x01      /* VGA input status bits */
#define VRETRACE            0x08

#define SCREEN_WIDTH        320       /* width in pixels of mode 0x13 */
#define SCREEN_HEIGHT       200       /* height in pixels of mode 0x13 */
#define SCREEN_SIZE         (word)(SCREEN_WIDTH*SCREEN_HEIGHT)
#define NUM_COLORS          256       /* number of colors in mode 0x13 */

#define BITMAP_WIDTH        32
#define BITMAP_HEIGHT       25
#define ANIMATION_FRAMES    24
#define TOTAL_FRAMES        140
#define VERTICAL_RETRACE              /* comment out this line for more
                                         accurate timing */
typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;

byte *VGA=(byte *)0xA0000000L;        /* this points to video memory. */
word *my_clock=(word *)0x0000046C;    /* this points to the 18.2hz system
                                         clock. */

typedef struct tagBITMAP              /* the structure for a bitmap. */
{
  word width;
  word height;
  byte palette[256*3];
  byte *data;
} BITMAP;

typedef struct tagOBJECT              /* the structure for a moving object
                                         in 2d space; used for animation */
{
  int x,y;
  int dx,dy;
  byte width,height;
} OBJECT;

/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}
```

```
/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/

void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  set_unchained_mode                                                    *
 *    resets VGA mode 0x13 to unchained mode to access all 256K of memory *
 **************************************************************************/

void set_unchained_mode(void)
{
  word i;
  dword *ptr=(dword *)VGA;            /* used for faster screen clearing */

  outp(SC_INDEX,  MEMORY_MODE);       /* turn off chain-4 mode */
  outp(SC_DATA,   0x06);

  outpw(SC_INDEX, ALL_PLANES);        /* set map mask to all 4 planes */

  for(i=0;i<0x4000;i++)               /* clear all 256K of memory */
    *ptr++ = 0;

  outp(CRTC_INDEX,UNDERLINE_LOCATION);/* turn off long mode */
  outp(CRTC_DATA, 0x00);

  outp(CRTC_INDEX,MODE_CONTROL);      /* turn on byte mode */
  outp(CRTC_DATA, 0xe3);
}

/**************************************************************************
 *  page_flip                                                             *
 *    switches the pages at the appropriate time and waits for the        *
 *    vertical retrace.                                                   *
 **************************************************************************/

void page_flip(word *page1,word *page2)
{
  word high_address,low_address;
  word temp;

  temp=*page1;
  *page1=*page2;
  *page2=temp;

  high_address = HIGH_ADDRESS | (*page1 & 0xff00);
  low_address  = LOW_ADDRESS  | (*page1 << 8);
  #ifdef VERTICAL_RETRACE
    while ((inp(INPUT_STATUS_1) & DISPLAY_ENABLE));
  #endif
  outpw(CRTC_INDEX, high_address);
  outpw(CRTC_INDEX, low_address);
  #ifdef VERTICAL_RETRACE
    while (!(inp(INPUT_STATUS_1) & VRETRACE));
  #endif
}
/**************************************************************************
 *  show_buffer                                                           *
 *    displays a memory buffer on the screen                              *
 **************************************************************************/

void show_buffer(byte *buffer)
{
  #ifdef VERTICAL_RETRACE
    while ((inp(INPUT_STATUS_1) & VRETRACE));
    while (!(inp(INPUT_STATUS_1) & VRETRACE));
  #endif
  memcpy(VGA,buffer,SCREEN_SIZE);
}
/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory.                                    *
 **************************************************************************/

void load_bmp(char *file,BITMAP *b)
{
  FILE *fp;
  long index;
  word num_colors;
  int x;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }

  /* read in the width and height of the image, and the
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;

  /* try to allocate memory */
  if ((b->data = (byte *) malloc((word)(b->width*b->height))) == NULL)
  {
    fclose(fp);
    printf("Error allocating memory for file %s.\n",file);
    exit(1);
  }

  /* read the palette information */
  for(index=0;index<num_colors;index++)
  {
    b->palette[(int)(index*3+2)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+1)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+0)] = fgetc(fp) >> 2;
    x=fgetc(fp);
  }

  /* read the bitmap */
  for(index = (b->height-1)*b->width; index >= 0;index-=b->width)
    for(x = 0; x < b->width; x++)
      b->data[(int)(index+x)]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  set_palette                                                           *
 *    Sets all 256 colors of the palette.                                 *
 **************************************************************************/

void set_palette(byte *palette)
{
  int i;

  outp(PALETTE_INDEX,0);              /* tell the VGA that palette data
                                         is coming. */
  for(i=0;i<256*3;i++)
    outp(PALETTE_DATA,palette[i]);    /* write the data */
}

/**************************************************************************
 *  plot_pixel                                                            *
 *    Plots a pixel in unchained mode                                     *
 **************************************************************************/

void plot_pixel(int x,int y,byte color)
{
  outp(SC_INDEX, MAP_MASK);          /* select plane */
  outp(SC_DATA,  1 << (x&3) );

  VGA[(y<<6)+(y<<4)+(x>>2)]=color;
```

```
}

/**************************************************************************
 *  Main                                                                  *
 **************************************************************************/

void main(int argc, char *argv[])
{
  word bitmap_offset,screen_offset;
  word visual_page = 0;
  word active_page = SCREEN_SIZE/4;
  word start;
  float t1,t2;
  int i,repeat,plane,num_objects=0;
  word x,y;
  byte *double_buffer;
  BITMAP bmp;
  OBJECT *object;

