

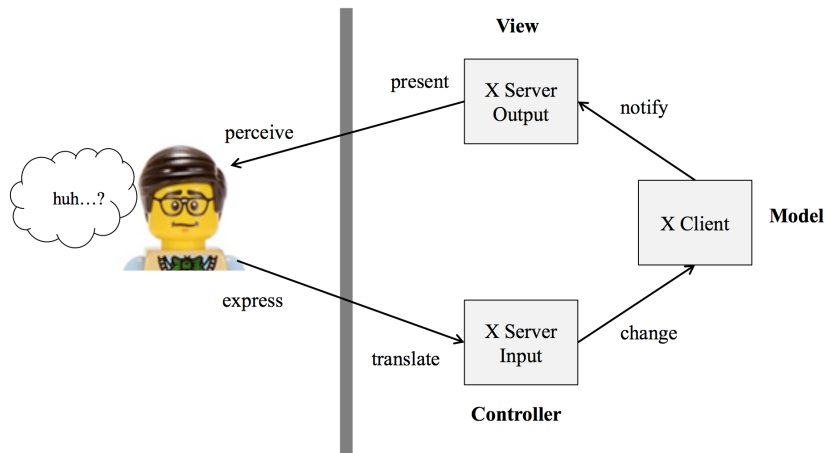
CS 349 Midterm Review

Background & History

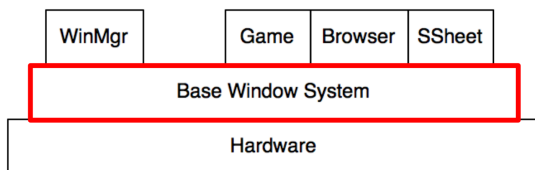
- **User interface:**
 - The place where a person expresses intention to an artifact, and the artifact presents feedback to the person
 - The way people (mental model) ↔ technology (system model) interact
- **Interface:** external presentation (visual, physical, auditory) to the user
- **Interaction:** actions invoked by user and corresponding responses (behaviour)
- Batch interfaces (1945-1965)
 - Sets of instructions fed via punch cards; only used by highly trained individuals
- Conversationalist interface (1965-1985+)
 - Text-based feedback and input; I/O is in system language, not task language
- Graphic user interface (1984+)
 - High resolution graphics display, standard keyboard, & pointing device
 - **WIMP interface:** windows, icons, menus & pointer
 - Benefits of GUI:
 - Keeps the user in control
 - Emphasize recognition (discovery of options) over recall (memorizing commands)
 - Uses metaphor; makes interaction language closer to user's language
- Notable people:
 - Vannevar Bush – conceptualized the memex, a desk with integrated display, input, and data storage
 - Ivan Sutherland – created the Sketchpad, an early graphical interface with a light pen and direct manipulation
 - Douglas Engelbart – invented the mouse, introduced copy/paste
 - Alan Kay – worked on the Xerox Star, first commercial computer with GUI

Windowing Systems & X11

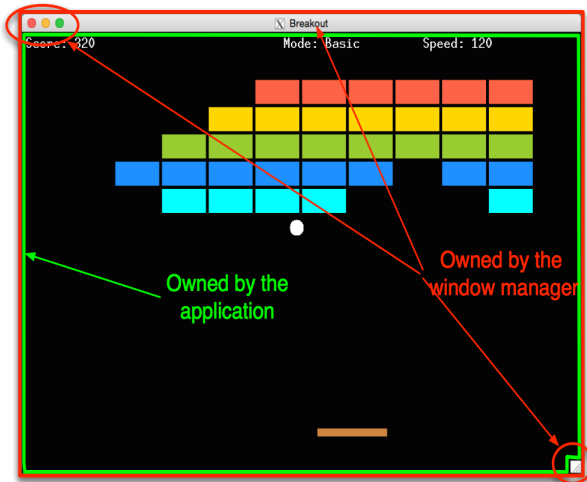
- **Windowing system:** provides input, output, and window management capabilities to the OS
- **X Windows (X11):**
 - Standard windowing system for Unix-based systems
 - X Client handles all application logic
 - X Server handles all user input & display output
 - There may be many clients – each client is an application; server draws all clients onto one screen and reads all input



- X11 is a **base windowing system**:



- A standard/protocol for creating windows, low-level graphical output, and user input
- Does not specify the style of each application's UI
- Provides each application with a window and manages its access
- Each application (only) owns a canvas; shielded from details such as visibility, other windows
- Some design goals of X11/BWS:
 - Display- & device- independent
 - Supports multiple overlapping & resizable windows
 - A display may have multiple screens (monitors) and a window may span multiple screens
 - High-performance, high-quality text, 2D graphic & imaging
- **Window manager**:
 - Provides interactive components (e.g. menus, close button, resizing)
 - Separation of the WM from the BWS enables many alternative “look and feels”



