Building a Java First-Person Shooter

Episode 5 – Playing with Pixels! [Last update 5/2/2017]

Objective

This episode does not really introduce any new concepts. Two software defects are fixed (one poorly) and we play around with drawing pixels by animating the 256 x 256 random colored pixels we created in the last episode.

URL

https://www.youtube.com/watch?v=dwn_UrItwfw&index=6&list=PL656DADE0DA25ADBB

Discussion

Fix Defect #1 Drawing off the target Render object

The first defect to fix is in the `draw()` method in the `Render` class. As long as the `xOffset`, and `yOffset` stays within the target `Render` object everything works.

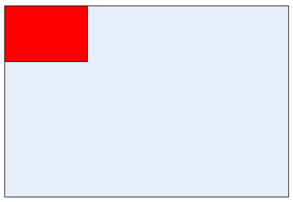


Figure 1 - Using draw() to move 256x256 to the top

We had no problem with the `draw()` command in the last episode since the entire 256 x 256 `Render` object we copied into the target `Render` object (which was 800 x 600) fit since we copied to from the source (0,0), . . . (255, 255) to the same location in the target `Render` object.

But if the `xOffset` or `yOffset` starts with a value that will have eventually have us try to insert a pixel in the source in an area outside the target the program will generate an exception.

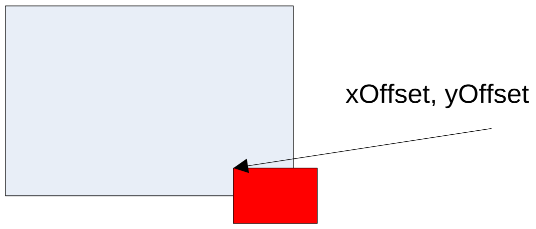


Figure 2 - Using draw() where the source is outside the range of the target

The above figure illustrates the problem. If the xOffset and yOffset starts at (700, 540) the drawing starts fine until we get beyond the pixels defining the target Render object (in gray).

You can test this out by changing the render method in the Screen class to:

```
public void render() {
	draw(test, 700, 540);
}
```

To see where the code breaks add a println statement to the Render draw() method:

Table 1 - Render's draw() method

```
public void draw(Render render, int xOffset, int yOffset) {
	for ( int y = 0; y < render.height; y++) {
		int yPix = y + yOffset;
		for ( int x = 0; x < render.width; x++) {
			int xPix = x + xOffset;
			System.out.println("yPix: " + yPix + "\txPix: " + xPix);

			pixels[xPix + yPix * width] =
			render.pixels[x + y * render.width];
		}
	}
}
```

If you run the above you will see the following output sent to the console:

Table 2 - Output from program

```
yPix: 599	xPix: 786
yPix: 599	xPix: 787
yPix: 599	xPix: 788
yPix: 599	xPix: 789
yPix: 599	xPix: 790
yPix: 599	xPix: 791
yPix: 599	xPix: 792
yPix: 599	xPix: 793
yPix: 599	xPix: 794
yPix: 599	xPix: 795
yPix: 599	xPix: 796
yPix: 599	xPix: 797
yPix: 599	xPix: 798
yPix: 599	xPix: 799
yPix: 599	xPix: 800
Exception in thread "Thread-3" java.lang.ArrayIndexOutOfBoundsException: 480000
	at com.mime.minefront.graphics.Render.draw(Render.java:22)
	at com.mime.minefront.graphics.Screen.render(Screen.java:19)
	at com.mime.minefront.Display.render(Display.java:69)
	at com.mime.minefront.Display.run(Display.java:85)
	at java.lang.Thread.run(Thread.java:619)
```

As you can see from the above since the target Render object ranges from (0,0) . . . (799, 599) the xPix value of 800 into the target “craps” the program out we get an “ArrayIndexOutOfBoundsException.”

The fix is to add checks against the target Render objects width and height.

Table 3 - Fixing Render's draw() method

```
	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];
			}
		}
	}
```

After fixing the above and running the program again we see the following screen:

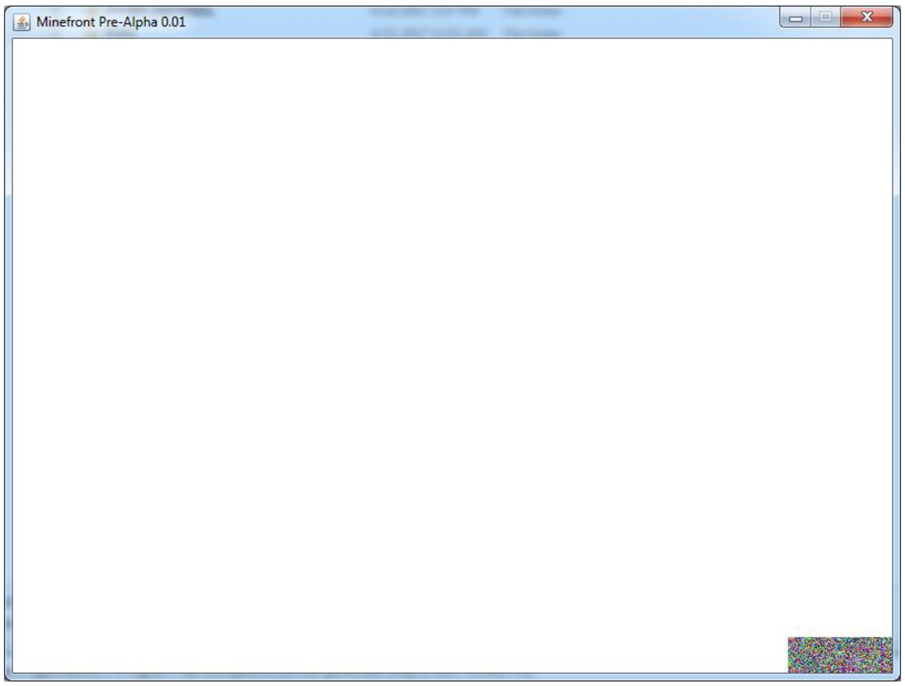


Figure 3 - Result of fixing defect #1

Fix Defect #2 Clearing the screen

The first thing we will do is demonstrate the second software defect we have – we don’t clear the screen before drawing. This is not noticeable as long as things are not moving or being animated on the screen. The first thing we will is center the randomly colored 256 x 256 block on the screen by setting the offsets so that the middle of the block falls into the middle of the screen.

Table 4 - Screen.java version with centered block

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

import java.util.Random;

public class Screen extends Render {

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

	public Screen(int width, int height) {
		super(width, height);
		Random random = new Random();
		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() {
		int xCenter = (width - BLOCK_SIZE) / 2;
		int yCenter = (height - BLOCK_SIZE) / 2;

		draw(test, xCenter, yCenter);
	}
}
```

Note, that unlike the video tutorial version we introduced the use of the constant BLOCK_SIZE. This will allow us to play with the size of the square block while we are figuring out how the animation works.

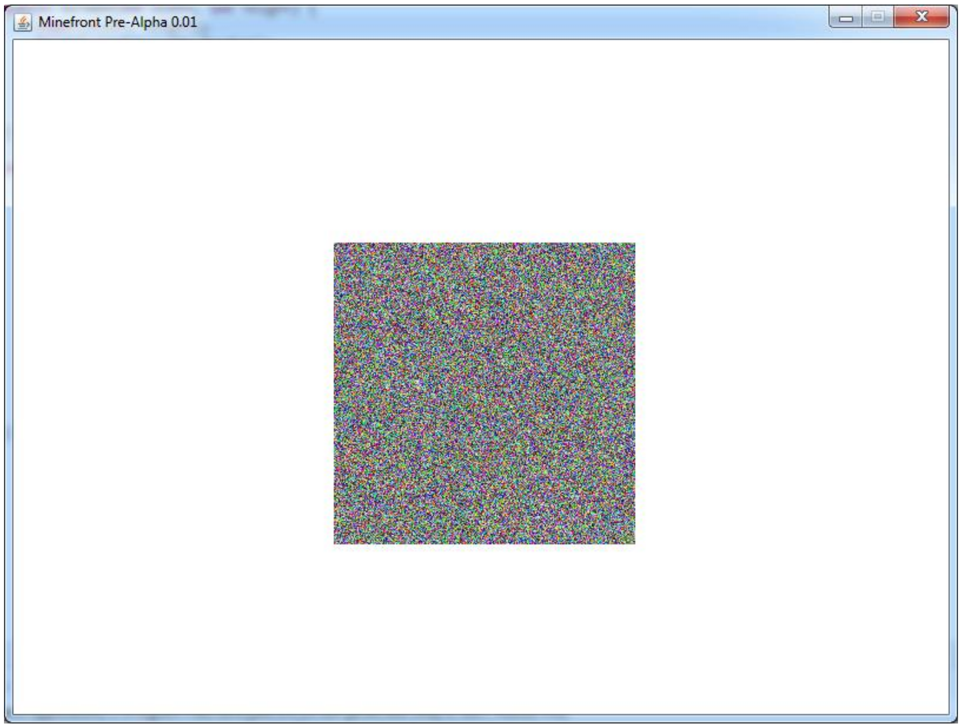


Figure 4 - Centered colored square

We will now introduce animation that will only affect the starting xOffset. We will make use of two new Java methods:

System.currentTimeMillis()

The method System.currentTimeMillis() returns the current time in milliseconds since January 1, 1970. So it basically returns some number that keeps getting larger and larger.

The following program shows you the result of running in one moment in time:

Table 5 - TestCurrentTimeMillis

