Magic Lenses and Two-Handed Interaction
Spot the difference between these examples and GUls

- A student turns a page of a book while taking notes
- A driver changes gears while steering a car
- A recording engineer fades out the drums while bringing up the strings

[Examples ref. Buxton]
Quick Motivation

- The desktop paradigm does not demand much (physically) of its user.

- Then again, it doesn’t take advantage of the physical abilities of the user either.

- Many tasks are handled more easily with multiple hands.
Two-handed Interaction

- Not just two hands on a keyboard...
  - Discrete actions from both hands (hitting keys)
- More often, either:
  - Continuous action -- both hands in motion
  - Compound action -- one hand moves to target and the other performs an action
- Takes advantage of how we naturally work
  - Drawing/drafting
  - Lab work
  - Surgeons, dentists, ...
  - etc.
Quick Quiz

- What was the first use of two-handed input with a computer?
Quick Quiz

- What was the first use of two-handed input with a computer?

- Douglas Englebart in 1968
  - Point with mouse
  - Operate chord keyboard
Next Quiz

- Why has the PC so committed to having a single pointing device?
Next Quiz

- Why has the PC so committed to having a single pointing device?

- Lots of historical baggage
  - Technical: Early systems couldn’t keep up with multiple continuous devices
  - Experimental: Fitts Law has only two parameters, target distance and size; performance studies typically focus on just a single hand
Lots of Recent Interest

  - Coordinated pen-and-thumb interaction without any additional technology on contact closure PDA (e.g., Palm or PocketPC device).


- MID: Multiple Input Devices
Toolglasses and Magic Lenses

- GUI interaction technique meant to capture a common metaphor for two-handed interaction
  - Basic idea:
    - One hand moves the lens
    - The other operates the cursor/pointer
  - “See through” interfaces
  - The lens can affect what is “below” it:
    - Can change drawing parameters
    - Change change input that happens “through” the lens
- For the purpose of this lecture, I’m combining both of these under the term “magic lens”
Quick Examples

- Magnification (and arbitrary transforms)
- Render in wireframe/outline
- Object editing
  - E.g., click-through buttons: position color palette over object, click through the palette to assign the color to the object

- Important concept: lenses can be \textit{composed} together
  - E.g., stick an outline lens and a color palette lens together to change the color of an object’s outline

- Second important concept: lenses don’t just have to operate on the final rendered output of the objects below them
  - Can take advantage of application data structures to change presentation and semantics
Reading:


Note...

- These techniques are patented by Xerox

- Don’t know scope of patent, but its likely you would need to license to use them commercially
Advantages of lenses

- In context interaction
  - Little or no shift in focus of attention
    - tool is at/near action point
  - Alternate views in context and on demand
    - can compare in context
    - useful for “detail + context” visualization techniques
Detail + context visualization

- Broad category of information visualization techniques
  - Present more detail in area of interest
    - More than you could typically afford to show everywhere
    - Details may be very targeted
  - Present in context of larger visualization
Advantages of lenses

- Two handed interaction
  - Structured well for 2 handed input
    - non-dominant hand does coarse positioning (of the lens)
      - examples also use scroll wheel with non-dominant hand
        - scaling: again a coarse task
    - dominant hand does fine work
Advantages of lenses

- Spatial modes
  - Alternative to more traditional modes
  - Use “where you click through” to establish meaning
  - Typically has a clear affordance for the meaning
    - lens provides a “place to put” this affordance (and other things)
Examples

- Lots of possible uses, quite a few given in paper and video

- Property palettes
  - Click through interaction
  - Again: no context shift + spatial mode
Examples

- **Clipboards**
  - **Visible**
    - invisibility of typical clipboard is a problem
  - Lots of interesting variations
    - multiple clipboards
    - “rubbings”
  - Can do variations, because we have a place to represent them & can do multiple specialized lenses
Examples

- Previewing lenses
  - Very useful for what-if
  - Can place controls for parameters on lens
- Selection tools
  - Can filter out details and/or modify picture to make selection a lot easier
Examples

- Grids
  - Note that grids are aligned with respect to the object space not the lens
Examples

- Debugging lenses
  - Show hidden internal structure in a GUI
  - Not just surface features

Implementation of lenses

- Done in a shared memory system
  - All “applications” are in one address space
  - Can take advantage of application-internal data structures
    - Different than OS-provided magnifying glass, for example
    - Like one giant interactor tree
  - Also assumes a common command language that all applications respond to
Implementation of lenses

- Lens is an additional object “over the top”
  - Drawn last
  - Can leave output from below and add to it (draw over top)
  - Can completely overwrite output from below
    - can do things like “draw behind”
Implementation of lenses

- Input side
  - Changed way they did input
    - originally used simple top-down dispatch mechanisms
    - now lens gets events first
      - can modify (e.g., x,y) or consume
    - possibly modified events then go back to root for “normal dispatch"
Implementation of lenses