  /* get command-line options */
  if (argc>0) num_objects=atoi(argv[1]);
  if (num_objects<=0) num_objects=8;

  /* allocate memory for double buffer and background image */
  if ((double_buffer = (byte *) malloc(SCREEN_SIZE)) == NULL)
  {
    printf("Not enough memory for double buffer.\n");
    exit(1);
  }
  /* allocate memory for objects */
  if ((object = (OBJECT *) malloc(sizeof(OBJECT)*num_objects)) == NULL)
  {
    printf("Not enough memory for objects.\n");
    free(double_buffer);
    exit(1);
  }

  /* load the images */
  load_bmp("balls.bmp",&bmp);

  /* set the object positions */
  srand(*my_clock);
  for(i=0;i<num_objects;i++)
  {
    object[i].width   = BITMAP_WIDTH;
    object[i].height  = BITMAP_HEIGHT;
    object[i].x       = rand() % (SCREEN_WIDTH - BITMAP_WIDTH );
    object[i].y       = rand() % (SCREEN_HEIGHT- BITMAP_HEIGHT);
    object[i].dx      = (rand()%5) - 2;
    object[i].dy      = (rand()%5) - 2;
  }

  set_mode(VGA_256_COLOR_MODE);       /* set the video mode. */
  set_palette(bmp.palette);

  start=*my_clock;                    /* record the starting time. */
  for(repeat=0;repeat<TOTAL_FRAMES;repeat++)
```

```
{
  if ((repeat%ANIMATION_FRAMES)==0) bitmap_offset=0;
  /* clear background */
  memset(double_buffer,0,SCREEN_SIZE);

  for(i=0;i<num_objects;i++)
  {
    screen_offset = (object[i].y<<8) + (object[i].y<<6) + object[i].x;
    /* draw the object. */
    for(y=0;y<BITMAP_HEIGHT*bmp.width;y+=bmp.width)
      for(x=0;x<BITMAP_WIDTH;x++)
        if (bmp.data[bitmap_offset+y+x]!=0)
          double_buffer[screen_offset+y+x]=bmp.data[bitmap_offset+y+x];
    /* check to see if the object is within boundries */
    if (object[i].x + object[i].dx < 0 ||
        object[i].x + object[i].dx > SCREEN_WIDTH-object[i].width-1)
          object[i].dx=-object[i].dx;
    if (object[i].y + object[i].dy < 0 ||
        object[i].y + object[i].dy > SCREEN_HEIGHT-object[i].height-1)
          object[i].dy=-object[i].dy;
    /* move the object */
    object[i].x+=object[i].dx;
    object[i].y+=object[i].dy;
  }

  /* point to the next image in the animation */
  bitmap_offset+=BITMAP_WIDTH;
  if ((bitmap_offset%bmp.width)==0)
    bitmap_offset+=bmp.width*(BITMAP_HEIGHT-1);

  /* show the buffer */
  show_buffer(double_buffer);
}
t1=(*my_clock-start)/18.2;          /* calculate how long it took. */

free(double_buffer);                /* free up memory used */

/************************************************************************/

set_unchained_mode();               /* set unchained mode */

start=*my_clock;                    /* record the starting time. */
for(repeat=0;repeat<TOTAL_FRAMES;repeat++)
{
  if ((repeat%ANIMATION_FRAMES)==0) bitmap_offset=0;
  /* clear background */
  outpw(SC_INDEX,ALL_PLANES);
  memset(&VGA[active_page],0,SCREEN_SIZE/4);

  outp(SC_INDEX, MAP_MASK);          /* select plane */
  for(i=0;i<num_objects;i++)
  {
    screen_offset = (object[i].y<<6) + (object[i].y<<4) + (object[i].x>>2);
    /* draw the object. */
    for(plane=0;plane<4;plane++)
    {
      /* select plane */
```

```
        outp(SC_DATA,  1 << ((plane+object[i].x)&3) );
        for(y=0;y<BITMAP_HEIGHT*bmp.width;y+=bmp.width)
          for(x=plane;x<BITMAP_WIDTH;x+=4)
            if (bmp.data[bitmap_offset+y+x]!=0)
              VGA[active_page+screen_offset+(y>>2)+((x+(object[i].x&3)) >> 2)]=
                bmp.data[bitmap_offset+y+x];
      }
      /* check to see if the object is within boundries */
      if (object[i].x + object[i].dx < 0 ||
          object[i].x + object[i].dx > SCREEN_WIDTH-object[i].width-1)
            object[i].dx=-object[i].dx;
      if (object[i].y + object[i].dy < 0 ||
          object[i].y + object[i].dy > SCREEN_HEIGHT-object[i].height-1)
            object[i].dy=-object[i].dy;
      /* move the object */
      object[i].x+=object[i].dx;
      object[i].y+=object[i].dy;
    }

    /* point to the next image in the animation */
    bitmap_offset+=BITMAP_WIDTH;
    if ((bitmap_offset%bmp.width)==0)
      bitmap_offset+=bmp.width*(BITMAP_HEIGHT-1);

