# Games in Java

## For programs go to

http://www.brackeen.com/javagamebook/

You will need *apache ant* to build some of these -- download 1.8.2 it at

http://ant.apache.org/

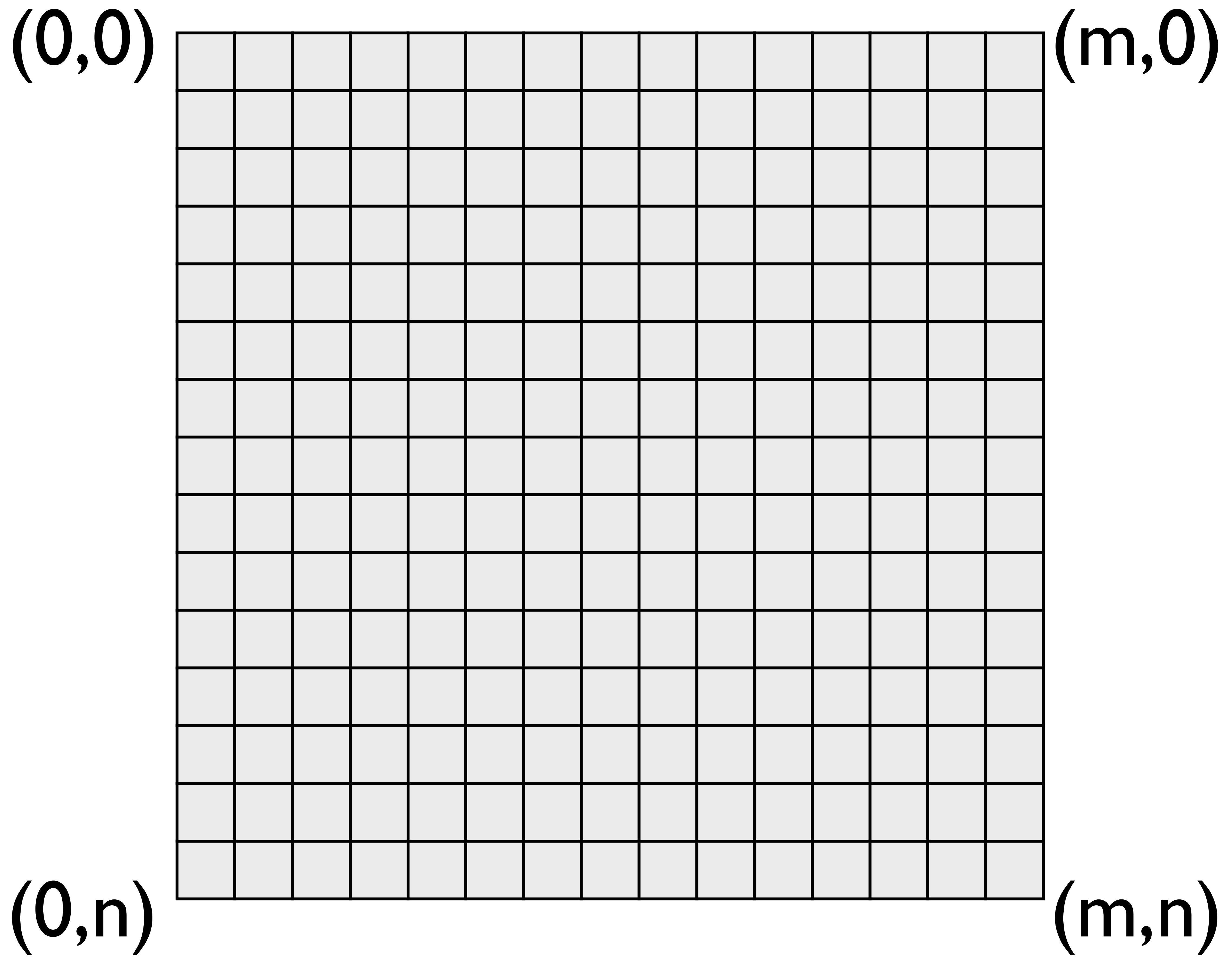
## Three ways to do Java games

1. Applet games -- run as a Java *applet* within a browser
   - Easiest for the user to run
   - Security restrictions on applets prevent full use and destruction of the computer (i.e. cannot write to files, limited internet access, etc.)
2. Windowed games -- full Java programs, escape from applet security restrictions, but have UI elements that are distracting (buttons, frames)
3. Full screen game - what we will discuss, but can do 2 for project

## Three tasks

- Create and move images
- Create and use sounds
- Create game logic that drives images and sounds

# Display properties



- Screen layouts
- Pixel color and bit depth
- Refresh rate

```
JFrame window = new Jframe();
DisplayMode displayMode = new DisplayMode(800,600,16,75);

// get the GraphicsDevice
GraphicsEnvironment environment = GraphicsEnvironment.getLocalGraphicsEnvironment();
Graphics device = environment.getDefaultScreenDevice( );

// use the JFrame as the full screen window
device.setFullScreenWindow(window);

// change the display mode
device.setDisplayMode(displayMode);

// to switch back to the previous display mode:
device.setFullScreenWindow(null);
```

This is not complete code

Some systems won't allow you to change the display mode. If so, setDisplayMode( ) throws an IllegalArgumentException.

*JFrame window = new Jframe();* This is a window object
*DisplayMode displayMode = new DisplayMode(800,600,16,75);*

**These are the display characteristics**

**These connect your program to the physical display and allows you to change its characteristics**

*// get the GraphicsDevice*
*GraphicsEnvironment environment = GraphicsEnvironment.getLocalGraphicsEnvironment( );*
*Graphics device = environment.getDefaultScreenDevice( );*

*// use the JFrame as the full screen window*
*device.setFullScreenWindow(window);*

Connects the JFrame to the display device

*// change the display mode*
*device.setDisplayMode(displayMode);*

Sets the device characteristics

*// to switch back to the previous display mode when finished:*
*device.setFullScreenWindow(null);*

## FullScreenTest.java from Brackeen's site

```
import java.awt.*;
import javax.swing.JFrame;

public class FullScreenTest extends JFrame {

    public static void main(String[] args) {

        DisplayMode displayMode;

        if (args.length == 3) {
            displayMode = new DisplayMode(
                Integer.parseInt(args[0]),
                Integer.parseInt(args[1]),
                Integer.parseInt(args[2]),
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }
        else {
            displayMode = new DisplayMode(800, 600, 16,
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }

        FullScreenTest test = new FullScreenTest();
        test.run(displayMode);
    }

    private static final long DEMO_TIME = 5000;
```

```
public void run(DisplayMode displayMode) {
    setBackground(Color.blue);
    setForeground(Color.white);
    setFont(new Font("Dialog", 0, 24));

    SimpleScreenManager screen = new SimpleScreenManager();
    try {
        screen.setFullScreen(displayMode, this);
        try {
            Thread.sleep(DEMO_TIME);
        }
        catch (InterruptedException ex) { }
    }
    finally {
        screen.restoreScreen();
    }
}


public void paint(Graphics g) {
    g.drawString("Hello World!", 20, 50);
}
}
```

```
    public void run(DisplayMode displayMode) {
        setBackground(Color.blue);
        setForeground(Color.white);
        setFont(new Font("Dialog", 0, 24));

        SimpleScreenManager screen = new SimpleScreenManager();
        try {
            screen.setFullScreen(displayMode, this);
            try {
                Thread.sleep(DEMO_TIME);
            }
            catch (InterruptedException ex) { }
        }
        finally {
            screen.restoreScreen();
        }
    }


    public void paint(Graphics g) {
        g.drawString("Hello World!", 20, 50);
    }
}
```

multithreading could be used to give responsiveness. Would need to implement Runnable.

Ensures that the screen is restored even if an exception is thrown in the try block. What happens if `halt` or `exit` are called?

```
    public void run(DisplayMode displayMode) {
        setBackground(Color.blue);
        setForeground(Color.white);
        setFont(new Font("Dialog", 0, 24));

        SimpleScreenManager screen = new SimpleScreenManager();
        try {
            screen.setFullScreen(displayMode, this);
            try {
                Thread.sleep(DEMO_TIME);
            }
            catch (InterruptedException ex) { }
        }
        finally {
            screen.restoreScreen();
        }
    }


    public void paint(Graphics g) {
        g.drawString("Hello World!", 20, 50);
    }
}
```

The Graphics object provides support for putting many shapes, text, etc. onto a screen. 20 and 50 are the location of what is painted.

Note that paint is never explicitly called! FullScreenTest is a JFrame. When a JFrame, or analogous component is run, the JFrame's paint is called. To force paint to be called again (to, for example, display something different) call repaint, which signals AWT to run paint again.