- **Drawing**

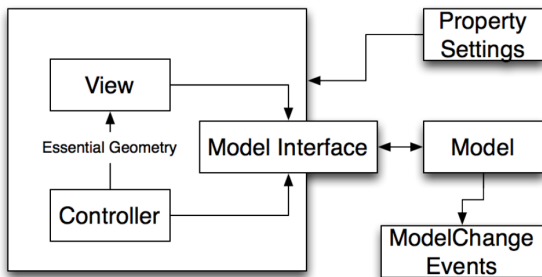
- Three conceptual drawing models:
 - Pixel (e.g. images)
 - Stroke (e.g. lines, outlines of shapes)
 - Region (e.g. text, filled shapes)
- X11 uses graphics contexts to store drawing options/parameters – stored on X server
- **Clipping**: exposing only a particular region (specified by a mask) of an underlying image
 - Only exposed area is repainted – more efficient
- **Painter’s Algorithm**: draw shapes in layers from back to front to create composite shapes

Widgets

- **Widgets**: parts of an interface that have their own behaviour
 - Control their own appearance; receive and handle their own events
 - Design goals:
 - Complete – covers wide range of functionality
 - Consistent – look-and-feel across components
 - Customizable – developers can extend functionality

Heavyweight widgets: <ul style="list-style-type: none"> • Wrappers around OS’s native GUI & windowing system • e.g. Java AWT 	Lightweight widgets: <ul style="list-style-type: none"> • OS provides top-level window in which widgets are drawn • Toolkit is responsible to passing events to widgets
Advantages: <ul style="list-style-type: none"> • Events passed directly to OS/BWS • Preserves the OS look-and-feel 	Advantages: <ul style="list-style-type: none"> • Consistent look-and-feel across platforms • Consistent widget set across platforms
Disadvantages: <ul style="list-style-type: none"> • OS-specific programming • Small set of common widgets across different platforms 	Disadvantages: <ul style="list-style-type: none"> • May appear “non-native”

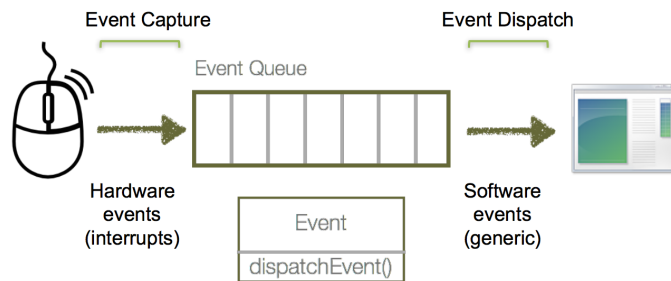
- Widgets as logical input devices



- Characteristics:
 - Model manipulated by the widget (e.g. number, text)
 - Events generated by the widget (e.g. changed)
 - Properties (behaviour and appearance) of the widget (e.g. colour, size, allowed values)
 - e.g. radio button: model = *Boolean*; events = *changed*; properties = *size, colour etc.*

Events

- **Event-driven programming:** flow of program is determined by events such as user input (key press, mouse click, input focus change) or messages from other programs/threads
 - Objective: need to map input from real-world devices → actions within a system
 - Events are pushed into an event queue by the BWS (i.e. event capture)
 - Implementation in X11:
 - Use `XSelectInput()` and event masks to register/subscribe to types of events & filter out unneeded events
 - Use `XNextEvent()` to dequeue the next event; *may block if no events*
 - ◇ Use `XPending()` to check for # of events before dequeuing
 - Should dequeue *all events* before repainting to avoid input lag
 - Should subtract time spent in event loop from `sleep()` to maintain consistent FPS
 - Should draw all images to a *buffer* (`XCreatePixmap()`), then copy the buffer onto the screen in one go (`XCopyArea()`) (aka. double buffering)
 - ◇ Avoids displaying an intermediate image (i.e. flickering)