```
package com.mime.minefront.testarea;

public class TestCurrentTimeMillis {


	public static void main(String[] args) {
		for (int i=0; i < 10; i++) {
			System.out.println(System.currentTimeMillis());
		}
	}
}
```

Table 6 - Result of run #1

```
1356703521896
1356703521897
```

```
1356703521897
1356703521897
1356703521897
1356703521897
1356703521897
1356703521897
1356703521897
1356703521897
```

Later in time (let's figure out how much later…) the results:

Table 7 - Result of run #2

```
1356703651287
1356703651288
1356703651288
1356703651288
1356703651288
1356703651288
1356703651288
1356703651288
1356703651288
1356703651288
```

The accuracy of the data returned varies. How time elapsed between the first and second run?

`(1356703651288 – 1356703521897) ms * (1 sec / 1000 ms) * (1 min /60 sec) = 2.2 min`

The first animation equation used by CHERNO the above value (which you should see as an increasing large number) is taken mod (%) 1000.0. This means that

`(System.currentTimeMillis() % 1000.0`

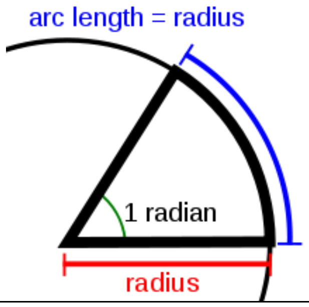
will generate a number from 0..999

Math.sin(double a)

This function returns the trigonometric sine of an angle. The argument a is in radians.

Table 8 - What is radians?

What is radians?



From: http://en.wikipedia.org/wiki/Radian
The radian is the standard unit of angular measure, used in many areas of mathematics. An angle's measurement in radians is numerically equal to the length of a corresponding arc of a unit circle.

The circumference of the unit circle is 2π so the specification of an angle ranges from 0 … 2π (matches our concept of degrees from 0..360°).

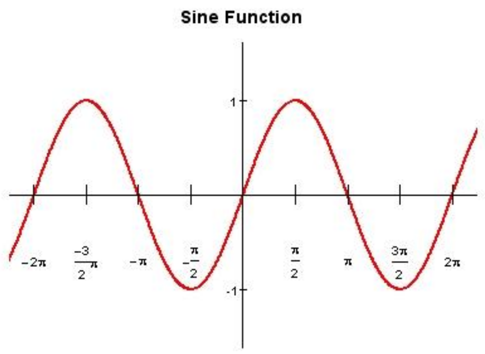


Figure 5- Sine function

As you can see from the above figure the sine function will go from 0..1..-1 and back to 0 while it moves from 0 radians to 2π radians.

In using the following equation:

```
Math.sin(((currentTime % 10000.0 + i) / 10000) * Math.PI * 2) * 100
```

We will be generating a value of sine that goes from -1..0..1. The modulo (%) function guarantees that we will go back and forth that is we will oscillate back and forth since (`System.currentTimeMillis() % 1000.0`) will go back and forth from 0..999 as the time progresses forward. Let's refer to this part of the equation as our timeModulo.

This is a neat trick. Let's imagine that timeModulo gives us 0. Then 0/1000 wil be 0 and that time Math.PI * 2 is just 0 which will generate a Math.sin(0) value of 0. This is in turn multiplied to 100 to give us 0 which means anim = 0. If on the next invocation of render() the timeModulo moves up to let's say 100. Then we get 100/1000 or .1. Which when multiplied by Math.PI * 2 gives us radian value … heck let's write a program to see what is going on!

To see how anim will give us an oscillating value from -100 .. 100. I wrote the following program:

Table 9 - TestSimpleOsillator.java

```
package com.mime.minefront.testarea;

public class TestSimpleOsillator {
	public static void main(String[] args) throws Exception {
		for (int i=0; i < 10; i++) {
			long currentTime = System.currentTimeMillis();
			int timeModulo = (int) (currentTime % 1000);
			double fracTime = timeModulo / 1000.0;
			double radianValue = fracTime * Math.PI * 2;
			double sinValue = Math.sin(radianValue);
			int anim = (int) (sinValue * 100);

	System.out.println("time\ttimeModulo\tfracTime\trandianValue\tsinValue\tanim")
;
			System.out.println(""+currentTime +"\t" + timeModulo + "\t" +
fracTime +"\t"
					+ radianValue+"\t"+sinValue +"\t"+anim + "\t");
			Thread.sleep(100);
		}
	}
}
```

Which creates the following table (note: With the loop I generate only 10 values and sleep for 100 ms before heading back for the next iteration.)

currentTime	timeModulo	fracTime	radianValue	sinValue	Anim value
1356712597666	666	0.666	4.184601	-0.863923	-86
1356712597767	767	0.767	4.81920	-0.994300	-99
1356712597867	867	0.867	5.447521	-0.741741	-74
1356712597967	967	0.967	6.075840	-0.205862	-20
1356712598067	67	0.067	0.420973	0.408649	40
1356712598167	167	0.167	1.049291	0.867070	86
1356712598267	267	0.267	1.677610	0.994300	99
1356712598367	367	0.367	2.305929	0.741741	74
1356712598467	467	0.467	2.934247	0.205862	20
1356712598567	567	0.567	3.562566	-0.408649	-40

As you can see this is a cool trick to get some value (in this case our xOffset) to move smoothly between two points -100 … 100 or xCenter – 100 …xCenter + 100. So there is never any reason you can't get that game character (esp. the villains or boiling pots of oil) to oscillate on the screen.

Figure 6 - Megaman game screen

In fact, I recall many of the easy obstacles in Megaman (and other games) only requiring that you figure out the rate in which things move up or down or back and forth in a similar oscillating fashion.

Playing with Animation

At this point you should be comfortable with the formula below and what it does.

