Programming VGA Mode 13h, part 1

Phil Inch, Game Developers Magazine

One of the more common VGA modes used for games is mode 13h ('h' stands for hexadecimal, so this mode is actually mode 19). This article will teach you the basics for directly manipulating this video mode, which is the fastest method for using it in terms of raw display speed.

Your programming language probably already offers you some graphics commands, and possibly these work with mode 13h. So why, you ask, would I want to manipulate the screen directly?

The answer, in short, is ***speed***. Graphics commands and add-ons with most languages use the computer's in-built functions for doing graphics work, which are sloo-oo-oo-oow. Far too slow for any game that involves a little movement.

"Graphics commands ... in most languages are far too slow for any game that involves movement"

Getting in to mode 13h

The resolution of mode 13h is 320 x 200 x 256. That is, 320 pixels wide by 200 pixels high (a total of 64,000 pixels) with each pixel capable of being one of 256 colours. The first thing we need to know is, how do we tell our video card how to change itself into video mode 13h?

This will vary from language to language. Some languages, such as BASIC, contain a SCREEN statement, and depending on which version of BASIC you're using and how old it is, this command may allow you to set mode 13h directly. You'll have to consult your manual to check this out.

Failing that, in languages such as C, Pascal, and Assembler we can use one the functions available to us in the BIOS. In this case, the function is Interrupt 10h, Function 0h. (If you're not familiar with registers or interrupts, see the sections "Introduction to Registers" and "Introduction to Interrupts") elsewhere in this magazine.

Int 10h Func 00h takes only one parameter - a screen mode number in the AL register. So, to set mode 13h, we need to put the value 13h in the AL register, 0 in the AH register, and call interrupt 10. In assembler, this looks like this:

```
MOV AH, 0
MOV AL, 13h
INT 10h
```

C and Pascal have specialised functions which allow you to execute interrupts, and I believe some dialects of BASIC do too. Consult your manual for further details, and please let me know the results so I can publish the exact details in a later issue.

Getting out of mode 13h

When you've finished working with graphics, remember you'll have to put your screen back into text mode!

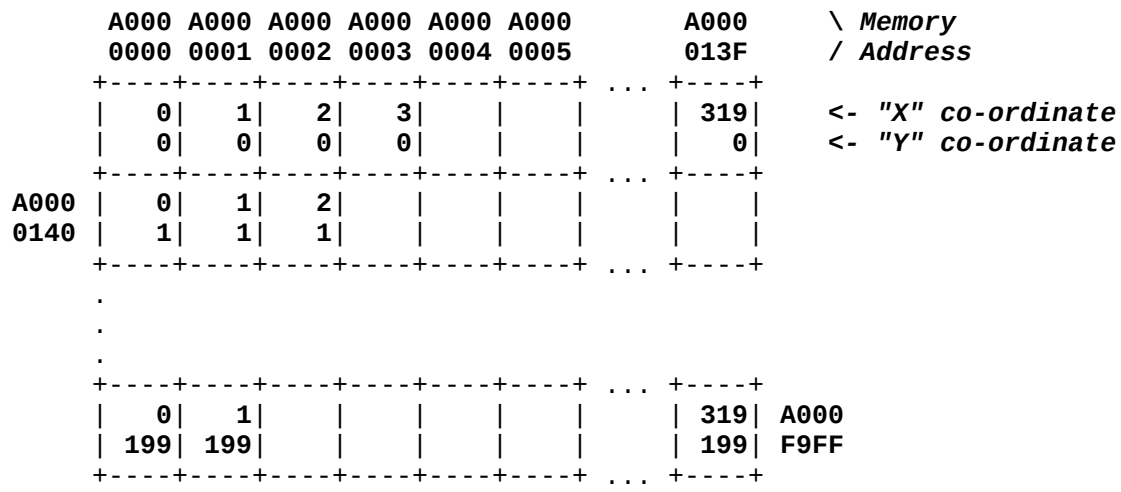
"Normal" 80x25 colour text mode is video mode 3h - see if you can write a routine to put you back into text mode, based on what you've learned above! Answers in part two next month.

