Part I.

A stimulus display framework

2. Hello Psychophysicist

In this chapter

- Initiating the full screen exlusive mode (FSEM)
- Active vs. passive rendering; Double buffering
- Displaying text and image on the screen
- Pausing (sleeping) for a while
- Hiding and showing the cursor
- Terminating the FSEM

We start with a simple example, which displays the text “Hello Psychophysicist” and two images on an otherwise entirely blank screen (see Chapter 1, *Development environments* Section for information on compilation and execution)

```
/*
 * chapter 2: HelloPsychophysicist.java
 *
 * displays the text "Hello Psychophysicist" and two images
 * on an otherwise entirely blank screen
 *
 */
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

public class HelloPsychophysicist {

  public static void main(String[] args) {

    FullScreen1 fs = new FullScreen1();
    fs.setNBuffers(2);

    try {
      fs.displayText("Hello Psychophysicist");
      fs.updateScreen();
      Thread.sleep(2000);
      fs.blankScreen();
      BufferedImage bi1 = ImageIO.read(new File("psychophysik.png"));
      fs.displayImage(bi1);
      fs.updateScreen();
      fs.hideCursor();
      Thread.sleep(2000);
      fs.blankScreen();
      BufferedImage bi2 = ImageIO.read(new File("fechner.png"));
      fs.displayImage(0,0,bi2);
      fs.updateScreen();
      Thread.sleep(2000);
    } catch (IOException e) {
      System.err.println("File not found");
      e.printStackTrace();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    finally {
      fs.closeScreen();
    }
  }
}
```

Initiating Full Screen Exclusive Mode

The first line in the main() method

```
FullScreen1 fs = new FullScreen1();
```

creates an object of the FullScreen1 class. As soon as a FullScreen1 object is created it switches to full screen exclusive mode (FSEM). The visible effect is that the screen goes completely blank and all the OS and Window Manager related decorations disapear, such as a task bar.

Setting the number of video buffers

In the next line

```
fs.setNBuffers(2);
```

sets the total number of video buffers to 2, one is the actual display screen, the other is a back buffer. See Section 2.2 below for more on Double Buffering. If using 2 buffers doesn't work on your hardware/OS, try using 1 or 3. Single buffer (fs.setNBuffers(1)) almost always works, but this results in sacrificing double buffering and introduces tearing artifacts. See chapter XX for more on fine tuning the Buffer Strategies.

Displaying Text, Image, and blank screen

In the next line we start a try-catch-finally clause: it *tries* to perform the statements inside the try{...} part, if anything goes wrong, the execution jumps to the catch{...} part. If nothing goes wrong the catch{...} part is skipped. In either case finally{...} part is always executed. This is a better programming style and robust way of dealing with errors at run time: If an error happens to ocur during execution, the program doesn't have to terminate abruptly, instead it can try to fix the problem and continue its execution. Even if everything fails, it may still terminate, but terminate in a cleaner way, in a way decided by the programmer a-priori. Moreover, placing short descriptive messages inside catch{...} will surely be helpful.

The lines inside the try clause

```
fs.displayText("Hello Psychophysicist");
fs.updateScreen();
Thread.sleep(2000);
fs.blankScreen();
BufferedImage bi1 = ImageIO.read(new File("psychophysik.png"));
fs.displayImage(bi1);
fs.updateScreen();
fs.hideCursor();
Thread.sleep(2000);
fs.blankScreen();
BufferedImage bi2 = ImageIO.read(new File("fechner.png"));
fs.displayImage(0,0,bi2);
fs.updateScreen();
Thread.sleep(2000);
```

do exactly what the names of the methods imply: display text, display image, blank the screen, hide the cursor, read in images, and sleep for a while. displayText(), displayImage and blankScreen() methods actually manipulate the back video buffer (given that there is a back buffer, recall that we turned on double buffering by setting the number of buffers to 2), but those changes are not actually displayed on the screen until the the updateScreen() method is invoked. Details of updateScreen() method is given below in Section XXX.

displayText(String message) displays the given text at the center of the screen. The overloaded version of this method **displayText(int x, int y, String message)** displays the given text at the location (x,y) relative to the upper left corner of the screen.

