

# Avoid Common Pitfalls when Programming 2D Graphics in Java: Lessons Learnt from Implementing the Minueto Toolkit

*Minueto is a platform independent game development framework written in Java, specifically designed to support teaching project-based computer science courses at the undergraduate level. It offers a simple object-oriented API that is ideal for 2D game development. Despite Java's poor reputation for game development, impressive frames-per-second performance was achieved. This paper presents the necessary knowledge required to properly develop 2D games in Java, while describing the common pitfalls that can easily degrade display performance.*

## 1. Introduction

The programming component of the first year of undergraduate studies at McGill University in Computer Science heavily focuses on object-oriented programming in Java. Thus, it is not surprising that students often choose this language when developing large-scale projects during their latter school years. These projects sometimes include game development, such as the one in the course COMP-361 Systems Development Project, a mandatory course for software engineering majors.

Java provides two frameworks for designing graphical user interfaces: AWT and Swing. AWT is mostly use for applet development; its components are considered light-weight. Swing components are more flexible, offer a wider variety of features and are more actively supported, but tend to render slower. When developing Java games, it is best to avoid both frameworks, since both frameworks are extremely slow and ill suited for game development.

The Minueto game development framework [1] was developed to help the COMP-361 students complete their game development project. Before its introduction, many students struggled to get adequate performance from their Java game. Minueto hides many of the Java-specific knowledge required to get good 2D performance, and hence students can focus on the design and implementation of the game functionality.

It took over 8 months to create the Minueto framework. A tremendous amount of work was spent into optimizing the drawing engine, hiding tricky implementation details behind an elegant and simple API. This article presents the Java 2D knowledge that was acquired during the development of this tool. It also presents the pitfalls commonly encountered by Java 2D programmers.

### 1.1 Basic Building Blocks

There are 2 basic building blocks for graphics in a 2D game, a drawing canvas and a collection of

images. The canvas is either displayed on the screen as one or several windows, or takes up the entire screen. Images are the blocks that get drawn on the canvas. These images are drawn repetitively on the screen to produce animation.

In Java, the canvas is either a Frame or a JFrame object. The images are stored in BufferedImage objects. Images can be drawn on the canvas or over other images through the use of a Graphics2D object. The following sections explain how to properly build your canvas (section 2), your images (section 3) and what other optimizations to consider (section 4) to ensure optimal Java 2D graphics performance.

## 2. The Canvas

As previously mentioned, the drawing canvas should be a simple JFrame. To insure flicker free performance, the "ignore repaint" property should be enabled (using the setIgnoreRepaint method). This prevents the Java virtual machine (JVM) from trying to modify the content of the canvas.

### 2.1 Graphic Device and Configuration

The GraphicsDevice class is used to describe graphic output devices, such as your computer's graphic card. It contains a collection of GraphicsConfiguration objects that describe all the possible configurations for the graphic card, i.e. all the supported resolutions and color modes.

```
// Acquiring the current Graphics Device and Graphics
Configuration
GraphicsEnvironment graphEnv =
GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice graphDevice =
graphEnv.getDefaultScreenDevice();
GraphicsConfiguration graphicConf =
graphDevice.getDefaultConfiguration();
```

When creating a new JFrame, we highly suggest passing the current graphic configuration constructor as a parameter. This allows the JFrame to be specially constructed for that graphic configuration and avoids common color mode problems (as described in section 3.3).

```
//Creating the JFrame
JFrame jFrame = new JFrame(graphicConf);
jFrame.setIgnoreRepaint(true);
jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jFrame.setTitle("Minueto");
jFrame.setFocusTraversalKeysEnabled(false);
```

In addition, the current graphic configuration is needed to produce hardware accelerated images (as described in section 3.4).

## 2.2 Proper Size for a Window

Creating a drawing surface of a specific size can be tricky with a normal windowed frame. A 640 by 480 pixels JFrame will have a drawing surface of approximately 600 by 470 pixels (see figure 1). This loss of space can be explained by the title bar and the window borders which are include in the original size of 640 by 480 pixels [5].

An easy solution to this problem is to insert a 640x480 Canvas object (with ignore repaint set to true) inside the JFrame. This will force the JFrame to contain a true 640 by 480 drawing surface, regardless of the size of the title bar or windows border. Note that this solution requires all drawing to be done on the Canvas, instead of the JFrame. We found that this solution is practical and incurs no performance impact.