• Input side
  • Special mechanism to avoid sending events back to lens
  • Also has mechanism for attaching “commands” to events
    ▪ assumes unified command lang
  • command executed when event delivered
Implementation of lenses

- Output side
- Damage management
  - Lenses need to be notified of all damage
    - Lens may need to modify area due to manipulation of output (e.g. mag)
Implementation of lenses

- Output side
- Redraw
  - Several different types of lenses
    - Ambush
    - Model-in / model-out
    - Reparameterize and clip
Types of lens drawing

- Ambush
  - catch the low level drawing calls
    - typically a wrapper around the equivalent of the Graphics object
  - and modify them
    - e.g. turn all colors to “red”
  - Works transparently across all apps
  - But somewhat limited
Types of lens drawing

- Reparameterize & clip
  - similar to ambush

- modify global parameters to drawing
- redraw, but clipped to lens
- best example: scaling
Types of lens drawing

- Model-in / model-out
  - create new objects and transform them
    - transforms of transforms for composition
  - very powerful, but…
    - cross application is an issue
    - incremental update is an issue
Lenses in subArctic

- Implemented with special "lens parent" & lens interactors
- Input
  - Don’t need to modify input dispatch
  -Lens may need to change results of picking (only positional is affected)
    - in collusion with lens parent
Lenses in subArctic

- Damage management
  - Lens parent forwards all damage to all lenses
  - Lenses typically change any damage that overlaps them into damage of whole lens area
Lenses in subArctic

- Replace vs. draw-over just a matter of clearing before drawing lens or not
- Two kinds of output support
  - Ambush
    - Via wrappers on drawable
    - Extra features in drawable make ambush more powerful
  - Traversal based (similar to MIMO)
Ambush features in drawable

- boolean start_interactor_draw()
- end_interactor_draw()
  - called at start/end of interactor draw
  - allows tracking of what is being drawn
  - drawing skipped if returns false
- allows MIMO effects in ambush
  - isolated drawing
  - predicate selected drawing
Lenses in subArctic

- Also support for doing specialized traversal
  - walk down tree and produce specialized output
  - can do typical MIMO effects
Example: Debugging Lens

- No Arrow Heads
- One Arrow Head
- Two Arrow Heads

**Arrow Head Angle:** 24

**Arrow Head Length:** 20

Applet Viewer: sub_arctic.test.debug_top_test.class

Interactor-Scope

- line_display
- line_display

Show:
- Class Name
- Edges
- X Coordinate
- Y Coordinate
- Width
- Height
- Depth: 1-10
- Children: 1-50

Applet started.
Lenses in Swing

- Two things to do:
  - #1: Make sure that your lens is drawn over other components
    - Easiest way: add a special component as the “Glass Pane” of a JFrame
    - GlassPane is hidden by default; when visible, it’s like a sheet of glass over the other parts of your frame.
    - Generally, set a custom component as the glass pane with a `paintComponent()` method to cause things to be drawn
      - `myFrame.setGlassPane(myNewLensPane)`
      - `myNewLensPane.setVisible(true)`
  - #2 Create your lens class itself
    - Extend `JComponent`
    - Implement whatever listeners you want to get events for
    - Implement `paintComponent` so that when you draw yourself, you actually draw components under you (however you want to draw them) -- note that the lens itself likely won’t have children
Swing GlassPane

- Hidden, by default
- Like a sheet of glass over all other parts of the JFrame; transparent unless you set it to be a component that has an implementation of `paintComponent()`
  - Don’t actually have to do anything in `paintComponent` unless you want the pane itself to be visible
- Useful when you want to catch events or paint over an area that already contains components
  - E.g., deactivate mouse events by installing a class pane that intercepts the events
GlassPane Resources

- Tutorial on how to use the various panes in a JFrame:

- Example of using glass pane:

- Another example of using glass panes for graphical overlay:
  - http://weblogs.java.net/blog/joshy/archive/2003/09/swing_hack_3_ov.html
Making a Lens

- Basically, a specialized component that’s a child of the glass pane
  - Output:
    - The lens should draw itself (title bar, gizmo to make it go away, its borders)
    - Also draw the components in the frame that are under it, although perhaps not in their original form
  - Input:
    - Redispatch events to components in the content pane
    - May need to tweak their coordinates/details (transform to the new component’s coordinate system, for example)
      - See SwingUtilities.convertMouseEvent(), SwingUtilities.convertPoint(), etc.
Lens Resources

- Swing Hacks, hack #56: Create a Magnifying Glass Component
- Blog entry on magic lenses in Swing:
  - http://weblogs.java.net/blog/joshy/archive/2003/11/swing_hack_5_a.html
- Lens details from an earlier version of this class:
- Passing events through to underlying components
- Tweaking component drawing
  - SwingUtilities.paintComponent
  - Lets you call a component’s paint method on an arbitrary graphics object (e.g., one of your own choosing; can disable/reimplement certain functions, look at the call stack, etc., in drawing)
- Drawing the lens itself
  - Consider using JInternalFrame as the base class for your Lens, as you’ll get some basic window decorations.