Building a Java First-Person Shooter

Episode 4 – Drawing Pixels [Last updated 4/30/2017]

URL

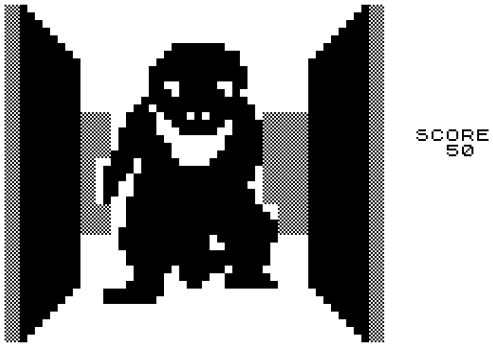
https://www.youtube.com/watch?v=bKbZECPjQs0

Objectives:

This episode[1] makes use of several Java classes that support drawing to the screen. But before we get into the actual code for this episode I would like to present and discuss the basics behind drawing and color. In addition, I would like to describe the techniques that this and future versions of the game shall be using to display game frames in a smooth manner. The topic of frame rates does not come up yet but you should realize that the screen is being constantly updated (right now the same thing is being shown) every time in the game loop.

From Wikipedia: **Everything you wanted to know about frame rates but were afraid to ask!**

A common term that is mentioned with respect to games is *frame rate*. This is the frequency in which the image on the computer screen is being updated. The frame rate is usually expressed as frames per second (fps).



Frame rates are considered important in video games. The frame rate can make the difference between a game that is playable and one that is not. The first 3D first-person adventure game for a personal computer, 3D Monster Maze, had a frame rate of approximately 6 fps, and was still a success, being playable and addictive. In modern action-oriented games where players must visually track animated objects and react quickly, frame rates of between 30 to 60 fps are considered minimally acceptable by some, though this can vary significantly from game to game. Most modern action games, including popular first person shooters such as Halo 3, run around 30 frames a second, while others, such as Call of Duty 4, run at 60 frames a second. The frame rate within most games, particularly PC games, will depend upon what is currently happening in the game.

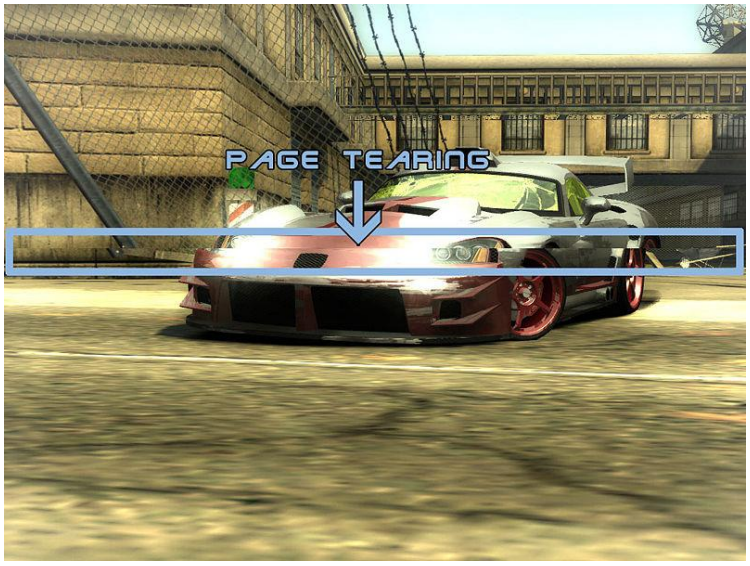
QUESTION: If we are moving the ball every 60 milliseconds on the screen, what is the perceived frame rate?

[1] This episode was very disappointing in how code was tested (has anyone heard of test-driven development?) The lack of explanations for key graphics functions involving images puzzles me. I am continuing this project but very disappointed in the lack of convention and good programming techniques.

A culture of competition has arisen among game enthusiasts with regards to frame rates, with players striving to obtain the highest fps count possible. Indeed, many benchmarks released by the marketing departments of hardware manufacturers and published in hardware reviews focus on the fps measurement. Modern video cards, often featuring NVIDIA or ATI chipsets, can perform at over 160 fps on graphics intensive games such as F.E.A.R. One single GeForce 8800 GTX has been reported to play F.E.A.R. at up to 386 fps (at a low resolution). This does not apply to all games: some games apply a limit on the frame rate. For example, in the *Grand Theft Auto* series, *Grand Theft Auto III* and *Grand Theft Auto: Vice City* have a standard 30 fps (*Grand Theft Auto: San Andreas* runs at 25 fps) and this limit can only be removed at the cost of graphical and gameplay stability. It is also doubtful whether striving for such high frame rates are worthwhile. An average 17" monitor can reach 85 Hz, meaning that any performance reached by the game over 85 fps is discarded. For that reason it is not uncommon to limit the frame rate to the refresh rate of the monitor in a process called vertical synchronization. However, many players feel that not synchronizing every frame produces sufficiently better game execution to justify some "tearing" of the images.

It should also be noted that there is a rather large controversy over what is known as the "feel" of the game frame rate. It is argued that games with extremely high frame rates "feel" better and smoother than those that are just getting by. This is especially true in games such as a first-person shooter. There is often a noticeable choppiness perceived in most computer rendered video, despite it being above the flicker fusion frequency (as, after all, one's eyes are not synchronized to the monitor).

