Building a Java First-Person Shooter

Episode 9 – Beginning 3D [Last Update 5/7/2017]

Objective

In this episode we create the class that we plan on using to render 3D images. We create a new class under the package com.mime.minefront.graphics called Render3D. The class extends our anemic Render class.

URL

https://www.youtube.com/watch?v=RKPEQfkhbAY

Table 1 - Render3D.java

```
package com.mime.minefront.graphics;

public class Render3D extends Render {

	public Render3D(int width, int height) {
		super(width, height);
	}

	public void floor() {
		for (int y=0; y < height; y++) {
			double yDepth = y - height / 2;
			double z = 100.0 / yDepth;
			for (int x = 0; x < width; x++) {
				double xDepth = x - width / 2;
				xDepth *= z;
				int xx = (int) (xDepth);
				pixels[x+y*width] = xx;
			}
		}
	}
}
```

We also update the Screen.java class to utilize this class.

```
package com.mime.minefront.graphics;

import java.util.Random;

import com.mime.minefront.Game;

public class Screen extends Render {

	private Render test;
	private final int BLOCK_SIZE = 256;	// temporary used for testing
```

```
	private Render3D render;

	public Screen(int width, int height) {
		super(width, height);
		Random random = new Random();
		render = new Render3D(width, height);
		/*
		test = new Render(BLOCK_SIZE, BLOCK_SIZE);
		for (int i=0; i < BLOCK_SIZE * BLOCK_SIZE; i++) {
			test.pixels[i] = random.nextInt();
		}
		*/
	}


	public void render(Game game) {
		// let's first clear the screen
		for (int i = 0; i < (width * height); i++) {
			pixels[i] = 0;
		}

		/*
		int xCenter = (width - BLOCK_SIZE) / 2;
		int yCenter = (height - BLOCK_SIZE) / 2;


		for (int i = 0; i < 30; i++){
			//int anim1 = (int) (Math.sin( ((game.time % 1000 + i) / 1000.0)
* 2 * Math.PI) * 100);
			//int anim2 = (int) (Math.cos( ((game.time % 1000 + i) / 1000.0)
* 2 * Math.PI) * 100);
		}
		*/
		render.floor();
		draw(render, 0, 0);
	}
}
```

The results of these changes creates the following image:

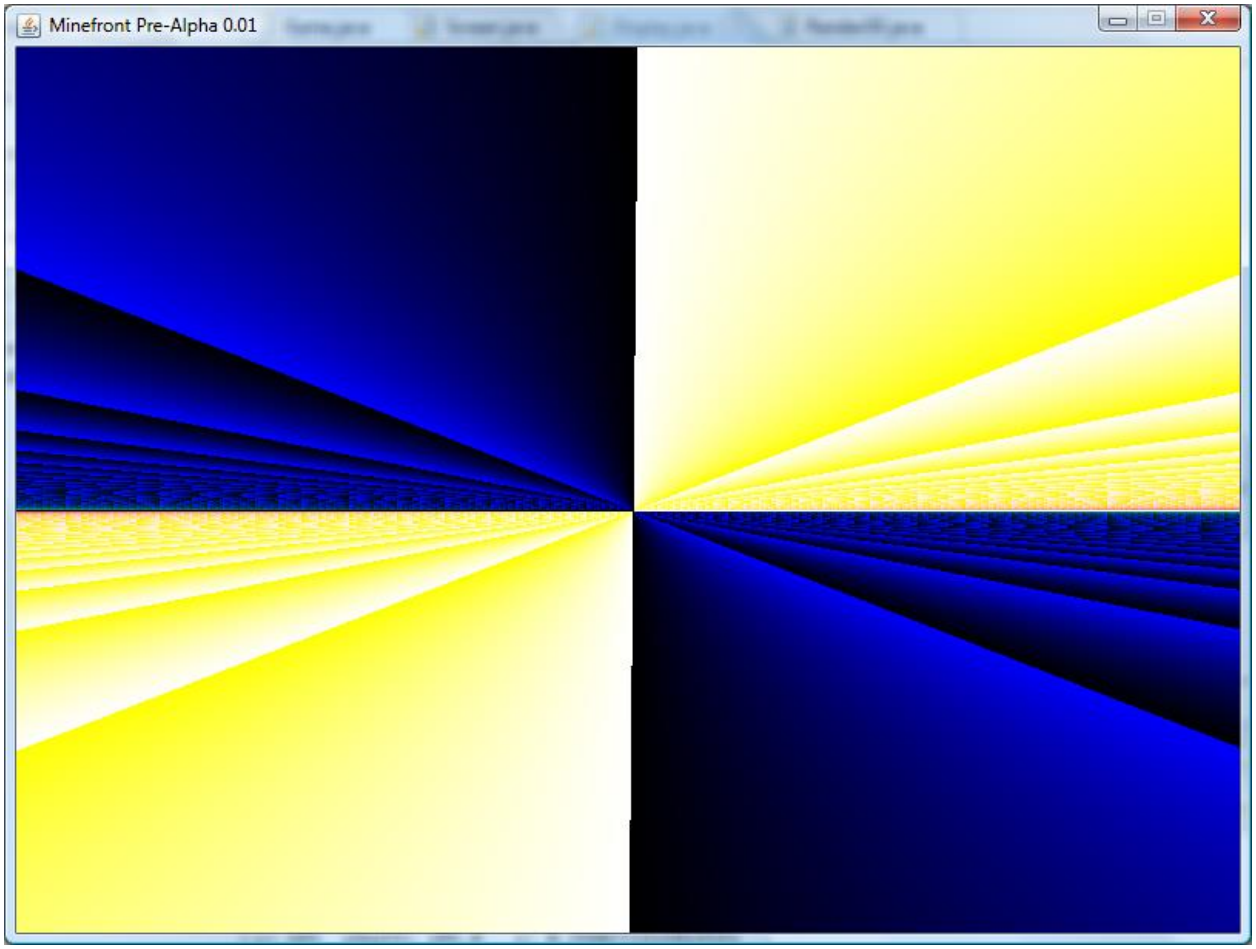

Figure 1 - The result of the initial Render3D.java class

Let's first explain how the above works and then make some changes to get something slightly different.

Let's see why the code partitions into four quadrants. The outermost loop iterates column by column. The inner loop row by row which means we are laying out the screen row by row.

The yDepth value goes from -300 .. 299. The calculation of x comes from the dividing 100 by an increasing value of y that starts at -300. So we get:

100/-300, 100/-299, .. 100/-1,0, 100/1, 100/2, . . . 100/299

The value for z range from -0.33333333 …-100, 0, 100, …

The pattern that is significant is that the z value decreases faster and faster to -100.0 and the pattern repeats on the other side of 0. We can safely suspect that the z pattern accounts or contributes to the dividing border between blue and yellow.

For each column value above we move down row by row. In the inner loop xDepth is initially being set at -400, -399, …0,1…399. We take that value and multiply by z.

We know that the fact that we are seeing blue means that the integer specifiying the color must be small. We know this because the integer representation is as shown below:

Sample Length:	8								8								8								8							
Channel Membership:	Alpha								Red								Green								Blue							
Bit Number:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

It makes sense to see blue since we are multiping x for a very long tile with small values of z. Let's calculate a color close to the border where y = 0 and x = 350 (looks black!)

```
double yDepth = y - height / 2;
```

yDepth = 0 – 600 / 2 = -300;

```
double z = 100.0 / yDepth;
```

z = 100.0 / -300 = - 0.333333..

```
double xDepth = x - width / 2;
```

this leaves x at x = 350 – 800 /2 = 350 – 400 = -50;

So `xDepth *= z;`

Leave us with xDepth = 16 which is very close to black (0x000000[1]), that explains why the the line gets darker as we move along row 0. Note since both xDepth and z are negative the result is negative.

Now why the transition from white to yellow?

From the color chart we see that yellow in RGB format is close to 0xffff00 and of course white is 0xffffff.

Let's do two calculations when x moves over the half way point let's say (420, 0) and closer to the edge (750, 0). You should first note that all our results will be negative when x crosses the half way point since will be `x - width / 2` > 0 and z will be the same -0.33333…

At (420,0) the value of xDepth will be (20 * -.333) or -6.66666666 or in binary

11111111 11111111 11111111 11111010

Which is pretty close to white!

[1] I am leaving the alpha octet off since I just remove from drawing in my version of the code.

When we get to the end of the row (750, 0) the color value is (350 * -0.3333333.) or -116.66666 or in binary:

11111111 11111111 11111111 10001100

The above is pretty darn close to yellow.

The numbers to a flip when we cross the y bounday (and flip the number back to negative and positive as we go across).

Note: The above may not match what you see in the video episode because we fixed the alpha detection in our code.

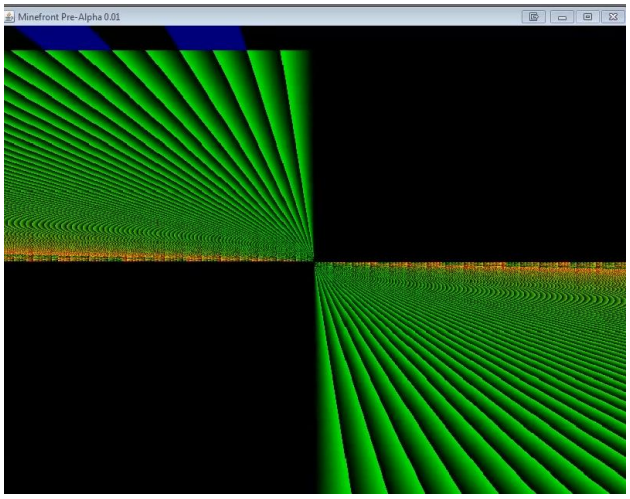

Figure 2 - Example video screen

Shifting and Anding What is gong on?