How memory is organised in mode 13h

One of the greatest advantages of mode 13h for the programmer is the way the video memory is organised. Each pixel takes one byte of memory, and as we move horizontally across each line, we are moving one byte forward in memory. That is, mode 13h has a ***linear*** memory structure. The first pixel on the mode 13h screen (the one in the top left hand corner) is at memory address A000:0000. This is called the 'screen base address'.

(If you don't understand the way this memory address is written, please read the article 'Understanding Segmented Memory Addresses' elsewhere in the magazine.)

This is illustrated in the following diagram:



Each 'box' represents one pixel. The pixel in the top left-hand corner is considered to have co-ordinates (0,0) and it exists at memory location A000:0000. The next pixel has co-ordinates (1,0) at it exists at memory location A000:0001. Continuing to the end of the line, we see that the pixel at (319,0) lives at memory location A000:013F.

The first pixel on the following line (0,1) lives at A000:0140, which is exactly one byte past the last pixel on the previous line. You should now understand what is meant by linear memory!

This pattern continues right down to the last pixel of the last line (319,199) which lives at A000:F9FF.

"One of the greatest advantages of mode 13h is ... that it has a linear memory structure."

Each of these memory locations contains a number in the range 0 to 255 which corresponds to a VGA colour. To place a pixel of colour 11 at pixel location (38,27), we need to figure out the memory location at this pixel, then put the value 11 into that location.

How can we work out the memory location? Well, we know that each line on the display takes 320 bytes of memory, (because each line is 320 pixels wide) so we can find the right line by multiplying the y co-ordinate by 320 and adding it to the screen base address. Then, to get the offset across the line, we simply add the X co-ordinate.

Therefore, we end up with this formula:

```
MEMORY_ADDRESS = BASE_ADDRESS + ( 320 * Y ) + X
```

Now we can put the value 11 into MEMORY_ADDRESS. In **C**, this could be done using a **far pointer**, eg:

```
void far *memory_address;
void far *base_address;
set_mode13h();
base_address = 0xA0000; /* MUST BE Four zeroes */
memory_address = base_address + ( 320 * Y ) + X;
*(memory_address) = colour;
```

In **BASIC**, you'll need to do a **DEF SEG** followed by a **POKE**. For BASIC, you don't need to add BASE_ADDRESS in the formula, because that's effectively what the DEF SEG (DEFine SEGment) does:

```
DEF SEG = &hA000
MEMORY_ADDRESS = ( 320 * Y ) + X
POKE MEMORY_ADDRESS, colour
```

In **assembler**, you need something like the following. Again, the BASE_ADDRESS is not required:

```
MOV AX,A000h
MOV ES,AX
MOV AX,( 320 * Y ) + X
MOV DI,AX
MOV [ES:DI],colour
```

(Please note: the examples above are intended to be illustrative of the concept, and may or may not compile as shown. Refer to your documentation for exact details. Sorry, I can't include a Pascal example as I don't use it, so if you can translate the above into pascal and contribute it, I would be grateful.)

Multiple pixels = lines!

If you have followed everything above, you should now be able to see that to draw multiple pixels, such as a horizontal or vertical line, just requires you to work out the (X,Y) location for each pixel, calculate the memory address for each pixel, and place the appropriate colour number into that memory location.

As an exercise, try to create your own routines for plotting pixels and two routines for drawing horizontal and vertical lines. These should be two routines, for reasons which should become clear when you create them. Once your routines are able to draw the lines, try to make the routines as fast as possible. There are a number of corners you can cut when drawing purely horizontal and purely vertical lines.

Multiple pixels also = sprites!

(Def: SPRITE: Any small image intended for animation & movement, eg spaceships, bullets, hedgehogs ...)

I'm sure that if you have read this whole article, you probably don't feel that you haven't progressed much. All you have really learned to do is draw dots. Totally unexciting, really. Until you combine those dots to make a SPRITE!

"A sprite is just a collection of dots ... if you can draw dots, you can draw sprites!"

Yes, that's right. Remember, all a sprite is, is a collection of dots. So, if you can draw dots, you can draw sprites! You can also draw anything else, including background pictures!

