Repaint and Mouse Tracking

Thanks to Nick Parlante for much of this handout

Repaint Features

Coalescing

- Using `repaint()` to make redraw requests gives us the advantage of "coalescing" -- the system can intelligently combining multiple `repaint()` requests in the event queue to a single draw operation.
- There is not a 1-1 correspondence between calls to `repaint()` and `paintComponent()` -- multiple repaints are coalesced by the system and handled by a single call to `paintComponent()`.
- Time: Multiple repaint requests for a region in quick succession are coalesced into one draw operation. You can `repaint()` 3 times in succession, but it just draws once.
- Space: repaint regions can overlap, but the area of intersection is just painted once.

Coalescing Example - JSlider

- In the Widget code (below) the `JSlider` moves, it sends a `setCount()` to the widget, which does a `repaint()`
- Suppose we move the slider quickly -- generating three `setCounts()`, 10, 11, 12, 13 in quick succession. Each call sets the model in the Widget to the ints 10, 11, 12, 13 in turn, each time calling `repaint()`.
- This does not mean we need to draw the Widget 4 times. If we did, all but the last would just be overwritten anyway -- a complete waste.
- The 4 `repaint()` calls can be coalesced into a single draw, if they are close enough together in real time. When `paintComponent()` comes through, it just sees the most recent value in the model -- 13.

Swing Timer

- A way to run some code periodically on the swing thread -- convenient and correct -- runs your code on the Swing thread
- Takes an action listener, and a "delay", like 500 milliseconds
- Calls its listener at roughly that rate, on the swing thread
- (See example below)

Repaint Sequence

1. Repaint -- region to draw

- `repaint()` tells the system that an area on screen needs to be redrawn. `repaint()` **does not call** `paintComponent()` **directly.** Repaint adds a request to draw that region to the event queue and returns immediately.
- `repaint()` is sent to a component, but the command to draw is translated to a region -- typically the bounds of that component.
- `component.repaint()` -- specifies the entire bounds of that component -- used most often
- component.repaint(<rectangle>) -- variant that specifies a sub rectangle inside the component

2. Repaint -> Update Region / Queue

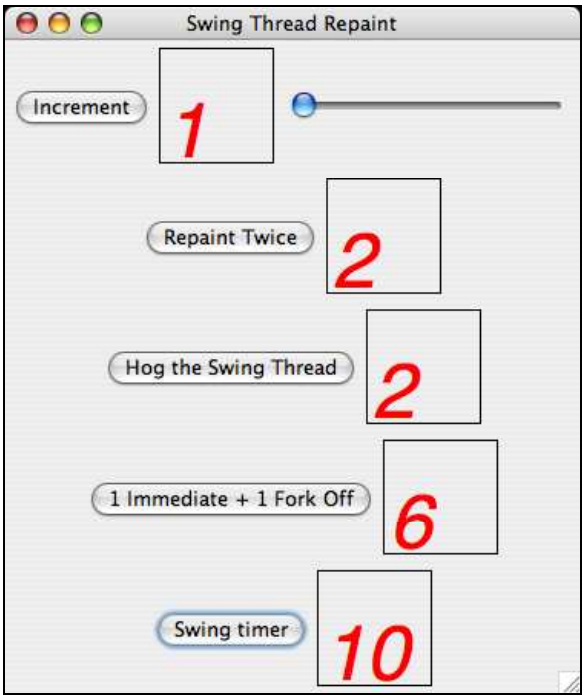
- Suppose that the system maintains a global "update region" -- a 2-d representation of areas that need to be redrawn.
- Repaint -> adds a region to the update region
- In effect, the `repaint()` adds draw-region to be processed later by the swing thread
- How this is implemented depends on the JVM -- rather than an update region, it may work by recording some data in an object in the event queue.

3. System paint thread

- On the GUI thread, the system...
- 1. Notices non-empty update region
- 2. Compute intersection of that region vs. components
- 3. Initiates draw recursion from the frame down ... eventually calling `paintComponent()` on all the components that need it. Composites the pixels together back-to-front to create the right on screen output.

