Output: Hardware
Output System Layers

- Application 1: Swing
- Application 2: SWT
- Application 3: UIKit
- Window System
- Operating System
- Hardware (e.g., graphics card)
Output Hardware
Start with some basics: display devices

- Just how do we get images onto a screen?
- Most prevalent device: CRT
  - Cathode Ray Tube
  - AKA TV tube
Cathode Ray Tubes

- Cutting edge 1930’s technology
  - (basic device actually 100 yrs old)
  - Vacuum tube (big, power hog, …)
  - Refined some, but no fundamental changes
- But still dominant
  - Because TVs are consumer item
  - LCD’s just starting to challenge
How a CRT works (B/W)

Vacuum Tube

Phosphor Coating

Deflection Coils

Electron Gun

Negative charge

Positive charge

15-20 Kv
Move electron beam in fixed scanning pattern

- “Raster” lines across screen
- Modulate intensity along line (in spots) to get pixels
Pixels determined by 2D array of intensity values in memory

- “Frame buffer”
  - Each memory cell controls 1 pixel

- All drawing by placing values in memory
Adding color

• Use 3 electron guns
• For each pixel place 3 spots of phosphor (glowing R, G, & B)
• Arrange for red gun to hit red spot, etc.
  • Requires a lot more precision than simple B/W
  • Use “shadow mask” behind phosphor spots to help
Color frame buffer

- Frame buffer now has 3 values for each pixel
  - each value drives one electron gun
  - can only see ~ $2^8$ gradations of intensity for each of R, G, & B
  - 1 byte ea => 24 bits/pixel => full color
Other display technologies: LCD

- Liquid Crystal Display
- Discovered in 1888 (!) by Reinitzer
- Uses material with unusual physical properties: liquid crystal
  - rest state: rotates polarized light 90°
  - voltage applied: passes as is
Layered display

- Layers

<table>
<thead>
<tr>
<th>Horizontal Polarizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Liquid Crystal</td>
</tr>
<tr>
<td>Vertical Polarizer</td>
</tr>
</tbody>
</table>

- In rest state: light gets through
  - Horizontally polarized, LC flips 90°, becomes vertically polarized
  - Passes through
Layered display

- Layers

Horizontal Polarizer

Liquid Crystal

Vertical Polarizer

- In powered state: light stopped
  - Horizontally polarized, LC does nothing, stopped by vertical filter
Lots of other interesting/cool technologies

• Direct retinal displays
  • University of Washington HIT lab
• Set of 3 color lasers scan image directly onto retinal surface
  • Scary but it works
  • Very high contrast, all in focus
  • Potential for very very high resolution
  • Has to be head mounted
All these systems use a frame buffer

- Again, each pixel has 3 values
  - Red, Green Blue
- Why R, G, B?
  - R, G, and B are particular freq of light
  - Actual light is a mix of lots of frequencies
  - Why is just these 3 enough?
Why R, G, & B are enough

• Eye has receptors (cones) that are sensitive to (one of) these
  • Eye naturally quantizes/samples frequency distribution

• 8-bit of each does a pretty good job, but… some complications
Complications

- Eye’s perception is not linear (logarithmic)
- CRT’s (etc.) do not respond linearly
- Different displays have different responses
  - different dynamic ranges
  - different color between devices!
- Need to compensate for all of this
Gamma correction

- Response of all parts understood (or just measured)
- Correct: uniform perceived color
  - Normally table driven
    - 0…255 in (linear intensity scale)
    - 0…N out to drive guns
      - N=1024 or 2048 typical
Unfortunately, gamma correction not always done

- E.g., TV is not gamma corrected

Knowing RGB values does not tell you what color you will get!

- For systems you control: do gamma correction
24 bits/pixel => “true color,” but what if we have less?

- 16 bits/pixel
  - 5 each in RGB with 1 left over
  - decent range (32 gradations each)
- Unfortunately often only get 8
  - 3 bits for GB, 2 for R
  - not enough
  - Use a “trick” instead
Color lookup tables (CLUTs)

- Extra piece of hardware
  - Use value in FB as index into CLUT
    - e.g. 8 bit pixel => entries 0…255

- Each entry in CLUT has full RBG value used to drive 3 guns
Palettes

- 8 bits / pixel with CLUT
  - Gives “palette” of 256 different colors
  - Chosen from 16M
  - Can do a lot better than uniform by picking a good palette for the image to be displayed (nice algorithms for doing this)
Imaging Models

*What does the hardware “look like” to the higher levels of software?*
Software models of output (Imaging models)

- Start out by abstracting the HW
  - *What does the hardware “look like” to the higher levels of software?*
- Earliest imaging models abstracted early hardware: vector refresh
  - stroke or vector (line only) models
- “Display list” containing end points of lines to be drawn
  - System just cycles through the display list, moving the electron gun between endpoints
  - Arbitrarily positionable electron gun, rather than the “sweep” pattern seen in raster imaging.
Vector models

- Advantages
  - can freely apply mathematical xforms
    - Scale rotate, translate
    - Only have to manipulate endpoints
- Disadvantages
  - limited / low fidelity images
    - wireframe, no solids, no shading
Current dominant: Raster models

- Most systems provide model pretty close to raster display HW
  - integer coordinate system
  - 0,0 typically at top-left with Y down
  - all drawing primitives done by filling in pixel color values (values in FB)
Issue: Dynamics

- Suppose we want to “rubber-band” a line over complex background

- Drawing line is relatively easy
- But how do we “undraw” it?
Undrawing things in raster model

- Ideas?