Next method **blankScreen()** blanks the entire screen with the default background color.

displayImage() method displays an image centered on the screen. This method accepts images of the type BufferedImage. To obtain a BufferedImage from the file we first need to create a File object

```
new File("psychophysik.png")
```

very roughly speaking, this is similar to opening the file for reading its contents. This File object is then passed to the ImageIO.read() method, which actually reads the file and creates a BufferedImage object

```
BufferedImage bi1 = ImageIO.read(new File("psychophysik.png"));
```

We then display this BufferedImage object

```
fs.displayImage(bi1);
```

(See Chapter XX for other ways of creating images.) ImageIO.read() also throws an Exception, namely IOException, signaling that an I/O problem of some sort has occurred. This time we warn the user about the exception

```
catch (IOException e) {
  System.err.println("File not found");
  e.printStackTrace();
}
```

printStackTrace() method reports further details about the Exception.

Pausing (sleeping) for a while

sleep(long msec) method of the Thread class (one of the classes in the standard Java Core API) pauses the execution of the program for given number of milliseconds, msec. But this method *throws* an Exception. That means, something may go wrong when we invoke this method and Java compiler forces us to be aware of this and take precautions. That's why we need to *catch* the *Exception* that sleep() *throws*

```
catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
```

In this example we don't have to worry too much because we have a single running thread (see Chapter XX for more on threads). We only invoke the interrupt() method of Thread class which clears the interrupted state of the current thread. In Chapter XX I will provide more details about *Exception* handling in Java.

Terminating FSEM

In finally{...} we cleanly close the FSEM and let the program terminate normally

```
finally {
  fs.closeScreen();
}
```

Summary

Here is a quick summary for displaying visual stimulus using FullScreen (or FullScreen1) class

1. To Initiate the Full Screen Exclusive Mode create a FullScreen object, by invoking the constructor method **FullScreen()**.
2. (Optional) Adjust the number of video buffers by invoking **setNBuffers(int n)** method.
3. Put your stimulus display code inside the try{} portion of a try-catch-finally clause.
4. To display images, text and blank screen, use **displayImage()**, **displayText()**, **blankScreen()** methods, followed by **updateScreen()**, which actually applies and shows the changes on the screen.
5. (Optional) Hide the cursor during the animation using **hideCursor()** method.
6. ***Properly terminate FSEM***. Put the call to **closeScreen()** method inside the finally{} portion of the try-catch-finally clause. This way you can be sure that the FSEM will be terminated properly. Otherwise your system may hang.

In the sections below I will explain in further detail how the FullScreen1 object is created in FSEM and how its methods work.

2.1. Full Screen Exclusive Mode (FSEM)

The first method invocation in the main() method

```
    FullScreen1 fs = new FullScreen1();
```

creates an object of the FullScreen1 class and as soon as it is created the windowing environment switches to full screen exclusive mode (FSEM). FSEM feature was first introduced with Java 2 Standard Edition (J2SE) 1.4 (current J2SE version is 1.5.) It allows programmers to suspend the windowing system of the underlying OS, and directly access the video card and draw on the screen. If FSEM is not supported a regular window is positioned at (0,0) and resized to fit the whole screen to imitate full screen exclusive mode (for further details see Full-Screen Exclusive Mode API tutorial at `http://java.sun.com/docs/books/tutorial/extra/fullscreen`.)

Here are the class decleration, global fields and the constructor of the FullScreen1 class

```
    public class FullScreen1 extends JFrame {

      private static final GraphicsEnvironment gEnvironment =
          GraphicsEnvironment.getLocalGraphicsEnvironment();
      private static final GraphicsDevice gDevice =
          gEnvironment.getDefaultScreenDevice();
      private static final GraphicsConfiguration gConfiguration =
          gDevice.getDefaultConfiguration();

      private int nBuffers = 1;
      private Color bgColor = Color.BLACK;
      private Color fgColor = Color.LIGHT_GRAY;
      private Font defaultFont = new Font("SansSerif", Font.BOLD, 36);

      public FullScreen1() {

        super(gConfiguration);
        try {
          setUndecorated(true);
          setIgnoreRepaint(true);
          setResizable(false);
          setFont(defaultFont);
          setBackground(bgColor);
          setForeground(fgColor);
          gDevice.setFullScreenWindow(this);
          setNBuffers(nBuffers);
        }
        finally {}
      }
      //...
    }
```

First note that FullScreen1 *extends* JFrame

```
public class FullScreen1 extends JFrame
```

this means that FullScreen1 *inherits* all the methods and fields of the JFrame class, which is one of the more than 3,000 classes in the core Java Application programming interface (API). In other words, FullScreen1 *is-a* JFrame *and more*. This is called *inheritance* in object oriented programming.