Programming VGA Mode 13h, part 2

Phil Inch, Game Developers Magazine

DOWNLOAD ... *The example files mentioned in this article are contained in the file **LINEDRAW.ZIP** (55,962 bytes) which can be downloaded by clicking this disk icon.*

Introduction

Welcome back! Last month we looked at how to enter and leave mode 13h, how the memory is organised, and how to plot points of any colour anywhere on the screen.

This month, as promised, we'll draw some lines, and in doing so we'll learn a thing or two about optimising your programs to run as fast as possible!

This article assumes you've read and understood everything in part 1.

Let's draw a horizontal line!

Last article I left you with the challenge to come up with two routines, one for drawing horizontal lines and one for drawing vertical lines. At the time I also pointed out that these routines can be highly optimised because there are certain assumptions you can make when you draw purely horizontal or purely vertical lines. How did you do?

Without further ado, let's get into the first Pseudo-Code of the issue, and look at a simple horizontal line drawing algorith:

```
            Parameters to routine:
                COLOUR: Colour of line
                X1: X co-ordinate of one end of line
                X2: X co-ordinate of the other end of the line
                Y: Y co-ordinate of the line (same at both ends because line is
horizontal!)

            Returned values:
                None

            Assumptions
                1. The screen should already be in mode 13
                2. X2 should be greater that X1.  Check this!
                3. All co-ordinates are legal for mode 13h.
                   Legal co-ordinates range from (0,0) to (319,199)

            The routine
                FOR XCOUNT = X1 TO X2
                    SCREEN_POSITION = A000h + ( Y * 320 ) + XCOUNT
                    POKE SCREEN_POSITION, COLOUR
                NEXT XCOUNT
```

[Although this routine looks like BASIC, this program won't work properly in basic. See Part 1 for examples on how to correctly place values in video memory - Phil]

So, this routine takes every X co-ordinate in the line, from one end to the other, calculates the screen position for that co-ordinate, and puts the colour of the line into that memory location. Pretty simple stuff, right!

OK. Now, many (or maybe all) of you will see that this routine is not very efficient. Why? Because it does so much calculation inside the "loop". But we need to know the memory location for each pixel, so how can we change that ... ?

We learnt last month that mode 13h is "linear". That is, consecutive bytes in video memory map to consecutive pixels on the screen. So if we know the memory location of a given pixel (say the left hand side of the line), then surely to draw a line to the right of that, all we need to do is keep adding one to the memory location?

Look at this routine:

```
SCREEN_POSITION = A000h + ( Y * 320 ) + X1
FOR XCOUNT = X1 TO X2
    POKE SCREEN_POSITION, COLOUR
    SCREEN_POSITION = SCREEN_POSITION + 1
NEXT XCOUNT
```

In this routine, before we start the loop, we calculate the position in video memory of the leftmost pixel in the line. Then inside the loop, we put the colour into the memory location, and then add one to the memory location. Because we're counting from X1 to X2, we still draw the same number of pixels, but we don't do very much calculation.

I have written a C program to draw random lines, first using the "slow" method, then using the "optimised" method. The program will suggest the number of lines you should draw for your system. Choose carefully, because the program cannot be stopped once it's running!

(The program is HLD.EXE, and the source is HLD.C)

Depending on your system you may be able to see the difference, but if not the program will tell you the two times taken. You should see a significant difference in times!

Not only have you just learnt how to draw horizontal lines, you've also learned an important lessons about how to optimise your code:

"Always look inside your loops for redundant calculations."

Remember, one millisecond (1/1000th of a second) saved in a loop which executes 1,000 times equals a one second saving in your program. This may not sound like much, but if that routine is required every time you re-draw the screen in your game, it could well make the difference between a playable and an unplayable game.

OK, now let's draw a vertical line!