This choppiness is not a perceived flicker, but a perceived gap between the object in motion and its afterimage left in the eye from the last frame. A computer samples one point in time, then nothing is sampled until the next frame is rendered, so a visible gap can be seen between the moving object and its afterimage in the eye. Many driving games have this problem, like *NASCAR 2005: Chase for the Cup* for Xbox, and *Gran Turismo 4*. The polygon count in a frame may be too much to keep the game running smoothly for a second. The processing power needs to go to the polygon count and usually takes away the power from the framerate.



The reason computer rendered video has a noticeable afterimage separation problem and camera captured video does not is that a camera shutter interrupts the light two or three times for every film frame, thus exposing the film to 2 or 3 samples at different points in time. The light can also enter for the entire time the shutter is open, thus exposing the film to a continuous sample over this time. These multiple samples are naturally interpolated together on the same frame. This leads to a small amount of motion blur between one frame and the next which allows them to smoothly transition.

An example of afterimage separation can be seen when taking a quick 180 degree turn in a game in only 1 second. A still object in the game would render 60 times evenly on that 180 degree arc (at 60 Hz frame rate), and visibly this would separate the object and its afterimage by 3 degrees. A small object and its afterimage 3 degrees apart are quite noticeably separated on screen.

The solution to this problem would be to interpolate the extra frames together in the back-buffer (field multisampling), or simulate the motion blur seen by the human eye in the rendering engine. When

vertical sync is enabled, video cards only output a maximum frame rate equal to the refresh rate of the monitor. All extra frames are dropped. When vertical sync is disabled, the video card is free to render frames as fast as it can, but the display of those rendered frames is still limited to the refresh rate of the monitor. For example, a card may render a game at 100 FPS on a monitor running 75 Hz refresh, but no more than 75 FPS can actually be displayed on screen.
Certain elements of a game may be more GPU-intensive than others. While a game may achieve a fairly consistent 60 fps, the frame rate may drop below that during intensive scenes. By achieving frame rates in excess of what is displayable, it makes it less likely that frame rates will drop below what is displayable during stressful scenes.

A future episode will have discussions about frame rates since it strongly correlates to the game loop. I will also note that having high frame rates does not always make sense since the ultimate bottleneck is how frequently the monitor updates the physical screen.[2] But since this week's episode presents Java 2D classes that handle screen update performance I thought it would be interesting to present a topic that is always at the forefront of many game developers mind – having the game run smoothly.

By the time you reach this episode you may be confused by the nomenclature being used. There is `Display`, `Render`, and `Screen` classes all which could mean the same thing. The `Render` has a `draw()` method but the actual drawing to the screen takes place in the `Display` class. The `Display` and `Screen` classes both have a `render()` methods where one method appears to draw to the screen and the other merely move pixels by invoking the underlying `Render's` class `draw()` method. If you are doing a bit of head scratching here you are not alone.

The game will actually run on your monitor or display. So let's think of the `Display` class as being responsible for whatever gets displayed on the screen. Any decent game will have more than one screen where drawing takes place. One screen will hold or contain what is currently being displayed and the other screen is called a back buffer and holds the next screen. This technique is called double buffering.

[2] http://en.wikipedia.org/wiki/Refresh_rate

Double Buffering

FROM: http://docs.oracle.com/javase/tutorial/extra/fullscreen/doublebuf.html

Double buffering is a technique for drawing graphics that show no flicker or tearing on the screen. The way it works is not to update the screen directly but to create an updated version of the screen in another area (a buffer) and when you have finished moving the aliens, killing or removing the debris and moving the player you then move or copy the updated screen to the video screen in one step or as quickly as possible when the video monitor is moving to re-set to draw a new screen.

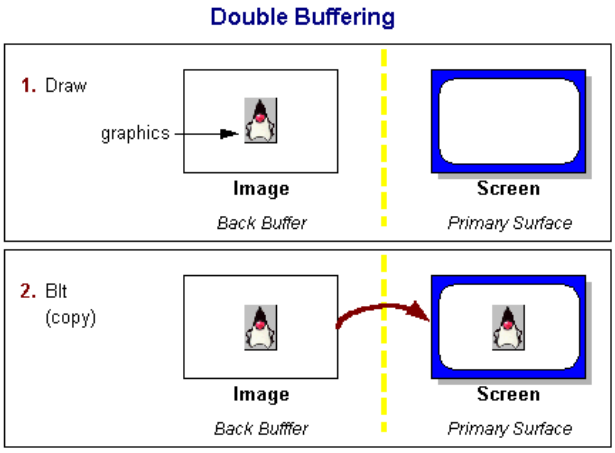


Figure 1 - Double buffering

In double buffering we reserve an area in memory (RAM) that we update and then copy or what most programmers refer to as blit (bit blit or bit block) the entire memory area into the video area. The screen surface is commonly referred to as the *primary surface*, and the offscreen image used for double-buffering is commonly referred to as the *back buffer*. You can actually (and often) use more than one back buffer.