Tip: Have a look at the definition of fields and methods in JFrame at `http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/JFrame.html`. You can apply all those methods to a FullScreen1 object.

Hint: Did you notice the private keyword in the field declerations? This keyword ensures that no one, other than the methods in FullScreen1 itself can access those fields. This is called *Data Hiding* and is indeed an extremely useful feature in object oriented programming. Why? we will see more later.

The method

```
public FullScreen1() {
  //...
}
```

constructs a FullScreen1 object, therefore it is called a *constructor*. Moreover, this is a *default constructor* because it accepts no arguments. First line

```
super(gConfiguration);
```

is a call to the *super* class JFrame, the class from which FullScreen1 is inherited. The argument is a GraphicsConfiguration object. This allows us to create a FullScreen1 object using the characteristics of our current graphics destinations such as our monitor.

The next methods

```
setUndecorated(true);
setIgnoreRepaint(true);
setResizable(false);
```

are all needed for a proper switch to FSEM: we set FullScreen1 undecorated because we want the screen completely blank, without any window borders; we set FullScreen1 ignore the repaint commands of the underlying OS, because we want to take the whole control of the display - this is called *Active Rendering* (see Section 2.2 below); we set FullScreen1 non-resizable, becuase we want it to always occupy the entire screen, we don't want anyone to resize it.

Next lines

```
setFont(defaultFont);
setBackground(bgColor);
setForeground(fgColor);
```

set the default foreground and background colors, and the default font. Note that text will be rendered with this font and with this foreground color.

The next method

```
gDevice.setFullScreenWindow(this);
```

initiaties the FSEM, now our program has the full control of the graphics device through this FullScreen1 object. Only after this we can adjust the number of buffers we want to use by invoking the method

```
setNBuffers(nBuffers);
```

By default the number of buffers is set to 1.

2.2. Active Rendering and Double Buffering

Whether in FSEM or not, we almost always want to use active rendering as opposed to passive rendering in our experiments - in passive rendering the underlying OS may intervene and send directives to the rendering program, whereas in active rendering the program itself is responsible of drawing and re-drawing the contents on the screen without the OS's intervention, that means more control over the behavior of the stimulus. To do this, we first draw the image on an *offscreen buffer*, often called as *back buffer.* Only after rendering onto back buffer is finished the image is brought to the screen, or *front buffer*. The critical variable is the mechanism of this process. There are several mechanisms to bring the back buffer to front. One way is rendering to a back buffer and then moving just the video pointer. In this way nothing is copied between different locations on the video memory, only a memory pointer is flipped internally: what was back buffer before becomes the front, what was front buffer becomes the back. This is called *page flipping.* The other possibility is actually copying the entire memory to the front buffer, this is called *blit buffering,* or *blitting* in short. This can happen in two ways: accelerated or unaccelerated. See Chapter XX for more details.

Here is how we create the appropriate BufferStrategy in the setNBuffers() method

```
public void setNBuffers(int n) {

  try {
    createBufferStrategy(n);
    nBuffers = n;
  } catch (IllegalArgumentException e) {
    System.err.println(
        "Exception in FullScreen.setNBuffers: "
        +"requested number of Buffers is illegal - falling back to default");
    createBufferStrategy(nBuffers);
  }
  try{
    Thread.sleep(200);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
}
```

this method attempts to create a new strategy with n buffers (including the front buffer) and with the best available strategy: it tries page-flipping first, if not available tries blitting with accelerated buffers, if that is also not available it tries unaccelerated blitting. The BufferStrategy class represents the mechanism of how the memory on a particular Canvas or Window is organized. Note that we needed to include an arbitrary amount of sleep time (200 ms.) because unfortunately createBufferStrategy() method is asynchronous and it is necessary to implement this rather inelegant work around. However, this is done only once during the creation of the FullScreen1 object. See Chapter XX for more details on fine tuning your buffer strategies.

