Building a Java First-Person Shooter

Episode 1 [Last update: 2/15/2013]

These notes are intended to accompany the video sessions being presented on the youtube channel “3D Java Game Programming” by youtube member “The Cherno” at https://www.youtube.com/playlist?list=PL656DADE0DA25ADBB. I created them as a way to review the material and explore in more depth the topics presented. I am sharing with the world since the original work is based on material freely and openly available.

Prerequisites

You should be comfortable with basic Java programming knowledge that would be covered in the one-semester college course. If you are not then I recommend the following websites for tutorials and online textbooks:

- Java Language and Virtual Machine Specifications at http://docs.oracle.com/javase/specs/
  - This is not intended to be used by those who want to learn Java but more as a reference or to lookup of areas you are not familiar with once you have gone through a course or one of the books suggested in this list.
  - This too is less of a “learn Java” book but is a great book to read while learning Java to get into the Java frame of mind
  - This is an excellent free e-book that could be used in any college course

You should also know your way around the Eclipse IDE. If you don’t then I recommend the following links for tutorials and information:

- Video tutorials: http://eclipsetutorial.sourceforge.net/totalbeginner.html

I think many youtube programming video presenters failed to identify the starting level or prerequisites required to view and learn from their videos. This presents a problem for the viewer since they discuss at length topics the viewer should clearly be comfortable with and not enough on the topics that may be new after a one semester course. The creator of these videos makes the same mistake. It can be irksome to developers with experience who just want to get to the heart of the Java classes and techniques applicable to the subject at hand and disconcerting to the novice since the very brief discussion on arrays, for-loops is not enough time to solidify these concepts. Setting the expectation that viewers can learn Java at the same time as learning the concepts involved in creating a 3D game is
not realistic and just demoralizes the average person (proof is the number of people who view the first video but don’t get to the second or the third!) Of course it is surprising how smart and hardworking many of the viewers in pursuing and understanding of the material and I hope these notes can assist in that journey.

Lessons/Episodes
Each of the major video episodes will have a corresponding pdf to explain independently of the video how the code works and more details on the Java libraries used to accomplish the graphics. I also added additional notes on other related topics (e.g. Java Graphics Programming, Random Dungeon Generator, etc). I felt the explanation offered in the videos and just blind use of Java classes without an explanation of their purpose and function in the final version of the code is very frustrating in the event that something does not work or if I truly plan on building my own 3D game. I hope these notes provide more of an explanation to the Java classes that are used to build a 3D world and prove useful to others.

The total number of videos is 39.

- Episode 1 – Window
- Episode 2 – Game Loop
- Episode 3 – Arrays
- Episode 4 – Drawing Pixels
- Episode 4.5 – How Rendering Works
- Episode 5 – Playing with Pixels!
- Episode 6 – Performance Boosting
- Episode 7 – FPS Counter
- Episode 8 – Alpha Support and More
- Episode 9 – Beginning 3D
- Episode 10 – Floors and Animation
- Episode 11 – Rotation
- Episode 12 – User Input
- Episode 13 – Render Distance Limiter!
- Episode 14 – Basic Mouse Movement
- Episode 15 – Textures + More!
- Episode 16 – Walking, Crouching, Sprinting + More
- Episode 16.5 – Exporting Runnable Jars
- Episode 17 – Small Adjustments + Birthday!
- Episode 17.5 – Creating an Applet
- Episode 18 – The Beginning of Walls
- Episode 18.1 – A Few More Things
- Episode 18.5 – Creating an EXE File in Java
- Episode 19 – Rendering Walls
- Episode 20 – Continuing Walls, Fixing Bugs, and M
- Episode 21 – Texturing Walls, Fixing Clipping and
Episode 1

In this episode you are presented with enough information to:

- Create a window
- Set it to a certain size
- Have it close cleanly
- Display a title

Video URL:

STEP 1:
Create a new Java project named Minefront. (Note: for these notes I first created a new Eclipse workbench called JAVA_MINEFRONT under C:\JAVA_MINEFRONT).

