Programming Languages and Techniques (CIS120)

Lecture 35

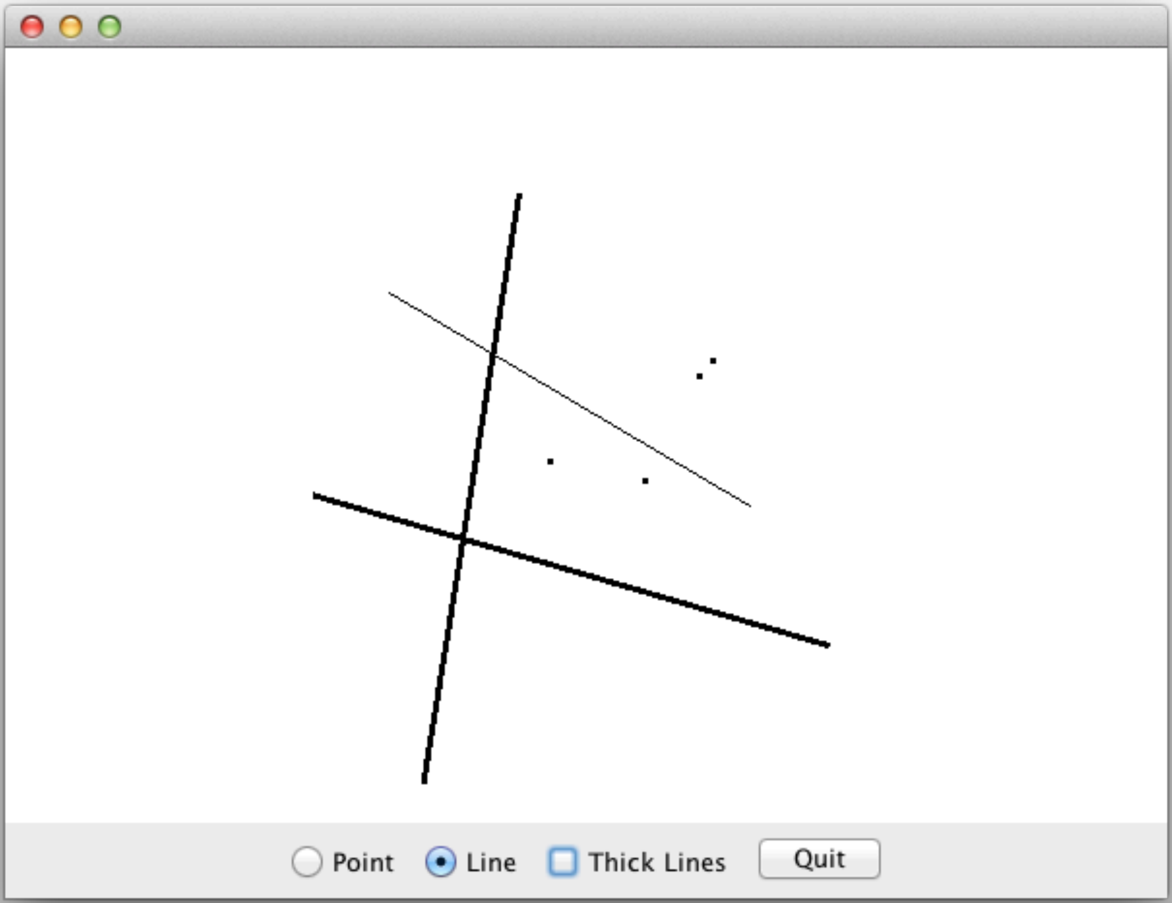
April 15, 2013

Swing III: OO Design, Mouse Interaction

Announcements

- HW10: Game Project is out, due Tuesday, April 23rd at midnight
  - If you want to do a game other than one of the ones listed, send email to tas120@seas.upenn.edu (or check on Piazza)

Java Paint



Basic structure

- Main frame for application (class Paint) the *MODEL*
- Drawing panel (class Canvas, inner class of Paint) the *VIEW*
- Control panel (class JPanel) the *CONTROLLER*
  - Contains radio buttons for selecting shape to draw
  - Line thickness checkbox, undo and quit buttons

- Paint class contains the state of the application
  - List of shapes to draw
  - Preview shape (if any…)
  - The current color (will always be BLACK today)
  - The current line thickness
  - References to UI components: canvas, modeToolbar

Program Design

How does our treatment of shape drawing in Java compare with the OCaml GUI project?