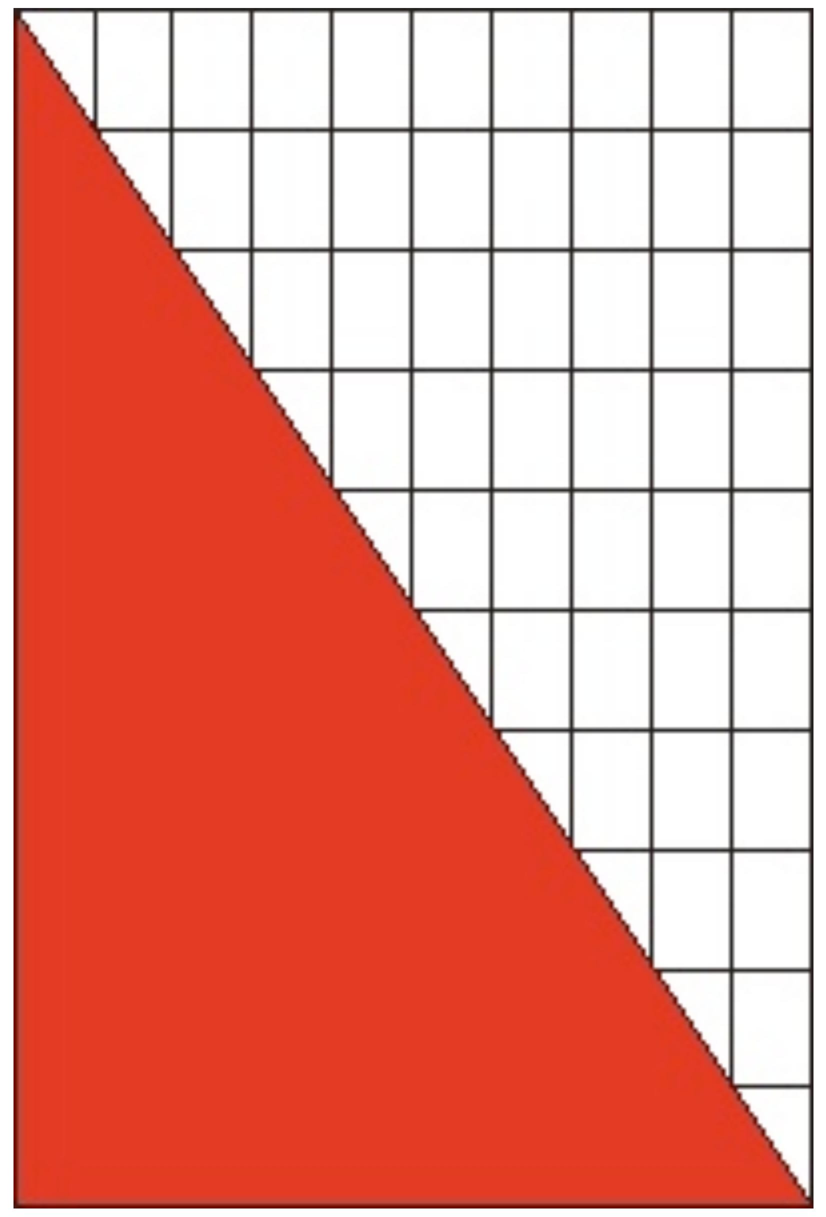
```
import java.awt.*;
import javax.swing.JFrame;
public class FullScreenTest extends JFrame {

    public static void main(String[] args) {
        DisplayMode displayMode;
        if (args.length == 3) {
            displayMode = new DisplayMode(
                Integer.parseInt(args[0]),
                Integer.parseInt(args[1]),
                Integer.parseInt(args[2]),
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }
        else {
            displayMode = new DisplayMode(800, 600, 16,
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }
        FullScreenTest test = new FullScreenTest();
        test.run(displayMode);
    }
    private static final long DEMO_TIME = 5000;
```

displayMode inherits from Java.lang.AWT and controls the display attributes.

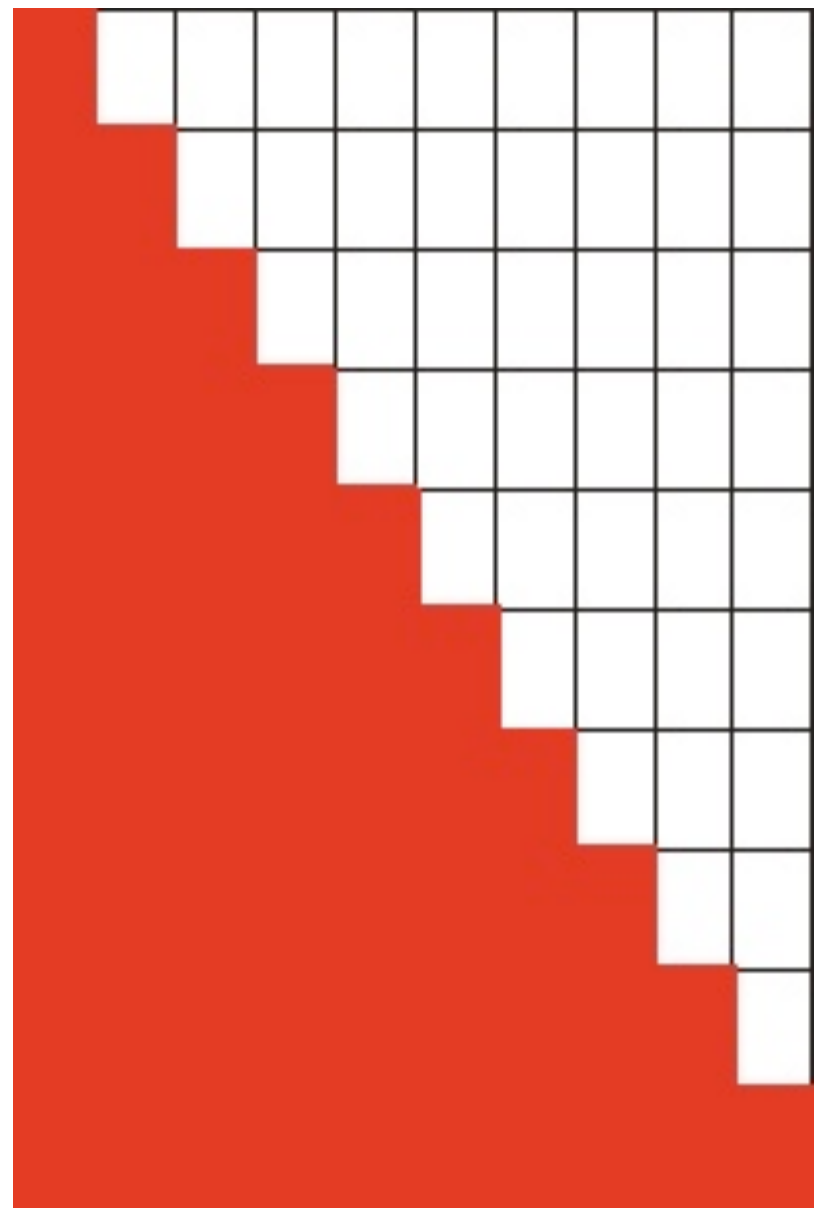
```
    public void run(DisplayMode displayMode) {
        setBackground(Color.blue);
        setForeground(Color.white);
        setFont(new Font("Dialog", 0, 24));

        SimpleScreenManager screen = new SimpleScreenManager();
        try {
            screen.setFullScreen(displayMode, this);
            try {
                Thread.sleep(DEMO_TIME);
            }
            catch (InterruptedException ex) { }
        }
        finally {
            screen.restoreScreen();
        }
    }


    public void paint(Graphics g) {
        g.drawString("Hello World!", 20, 50);
    }
}
```

SimpleScreenManager is a user defined function that will be discussed later

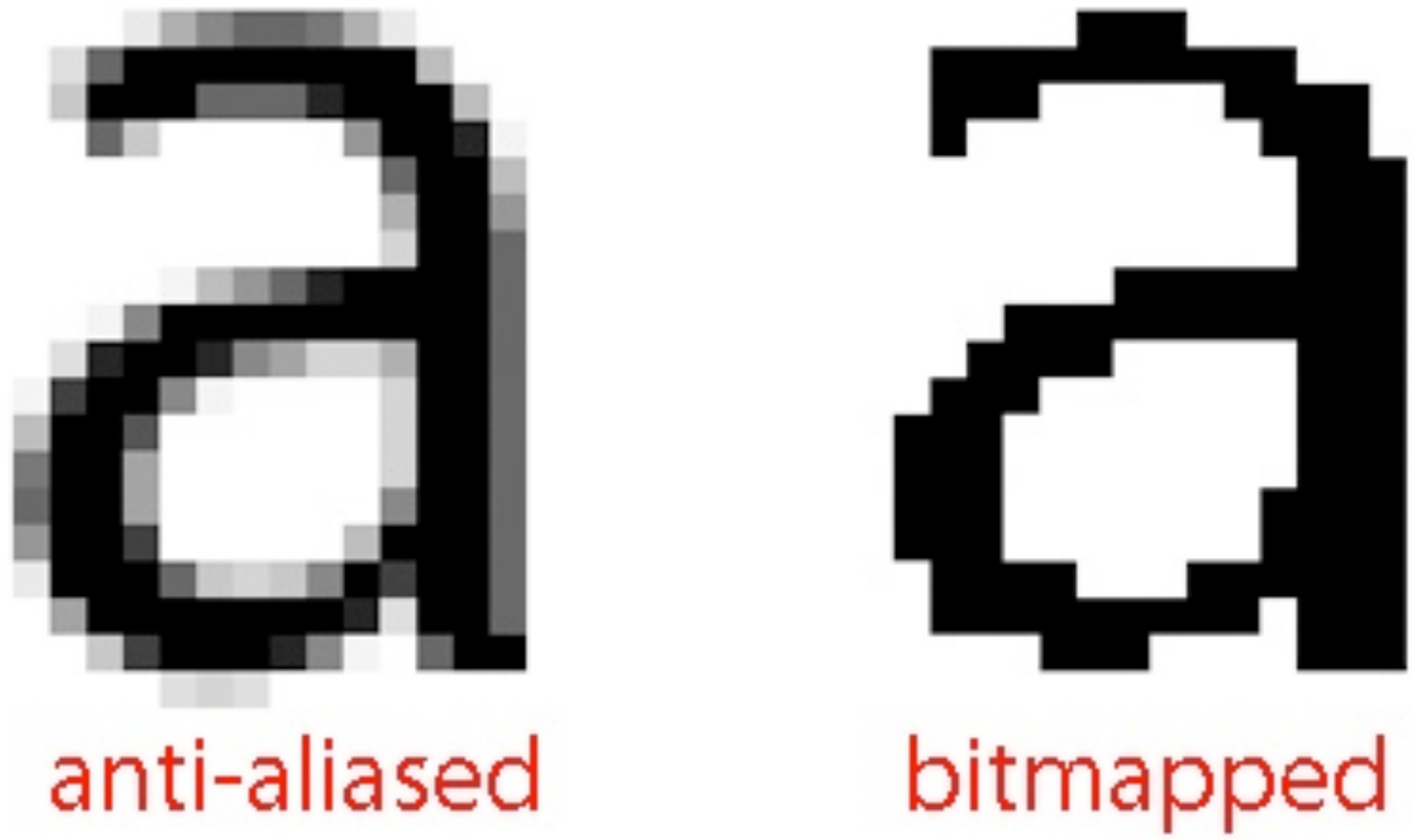
## anti-aliasing



What we would like



What a monitor provides



# Using anti-aliasing

```
public void paint(Graphics g) {
    if (g instanceof Graphics2D) {
      Graphics2D g2 = (Graphics2D) g;
      g2.setRenderingHint(
        RenderingHints.KEY_TEXT_ANTIALIASING,
        RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    }
    . . .
}
```

