CS 2110 Lab 19
GBA Movement

The TA's :)

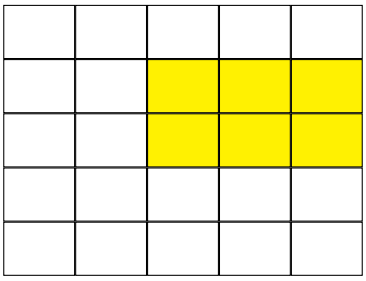
October 31, 2018

1 Recitation Outline

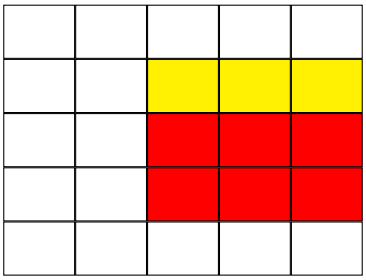
1. Movement with the gameboy
2. waitForVBlank()

2 Movement

Whenever we draw an image or a rectangle on a screen, we tend to use a drawrectangle function that draws a rectangle of specific width, height, and color on the screen at the desired location. For example: drawrectangle(row = 1, col = 2, width = 3, height = 2, color = yellow) would result in the following image:



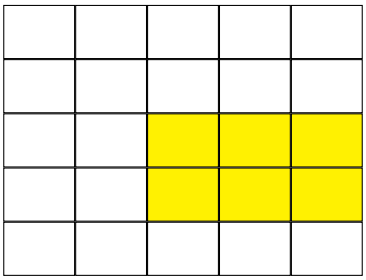
The videobuffer will not be reset automatically, so the pixels will remain yellow. For example, if we drew another rectangle: drawrectangle(row = 2, col = 2, width = 3, height = 2, color = red) would result in the following image:



Consequently, some of the pixels got written over by drawing the new rectangle, but our past operation remains. This especially becomes a problem when we introduce movement because moving a rectangle or an image is the same as redrawing it in a different place. We could redraw the entire background then draw our image/rectangle in its new spot but that is inefficient!

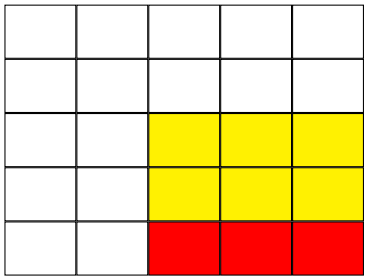
Instead we want to redraw only the portion of the screen that got left behind when our image moved. From the example above if we do

`drawrectangle(row = 1, col = 2, width = 3, height = 2, color = yellow)`
Then moved the rectangle down by one:
`drawrectangle(row = 2, col = 2, width = 3, height = 2, color = yellow)`
We would then need to redraw the background of the rectangle that got left behind:
`drawrectangle(row = 1, col = 2, width = 3, height = 1, color = white)`



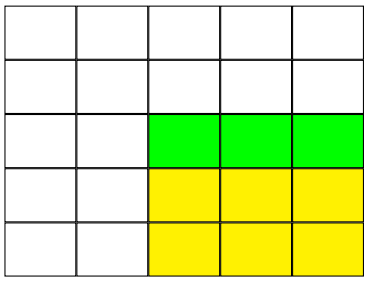
There are four directions for the rectangle to move.

- Rectangle moves up.
  When the rectangle moves up, we would need to redraw the background represented by red in the image below



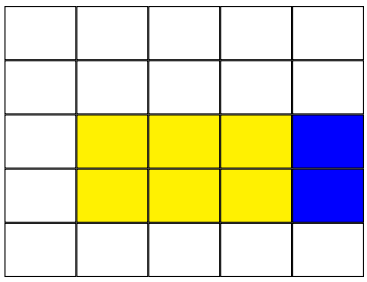
  To draw this we would make a call to drawRectangle like so:
  `drawrectangle(row = oldrow + height - 1, col = col, width = width, height = 1, color = red)` where oldrow is the row where the rectangle was before movement, col, and width are the col and width of the rectangle

- Rectangle moves down
  When the rectangle moves down we would need to redraw the background represented by green in the image below



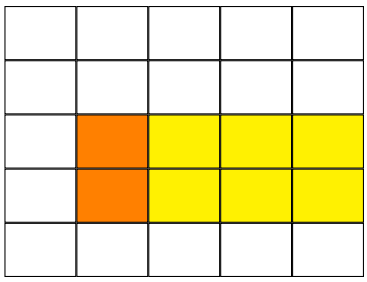
  To draw this, we would make the following call to drawRectangle:
  `drawrectangle(row = oldrow, col = col, width = width, height = 1, color = green)` where oldrow is the topmost row before movement, and col/width are the col and width respectively of the rectangle before movement.

- Rectangle moves left
  When the rectangle moves left we would need to redraw the background represented by blue in the image below



  To draw this, we would make the following call to drawRectangle:
  `drawrectangle(row = row, col = oldcol + width - 1, width = 1, height = height, color = blue)` where oldcol is the top-leftmost col before movement, and row/height are the row/height respectively of the rectangle before movement.

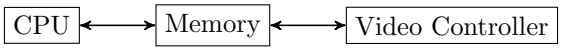
- Rectangle moves right
  When the rectangle moves right we would need to redraw the background represented by orange in the image below



  To draw this, we would make the following call to drawRectangle:
  `drawrectangle(row = row, col = oldcol, width = 1, height = height, color = orange)` where oldcol is the leftmost column of the rectangle before movement, and row/height are the row and height respectively of the rectangle before movement.

