UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Lesson 7 Screen02

The Screen02 lesson builds on Screen01, by teaching how to draw lines, and also a small feature on generating pseudo random numbers. It is assumed you have the code for the Lesson 6: Screen01 operating system as a basis.

Contents

1 Dots

Now that we've got the screen working, it is only natural to start wating to create sensible images. It would be very nice indeed if we were able to actually draw something. One of the most basic components in all drawings is a line. If we were able to draw a line between any two points on the screen, we could start creating more complicated drawings just using combinations of these lines.

We will attempt to implement this in assembly code, but first we could really use some other functions to help. We need a function I will call SetPixel that changes the colour of a particular pixel, supplied as inputs in r0 and r1. It will be helpful for future if we write code that could draw to any memory, not just the screen, so first of all, we need some system to control where we are actually going to draw to. I think that the best way to do this would be to have a piece of memory which stores where abouts we are going to draw to. What we should end up with is a stored address which normally points to the frame buffer structure from last time. We will use this at all times in our drawing method. That way, if we want to draw to a different image in another part of our operating system, we could make this value the address of a different structure, and use the exact same code. For simplicity we will use another piece of data to control the colour of our drawings.

To allow complex drawing, some systems use a colouring function rather than just one colour to draw things. Each pixel calls the colouring function to determine what colour to draw there.

Copy the following code to a new file called 'drawing.s'.

```
.section .data
.align 1
foreColour:
.hword 0xFFFF

.align 2
graphicsAddress:
.int 0

.section .text
.globl SetForeColour
SetForeColour:
cmp r0,#0x10000
movhs pc,lr
ldr r1,=foreColour
strh r0,[r1]
mov pc,lr

.globl SetGraphicsAddress
SetGraphicsAddress:
```

```
ldr r1,=graphicsAddress
str r0,[r1]
mov pc,lr
```

This is just the pair of functions that I described above, along with their data. We will use them in 'main.s' before drawing anything to control where and what we are drawing.

Our next task is to implement a SetPixel method. This needs to take two parameters, the x and y co-ordinate of a pixel, and it should use the graphicsAddress and foreColour we have just defined to control exactly what and where it is drawing. If you think you can implement this immediately, do, if not I shall outline the steps to be taken, and then give an example implementation.

Building generic methods like SetPixel which we can build other methods on top of is a useful idea. We have to make sure the method is fast though, since we will use it a lot.

1. Load in the graphicsAddress.
2. Check that the x and y co-ordinates of the pixel are less than the width and height.
3. Compute the address of the pixel to write. (hint: frameBufferAddress + x + y * width * pixel size)
4. Load in the foreColour.
5. Store it at the address.

An implementation of the above follows.

1.
```
.globl DrawPixel
DrawPixel:
px .req r0
py .req r1
addr .req r2
ldr addr,=graphicsAddress
ldr addr,[addr]
```

2.
```
height .req r3
ldr height,[addr,#4]
sub height,#1
cmp py,height
movhi pc,lr
.unreq height

width .req r3
ldr width,[addr,#0]
sub width,#1
cmp px,width
movhi pc,lr
```
Remember that the width and height are stored at offests of 0 and 4 into the frame buffer description respectively. You can refer back to 'frameBuffer.s' if necessary.

3.
```
ldr addr,[addr,#32]
add width,#1
mla px,py,width,px
.unreq width
.unreq py
add addr, px,lsl #1
.unreq px
```
Admittedly, this code is specific to high colour frame buffers, as I use a bit shift directly to compute this address. You may wish to code a version of this function without the specific requirement to use high colour frame buffers, remembering to update the SetForeColour code. It may be significiantly more complicated to implement.

4.
```
fore .req r3
ldr fore,=foreColour
ldrh fore,[fore]
```
As above, this is high colour specific.

5.
```
strh fore,[addr]
.unreq fore
.unreq addr
mov pc,lr
```
As above, this is high colour specific.