Swing Thread Repaint Example

- This example demonstrates the interplay between the swing thread and `repaint()`




```
Widget.java :
 -has int count
 -paintComponent() draws the count in a box

Has these setters:
    /*
     Typical setter -- calls repaint() to alert the
     system that we need to be redrawn.
    */
    public void setCount(int newCount) {
```

```
        if (newCount != count) {
            count = newCount;  // 1. change state
            repaint();         // 2. repaint()
        }
    }

    // Increases count by 1.
    public void increment() {
        setCount(count + 1);
    }
```

```
// SwingThreadRepaint.java
/*
 Demonstrates the role of the swing thread and repaint.

 The Widget class represents a typical Swing object.
 A Widget encapsulates a single int value (that is its "model")
 which it draws. It responds to the increment() message.
 The increment() message should only be sent on the Swing thread
 since the widget is an on-screen component (just like JLabel).
 */
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import java.awt.event.*;


public class SwingThreadRepaint extends JFrame {
    // On screen Widgets, edited by the various controls
    private Widget a, b, c, d, e;
    public final int SIZE = 75;


    public SwingThreadRepaint() {
        super("Swing Thread Repaint");

        JPanel panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
        setContentPane(panel);


        // 1. Simple
        // Just call increment() -- fine, we're on the swing thread
        // so we are allowed to message and change swing state.
        // Result: works fine
        a = new Widget(SIZE, SIZE);
        JButton button = new JButton("Increment");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                a.increment();
            }
        });

        panel = new JPanel();
        panel.add(button);
        panel.add(a);
        // Add a slider that changes the widget too
        final JSlider slider = new JSlider(0, 100, 0);
        panel.add(slider);
        slider.addChangeListener(
                new ChangeListener() {
                    public void stateChanged(ChangeEvent e) {
                        a.setCount(slider.getValue());
                        // note: can refer to "slider" here since
                        // it is a *final* local variable outside.
                        // Regular outside local vars do not work.
                    }
                }
        );
        add(panel);
```

```
// 2. Coalescing
// Extra call to repaint() -- first of all, it's not necessary
// since increment() calls repaint() internally. Second of all, the two repaints
// are coalesced into a single draw operation anyway.
// The Swing thread only has a chance to do anything about it after
// this actionPerformed() exits.
// Result: the extra repaint() is basically harmless, so works fine
b = new Widget(SIZE, SIZE);
button = new JButton("Repaint Twice");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        b.increment();  // calls repaint() internally
        b.repaint();    // redundant
    }
});

panel = new JPanel();
panel.add(button);
panel.add(b);
add(panel);



// 3. Hog The Swing thread
// Notice how the UI locks up while we hog the swing thread.
// Also, the widget *never* shows an odd number. The draw thread
// only gets a chance to do anything after we exit actionPerformed()
// and by then we've always bumped the int (model) in the widget
// up to an even number.
// Result: button goes in, stays in for a few seconds, UI locks up,
// then button pops out and widget draws with +2 value
// Lesson: don't hog the swing thread
c = new Widget(SIZE, SIZE);
button = new JButton("Hog the Swing Thread");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        c.increment();

        // wait 5 seconds
        try {
            Thread.sleep(5000);
        }
        catch (InterruptedException ignored) { }

        c.increment();
    }
});

panel = new JPanel();
panel.add(button);
panel.add(c);
add(panel);



// 4. Here we increment once immediately,
// and then fork off a worker to do something time-consuming
// and then increment when it is done. Notice that the UI
// remains responsive while the worker is off doing its thing.
// The worker uses the standard SwingUtilities.invokeLater() call
// to communicate back to swing.
// Result: button goes in and out normally, one increment happens
// immediately, UI remains responsive, after a few seconds, the
// widget increments a second time on its own. It is possible to
// click the button multiple times to fork off multiple, concurrent
// workers, or could use an interruption strategy on the previous worker.
d = new Widget(SIZE, SIZE);
button = new JButton("1 Immediate + 1 Fork Off");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // 1. Do an increment right away
        // -- ok, we're on the swing thread
        d.increment();
```

```
                // 2. Fork off a worker to do the iterations
                // on its own, followed by the increment()
                Thread worker = new Thread() {
                    public void run() {

                        // wait 2 seconds
                        try {
                            Thread.sleep(2000);
                        }
                        catch (InterruptedException ignored) { }

                        // 3. When worker wants to communicate back to swing,
                        // must go through invokeLater/runnable
                        SwingUtilities.invokeLater(
                                new Runnable() {
                                    public void run() {
                                        // Q: writer/writer conflict here?
                                        d.increment();
                                    }
                                }
                        );

                    }
                };
                worker.start();
            }
        });
        panel = new JPanel();
        panel.add(button);
        panel.add(d);
        add(panel);


        // 5. Make and start a Swing "Timer"
        // A Swing Timer calls actionPerformed() periodically on the
        // Swing thread. See TimerListener below.
        e = new Widget(SIZE, SIZE);

        button = new JButton("Swing timer");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // Timer fires every 500 msecs
                Timer timer = new Timer(500, new TimerListener());
                timer.start();
            }
        });
        panel = new JPanel();
        panel.add(button);
        panel.add(e);
        add(panel);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    // Listener for the timer
    private class TimerListener implements ActionListener {
        int count = 0;
        public void actionPerformed(ActionEvent evt) {
            e.increment(); // Q: writer/writer conflict here if multiple timers going?
            count++;
            // On the 10th time, send stop() to the timer
            // Get a pointer to the timer with evt.getSource()
            if (count==10) {
                ((Timer)evt.getSource()).stop();
            }
        }
    }

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

Mouse Tracking

- Use `MouseListener` and `MouseMotionListener` to get notifications about mouse events over a component.
- The component itself is the source of the notifications -- add the listener to the component.

Listener vs. Adapter Style

- Problem
  - Listener has a bunch of abstract methods -- e.g. 5 in `MouseListener` (below).
  - You typically only care about one or two, so implementing all 5 is a bore.
- Solution
  - "Adpater" class has empty { } definitions of all the methods
  - Then you only need to implement the ones you care about -- the adapter catches the others.
- Bug
  - If you type the prototype slightly wrong, your method will be ignored -- e.g. MousePressed() instead of the correct `mousePressed()`

MouseListener Interface

