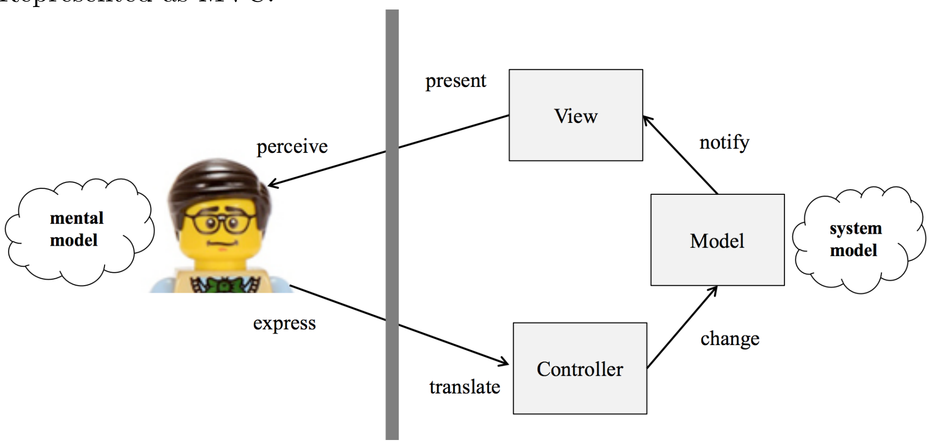
CS 349 Midterm Review

Background & History

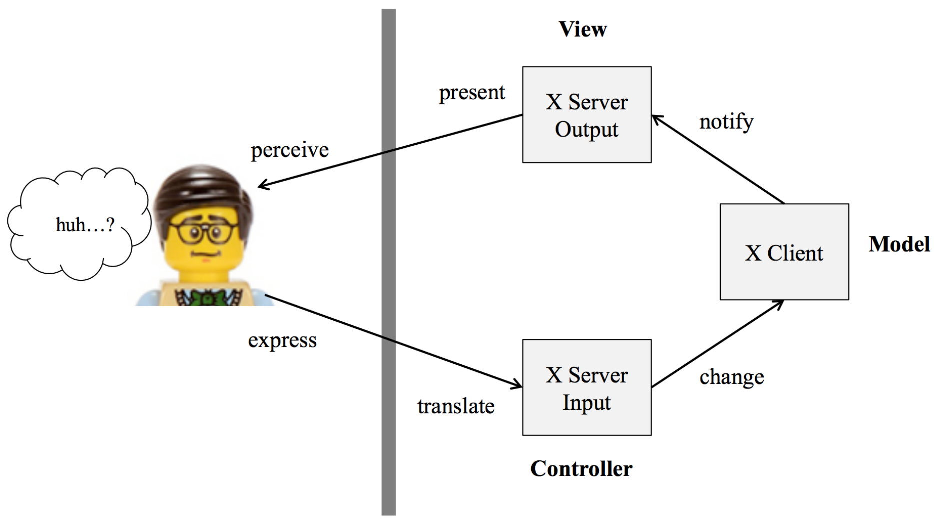
- **User interface**:
  - The place where a person expresses intention to an artifact, and the artifact presents feedback to the person
  - The way people (mental model) and technology (system model) interact
  - Represented as MVC:



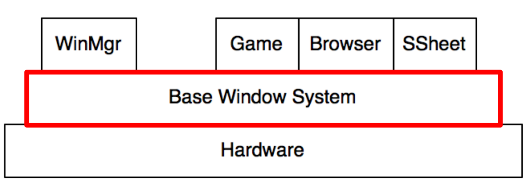
- **Interface**: external presentation (visual, physical, auditory) to the user
  - e.g. controls
- **Interaction**: actions invoked by user and corresponding responses (behaviour)
  - e.g. action and dialog
- Batch interfaces (1945-1965)
  - Sets of instructions fed via punch cards; only used by highly trained individuals
- Conversationalist interface (1965-1985+)
  - Text-based feedback and input; I/O is in system language, not task language
- Graphic user interface (1984+)
  - **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
- X11 architecture
  - 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



