Benjamin Fenster
CS-296 (Damon)
19 May 2005

Implementation

I. Model View Controller
  a. Model
    i. Underlying Application
    ii. This isn't the UI stuff; it's the rest of the code
  b. View
    i. This is how users see the model
    ii. It includes everything that presents information to the user
  c. Controller
    i. This is how user's change what's in the model
    ii. View + Controller comprise the UI
  d. Problems
    i. Realistically you'll have several View-Model-Controller sets for different parts of the application.
    ii. Also, View and Controller aren't really separate. They interact and share the same controls (a text box is used to view and to edit text)
  e. PAC
    i. Have the Abstraction (the underlying application)
    ii. Controller gets and sets data in the abstraction and decides what to display
    iii. Presentation actually does the display.
    iv. See [CS-296-2005-01-SLIDES-20:18]
  f. How do they Interact
    i. Function calls? Would need a unique function call for each type of data! Also, function calls are synchronous, so the calling code wouldn't get an answer for a while.
    ii. Callbacks? Solves the anonymity problem – anybody that's got a callback can be used in the UI code. It'll just ask you for data.
    iii. Events
      1. Asynchronous and Anonymous
      2. Whenever something interesting happens (in either direction), fire an event. Interested parties (all of them) receive it
      3. Certainly retains anonymity
      4. It's also asynchronous: once you've fired the event you can go about your business.
      5. When the listener can see the event queue, it can even skip repeated events (e.g. mouse movement) to catch up if it starts to fall behind

II. Programming GUIs
  a. Create controls
  b. Listen for events (and link to the underlying application)
  c. All GUIs are done with events now (excluding older stuff that's still out there, such as older parts of the Windows API that still use callbacks)
  d. Start in main(), create controls, then go completely passive and wait for the user dialog to define what happens next. It's completely reactive.

III. Awt vs. Swing
  a. AWT = Abstract Window Toolkit. Some parts of it are still in broad use, but mostly not.
  b. Swing
    i. Introduced in Java 1.2
    ii. Three Big Changes
      1. Swappable look and feel (will look appropriate to the current platform). Requires no coding changes whatsoever.
      2. Improved class hierarchy
      3. Brought drawing abilities up closer to the high end (i.e. more like Photoshop than like MS Paint)
    iii. Components

1. Everything visible on the screen (and some other stuff too) is a JComponent
2. Swing components are lightweight (They don't have their own buffer for drawing).

iv. Actions
   1. Encapsulated listener for various events
   2. Makes it easy to change to a different component without changing the control at all.

v. Borders: Can make really fancy or really simple borders

vi. Pluggable Look and Feel
   1. Can subclass your own unique look and feel
   2. Can programmatically change the look too, if you want
   3. Could have a "Mac" button that uses the Mac appearance, and a "Windows" button that looks like a Windows button.

vii. Tool Tips
   1. Can override a tip when the control is disabled or whatever
   2. Can override the appearance of the popup too

IV. Swing Drawing

a. Have double-precision floating point coordinates (not just integers) between pixels

b. Differences from Graphics (AWT)
   i. Paint extends color
   ii. Clip can now be any arbitrary shape

c. Shapes
   i. Defined with PathIterator
   ii. It's a sequence of edges which may be straight or curved
   iii. It's not a true Iterator object, but behaves about the same.
   iv. Has isDone() and next()
   v. currentSegment() gives coordinates and details
   vi. FlatteningIterator returns the same segments without curves. You can vary the allowable error so you either got lots of little tiny segments (but a shape that looks more like the original) or a few straight segments that don't look much like the original.

d. Shape Geometry
   i. A point is "inside" a shape if it's clearly visible or if it's on the top/left edge
   ii. If it's on the bottom/right edge it's outside.
   iii. That way when two shapes are adjacent, no point is in both.
   iv. Everything defines geometry. See [CS-296-2005-01-SLIDES-26:12]

e. Area
   i. Special shape with Constructive Area Geometry
   ii. That means you can add and subtract Areas, take the intersection, or xor any two areas to get a new Area.
   iii. For clipping, this is great!

f. Translation
   i. All shapes really start at (0, 0)
   ii. Can use translation to shift the shape elsewhere.
   iii. Can also scale, rotate, and shear shapes.

g. Strokes
   i. This is the pen style.
   ii. Supports dashed lines, different line caps, and different ways lines intersect at corners.

h. Paints
   i. Paints define how things are filled
   ii. Colors are simple paints, but bitmaps and gradients work too

i. Anti-Aliasing
   i. Fill in pixels with half intensity when you'd like half a pixel to look filled-in
   ii. The eye perceives this correctly.

- iii. Also blends color to get half what you’re drawing and half what was underneath.
- iv. (May use any fraction, not just half)

- j. Alpha Compositing
  - i. How transparent is the object?
  - ii. There’s a formula for deciding how to blend the drawing color and what’s underneath to make something look partially transparent
- k. Color
  - i. RGB, SRGB (supposedly *standard* across monitors)
  - ii. HSV (Hue, Saturation, Value)
  - iii. CMYK (Cyan, Magenta, Yellow, Black – common for printing)
  - iv. CIEXYZ. International standard for representing any color in any color space. That is, RGB and CMYK can’t represent all the same colors, but this standard can represent anything.
- l. Custom Compositing
  - i. Alpha compositing defines rules for combining colors.
  - ii. You might set your own rules (for mixing paint, for example)
  - iii. You’re given two color spaces, a source, and a target.
  - iv. Defines the destination = source OP destination for some operation

- V. Typography
  - a. Terminology
    - i. A *glyph* is any printable character
    - ii. A font is a collection of glyphs all the same size, weight, and family
    - iii. Typeface is family + weight
    - iv. Family is the look of a glyph (e.g. Arial)
  - b. Sizes
    - i. Baseline is immediately underneath letters (but above hanging pieces)
    - ii. Midline is above letters (but underneath the tops of tall letters)
    - iii. The ascent and descent describe how far above and below the baseline and midline letters extend
  - c. Kerning helps short letters fit under tall letters by rendering them closer together.
  - d. Ligature combines two letters into one when they’re close together to avoid strange rendering (ff, for example)
  - e. Rendering Text
    - i. Early Fonts
      1. Early fonts (and ASCII terminals before that) were bitmapped.
      2. That meant they were fixed width (at first) and each letter had a simple map of which glyph to draw.
      3. It was really fast, but really ugly.
      4. It was also limited to sizes the artist explicitly created
    - ii. Outline Fonts
      1. Defined as a curve from x to y, a gentler curve from y to z, et cetera
      2. Scale to any size you want
      3. Will also anti-alias now
      4. Have to calculate lots of stuff, so there’s a noticeable slowdown.
      5. TrueType is a way of (language for) describing outlines.