

Building a Java First-Person Shooter

Episode 6 – Performance Boosting [last update 5/6/2017]

Objective

In this episode we improve performance of the drawing by drawing less but filling out the same space on the screen. This improves the speed of drawing of multiple rectangles. In addition we enhance the `draw()` method in `Render` to only draw pixels that are “viewable.”

URL

<https://www.youtube.com/watch?v=2tkeKddlVMM>

Discussion

This episode starts off with incorrectly fixing the hard-coding of the width and length values in the `Render.draw()` method.

One of the most important tasks you must do as a programmer is to define clearly the purpose of each class and each method within a class. Having the definition of the roles variables play and the functions methods perform will enable you to fix things much quicker. Because the author never defined the role of a source `Render` class with that of the target `Render` object (the “`this`” in the `draw()` method) he incorrectly adds a `Display` object to the `Render` class. Here is the `Render` class from the last episode (a correctly fixed version).

I recommend ignoring for now the addition of a `Display` object to the `Render` class.

Table 1 - `Render.java`

```
public class Render {


    public final int width;
    public final int height;
    public final int[] pixels;

    public Render(int width, int height) {
        this.width = width;
        this.height = height;
        pixels = new int[width * height];
    }

    public void draw(Render render, int xOffset, int yOffset) {
        for (int y = 0; y < render.height; y++) {
            int yPix = y + yOffset;
            if (yPix < 0 || yPix >= height) continue;
            for (int x = 0; x < render.width; x++) {
                int xPix = x + xOffset;
            }
        }
    }
}
```

```
        if (xPix < 0 || xPix >= width) continue;
        pixels[xPix + yPix * width] =
            render.pixels[x + y * render.width];
    }
}
}
```

I would also like to add since the WIDTH and HEIGHT variables in the Display class are public static fields any programmer knows YOU DON'T NEED TO ADD A DISPLAY OBJECT to Render! Yes, I shouted. One merely references these fields in Render (if you really wanted to limit the usefulness of this method) by using Display.WIDTH and Display.HEIGHT.

	<p>In my opinion, any video of length 20 min that is watched by let's say 4000 viewers should be as correct as possible. We are talking about over 55 days worth of viewing. The ability to influence thousands of programmers to the correct way of thinking about coding and implementing simple concepts is so important.</p>
---	--

How to get more with less – improve performance

A change to code is valid as long as the end result “looks” the same but less effort is expended it generating the screen.