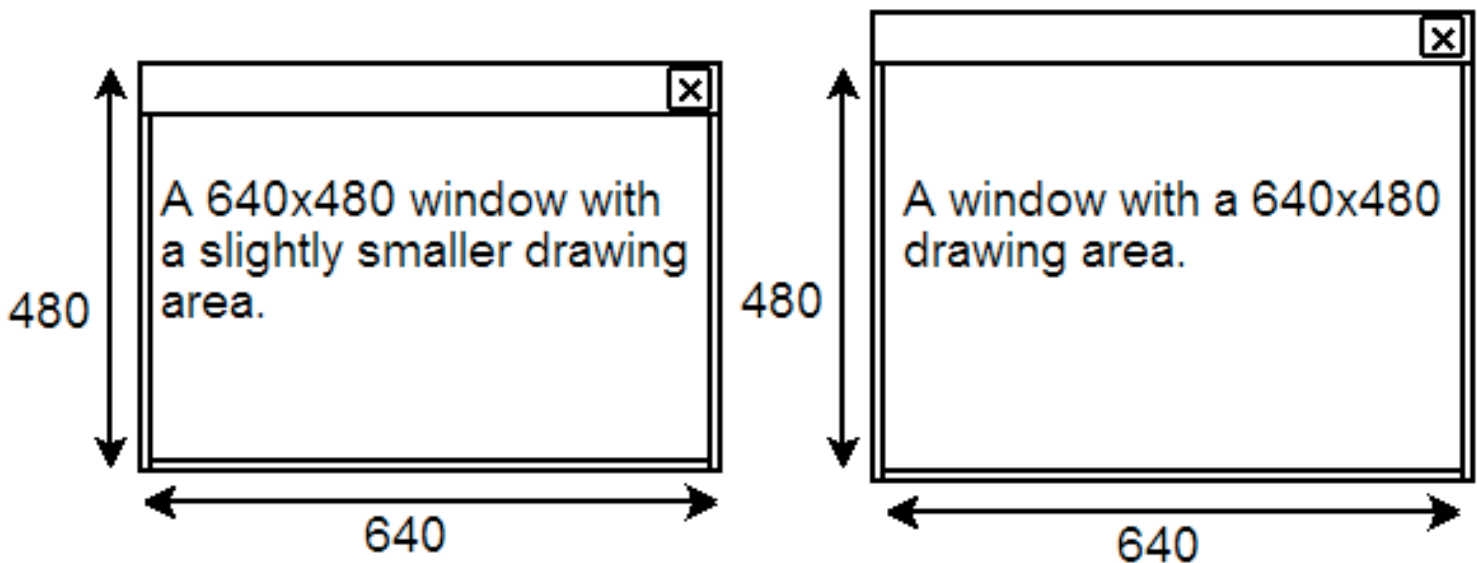


Figure 1 : Difference between a 640x480 window and a 640x480 drawing area.

Note: This problem does not exist for full screen windows or border-less windows, as they don't have borders or title bars to reduce the size of the drawing area.

## 2.3 Fullscreen Window and Changing the Resolution

The GraphicsDevice object can also be used to change the resolution of the screen or transform a frame so it covers the whole screen (full screen mode). Note that resolution changes using a pre-1.5 JVM are dangerous and could cause the process to fail [5]. In addition, resolution changes are not supported with a pre-1.4 JVM under Linux.

```
//Switching Resolution and to Full Screen
DisplayMode disMode = new DisplayMode( width, height,
32, DisplayMode.REFRESH_RATE_UNKNOWN);
graphDevice.setFullScreenWindow(jFrame);
graphDevice.setDisplayMode(disMode);
```

## 2.4 Double Buffering

The key to smooth animation in Java is to use double buffering. Double buffering is a technique where a memory buffer is created as a workspace for changes to appear on the screen. The content of this buffer is displayed on screen only once all the changes to the current frame have been completed (see figure 2). This avoids flickering and other graphic artifacts commonly produced by the progressive updating of the screen. Double buffering also allows the synchronization of the screen updates with the refresh rate of the monitor.