```
int anim = (int) (Math.sin( ((System.currentTimeMillis() % 1000.0) / 1000) * Math.PI
* 2) * 100);
```

We are basically moving along the sine wave (see Figure 5 above) and multiplying the result with 100.

Here is the updated version of Screen.java:

Table 10 - Screen.java

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

import java.util.Random;

public class Screen extends Render {

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

	public Screen(int width, int height) {
		super(width, height);
		Random random = new Random();
		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() {
		int xCenter = (width - BLOCK_SIZE) / 2;
		int yCenter = (height - BLOCK_SIZE) / 2;

		int anim = (int) (Math.sin( ((System.currentTimeMillis() % 1000.0) / 
1000) * Math.PI * 2) * 100);
		draw(test, xCenter + anim, yCenter);
	}
}
```

This result of this version is that the square will move left and right as the anim goes from -100 .. 100.

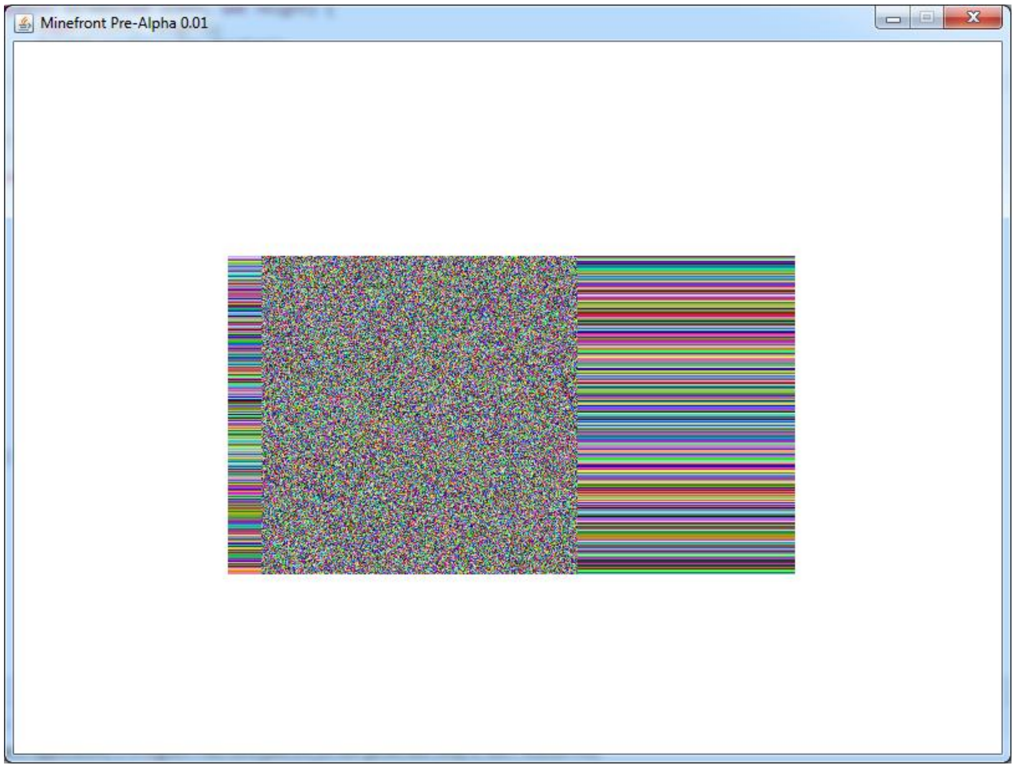


Figure 7 - Demonstration of oscillating square

The key point to the above example is to demonstrate how we are failing to clean up the screen between screen renders. The way to solve the problem is quite simple. Before the render() function invokes draw() it should first clear the entire Screen render area.