2.3. Displaying Image and Text

displayImage() method draws a BufferedImage on a back buffer (if available), but not directly on to the screen (it draws directly on the screen if the total number of buffers is 1, i.e. there is no back buffer). This method is *overloaded:* It is possible to invoke the displayImage() method in two different ways. If the

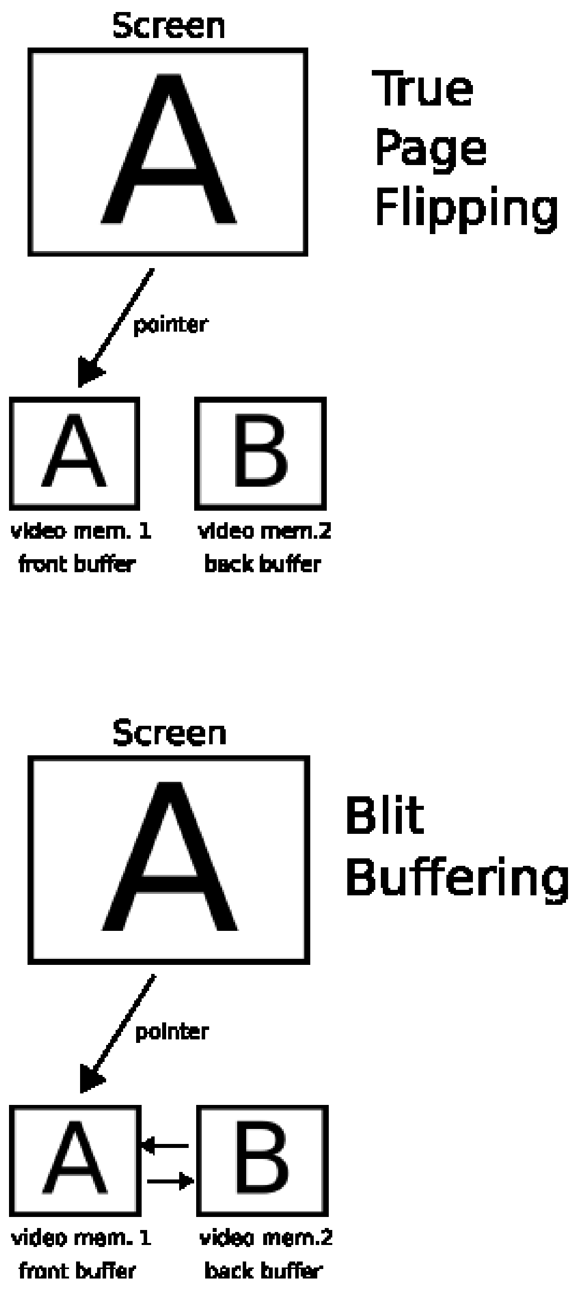

Figure 2.1.: True page flipping and blit buffering. In true page flipping only a pointer changes. In Blit Buffering the video memories are sawpped.

argument is only a BufferedImage, the method positions the image at center. The second displayImage(int x, int y, BufferedImage bi) method places upper left corner of the image at the given coordinates, x and y.

The first method simply calculates the coordinates of the upper left corner of the image to position it at the center and then invokes the other version

```
public void displayImage(BufferedImage bi) {

  double x = (getWidth() - bi.getWidth()) / 2;
  double y = (getHeight() - bi.getHeight()) / 2;
  displayImage((int) x, (int) y, bi);
}
```

here is the second overloaded version

```
public void displayImage(int x, int y, BufferedImage bi) {

  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null)
      g.drawImage(bi, x, y, null);
  }
  finally {
    g.dispose();
  }
}
```

First we obtain the Graphics2D object associated with the video buffer by invoking the getDrawGraphics() method.

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

Graphics2D class is the fundamental class for rendering 2-dimensional shapes, text and images. Then we simply draw our image on this Graphics2D, effectively drawing the image on the video buffer. Finally we dispose the Graphics2D object that we obtained in the beginning.