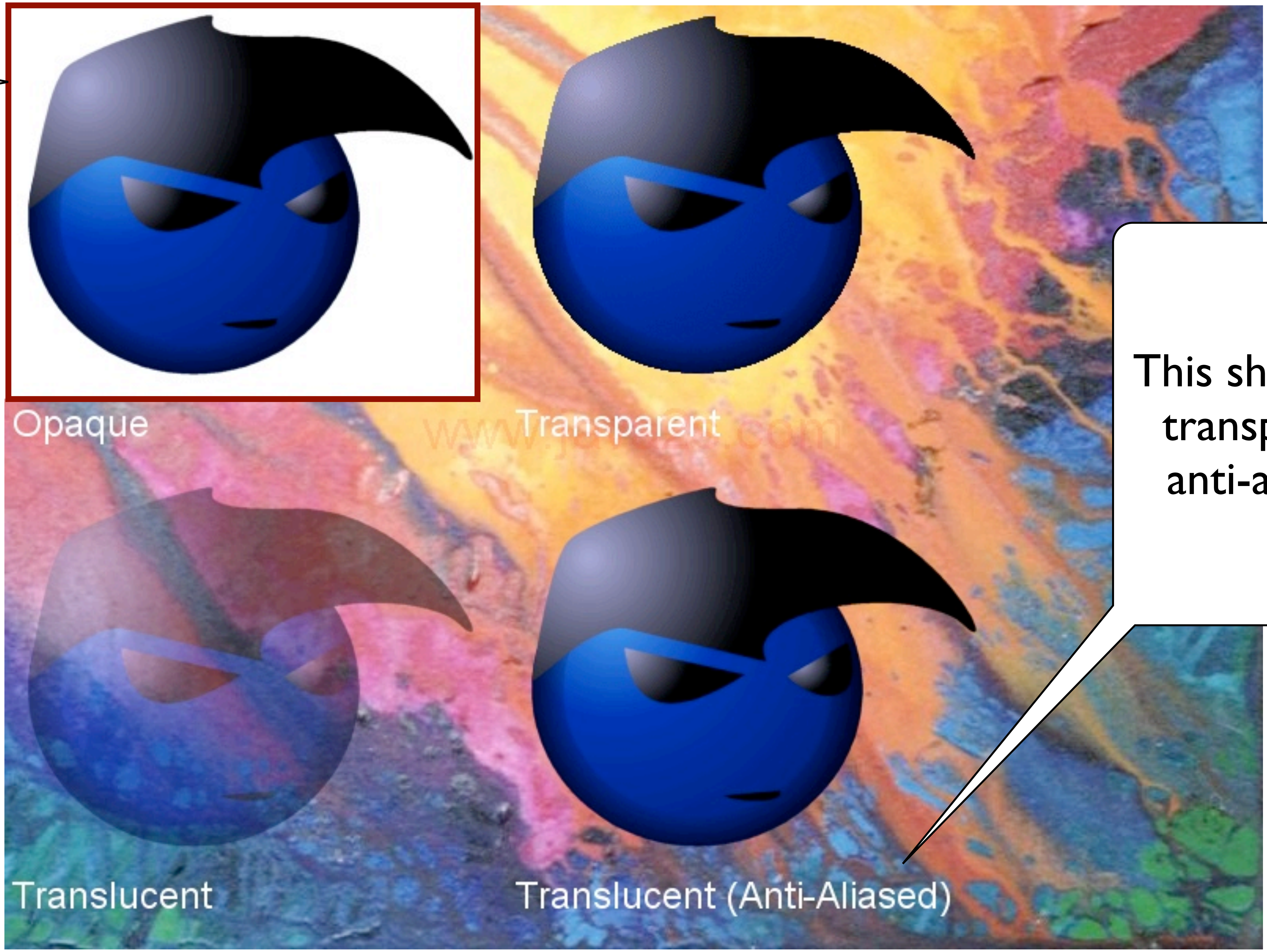
# What display mode to use

- Allow different screen resolutions, and at least two. Using the current resolution is a good idea.
- For the bit depth, 16 is faster, 24 or 32 will look better
  - if your game is black and white 16 is more than enough (but B&W will be incredibly boring)
- A refresh rate between 75 and 85 Hz is good
- DisplayMode will not allow you to set a destructive display mode

## Images

- Transparency -- i.e. is the background part of the image visible
    - Opaque: Every pixel in the image is visible
    - Transparent: Every pixel in the image is either completely visible or completely invisible.
    - Translucent: every pixel is partially transparent

The image is everything in the reddish box, not just the thing with concrete hair

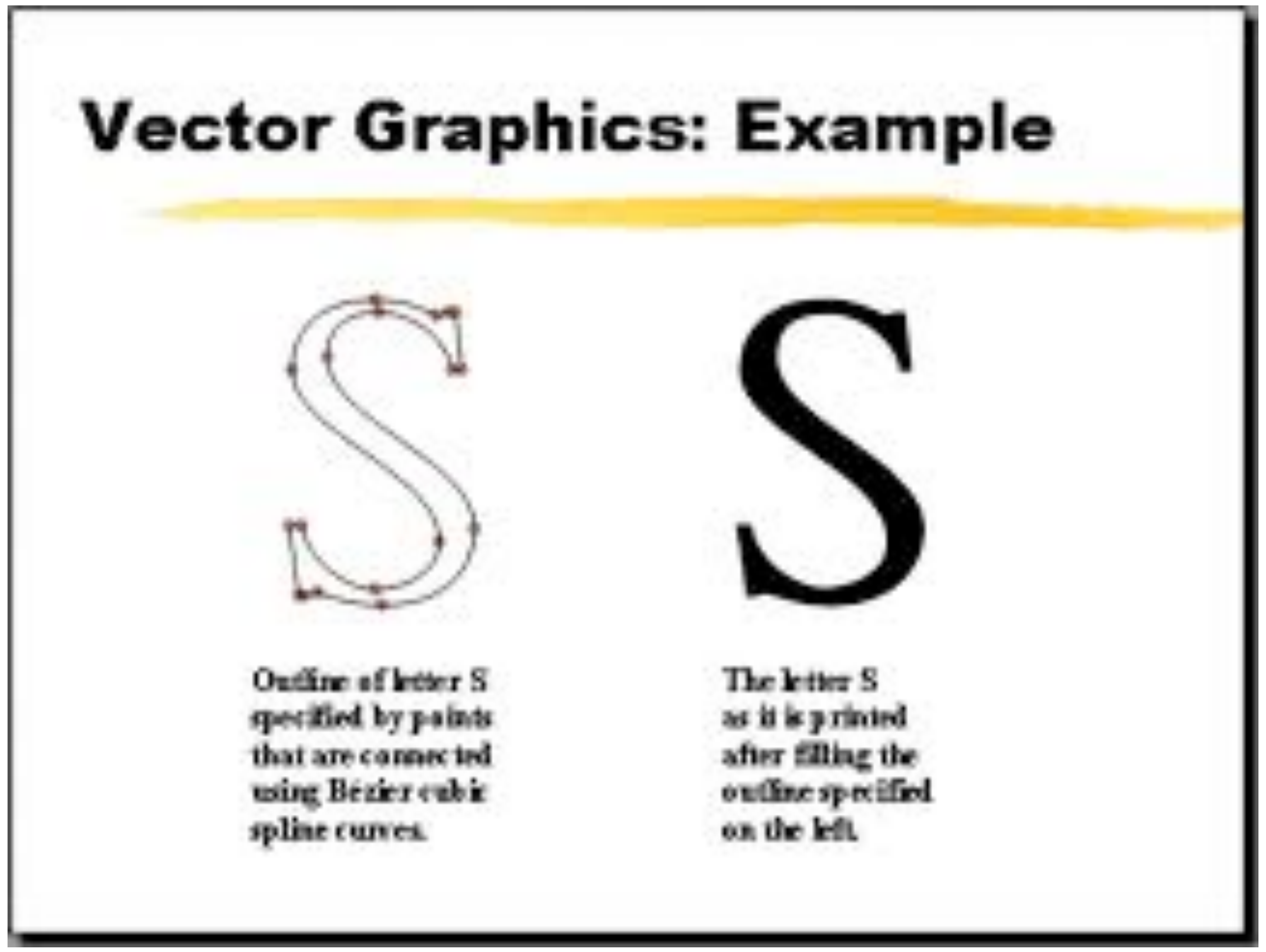


This should be transparent anti-aliased

# Image formats

- File formats
  - Vector: describes an image geometrically, and can easily be resized without loss of image quality. Use Apache's *Scalable Vector Graphics* (SVG), called Batik, available at xml.apache.org/batik
  - Raster - supported by Java
    - GIF: opaque or transparent, 8-bit color or less, PNG better
    - PNG: any kind of transparency, up to 24 bit color depth
    - JPEG: opaque, 24 bit only. High but lossy compression
  - Images can be created in Photoshop, Jasc Paint Shop Pro (www.jasc.com) or GIMP (www.gimp.org).
  - Can also get them off the web (beware of copyright restrictions.)