- **Event dispatch:** dequeuing events from event queue and pushing to appropriate applications
- **Interactor tree** – hierarchy of containers and their nested widgets
- **Positional dispatch** – input sent to widget under mouse cursor location
 - Bottom-up dispatch:
 - Event is routed to leaf (lowest) widget in interactor tree; can pass to parent
 - ◇ e.g. sometimes may be better for the container of a widget to handle the event
 - Advantage: event does not have to traverse through entire tree to arrive at widget
 - Top-down dispatch:
 - Event is routed to highest-level node that contains mouse cursor; can pass to child
 - Advantages:
 - ◇ Parent widget can enforce policies (e.g. make children view-only)
 - ◇ Easy event logging (as it traverses down through the tree)
 - Pure positional dispatch can be problematic
 - e.g. mouse-down inside a button, mouse-up outside
 - e.g. dragging scrollbar but mouse moves out of scrollbar
- **Focus dispatch**
 - At most one widget each can be in keyboard & mouse focus at a given time
 - Focus dispatch also needs positional dispatch to change focus (i.e. mouse click)
 - Accelerator keys (i.e. keyboard shortcuts) can bypass focus dispatch – they're handled before

widget receives events

Heavyweight toolkits: <ul style="list-style-type: none">• BWS has visibility to all widgets• Can use top-down or bottom-up dispatch	Lightweight toolkits: <ul style="list-style-type: none">• BWS only has visibility to application window• Toolkit then dispatches event to widget• Can only use top-down dispatch
---	---

- **Event handling:** interpreting events in widget's application code
 - Design goals of event-code binding:
 - Easy to understand
 - Easy to implement
 - Easy to debug
 - Good performance
 - Event loop & switch statement (X11):
 - All events are consumed in one event loop
 - Switch statement selects the appropriate code for each event
 - Downsides: switch statement needs to encompass every type of event (too many!)
 - Inheritance binding (Java, OS X):
 - Events are dispatched to base widget class with predefined event handling methods
 - Child widget overrides methods with custom behaviour
 - Downsides:
 - ◇ Event handling code in application logic (child widget) – no separation of concerns
 - ◇ Difficult to add new events
 - Listener binding (Java):
 - Interface binding – widget class implements event listener interfaces

```
public class A implements Listener { // implement all methods }
```
 - Object binding – widget class holds listener objects (implement listener interface as a nested class)
 - ◇ Event handling & application code are decoupled

```
this.addListener(new Listener() { // implement all methods });
```
 - Adapter pattern – widget class holds adapter objects (class with boilerplate implementations)
 - ◇ Custom adapter only needs to extend methods that are used

```
this.addListener(new ListenerAdapter() { // override some methods });
```
 - Delegate binding (.NET):
 - Delegates “point” to a method (or methods); invoking delegate calls all associated methods

```
delegate = object.Method1; delegate += object.Method2; delegate(args);
```

Layouts

- **Dynamic layout** – dynamically adjusts screen composition
 - Provides spatial layout for widgets in a container
 - Handles container resize by adjusting location, size, visibility or look-and-feel of widgets
- Widgets may define constraints for size (e.g. min, preferred, max), position (e.g. anchors)
- Layout managers provide algorithms to size & position widgets
- **Composite pattern** – group/container of widgets and individual widgets are treated uniformly
 - Widgets are organized in a tree hierarchy
- **Strategy pattern** – abstract out the algorithm so that it can be changed at run-time
 - Layout manager can employ different layout strategies
- Types of layouts:
 - Fixed – widgets have fixed size & position
 - e.g. set `LayoutManager` to null
 - Intrinsic size – parent widget's size depends *solely* on contained widgets
 - Bottom-up approach – query each child widget for exact size, then set size for parent
 - e.g. `BoxLayout`, `FlowLayout`
 - Variable intrinsic size – widget size depends on both parent and contained widgets' preferred sizes
 - Both bottom-up & top-down approach
 - e.g. `GridBagLayout`, `BorderLayout`
 - Struts and Springs – layout specified by constraints and anchors
 - Strut widgets are fixed in size; spring/glue widgets stretch to fill space
 - e.g. `SpringLayout`