(red, su, xo, pal, fwd)
Undrawing things in raster models

Four solutions:

1) Redraw method
   - Redraw all the stuff under
   - Then redraw the line
   - Relatively expensive (but HW is fast)
   - Note: don’t have to redraw all, just “damaged” area
   - Simplest and most robust
How to undraw

2) “Save-unders”
   - When you draw the line, remember what pixel values were “under” it
   - To undraw, put back old values
   - Issue: (what is it?)
How to undraw

- 2) “Save-unders”
  - When you draw the line, remember what pixel values were “under” it
  - To undraw, put back old values
  - Issue: what if “background” changes

- Tends to either be complex or not robust
  - Typically used only in special cases
How to undraw

- 3) Use bit manipulation of colors
  - Colors stored as bits
  - Instead of replacing bits XOR with what is already there
    - $A \oplus B \oplus B = ?$
How to undraw

3) Use bit manipulation of colors
   - Colors stored as bits
   - Instead of replacing bits XOR with what is already there
     - \( A \oplus B \oplus B = A \) (for any \( A \) and \( B \))
   - Draw line by XOR with some color
   - Undraw line by XOR with same color
Issue with XOR?

- What is it?
Issue with XOR

- Colors unpredictable
  - `SomeColor ^ Blue == ??`
    - Don’t know what color you will get
    - Not assured of good contrast
      - Ways to pick 2nd color to maximize contrast, but still get “wild” colors
Undraw with XOR

- Advantage of XOR undraw
  - Fast
  - Don’t have to worry about what is “under” the drawing, just draw
- In the past used a lot where dynamics needed
  - May not be justified on current HW
How to undraw

• 4) Simulate independent bit-planes using CLUT “tricks”
  • Won’t consider details, but can use tricks with CLUT to simulate set of transparent layers
  • Probably don’t want to use this solution, but sometimes used for special cases like cursors
Higher level imaging models

- Simple pixel/raster model is somewhat impoverished
  - Integer coordinate system
  - No rotation (or good scaling)
  - Not very device independent
Higher level imaging models

• Would like:
  • Real valued coordinate system
    • oriented as Descarte intended?
  • Support for full transformations
    • real scale and rotate
  • Richer primitives
    • curves
Stencil-and-Paint
Higher level imaging models

- Would like:
  - Real valued coordinate system
    - oriented as Descarte intended?
  - Support for full transformations
    - real scale and rotate
  - Richer primitives
    - curves
Stencil and paint model

- All drawing modeled as placing paint on a surface through a “stencil”
  - Stencil modeled as closed curves (e.g., splines)
- Issue: how do we draw lines?
Stencil and paint model

- All drawing modeled as placing paint on a surface through a “stencil”
  - Modeled as closed curves (splines)
- Issue: how do we draw lines?
  - (Conceptually) very thin stencil along direction of line
  - Actually special case & use line alg.
Stencil and paint model

- Original model used only opaque paint
  - Modeled hardcopy devices this was developed for (at Xerox PARC)
- Current systems now support “paint” that combines with “paint” already under it
  - e.g., translucent paint (“alpha” values)
Stencil and paint examples

- In most cases, implemented at a much higher layer than the raw hardware (e.g., in the Window System or Toolkit, which we’ll talk about soon...)

- Postscript is based on this approach
  - Implemented in printer’s hardware (often)

- NeXTstep: Display Postscript
  - Brought same imaging model used for hardcopy output to interactive graphics

- Mac OS X
  - Derived from NeXTstep, uses DisplayPDF as its imaging model

- Windows, starting with Vista
  - Aero

- New Java drawing model (Java2D) provides a stencil-and-paint imaging model, implemented completely in the Toolkit
Stencil and paint pros and cons

- Advantages
  - Resolution & device independent
    - does best job possible on avail HW
    - Don’t need to know size of pixels
  - Can support full transformations
    - rotate & scale
Stencil and paint pros and cons

- Disadvantages
  - Slower
    - Less and less of an issue
    - But interactive response tends to be dominated by redraw time
  - Much harder to implement
Stencil and paint pros and cons

- Stencil and paint type models generally the way to go
  - But have been slow to catch on
    - Market forces tend to keep us with old models
    - Much harder to implement
  - Finally became mainstream around 2006