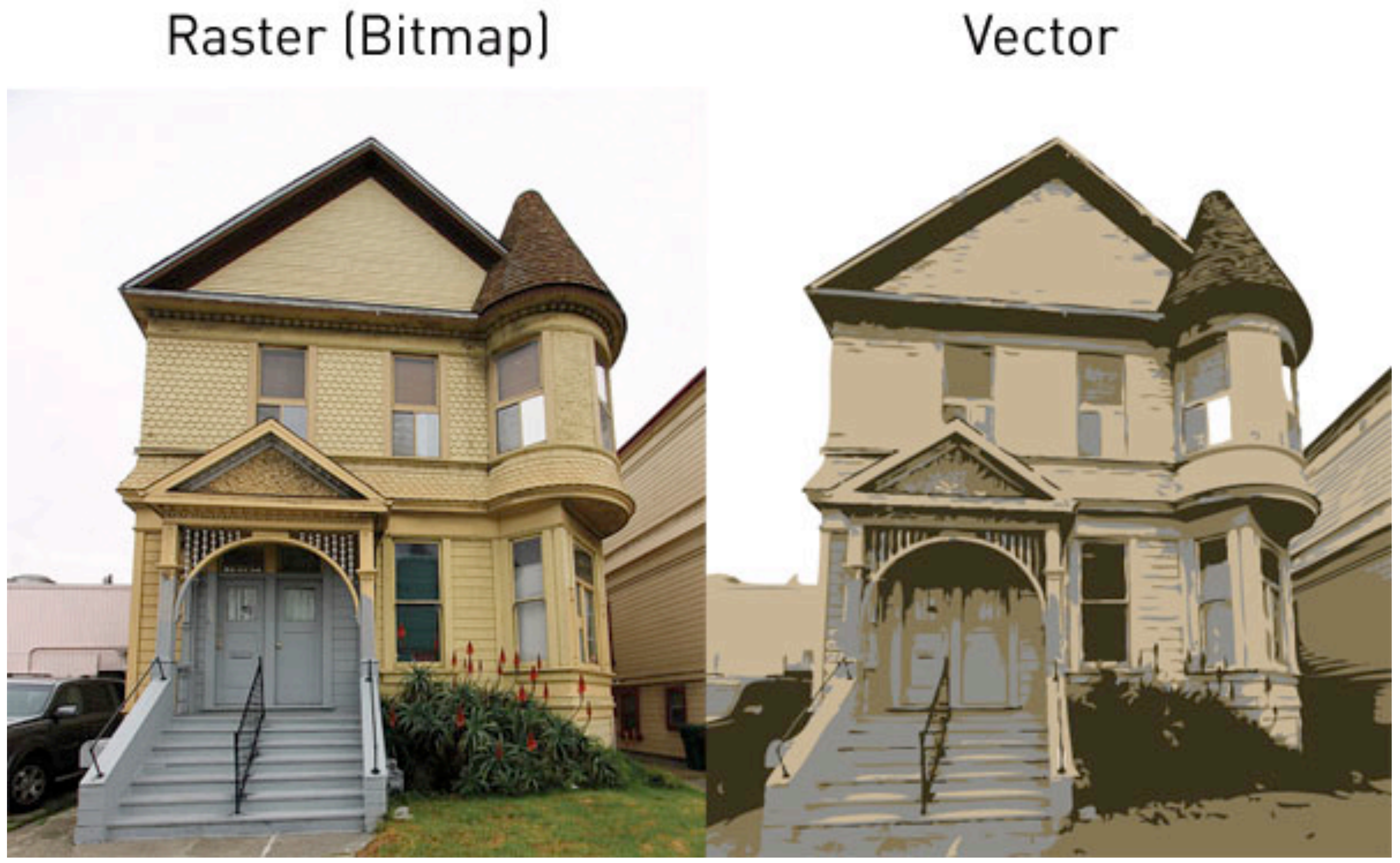
# Vector and Raster examples



Vector Graphics: Example

Outline of letter S specified by points that are connected using Bézier cubic spline curves.

The letter S as it is printed after filling the outline specified on the left.



# Making an image displayable

- Use Tookit's `getImage( )` method. It converts an image to a usable image object

```
Toolkit kit = Toolkit.getDefaultTookit( );
Image img = kit.getImage(fileName);
```

- These load and convert but do not draw the image in another thread
- Can use a *MediaTracker* object to track the image loading so that you don't display it before it is loaded
- There is an easier way …

## Displayable images, take 2

- Use the ImageIcon class to load the image.

```
ImageIcon icon = new ImageIcon(imageFile);

Image img = icon.getImage( );
```

- Look at the *ImageTest* Java program for an example of this.

# Hardware acceleration

- Java will try to hardware accelerate any image loaded using `ToolKit`'s `getImage( )`
- Three conditions will prevent this
  1. constantly changing the image (which is *not* the same as constantly changing the image's position on the screen)
  2. the image is translucent
  3. acceleration is not supported by the underlying system
- If confronted with 1 or 2 above, can use *VolatileImages* to force acceleration

# VolatileImages

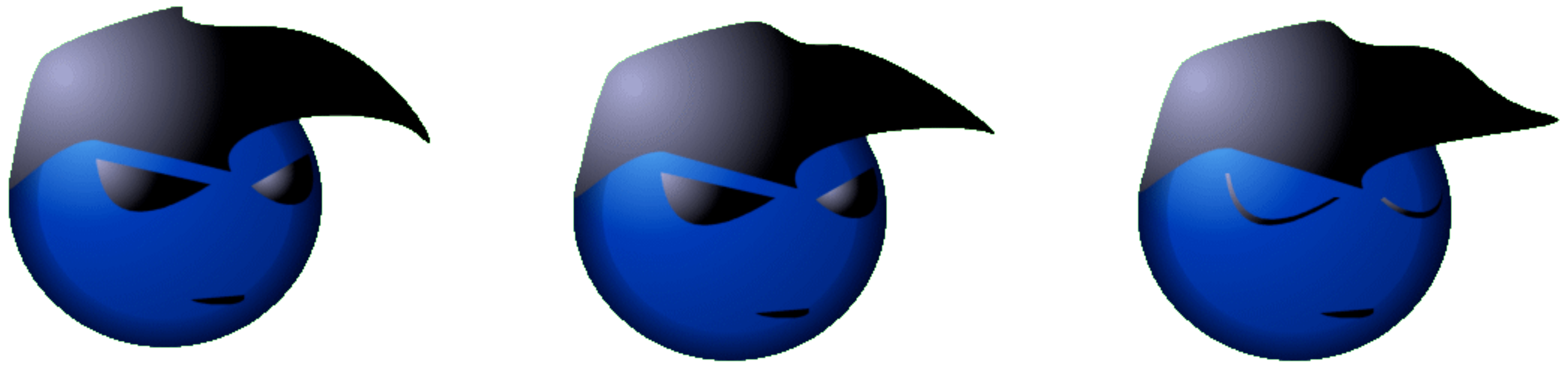
- Supported by a system class
- Refers to images whose contents may go away and need to be re-created at any time
- Make them good candidates for storing a copy in a non-volatile area, such as video ram
- When the image goes away storage is garbage collected
- Garbage collection may not happen immediately, causing un-garbage collected images to fill up video ram or other limited system resources
- *flush* method of VolatileImages class can be used to force a garbage collection

# How fast are images drawn?

- check out `ImageSpeedTest`
- On an *old* (at least 6 years old) Mac Powerbook with a 2.4GHz Intel Core Duo, running other stuff:
  - Opaque: 7847.3335 images/sec
  - Transparent: 6028.6665 images/sec
  - Translucent: 1816.6666 images/sec
  - Translucent (Anti-Aliased): 3115.3333 images/sec

# Animation

- Made of several images
- Each image can display for a different amount of time
- The result appears to move



# Animation

- Made of several images
- Each image can display for a different amount of time
- The result appears to move

## Animation

## Animation

# Animation

# Animation

## Animation

## Animation

# Animation.java file

```
private ArrayList frames;
. . .

    /**
        Creates a new, empty Animation.
    */
    public Animation() {
        frames = new ArrayList();
        totalDuration = 0;
        start(); // not Thread.start( )
    }            // initializes counters

    /**
        Adds an image to the animation with the specified
        duration (time to display the image).
    */
     public synchronized void addFrame(Image image,
         long duration)
     {
         totalDuration += duration;
         frames.add(new AnimFrame(image, totalDuration));
     }
```

Sets up a sequence of frames to be animated.
There are three important methods

AddFrame adds the frames to a *container* holding all of the frames in the animation

```java
/**
   Updates this animation's current image (frame), if
   neccesary.
*/
public synchronized void update(long elapsedTime) {
    if (frames.size() > 1) {
        animTime += elapsedTime;

        if (animTime >= totalDuration) {
             animTime = animTime % totalDuration;
            currFrameIndex = 0;
        }

         while (animTime > getFrame(currFrameIndex).endTime) {
             currFrameIndex++;
         }
    }
}
```

## update

determines the image that is to be called next based on the current time

```
/**
    Gets this Animation's current image. Returns
    null if this animation has no images.
*/
public synchronized Image getImage() {
    if (frames.size() == 0) {
        return null;
    }
    else {
        return getFrame(currFrameIndex).image;
    }
}
```

## getImage

returns the image identifed in update

## putting new images on the screen

- Could use the *paint* method
  - relies on the AWT thread to actually draw the image onto the screen
  - The AWT thread might be busy doing something else causing a delay and jerky animation
- Using *Active Rendering* is a way to get around this
  - thread that would have called paint/repaint draws the image directly on the screen
  - more control over when the screen actually gets drawn

## An example of *active rendering*

```
Graphics g = screen.getFullScreenWindow( ).getGraphics;
draw(g);
g.dispose( );
```

screen is a screen such as has been defined previously

g contains the graphics context for the screen

when done drawing you need to get rid of the graphics context g or otherwise a memory leak will occur.