Java Version of Paint

```
public interface Shape {
    public void draw(Graphics2D gc);
}
```

Interface describes what shapes can do

```
public class PointShape implements Shape { … }
public class LineShape implements Shape { … }
```

Classes describe how to draw themselves

```
private class Canvas extends JPanel {
   public void paintComponent(Graphics gc) {
        super.paintComponent(gc);
        for (Shape s : actions)
            s.draw((Graphics2D)gc);
        if (preview != null)
            preview.draw((Graphics2D)gc);
  }
}
```

Canvas uses dynamic dispatch to draw the shapes

OCaml Version of Paint

```
type shape =
  | Points   of Gctx.color * int * point list
  | Line     of Gctx.color * int * point * point

let repaint (g:Gctx.t) : unit =
    let draw_shape (s:shape) : unit =
        begin match s with
            | Points (c,t,ps) -> …
            | Line (c,t,p1,p2) -> …
    end in
    Deque.iterate draw_shape paint.shapes;
    begin match paint.preview with
    | None -> ()
    | Some d -> draw_shape d
    end
```

Datatypes define the structure of information.

Drawing operation is defined externally to the datatype and uses case analysis to dispatch.

The “main” loop looks very similar.

Comparison with OCaml

- How does our treatment of shape drawing in the Java Paint example compare with the OCaml GUI project?

- Java:
  - Interface Shape for drawable objects
  - Classes implement that interface
  - Canvas uses dynamic dispatch to draw the shapes
  - Add more shapes by adding more implementations of "Shape"
- OCaml
  - Datatype specifies variants of drawable objects
  - Canvas uses pattern matching to draw the shapes
  - Add more shapes by adding more variants, and modifying drawit

Datatypes vs. Objects

Datatypes

- Focus on how the data is stored
- Easy to add new operations
- Hard to add new variants

- Best for: situations where the *structure* of the data is fixed (i.e. BSTs)

Objects

- Focus on what to do with the data
- Easy to add new variants
- Hard to add new operations

- Best for: situations where the *interface* with the data is fixed (i.e. Shapes)

Mouse Interaction

How do we draw shapes on the canvas?

Mouse Interaction

- One Option: Copy OCaml structure

```
public enum Mode {
   PointMode, LineStartMode, LineEndMode
}
private Mode mode = Mode.PointMode;
```

- Button press switches between PointMode and LineStartMode
- Mouse click in PointMode → add a new point to the list of shapes
- Mouse press in LineStartMode → remember location, switch to LineEndMode, remember preview shape
- Mouse movement in LineEndMode → update preview shape
- Mouse release in LineEndMode → add a new line to list of shapes, switch to LineStartMode, set preview to null

Two interfaces for mouse listeners

```
interface MouseListener extends EventListener {
  public void mouseClicked(MouseEvent e);
  public void mouseEntered(MouseEvent e);
  public void mouseExited(MouseEvent e);
  public void mousePressed(MouseEvent e);
  public void mouseReleased(MouseEvent e);
}
```

```
interface MouseMotionListener extends EventListener {
   public void mouseDragged(MouseEvent e);

   public void mouseMoved(MouseEvent e);
}
```

Lots of boilerplate

- There are seven methods in the two interfaces.
- We only want to do something interesting for three of them.
- Need "trivial" implementations of the other four to implement the interface…

```
public void mouseMoved(MouseEvent e)   { return; }
public void mouseClicked(MouseEvent e) { return; }
public void mouseEntered(MouseEvent e) { return; }
public void mouseExited(MouseEvent e)  { return; }
```

- Solution: MouseAdapter class…

Adapter classes:

- Swing provides a collection of abstract event adapter classes
- These adapter classes implement listener interfaces with empty, do-nothing methods
- To implement a listener class, we extend an adapter class and override just the methods we need

```
private class Mouse extends MouseAdapter {
   public void mousePressed(MouseEvent e) { … }
   public void mouseReleased(MouseEvent e) { … }
   public void mouseDragged(MouseEvent e) { … }
}
```

OO Mouse Interaction

What about OO version