Building a Java First-Person Shooter

Episode 3 – Arrays [Last update: 4/30/2017]

Objectives

This episode discusses the use of arrays in Java and the introduction of a class that will handle what gets rendered/drawn to the screen. This episode adds a new class to the project: `Render.java` and two essential methods are added to our game loop – `tick()` and `render();`

Discussion

A new class is introduced called `Render.java`. It is placed in a new package `com.mime.minefront.graphics`. The class is being created with the intention of being subclassed in a later episode by the class `Screen`. Its purpose is to specify what should be drawn to the screen. The ideas presented in this episode are quite simple if you already know Java programming.

Table 1 - 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];
	}
}
```

The class is instantiated with two values the `width` and `height` of the object to be rendered or drawn. One example will be our `Screen` object that represents the screen display. The constructor then initializes an array named `pixels`.

What is a pixel?

The video display or monitor is composed of thousands (or millions) of pixels. A *pixel* is short for picture element, it represents a single point in a graphic image. A pixel can be constructed by one or more dots on the screen. As programmers we conceptualize or think of the screen as being composed of these pixels, where each pixel can be a particular color.

How the display screen is organized

For this discussion we will assume our screen size is 640x480, that is, there are 640 pixels across and 480 pixels down.
The direction across the screen from left-to-right is regarded as the X → direction.

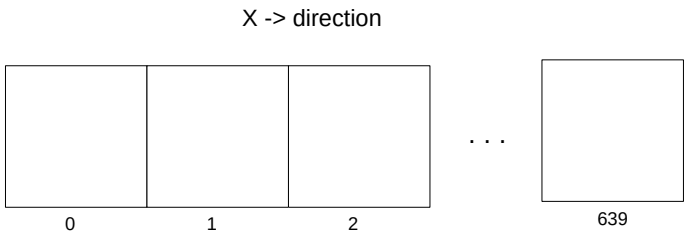

Figure 1 – 640 pixels in the x direction

Each pixel cell across is numbered from 0 to MAX_WIDTH-1 which for our example will be from 0 to 639.

This direction is referred to as the y → direction and the rows are numbered from 0 through MAX_HEIGHT-1.

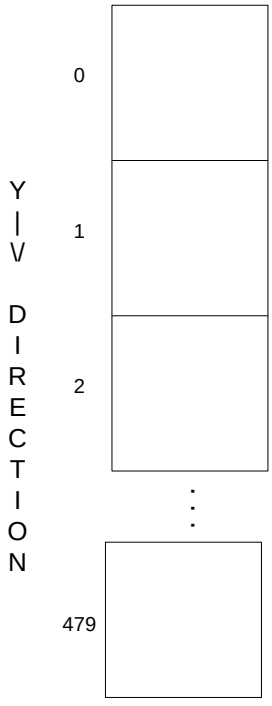

Figure 2 - Rows or y direction

In our example, the y value will be from 0 to 479.
You can view the video display as consisting of a grid of pixels.

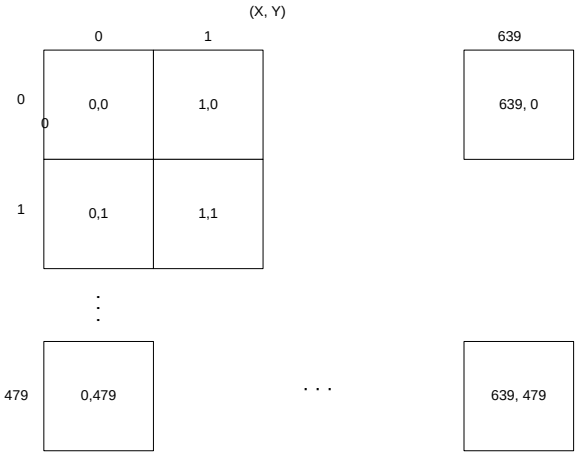

Figure 3 - Grid of pixels

Each pixel is addressable and can be set to a specific color. We usually refer to the pixel address as a tuple (x,y) the first number specifies the x location and the second the y location. The top-left most location (as shown in Figure 3) is location (0,0) and the rightmost address on that row has the address (639,0). On the second row the pixel address starts at (0,1) and goes through to (639,1). The bottom-right most pixel has the address (639, 470).

We will see later how we can map this grid of pixels into the one-dimensional array structure named pixels.

There are only minor changes made to `Display.java`. The program still will display the same empty window.

Table 2 - Episode 3 version of Display.java