## Performing an animation

(see AnimationTest1 for the full code)

```java
public void animationLoop() {
    long startTime = System.currentTimeMillis();
    long currTime = startTime;

    while (currTime - startTime < DEMO_TIME) {
        long elapsedTime =
            System.currentTimeMillis() - currTime;
        currTime += elapsedTime;

        // update animation
        anim.update(elapsedTime);

        // draw to screen
        Graphics g =
            screen.getFullScreenWindow().getGraphics();
        draw(g);
        g.dispose();

        // take a nap
        try {
            Thread.sleep(20);
        }
        catch (InterruptedException ex) { }
    }
}
```

```
public void animationLoop() {
    long startTime = System.currentTimeMillis();
    long currTime = startTime;

    while (currTime - startTime < DEMO_TIME) {
        long elapsedTime =
            System.currentTimeMillis() - currTime;
        currTime += elapsedTime;

        // update animation
        anim.update(elapsedTime);

        // draw to screen
        Graphics g =
            screen.getFullScreenWindow().getGraphics();
        draw(g);
        g.dispose();

        // take a nap
        try {
            Thread.sleep(20);
        }
        catch (InterruptedException ex) { }
    }

}
```

Get time since the last character was displayed

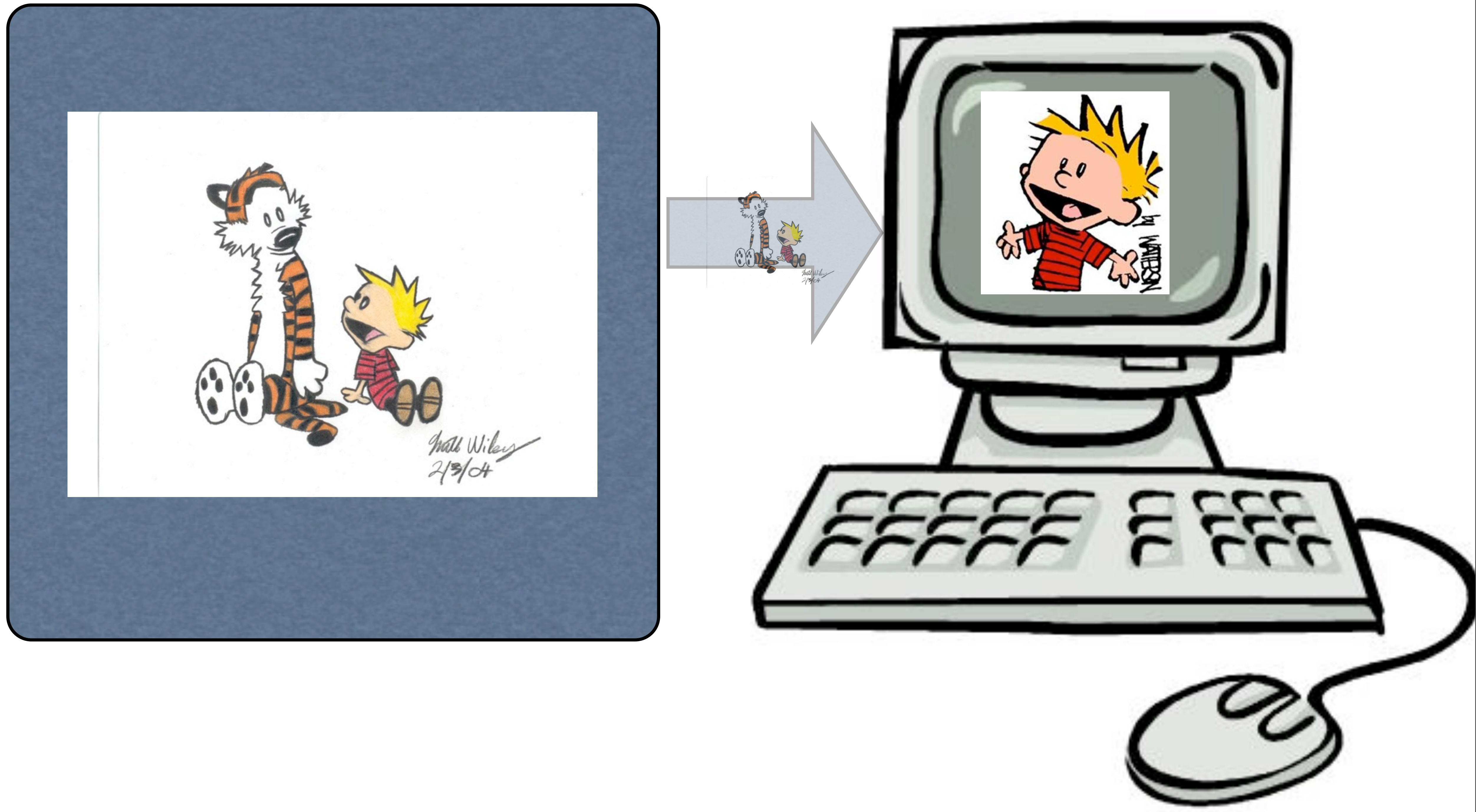
Select the image to display based on the time

This actually gets the image (using the *getImage( )* method to display at this point in time from the AnimationTest class, and draws it to the graphics context g.

Pause between displaying images in milliseconds. 20 yields ~50 frames/second

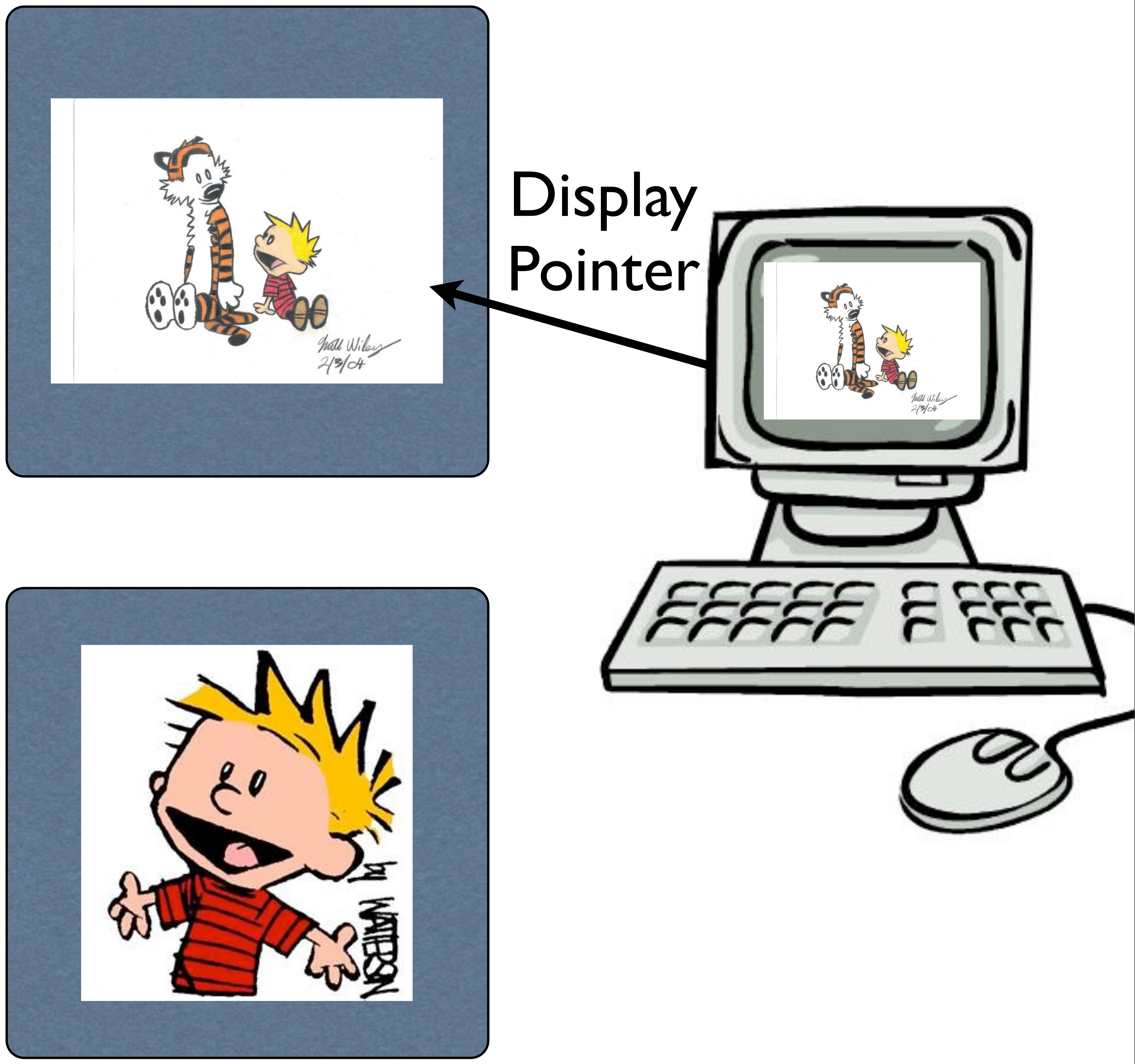
# Double buffering

- When you run AnimationTest1.java, you may notice flickering during image updates
- This is because the draw routine erases the character & background, and then redraws the character
- A better way to do this is to use *double buffering* to update a new screen image, and then switch the display to that