Page Flipping

Another technique used is page flipping. This takes advantage of the fact that many graphics cards have what is called a video pointer, which is simply an address in video memory. This pointer tells the graphics card where to look for the contents of the video to be displayed during the refresh cycle.

In some graphics cards and on some operating systems, this pointer can even be manipulated programmatically. Suppose you created a back buffer (in video memory) of the exact width, height, and bit depth of the screen, then drew to that buffer the same way as you would using double-buffering. Now imagine what would happen if, instead of blitting your image to the screen as in double-buffering, you simply changed the video pointer to your back buffer. During the next refresh, the graphics card would now use your image to display. This switch is called page-flipping, and the performance gain over blt-based double-buffering is that only a single pointer needs to be moved in memory as opposed to copying the entire contents from one buffer to another.

When a page flip occurs, the pointer to the old back buffer now points to the primary surface and the pointer to the old primary surface now points to the back buffer memory. This sets you up automatically for the next draw operation.

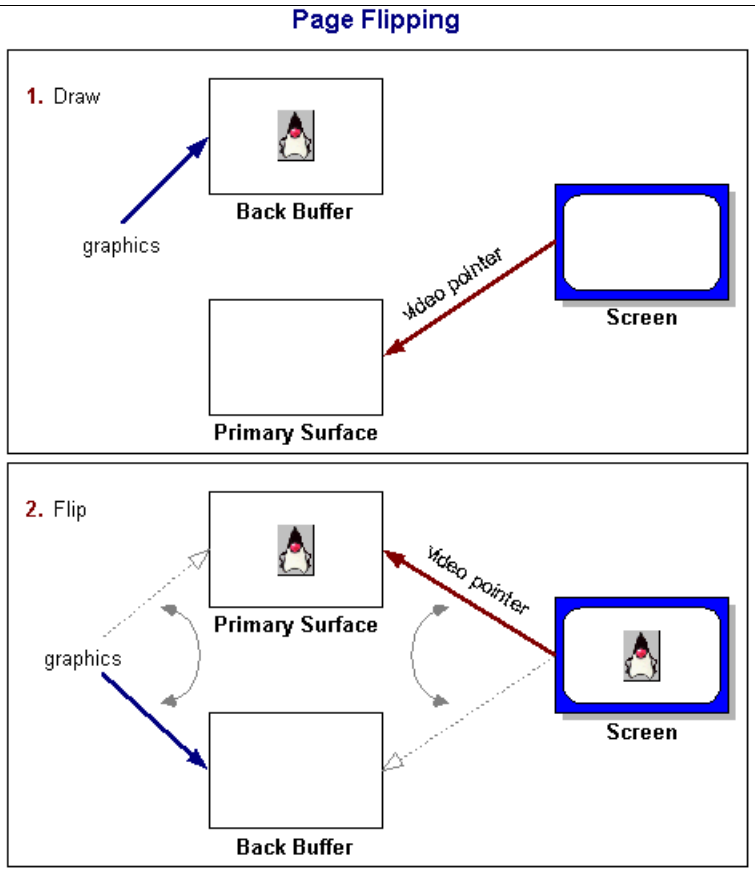


Figure 2 - Page Flipping

Sometimes it is advantageous to set up multiple back buffers in a *flip chain*. This is particularly useful when the amount of time spent drawing is greater than the monitor's refresh rate. A flip chain is simply two or more back buffers (sometimes called *intermediary buffers*) plus the primary surface (this is sometimes called triple-buffering, quadruple-buffering, etc.). In a flip chain, the next available back buffer becomes the primary surface, etc., all the way down to the rearmost back buffer that is used for drawing.

Benefits of Double-Buffering and Page-Flipping

If your performance metric is simply the speed at which double-buffering or page-flipping occurs versus direct rendering, you may be disappointed. You may find that your numbers for direct rendering far exceed those for double-buffering and that those numbers far exceed those for page-flipping. Each of these techniques is for used for improving perceived performance, which is much more important in graphical applications than numerical performance.

Double-buffering is used primarily to eliminate visible draws which can make an application look amateurish, sluggish, or appear to flicker. Page-flipping is used primarily to also eliminate *tearing*, a splitting effect that occurs when drawing to the screen happens faster than the monitor's refresh rate. Smoother drawing means better perceived performance and a much better user experience.

So what is going on is that while the user is viewing the current buffer on the actual display the game is creating and building the next frame to display in the back buffer. This technique makes elements on the screen appear smooth as aliens are bombed out of the sky or cars are making hairpin turns on some track.

In this episode we make use of Java provided libraries to setup and manage the double-buffering or page flipping.

Graphics Coordinate System

The program uses the `Graphics` object and the methods associated with it to draw on the screen. In order to understand how drawing works we first will need to understand how the graphical coordinate system works.

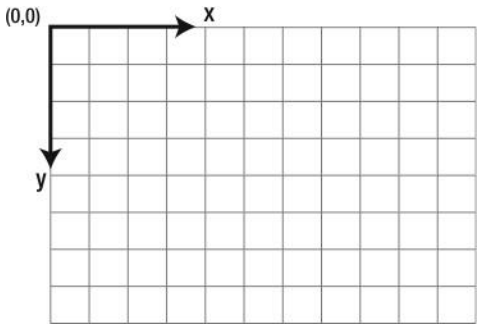


Figure 3 - 2D coordinate system

The top-left corner is at (0,0), as we move right along the x-axis the value of x increases as we move down along the y-axis the value of y increases. If the screen resolution is set to WIDTH x HEIGHT (in pixels) then the bottom-right position is at (WIDTH-1, HEIGHT-1).

