CS193j, Stanford
Winter, 2001-02

Handout #15
Nick Parlante

Drawing 4

From last time

Control listening vs. polling
Erasing fish hat example

Repaint 2

Today, we'll look at how the repaint system works in more detail.

1. Repaint -- region to draw

Repaint() tells the system that an area on screen needs to be redrawn
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
component.repaint(<rectangle>) -- specifies a sub rectangle inside the component

2. Repaint -> Update Region

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

3. System paint thread

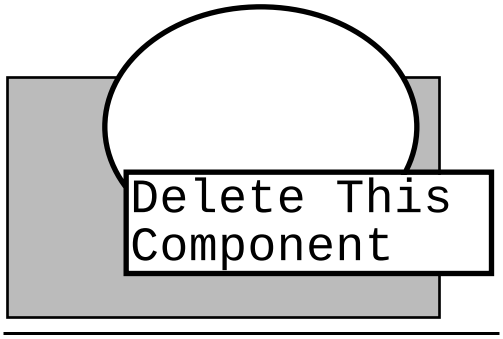
1. Notices non-empty update region
2. Compute intersection of that region vs. components
3. Initiates draw recursion down the component nesting hierarchy. Composites the pixels together back-to-front.

Region Based Drawing

The need to draw something is always expressed in terms of **regions** of pixels, not just components.
This scheme deals with intersection and z-order correctly

Overlap

Draw all the components that intersect the pixel region that needs to be redrawn.
Draw the components from back to front.



Move Component -> old bounds + new bounds -- "smart repaint"

Move a component from an old position to a new position.

What needs to be redrawn?

Both the old region and the new region -- the old region needs to be drawn with the component not there.

Smart repaint = repaint just the needed rectangles, not the entire component area.

The system gets this right automatically when moving components around with, say, a JPanel. See the setBounds() source code -- repaints the old+new regions.

Coalescing

Using repaint() to make redraw requests gives us the advantage of "coalescing" -- intelligently combining multiple repaint() requests into a single draw operation.

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 drawn once.

Coalescing Example - JSlider

Consider the JSlider/MyComponent example from last time

When the JSlider moves, it sends a setCount() to the widget, which does a repaint()

Suppose we move the slider quickly -- generating three setCounts() in quick succession, resulting to three repaint() calls.

This does not mean we need to draw the MyComponent three times. If we did, the first two draws would just be overwritten anyway -- potentially a complete waste.

The three repaints() can be coalesced into a single draw, if they are close enough together in realtime.

Mouse Tracking

Use MouseListener 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.

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 Code

```
public interface MouseListener extends EventListener {

    /**
     * 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);
}
```

Mouse Adapter Code

```
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) {}
}
```

Click : MouseListener

```
component.addMouseListener( new MouseAdapter() {
   public void mousePressed(MouseEvent e) {
      // called for mouse click on the component
```

Motion: MouseMotionListener

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 co-ord system of the component where they happened. (This is similar to the way paintComponent() works -- using the local co-ord system.)

The "delta" rule for mouse motion

Wrong: absolute
  Use the current co-ords of the mouse--
  Set the position of whatever it is to those co-ords
Right: relative
  Get the current co-ords
  Compare the last co-ords

- Apply that delta to whatever it is

Scenario

- An example of this being done wrong is a lower-right resize-knob that moves the lower-right corner **to** the mouse position instead of applying the delta to the lower-right corner. With the wrong strategy, a click-release with no motion can still move the corner.

Draw-Clip Optimization

Suppose we have a program that does smart repainting

Therefore, when paintComponent is called, often it is really just drawing a little area of the component. The "clip bounds" of the Graphics object will show the area were drawing really needs to happen. The clip bounds is probably the same rectangle that was set by the earlier smart repaint operation.

Optimization

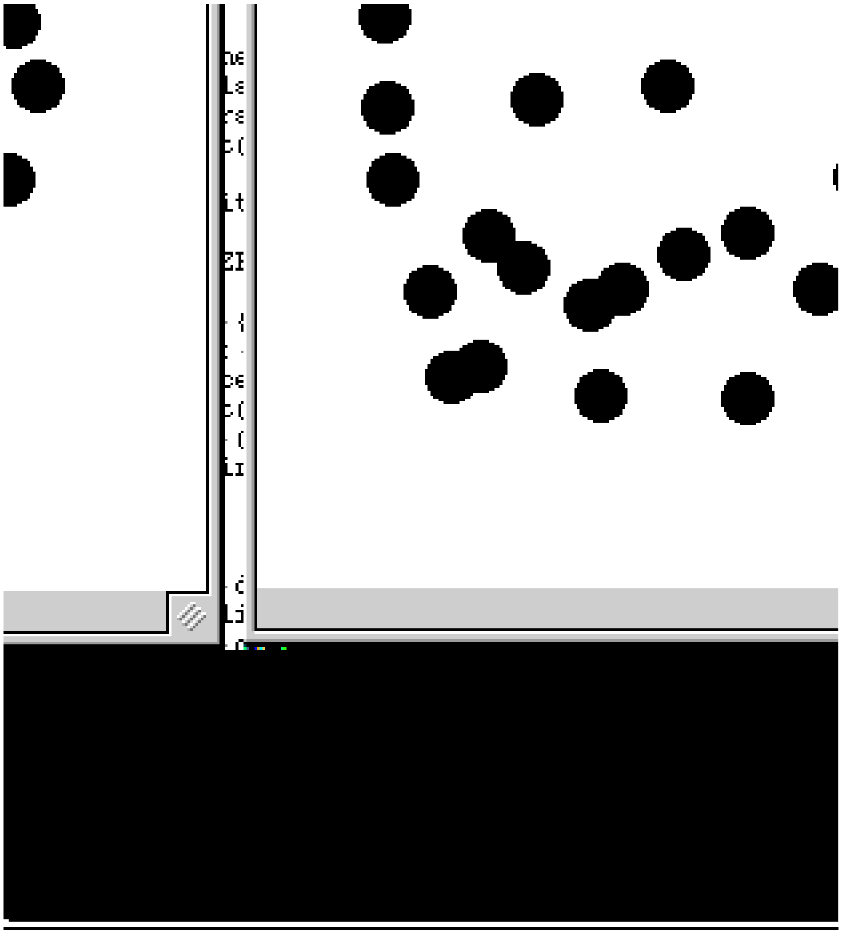
- Get the clipBounds from the Graphics object -- the little area where drawing is actually happening
- When drawing things, check to see if they intersect the clip bounds first -- if they do not, don't draw them