STEP 2:
Create a new class com.mime.minefront.Display that extends the Canvas class. This class will be the “kickoff” class for our game, that is, it will contain a main() method that will be invoked by the platform operating system (OS).

STEP 3:
Enter the following text and compile and execute.

Table 1 – Display.java (version #1)

```java
package com.mime.minefront;
import java.awt.Canvas;
import javax.swing.JFrame;
public class Display extends Canvas {
```
```java
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";

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..." firefight the fire.
}
```
How the code works

AWT and Swing
The code uses two classes from two different libraries. The first class `java.awt.Canvas` comes from what is referred to as the AWT library. AWT stands for Abstract Window Toolkit. This library came with the original or first version of Java (1995). AWT provided the “windowing, graphics and user-interface widget toolkit.” So it is the library to use to create a GUI application that is, applications with windows, button, menus and textboxes. In order to come up with something very quickly that could provide graphic functionality across many machines and platforms AWT was purposely designed as a thin layer between Java and the actual platform’s graphical APIs. That is the graphic component (windows, buttons, menus etc) are rendered by the platform operating system graphics library. There were two major problems with this approach. The first is the lowest common denominator of graphics and window functionality was provided and second all the applications took on the look and feel of their native platform so you could not get applications to look (and even behave) the same across platforms since it was their underlying operating systems APIs that was being used to draw and manage the screen.

The class `javax.swing.JFrame` comes from the Swing library. In Java version 1.2, Sun Microsystems introduced the Swing toolkit. Swing was first developed by the then Netscape Communications Corporation in 1996. The goal was to develop sharper and more elegant looking GUI components than what was provided by AWT. Swing was developed so that applications would appear the same across different platforms. In addition, the look and feel was intended to be pluggable. The library provided a richer set of widgets that were implemented strictly in Java.

The current version of Java handles more easily the mixing of components from both toolkits (that was once a problem). The two libraries are not independent from each other as the class hierarchy diagram below illustrates:

---

1 The first version of Java ran on the following platforms: Windows 95 and NT, Sun Solaris and later Mac OS 7.5.
Both AWT and Swing are now part of the Java Foundations Classes (JFC). The fundamental GUI object shared by both AWT and Swing is the java.awt.Component class. A component is an object having a graphical representation that can be displayed on the screen. So all the entities we see on a Java GUI screen derives from the component class. There are two types from components – lightweight and heavyweight. A lightweight component is not associated with a native window (this is true for all Swing components) and a heavyweight component is associated with a native window (this is true for all AWT components). A container is just an object that can contain other components.

It is easy to tell the difference between classes that are associated with AWT from those that are associated with Swing – all Swing classes start with the letter J (e.g. JComponent, JWindow, JFrame, etc).
Canvas

The Canvas component represents the area on the screen where your applications draw images, buttons, text, missiles, bombs, and dogs! Applications (like our example) subclass the class Canvas in order to gain access to the functionality offered by that class.

Typical use:

```java
public class MyGreatGameDisplay extends Canvas {
}
```

This episode does not involve any painting to the screen but typically you would use the `paint(Graphics g)` method to draw your blown up enemies on the screen (more on this in future episodes) if you were using AWT Frame to hold your window. Since we are using JFrame a Swing class we should override `paintComponent()` method instead. In this version of the program we create:

```java
public class Display extends Canvas {
    :
    :
}
```

It should be obvious why we are calling this class “Display” since it will handle the game display screen.