Graphics & Transformations

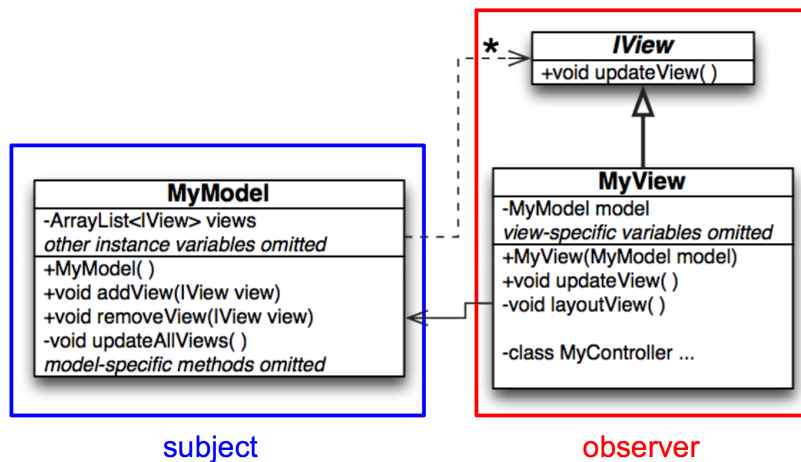
- **NOTE:** origin is located at the top-left when discussing graphics & transformations
- **Shape model** – data needed to draw a primitive shape (array of points, colour, location etc.)
- **Affine transformations:**
 - Translation: add scalar to x and/or y component
 - Scaling: multiply x and/or y components by scalars
 - Rotation (about the origin): $x' = x \cos(\Theta) - y \sin(\Theta)$, $y' = x \sin(\Theta) + y \cos(\Theta)$
 - Order of operations: scale \rightarrow rotate \rightarrow translate
 - $x' = s_x(x \cos(\Theta) - y \sin(\Theta)) + t_x$
 - $y' = s_y(x \sin(\Theta) + y \cos(\Theta)) + t_y$
 - Since scaling & rotation are about the origin, should translate to origin first, and translate back after scaling/rotation
 - Translation can't be done using 2×2 matrix – use homogeneous coordinates
 - $[x, y, w]$ represents a point at $[x/w, y/w]$; e.g. $[1, 2, 1] = [2, 4, 2]$
 - **Affine transformation matrix** (transformations are applied right to left \leftarrow)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\Theta) & -\sin(\Theta) & 0 \\ \sin(\Theta) & \cos(\Theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- **Scene graph** – each component has a transformation matrix & draws its child components relative to itself
 - The interactor tree is a type of scene graph
 - Each component has a transformation matrix (describes its location relative to parent)
 - Each component paints itself, then
 - Combine its matrix with child component's matrix, and tells child to paint itself using combined matrix
- Benefits of geometric manipulation:
 - Allows reuse of objects (create multiple instances via transformations)
 - Allows specification of object in its own coordinate system (e.g. relative to parent)
 - Simplifies repositioning of object after change (e.g. moving an object in animation)
- Coordinates given by events need to be transformed as they traverse the interactor tree
 - e.g. for inside tests/hit detection, mouse event coordinates must be transformed into a model's local coordinates
 - Transforming mouse → model coordinates
 - Only one transformation (of the mouse event) – take the inverse of model's affine matrix
 - Transforming model → mouse coordinates
 - Many transformations (of all objects in the scene) in order to find which one the mouse is inside of

Model-View-Controller

- **MVC** – multiple views *loosely coupled* with the underlying data model
 - Developed for Smalltalk-80 by Trygve Reenskaug
 - Tight coupling of data & presentation prevents easy modification and extension
 - Separation of concerns enables:
 - Alternate forms of interaction/presentation with the same data
 - Multiple, simultaneous views of data
 - Easy testing of data manipulations that are independent of the UI
 - View & controller can access the model through its interface; model only knows about the view
 - Controller → (notifies) → Model
 - View → (queries) → Model
 - Model → (updates) → View
 - Controller & view are tightly coupled in practice
 - ◇ Controller is just part of the view class that calls the model's interface based on input
- MVC is an instance of the **observer pattern**
 - Allows objects to communicate without knowing each others' specific types
 - In Java, the view implements `Observer` (like `IView`); model extends `Observable`



Input

- Computer input can be classified by sensing method (e.g. mechanical, motion, contact), continuous vs. discrete, degrees of freedom
- Specific vs. general input
 - Specific inputs are optimized for certain tasks, but general inputs can be adapted to many tasks
- **Text input**
 - QWERTY has many *perceived* inefficiencies; Dvorak attempts to address these problems, but actual difference in speed is discernible
 - Portability (smaller, lower-profile keys) of keyboards also interfere with typing performance
 - Soft/virtual keyboards lack haptic feedback, but improves aesthetics – good for when the amount of input is limited
- **Positional input**
 - Isometric (force) vs. isotonic (displacement) sensing
 - Device senses displacement (mouse) or force (joystick)
 - Position vs. rate control
 - Change in input device maps to change in position (mouse) or speed (joystick)
 - Usually, isometric → rate, isotonic → position
 - Absolute vs. relative position
 - 1:1 mapping between input & output position (touchscreen) or non-1:1 mapping (mouse)
 - Direct vs. indirect contact
 - Input takes place on the same surface as output (touchscreen) or on a different surface (mouse)
 - Dimensions sensed – 1 (dial) vs. 2 (mouse) vs. 3 (Wiimote)
 - Control-display gain = ratio between the movement speeds of pointer on screen & physical mouse
- **Keystroke Level Model (KLM):**
 - Use operators to estimate how long an input task should take
 - Keystroke, pointing, mouse button press, hand move between mouse & keyboard, mental preparation