Building a Java First-Person Shooter

Episode 7 – FPS Counter [Last Update 5/6/2017]

Objectives

In this episode we determine how fast our screen gets updated in the game loop. The simple way to get this done is to imagine you start a stopwatch and count how many times you paint or render the screen in one second. You stop the watch as soon as one second goes by and report your findings. You immediately start counting again.

URL

https://www.youtube.com/watch?v=piCiSPCaRG0

Discussion

In video episode 3 we added two empty methods `tick()` and `render()`. The methods represent to key part to our *game loop*. The `tick()` is where we will place the game logic that updates the game objects – e.g. moves the enemies, updates the player position, etc. In many game engines it is called `update()`. The goal is to update the game elements at a certain frame rate – usually 60 frames/second. The `render()` method is responsible for displaying the drawing the current game state to the screen. The `render()` can and should be done as fast as the machine allows.

The key Java class used to measure time is the `System.nanoTime()`. Using this call instead of `System.currentTimeMillis()` actually uses more CPU time. In addition, some systems like Windows have a time slice granularity of 1000ms/64 or 15625000 nanoseconds.

The underlying assumption not mentioned (or maybe I was not paying attention) is that the ideal frame rate we would like for our game is 60 frames per second. We will probably not be able to hit this rate since we are drawing the 256 x 256 several times over in the loop.

Imagine you laid out a second on real-number line. The ideal would be that every 1/60 of a second (this by the way is our `secondsPerTick` value) we update the game.

Figure 1 - Idealized frame rate 60 fps

The best counter you have available using Java is `System.nanoTime()`. The only thing we can do with this method is get the difference between the last time it was called and the current time. We then take the difference

```
long passedTime = currentTime - previousTime;
```

We take the passedTime and convert into seconds (which of course will be a fraction of a second since it is the number of nanoseconds that have passed) and add it to the variable that tracks the time that has elapsed since the last tick – unprocessedSeconds.

```
unprocessedSeconds = unprocessedSeconds + (passedTime / 1000000000.0);
```

Let's blow up what is going on in two situations when the render/update is faster than a tick (so we have to wait) and when it is more.

CASE #1: Screen update is taking place faster than 1/60 of a second.

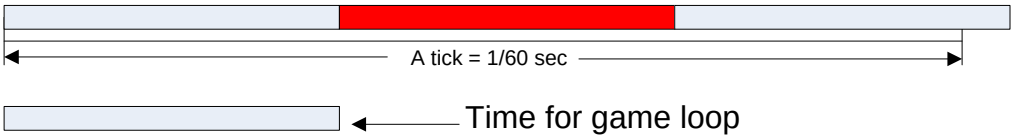

Figure 2 - Diagram of when game loop is fast

From the above you should see that each time we get the passedTime it adds about 1/90 sec to the unprocessedSeconds. Which means we will skip adding to the tickCount or bypass the while loop:

```
while (unprocessedSeconds > secondsPerTick) {
                                :
                                :
}
```

When we get to the third timeframe we now get the condition that `unprocessedSeconds > secondsPerTick` and will then enter the loop:

```
while (unprocessedSeconds > secondsPerTick) { // Has 1/60 sec elapsed, i.e. a tick
	tick();		// do it!
	unprocessedSeconds -= secondsPerTick;	// remember the time that went over a
tick
	ticked = true;				// signal to render a frame
	tickCount++;
	if (tickCount % 60 == 0) {		// approximately a second has passed, output

		System.out.println(frames + "fps");	// the number of frames
		previousTime += 1000;			// fudge factor ???
		frames = 0;
	}
}
```

We only enter the while loop if the amount of elapsed time exceed at least one clock tick. Note, if the rendering is slow unprocessedSeconds could actually exceed more than one clock tick as shown below:

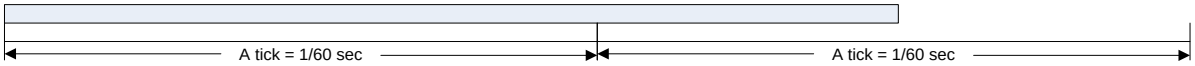

Figure 3 - Diagram of when the game loop is slow

The above shows what will happen if the game loop is slow. I intentionally made the elapsed time equal to 1.5 ticks. I will demonstrate how the code will correctly count 3 ticks.