    /* flip the pages */
    page_flip(&visual_page,&active_page);
  }
  t2=(*my_clock-start)/18.2;          /* calculate how long it took. */

  free(bmp.data);
  free(object);

  set_mode(TEXT_MODE);                /* set the video mode back to
                                         text mode. */
  /* output the results... */

  printf("Results with %i objects",num_objects);
  #ifdef VERTICAL_RETRACE
    printf(":\n");
  #else
    printf(" (vertical retrace *ignored*):\n");
  #endif
  printf("  Mode 0x13 with double buffering:\n");
  printf("    %f seconds,\n",t1);
  printf("    %f frames per second.\n",(float)TOTAL_FRAMES/t1);
  printf("  Unchained mode with page flipping:\n");
  printf("    %f seconds,\n",t2);
  printf("    %f frames per second.\n",(float)TOTAL_FRAMES/t2);
  if (t2 != 0)
    printf("  Unchained mode with page flipping was %f times faster.\n",t1/t2);

  return;
}
```

« Back to Double Buffering, Page Flipping, & Unchained Mode

DAVID BRACKEEN

256-Color VGA Programming in C > Double Buffering, Page Flipping, & Unchained Mode

modes.c

View as plain text

```
/**************************************************************************
 * modes.c                                                                *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * Tab stops are set to 2.                                                *
 * This program compiles with DJGPP! (www.delorie.com)                    *
 * To compile in DJGPP: gcc modes.c -o modes.exe                          *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA, or compatible video adapter.                            *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demonstrates various unchained modes         *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <sys/nearptr.h>
#include <conio.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */


#define MISC_OUTPUT         0x03c2    /* VGA misc. output register */
#define SC_INDEX            0x03c4    /* VGA sequence controller */
#define SC_DATA             0x03c5
#define PALETTE_INDEX       0x03c8    /* VGA digital-to-analog converter */
#define PALETTE_DATA        0x03c9
#define CRTC_INDEX          0x03d4    /* VGA CRT controller */

#define MAP_MASK            0x02      /* Sequence controller registers */
#define MEMORY_MODE         0x04

#define H_TOTAL             0x00      /* CRT controller registers */
#define H_DISPLAY_END       0x01
#define H_BLANK_START       0x02
#define H_BLANK_END         0x03
#define H_RETRACE_START     0x04
```

```
#define H_RETRACE_END       0x05
#define V_TOTAL             0x06
#define OVERFLOW            0x07
#define MAX_SCAN_LINE       0x09
#define V_RETRACE_START     0x10
#define V_RETRACE_END       0x11
#define V_DISPLAY_END       0x12
#define OFFSET              0x13
#define UNDERLINE_LOCATION  0x14
#define V_BLANK_START       0x15
#define V_BLANK_END         0x16
#define MODE_CONTROL        0x17

#define NUM_COLORS          256       /* number of colors in mode 0x13 */

/* macro to return the sign of a number */
#define sgn(x) \
  ((x<0)?-1:((x>0)?1:0))

/* macro to write a word to a port */
#define word_out(port,register,value) \
  outpw(port,(((word)value<<8) + register))

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;

/* the structure for a planar bitmap. */
typedef struct tagPLANAR_BITMAP
{
  word width;
  word height;
  byte palette[256*3];
  byte *data[4];
} PLANAR_BITMAP;

byte *VGA = (byte *)0xA0000;          /* this points to video memory. */
word *my_clock = (word *)0x046C;      /* this points to the 18.2hz system
                                         clock. */
word screen_width, screen_height;

/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/
```

```
void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  set_unchained_mode                                                    *
 *    Resets VGA mode 0x13 to unchained mode to access all 256K of        *
 *    memory.  width may be 320 or 360, height may be 200, 400, 240 or    *
 *    480.  If an invalid width or height is specified, it defaults to    *
 *    320x200                                                             *
 **************************************************************************/

void set_unchained_mode(int width, int height)
{
  word i;
  dword *ptr=(dword *)VGA;

  /* set mode 13 */
  set_mode(VGA_256_COLOR_MODE);

  /* turn off chain-4 mode */
  word_out(SC_INDEX, MEMORY_MODE,0x06);

  /* set map mask to all 4 planes for screen clearing */
  word_out(SC_INDEX, MAP_MASK, 0xff);

  /* clear all 256K of memory */
  for(i=0;i<0x4000;i++)
    *ptr++ = 0;

  /* turn off long mode */
  word_out(CRTC_INDEX, UNDERLINE_LOCATION, 0x00);

  /* turn on byte mode */
  word_out(CRTC_INDEX, MODE_CONTROL, 0xe3);


  screen_width=320;
  screen_height=200;

  if (width==360)
  {
    /* turn off write protect */
    word_out(CRTC_INDEX, V_RETRACE_END, 0x2c);

    outp(MISC_OUTPUT, 0xe7);
    word_out(CRTC_INDEX, H_TOTAL, 0x6b);
    word_out(CRTC_INDEX, H_DISPLAY_END, 0x59);
    word_out(CRTC_INDEX, H_BLANK_START, 0x5a);
    word_out(CRTC_INDEX, H_BLANK_END, 0x8e);
    word_out(CRTC_INDEX, H_RETRACE_START, 0x5e);
    word_out(CRTC_INDEX, H_RETRACE_END, 0x8a);
    word_out(CRTC_INDEX, OFFSET, 0x2d);