JFrame
Figure 4 - JFrame class hierarchy

A Frame is a top-level window with a title and a border. The size of the frame includes an area designated for the border. The dimensions of the border area may be obtained using the `getInsets` method. Since the border area is included in the overall size of the frame, the border effectively obscures a portion of the frame, constraining the area available for rendering and/or displaying subcomponents to the rectangle which has an upper-left corner location of `(insets.left, insets.top), and has a size of width - (insets.left + insets.right) by height - (insets.top + insets.bottom).` What all this means is that even if you created the frame to be WIDTH x HEIGHT your Canvas object will have less space than that for drawing since the insets or window borders take room. The equations above detail how much less. In a future, episode CHERNO will make adjustments to ensure the Canvas or drawing area matches our desired WIDTH x HEIGHT specifications.

A frame, implemented as an instance of the `JFrame` class, is a window that has decorations such as a border, a title, and supports button components that close or iconify the window. Applications with a GUI usually include at least one frame. Applets sometimes use frames, as well.

Here is a picture of the extremely plain window created by the FrameDemo demonstration application.

The following FrameDemo code shows how to create and set up a frame.

```java
http://docs.oracle.com/javase/tutorial/uiswing/components/frame.html
```

This whole section is from this site.
Here are some details about the code:

1. The first line of code creates a frame using a constructor that lets you set the frame title. The other frequently used JFrame constructor is the no-argument constructor.

2. Next the code specifies what happens when your user closes the frame. The EXIT_ON_CLOSE operation exits the program when your user closes the frame. This behavior is appropriate for this program because the program has only one frame, and closing the frame makes the program useless.

3. The next bit of code adds a blank label to the frame content pane. If you’re not already familiar with content panes and how to add components to them, please read Adding Components to the Content Pane.

4. The pack method sizes the frame so that all its contents are at or above their preferred sizes. An alternative to pack is to establish a frame size explicitly by calling setSize or setBounds (which also sets the frame location). In general, using pack is preferable to calling setSize, since pack leaves the frame layout manager in charge of the frame size, and layout managers are good at adjusting to platform dependencies and other factors that affect component size.

This example does not set the frame location, but it is easy to do so using either the setLocationRelativeTo or setLocation method. For example, the following code centers a frame onscreen:

```java
frame.setLocationRelativeTo(null);
```

5. Calling setVisible(true) makes the frame appear onscreen. Sometimes you might see the show method used instead. The two usages are equivalent, but we use setVisible(true) for consistency’s sake.

The program creates a JFrame object and adds our canvas to the frame:
JFrame frame = new JFrame();
frame.add(game);

The next couple of instructions just set certain characteristics of our window:

frame.setTitle(TITLE);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(WIDTH, HEIGHT);
frame.setLocationRelativeTo(null);
frame.setResizable(false);
frame.setVisible(true);

The code above sets the title of the window, what action to take when the user clicks on the "close window" icon, the size of our window, where to place the window on the screen, if our window is resizeable, and then to make it visible to the user.

The following presented more details on the other methods in our first episode:

 setLocationRelativeTo

public void setLocationRelativeTo(Component c)

Sets the location of the window relative to the specified component. If the component is not currently showing, or c is null, the window is placed at the center of the screen. The center point can be determined with GraphicsEnvironment.getCenterPoint.

 setResizable

public void setResizable(Boolean resizable)

Sets whether this frame is resizeable by the user.

 add

public Component add(Component comp)

Appends the specified component to the end of this container.

 serialVersionUID?

The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If
the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException. A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:

```java
static final long serialVersionUID = 42L;
```

If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class—serialVersionUID fields are not useful as inherited members.

FROM: http://stackoverflow.com/questions/285793/what-is-a(serialVersionUID-and-why-should-i-use-it

If the objects you create from the class can be saved or serialized to a file for later processing (e.g. the user issues a “save game” operation you will want to save all the game objects) than you will want to create a unique serialVersionUID for the object associated with the class. Doing so allows you to “check” if a future version of the class can handle the version saved to a file.