But before we insert the code to do this let's discuss the various parts of the anim equation and what changing them will do for us.

```
int anim = (int) (Math.sin( ((System.currentTimeMillis() % 1000.0) / 1000) * Math.PI 
* 2) * 100);
```

The 100 at the end is the portion or adjustment that is added to xCenter. If you wanted the colored square to move further along the screen you would just need to adjust this value. For example if you wanted to have it move 200 pixels to the left and right of the starting xCenter position just change the value. Try it.

```
int anim = (int) (Math.sin( ((System.currentTimeMillis() % 1000.0) / 1000) * Math.PI
* 2) * 100);
```

If you wanted to slow down the movement of the colored square then you would just need to make the differences between each iteration smaller so that it will basically map to the same sine value. Try changing the highlighted 1000.0 to 10000.0 and 1000 to 10000. You should see things slow down quite a bit. This is because during each iteration more values are mapping into the same integer. It is like clumping more values together because 0.00001 is effectively the same as 0.00002.

The last change CHERNO makes is to the value we multiply Math.PI by:

```
int anim = (int) (Math.sin( ((System.currentTimeMillis() % 1000.0) / 1000) * Math.PI
* 2) * 100);
```

Changing this value to a value greater than 2 that is not a multiple of 2 (e.g. 3 or 5) will make the colored square “jump”. This is because we have changed the sine wave as so it iterates before its cycle is reached.

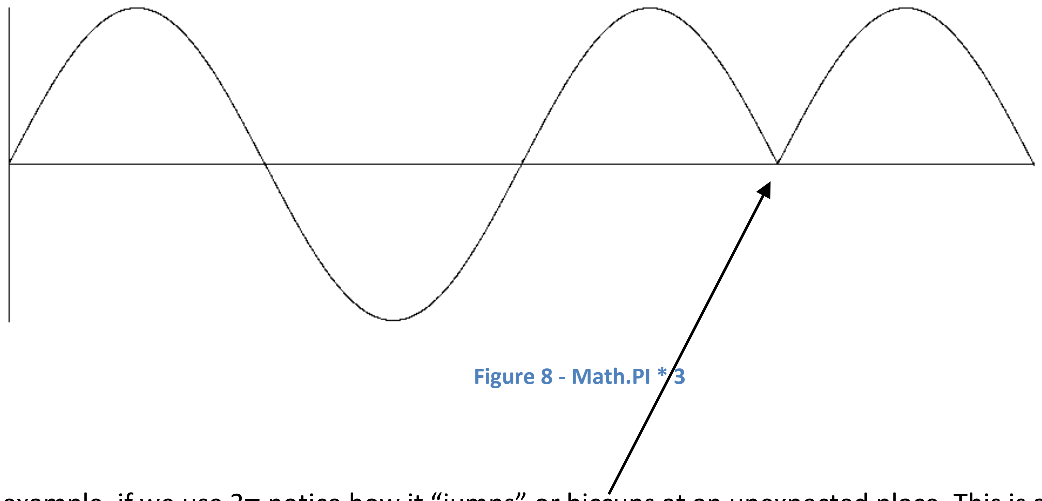


Figure 8 - Math.PI * 3

For example, if we use 3π notice how it “jumps” or hiccups at an unexpected place. This is another great technique for introducing a “hiccup” in the movement of an object. A value less than 2π will just have it cut short its cycle and return back to the start.

Before we fix the clear screen problem lets add a second animation variable (anim2) where we update the yOffset value. Rather than using sin function we will use the cosine function.

Table 11 - Updated Screen.render() method

```
	public void render() {
		// let's first clear the screen
		//for (int j = 0; j < height)
		int xCenter = (width - BLOCK_SIZE) / 2;
		int yCenter = (height - BLOCK_SIZE) / 2;

		int anim = (int) (Math.sin( ((System.currentTimeMillis() % 10000.0) / 
10000) * Math.PI * 2) * 200);
		int anim2 = (int) (Math.cos( ((System.currentTimeMillis() % 10000.0) / 
10000) * Math.PI * 2) * 200);
		draw(test, xCenter + anim, yCenter + anim2);
	}
```

An interesting pattern will now emerge since the following principle is true about sine, cosine and a circle:

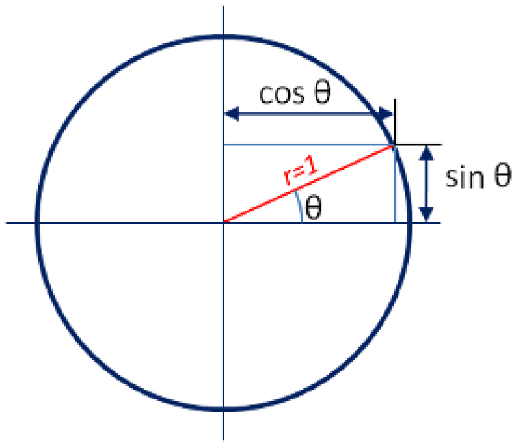


Figure 9 - sine, cosine and living on a circle

As you can see by definition the (cos θ, sin θ)[1] defines a point that resides on the circumference of the unit circle.

The screen now looks like the image below.

[1] Our x's and y's were reversed in this example but it does not make a difference …try reversing the sin and cos in the program.

