EECS205: Fundamentals of Computer Systems Software
Assignment 3 – Bit Blitting
Due: Thursday, February 14, 2019 5:00PM

Overview

This quarter you will implement a single player action video game in IA32 assembly language. The programming assignments will require you to build small models of the game including graphics, game play, and sound which will all be put together by the end of the quarter. This assignment is the next step in your quarter long video game project in IA32 assembly language. For this assignment you will implement blitting routines. Biltters are usually performance critical loops that draw graphics for game-play including sprites for on-screen characters and background images. Because there is such an emphasis on speed, they are often written in assembly language. So, this assignment is not contrived at all. You will write the BasicBlit routine which draws foreground sprites. This assignment will test your ability to implement conditional statements and work with arrays and pointers. You will also write the RotateBlit routine which is capable of drawing a rotated bitmap.

You will need to write a few helper routines. Specifically, you will need to write some trig functions to help with rotation. You will also have to implement DrawPixel (used in the last assignment). Finally, you will need to become acquainted with fixed point math.

This assignment can be rather challenging. You may not be able to finish if you wait until the day before the assignment is due.

Trig Functions

The first two functions that you will write: FixedCos and FixedSin should take single parameters as inputs. These input parameters will be radian angles in fixed point. They should return (in EAX) the fixed point value corresponding to the cosine and sine of the corresponding radian values. For example, if you used the following code sequence:

```
xor eax, eax
INVOKE FixedCos, eax
mov cosx, eax

xor eax, eax
INVOKE FixedSin, eax
mov sinx, eax
```

The variables cosx and sinx should contain the cosine and sine of the radian angle 0.

To make your life a little easier, we have supplied a table of sine values. This table holds the values of the sin (x) for x in the range of [0,Pi/2) in increments of Pi/256. This table is held in memory as an array of WORDs (16 bit elements). The word values correspond to the fractional portion of the sine's fixed point value. For example, the first entry in the table holds the value 0 (since sin (0) = 0). The second value in the table (index 1) holds, the fractional component of sin (Pi / 256). In general, index i of the table holds sin (i * Pi / 256). The table is named SINTAB and holds 128 values total. For this assignment whenever you encounter an angle that does not fit evenly into one of the SINTAB entries, you can choose either of its neighbors. For example, If

you had to find the find the sine of the radian angle 1 = sin(1.0), you would probably quickly figure out that 1 did not have its own table entry. If it did have an entry, it would be between index 81 (whose angle is 0.99401955055) and index 82 (whose entry is 1.00629139685).

```
FixedSin PROTO angle:FXPT
FixedCos PROTO angle:FXPT
```

So you could choose either of these angles as an acceptable result. Note: There are better ways to compute this. See challenge at end of this assignment. So how do we compute the sine for x > Pi/2? Glad you asked. We can use trig identities to compute the sine (and cosine of any angle).

```
sin (x + 2 Pi) = sin (x)
sin (x + Pi) = - sin(x)
sin (x) = sin (Pi – x) (for Pi/2 < x < Pi)
cos (x) = sin (x + Pi/2)
```

So, you can implement the FixedSin function using the first three identities and then implement the FixedCos function by calling FixedSin. This should be relatively easy once you figure out what conditions you need and what calculations you need to perform on the angles. Take a look at the template files. We have provided some useful constants (like Pi/2, Pi, 2 Pi, etc.) in the template file. Feel free to define local variables or helper functions if needed...actually, we think this can make your life a whole lot easier.

You may find it useful to declare local variables to store intermediate calculations or useful values. You may also implement helper functions as needed. Also, it is a good strategy to try to complete this part in phases:

1. Get the sine function working with angles in the range of [0, Pi/2)
2. Improve on your sine function so that it works in the range of [0, Pi)
3. Improve on your sine function so that it works in the range of [0, 2 Pi)
4. Complete your sine function (any angle) and implement cosine