    /* set vertical retrace back to normal */
    word_out(CRTC_INDEX, V_RETRACE_END, 0x8e);

    screen_width=360;
  }
  else
  {
    outp(MISC_OUTPUT, 0xe3);
  }

  if (height==240 || height==480)
  {
    /* turn off write protect */
    word_out(CRTC_INDEX, V_RETRACE_END, 0x2c);

    word_out(CRTC_INDEX, V_TOTAL, 0x0d);
    word_out(CRTC_INDEX, OVERFLOW, 0x3e);
    word_out(CRTC_INDEX, V_RETRACE_START, 0xea);
    word_out(CRTC_INDEX, V_RETRACE_END, 0xac);
    word_out(CRTC_INDEX, V_DISPLAY_END, 0xdf);
    word_out(CRTC_INDEX, V_BLANK_START, 0xe7);
    word_out(CRTC_INDEX, V_BLANK_END, 0x06);
    screen_height=height;
  }

  if (height==400 || height==480)
  {
    word_out(CRTC_INDEX, MAX_SCAN_LINE, 0x40);
    screen_height=height;
  }



}

/**************************************************************************
 *  draw_bitmap                                                           *
 *    Draws a bitmap. Bitmaps are stored in a four-plane format to ease   *
 *    the drawing process (the plane is changed only four times)          *
 **************************************************************************/

void draw_bitmap(PLANAR_BITMAP *bmp,int x,int y)
{
  int j,plane;
  word screen_offset;
  word bitmap_offset;

  for(plane=0; plane<4; plane++)
  {
    outp(SC_INDEX, MAP_MASK);          /* select plane */
    outp(SC_DATA,  1 << ((plane+x)&3) );
    bitmap_offset=0;
    screen_offset = ((dword)y*screen_width+x+plane) >> 2;
    for(j=0; j<bmp->height; j++)
    {
      memcpy(&VGA[screen_offset], &bmp->data[plane][bitmap_offset], (bmp->width >> 2));

      bitmap_offset+=bmp->width>>2;
      screen_offset+=screen_width>>2;
    }
  }
}

/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory. Only works for bitmaps whose width *
 *    is divible by 4!                                                    *
 **************************************************************************/

void load_bmp(char *file,PLANAR_BITMAP *b)
{
  FILE *fp;
  long index, size;
  word num_colors;
  int x, plane;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }

  /* read in the width and height of the image, and the
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;

  size=b->width*b->height;

  /* try to allocate memory */
  for(plane=0;plane<4;plane++)
  {
    if ((b->data[plane] = (byte *) malloc((word)(size>>2))) == NULL)
    {
      fclose(fp);
      printf("Error allocating memory for file %s.\n",file);
      exit(1);
    }
  }

  /* read the palette information */
  for(index=0;index<num_colors;index++)
  {
    b->palette[(int)(index*3+2)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+1)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+0)] = fgetc(fp) >> 2;
    x=fgetc(fp);
  }

  /* read the bitmap */
  for(index = (b->height-1)*b->width; index >= 0;index-=b->width)
    for(x = 0; x < b->width; x++)
      b->data[x&3][(int)((index+x)>>2)]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  set_palette                                                           *
 *    Sets all 256 colors of the palette.                                 *
 **************************************************************************/

void set_palette(byte *palette)
{
  int i;

  outp(PALETTE_INDEX,0);              /* tell the VGA that palette data
                                         is coming. */
  for(i=0;i<256*3;i++)
    outp(PALETTE_DATA,palette[i]);    /* write the data */
}

/**************************************************************************
 *  plot_pixel                                                            *
 *    Plots a pixel in unchained mode                                     *
 **************************************************************************/

void plot_pixel(int x,int y,byte color)
{
  dword offset;

  outp(SC_INDEX, MAP_MASK);          /* select plane */
  outp(SC_DATA,  1 << (x&3) );

  offset=((dword)y*screen_width+x) >> 2;

  VGA[(word)offset]=color;
}

/**************************************************************************
 *  Main                                                                  *
 **************************************************************************/

void main(void)
{
  int x,y,plane,choice=1;
  int x_size[2]={320,360};
  int y_size[4]={200,240,400,480};
  PLANAR_BITMAP bmp;

  if (__djgpp_nearptr_enable() == 0)
  {
    printf("Could get access to first 640K of memory.\n");
    exit(-1);
  }

  VGA+=__djgpp_conventional_base;
  my_clock = (void *)my_clock + __djgpp_conventional_base;

  /* load the images */
  load_bmp("ghosts.bmp",&bmp);

  while (choice!=8)
  {
    /* present menu */
    printf("Select a mode to test\n\n");
    printf("0. 320x200\n");
    printf("1. 320x240\n");
    printf("2. 320x400\n");
    printf("3. 320x480\n");
    printf("4. 360x200\n");
    printf("5. 360x240\n");
    printf("6. 360x400\n");
    printf("7. 360x480\n");
    printf("8. Quit\n\n");
    printf("Your choice? ");

    /* get input */
    scanf("%i",&choice);
    fflush(stdin);

    if (choice!=8)
    {
      set_unchained_mode(x_size[(choice&4)>>2],y_size[choice&3]);
      set_palette(bmp.palette);