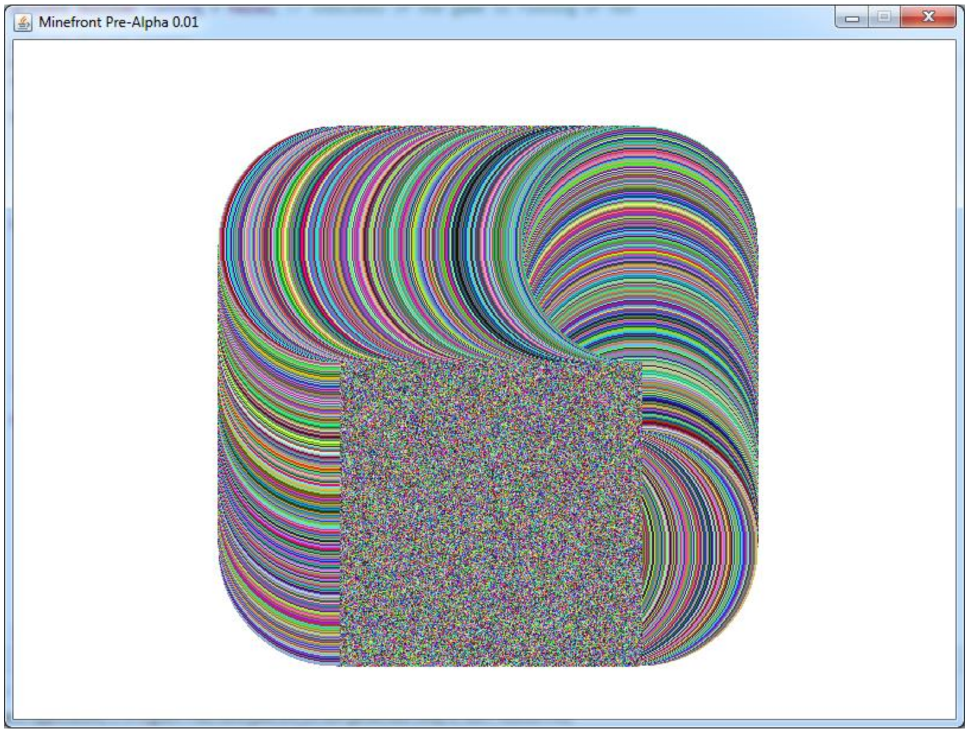


Figure 10 - xOffset and yOffset moving in a circular pattern

Clearing the Screen

Clearing the screen is quite simple and only requires that before we invoke the draw method in Screen.render() method that we “black” out the screen.

Table 12 - Final version of Screen.render() for this episode

```
	public void render() {
		// 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;

		int anim = (int) (Math.sin( ((System.currentTimeMillis() % 10000.0) / 
10000) * Math.PI * 2) * 200);
		int anim2 = (int) (Math.cos( ((System.currentTimeMillis() % 10000.0) / 
10000) * Math.PI * 2) * 200);
		draw(test, xCenter + anim, yCenter + anim2);
	}
```

The final version looks like the figure below:

B

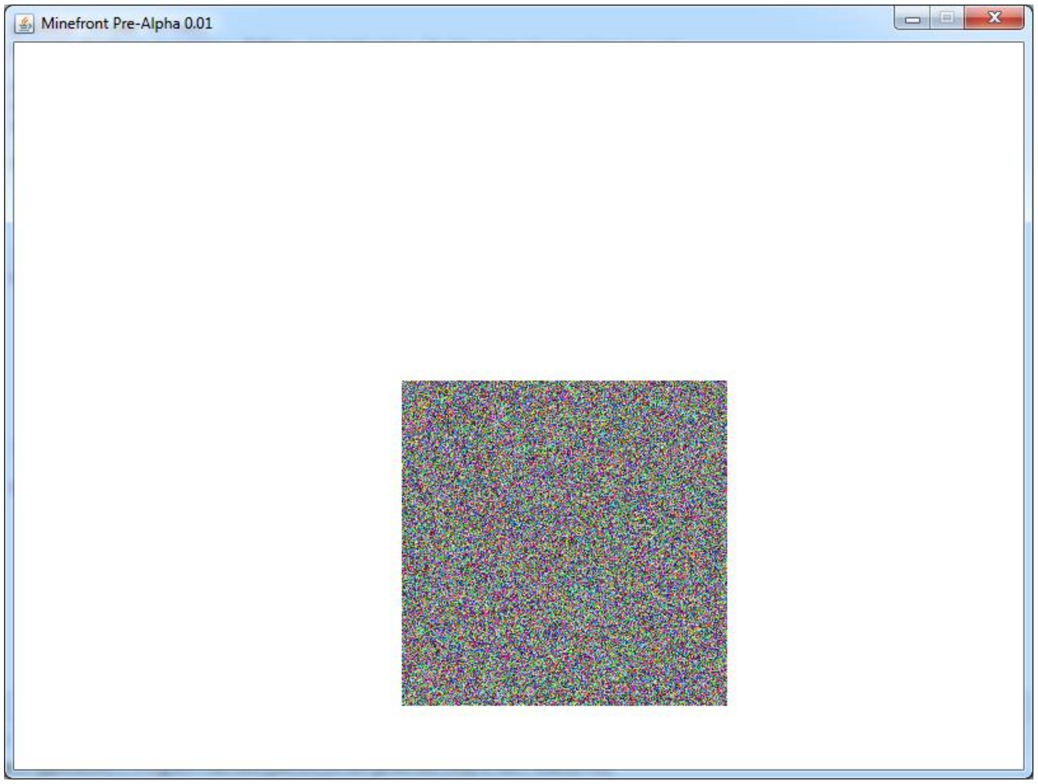


Figure 11 - Clearing the screen between frames

One Last Thing

The last trick to look at was drawing multiple copies of the colored square where each version is slightly offset from the previous one.

The trick is to create a loop that invokes the Render.draw() method several times and each time we update the animation variables slightly. I have modified the code slightly from the video since I want to use the same timeModulo in both statements.

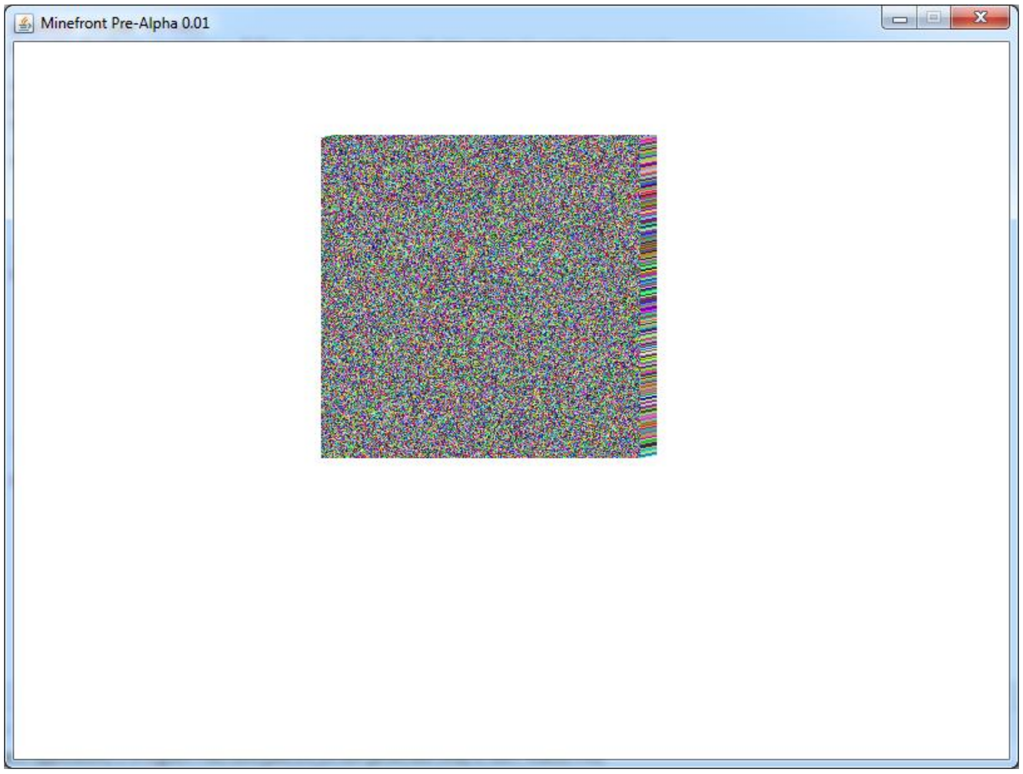


Figure 12 - Drawing 200 times our colored square

The code to print the above and its shadow is the following:

Table 13 – Drawing 200 copies of the colored square

```
	public void render() {
		// 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;

		long currentTime = System.currentTimeMillis();
		for (int i = 0; i < 200; i++){
			int anim = (int) (Math.sin( ((currentTime % 10000.0 + i) / 10000)
* Math.PI * 2) * 200);
			int anim2 = (int) (Math.cos( ((currentTime % 10000.0 + i) /
10000) * Math.PI * 2) * 200);
			draw(test, xCenter + anim, yCenter + anim2);
		}
	}
```

You should be suspicious about the fact that the previous figure does not look like it drew 200 pixel squares. It definitely generated 200 colored squares but given how slow (10000.0) the timeModulo is moving that means there are many overlaps.

