Building a Java First-Person Shooter

Episode 4 – How Rendering Works [Last updated 5/02/2017]

URL

https://www.youtube.com/watch?v=6dOIju7--Jg&index=5&list=PL656DADE0DA25ADBB

Objectives

This is a very short episode that tries to explain how rendering works from the previous episode.

Discussion

There is nothing new in this episode so we will try to explain how the program works at a high level and present all the internal and Java classes utilized in a formal manner and finally the current version of all the class files up to this point in the video series.

Display.java

This class is where the `main()` function resides and key functions of the application resides:

- GUI window gets created
- The drawing area – Canvas is established
- The game loop is created in a thread to basically update and render or draw the current state of the game.

JFrame

This class is used to create a window that has borders, a title and supports other components (e.g. buttons or a Canvas). In addition, it has button components that allow the window to be minimized, opened and closed.

Canvas

The `Canvas` class represents a rectangular area where we can draw on. In addition, it can receive user input such as mouse actions and keyboard entries. This is what we use in the program to draw on. For all intents and purposes we use this object to represent our display area to the user. In the program when we draw we are drawing on this object. We set the size of this object to the size of drawing area – WIDTH * HEIGHT.

BufferedImage

This is our screen! This object is set to the same size as our `Canvas`:

```java
img = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
```

In addition, we describe the color model we will using – that is how to interpret the value we will be writing into the `BufferedImage` object `img`. The object `img` is a rectangular area consisting of pixels whose underlying representation is a one-dimensional array. We obtain the pixel or array representation and draw directly into it!

```
pixels = ((DataBufferInt)img.getRaster().getDataBuffer()).getData();
```

So now pixels can be thought of as the window screen: `pixels[WIDTH][HEIGHT]`

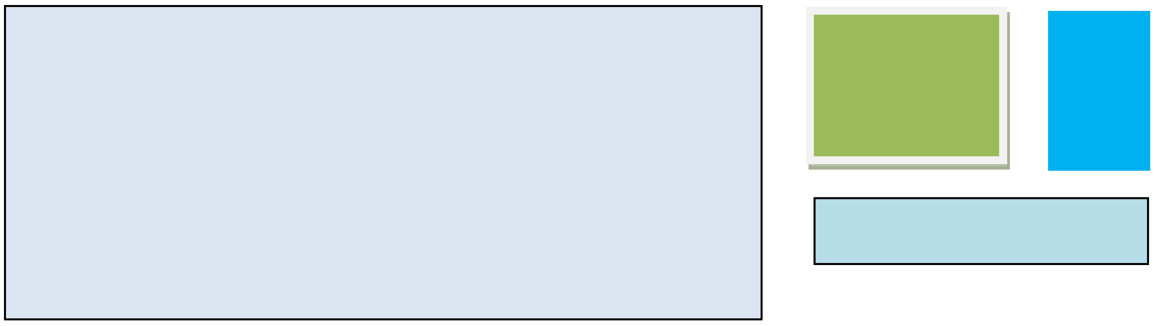
Render

This class represents a *drawable* object. Anything that can be drawn on the screen is defined by a width, height and an array of pixels representing the object.

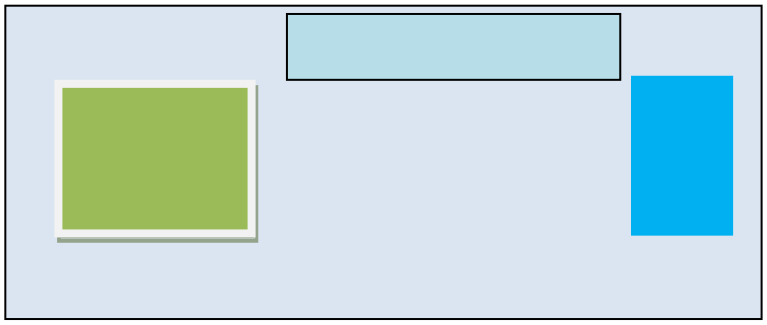
draw(Render render, int xOffset, int yOffset)

The `Render` method draw is used by subclasses to draw a `Render` object into a location within another `Render` object.

A good explanation for this is let's consider we have the following `Render` objects with the pixel contents represented by the colors we see.



All of the above rectangles are either `Render` objects or subclass the `Render` class. Each one has different width and height and pixel content (as shown by the color). Suppose we have each of the smaller (to the right) `Render` objects sent as the first argument of the `draw()` method of the same `Render` (pale blue) render object. Each one will use a different xOffset and yOffset into the (pale blue) object.



The above is an example of how we can use the `draw()` method to write rectangular object into another rectangular object. Ideally the main `Render` object will be an object that represents our screen.

Screen

This is an object representing our screen or the object we want to draw to the screen. For testing purposes we create a smaller `Render` object named `test` to demonstrate how we invoke the draw of `Screen` to "draw" the test object to the screen.

To see this at work change do the following:

Step 1:

Copy the EPISODE_04 → EPISODE_04_5.

Step 2:

Change Screen.java to create and display a red, green and blue rectangle. In addition, we will change the background to white.