The code in this episode that actually draws to the screen is:

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

The above draws the contents of img starting at the (0,0) location on the screen. The WIDTH and HEIGHT specifies how much of the image to draw. If the WIDTH and HEIGHT do not match the size of the image then the image is scaled (stretched or shrunk) to fit the specified area. When CHERNO changed the code to:

```
g.drawImage(img, 0, 0, WIDTH * 20, HEIGHT * 20, null);
```

We saw how each pixel was repeated 20 times across and down the screen in order to fit the target rectangle specified. Of course since the `img` is only 800 x 600 that left most of our game screen image off to the right and bottom (see Figure 13).

Color and RGB

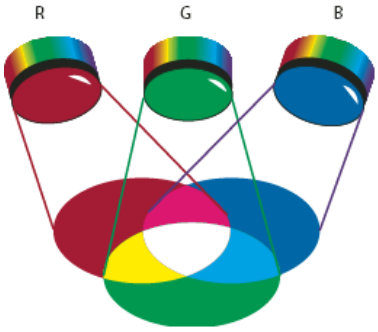


Figure 4 - Red, Green and Blue to make any color in spectrum

There are various ways to compose or specify the color you want to draw in. You can set the color of the background or the pen that is used to draw in by using the Graphics method setColor(Color c) method. The Color class has many static members you can use to specify a color.

Table 1 - Color static variables

Color.black	Color.lightGray	Color.blue
Color.magenta	Color.cyan	Color.orange
Color.darkGray	Color.pink	Color.gray
Color.red	Color.green	Color.white
Color.yellow		

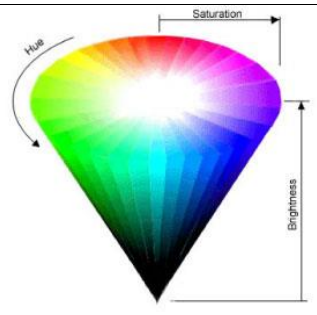
You can also create your own color using RGB(Red, Green, Blue) color model when creating a new Color object.

```
new Color(red, green, blue)
```

where each color parameter is an integer between 0 and 255 to indicate the amount of red, green and blue, respectively.

You can also mix your own colors using the HSB color model.

Table 2 - HSB: Hue, Saturation, and Brightness

	From: http://www.tomjewett.com/colors/hsb.html ***HSB: Hue, Saturation, and Brightness*** This scheme provides a device-independent way to describe color. HSB may be the most complex scheme to visualize, especially since color selection software has to reduce its three descriptive dimensions to two dimensions on the monitor screen. But once learned, it can be useful in many instances.

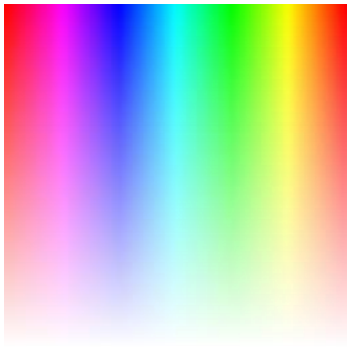
The easiest way to visualize this scheme is to think of the H, S, and B values representing points within an upside-down cone. At the edge of the cone base, think of the visible light spectrum, cut from the page and pasted into a circle with shading added to smooth the transition between the (now joined) red and magenta ends.

- **Hue** is the actual color. It is measured in angular degrees counter-clockwise around the cone starting and ending at red = 0 or 360 (so yellow = 60, green = 120, etc.)
- **Saturation** is the purity of the color, measured in percent from the center of the cone (0) to the surface (100). At 0% saturation, hue is meaningless.
- **Brightness** is measured in percent from black (0) to white (100). At 0% brightness, both hue and saturation are meaningless.

Pixels and Color

Every pixel has a color. We can regard the color as being defined by four components – Red, Green, Blue and Alpha values.

Table 3 - What is Alpha?



What is Alpha?

From: http://en.wikipedia.org/wiki/RGBA_color_space
RGBA stands for red green blue alpha. While it is sometimes described as a color space, it is actually simply a use of the RGB color model, with extra information. The color is RGB, and may belong to any RGB color space, but the integral alpha as invented by Catmull and Smith between 1971 and 1972 enables alpha compositing. The inventors named alpha after the Greek letter in the classic linear interpolation formula $\alpha A + (1-\alpha)B$.
The alpha channel is normally used as an opacity channel. If a pixel has a value of 0% in its alpha channel, it is fully transparent (invisible), whereas a value of 100% in the alpha channel gives a fully opaque pixel (traditional digital images). Values between 0% and 100% make it possible for pixels to show through a background like a glass (translucency), an effect not possible with simple binary (transparent or opaque) transparency. It allows easy image compositing.

Here is an example of a before and after picture of a game where in the second screen you can see how the alpha value for the “pink slime” was changed so that the background shows through more.