```
public interface MouseListener extends EventListener {

    /**
     * Invoked when the mouse has been clicked on a component.
       (press+release)
     */
    public void mouseClicked(MouseEvent e);

    /**
     * Invoked when a mouse button has been pressed on a component.
     */
    public void mousePressed(MouseEvent e);

    /**
     * Invoked when a mouse button has been released on a component.
     */
    public void mouseReleased(MouseEvent e);

    /**
     * Invoked when the mouse enters a component.
     */
    public void mouseEntered(MouseEvent e);

    /**
     * Invoked when the mouse exits a component.
     */
    public void mouseExited(MouseEvent e);
}
```

Mouse Adapter Class

```
public abstract class MouseAdapter implements MouseListener {
    /**
     * Invoked when the mouse has been clicked on a component.
     */
    public void mouseClicked(MouseEvent e) {}

    /**
     * Invoked when a mouse button has been pressed on a component.
     */
    public void mousePressed(MouseEvent e) {}

    /**
     * Invoked when a mouse button has been released on a component.
     */
    public void mouseReleased(MouseEvent e) {}

    /**
     * Invoked when the mouse enters a component.
```

```
     */
    public void mouseEntered(MouseEvent e) {}

    /**
     * Invoked when the mouse exits a component.
     */
    public void mouseExited(MouseEvent e) {}
}
```

Press : MouseListener

- How to hear about a mouse press on a component...

```
component.addMouseListener( new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        // called when mouse button first pressed on component
```

Motion: MouseMotionListener

- How to hear about a mouse gesture with mouse button held down...