2 Lines

The trouble is, line drawing isn't quite as simple as you may expect. By now you must realise that when making operating system, we have to do almost everything ourselves, and line drawing is no exception. I suggest for a few minutes you have a think about how you would draw a line between any two points.

I expect the central idea of most strategies will involve computing the gradient of the line, and stepping along it. This sounds perfectly reasonable, but is actually a terrible idea. The problem with it is it involves division, which is something that we know can't easily be done in assembly, and also keeping track of decimal numbers, which is again difficult. There is, in fact, an algorithm called Bresenham's Algorthm, which is perfect for assembly code because it only involves addition, subtraction and bit shifts.

When programming normally, we tend to be lazy with things like division. Operating Systems must be incredibly efficient, and so we must focus on doing things as best as possible.

Bresenham's Line Algorithm was developed in 1962 by Jack Elton Bresenham, 24 at the time, whilst studying for a PhD.

Algorithm derivation

Bresenham's Algorithm for drawing a line can be described by the following pseudo code. Pseduo code is just text which looks like computer instructions, but is actually intended for programmers to understand algorithms, rather than being machine readable.

```
/* We wish to draw a line from (x0,y0) to (x1,y1), using only a function setPixel(x,y)
which draws a dot in the pixel given by (x,y). */
if x1 > x0 then
    set deltax to x1 - x0
    set stepx to +1
otherwise
    set deltax to x0 - x1
    set stepx to -1
end if

set error to deltax - deltay
until x0 = x1 + stepx or y0 = y1 + stepy
    setPixel(x0, y0)
    if error × 2 ≥ -deltay then
        set x0 to x0 + stepx
        set error to error - deltay
    end if
    if error × 2 ≤ deltax then
        set y0 to y0 + stepy
        set error to error + deltax
    end if
repeat
```
Rather than numbered lists as I have used so far, this representation of an algorithm is far more common. See if you can implement this yourself. For reference, I have provided my implementation below.

```
.globl DrawLine
DrawLine:
push {r4,r5,r6,r7,r8,r9,r10,r11,r12,lr}
x0 .req r9
x1 .req r10
y0 .req r11
y1 .req r12

mov x0,r0
mov x1,r2
mov y0,r1
mov y1,r3

dx .req r4
dyn .req r5 /* Note that we only ever use -deltay, so I store its negative for speed.
(hence dyn) */
sx .req r6
sy .req r7
err .req r8

cmp x0,x1
subgt dx,x0,x1
movgt sx,#-1
suble dx,x1,x0
movle sx,#1

cmp y0,y1
subgt dyn,y1,y0
movgt sy,#-1
suble dyn,y0,y1
movle sy,#1

add err,dx,dyn
add x1,sx
add y1,sy

pixelLoop$:
    teq x0,x1
    teqne y0,y1
    popeq {r4,r5,r6,r7,r8,r9,r10,r11,r12,pc}

    mov r0,x0
    mov r1,y0
    bl DrawPixel

    cmp dyn, err,lsl #1
    addle err,dyn
    addle x0,sx

    cmp dx, err,lsl #1
    addge err,dx
    addge y0,sy

    b pixelLoop$

.unreq x0
.unreq x1
.unreq y0
.unreq y1
.unreq dx
.unreq dyn
.unreq sx
.unreq sy
.unreq err
```

3 Randomness

So, now we can draw lines. Although we could use this to draw pictures and whatnot (feel free to do so!), I thought I would take the opportunity to introduce the idea of computer randomness. What we will do is select a pair of random co-ordinates, and then draw a line from the last pair to that point in steadily incrementing colours. I do this purely because it looks quite pretty.

So, now it comes down to it, how do we be random? Unfortunately for us there isn't some device which generates random numbers (such devices are very rare). So somehow using only the operations we've learned so far we need to invent 'random numbers'. It shouldn't take you long to realise this is impossible.