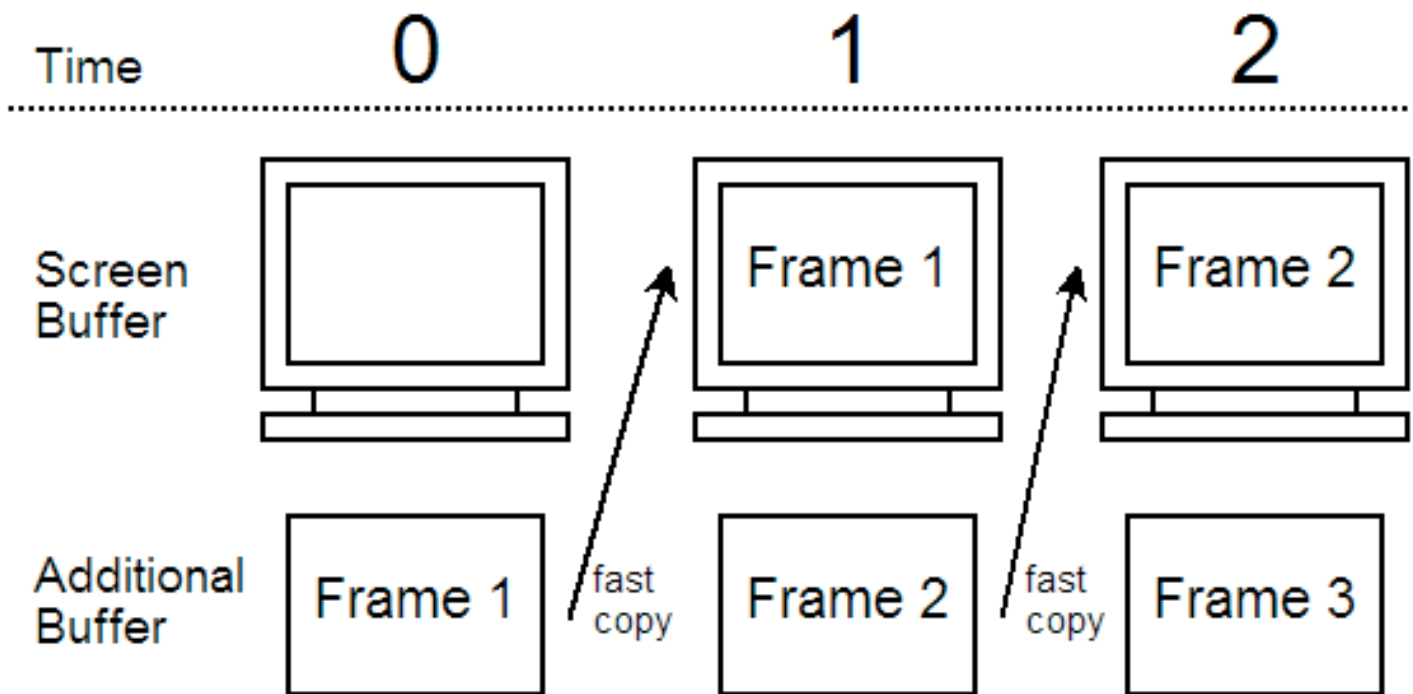


Figure 2 : Double Buffering.

In Java, double buffering is provided by the `BufferStrategy` class. Although `BufferStrategy` allows for n-buffering, we found that double-buffering (that is, creating a `BufferStrategy` with 2 buffers) was optimal.

```
//Setting up Double Buffering
jFrame.createBufferStrategy(2);
BufferStrategy bufferStrategy =
jFrame.getBufferStrategy();
Graphics2D graph2D =
(Graphics2D)bufferStrategy.getDrawGraphics();
```

## 3. Buffered Images

Chet Haase, an engineer on the Java2D team, documented no less than 38 different methods to create an image with the Java 1.4 API [2]. The different methods produce images with various characteristics. We found that the optimal methods for producing efficient BufferedImage object were found in the GraphicsConfiguration object [3] used to create the JFrame (as describe in the section 2.1). They assure greatest compatibility between the images and the drawing canvas.

```
// Creating a hardware accelerated image with bitmask
transparencies.
BufferedImage imageBitMask =
graphicConf.createCompatibleImage(x, y,
Transparency.BITMASK);
```

### 3.1 Volatile Images vs. Managed Images

Two types of images in Java are hardware accelerated: volatile images and managed images. Volatile images are stored only in video memory and are extremely fast. However, if the video memory is overwritten by another application, the content is lost. Managed images are stored in both video memory and conventional memory, thus do not suffer from the "content lost" problem. Although this redundancy incurs a performance cost, we found manually managing volatile images would be too much overhead.