```
component.addMouseMotionListener( new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
        // called as mouse is dragged, after initial click
```

JComponent = source

- The `JComponent` where the click began is the "source" object for the mouse events. Register with the component to hear about clicks on it.

Local Co-Ords

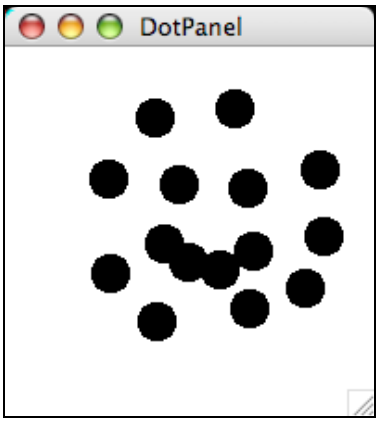
- Notifications about the mouse event will use the local coord system of the component where they happened. (This is similar to the way `paintComponent()` works -- using the local coord system.)

The "delta" rule for mouse motion

- Wrong: absolute
  - Use the current coords of the mouse
  - Set the position of whatever it is to those coords
- Right: relative
  - Get the current coords
  - Compare the last coords
  - Apply that delta to whatever it is
- Test case
  - A click-release with no motion should not change any state -- relative mouse tracking gets this right.

Dot Example (mouse tracking, and smart repaint)

- This example `JPanel` has a model -- a list of `DotModel` { x, y, color} objects
- It draws the dots, and supports mouse operations on the dots
- Also can do smart-repaint.
- This example is also used to show file-saving later on.
- This whole example is available in the hw dir to play with.




```
//DotPanel.java
/**
 A Panel that draws a series of dots.
 The data model is a list of DotModel objects.
 */

import java.awt.*;

import javax.swing.*;

import java.util.*;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.beans.*;
import java.io.*;


public class DotPanel extends JPanel  {
    private ArrayList<DotModel> dots; // our data model is a list of DotModel objects
    public final int SIZE = 20; // diameter of one dot

    // remember the last dot for mouse tracking
    private int lastX, lastY;
    private DotModel lastDot;

    // Booleans that control how we draw
    private boolean print;
    private boolean smartRepaint;
    private boolean oldRepaint;
    private boolean redPaint;


    // dirty bit = changed from disk version
    private boolean dirty;


    /**
     Creates an empty DotPanel.
     */
    public DotPanel(int width, int height) {
        setPreferredSize(new Dimension(width, height));

        // Subclasing off JPanel, these things work
        setOpaque(true);
        // optimization: set opaque true if we fill 100% of our pixels
        setBackground(Color.white);

        dots = new ArrayList<DotModel>();
        clear();

        // Controls for debugging options
        print = false;
        smartRepaint = true;
        oldRepaint = true;
```

```
    redPaint = false;


    /*
     Mouse Strategy:
     -if the click is not on an existing dot, then make a dot
     -note where the first click is into lastX, lastY
     -then in MouseMotion: compute the delta of this position
     vs. the last
     -Use the delta to change things (not the abs coordinates)
     */

    addMouseListener( new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            if (print) System.out.println("press:" + e.getX() + " " + e.getY());

            DotModel dotModel = findDot(e.getX(), e.getY());
            if (dotModel == null) {   // make a dot if nothing there
                dotModel = doAdd(e.getX(), e.getY());
            }

            // invariant -- dotModel now determined, one way or another

            // Note the starting setup to compute deltas later
            lastDot = dotModel;
            lastX = e.getX();
            lastY = e.getY();

            // Change color of dot in some cases
            // shift -> change to black
            // double-click -> change to red
            if (e.isShiftDown()) {
                doSetColor(dotModel, Color.BLACK);
            }
            else if (e.getClickCount() == 2) {
                doSetColor(dotModel, Color.RED);
            }

        }
    });


    addMouseMotionListener( new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) {
            if (print) System.out.println("drag:" + e.getX() + " " + e.getY());

            if (lastDot != null) {
                // compute delta from last point
                int dx = e.getX()-lastX;
                int dy = e.getY()-lastY;
                lastX = e.getX();
                lastY = e.getY();