```
package com.mime.minefront;

import java.awt.Canvas;
import javax.swing.JFrame;

import com.mime.minefront.graphics.Render;

public class Display extends Canvas implements Runnable{

	private static final long serialVersionUID = 1L;

	public static final int WIDTH = 800;
	public static final int HEIGHT = 600;
	public static final String TITLE = "Minefront Pre-Alpha 0.01";

	private Thread thread;
	private boolean running = false; // indicates if the game is running or not

	private Render render;

	public Display() {
		render = new Render(WIDTH, HEIGHT);
```

```
	}
	private void start() {
		if (running) {
			return;
		}

		running = true;
		thread = new Thread(this);
		thread.start();
		System.out.println("Working");
	}

	private void stop() {
		System.out.println("stop() method invoked.");
		if (!running) {
			return;
		}
		running = false;
		try {
			thread.join();
		} catch (Exception e) {
			e.printStackTrace();
			System.exit(0);
		}
	}

	private void tick() {

	}

	private void render() {

	}


	@Override
	public void run() {
		while(running) {
			tick();		// handles the timing
			render();	// handles rendering things to the screen
		}

	}
	public static void main(String[] args) {
		Display game = new Display();
		JFrame frame = new JFrame();
		frame.add(game);
		frame.setTitle(TITLE);
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.setSize(WIDTH, HEIGHT);
		frame.setLocationRelativeTo(null);
		frame.setResizable(false);
		frame.setVisible(true);

		System.out.println("Running...");
```

```
            game.start();
        }

}
```

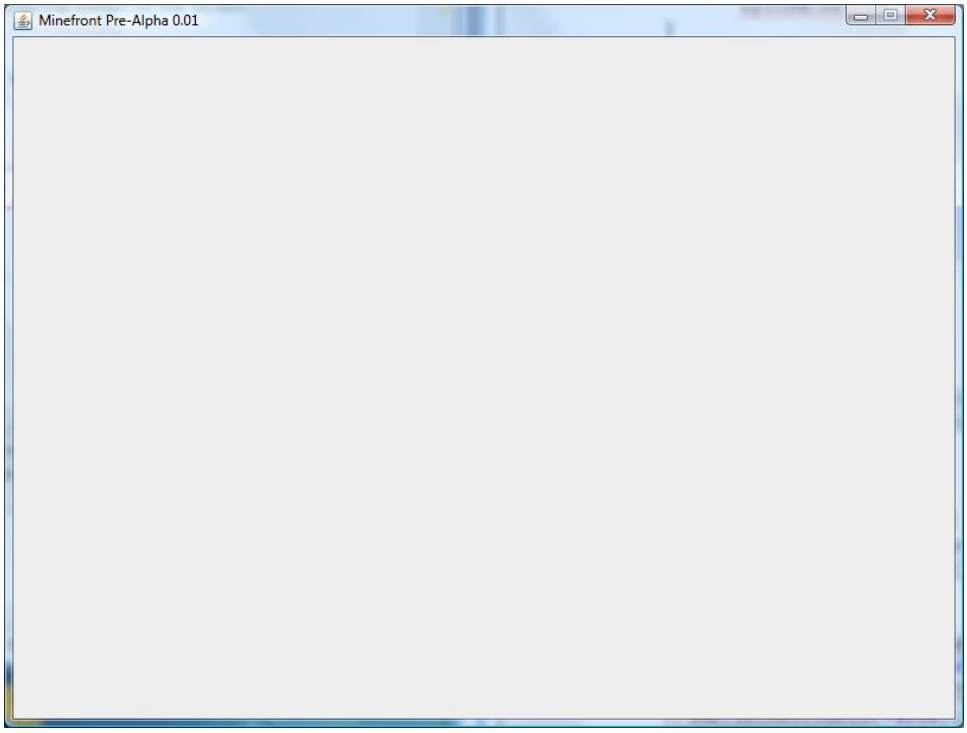

Figure 4 - Episode version of the application

The episode 3 version of `Display.java` introduces the following changes that make no impact to the end result:

- Add a new variable object `render` to the code.
- Added a new constructor to the class that initializes the render object giving it the `WIDTH` and `HEIGHT` of our screen.
- We added two empty methods `tick()` and `render()`
- We added references to these methods to the `run()` method (remember this runs our game loop)

Arrays

The key point of this episode is the use of an array to hold our screen.

What is an array?

An array is a convenient way programmers have developed for accessing many variables using the same name but a different index. Huh? Here is an example. I have a bad memory for names. So when confronted with a new set of students or folks to talk to:

Figure 5 - Classroom of potential programmers

It is easier to give each one a number and refer to a student by the name “Student 1” or “Student 10” to call on a student. Of course, somewhere I let everyone know their number and they remember it (maybe not fondly) and go along with the new system. Did I mention that programmers are generally very lazy and would prefer this method to having to remember “Sam”, “Bob”, “Jane”, or “David”.

A more practical programming example would be involving throwing a die. Suppose I wanted to test a random number generator program in simulating the throw of a die.

In Java there are two means of generating random numbers using the Java libraries.

- The `Random` class generates random integers, doubles, longs and so on, in various ranges.
- The static method `Math.random` generates doubles between 0 (inclusive) and 1 (exclusive)

To generate random integers use the `Random` class to generate random integers between 0 and N (exclusive).

Figure 6 - A red die

I plan on writing a program that throws the die 10000 times and remembers how many times each number of the die [1..6] was thrown by the random number generator. I decide to create six variables to track each die count:

```
private int dieValue1 = 0;
private int dieValue2 = 0;
private int dieValue3 = 0;
private int dieValue4 = 0;
private int dieValue5 = 0;
private int dieValue6 = 0;
```

The plan will be to throw the die, determine the value and figure out which variable should be updated. At the end we will print out the results. If the random number generator is a good one each dieValueX variable should be close to 10000 / 6 or 1666[1]. The problem is that when we do something like this:

```
dieValueThrown = randomNumber(1,6);
```

we have to figure out which dieValueX variable to update. The easiest thing will be to use a switch statement:

```
switch (dieValueThrown) {
      case 1:
            dieValue1++;
            break;
      case 2:
            dieValue2++
            break;
            :
            :
      case 6:
            dieValue6++;
            break;
}
```

The actual program follows:

[1] We should not expect the count to be exactly 1666.

Table 3 - Not ideal version of counting die throws

```
import java.util.Random;

public class ThrowDieExample01 {

	private final int MAX_THROWS = 10000;

	private int dieValue1 = 0;
	private int dieValue2 = 0;
	private int dieValue3 = 0;
	private int dieValue4 = 0;
	private int dieValue5 = 0;
	private int dieValue6 = 0;

	private Random randomNumberGenerator;

	public ThrowDieExample01 () {
		randomNumberGenerator = new Random();
	}

	public int randomNumber(int lowRange, int highRange) {
		return randomNumberGenerator.nextInt(highRange) + lowRange;
	}

	public void run() {
		int dieValueThrown  = 0;
		for ( int i = 0; i < MAX_THROWS; i++) {
			// throw the die
			dieValueThrown = randomNumber(1,6);
			switch (dieValueThrown) {
				case 1:
					dieValue1++;
					break;
				case 2:
					dieValue2++;
					break;
				case 3:
					dieValue3++;
					break;
				case 4:
					dieValue4++;
					break;
				case 5:
					dieValue5++;
					break;
				case 6:
					dieValue6++;
					break;
				default:
					System.out.println("Something is wrong Dexter! The
number was not in [1..6] it was "
							+ dieValueThrown);
			}
```

```
            }
      }

      public void printResults() {
            System.out.println("The results:");
            System.out.println("Times 1 was thrown: " + dieValue1);
            System.out.println("Times 2 was thrown: " + dieValue2);
            System.out.println("Times 3 was thrown: " + dieValue3);
            System.out.println("Times 4 was thrown: " + dieValue4);
            System.out.println("Times 5 was thrown: " + dieValue5);
            System.out.println("Times 6 was thrown: " + dieValue6);
      }

      public static void main(String args[]) {
            ThrowDieExample01 simulation = new ThrowDieExample01();
            simulation.run();
            simulation.printResults();
      }
}
```

The results:

```
The results:
Times 1 was thrown: 1689
Times 2 was thrown: 1558
Times 3 was thrown: 1692
Times 4 was thrown: 1700
Times 5 was thrown: 1709
Times 6 was thrown: 1652
```

Gee, notice the bit of repetitive coding to find the correct variable to increment by one and to print the results. Did I mention programmers are generally lazy? That is not just my opinion. There is a rather popular quote from the inventor of Perl, Larry Wall, “We will encourage you to develop the three great virtues of a programmer: laziness, impatience, and hubris.”

Laziness

> The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it. Hence, the first great virtue of a programmer

So in being good programmers programming language designers invented a smart way to access variables that can hold similar things (in our case an integer value) and use the same name – an array. As you recall when you create a variable you can mentally think of it as creating a container with a special name:

public int myVariable;

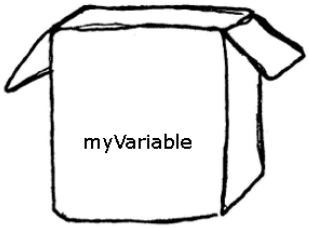

Figure 7 - Container (memory area) holding contents of myVariable

In creating the variable you declare the type of value that will be placed in the container. In a real computer we know that enough memory is set aside to hold the largest possible integer value (given the computer architecture). When we create an array you specify the number of elements it will hold and the type.

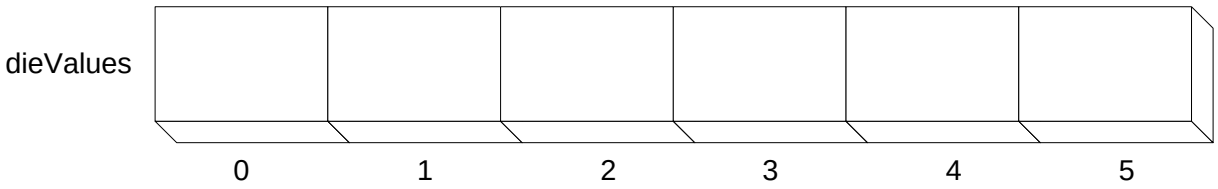

Figure 8 - Using an array

The compiler sets aside in memory the number of elements specified where each one has a unique index. Since Java has similar features to C it adopted the same convention that all arrays start with the index 0. In order to create the above array to hold the size die counter variables we code it in Java as:

```
private int[] dieValues = new int[6];
```

`dieValues` actually represents six variables each with the name `dieValues[0]`, `dieValues[1]`, …, `dieValues[5]`. Since our die value ranges from `[1..6]` we will just map each value to `[0..5]`. The code is now greatly simplified:

Table 4 - An improved way to count the die values thrown

```
import java.util.Random;


public class ThrowDieExample02 {
	private final int MAX_THROWS = 10000;

	private int[] dieValues;

	private Random randomNumberGenerator;

	public ThrowDieExample02 () {
		dieValues = new int[6];    // create an array to hold six elements
```

```
		for (int i = 0; i < 6; i++) {
			dieValues[i] = 0; // initialize all die counts to 0
		}
		randomNumberGenerator = new Random();
	}

	public int randomNumber(int lowRange, int highRange) {
		return randomNumberGenerator.nextInt(highRange) + lowRange;
	}

	public void run() {
		int dieValueThrown  = 0;
		for (int i = 0; i < MAX_THROWS; i++) {
			// throw the die
			dieValueThrown = randomNumber(1,6);
			dieValues[dieValueThrown-1]++;
		}
	}

	public void printResults() {
		System.out.println("The results:");
		for (int i = 0; i < 6; i++) {
			System.out.println("Times " + (i+1) + " was thrown: " +
dieValues[i]);
		}
	}

	public static void main(String args[]) {
		ThrowDieExample02 simulation = new ThrowDieExample02();
		simulation.run();
		simulation.printResults();
	}

}
```

If you notice the code now uses more loops. The code uses a loop to first initialize all the array elements to 0.