The logic of displayText() methods are similar to displayImage() methods

```
public void displayText(String text) {

  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  if( g!=null && text!= null){
    Font font = getFont();
    g.setFont(font);
    FontRenderContext context = g.getFontRenderContext();
    Rectangle2D bounds = font.getStringBounds(text, context);
    double x = (getWidth() - bounds.getWidth()) / 2;
    double y = (getHeight() - bounds.getHeight()) / 2;
    double ascent = -bounds.getY();
    double baseY = y + ascent;
    displayText((int) x, (int) baseY, text);
  }
}
public void displayText(int x, int y, String text) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && text!=null){
      g.setFont(getFont());
      g.setColor(getForeground());
      g.drawString(text, x, y);
    }
  }
  finally {
    g.dispose();
  }
}
```

The active rendering mechanism is exactly the same as in the displayImage() method. The difference arises from the complexity of finding the right coordinates to center the text on the screen. To get the necessary information we use a Graphics2D object rather then a Graphics object.
blankScreen() method clears and blanks the entire screen

```
  public void blankScreen() {

    Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
    try {
      if(g!=null){
        g.setColor(getBackground());
        g.fillRect(0, 0, getWidth(), getHeight());
      }
    }
    finally {
      g.dispose();
    }
  }
```

As you see, blankScreen() uses active rendering in a very similar way as the displayImage() and displayText() methods. First we obtain a Graphics object and on this object we draw a rectangle which covers the entire screen. This is achieved by a call to the fillRect() method of the Graphics class. I will also introduce two methods to get and set background color. Normally this shouldn't be necessary because JFrame has the two methods, getBackground() and setBackground(), to do that. But I found that those methods don't work reliably in FSEM so I will overide them

```
    public Color getBackground(){

      return bgColor;
    }

    public void setBackground(Color bg){
      bgColor = bg;
    }
```

We drawed the text or image on the video buffer, how are we going to actually display it on the screen? To display the video buffer on the screen, the program must invoke the updateScreen() method to apply the change to the actual display screen. This method draws the back buffer on the screen using the BufferStrategy created above

```
    public void updateScreen() {
      if (getBufferStrategy().contentsLost())
        setNBuffers(nBuffers);
      getBufferStrategy().show();
    }
```

We use contentLost() method of BufferStrategy class to check whether the video memory content is demaged. Sometimes the video memory may get lost, if this is the case we should re-allocate the memory, the best way to do this is to freshly create a new BufferStrategy object

```
    if (getBufferStrategy().contentsLost())
      setNBuffers(nBuffers);
```

In the displayImage() or displayText() methods, the image is drawn on the back buffer but not yet on the display screen. The show() method takes the necessary step to make the back buffer visible on the display using the strategy previously created

```
      getBufferStrategy().show();
```

Note: show() method of BufferStrategy waits for a vertical-sync (VSync) signal from the screen, this means that the back buffer will be brought to the front in synchrony with the screen's vertical refresh frequency. This eliminates the much undesired tearing effect in psychophysics experiments. Nevertheless all this is true only in principle, you should still check and fine tune your buffer strategies for the desired performance. Moreover, vertical-sync may not be available on all platforms and with all video cards. I had both satisfactory results and failures with various OSs and video cards. Newer NVidia chip based cards under Windows XP and Mac OS X seem to usually work. Under Linux JSE 1.5 doesn't always get the synchronization right, however JSE 1.6 seems to work pretty well with the opengl pipeline enabled. See Chapter XX for more details on vertical-sync.

2.4. Hiding and showing the cursor

hideCursor() and showCursor() methods hide and show the cursor respectively

```
    public void hideCursor() {

      Cursor noCursor = null;
      Toolkit tk = Toolkit.getDefaultToolkit();
      Dimension d = tk.getBestCursorSize(1,1);
      if((d.width|d.height)!=0)
        noCursor = tk.createCustomCursor(
            gConfiguration.createCompatibleImage(d.width, d.height),
            new Point(0, 0), "noCursor");
      setCursor(noCursor);
    }

    public void showCursor() {
      setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
    }
```

There isn't really any easy way to hide the cursor. We create a compatible BufferedImage (See Chapter XX for more on BufferedImage types) and use this image as the cursor without even initializing it. When the image passed to createCustomCursor() method is not initialized, Java does not display any cursor. **showCursor()** method displays the default cursor.

2.5. Terminating FSEM

In the end we want to cleanly close the FSEM and give the resources back. Here is how we implement the closeScreen() method

```
    public void closeScreen() {
      gDevice.setFullScreenWindow(null);
      dispose();
    }
```

We close the FSEM by invoking setFullScreenWindow(null), and finally release all the resources used by this FullScreen1 object and hand them back to the OS with a call to the dispose() method. In this current example we didn't change the display mode, but we could have done that (See Chapter XX on Managing Displays). Had we actually changed the display mode, then it would be a good idea to hand display back to the OS in its original mode.