                // For fun, if shift-key is down, multiply dx,dy * -2
                // demonstrates that the UI appearance of "drag" is a just
                // a careful illusion
                if (e.isShiftDown()) {
                    dx *= -2;
                    dy *= -2;
                }

                // apply the delta to that dot model
                doMove(lastDot, dx, dy);
            }
        }
    });

}

// Clears out all the data (used by new docs, and for opening docs)
public void clear() {
    dots.clear();
    dirty = false;
    repaint();
```

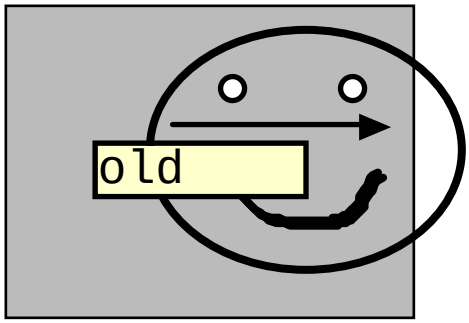
```
}

// Default ctor, uses a default size
public DotPanel() {
    this(300, 300);
}


/**
 Moves a dot from one place to another.
 Does the necessary repaint.
 This animation can repaint two ways.
 Plain repaint: repaint the whole panel
 Smart repaint: repaint just the old+new bounds of the dot
 */
public void doMove(DotModel dotModel, int dx, int dy) {
    if (!smartRepaint) {
        // Change the data model, then repaint the whole panel
        dotModel.moveBy(dx, dy);
        repaint();
    }
    else {
        // Smart repaint: old + new
        // Repaint the "old" rectangle
        if (oldRepaint) {
            repaintDot(dotModel);
        }
        // Change the model
        dotModel.moveBy(dx, dy);
        // Repaint the "new" rectangle
        repaintDot(dotModel);
    }

    setDirty(true);
}

/**
 Utility -- change the color of the given dot model,
 and then do the needed repaint/setDirty.
 */
private void doSetColor(DotModel dot, Color color) {
    dot.setColor(color);  // change the model
    repaint();      // bookeeping for the view: repaint and set dirty
    setDirty(true);
}

/**
 Utility -- does a repaint rect just around one dot. Used
 by smart repaint when dragging a dot.
 */
private void repaintDot(DotModel dot) {
    repaint(dot.getX()-SIZE/2, dot.getY()-SIZE/2, SIZE+1, SIZE+1);
}


/**
 Utility -- given a completed dot model, adds it and sets things up.
 This is the bottleneck for adding a dot.
 */
public void doAdd(DotModel dotModel) {
    dots.add(dotModel);
    repaint();
    setDirty(true);
}


/**
  Convenience doAdd() that takes an int x,y, adds and returns
  a dot model for it.
 */
public DotModel doAdd(int x, int y) {
    DotModel dotModel = new DotModel();
    dotModel.setXY(x, y);
    doAdd(dotModel);
    return dotModel;
```

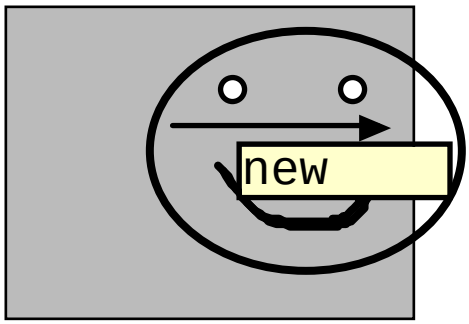
```
}


/**
 Finds a dot in the data model that contains
 the given x,y or returns null.
 */
public DotModel findDot(int x, int y) {
    // Search through the dots in reverse order, so
    // hit topmost ones first.
    for (int i=dots.size()-1; i>=0; i--) {
        DotModel dotModel = dots.get(i);
        int centerX = dotModel.getX();
        int centerY = dotModel.getY();