Smart repaint is the most important draw optimization.

The draw-clip optimization is nice, but secondary. You can skip it and still get good performance.

See the Dots example -- does smart repaint on dot move, and draw-clip opt in paintComponent.

DotPanel Example



```
// DotPanel.java
/**
 The DotPanel class demonstrates a few things...

 -Mouse tracking -- clicking makes a new point, clicking
 on an existing point moves it. The data model is the collection
 of points where there is a dot on screen.

 -Smart repaint -- only repaints the needed rectangle when a dot moves

 -Draw-clip optimization -- looks at the clip bounds when drawing
*/

import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.event.*;


class DotPanel extends JPanel implements DocPanel {
   private ArrayList dots; // represent each dot by its center point
   public final int SIZE = 20;   // diameter of one dot

   // remember the last point for mouse tracking
```

```
private int lastX, lastY;
private Point lastPoint;

public boolean smartRepaint = true;


private boolean dirty;



/**
 Utility test-main creates a DotPanel in a window.
*/
public static void main(String[] args) {
   JFrame frame = new JFrame("Dot Panel");

   JComponent container = (JComponent) frame.getContentPane();

   DotPanel dotPanel = new DotPanel(300, 300, null);

   container.add(dotPanel);


   frame.addWindowListener(
      new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      }
   );

   frame.pack();
   frame.setVisible(true);
}


/**
 Create an empty DotPanel. Load the contents of the
 given File if it is non-null.
*/
public DotPanel(int width, int height, File file) {
   super();
   setPreferredSize(new Dimension(width, height));
   setBackground(Color.white);

   dirty = false;
   dots = new ArrayList();

   if (file != null) {
      load(file);
   }
   /*
    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) {
         //System.out.println("press:" + e.getX() + " " + e.getY());

         Point point = findDot(e.getX(), e.getY());
         if (point == null) { // make a dot if nothing there
            point = addDot(e.getX(), e.getY());
         }

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

      }
   });


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

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

            // apply the delta to that point
            moveDot(lastPoint, dx, dy);
         }
      }
   });
}

/**
 Generates a repaint for the rect around one dot
 smart: repaint the rect just around the dot
 standard: repaint the whole panel
*/
public void repaintDot(Point point) {
   if (smartRepaint) {
      repaint(point.x-SIZE/2, point.y-SIZE/2, SIZE, SIZE);
   }
   else {
      repaint();
   }
}
```

```
/**
 Moves a dot from one place to another.
 Trick: needs to repaint both the old and new locations
 Moving components get this right automatically --
 see component.setBounds().
*/
public void moveDot(Point point, int dx, int dy) {
   repaintDot(point);   // repaint its old rectangle
   point.x += dx;
   point.y += dy;
   repaintDot(point);   // repaint its new rectangle

   setDirty(true);
}


/**
 Private utility -- adds a dot to the data model.
*/
private Point addDot(int x, int y) {
   Point point = new Point(x, y);
   dots.add(point);
   repaintDot(point);

   setDirty(true);

   return(point);
}


/**
 Finds a dot in the data model that contains
 the given point, or return null.
*/
public Point findDot(int x, int y) {
   Iterator it = dots.iterator();
   while (it.hasNext()) {
      Point point = (Point)it.next();
      int left = point.x-SIZE/2;
      int top = point.y-SIZE/2;
      if (left<=x && x<left+SIZE &&
         top<=y && y<top+SIZE) {
         return(point);
      }
   }
   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 background for us.
       super.paintComponent(g);

       Iterator it = dots.iterator();

       boolean CLIP_OPTIMIZE = true;

       if (!CLIP_OPTIMIZE) {
          // standard draw: just iterate through and draw them all.
          // the performance of this is fine actually
          while (it.hasNext()) {
             Point point = (Point)it.next();
             g.fillOval(point.x - SIZE/2, point.y-SIZE/2, SIZE, SIZE);
          }
       }
       else {
          // clip optimize draw: only draw the dots that intersect
          // the current clip bounds
          Rectangle clip = g.getClipBounds();
          Rectangle temp = new Rectangle();

          while (it.hasNext()) {
             Point point = (Point)it.next();

             temp.x = point.x - SIZE/2;
             temp.y = point.y - SIZE/2;
             temp.width = SIZE;
             temp.height = SIZE;

             if (clip.intersects(temp)) {
                g.fillOval(temp.x, temp.y, temp.width, temp.height);
             }
          }

       }
    }
 }
```