```
	for (int i = 0; i < 6; i++) {
		dieValues[i] = 0; // initialize all die counts to 0
	}
```

A loop to print out the results:

```
	for (int i = 0; i < 6; i++) {
		System.out.println("Times " + (i+1) + " was thrown: " + dieValues[i]);
	}
```

And finally it is rather straightforward to find the array slot to update using the die value to determine the correct index. Since the dieValueThrown will be from 1..6 we adjust the value to map into the array index of [0..5] by decrementing by 1. We then increment the current counter value.

```
dieValues[dieValueThrown-1]++;
```

It is easy to appreciate why programmers love arrays (and for loops). You use the same name (pixels or dieValues) and distinguish or specify the exact variable by using an index value.

Many of the objects in many games lend themselves organizing the information in grids (tetris?).

Two-dimensional Array

Many games are easier to solve with a data structure that looks more like a table with rows and columns. We call such as structure a two-dimensional array.

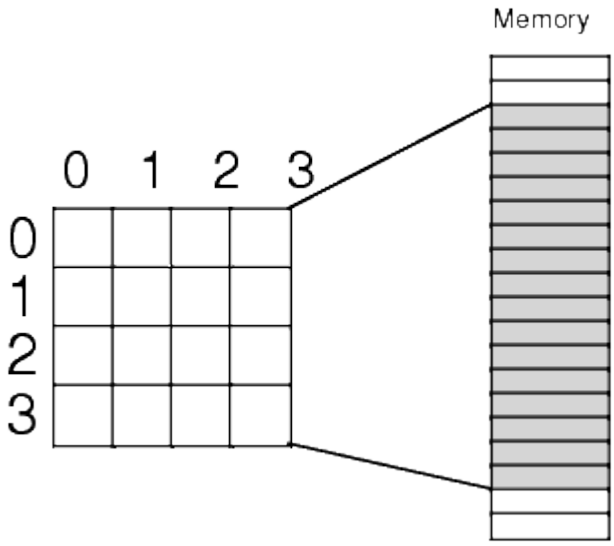

Figure 9 - Two-dimensional array

The above figure illustrates a 4x4 two-dimensional array. It was 4 rows and 4 columns. In reality the two-dimensional array (as shown in memory) is really just our original array where the first row (elements designated as array[0][0] . . . array[0][3] can be stored as the first four elements in the array and elements in the second row array[1][0] . . . array[1][3] are the next four elements.

The code for this episode creates an array:

```
public final int[] pixels;

pixels = new int[width * height];
```

As you can see we are actually storing the screen dimensions – the number of pixel columns (width) by the number of pixel rows (height). Intuitively, we should try to understand that locations

`pixels[0] . . . pixels[width-1]` represent the first row of the screen

Another way to express the pixels in row 0 (the first row) knowing that x is changing from 0..width-1 is:

`pixels[row * width + 0] . . . pixels[row * width + lastX in row]` **or**

`pixels[row * width + 0] . . . pixels[row * width + (width-1)]`

row = y value which is 0.

In our case the first pixel row is represented by pixels[0] . . . pixels[799] (width=800, height=600).

x	y	pixel position
0	0	0
1	0	1
2	0	2
3	0	3
:	:	:
799	0	799

The second row is where y = 1 and x again goes from 0..width-1

`pixels[1 * width] . . . pixels[2 * width -1]` represents the second row

Using the formula

`pixels[row * width + 0] . . . pixels[row * width + (width-1)]`

we can calculate the location of the second row in our pixels[] array:

x	y	pixel position
0	1	y * width +x = 800
1	1	801
2	1	802
3	1	803
:	:	:
799	1	1599

`pixels[(height-1) * width + 0]. . . pixels[(height-1) * width + (width-1)]` represents the last row

x	y	pixel position
0	599	y * width +x = 479200
1	599	479201

2	599	479202
3	599	479203
:	:	:
799	599	479999

The above makes sense we have 800 * 600 pixels = 480000, which will range from 0..479999 in our array!

So what does the above really mean??

If we want to “color” something on the screen in location (0, 5) we will need to write to pixels[y*width+x] or pixels[5*800+0] = pixels[4000].

We will work with this some more in future episodes.