      /* tile the images */
      for(x=0;x<=screen_width-bmp.width;x+=bmp.width)
        for(y=0;y<=screen_height-bmp.height;y+=bmp.height)
          draw_bitmap(&bmp,x,y);

      /* use getchar(); here if your compiler doesn’t have getch(); */
      getch();

      set_mode(TEXT_MODE);
    }
  }
  for(plane=0;plane<4;plane++)
    free(bmp.data[plane]);

  __djgpp_nearptr_disable();

  return;
}
```

« Back to Double Buffering, Page Flipping, & Unchained Mode

DAVID BRACKEEN

modes.c

View as plain text

```
/**************************************************************************
 * modes.c                                                                *
 * written by David Brackeen                                              *
 * http://www.brackeen.com/home/vga/                                      *
 *                                                                        *
 * This is a 16-bit program.                                              *
 * Tab stops are set to 2.                                                *
 * Remember to compile in the LARGE memory model!                         *
 * To compile in Borland C: bcc -ml modes.c                               *
 *                                                                        *
 * This program will only work on DOS- or Windows-based systems with a    *
 * VGA, SuperVGA or compatible video adapter.                             *
 *                                                                        *
 * Please feel free to copy this source code.                             *
 *                                                                        *
 * DESCRIPTION: This program demonstrates various unchained modes         *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <mem.h>
#include <conio.h>

#define VIDEO_INT           0x10      /* the BIOS video interrupt. */
#define SET_MODE            0x00      /* BIOS func to set the video mode. */
#define VGA_256_COLOR_MODE  0x13      /* use to set 256-color mode. */
#define TEXT_MODE           0x03      /* use to set 80x25 text mode. */


#define MISC_OUTPUT         0x03c2    /* VGA misc. output register */
#define SC_INDEX            0x03c4    /* VGA sequence controller */
#define SC_DATA             0x03c5
#define PALETTE_INDEX       0x03c8    /* VGA digital-to-analog converter */
#define PALETTE_DATA        0x03c9
#define CRTC_INDEX          0x03d4    /* VGA CRT controller */

#define MAP_MASK            0x02      /* Sequence controller registers */
#define MEMORY_MODE         0x04

#define H_TOTAL             0x00      /* CRT controller registers */
#define H_DISPLAY_END       0x01
#define H_BLANK_START       0x02
#define H_BLANK_END         0x03
```

```
#define H_RETRACE_START     0x04
#define H_RETRACE_END       0x05
#define V_TOTAL             0x06
#define OVERFLOW            0x07
#define MAX_SCAN_LINE       0x09
#define V_RETRACE_START     0x10
#define V_RETRACE_END       0x11
#define V_DISPLAY_END       0x12
#define OFFSET              0x13
#define UNDERLINE_LOCATION  0x14
#define V_BLANK_START       0x15
#define V_BLANK_END         0x16
#define MODE_CONTROL        0x17

#define NUM_COLORS          256       /* number of colors in mode 0x13 */

/* macro to return the sign of a number */
#define sgn(x) \
  ((x<0)?-1:((x>0)?1:0))

/* macro to write a word to a port */
#define word_out(port,register,value) \
  outpw(port,(((word)value<<8) + register))

typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned long  dword;

/* the structure for a planar bitmap. */
typedef struct tagPLANAR_BITMAP
{
  word width;
  word height;
  byte palette[256*3];
  byte *data[4];
} PLANAR_BITMAP;

byte *VGA=(byte *)0xA0000000L;        /* this points to video memory. */
word screen_width, screen_height;

/**************************************************************************
 *  fskip                                                                 *
 *     Skips bytes in a file.                                             *
 **************************************************************************/

void fskip(FILE *fp, int num_bytes)
{
   int i;
   for (i=0; i<num_bytes; i++)
      fgetc(fp);
}

/**************************************************************************
 *  set_mode                                                              *
 *     Sets the video mode.                                               *
 **************************************************************************/
```

```
void set_mode(byte mode)
{
  union REGS regs;

  regs.h.ah = SET_MODE;
  regs.h.al = mode;
  int86(VIDEO_INT, &regs, &regs);
}

/**************************************************************************
 *  set_unchained_mode                                                    *
 *    Resets VGA mode 0x13 to unchained mode to access all 256K of        *
 *    memory.  width may be 320 or 360, height may be 200, 400, 240 or    *
 *    480.  If an invalid width or height is specified, it defaults to    *
 *    320x200                                                             *
 **************************************************************************/

void set_unchained_mode(int width, int height)
{
  word i;
  dword *ptr=(dword *)VGA;

  /* set mode 13 */
  set_mode(VGA_256_COLOR_MODE);

  /* turn off chain-4 mode */
  word_out(SC_INDEX, MEMORY_MODE,0x06);

  /* set map mask to all 4 planes for screen clearing */
  word_out(SC_INDEX, MAP_MASK, 0xff);

  /* clear all 256K of memory */
  for(i=0;i<0x4000;i++)
    *ptr++ = 0;

  /* turn off long mode */
  word_out(CRTC_INDEX, UNDERLINE_LOCATION, 0x00);