The shifting and anding (&) will just move the bits around a bit. When we change the color from:

```
int xx = (int) (xDepth);
```

to

```
int xx = (int) (xDepth) & 15;
```

we have making major changes to the color. Regardless of the xDepth value the only bits that are going to matter in determining the color are the last 4 bits since anding xDepth with 0x000001 only make the last four bits of the number matter. In addition, CHERNO makes the following additional change:

FROM:

```
pixels[x+y*width] = xx  & 0x00ffffff;
```

to

```
pixels[x+y*width] = (xx * 128)  & 0x00ffffff;
```

Since we have already change xx to a number in this format 0x00000?

Where ? = xxxx I don't know what the last four bits are but they are the only ones that came out alive and intact after the (& 15) operation.

Multiplying by 128 is equivalent to shifting left by 7 bits. So in our color bit world we get something like

00000000 00000000 00000xxx x0000000

So the final color is close to blue (since 50% of the chance we can have a blue value of 127) or black (with a small touch of green!). See the Screen below.

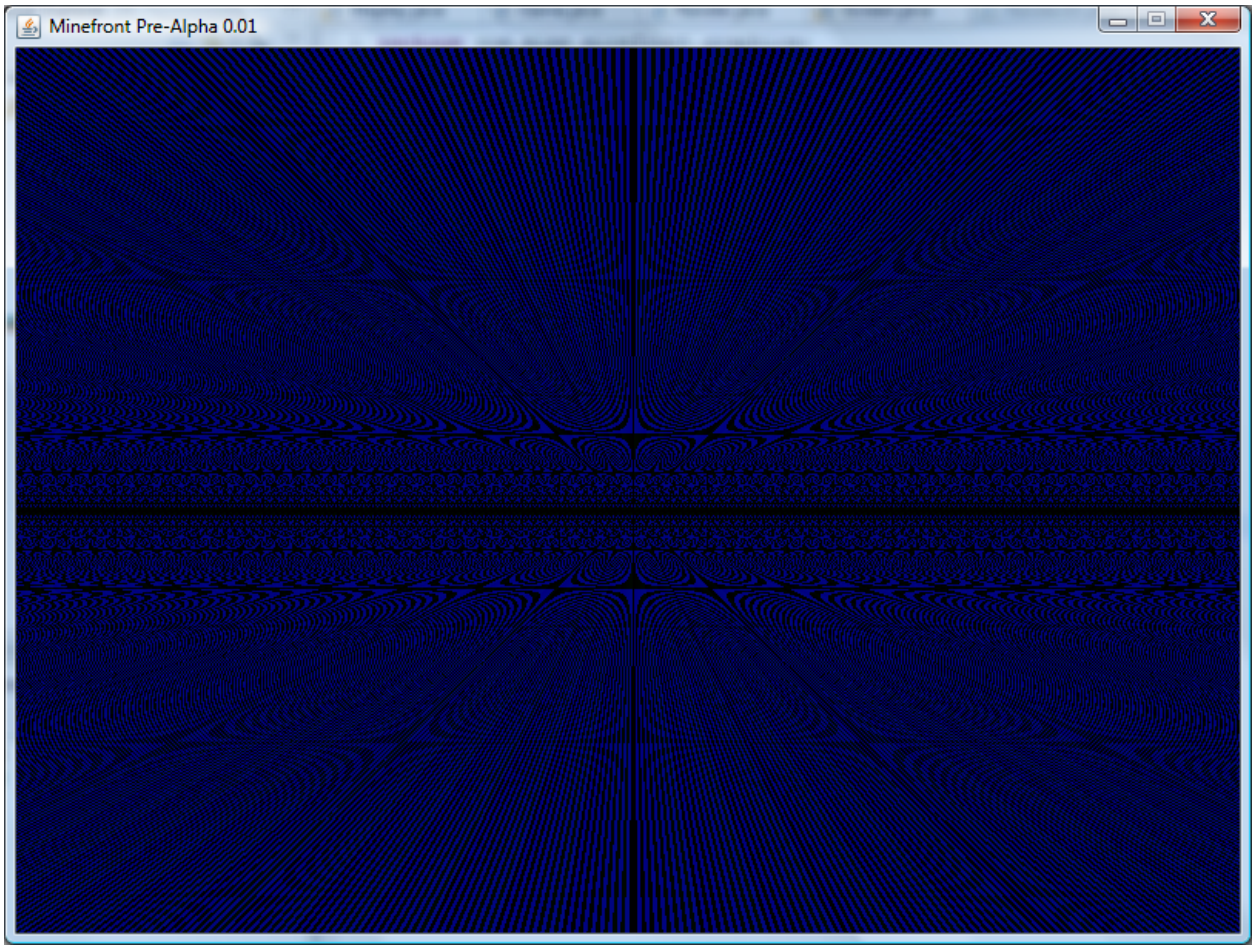

Figure 3 - Playing around with the colors!

I am going to leave this alone. I was trying to find some references for algorithms that automate the generation of textures, floors and walls for games.

Send me mail if you find anything useful that can augment this episode.