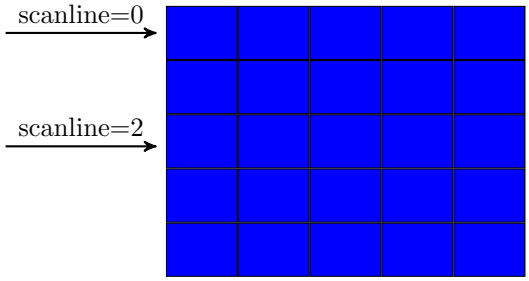
3 WaitForVBlank

The Gameboy Advance (GBA) Architecture looks like:

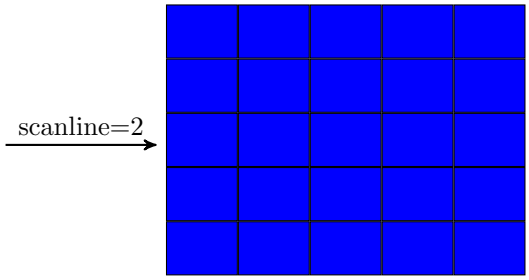


That is, the CPU writes to designated regions of memory containing drawing information, and at some interval, the video controller reads this region of memory and acts on it.
The GBA supports several ways of interpreting this special memory, but this class focuses on Mode 3: a graphics mode in which video memory contains a huge 1D array where each entry is a 16-bit pixel. (You already know how this works if you've done Homework 09!)

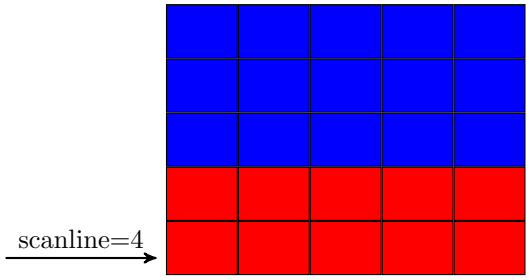
The video controller reads this "video buffer" row-by-row about 60 times per second and updates the screen accordingly. The "scanline counter" is the current row of the video buffer that the video controller is reading. When the scanline counter hits the bottom of the screen, it jumps back to the top. So it goes around and around and around and around, row-by-row. Drawing of a couple of places the scanline could be from our toy example of a 5x5 screen:



However, the CPU and video controller are operating concurrently, which means they don't care about each other. Imagine if, in our toy example from earlier, we had a nice blue screen and we wanted to turn it red, but the scanline counter was halfway down the screen.

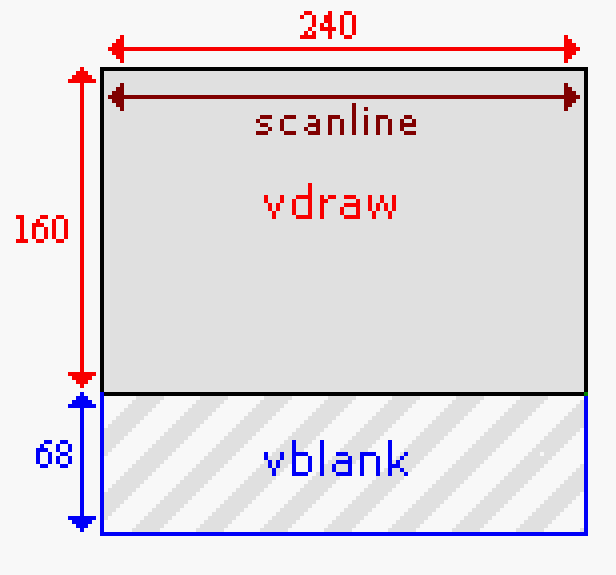


So while the video controller is halfway through drawing the screen, we change the whole videoBuffer to be all red. Then when the video controller finishes drawing the other half, we have a half-blue, half-red screen:



This is called **tearing**, and you want to avoid it because it looks awful. If you are one of those guys who does "video gaming" you might recognize this.

Fortunately, the GBA provides a solution: the vblank! After drawing the actual 160 rows of the screen, the video controller pretends to draw another 68 rows that don't really exist, and takes the same time as it would to actually draw 68 rows for real:



So while the video controller is idling for 68 rows worth of time, we can use this period of time — the "vblank" — to draw whatever we want on the screen. So our game loop should look like:

```
while (1) {
    // Spin until the video controller hits the vblank
    while (*SCANLINECOUNTER < 160) {
        // wait for the beginning of the vblank
    }

    // DRAW STUFF HERE
}
```

The inner while loop spins (hangs) until the scanline counter hits the bottom of the screen, which indicates that the video controller is entering the vblank (starting to idle) so we are safe to draw on the screen without tearing, since we'll have the whole vblank to update videoBuffer.

But wait, what if we enter our loop like *right* at the end of the vblank and we have a lot of drawing to do? We might still tear because we are still drawing even when the vblank finishes, the scanline wraps around, and the video controller starts drawing the top of the screen for real! So we need to add one more loop:

```
while (1) {
    // If we're already in vblank, spin until the scanline has wrapped back around again
    while (*SCANLINECOUNTER > 160) {
        // wait for the scanline to escape the vblank
    }
    // Spin until the video controller hits the vblank
    while (*SCANLINECOUNTER < 160) {
        // wait for the beginning of the vblank
    }

    // DRAW STUFF HERE
```

```
}
```

Now, shorten this:

```
void waitForVblank(void) {
    // If we're already in vblank, spin until the scanline has wrapped back around again
    while (*SCANLINECOUNTER > 160);
    // Spin until the video controller hits the vblank
    while (*SCANLINECOUNTER < 160);
}

void main(void) {
    // (do stuff to configure the video controller here)

    while (1) {
        waitForVblank();

        // DRAW STUFF HERE
    }
}
```

Questions?

Now go do your lab, Happy Halloween bud