  /* turn on byte mode */
  word_out(CRTC_INDEX, MODE_CONTROL, 0xe3);


  screen_width=320;
  screen_height=200;

  if (width==360)
  {
    /* turn off write protect */
    word_out(CRTC_INDEX, V_RETRACE_END, 0x2c);

    outp(MISC_OUTPUT, 0xe7);
    word_out(CRTC_INDEX, H_TOTAL, 0x6b);
    word_out(CRTC_INDEX, H_DISPLAY_END, 0x59);
    word_out(CRTC_INDEX, H_BLANK_START, 0x5a);
    word_out(CRTC_INDEX, H_BLANK_END, 0x8e);
    word_out(CRTC_INDEX, H_RETRACE_START, 0x5e);
    word_out(CRTC_INDEX, H_RETRACE_END, 0x8a);
    word_out(CRTC_INDEX, OFFSET, 0x2d);

    /* set vertical retrace back to normal */
    word_out(CRTC_INDEX, V_RETRACE_END, 0x8e);

    screen_width=360;
  }
  else
  {
    outp(MISC_OUTPUT, 0xe3);
  }

  if (height==240 || height==480)
  {
    /* turn off write protect */
    word_out(CRTC_INDEX, V_RETRACE_END, 0x2c);

    word_out(CRTC_INDEX, V_TOTAL, 0x0d);
    word_out(CRTC_INDEX, OVERFLOW, 0x3e);
    word_out(CRTC_INDEX, V_RETRACE_START, 0xea);
    word_out(CRTC_INDEX, V_RETRACE_END, 0xac);
    word_out(CRTC_INDEX, V_DISPLAY_END, 0xdf);
    word_out(CRTC_INDEX, V_BLANK_START, 0xe7);
    word_out(CRTC_INDEX, V_BLANK_END, 0x06);
    screen_height=height;
  }

  if (height==400 || height==480)
  {
    word_out(CRTC_INDEX, MAX_SCAN_LINE, 0x40);
    screen_height=height;
  }



}

/**************************************************************************
 *  draw_bitmap                                                           *
 *    Draws a bitmap. Bitmaps are stored in a four-plane format to ease   *
 *    the drawing process (the plane is changed only four times)          *
 **************************************************************************/

void draw_bitmap(PLANAR_BITMAP *bmp,int x,int y)
{
  int j,plane;
  word screen_offset;
  word bitmap_offset;

  for(plane=0; plane<4; plane++)
  {
    outp(SC_INDEX, MAP_MASK);          /* select plane */
    outp(SC_DATA,  1 << ((plane+x)&3) );
    bitmap_offset=0;
    screen_offset = ((dword)y*screen_width+x+plane) >> 2;
    for(j=0; j<bmp->height; j++)
    {
      memcpy(&VGA[screen_offset], &bmp->data[plane][bitmap_offset], (bmp->width >> 2));

      bitmap_offset+=bmp->width>>2;
      screen_offset+=screen_width>>2;
    }
  }
}

/**************************************************************************
 *  load_bmp                                                              *
 *    Loads a bitmap file into memory. Only works for bitmaps whose width *
 *    is divible by 4!                                                    *
 **************************************************************************/

void load_bmp(char *file,PLANAR_BITMAP *b)
{
  FILE *fp;
  long index, size;
  word num_colors;
  int x, plane;

  /* open the file */
  if ((fp = fopen(file,"rb")) == NULL)
  {
    printf("Error opening file %s.\n",file);
    exit(1);
  }

  /* check to see if it is a valid bitmap file */
  if (fgetc(fp)!='B' || fgetc(fp)!='M')
  {
    fclose(fp);
    printf("%s is not a bitmap file.\n",file);
    exit(1);
  }

  /* read in the width and height of the image, and the
     number of colors used; ignore the rest */
  fskip(fp,16);
  fread(&b->width, sizeof(word), 1, fp);
  fskip(fp,2);
  fread(&b->height,sizeof(word), 1, fp);
  fskip(fp,22);
  fread(&num_colors,sizeof(word), 1, fp);
  fskip(fp,6);

  /* assume we are working with an 8-bit file */
  if (num_colors==0) num_colors=256;

  size=b->width*b->height;

  /* try to allocate memory */
  for(plane=0;plane<4;plane++)
  {
    if ((b->data[plane] = (byte *) malloc((word)(size>>2))) == NULL)
    {
      fclose(fp);
      printf("Error allocating memory for file %s.\n",file);
      exit(1);
    }
  }

  /* read the palette information */
  for(index=0;index<num_colors;index++)
  {
    b->palette[(int)(index*3+2)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+1)] = fgetc(fp) >> 2;
    b->palette[(int)(index*3+0)] = fgetc(fp) >> 2;
    x=fgetc(fp);
  }

  /* read the bitmap */
  for(index = (b->height-1)*b->width; index >= 0;index-=b->width)
    for(x = 0; x < b->width; x++)
      b->data[x&3][(int)((index+x)>>2)]=(byte)fgetc(fp);

  fclose(fp);
}

/**************************************************************************
 *  set_palette                                                           *
 *    Sets all 256 colors of the palette.                                 *
 **************************************************************************/

void set_palette(byte *palette)
{
  int i;

  outp(PALETTE_INDEX,0);              /* tell the VGA that palette data
                                         is coming. */
  for(i=0;i<256*3;i++)
    outp(PALETTE_DATA,palette[i]);    /* write the data */
}

/**************************************************************************
 *  plot_pixel                                                            *
 *    Plots a pixel in unchained mode                                     *
 **************************************************************************/

void plot_pixel(int x,int y,byte color)
{
  dword offset;

  outp(SC_INDEX, MAP_MASK);          /* select plane */
  outp(SC_DATA,  1 << (x&3) );

  offset=((dword)y*screen_width+x) >> 2;