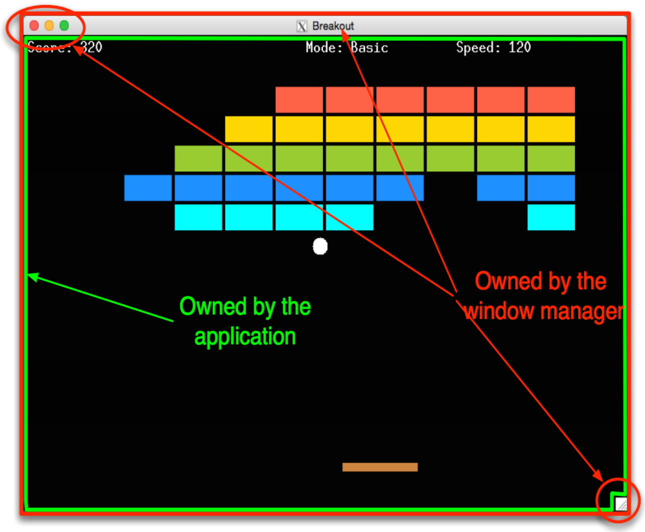
- Structure of an X program (application is run on the X client):
  - Perform client initialization
  - Connect to X server (e.g. `XOpenDisplay()`, `XCreateWindow()`)
  - Perform X related initialization (e.g. create graphic contexts with `XCreateGC()`; put window on the screen with `XMapRaised()`)
  - Event loop
    - Get next event from server (e.g. `XNextEvent()`)
    - Handle event (e.g. `XLookupKeysym()`)
    - Send draw request to server (e.g. flush output buffer with `XFlush()`)
  - Close down connection to X server (e.g. `XCloseDisplay()`)
  - Perform client cleanup
- 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)
  - The WM owns each application's window itself (while application owns the canvas)
    - i.e. application developers usually cannot change the window style
  - 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
    - `XSetClipMask()`, `XSetClipRectangles()`
    - Only exposed area is repainted – more efficient
  - **Painter's Algorithm**: draw shapes in layers from back to front to create composite shapes
    - `Displayable` class with abstract `paint()` method; implement `paint()` in each subclass
    - Draw list of `Displayables` from back to front, clear screen on every repaint

- **Events & animation**
  - Objective: need to map input from real-word devices to actions within a system
  - **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

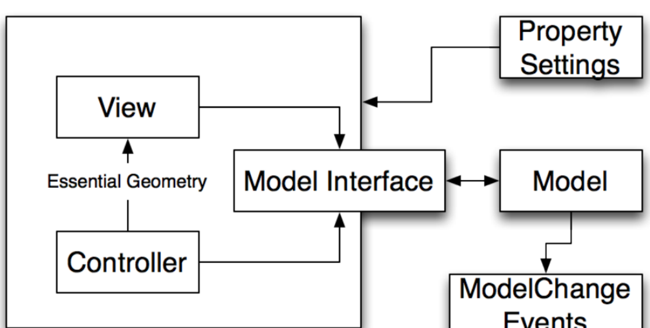
- Events are pushed into an <u>event queue</u> by the BWS (i.e. <u>event capture</u>)
- Implementation in X11:
  - Use `XSelectInput()` and event masks (e.g. `KeyPressMask` etc.) to register/subscribe to types of events
    - Filters out unneeded events
  - Use `XNextEvent()` to dequeue the next event; *may block if no events*
    - Use `XPending()` to check for # of events before dequeueing
  - 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. <u>double buffering</u>)
    - Avoids displaying an intermediate image (i.e. flickering)

Widgets & Events

- **Widgets**: parts of an interface that have their own behaviour
  - Control their own appearance; recieve and handle their own events
  - Widgets toolkit defines a set of GUI components
  - Design goals:
    - Complete – covers wide range of functionality
    - Consistent – look-and-feel across components
    - Customizable – developers can extend functionality
  - Consistent behaviour of components helps users anticipate how the interface will react, and promotes easier *discoverability* of features

Heavyweight widgets: • Wrappers around OS's native GUI & windowing system • e.g. Java AWT	**Lightweight widgets**: • OS provides top-level window in which widgets are drawn • Toolkit is responsible to passing events to widgets
Advantages: • Events passed directly to OS/BWS • Preserves the OS look-and-feel	Advantages: • Consistent look-and-feel across platforms • Consistent widget set across platforms • Allows for highly optimized widgets
Disadvantages: • OS-specific programming • Small set of common widgets across different platforms	Disadvantages: • 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.*