Figure 5 - Pink slime completely opaque (alpha = 100%)

Figure 6 - "Pink slime" ghosting out (alpha = 20%)

In this episode we will populate a 256 x 256 portion of the screen with “colored” pixels. The way the code does the generation of random colors is by using the following technique:

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

The above creates a 256 x 256 rectangular area and populates each pixel in that space with a random integer. The current episode’s code will show the following image:

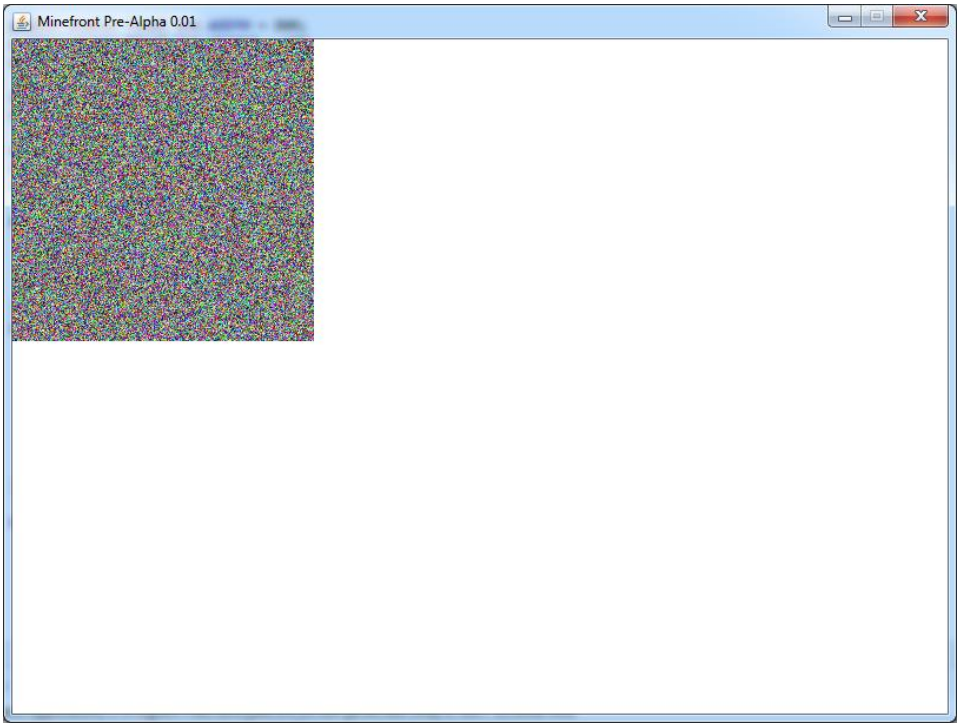


Figure 7 – Minefront for Episode 4

You may be wondering how a random integer can specify a color. In computer graphics[3], pixels encoding RGBA Color Space information may be stored in computer memory (or in files on disk), in well-defined formats. In the most common format the intensity of each channel sample is defined by 8 bits (which holds a value from 0-255), and are arranged in memory in such a manner that a single 32-bit unsigned integer has the Alpha sample in the highest 8 bits, followed by the Red sample, Green sample, and the Blue sample in the lowest 8 bits. This is often called “ARGB”.

[3] From: http://en.wikipedia.org/wiki/RGBA_color_space

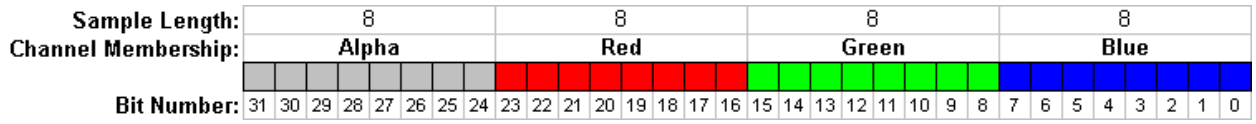


Figure 8 - Integer representation of a "color" value

So knowing the above we can see how storing an integer value into a pixel location translates into a color value. A more natural way to have done this without knowledge of the underlying representation of a Color object was to have defined the pixel array as:

```
public final Color[] pixels;
```

and populating each pixel position with a random color using:

```
test.pixels[i] = new Color(random.nextInt(256),random.nextInt(256),
                                    random.nextInt(256), random.nextInt(256));
```

where each argument provides the Red, Green, Blue and Alpha component respectively. There are ways to extract the actual Red, Blue, Green and Alpha values by "anding" (and shifting) the integer value with the appropriate hex string.

Another question that may have occurred to you is why most of the screen background color black for you and white for me. When you create a `Render` object (or `Screen` object in this case) the `pixel[]` array is pre-populated with the value 0, which naturally maps on to the color black. I changed my Render object to automatically make the background for any Render object created to white by adding the following code to the constructor:

Table 4 - 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];

		for (int y=0; y < height; y++) {
			for (int x=0; x < width; x++) {
				pixels[y * width + x] = 0xFFFFFFFF; // white color
			}
		}
	}
```

```
        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 reason I decided to do this is to save on toner cartridge!

Images

From: http://docs.oracle.com/javase/tutorial/2d/overview/images.html

In the Java 2D API an image is typically a rectangular two-dimensional array of pixels, where each pixel represents the color at that position of the image and where the dimensions represent the horizontal extent (width) and vertical extent (height) of the image as it is displayed.