  VGA[(word)offset]=color;
}

/**************************************************************************
 *  Main                                                                  *
 **************************************************************************/
```

```
void main(void)
{
  int x,y,plane,choice=1;
  int x_size[2]={320,360};
  int y_size[4]={200,240,400,480};
  PLANAR_BITMAP bmp;

  /* load the images */
  load_bmp("ghosts.bmp",&bmp);

  while (choice!=8)
  {
    /* present menu */
    printf("Select a mode to test\n\n");
    printf("0. 320x200\n");
    printf("1. 320x240\n");
    printf("2. 320x400\n");
    printf("3. 320x480\n");
    printf("4. 360x200\n");
    printf("5. 360x240\n");
    printf("6. 360x400\n");
    printf("7. 360x480\n");
    printf("8. Quit\n\n");
    printf("Your choice? ");

    /* get input */
    scanf("%i",&choice);
    fflush(stdin);

    if (choice!=8)
    {
      set_unchained_mode(x_size[(choice&4)>>2],y_size[choice&3]);
      set_palette(bmp.palette);

      /* tile the images */
      for(x=0;x<=screen_width-bmp.width;x+=bmp.width)
        for(y=0;y<=screen_height-bmp.height;y+=bmp.height)
          draw_bitmap(&bmp,x,y);

      /* use getchar(); here if your compiler doesn’t have getch(); */
      getch();

      set_mode(TEXT_MODE);
    }
  }

  for(plane=0;plane<4;plane++)
    free(bmp.data[plane]);


  return;
}
```

« Back to Double Buffering, Page Flipping, & Unchained Mode

Links and Other Resources

Contents in this section:

DOS C Compilers

- DJGPP. A free 32-bit DOS compiler. Recommended. See also Brennan’s DJGPP + Game Resources.
- Borland Turbo C 2.01. A free download.
- Open Watcom C/C++. See also Paul Hsieh’s Home Page.

DOS Emulators

Can’t run DOS programs? Check out these DOS emulators.

- DOSBox. Emulator for Windows, Mac, Linux, and more. Recommended.
- DOSEMU. Emulator for Linux.

Links

- PC-GPE online. DOS game programming reference.
- comp.os.msdos.programmer FAQ. General DOS programming reference.
- Ralf Brown’s Interrupt List. A massive guide to DOS interrupts — A must-see for DOS programmers.
- Steel’s Programming Resource Page. Tutorials on 3D, VESA SVGA, and fractals. Lots of links, too.
- Brennan’s DJGPP + Game Resources. Great resource for game programming using DJGPP.
- Paul Hsieh’s Home Page. Information on programming, graphics, games, optimizations, Watcom C/C++, etc.
- Programmers’ Lair. Tutorials on graphics, sound and artificial intelligence. Also includes game source, links, and PDFs.
- Yahoo - Game Programming. Links to many DOS-related sites.
- x2ftp. Huge DOS game programming resource. Note, this link is a mirror — the original archive appears to be no longer be available.

DAVID BRACKEEN

256-Color VGA Programming in C

Download

Contents in this section:

Download source code and executables

Download all source code, makefiles, executables, and bitmaps:

DJGPP 2.0
    djgpp-all.zip
Borland C, Turbo C, etc.
    bc-all.zip

Or download only the programs you need:

DJGPP 2.0

pixel.zip	Pixel plotting and writing directly to video memory.
lines.zip	Line drawing using Bresenham’s algorithm.
rect.zip	Rectangle drawing.
circle.zip	Circle drawing using tables.
bitmap.zip	Bitmap drawing and transparency.
palette.zip	Modifying and rotating the color palette.
mouse.zip	Mouse support, animation, and buttons.
unchain.zip	Unchained mode (mode-x), double buffering, and page flipping.
modes.zip	Various unchained modes, from 320x200 to 360x480.

Borland C, Turbo C, etc.

pixel.zip	Pixel plotting and writing directly to video memory.
lines.zip	Line drawing using Bresenham’s algorithm.
rect.zip	Rectangle drawing.
circle.zip	Circle drawing using tables.
bitmap.zip	Bitmap drawing and transparency.
palette.zip	Modifying and rotating the color palette.
mouse.zip	Mouse support, animation, and buttons.
unchain.zip	Unchained mode (mode-x), double buffering, and page flipping.
modes.zip	Various unchained modes, from 320x200 to 360x480.

Download the entire tutorial

This zip file contains the entire tutorial, including source and binaries for both DJGPP 2.0 and Borland C.

Entire tutorial
vga-all.zip (1.4 MB)

About uploading

You can upload the tutorial, in whole or in part, to your site or any other site. No problem!

DAVID BRACKEEN

256-Color VGA Programming in C

FAQ and Troubleshooting

Contents in this section:

Introduction

Here are some tidbits about a few compilers and some solutions to common VGA programming problems. Some of this information is from my experiences, other information is from people e-mailing me about their experiences. If anyone has any information about getting the code at this site to work with a particular compiler, let me know.

How do I compile the source code using DJGPP 2.0?

Well, the good news is that DJGPP source is now included on the site. In case you are wondering, here are the changes.

DJGPP creates protected mode programs, so you can’t access the video segment directly without some trickery. You can use the `__djgpp_nearptr_enable()` function to get access to the first 640K of memory. So, in `main()` add these lines at the beggining:

```
if (__djgpp_nearptr_enable() == 0)
  exit(-1);

VGA=(byte *)(0xa0000 + __djgpp_conventional_base);
my_clock=(word *)(0x0046c + __djgpp_conventional_base);
```

At the end of the program, use `__djgpp_nearptr_disable();` to get the program back to regular protected mode. Also, don’t forget to #include `<sys/nearptr.h>`.

This should get all of the programs to work except `mouse.c`.

How do I compile the source code using Watcom C/C++ 10.6?

Like DJGPP, Watcom C/C++ creates protected-mode programs, but the first megabyte or memory can be access directly by your program without any changes. Memory is linear instead of based on segments and offsets. So, change this

```
byte *VGA=(byte*)0xA0000000L;
```

to this:

```
byte *VGA=(byte*)0xA0000L;
```

Also, chage all `int86();` functions to `int386();` and make sure you #include `<i86.h>`.

In the `plot_pixel_slow();` function in `pixel.c`, change this

```
regs.x.cx = x;
regs.x.dx = y;
```

to this:

```
regs.w.cx = x;
regs.w.dx = y;
```

That should do it! All the programs should work except `mouse.c`.

How do I compile the source code using Turbo C 2.01?

Compiling the code at this site with Turbo C 2.01 (a free compiler from Borland) is fairly straightforward. In modes.c and unchain.c, change every instance of **outpw** to **outport**. Be sure to compile in the large memory model, like so:

```
tcc -ml -N filename.c
```

How do I compile the source code using an older version of Turbo C or Borland C?

Some people have told me the Turbo C compiler does not accept direct pointers. For those of you with Turbo C, every section of code that looks like this:

```
byte *VGA=(byte *)0xA0000000L;
word *my_clock=(word *)0x0000046C;
```

Should be changed to:

```
byte *VGA=(byte *)MK_FP(0xA000,0);
word *my_clock=(word *)MK_FP(0,0x046C);
```

Also, make sure to include `<dos.h>`.

How do I compile the source code using Quick C?

For users of Quick C, change `#include <mem.h>` to `#include <memory.h>`

How do I compile the source code using GCC?

You can’t. The code is DOS-mode only, so you’ll need a DOS compiler. GCC on Linux or other operating systems won’t work, and the Windows port of GCC (Cygwin) won’t work either. Try DJGPP, which is a DOS port of GCC.

How do I compile the source code using Visual C++?

You’ll need Visual C++ 1.42, which isn’t available anymore. Visual C++ 2.0 and above are for Windows only.

The code compiles fine, but I run the program the screen goes black and/or Windows locks up.

For Turbo C, Borland C, and a few others, the programs must be compiled in the LARGE memory model. Otherwise, the programs will crash when they try to access the video memory area.

When I compile, I get the error “Code has no effect in function” and it points to the outp() function.

I’ve heard about this happening in Turbo C 3.0, and Borland C 4.5. At the top of the program, insert the code `#undef outp`.

When I run the program, I get the error “Not enough memory for double buffer”.

I thinks this only happens in older versions of Borland C/Turbo C. If you are running the program from the IDE, there is not enough *conventional* memory to run both the IDE and the program at the same time. After you compile, either exit Borland C to run the program or go to a dos shell to run the program.

Some people wrote in and said this: Change the debugger memory. Go to the options menu, then debugger. Change the program heap size to around 350K or so.

I have Borland C++ 5.0 for Windows 95 and I can’t get the programs to compile. It says undefined structure ’REGS’.

Make sure you compile the programs for DOS, not Windows.

The programs only seem to work with the bitmaps that come with each program (i.e. rocket.bmp and mset.bmp).

Here are four reasons your bitmaps may not be working:

1. Make sure the bitmap’s width is divisible by 4 (for example, a width of 24 is okay but a width of 25 is not)
2. In your paint program, do not save your bitmaps as compressed or RLE bitmaps. The code can only handle uncompressed format BMP files.
3. In your paint program, make sure you save in 256-color (8 bit) format.
4. Use the code from palette.c (rather than bitmap.c) if your bitmap uses a non-standard palette.

If you’ve done these four things and you are still having a problem, email me the bitmap and I will try to figure it out.

How do you display text in mode 0x13?

You can use the BIOS fonts, but it’s best to write your own font routines. You can easily create a bitmap for the ASCII characters in a paint program, and then write the code to read the bitmaps and display a string. Plus, bitmap fonts look a lot better than the BIOS fonts.

What is a good graphics program?

Paint Shop Pro is a great graphics program for Windows, and I use it all the time. Download the shareware version at www.corel.com. You could use Microsoft Paint that comes with Windows, but you’ll be better off with a more powerful graphics program.

Could you please send me information on 640x480 or 800x600 or [enter your favorite video mode here]?

Sorry, I really don’t have any VESA SVGA code to give out. Check out Steel’s page and look at the “VESA” programming information.

ARGH! Help me! I’ve got an assignment due tomorrow, could you please modify your code to my assignment’s specifications and send it to me?

This always makes me laugh. The funny thing is, this question gets asked quite often. Short answer: No. Long answer: Do your own work.