Here is the complete listing of FullScreen1.java

```
    /*
     * chapter 2: FullScreen1.java
     *
     * Provides methods to switch to FSEM and
     * display text and image on the screen
     *
     */
    import java.awt.*;
    import java.awt.font.FontRenderContext;
    import java.awt.geom.Rectangle2D;
    import java.awt.image.BufferedImage;
    import javax.swing.JFrame;
    public class FullScreen1 extends JFrame {
      private static final GraphicsEnvironment gEnvironment = GraphicsEnvironment
          .getLocalGraphicsEnvironment();
```

```
private static final GraphicsDevice gDevice = gEnvironment
    .getDefaultScreenDevice();
private static final GraphicsConfiguration gConfiguration = gDevice
    .getDefaultConfiguration();

private int nBuffers = 1;
private Color bgColor = Color.BLACK;
private Color fgColor = Color.LIGHT_GRAY;
private final Font defaultFont = new Font("SansSerif", Font.BOLD, 36);

public FullScreen1() {
  super(gConfiguration);
  try {
    setUndecorated(true);
    setIgnoreRepaint(true);
    setResizable(false);
    setFont(defaultFont);
    setBackground(bgColor);
    setForeground(fgColor);
    gDevice.setFullScreenWindow(this);
    setNBuffers(nBuffers);
  }
  finally {}
}
public void setNBuffers(int n) {
  try {
    createBufferStrategy(n);
    nBuffers = n;
  } catch (IllegalArgumentException e) {
    System.err.println(
        "Exception in FullScreen.setNBuffers(): "
        +"requested number of Buffers is illegal - falling back to default");
    createBufferStrategy(nBuffers);
  }
  try{
    Thread.sleep(200);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
}
public int getNBuffers() {
  return nBuffers;
}
public void updateScreen() {

  if (getBufferStrategy().contentsLost())
    setNBuffers(nBuffers);
  getBufferStrategy().show();
```

```
}

public void displayImage(BufferedImage bi) {
  if( bi!=null){
    double x = (getWidth() - bi.getWidth()) / 2;
    double y = (getHeight() - bi.getHeight()) / 2;
    displayImage((int) x, (int) y, bi);
  }
}
public void displayImage(int x, int y, BufferedImage bi) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && bi!=null)
      g.drawImage(bi, x, y, null);
  }
  finally {
    g.dispose();
  }
}
public void displayText(String text) {
  Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
  if( g!=null && text!= null){
    Font font = getFont();
    g.setFont(font);
    FontRenderContext context = g.getFontRenderContext();
    Rectangle2D bounds = font.getStringBounds(text, context);
    double x = (getWidth() - bounds.getWidth()) / 2;
    double y = (getHeight() - bounds.getHeight()) / 2;
    double ascent = -bounds.getY();
    double baseY = y + ascent;
    displayText((int) x, (int) baseY, text);
  }
  g.dispose();
}
public void displayText(int x, int y, String text) {
  Graphics2D g =  (Graphics2D)getBufferStrategy().getDrawGraphics();
  try {
    if(g!=null && text!=null){
      g.setFont(getFont());
      g.setColor(getForeground());
      g.drawString(text, x, y);
    }
  }
  finally {
    g.dispose();
  }
}
public void blankScreen() {
    Graphics2D g = (Graphics2D)getBufferStrategy().getDrawGraphics();
    try {
      if(g!=null){
        g.setColor(getBackground());
        g.fillRect(0, 0, getWidth(), getHeight());
      }
    }
    finally {
      g.dispose();
    }
  }

  public Color getBackground(){

    return bgColor;
  }

  public void setBackground(Color bg){

    bgColor = bg;
  }

  public void hideCursor() {
    Cursor noCursor = null;
    Toolkit tk = Toolkit.getDefaultToolkit();
    Dimension d = tk.getBestCursorSize(1,1);
    if((d.width|d.height)!=0)
      noCursor = tk.createCustomCursor(
          gConfiguration.createCompatibleImage(d.width, d.height),
          new Point(0, 0), "noCursor");
    setCursor(noCursor);
  }

  public void showCursor() {
    setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
  }

  public void closeScreen() {
    gDevice.setFullScreenWindow(null);
    dispose();
  }
}
```