References

- http://home.cogeco.ca/~ve3ll/jatutorg.htm
- http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/Canvas.html
- http://docs.oracle.com/javase/tutorial/uiswing/components/frame.html

Improvements/Suggestions

I would have recommended another name for this class rather than Display. It will get confusing later with the addition of a class called Render and Screen since they could all mean the same thing. In addition, it would make sense to create a class which subclasses JFrame. The class would have the name of your game and the second class would subclass Canvas. This is the general form of your game class:

```
package skeleton;
```
import javax.swing.JFrame;

public class MyCoolGame extends JFrame {

    public final int WIDTH = 300;
    public final int HEIGHT = 280;

    public MyCoolGame() {
        add(new Board());
        setTitle("My Cool Game");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(WIDTH, HEIGHT);
        setLocationRelativeTo(null);
        setVisible(true);
        setResizable(false);
    }

    public static void main(String[] args) {
        new MyCoolGame();
    }
}

package skeleton;

import javax.swing.JPanel;

public class Board extends JPanel {

    public Board() {
    }

}

The other fact to note is that the playable area will not be WIDTH x HEIGHT but a little less than that since some pixels are taken up for the actual window borders.
Even though we defined the frame to be 300 x 280 the canvas or displayable area will be less than this. What we should do is reset the window size to make our displayable area match our expectations of being WIDTH x HEIGHT. The way to do this is to obtain the border values and re-adjusting the window size. Here is how:

```java
package skeleton;
```
import java.awt.Insets;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class MyCoolGame extends JFrame {
    public static final int DISPLAY_WIDTH = 300;
    public static final int DISPLAY_HEIGHT = 280;

    private static int DISPLAY_X;
    private static int DISPLAY_Y;

    public MyCoolGame() {
        add(new Board());
        setTitle("My Cool Game");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(DISPLAY_WIDTH, DISPLAY_HEIGHT);
        setLocationRelativeTo(null);
        setResizable(false);
        setVisible(true);

        Insets insets = getInsets();
        DISPLAY_X = insets.left;
        DISPLAY_Y = insets.top;
        resizeToInternalSize(DISPLAY_WIDTH, DISPLAY_HEIGHT);
    }

    public void resizeToInternalSize(int internalWidth, int internalHeight) {
        Insets insets = getInsets();
        final int newWidth = internalWidth + insets.left + insets.right;
        final int newHeight = internalHeight + insets.top + insets.bottom;

        Runnable resize = new Runnable() {
            public void run() {
                setSize(newWidth, newHeight);
            }
        };

        if (SwingUtilities.isEventDispatchThread()) {
            try {
                SwingUtilities.invokeLater(resize);
            } catch (Exception e) {
                // ignore ...but will be no no if using Sonar!
            }
        } else {
            resize.run();
        }
    }

    public static void main(String[] args) {
        new MyCoolGame();
    }
}
The new version of MyCoolGame.java adds a new method `resizeToInternalSize` that makes the adjustment to the displayable area (or in this case our Canvas) by adjusting the windows width and height by taking into account the actual border values. How it actually gets this done looks a bit esoteric and complex since we are creating a thread to perform the adjustment. The next episode will discuss threads and the EventDispatcher in more detail so we will not discuss this code until then.

Note: The above is additional material not discussed in the video episodes. I learned this technique from the book *Java 1.4 Game Programming* by Andrew Mulholland and Glenn Murphy. The book did not do so well when released mostly because it made the same mistake most books and video tutorials do — try to teach java and game programming at the same time. If someone new to programming ever gets to the pages discussing the game and graphics they would have been totally clueless and I imagine ready to pull out their hair. The ones who already knew how to code would have been equally frustrated and ready to pull their hair having to go through two hundred pages before getting into anything related to creating a game. Having said that I enjoyed the book and found many of the chapters worthwhile in creating 2D games. It would have been a GREAT book if it would have immediately focused on games and actually have the users build a classic 2D game from beginning to end rather than just presenting the ideas and sample programs. You can actually find the book online. I should have a link somewhere on brainycode.com

In summary, this episode created a simple GUI window that will be the foundation of our 3D java application as we add classes and functionality.