Certain operations that can be applied to a BufferedImage object, such as manipulating the image at the pixel level, can make an object lose it's hardware accelerated features. When this occurs, the image is no longer stored in video memory. To make it regain its hardware acceleration, a manipulated image should be copied to a new accelerated BufferedImage before it is used.

### 3.2 Color Mode

The key to successful image acceleration is to create an image with the same characteristics as the current display. For example, if the current display is switched to RBGA 32-bit color mode, then all images should be created, loaded and/or converted to this color mode. To this extent, we copy every loaded image into a new accelerated BufferedImage, which automatically has the correct color mode.

If this is not done, and images are loaded and stored using different color modes, then they have to be converted to the color mode of the screen every time they are drawn to the canvas, causing tremendous slowdowns.

### 3.3 Transparencies

Java supports three type of transparencies: opaque (no transparencies), bitmask (a pixel is either transparent or not) and alpha (variable level of transparency for each pixel). We found the use of

either opaque or bitmask transparencies produces impressive performance, even on modest graphic hardware. Alpha transparencies are much slower, especially when the graphic card does not provide hardware support for transparency hardware. We therefore recommend avoiding alpha transparencies, unless they are specifically needed.

```
// Creating a hardware accelerated image with bitmask
alpha.
BufferedImage imageAlpha =
graphicConf.createCompatibleImage(x, y,
Transparency.TRANSLUCENT);
```

### 3.4 Rendering to the screen

As previously mentioned in section 2.4, images should first be drawn on the memory buffer.

```
//Drawing an image on the memory buffer
graph2D.drawImage(bufferedImage, x, y, null);
```

Once the frame is ready, we can render the content of the memory buffer to the screen.

```
Rendering the current buffer to screen
graph2D.dispose();
bufferStrategy.show();
graph2D =
(Graphics2D)bufferStrategy.getDrawGraphics();
```

## 4. Other Optimizations

Although not directly related to Java2D, we have found the following optimizations to be beneficial to obtaining stable performance.

### 4.1 Keyboard and Mouse Queuing

Keyboard and mouse input is event driven in Java: whenever an input event, such as a mouse click, occurs, a thread calls a so-called listener method to execute the behavior attached to the event. Although this model is ideal for most applications with graphical user interfaces, it is not ideal for games in which the display is updated with a high frame rate.

In order to assure a constant frame, we decided to store keyboard and mouse events in queues instead of processing them immediately when they occurred. This makes it possible to process the events in a control fashion, at a time, in a moment when the smooth game play is not affected.

In games with a simple game loop, the input events can be taken from the queue and processed with the same thread that performs the updating and rendering of the display. As a result, the multi-threading issues commonly found in event driven applications are avoided.

## 4.2 : Yielding vs Sleeping

At the heart of most games is a game loop that processes input events, updates the game state and refreshes the display. If proper care is not taken, the main thread executing the game loop can starve other threads, as the loop runs infinitely at high speed. To avoid this, Java programmers should add either a yield command or a sleep command to the code running inside the loop.

Although we have found that a rendering loop with the sleep command uses less CPU power, our experience has shown that sleep does not return in a timely fashion. We found that to provide a constant frame rate, we were forced to use the more CPU intensive yield.

## 5. Conclusion and Future Work

Despite our initial apprehension about using Java for game development, Java 2D has shown to be very efficient, if used properly. Two years of using Minueto for the game development of COMP-361 projects has shown that it performs extremely well, thanks to the different optimizations presented in this article. Because of its simple design and extensive documentation, Minueto is very easy to learn. The students did not encounter any major problems when building a specific object-oriented game architecture on top of the Minueto API.

It comes to no surprise that some students have shown interest in the integration of Swing components with Minueto: some games require GUI components such as text boxes and buttons as part of their graphical user interface. Our current research plans involve understanding the inner workings of Swing and exploring ways to efficiently combine Swing and Java 2D technology.

## References

1

Alexandre Denault, *Minueto, an Undergraduate Teaching Development Framework*, Master's Thesis, McGill University, August 2005.

2

Chet Haase, *BufferedImage as Good as Butter Part 1*, August 14, 2003, < [Chet Haase's Blog](#)>.

3

Chet Haase, *ToolkitBufferedVolatileManagedImage Strategies*, August 11, 2004, < [Chet Haase's Blog](#)>.

4

Clingman / Kendall / Mesdaghi, *Practical Java Game Programming*, Charles River Media, June 2004, 508 pages.