The most important image class for representing such images is the `java.awt.image.BufferedImage` class. The Java 2D API stores the contents of such images in memory so that they can be directly accessed.

Applications can directly create a `BufferedImage` object or obtain an image from an external format such as PNG or GIF. In either case, the application can then draw onto the image by using Java 2D API graphics class. So, images are not limited to displaying photographic type images. Different objects such as line art, text, and other graphics and even other images can be drawn onto an image (as shown on the images).

Figure 9 - Image with other images drawn on

There are a number of common tasks when working with images:

- Loading an external GIF, PNG, and JPEG image format files into Java 2D internal image representation.
- Directly creating a Java 2D image and rendering to it
- Drawing the contents of a Java 2D image on to a drawing surface
- Saving the contents of a Java 2D image to an external GIF, PNG, or JPEG image file

There are two main classes that you must learn about to work with images:

- The `java.awt.Image` class is the superclass that represents graphical images as rectangular arrays of pixels
- The `java.awt.image.BufferedImage` class, which extends the Image class to allow the application to operate directly with image data (for example, retrieving or setting up the pixel color). Applications can directly construct instances of this class.

The `BufferedImage` class is the cornerstone of the Java 2D immediate-mode imaging API. It manages the image in memory and provides methods for storing, interpreting, and obtaining pixel data. A `BufferedImage` has a `ColorModel` and a `Raster` of image data. The `ColorModel` provides a color interpretation of the image's pixel data.

The Raster performs the following functions:

- Represents the rectangular coordinates of the image
- Maintains image data in memory
- Provides a mechanism for creating multiple subimages for a single image data buffer
- Provides methods for accessing specific pixels within the image.

From: http://docs.oracle.com/javase/6/docs/technotes/guides/2d/spec/j2d-image.html

The immediate mode imaging mode supports fixed-resolution images stored in memory. The model also supports filtering operations on image data. A number of classes and interfaces are used in this model.

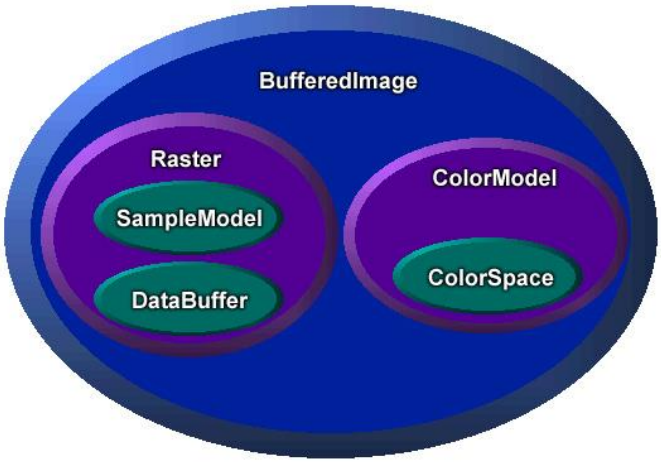


Figure 10 - BufferedImage and supporting classes

The `Raster` class provides image data management. It represents the rectangular coordinates of the image, maintains image data in memory, and provides a mechanism for creating multiple subimages from a single image data buffer. It also provides methods for accessing specific pixels within an image. A `Raster` object contains two other objects, a `DataBuffer` and a `SampleModel`. The `DataBuffer` class holds pixel data in memory. The `SampleModel` class interprets data in the buffer and provides it as individual pixels or rectangular ranges of pixels. The `ColorModel` class provides a color interpretation of pixel data provided by the image's sample model.

The constructor of BufferedImage used in this episode is:

BufferedImage

```
public BufferedImage(int width,
                     int height,
                     int imageType)
```

Constructs a `BufferedImage` of one of the predefined image types. The `ColorSpace` for the image is the default sRGB space.

Parameters:

`width` - width of the created image

`height` - height of the created image

`imageType` - type of the created image

For this episode the `imageType` is set to `TYPE_INT_RGB` which means that the image is 8-bit RGB color components (the alpha value is ignored) packed into integer pixels.

In this episode we will change the main class `Display` to use the new class `Screen` rather than `Render`. But first let's discuss the changes to the `Render` class.

Render.java

The purpose of this class is to hold a rectangular set of pixels that could represent the screen but more than likely will hold any object that we which to render to the screen as long as it can be represented as a rectangular object (as our little Mario image).

The only thing new about this class is the addition of a new method `draw()`. It may appear strange that the `draw()` method inputs three arguments a `Render` object and an `xOffset` and `yOffset`. But it is used to move an object (e.g. Mario) represented in the pixels of the `Render` object into another `Render` object. I like to regard the `Render` object's who `draw()` method is being called as the "target" `Render` object and the `render` object being supplied as the first argument as the "source" render object. (see code for `draw()` below).