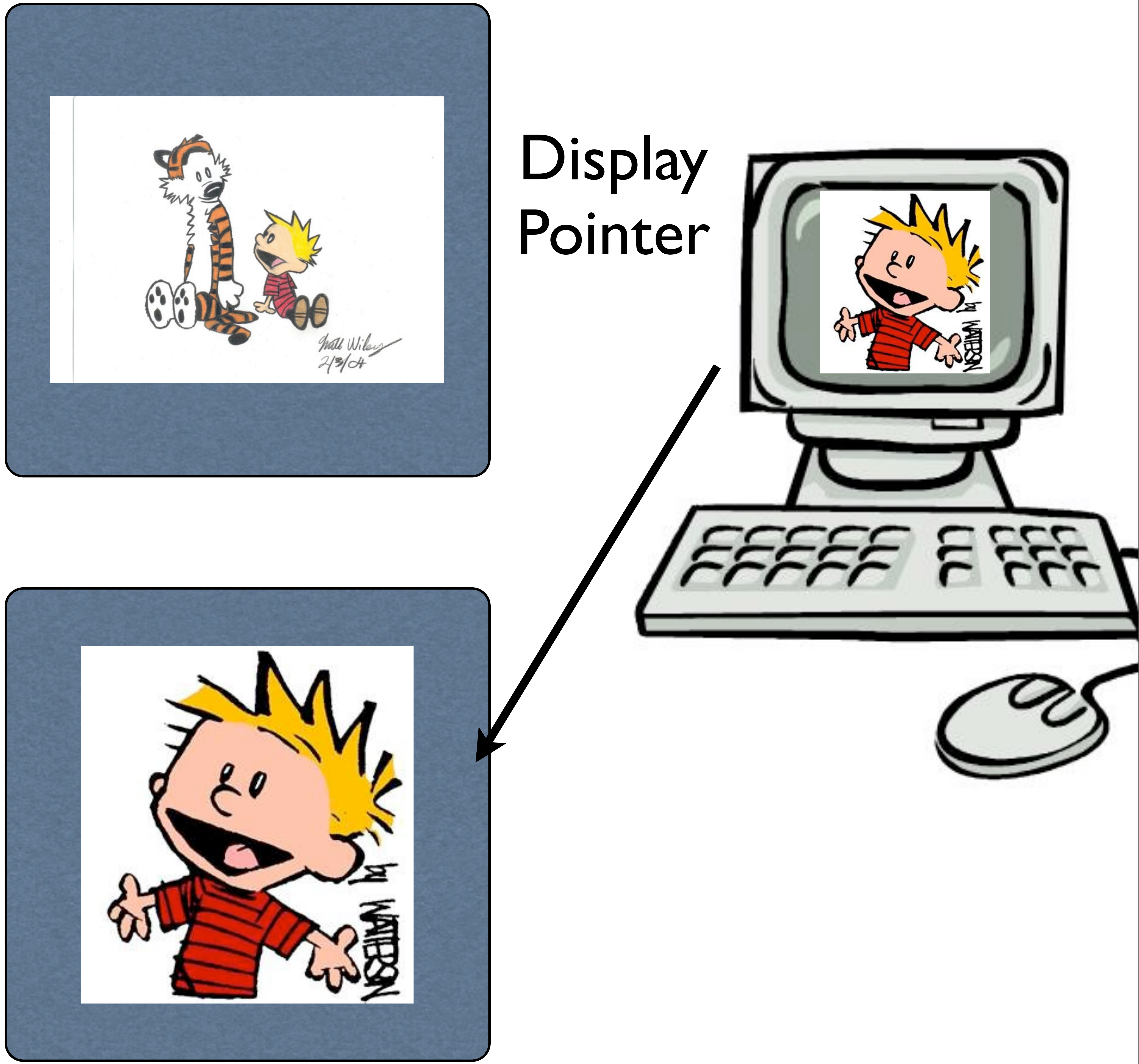
# page flipping

- Double buffering still requires the buffer to be copied into the display
  - this appears better than painting it onto the screen
  - still requires a lot of data movement - an 800x600x16 bit color depth screen requires 938KB of data. Almost a MB, 30 or more times/second
- Page flipping is a way around this



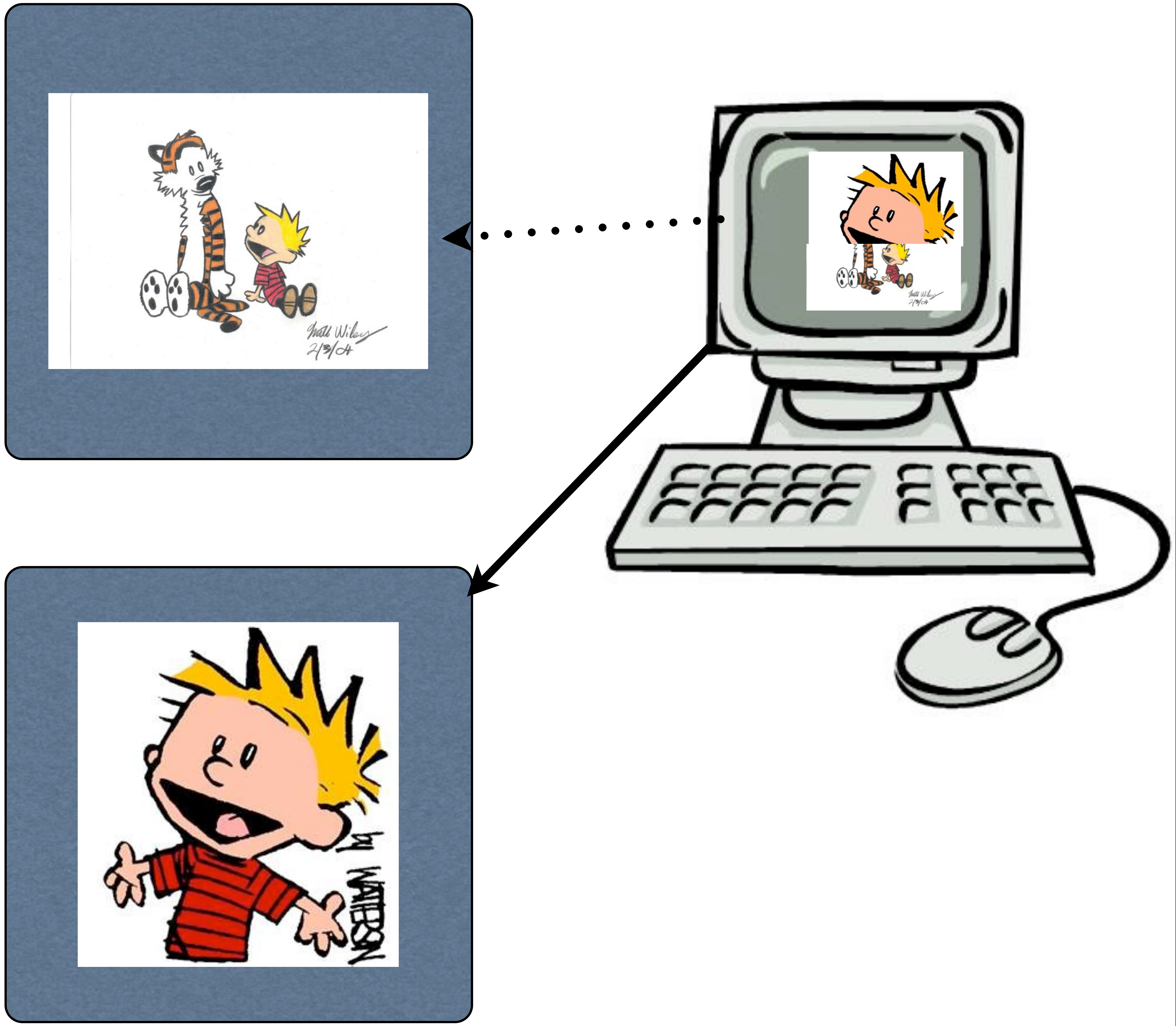
# page flipping

- The image is displayed out of the buffer pointed to by the display pointer.
- Only the pointer is changed to change the image



# Monitor refresh and tearing

- Our monitor is being refreshed at about 75 times a second
- What happens if the display pointer is reset during a refresh, or a buffer is copied during a refresh?
- Part of the display has the old image, part the new, for a fraction of a second. This is called *tearing*.
- If it happens frequently, our image becomes hideous.



```
public void setFullScreen(DisplayMode displayMode) {
    final JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setUndecorated(true);
    frame.setIgnoreRepaint(true);
    frame.setResizable(false);

    device.setFullScreenWindow(frame);

    if (displayMode != null &&
        device.isDisplayChangeSupported())
    {
        try {
            device.setDisplayMode(displayMode);
        }
        catch (IllegalArgumentException ex) { }
        // fix for mac os x
        frame.setSize(displayMode.getWidth(),
            displayMode.getHeight());
    }
    // avoid potential deadlock in 1.4.1_02
    try {
        EventQueue.invokeAndWait(new Runnable() {
            public void run() {
                frame.createBufferStrategy(2);
            }
        });
    }
    catch (InterruptedException ex) {
        // ignore
    }
    catch (InvocationTargetException  ex) {
        // ignore
    }
}
```

Use double buffering and page flipping

## When you look at ScreenManager.java

- Look at how
  - BufferStrategy is used to enable double buffering
  - getGraphics is used to get the graphics context
  - update is used to update the display
  - getCompatibleDisplayModes is used to determine the compatible modes
  - getCurrentDisplayMode is used to get the current display mode
  - findFirstCompatibleMode to get the first compatible mode from a list of compatible modes.

## Sprites

- Animations are all fine and good, but we would really like things to move about (rather than animate at a fixed location on) the screen
- Sprites are a way to do this
  - Sprites are graphic elements that move independently (from other graphics) about the screen
  - Sprites have three components: (1) an animation; (2) a position; (3) a velocity
    - Velocities have two components, specified in pixels/millisecond:

    1. a horizontal component
    2. a vertical component

## Sprites

- Velocities have two components, specified in pixels/millisecond:
  1. a horizontal component
  2. a vertical component
- Specifying a sprite this way gives machine independence
  - If we specified pixels/frame, for example, a faster machine with a faster refresh rate would cause the image to move across the screen faster. Would prefer the speed give smoothness.
  - Early computer games had this issue
- Look at the code in Sprite.java (which shows how to define a Sprite class, and SpriteTest1.java, which shows how to use sprites.