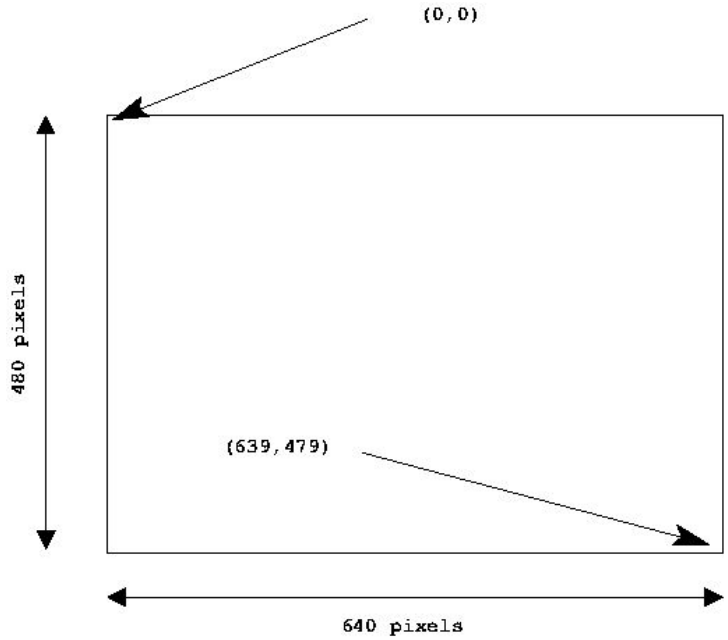
Optional: Note that because we are using lookup tables, we won't the most accurate answer for cases where the angle falls somewhere in between our multiples of Pi / 256 because the lookup process will truncate the answer down to the a multiple of Pi/256. There are a few ways to get better answers. If you feel up for a challenge implement a way to do this. You can for instance, use interpolation to get a better result.

DrawPixel

In the previous assignment, we gave you a handy routine for drawing on the screen...now we take away and make you write it...hahahahaha!!!

For this assignment, you will be responsible for drawing pixels in a backbuffer, a DirectX surface which serves a "workingspace" for your program. The library code that we provide will then copy this working space onto the screen using fast copy operations. This technique is fairly standard in video game design. By doing all the hard work off-screen, we can produce smooth, flicker free animation. You can think of a backbuffer as a large array which contains color values. These color values correspond to pixels on the screen. By changing the value of the colors, you

can determine what will be displayed on the screen. For this assignment, we will be working in a 640x480 8-bit (width,height,color depth) screen mode. That means that each color value is a byte. You could say that ScreenWidth = 640 and ScreenHeight = 480. The coordinate system that we will be using is pretty standard for computer graphics, but is a bit different from what you may use to in math classes. The top left corner of the screen has coordinates (0,0). The bottom right corner of the screen has coordinates (639,479). Under no circumstances do you want to try to draw a pixel off-screen (e.g. outside those boundaries) – bad things might happen if you do. Your code must include checks to make sure that you do not draw outside of the bounds...even if you are given bad/illegal values to your line drawing routine!



The key to accessing the backbuffer the the global variable ScreenBitsPtr. This a pointer which holds the address of the first pixel on the game screen (at coordinate (0,0)). To draw a pixel, you set the corresponding memory location in the backbuffer to a value which represents the color you want. You should think of the entire backbuffer being an array of bytes with a total size of ScreenHeight * ScreenWidth. Adjacent pixels in a row are adjacent in memory and the items in a row wrap around to the next row. By way of example, the first pixel of the first row is at (0,0) at the address held in ScreenBitsPtr, the next pixel of the first row (1,0) is at the next byte address and subsequent pixels in that row are at increasing addresses. When we reach the end of the row (639,0), the next byte contains the pixel color for (0,1) -- the first pixel of the second row. The next pixel (1,1) will be at the next memory address, and so on…

The prototype for the pixel drawing routine will look like this:

```
DrawPixel PROTO x:DWORD, y:DWORD, color:DWORD
```

You know how to use it from the last assignment. Now you need to figure out how to implement it.

Bitmaps

We will define a new structure (EECE205BITMAP) which describes the bitmaps that we are trying to draw on the screen. Take a look at the blit.inc file to see how this structure is defined. The bitmap is essentially a group of pixel values that describe how an image should appear on the screen. The important fields of this structure are dwWidth, dwHeight, bTransparent and lpBytes. They hold the width of the bitmap, its height, color for transparency, and correspond to the start of the bitmap's color values, respectively. Since the color depth for this assignment is 8-bits, the pixel data for a bitmap is of size dwWidth*dwHeight bytes. This is essentially a one-dimensional array of color values. The first value of this array is the first pixel of the bitmap. In general, the pixel corresponding to (x,y) would be index ((y * dwWidth) + x). There is one special bitmap color (specified by bTransparent) which indicates where the bitmap should be transparent. This allows you to have bitmaps which appear in the foreground (since transparent portions will not obscure the background). Note that this is a byte quantity, not a DWORD. You will write procedures that draw bitmaps (w/ transparency) onto the screen at specified coordinates. You should essentially copy the bitmap onto the screen, skipping over pixels that match the transparency color. In the process of drawing, your procedure might find that some or all of the bitmap won't fit into the screen (e.g. some portion of the bitmap doesn't fit into the 640x480 screen). In that case, you should clip the bitmap, drawing the portions that really fit into the screen and not attempt to draw regions which don't fit on the screen. We will provide startup code and library routines to support your module. Basically, all you need to do is write the following procedures (our library code will call them). You can use your star routine source code (star.asm) from the last assignment to create a nice background. You don't have to do anything extra with this code.