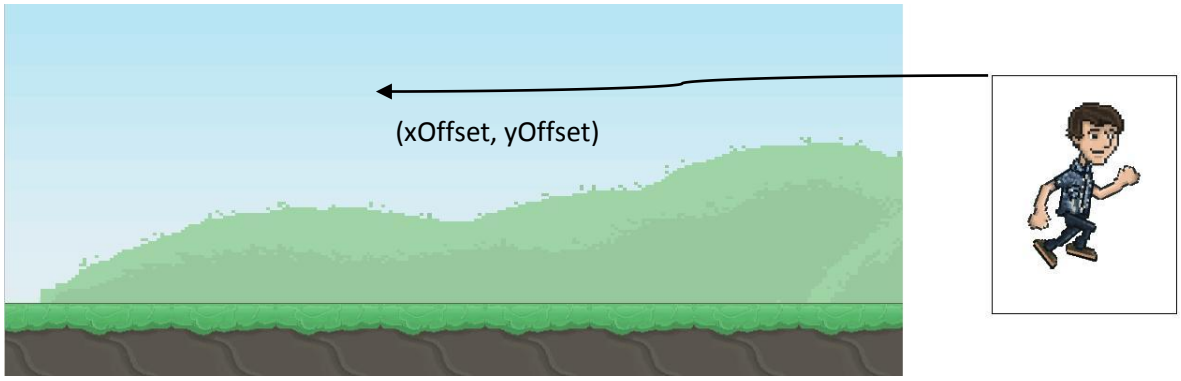


Figure 11 - Drawing our man "Fred" into the landscape (before)

In the example illustrated above the landscape `Render` object is the "target" and my man Fred the "source" `render` object. We are going to copy Fred `render` into the landscape by 1) providing the Fred `Render` object 2) specifying the `xOffset` into the target 3) specifying the `yOffset` into the target landscape `Render` object.

Figure 12 - The result of draw() "Fred" Render into landscape Render object (after)

Let's look at the code for the `Render` class.

```
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 `for` loops scan the entire source image (e.g. "Fred") row by row. We calculate the pixel offset in the target `Render` object (the landscape) and then place the corresponding pixel in our source. Since our pixel data structure is one-dimensional we need to use the formula X + Y * WIDTH to map to the correct rectangular location (see previous episode for explanation of the mapping formula).

I don't know if rather than the name `draw()` I would have named the above `moveRenderIntoRender` or just `move`. There is no "real" drawing going on as in drawing an image to the screen display.

Screen.java

A new class is added in this episode the `Screen.java` class. It will be used to hold the pixel contents representing our screen. The representation of this class for this episode contains lines of code that we will remove in subsequent episodes. I would have preferred a test-case (using jUnit) to test out the `Screen` class rather than injecting throwaway code but that was not done here. I will illustrate two version of this class the first is basic and what will be built on in subsequent episodes, the second contains logic to illustrate the use of the `Render.draw()` method.

Table 5 - Basic Screen.java class

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


public class Screen extends Render {


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

	public void render() {
	}
}
```

As you can see the `Screen` class extends the `Render` class and will provide additional functions related to the actual screen display.

The test will be to create a `Render` object to represent a 256 x 256 pixel object. The test Render object will be populated with random colors and moved/copied |`draw()` into our `Screen` render object at the top left-hand location.

Table 6 - Screen.java class with test Render object

```
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 above effectively puts a randomly colored 256 x 256 rectangle on the top-left hand side of the screen when the `render()` method from the `Screen` class is invoked.

So for now when you create a `Screen` object and invoke the `render()` method you will get the above (minus the background colors being used). The test `Render` object will have each pixels made of some randomly selected colors (see Figure 4).

The main `Display` class will swap out the use of `Render` for `Screen` class and create its own pixel array that will hold the "actual" data used to represent the information that will be sent to the device (monitor, tablet, or cell phone).

The entire new `Display.java`:

Table 7 - Display.java for episode 4

```
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);
		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();

		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();
        }


}
```

You will note the introduction of three new variables:

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

The screen object will hold a representation of our display screen or game. The `BufferedImage img` is used to hold and build the screen image to be sent or drawn to the monitor display and the `pixels[]` array will contain the actual pixel data associated with the BufferedImage object.

The `Display` class constructor:

```
        public Display() {
                screen = new Screen(WIDTH, HEIGHT);
                img = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
                pixels = ((DataBufferInt)img.getRaster().getDataBuffer()).getData();
        }
```

creates the initial `Screen` object screen. The `Screen` object has the same dimensions as our window screen. We will be essentially taking the contents of the "screen" and drawing to the window via the `img` object The `img` is initialized to the WIDTH and HEIGHT specified (800 x 600) and declared to hold an integer representing the Red, Green and Blue color components. The initialization of the `pixels` variable is just a long-winded way to get that data structure to point to the `img` data area holding the display image. The `getRaster()` method returns a `WritableRaster` object. The `WritableRaster` class extends `Raster` to provide pixel writing capabilities. The `getDataBuffer()` method returns the `DataBuffer` associated with the Raster. The `DataBufferInt` is used to extend `DataBuffer` and stores data internally as integers. Finally the `getData()` method returns the default (first) `int` data array in `DataBuffer`. In summary the line:

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

holds the representation of our "real" screen image data area. When we write into the `pixels` array we are effectively writing to our `BufferedImage` object `img`. Which in turn is used to draw on the screen.

The only other major change to our class is the filling in of the `render()` method. The code:

```
BufferStrategy bs = this.getBufferStrategy();
```

From: http://docs.oracle.com/javase/tutorial/extra/fullscreen/bufferstrategy.html

In Java 2 Standard Edition, you don't have to worry about video pointers or video memory in order to take full advantage of either double-buffering or page-flipping (see episode notes above). The new class `java.awt.image.BufferStrategy` has been added for the convenience of dealing with drawing to surfaces and components in a general way, regardless of the number of buffers used or the technique used to display them.

A buffer strategy give you two all-purpose method for drawing: `getDrawGraphics` and `show`.

From: http://content.gpwiki.org/index.php/Java:Tutorials:Double_Buffering

I mentioned before that when using double buffering, you draw to an offscreen image. While the *BufferStrategy* isn't an *Image* itself, it does own one. Even better, it presents itself as if it were an *Image* to make things very simple. This means that like an *Image*, you can get a 'Graphics *object from the* BufferStrategy*. Once you have a* Graphics *object, all of the standard drawing methods are available to you. Often, it is easiest to pass the* Graphics *object off to your own objects for them to draw on.*

Once you have everything drawn to the offscreen image, it is then time to display it. There are two ways this can be done:

- Draw from the offscreen image to the screen. (Called blitting.)
- Tell the program to use the offscreen image as the image for the screen, and vice versa. (Called flipping.)

The second method, flipping, is much faster, as it involves changing a pointer rather than drawing the entire screen. Normally, Java will take care of this for you, but it is possible that flipping is unsupported on some hardware, so it useful to know.

To display your backbuffer, use the *BufferStrategy*.show() method.

The render method:

```
private void render() {
	BufferStrategy bs = this.getBufferStrategy();
	if (bs == null) {
		this.createBufferStrategy(3);
		return;
	}

	screen.render();
```

```
            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();
      }
```

The first time `render()` is called `bs` the `BufferStrategy` will be null so it will call `createBufferStrategy(3)` method. This is one of the Canvas methods. The format is:

```
public void createBufferStrategy(int numBuffers)
```

The method creates a new strategy for multi-buffering on this component. Multi-buffering is useful for rendering performance. This method attempts to create the best strategy available with the number of buffers supplied. It will always create a BufferStrategy with that number of buffers. A page-flipping strategy is attempted first, then a blitting strategy using accelerated buffers. Finally, an unaccelerated blitting strategy is used.

The code then invokes the `screen.render()` to get the wild random rectangle and then copies the entire screen into the pixel area. We then use the `getDrawGraphics()` method to obtain a graphics object to use for drawing to the screen. The `drawImage` method draws our screen image to the display screen and shows the results.

In summary, this episode illustrates how the buffering will be handled and how we will write our pixels representing various objects to the screen.

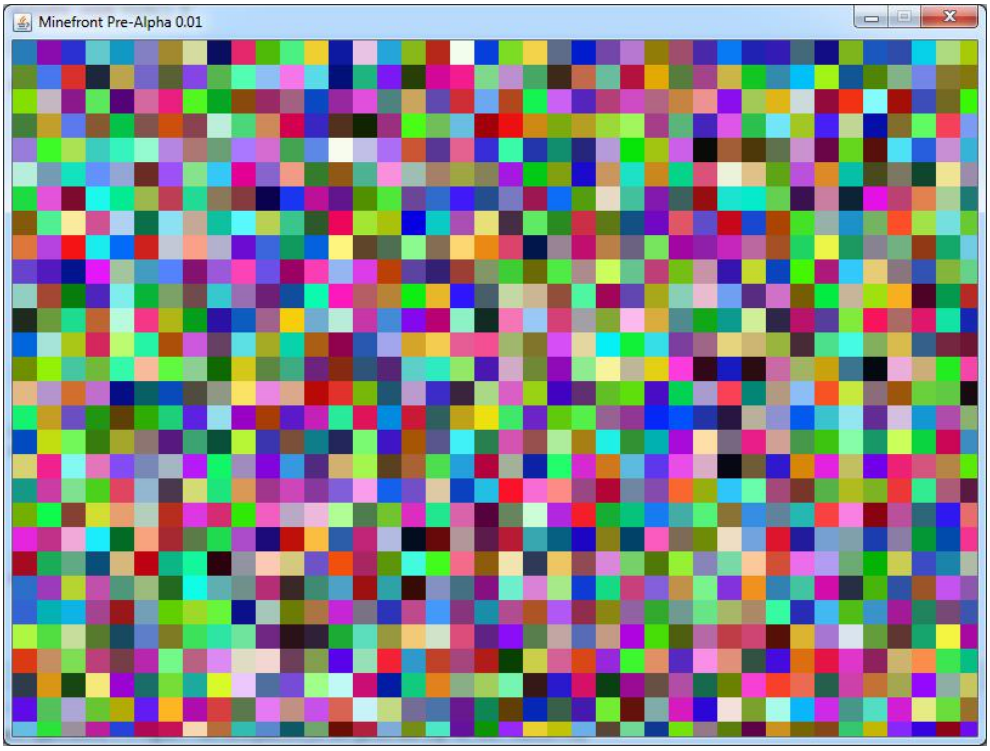


Figure 13 - Repeating each pixel 20 times across and down