First, we'll look at the "slow" method. It's almost identical to the horizontal line, but the parameters required are slightly different.

```
Parameters to routine:
    COLOUR: Colour of line
    X: X co-ordinate of the line (same at both ends because
                                  line is vertical!)
    Y1: Y co-ordinate of one end of line
    Y2: Y co-ordinate of the other end of the line

Returned values:
    None

Assumptions
    1. The screen should already be in mode 13
    2. Y2 should be greater that Y1.  Check this!
    3. All co-ordinates are legal for mode 13h.
       Legal co-ordinates range from (0,0) to (319,199)

The routine
    FOR YCOUNT = Y1 TO Y2
        SCREEN_POSITION = A000h + X + ( YCOUNT * 320 )
        POKE SCREEN_POSITION, COLOUR
    NEXT YCOUNT
```

Pretty much as you expected, I hope, and I won't go into an explanation of the routine. OK, how can we optimize this routine?

Well, we know that in mode 13h, the screen is 320 pixels wide, so if we want to move 'down' one pixel at a time, we just need to go ahead 320 bytes in memory, right!? Or, to move up, we move back 320 bytes.

So, look at the optimised routine:

```
    SCREEN_POSITION = A000h + X + ( Y1 * 320 )
    FOR YCOUNT = Y1 TO Y2
        POKE SCREEN_POSITION, COLOUR
        SCREEN_POSITION = SCREEN_POSITION + 320
    NEXT YCOUNT
```

Very much like the horizontal line drawing routine, we calculate the memory location of the first (topmost) pixel, then keep adding 320 bytes to the location to move "down" the line.

I have written a C program to draw random lines, first using the "slow" method, then using the "optimised" method. The program will suggest the number of lines you should draw for your system. Choose carefully, because the program cannot be stopped once it's running!

(The program is VLD.EXE, and the source is VLD.C)

Depending on your system you may be able to see the difference, but if not the program will tell you the two times taken. You should see a significant difference in times!

Congratulations! You know know (and hopefully understand) how to draw horizontal and vertical lines in mode 13h, and you've also learnt something about optimising your programs. When you're completely comfortable with everything above, we'll move on.

Drawing lines which are neither horizontal or vertical

This is where life gets really interesting <grin>. Clearly we're going to have five pieces of information:

```
X1: The X co-ordinate of one end
Y1: The Y co-ordinate of one end
X2: The X co-ordinate of the other end
Y2: The Y co-ordinate of the other end
COLOUR: The colour of the line
```

But how do we know which pixels to draw between the two ends?

There are a number of approaches to this problem, as you might expect. In this article, we'll examing the "slow" method.

We need to make a decision to either work from one X to the other, or from one Y to the other (you'll see what I mean in a moment). This decision is based on the slope of the line.

The first thing we need to know is the "distance" from X1 to X2 and the "distance" from Y1 to Y2. We'll call these two numbers XLENGTH and YLENGTH respectively, ie:

```
XLENGTH = ABSOLUTE ( X2 - X1 )
YLENGTH = ABSOLUTE ( Y2 - Y1 )
```

ABSOLUTE means, make the result positive regardless. ie: ABSOLUTE(-5) = 5. Most languages have an ABS function for this purpose. The reason we do this is, a length of -7 is longer than a length of 5 even though -7 < 5. (try drawing it and you'll see what I mean).

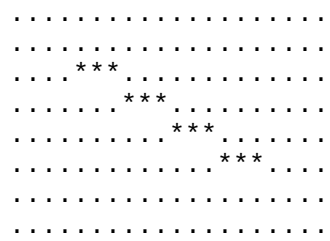
Now, if XLENGTH > YLENGTH, then we will work from X1 to X2. If YLENGTH is greater, we will work from Y1 to Y2. If they're equal, it doesn't make any difference. We call the axis along which we're moving the "step" axis, and the other the "other" axis.

Now, we need to be sure that the "From" step is less than the "To" step, ie: if we're working from X1 to X2, that X1 < X2, or if we're working from Y1 to Y2, that Y1 < Y2. If this is not the case, you must swap the co-ordinates around so that it is.

Remember, if you swap X1 and X2, to also swap Y1 and Y2, and vice versa!

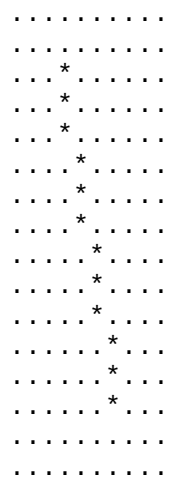
It's best to explain the point of all of this with two diagrams. In each diagram, a full stop is a location and an asterisk is a point on the line.