Hardware random number generators are

The operations always have well defined resutls, executing the same sequence of instructions with the same registers yields the same answer. What we instead do is deduce a sequence that is pseudo random. This means numbers that, to the outside observer, look random, but in fact were completely determined. So, we need a formula to generate random numbers. One might be tempted to just spam mathematical operators out for exmaple: $4x^2! / 64$, but in actuality this generally produces low quality random numbers. In this case for example, if x were 0, the answer would be 0. Stupid though it sounds, we need a very careful choice of equation to produce high quality random numbers.

occasionally used in security, where the predicatability sequence of random numbers may affect the secuirty of some encryption.

The method I'm going to teach you is called the quadratic congruence generator. This is a good choice because it can be implemented in 5 instructions, and yet generates a seemingly random order of the numbers from 0 to 2^{32}-1.

This sort of dicussion often begs the question what do we mean by a random number? We generally mean statistical randomness: A sequence of numbers that has no obvious patterns or properties that could be used to generalise it.

The reason why the generator can create such long sequence with so few instructions is unfortunately a little beyond the scope of this course, but I encourage the interested to research it. It all centralises on the following quadratic formula, where x_n is the nth random number generated.

$$x_{n+1} = ax_n^2 + bx_n + c \bmod 2^{32}$$

Subject to the following constraints:

1. a is even
2. $b = a + 1 \bmod 4$
3. c is odd

If you've not seen mod before, it means the remainder of a division by the number after it. For example $b = a + 1 \bmod 4$ means that b is the remainder of dividing $a + 1$ by 4, so if a were 12 say, b would be 1 as $a + 1$ is 13, and 13 divided by 4 is 3 remainder 1.

Copy the following code into a file called 'random.s'.

```
.globl Random
Random:
xnm .req r0
a .req r1

mov a,#0xef00
mul a,xnm
mul a,xnm
add a,xnm
.unreq xnm
add r0,a,#73

.unreq a
mov pc,lr
```

This is an implementation of the random function, with an input of the last value generated in `r0`, and an output of the next number. In my case, I've used a = $EF00_{16}$, b = 1, c = 73. This choice was arbitrary but meets the requirements above. Feel free to use any numbers you wish instead, as long as they obey the rules.

4 Pi-casso

OK, now we have all the functions we're going to need, let's try it out. Alter main to do the following, after getting the frame buffer info address:

1. Call SetGraphicsAddress with r0 containing the frame buffer info address.
2. Set four registers to 0. One will be the last random number, one will be the colour, one will be the last x co-ordinate and one will be the last y co-ordinate.

3. Call random to generate the next x-coordinate, using the last random number as the input.
4. Call random again to generate the next y-coordinate, using the x-coordinate you generated as an input.
5. Update the last random number to contain the y-coordinate.
6. Call SetForeColour with the colour, then increment the colour. If it goes above $FFFF_{16}$, make sure it goes back to 0.
7. The x and y coordinates we have generated are between 0 and $FFFFFFFF_{16}$. Convert them to a number between 0 and 1023_{10} by using a logical shift right of 22.
8. Check the y coordinate is on the screen. Valid y coordinates are between 0 and 767_{10}. If not, go back to 3.
9. Draw a line from the last x and y coordinates to the current x and y coordinates.
10. Update the last x and y coordinates to contain the current ones.
11. Go back to 3.

As always, a solution for this can be found on the downloads page.

Once you've finished, test it on the Raspberry Pi. You should see a very fast sequence of random lines being drawn on the screen, in steadily incrementing colours. This should never stop. If it doesn't work, please see our troubleshooting page.

When you have it working, congratulations! We've now learned about meaningful graphics, and also about random numbers. I encourage you to play with line drawing, as it can be used to render almost anything you want. You may also want to explore more complicated shapes. Most can be made out of lines, but is this necessarily the best strategy? If you like the line program, try experimenting with the SetPixel funciton. What happens if instead of just setting the value of the pixel, you increase it by a small amount? What other patterns can you make? In the next lesson, Lesson 8: Screen 03, we will look at the invaluable skill of drawing text.