You will write two bitmap routines: BasicBlit which draws a simple bitmap to the screen and RotateBlit which draws a bitmap to the screen rotated by a specified angle.

```
BasicBlit PROTO STDCALL ptrBitmap:PTR EECS205BITMAP, xcenter:DWORD,
ycenter:DWORD
```

ptrBitmap holds the address of a EECS205BITMAP. You will need to draw it so that the center of the bitmap appears at (xcenter, ycenter). This is relatively easy to implement. The most straightforward way to do this is with nested for loops that use DrawPixel to draw the bitmap contents to the screen.

```
RotateBlit PROTO STDCALL ptrBitmap:PTR EECS205BITMAP, xcenter:DWORD,
ycenter:DWORD, angle:FXPT
```

ptrBitmap holds the address of a EECS205BITMAP. You will need to draw it so that the center of the bitmap appears at (xcenter, ycenter) and in addition, the bitmap should be rotated by the given radian fixed point angle. You should use your trig functions (FixedSin/FixedCos) and the following pseudocode. This is not the most optimized code...you can actually rewrite it so that there are no multiplication operations in the inner loop. If you feel like a challenge, you can rewrite it so that it just uses addition...contact us if you are interested in this option.

```
/* Note that these are the only fixed point variables used in this code...everything else is integer
*/
cosa = FixedCos(angle)
sina = FixedSin(angle)

esi = lpBitmap

/* Note: Probably easiest to convert ints to fixed and then multiply two
fixed point values */
shiftX = (EECS205BITMAP PTR [esi]).dwWidth * cosa / 2   -  (EECS205BITMAP PTR
[esi]).dwHeight * sina / 2

shiftY = (EECS205BITMAP PTR [esi]).dwHeight * cosa / 2 +  (EECS205BITMAP PTR
[esi]).dwWidth * sina / 2

dstWidth= (EECS205BITMAP PTR [esi]).dwWidth +  (EECS205BITMAP PTR
[esi]).dwHeight;  dstHeight= dstWidth

for(dstX = -dstWidth; dstX < dstWidth; dstX++)
      for(dstY = -dstHeight; dstY < dstHeight; dstY++)
            srcX = dstX*cosa + dstY*sina
            srcY = dstY*cosa – dstX*sina
            if (srcX >= 0 && srcX < (EECS205BITMAP PTR [esi]).dwWidth &&
                srcY >= 0 && srcY < (EECS205BITMAP PTR [esi]).dwHeight &&
               (xcenter+dstX-shiftX) >= 0 && (xcenter+dstX-shiftX) < 639 &&
               (ycenter+dstY-shiftY) >= 0 && (ycenter+dstY-shiftY) < 479 &&
               bitmap pixel (srcX,srcY) is not transparent) then
                        DrawPixel(xcenter+dstX-shiftX, ycenter+dstY-shiftY,
                        bitmap pixel)
```

Getting Started

We will assume that you already have MASM32 downloaded and installed. It may be useful to have the assistance of a debugger for this assignment. Please contact us if you need help setting one up. Next download and unpack the assignment files from the courseweb page.

Then copy your stars.asm, stars.inc, lines.inc, and lines.asm files from the previous assignments into your Assignment 3 directory. In addition, to the usual Makefile tweaks, you will need to modify two files: `blit.asm` and `trig.asm`. Fill in your FixedSin and FixedCos routines in trig.asm. Fill in your own code to implement the two blit routines and `DrawPixel` in `blit.asm`. You will essentially be filling in function bodies. The library code (provided) will call those functions. You really don't have to do anything else. You may find it useful to declare helper variables to store intermediate calculations or useful values. Also, it is a good strategy to try to complete this assignment in phases (e.g. first get FixedSin working, then FixedCos, then move onto the blits (BasicBlit before RotateBlit)).

When working on the blits, first try to make sure that your loop conditions are right. Draw rectangles with solid colors before actually trying to copy the bitmaps. Then move on to copying the bitmap pixels (ignoring transparency). Once that works, implement the transparency checks. Writing in small pieces makes debugging much easier.

Logistics

This assignment is due Thursday February 14 at 5:00 PM. You should hand in your assignment via Canvas. Specifically, you will need to provide the assembly source files `trig.asm` and `blit.asm`. **Do not hand in the executable file (blit.exe)**. To receive full credit, your source files must be fully commented and must compile correctly. Your executable should not crash (e.g. due to some pointer weirdness) or cause images to wrap around the `blit.exe` screen.