## More fun with sprites

- You can use more sprites -- just create and draw them
- You can do transforms on your images on the fly

```
AffineTransform transform = new AffineTransform( );
transform.scale(2,2);
transform.translate(100,100); // needed to get it in the right place
                              // after scaling
g.drawImage(image, transform, null);
```

- Look at SpriteTest2.java in Brackeen as an example of how to create a sprite

## Games in Java - Interactivity and user interfaces

*Programs are in ch03src*

## First, Java packages and file organization

- In the ch03src/src/ directory, there are com/brackeen/javagamebook/graphics com/brackeen/javagamebook/input com/brackeen/javagamebook/test directories
- This is a package, and is a way of organizing files in Java
  - These directories contain code to be reused
  - One-off code is in the top src directory

```
package com.brackeen.javagamebook.graphics;

import java.awt.Image;
import java.util.ArrayList;
```

# Packages allow an additional granularity of protection

| **Modifier** | public | protected | *no modifier* | private |
|---|---|---|---|---|
| **Class** | Y | Y | Y | Y |
| **Package** | Y | Y | Y | N |
| **Subclass** | Y | Y | N | N |
| **World** | Y | N | N | N |

# Input devices

- "Official" java libraries support keyboard and mouse inputs
- There is no official support for joysticks in Java, but unofficial packages can be found at sourceforge.net/projects/javajoystick/

| **Modifier** | public | protected | *no modifier* | private |
|---|---|---|---|---|
| **Class** | Y | Y | Y | Y |
| **Package** | Y | Y | Y | N |
| **Subclass** | Y | Y | N | N |
| **World** | Y | N | N | N |

# GameCore.java

- Contains a game core to test additional capabilities
- The update and init methods don't do much, but will be overridden in other classes to do more useful work.
- Finally, use Apache Ant to compile this code when necessary.

## The AWT Event Model

- AWT has a thread devoted to dispatching events
- Receives mouse clicks, key presses, etc. from the OS
- When AWT receives an event, it looks to see if there is a *listener* for that event
  - E.g., key press listeners implement the *KeyListener* interface

## AWT event dispatch steps

1. A key is pressed
2. OS sends key event to the Java runtime
3. The Java runtime puts the event in the AWT event queue
4. The AWT event dispatch thread dispatches the event to all KeyListener(s)
5. The KeyListener(s) process the event using user code.

## Keyboard Input

- To capture keyboard events
  - create a KeyListener by instantiating an object from a class that implements the KeyListener interface
  - register the listener
- A KeyListener requires three methods:
  - keyTyped: not particularly interesting for us
  - keyPressed: called when a key is pressed
  - keyReleased: called when a key is released

```java
public class KeyTest extends GameCore implements KeyListener {

    public static void main(String[] args) {
        new KeyTest().run();
    }

    private LinkedList messages = new LinkedList();

    public void init() {
        super.init();

        Window window = screen.getFullScreenWindow();

        // allow input of the TAB key and other keys normally
        // used for focus traversal
        window.setFocusTraversalKeysEnabled(false);

        // register this object as a key listener for the window
        window.addKeyListener(this);

        addMessage("KeyInputTest. Press Escape to exit");
    }
    //   KeyPressed code here
 }
```

Adds the GameCore/ KeyListener object whose *init( )* method is being called to a list of objects wanting to know about keyboard events.

# One of the methods required by the KeyListner interface

```
// a method from (required by) the KeyListener interface
    public void keyPressed(KeyEvent e) {
        int keyCode = e.getKeyCode();

        // exit the program
        if (keyCode == KeyEvent.VK_ESCAPE) {
             stop();
        }
        else {
            addMessage("Pressed: " +
                KeyEvent.getKeyText(keyCode));

            // make sure the key isn't processed for anything else
            e.consume();
        }
```

A KeyEvent object describes the keyboard event.

# The KeyEvent parameter

- keyPressed, keyReleased and keyTyped all take a KeyEvent as a parameter
- Keys are defined in the form of KeyEvent constants such as KeyEvent.VK_xxx.
  - A "Q" is KeyEvent.VK_Q
  - Most events can be guessed (VK_3, e.g.) but confirm with the Java documentation for the KeyEvent class to be sure
  - Look at the code in KeyTest.java for examples of listening to key input events.

## Some quirks

- In *init( )*, *window.setFocusTraversalKeysEnabled(false);* disables focus traverse keys (e.g. tabs and so forth) and allows them to be examined like any other key. Otherwise they are lost
- The Alt key can cause problems
  - Normally used to specify a *mnemonic*, i.e. Alt+F often activates the file menu
  - AWT assumes these will be grabbed by a window, not the program running in the window, but this is not the case in a full screen game where we want to grab them
  - calling e.consume( ), where e is a KeyEvent, will process Alts and the following key like any other key.
- Keys sometimes behave differently on different systems

# Dual and single-mode interfaces

- A single-mode interface is like *Emacs* - all keys/key pairs have a defined meaning at all times,
  - some are input
  - some are commands
- A dual-mode interface is like *vi* - two modes:
  - *input* mode: all keys are input, and, e.g., ctl-f places a ctl-f character into the file
  - *output* mode: all keys are commands (valid or otherwise), and, e.g. , ctl-f moves the cursor down the screen
- Issues with Alt result from the UI being treated as single-mode

# Mouse Input

- Three types of mouse events
  - Mouse button clicks
  - Mouse motion
  - Mouse wheel
- Mouse buttons act like keyboard buttons, but without repeat
- Motion is given in $x, y$ coordinates
- Wheel event say how far the wheel was scrolled

## Mouse listeners -- three kinds

- MouseListener
- MouseMotionListener
- MouseWheelListener
- Each takes a MouseEvent as a parameter

# MouseListener

- methods for detecting presses, releases and clicks (press and release)

```
// from the MouseListener interface
public void mousePressed(MouseEvent e) {
    trailMode = !trailMode;
}
// from the MouseListener interface
public void mouseReleased(MouseEvent e) {
    // do nothing
}
```

```
// from the MouseListener interface
public void mouseClicked(MouseEvent e) {
    // called after mouse is released - ignore it
}
```

When multiple mouse buttons are pressed, each press, release, and click results in a separate event.
For example, if the user presses **button 1** followed by **button 2**, and then releases them in the same order, the following sequence of events is generated:

| **id** | **modifiers** | **button** |
|---|---|---|
| MOUSE_PRESSED: | BUTTON1_MASK | BUTTON1 |
| MOUSE_PRESSED: | BUTTON2_MASK | BUTTON2 |
| MOUSE_RELEASED: | BUTTON1_MASK | BUTTON1 |
| MOUSE_CLICKED: | BUTTON1_MASK | BUTTON1 |
| MOUSE_RELEASED: | BUTTON2_MASK | BUTTON2 |
| MOUSE_CLICKED: | BUTTON2_MASK | BUTTON2 |

## MouseTest program

- MouseTest moves "Hello World" around to follow the mouse.
- When a button is pushed, a linked list with count elements of positions is maintained
- Each time the mouse is moved, "Hello World" is displayed at the last count positions of the mouse.

## Relative mouse motion

- MouseTest follows *absolute* mouse motion
- When the mouse moves the cursor to the edge of the screen further movement has no effect (cursor stuck at the edge)
- What if you want to do Mouselook-style movement where the player looks in a direction based on that movement.
  - A sequence of mouse moves to look left, right, left, etc., can move you off of the keypad
- Relative mouse movement is not directly supported

## How to do relative movement

- The mouse starts off in the center of the screen -- we want to return it there after each movement

1. Mouse starts in the screen center
2. User moves the mouse, calculate how much and where it moved
3. Send an event to put the mouse back in the center
4. Ignore the recenter event when detected

# Mouselook - detect motion

```
// from the MouseMotionListener interface
public synchronized void mouseMoved(MouseEvent e) {
    // this event is from re-centering the mouse - ignore it
    if (isRecentering &&
        centerLocation.x == e.getX() &&
        centerLocation.y == e.getY())
    {
        isRecentering = false;
    }
    else {
        int dx = e.getX() - mouseLocation.x;
        int dy = e.getY() - mouseLocation.y;
        imageLocation.x += dx;
        imageLocation.y += dy;
        // recenter the mouse
        if (relativeMouseMode) {
            recenterMouse();
        }

    }

    mouseLocation.x = e.getX();
    mouseLocation.y = e.getY();

}
```

```
/**
    Uses the Robot class to try to position the mouse in the
    center of the screen.
    Note that use of the Robot class may not be available
    on all platforms.
*/
private synchronized void recenterMouse() {
    Window window = screen.getFullScreenWindow();
    if (robot != null && window.isShowing()) {
        centerLocation.x = window.getWidth() / 2;
        centerLocation.y = window.getHeight() / 2;
        SwingUtilities.convertPointToScreen(centerLocation,
            window);
        isRecentering = true;
        robot.mouseMove(centerLocation.x,
centerLocation.y);
    }
}
```

## Cursors

- Java API defines several cursors
  - CROSSHAIR_CURSOR (a thin plus sign)
  - DEFAULT_CURSOR (the normal arrow)
  - HAND_CURSOR (the normal grabby hand cursor)
  - TEXT_CURSOR (usually I shaped)
  - WAIT_CURSOR (hourglass)
- What if we don't want a cursor?