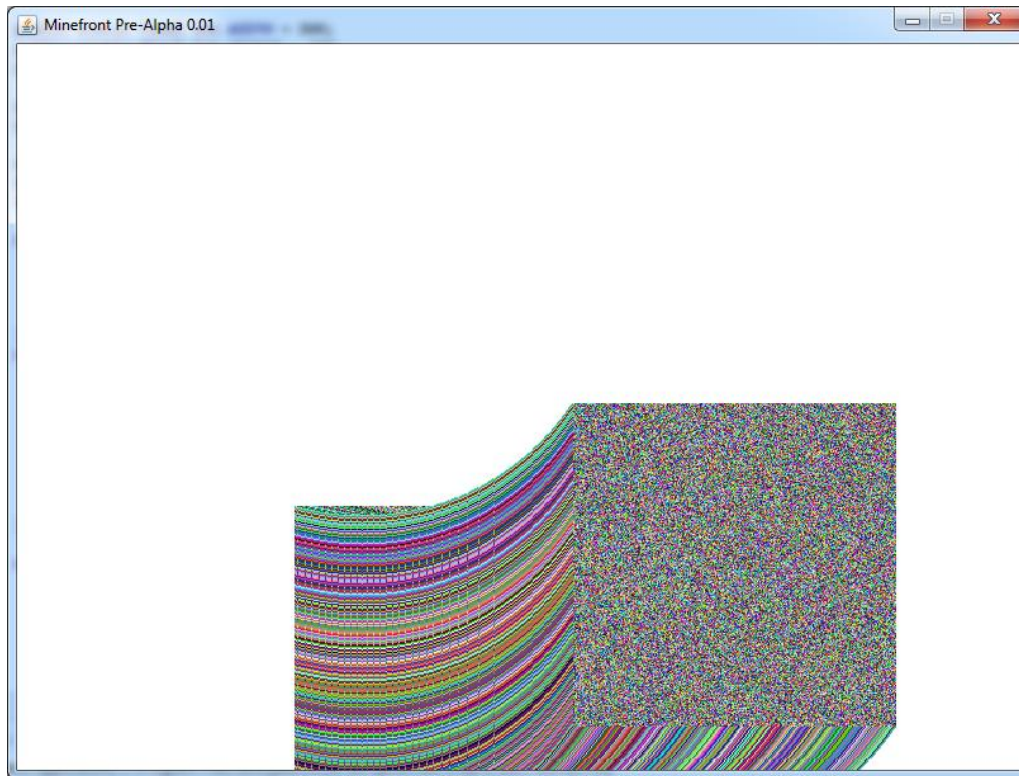


Figure 1 – Drawing 200 colored blocks

The code in our Screen class to generate the above is the following:

Table 2 - Screen.render()

```

public void render() {
    // let's first clear the screen
    for (int i = 0; i < (width * height); i++) {
        pixels[i] = 0xFFFFFFFF;
    }
    int xCenter = (width - BLOCK_SIZE) / 2;
    int yCenter = (height - BLOCK_SIZE) / 2;

    long currentTime = System.currentTimeMillis();
    for (int i = 0; i < 200; i++){
        int anim = (int) (Math.sin( ((currentTime % 1000.0 + i) / 1000) *
Math.PI * 2) * 200);
        int anim2 = (int) (Math.cos( ((currentTime % 1000.0 + i) / 1000)
* Math.PI * 2) * 200);
        draw(test, xCenter + anim, yCenter + anim2);
    }
}

```

We are going to try to draw less colored squares (decreasing the loop count from 200 to 100) but stretch out the squares a bit by multiplying the “i” variable in the equations by 2.

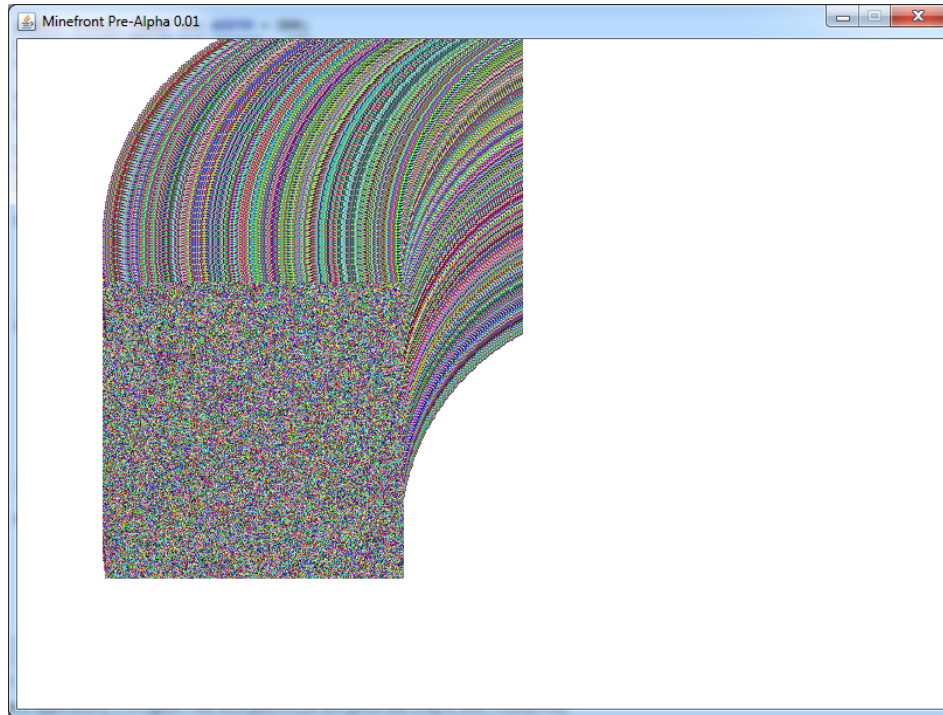


Figure 2 - Decreasing the number of iterations

The code below highlights the changes:

```
for (int i = 0; i < 100; i++){  
    int anim = (int) (Math.sin( ((currentTime % 1000.0 + i * 2) /  
1000) * Math.PI * 2) * 200);  
    int anim2 = (int) (Math.cos( ((currentTime % 1000.0 + i * 2) /  
1000) * Math.PI * 2) * 200);  
    draw(test, xCenter + anim, yCenter + anim2);  
}
```