        // figure x-squared + y-squared, see if it's
        // less than radius squared.
        // trick: don't need to take square root
        int xySquared = (x - centerX)*(x - centerX) +
        (y - centerY)*(y - centerY);
        int radiusSquared = (SIZE/2)*(SIZE/2);
        if (xySquared <= radiusSquared) return dotModel;
    }
    return null;
}


/**
 Standard override -- draws all the dots.
 */
public void paintComponent(Graphics g) {
    // As a JPanel subclass we need call super.paintComponent()
    // so JPanel will draw the white background for us.
    super.paintComponent(g);

    // Go through all the dots, drawing a circle for each
    for (DotModel dotModel : dots) {
        g.setColor(dotModel.getColor());
        g.fillOval(dotModel.getX() - SIZE/2, dotModel.getY() - SIZE/2,
                SIZE, SIZE);
    }

    // Draw the "requested" clip rect in red
    // (this just shows off smart-repaint)
    if (redPaint) {
        Rectangle clip = g.getClipBounds();
        if (clip != null) {
            g.setColor(Color.red);
            g.drawRect(clip.x, clip.y, clip.width-1, clip.height-1);
        }
    }
}
```

Smart Repaint (optional)

Move Repaint: old+new

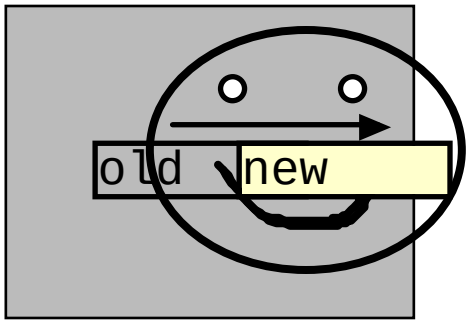
- Suppose we have a gray rectangle, with a partially transparent smiley face in front of it. In front of both, we have a yellow rectangle that moves left to right. What is the minimum area to redraw?





Old + New

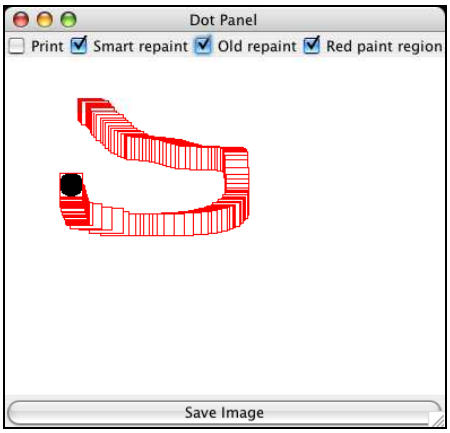
- Both the old region and the new region of the yellow rect need to be redrawn. The new region needs to draw with the yellow rect there. The old region needs to be drawn with the yellow rect **not there**.



Smart Repaint

- Smart repaint = repaint just the sub-rectangle within the component that needs to be redrawn, not the entire component.
- This makes the subsequent `paintComponent()` cycle faster, so we get faster, smoother drawing. This can be a large speedup, if the smart repaint rectangle is significantly smaller than the standard repaint of the whole bounds.
- There's a version of the `repaint()` method that takes a rectangle argument, and just repaints that rectangle (rather than the component bounds) -- `component.repaint(x, y, width, height)`
- e.g. -- just repaint the old+new rectangles when a component moves.
- The system gets this right automatically when moving Swing library components around within, say, a `JPanel`. See the `setBounds()` source code -- repaints just the old+new regions.
- For ordinary code, just calling `component.repaint()` -- redrawing the whole component bounds -- is good enough most of the time. In cases that we really care about animation performance, it can be worth the effort to do smart-repaint.

- Here is the dot example, showing the trail of repaint rects during a mouse drag in smart-paint mode:



Why Faster

- Smart repaint reduces the size of the area that must be drawn -- greatly reducing the number of pixels that need to be copied around.
- The `paintComponent()` code can still be written in the most straightforward way -- it does not need to do anything special to get the benefit of smart repaint. The `paintComponent()` will automatically run faster when given a smaller area to draw by smart repaint.