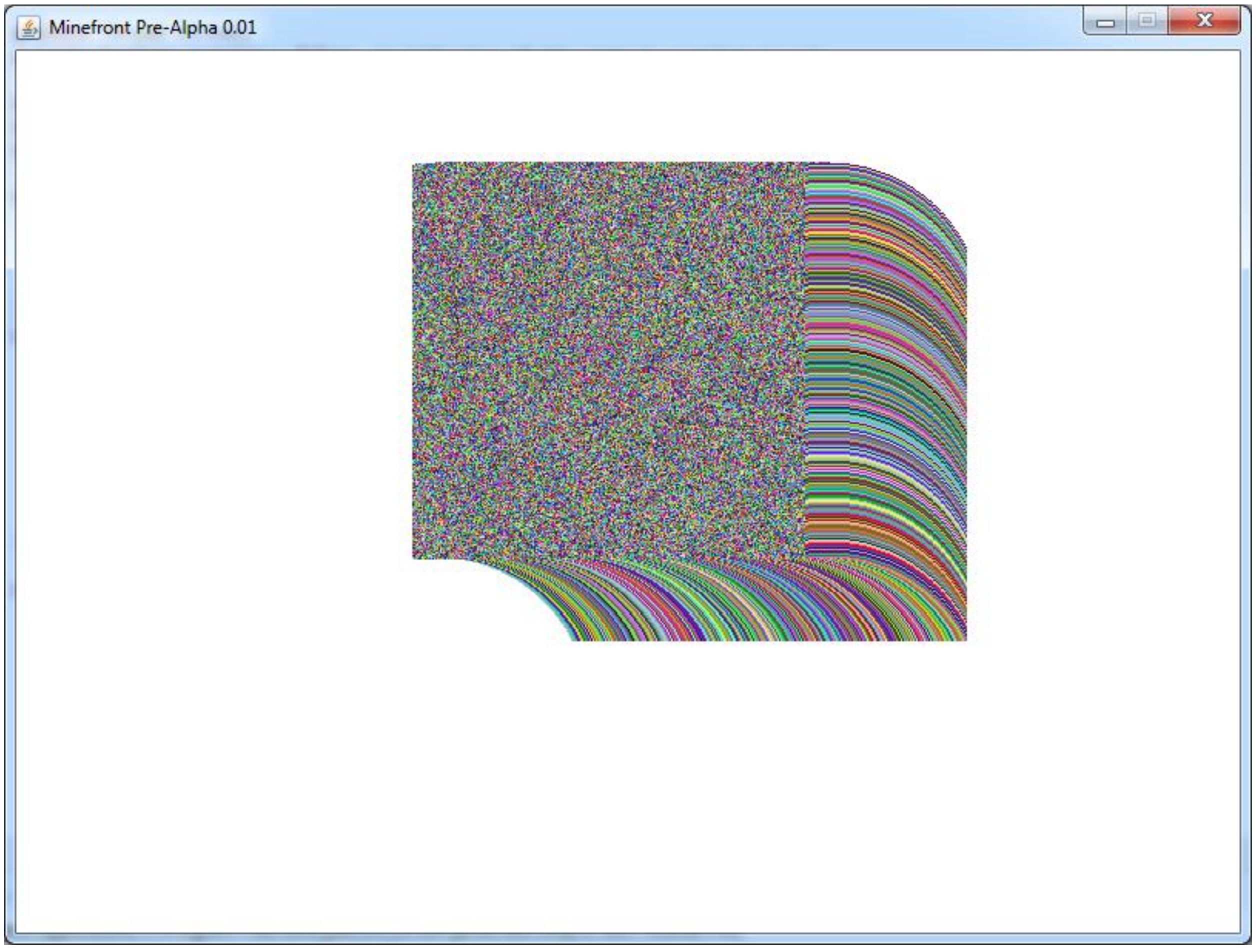


Figure 13 - Increasing the timeModulo

When we change it from 10000.0 to 1000.0 we see that it appears to be more colored squares but the fact is we are still only drawing 200 just spread out in a larger space.

Code

There were only two files updated in this episode.

Table 14 - Render.java

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

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];
			}
		}
	}
}
```

And Screen.java

Table 15 - Screen.java

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

import java.util.Random;

public class Screen extends Render {
	private Render test;

	public Screen(int width, int height) {
		super(width, height);
		Random random = new Random();
		test = new Render(256, 256);
		for (int i=0; i < 256 * 256; i++) {
			test.pixels[i] = random.nextInt();
		}
	}

	public void render() {
		// let's first clear the screen
		for (int i = 0; i < (width * height); i++) {
			pixels[i] = 0xFFFFFFFF;
		}
		int xCenter = (width - 256) / 2;
		int yCenter = (height - 256) / 2;

		long currentTime = System.currentTimeMillis();
		for (int i = 0; i < 200; i++){
			int anim = (int) (Math.sin( ((currentTime % 10000.0 + i) / 10000)
* Math.PI * 2) * 200);
			int anim2 = (int) (Math.cos( ((currentTime % 10000.0 + i) /
10000) * Math.PI * 2) * 200);
			draw(test, xCenter + anim, yCenter + anim2);
		}
	}
}
```