Table 1 - Screen.java (episode 04.5)

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

public class Screen extends Render {
	private Render redRectangle;
	private Render greenRectangle;
	private Render blueRectangle;

	public Screen(int width, int height) {
		super(width, height);

		// Clear the screen to white
		for (int i=0; i < width * height; i++) {
			pixels[i] = 0xFFFFFFFF;
		}


		// Create redRectangle
		redRectangle = new Render(100,100);
		for (int i=0; i < 100 * 100; i++) {
			redRectangle.pixels[i] = 0xFFFF0000;
		}
		// Create greenRectangle
		greenRectangle = new Render(25,50);
		for (int i=0; i < 25 * 50; i++) {
			greenRectangle.pixels[i] = 0xFF00FF00;
		}
		// Create blueRectangle
		blueRectangle = new Render(75,10);
		for (int i=0; i < 75 * 10; i++) {
			blueRectangle.pixels[i] = 0xFF0000FF;
		}
	}

	public void render() {
		draw(redRectangle, 0, 0);
		draw(greenRectangle, 200, 200);
		draw(blueRectangle, 500, 10);

	}

}
```

The Screen version above creates three different rectangles and displays them at different locations on the screen.

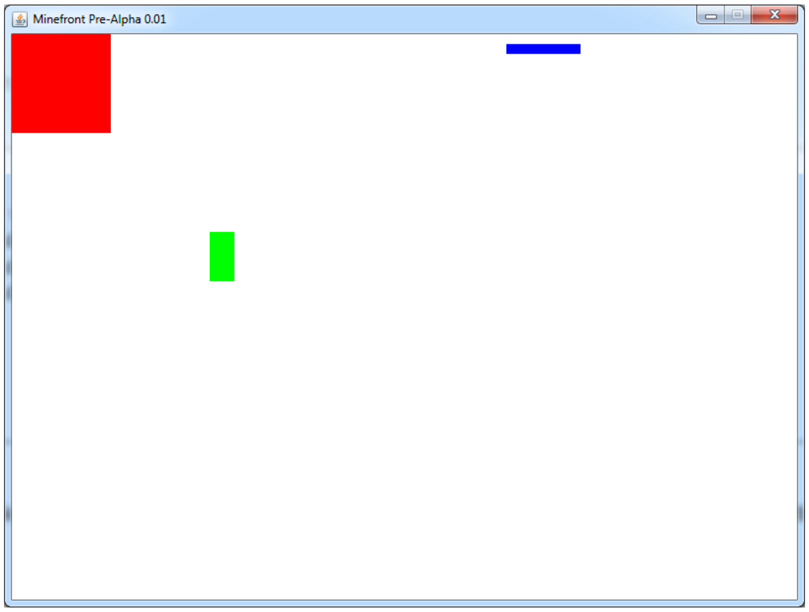


Figure 1 - Drawing render object

How Rendering Works

We will take an object representing our screen (`Screen.java`) and move the contents of our screen into image (`img`) and then `draw()` the `img` onto the `Canvas`.

Screen pixels → img pixels → draw to Canvas

Screen pixels → img pixels

```
		// draw 'screen' to the REAL SCREEN
		for (int i = 0; i < WIDTH * HEIGHT; i++) {
			pixels[i] = screen.pixels[i];
		}
```

img pixels → draw to Canvas

```
g.drawImage(img, 0, 0, WIDTH, HEIGHT, null);
```

Graphics

In order to draw anything to the screen you need to first obtain a `Graphics` object. Since we are using a `BufferStrategy` on the `Canvas` object we obtain the `Graphics` object using the `BufferStrategy` object `bs`.

```
Graphics g = bs.getDrawGraphics();
```

We render the screen by finally drawing the `img` object to the `Canvas`.

Episode Code

`Render.java` used to represent a "drawable" rectangular object. It provides a `draw()` method to more or draw the pixels of one `Render` object into another.

Table 2 - 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) {
		// iterate through each row 0..height-1
		for ( int y = 0; y < render.height; y++) {
			int yPix = y + yOffset;
			// iterate through each column x goes from 0..width-1
			for ( int x = 0; x < render.width; x++) {
				int xPix = x + xOffset;
				pixels[xPix + yPix * width] =
						render.pixels[x + y * render.width];
			}
		}
	}

}
```

The class `Screen` holds our game screen. It will just place one element – a `test` object to the screen. The `test` object is filled with random colors.

Table 3 - 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() {
		draw(test, 0, 0);
	}

}
```

The Display class is the main application class.

Table 4 - Display.java

```
package com.mime.minefront;

import java.awt.Canvas;
import java.awt.Graphics;
import java.awt.image.BufferStrategy;
import java.awt.image.BufferedImage;
import java.awt.image.DataBufferInt;

import javax.swing.JFrame;

import com.mime.minefront.graphics.Screen;

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 Screen screen;
	private BufferedImage img;
	private int[] pixels;

	public Display() {
		screen = new Screen(WIDTH, HEIGHT);
		img = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
		// Get the pixel array of the ACTUAL SCREEN
		pixels = ((DataBufferInt)img.getRaster().getDataBuffer()).getData();
	}
	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() {
		BufferStrategy bs = this.getBufferStrategy();
		if (bs == null) {
			this.createBufferStrategy(3);
			return;
		}

		screen.render();
		// draw 'screen' to the REAL SCREEN
		for (int i = 0; i < WIDTH * HEIGHT; i++) {
			pixels[i] = screen.pixels[i];
		}
		Graphics g = bs.getDrawGraphics();
		g.drawImage(img, 0, 0, WIDTH, HEIGHT, null);
		g.dispose();
		bs.show();
	}


	@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();
	}


}
```