# Create a custom cursor

- Do this by calling the createCustomCursor Toolkit method

```
Cursor invisibleCursor =
    Toolkit.getDefaultToolkit( ).createCustomCursor(
        Toolkit.getDefaultToolkit( ).getImage(""),
                new Point(0, 0),
                "invisible");
```

image for the cursor (nothing in this case)

The cursor *hotspot*, i.e. the point of the point on an extended cursor that says where the cursor is

A name for Java accessibility purposes

# Creating an input manager

- Look at the program GameAction.java (which uses InputManager.java)
- Some things we want to do
  - Handle all inputs (mouse, keyboard) at one point in the game loop
  - Set boolean variables in the game when events happen
  - For some actions (i.e. jump) take it when the key is pressed, for others (movement) do over time
- Can let user reconfigure keyboard, current InputManager does not do this
- It does handle all the above events when *it* wants to
  - detect initial press for some keys, whether held down for others
  - maps keys to generic game actions, such as spacebar ⇒ jump
  - (future) let programmer remap keys

# Other actions - pause

- Pausing the game - basically ignoring any input except for key that stops the pause (i.e., resumes the game)
- modify the animation loop:

```
if (!paused) {
        checkInput( );
        updateGameObjects( );
 } else { checkForUnpause( );}
```

- Look at Player.java

## Other actions - gravity

- Want a jump to give an initial velocity
- decrease this velocity by a downward gravitational force

```
velocityY = velocityY + GRAVITY * elapsedTime;
```

- Real gravitational constant may be too big - tune it to look right.  In the game in the book it is 0.002.
- Look at Player.java

- InputManagerTest.java implements a simple game
- Player can jump (spacebar), move (arrow keys), pause ("P"), exit (Escape)

# User interface design tips

- Keep it simple, and at any time only show needed options
- Make options easy to access
- Use *Tooltips* -- popups that say what an option does
- Give a response to every action
- Test interfaces on another person -- ask what might be easier, what is confusing, don't cut them off with "that's hard, that won't work ..."
- Be prepared to change the UI if necessary
- Use Swing components

## Swing overview

- It is the main Java widget toolkit
- Swing components ***are not*** thread safe
- Most UI elements are a class, i.e. JButton okButton = new JButton("OK");
- JLabel, JTextField, etc. supported
- JFrame frame = screen.getFullScreenWindow( );
- frame.getContentPane( ).add(okButton);
- okButton.setLocation(200, 100) to locate buttons -- or use a *layout manager*
- Uses the standards of the layout manager to position buttons

# Swing in full screen mode

- JFrame supports layered panes -- i.e. components appear in different layers. This allows tool tips to appear above buttons and be readable
- To draw all Swing components use paintComponents method( ).

```
public void draw(Graphics2D g) {
    super.draw(g);
    JFrame frame = super.screen.getFullScreenWindow();

    // the layered pane contains things like popups (tooltips,
    // popup menus) and the content pane.
    frame.getLayeredPane().paintComponents(g);
}
```

# Two Challenges

- Content pane draws the background which hides what is behind it

```
// make sure the content pane is transparent
if (contentPane instanceof JComponent) {
    ((JComponent) contentPane).setOpaque(false);
}
```

- Swing normally renders its own components -- this can cause tearing, etc.
- All repaint requests go to a RepaintManager in library code -- override this in the library with

```
// make sure Swing components don't paint themselves
NullRepaintManager.install();
```

# A simple menu

- Look at MenuTest.java which extends the InputManagerTest routine.

  Create a "listener" to detect button pushes

```
/**
    Called by the AWT event dispatch thread
    when a button is pressed.
*/
public void actionPerformed(ActionEvent e) {
    Object src = e.getSource();
    if (src == quitButton) {
        // fire the "exit" gameAction
        super.exit.tap();
    }
}
```

```
        else if (src == configButton) {
            // doesn't do anything (for now)
            configAction.tap();
        }
        else if (src == playButton || src ==
pauseButton) {
            // fire the "pause" gameAction
            super.pause.tap();
        }
```

## Configuring keyboards

- Check out KeyConfigTest.java

## Sound Effects and Music

*Chapter 4 of Developing Games in Java*

# Opening sounds files

- Use the AudioSystem class
- getAudioStream methods used to get an AudioInputStream object from a file

```
File file = new File("sound.wav");
AudioInputStream stream = AudioSystem.getAudioInputStreamFile(file);
AudioFormat format = stream.getFormat( );
```

- AudioFormat class provides a way to get info about the stream -- sample rate, number of channels, bytes per sample, etc.
  - Useful for determining memory requirements, i.e. a 1 second mono sound with 16-bit samples, 44,100KHz would be $44{,}100 \times 1 \times 2 = {\sim}88K$ bytes.

# Lines

- A *line* is an interface to send audio to the sound sub-system
- The Line interface supports several sub-interfaces
    - We use a SourceDataLine
    - Allows audio data to be written to the sound system
    - See Chapter 4 of *Game Development in Java* for a discussion of clips. Clips are more limited than SourceDateLines in the number that can be open and the number of sounds they can play at a time

# Playing a sound

- Look at program SimpleSoundPlayer.java
- Loads sample from AudioInputStream
- Converted to an InputStream

samples is a byte array

```
/**
    Opens a sound from a file.
*/
public SimpleSoundPlayer(String filename) {
    try {
        // open the audio input stream
         AudioInputStream stream =

AudioSystem.getAudioInputStream(
              new File(filename));
```

```
       format = stream.getFormat();

        // get the audio samples
        samples = getSamples(stream);
    }
    catch (UnsupportedAudioFileException ex) {
        ex.printStackTrace();
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

# getSamples

```
/**
    Gets the samples from an AudioInputStream as an array
    of bytes.
*/
private byte[] getSamples(AudioInputStream audioStream) {
    // get the number of bytes to read
     int length = (int)(audioStream.getFrameLength() * format.getFrameSize());

    // read the entire stream
    byte[] samples = new byte[length];
    DataInputStream is = new DataInputStream(audioStream);
    try {
        is.readFully(samples);
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }

    // return the samples
    return samples;
}
```

At this point the sound samples reside in the Java program

# readFully( )

Reads some bytes from an input stream and stores them into the buffer array b. The number of bytes read is equal to the length of b.

This method blocks until one of the following conditions occurs:

- b.length bytes of input data are available, in which case a normal return is made.
- End of file is detected, in which case an EOFException is thrown.
- An I/O error occurs, in which case an IOException other than EOFException is thrown.

If b is null, a NullPointerException is thrown. If b.length is zero, then no bytes are read. Otherwise, the first byte read is stored into element b[0], the next one into b[1], and so on. *If an exception is thrown from this method, then it may be that some but not all bytes of b have been updated with data from the input stream.*

```java
/**
       Plays a stream. This method blocks (doesn't return) until
       the sound is finished playing.
   */
   public void play(InputStream source) {

       // use a short, 100ms (1/10th sec) buffer for real-time
       // change to the sound stream
       int bufferSize = format.getFrameSize() *
           Math.round(format.getSampleRate() / 10);
       byte[] buffer = new byte[bufferSize];

       // create a line to play to
       SourceDataLine line;
       try {
           DataLine.Info info =
               new DataLine.Info(SourceDataLine.class, format);
           line = (SourceDataLine)AudioSystem.getLine(info);
           line.open(format, bufferSize);
       }
       catch (LineUnavailableException ex) {
           ex.printStackTrace();
           return;
       }

       // start the line
       line.start();

       // copy data to the line
       try {
           int numBytesRead = 0;
           while (numBytesRead != -1) {
               numBytesRead =
                   source.read(buffer, 0, buffer.length);
               if (numBytesRead != -1) {
                  line.write(buffer, 0, numBytesRead);
               }
           }
       }
       catch (IOException ex) {
           ex.printStackTrace();
       }

       // wait until all data is played, then close the
   line

       line.drain();
       line.close();

   }
```

```
/**
     Plays a stream. This method blocks (doesn't return) unti
     the sound is finished playing.
  */
  public void play(InputStream source) {

      // use a short, 100ms (1/10th sec) buffer for real-time
      // change to the sound stream
      int bufferSize = format.getFrameSize() *
          Math.round(format.getSampleRate() / 10);
      byte[] buffer = new byte[bufferSize];

      // create a line to play to
      SourceDataLine line;
      try {
          DataLine.Info info =
              new DataLine.Info(SourceDataLine.class, format);
          line = (SourceDataLine)AudioSystem.getLine(info);
          line.open(format, bufferSize);
      }
      catch (LineUnavailableException ex) {
          ex.printStackTrace();
          ret
      }
```

Format is a global variable of type AudioFormat

A SourceDataLine feeds into a *mixer*. Various sound source can feed into mixers before being played

Info is a nested class within DataLine that provides aditional information specific to DataLines (and not just Lines)

Obtains a line that matches the description in the specified Line.Info object.

The Line interface represents a mono or multi-channel audio feed.

Send the data to the line and its associated resources and play it.

```
/**
       Plays a stream. This method blocks (doesn't return) until
       the sound is finished playing.
   */
   public void play(InputStream source) {

       // use a short, 100ms (1/10th sec) buffer for real-time
       // change to the sound stream
       int bufferSize = format.getFrameSize() *
           Math.round(format.getSampleRate() / 10);
       byte[] buffer = new byte[bufferSize];

       // create a line to play to
       SourceDataLine line;
       try {
           DataLine.Info info =
               new DataLine.Info(SourceDataLine.class, format);
           line = (SourceDataLine)AudioSystem.getLine(info);
           line.open(format, bufferSize);
       }
       catch (LineUnavailableException ex) {
           ex.printStackTrace();
           return;
       }
```

This most commonly is the result of the being in use by another application.

```
public class SimpleSoundPlayer  {

    public static void main(String[] args) {
        // load a sound
        SimpleSoundPlayer sound =
            new SimpleSoundPlayer("../sounds/voice.wav");

        // create the stream to play
        InputStream stream =
            new ByteArrayInputStream(sound.getSamples());

        // play the sound
        sound.play(stream);

        // exit
        System.exit(0);
    }
```

Open file with sound

getSamples puts the sound into a byte array - user routine

create an internal class that provides the bytes in an orderly way

User routine that uses a Line to play the sound

## If you want looping sound ...

- Use a LoopingByteInputStream instead of the ByteArrayInputStream in the last slide
- Loops through the bytes of the sound sample to create the illusion of an "infinitely long" sample.
- Look at LoopingByteInputStream.java for code that does this

## Sounds filters

- Add echoes when a character is in a cave
- Make the sound shift from the left to the right speaker, or *vice versa*, for something moving
- Use the abstract SoundFilter class
- Look at file SoundFilter.java
- Can apply a sound filter to an array of samples -- this permanently affects the sound
- A better idea is to create an InputStream subclass that applies the SoundFilter - see FilteredSoundStream.java
- EchoFilter.java and Filter3d.java create filters that provide echoes and the illusion of 3d sounds

## Additional reading

- Look at Sound.java and SoundManager.java, which implements a good basis for manipulating sound in your program.
- Later sections in the book describe playing CD Audio and MP3 and Ogg Vorbis formats