When the variable unprocessedSeconds = 1.5 ticks it will enter the while loop update unprocessedSeconds to 0.5 ticks (subtracts a tick). Set the ticked flag to true indicating we should render a frame and increment the tickCount by 1. After the frame is rendered and the loop returns to update unprocessedSeconds it will now be at 2.0 ticks (0.5 remainder + 1.5 ticks it takes to render a frame). This will make the while loop iterate two times (each time decreasing unprocessedSeconds by 1 tick) and incrementing the tickCount. But, only one screen is rendered. In this case the frame rate will not be 60 frames per second but only 40 frames per second.

This is the entire Display.java with frame rate counting:

Table 1 - Episode 7 version of 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);
		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) {
			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() {
		// holds the number of frames per second
		int frames = 0;
		// accumulates time
		double unprocessedSeconds = 0;
		// start counting
		long previousTime = System.nanoTime();
		// ideal frame rate 60 ticks/sec
		double secondsPerTick = 1 / 60.0;
		// the number of ticks which should be 60 ticks/second
		// and we report the frame rate to the screen
		int tickCount = 0;
		// our signal indicator that a frame should be rendered
		boolean ticked = false;

		while(running) {
			// holds the time now
			long currentTime = System.nanoTime();
			// holds the time between now and last time
			long passedTime = currentTime - previousTime;
			// remember for the next check
			previousTime = currentTime;
			// real time elapsed
			unprocessedSeconds += passedTime / 1000000000.0;

			// has 1/60th sec elpased - i.e. a tick?
			while (unprocessedSeconds > secondsPerTick) {
				tick();		// do the update!
				// remember the time that went over a tick
				unprocessedSeconds -= secondsPerTick;
				// signal to render a frame to the screen
				ticked = true;
				tickCount++;
				// is it time to print the frame rate to the screen
				if (tickCount % 60 == 0) {
					System.out.println("" + frames +  " fps");
					// add a fudge factor
					previousTime += 1000;
					frames = 0;
				}
			}
			// always draw the screen as fast as possible
			render();
			frames++;
		}
	}

	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();
	}


}
```

The corresponding version of Render.java and Screen.java

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) {
		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;

				int alpha = render.pixels[x + y * render.width];

				if ( (alpha & 0x00ffffff)  > 0) {
					pixels[xPix + yPix * width] = alpha;
				}
			}
		}
	}
}
```

Table 3 - 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() {
		// let's first clear the screen
		for (int i = 0; i < (width * height); i++) {
			pixels[i] = 0xFFFFFFFF;
		}
		int xCenter = (width - BLOCK_SIZE) / 2;
		int yCenter = (height - BLOCK_SIZE) / 2;

		long currentTime = System.currentTimeMillis();
		for (int i = 0; i < 100; i++){
			int anim = (int) (Math.sin( ((currentTime % 1000.0 + i) / 1000) *
Math.PI * 2) * 200);
			int anim2 = (int) (Math.cos( ((currentTime % 1000.0 + i) / 1000)
* Math.PI * 2) * 200);
			draw(test, xCenter + anim, yCenter + anim2);
		}
	}
}
```

On my computer system I get the following results:

Table 4 - Actual frame rate

```
Running...
Working
116 fps
118 fps
119 fps
122 fps
121 fps
119 fps
	:
	:
```

The frame rate is quite high but on the original system I used 4 years ago (a six year old computer) I was getting only 21 fps!

On the original computer I was not reaching the ideal of 60 frames per second but only a 1/3 of that. Which means it is taking about 3/60 seconds to compose the screen.

To test it out you can lower the number of iterations (copies of the colored square) that are drawn to the screen you should see a difference in the frame rate. It may not be exactly linear.