In this first diagram, XLENGTH is greater, so in our routine we will be moving from X1 (the left) to X2 (the right):



If you follow the line from left to right, you'll see that every time we move one pixel right, sometimes we move down one pixel also, and sometimes we do not.

and in this diagram, YLENGTH is greater, so in our routine we will be moving from Y1 (the top) to Y2 (the bottom):



If you follow the line from top to bottom, you'll see that every time we move one pixel down, sometimes we also move one pixel right, and sometimes we do not.

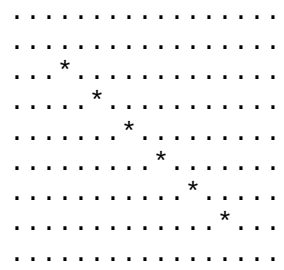
The trick with drawing "sloping" lines is to know when you have to move the "other" co-ordinate, and it's the reason we've been doing so much calculation above.

What is really happening is, for every move along our "step" axis, we're also moving a FRACTION on the "other" axis. But because we can't display a pixel at a fractional location, we display it at the closest possible pixel. This gives the line the "stepped" appearance which is so clear in the above diagrams.

The reason we went to so much trouble to determine whether the line is longer in X or Y is, we want determine which axis we can move along in increments LESS THAN

ONE. Put another way, if we're moving along the X axis, we want to also move along the Y axis in steps of LESS THAN ONE. Can you see why?

Clearly, if we move one pixel along one axis, we don't want to move more than one pixel along the other axis, or the line will end up with "holes" in it! If for every one-pixel step along Y we were to move two pixels in X, we would end up with a line that looked like this:



which is almost certainly NOT what we want. We're trying to draw a solid line!

By moving along the "other" axis in FRACTIONAL increments (<= 1) we can be sure we never "miss" any pixels.

Now we need to know what the fraction we should move is. Well, let's say that we're moving along the X axis. We know that we are going to take (X2 - X1) steps, and that by the time we have done that we want to have moved (Y2 - Y1) along the Y axis, so each step must be the fraction

$$YSTEP = \frac{Y2 - Y1}{X2 - X1}$$

Conversely, if we're moving along the Y axis, then we're going to move YLENGTH steps, and in doing so we want to move XLENGTH along the X axis, so the fraction is

$$XSTEP = \frac{X2 - X1}{Y2 - Y1}$$

IMPORTANT: You can NOT write these formulas as (XLENGTH/YLENGTH). Why? Because when we calculated those values, we forced them to be positive. If you use them in the above calculations, your line may "move" in the wrong direction! XLENGTH and YLENGTH were only useful when we wanted to work out along which axis the line was longer.

So here we are, ready to draw a line. We have the co-ordinates of the two ends, the colour, and the fraction which tells us how far to "move" along the "other" axis. Let's examine some code to draw the line!

```
Parameters to routine:
    COLOUR: Colour of line
    X1: X co-ordinate of one end of the line
```

```
    Y1: Y co-ordinate of one end of the line
    X2: X co-ordinate of the other end of the line
    Y2: Y co-ordinate of the other end of the line

Returned values:
    None

Assumptions
    1. The screen should already be in mode 13
    2. All co-ordinates are legal for mode 13h.
       Legal co-ordinates range from (0,0) to (319,199)

The routine
    XLENGTH = ABSOLUTE( X2 - X1 )
    YLENGTH = ABSOLUTE( Y2 - Y1 )
    IF ( XLENGTH > YLENGTH )
        [REM: WE'RE GOING TO MOVE ALONG THE X AXIS]
        IF ( X1 > X2 )
            SWAP( X1, X2 )
            SWAP( Y1, Y2 )
        ENDIF
        YSTEP = ( Y2 - Y1 ) / ( X2 - X1 )
        Y = Y1
        FOR XCOUNT = X1 TO X2
            PLOT_PIXEL( XCOUNT, Y, COLOUR )
            Y = Y + YSTEP
        NEXT XCOUNT
    ELSE
        [REM: WE'RE GOING TO MOVE ALONG THE Y AXIS]
        IF ( Y1 > Y2 )
            SWAP( X1, X2 )
            SWAP( Y1, Y2 )
        ENDIF
        XSTEP = ( X2 - X1 ) / ( Y2 - Y1 )
        X = X1
        FOR YCOUNT = Y1 TO Y2
            PLOT_PIXEL( X, YCOUNT, COLOUR )
            X = X + XSTEP
        NEXT YCOUNT
    ENDIF
```

Phew! Let's work through this one bit at a time.