The reason performance improves is because we are doing less work. We are just spreading the 100 colored squares into a larger space to give the illusion that what is being shown is the same. But, if you look carefully you can see that the colors on the screen will not be as rich. I would regard this as the “poor man’s” performance booster. It would be equivalent to improving your game performance by lowering the resolution that the screen is rendered in. You don’t get everything back – you compromise on something.

To Draw or Not to Draw

This part of the video episode is not correct. So we will present the correct version and prove we have it right.

As you recall the `Render.draw()` method actually copies pixels from a source `Render` object to a target `Render` object. I will repeat an example I used in episode 4 discussion notes.

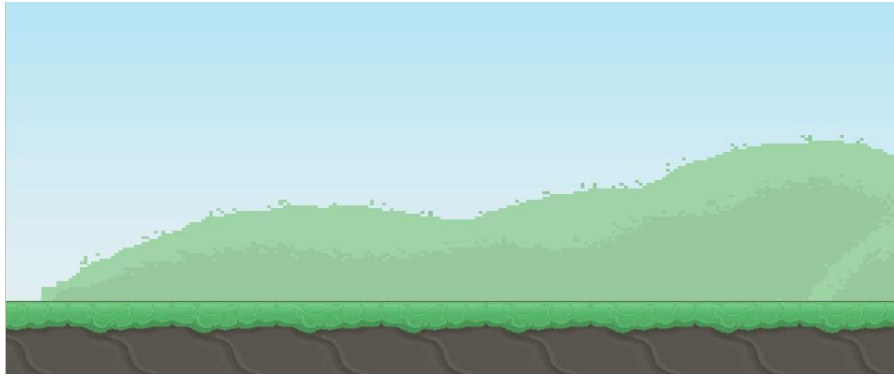


Figure 3 - Drawing our man "Fred" into the landscape

The above is an example of "Fred" image being my "source" Render object that I am copying into the "landscape" by "target" Render object. When I created my "Fred" image I made sure to specify that the "white" pixels around him were set to transparent pixels. That means they should not be drawn to the scene. This will allow the "Fred" image to display correctly once copied over into the landscape.

In order for this to work the `Render.draw()` code would need to test and ignore all pixels in "Fred" that are set to transparent.

This is the updated "landscape" Render object after drawing "Fred" in.



Figure 4 - Drawing all the non-transparent pixels.

CHERNO introduces an if statement to determine if the pixel being moved or copied > 0 , if it is then it is drawn.

Table 3 - `Render.draw()` for episode 6

```
public void draw(Render render, int xOffset, int yOffset) {
    for (int y = 0; y < render.height; y++) {
        int yPix = y + yOffset;
        if (yPix < 0 || yPix >= height) continue;
    }
}
```

```

        for ( int x = 0; x < render.width; x++) {
            int xPix = x + xOffset;
            if (xPix < 0 || xPix >= width) continue;

            int alpha = render.pixels[x + y * render.width];

            if (alpha > 0) {
                pixels[xPix + yPix * width] = alpha;
            }
        }
    }
}

```

The problem with the code is that

- ✚ We are losing sharpness in the image we are drawing
- ✚ It actually slows down

The real problem with the code is that we really should be “anding” alpha with 0x00ffffff, that is:

```

        if ( (alpha & 0x00ffffff) > 0) {
            pixels[xPix + yPix * width] = alpha;
        }

```

Why? Because if we are populating the alpha octet (see episode 4 discussion) in the integer representation of a color with a value greater than or equal to 128 the color number will be < 0 but will still be a valid RGB color!

For the given example, where most of the pixels will NOT be transparent, in fact the probability of encountering a black color (and hence not necessary to draw is rather low 256 / 16777215 or 0.00005). So the supposed “performance boost” we are trying to get DOES not work for the example we have. In most cases (think “Fred”) it will.

Another thing to keep in mind – a black color will not always be our transparent color. It depends on the information stored in the image file.