First, we calculate the lengths in X and Y, making sure both are Positive so they can be meaningfully compared.

If the X length is greater: first we make sure that X1 < X2. If this is not the case we swap the co-ordinates round so that it becomes true.

We then calculate the fractional 'step' we want to take along the Y axis for every step we move along the X axis.

We also set up a variable to keep track of our 'fractional' position on the Y axis, called Y. Initially, this has the value of the first Y co-ordinate, Y1.

Now, in a loop, we move one pixel at a time from X1 to X2. Each time, we plot a pixel of colour COLOUR at (XCOUNT, Y). We then add YSTEP to our fractional Y co-ordinate, and the loop continues.

(The PLOT_PIXEL routine is not defined, but it takes three parameters - the X co-ordinate, the Y co-ordinate and the colour of the pixel. You can write your own routine from what you learned last month!)

If the Y length is greater, the code is almost exactly the same, except we're moving along the Y axis and keeping track of a fractional X co-ordinate.

Remember, the calculation of XSTEP and YSTEP must use the ORIGINAL co-ordinates, not the calculated values XLENGTH and YLENGTH. This is so important that I'm repeating it again!

Now for the "ah-hah" bit where hopefully the whole line-drawing universe falls into place in your mind. Look again at the routine, but consider what happens if we try to draw a purely horizontal line ...

In this case, Y1 = Y2 and hence YLENGTH = 0. You'll very quickly find that the XSTEP cannot be calculated, as the bottom part of the fraction returns 0. So this routine can't draw horizontal lines.

For a purely vertical line, X1 = X2 and hence XLENGTH = 0. Now we can't calculate the YSTEP because the bottom of the fraction is 0, so this routine can't draw vertical lines either!

So, if you want to put together a general-purpose line drawing routine, which takes the five parameters X1, Y1, X2, Y2, COLOUR, it's going to be of a form something like this:

```
XLENGTH = ABSOLUTE( X2 - X1 )
YLENGTH = ABSOLUTE( Y2 - Y1 )
IF ( XLENGTH = 0 )
    DRAW_VERTICAL_LINE()
ELSE IF ( YLENGTH = 0 )
    DRAW_HORIZONTAL_LINE()
ELSE
    DRAW_SLOPING_LINE()
```

A general purpose C line drawing routine is included with the files attached to this article.

Disadvantages of this line drawing routine

Because this routine uses fractional numbers, this makes it run a lot more slowly. This is true in any language, and it a significant cause of performance loss in many applications.

Because the computer will round fractions off, it is possible, although unlikely, that from (X1,Y1) we may not end up at exactly (X2,Y2). We may be one or two pixels out on the "other" axis.

"Floating point (fractional) math ... makes it run a lot more slowly"

How can I optimise the line drawing routine?

Number 1, most important, most useful, get rid of the floating point (fractional) maths! This is not as easy as it sounds, of course, because we *need* fractions for the XSTEP and YSTEP ... don't we?

In a later article we'll look at another method of drawing sloping lines called "Bresenham's" line drawing algorithm. This eliminates the fractions and adds other improvements besides.

Your Homework

Now that you can draw points and lines, you can start to build up some routines for use later on.

When you've written your own routines for points and lines, try adding routines for:

- **hollow boxes** - they're just two horizontal and two vertical lines, and you can optimise these in a rather special way also (no hints - see how you go).
- **filled boxes** - just a series of horizontal or vertical lines!
- **circles** - only if you're familiar with sines and cosines
- **filled circles** - only if you're getting REALLY daring

and whatever else you like! You now know everything you need to know to get stuff onto the VGA screen, and in another article we'll deal with reading PCX files so you can put pictures on the screen!

Until then, have fun!