  - Model is abstracted into an interface/abstract class for more code reuse and customizability
    - Interface may provide many accessors, mutators & event-firing functions to be implemented by the custom widgets, allowing for easy manipulation of custom data
  - Essential geometry is computed by the view; controller interacts with it

- Event dispatch → event handling → notifying view & windowing system (MVC)
- **Event dispatch**: dequeueing 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
    - Widget can process the event or pass to its parent
    - e.g. widget belongs in a group/container – may be better for container 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
    - Widget can process the event or pass to child component
    - 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; dragging scrollbar but mouse moves out of scrollbar
- **Focus dispatch** – events dispatched to widget that has keyboard/mouse focus
  - 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:	**Lightweight toolkits**:
• BWS has visibility to all widgets • Can use top-down or bottom-up dispatch	• 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

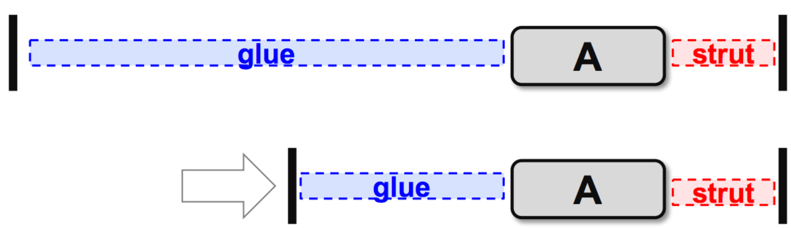
    ```
    this.addListener(new ListenerAdapter() { // override some methods });
    ```

- Delegate binding (.NET):
  - Delegates "point" to a method (or methods); invoking delegate calls all associated methods

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

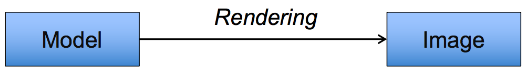
Layouts

- **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
- **Adaptive/responsive layout** – may go beyond spatial layout in order to adapt to different device sizes
- 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 on contained widgets
    - Bottom-up algorithm – query each child widget for preferred size, then set size for parent
    - e.g. `BoxLayout`, `FlowLayout`
  - Variable intrinsic size – widget size depends on both parent and contained widgets
    - Bottom-up & top-down algorithm
    - e.g. `GridBagLayout`, `BorderLayout`
  - Struts and Springs – layout specified by contraints and anchors
    - Strut widgets are fixed in size; spring/glue widgets stretch to fill space
    - e.g. `SpringLayout`



Graphics & Transformations

- **2D graphics**:
  - Shape model – data needed to draw a shape (array of points, colour, location etc.)
  - Rendering – using the properties to create an image to be displayed on the screen



- **NOTE**: origin is located at the top-left when discussing graphics & transformations
- Selection paradigms:
  - Click selection (for lines) – find closest line segment to mouse position
    - Check distance from mouse to each line segment using vector projection
    - Count as "selection" for distance under a certain threshold
  - Click selection (for closed shapes) – check if mouse position is within shape
    - For complex polygons, draw a ray extending from the point & count the # of intersections with the polygon's boundary
    - If odd # of intersections, the point is within the polygon; if even #, it is not
- **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 → rotate → translate
    - $x' = s_x(\cos(\Theta) - \sin(\Theta)) + t_x$
    - $y' = s_y(\sin(\Theta) + \cos(\Theta)) + t_y$
    - 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**: calculates all transformations using 3×3 matrix

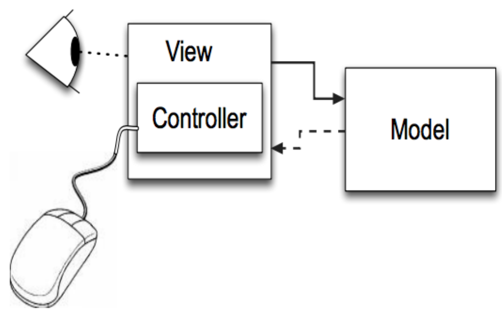
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\Theta) & -\sin(\Theta) & 0 \\ \sin(\Theta) & \cos(\Theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix}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

    - Transformations are applied right to left ←
- **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)
    - 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)

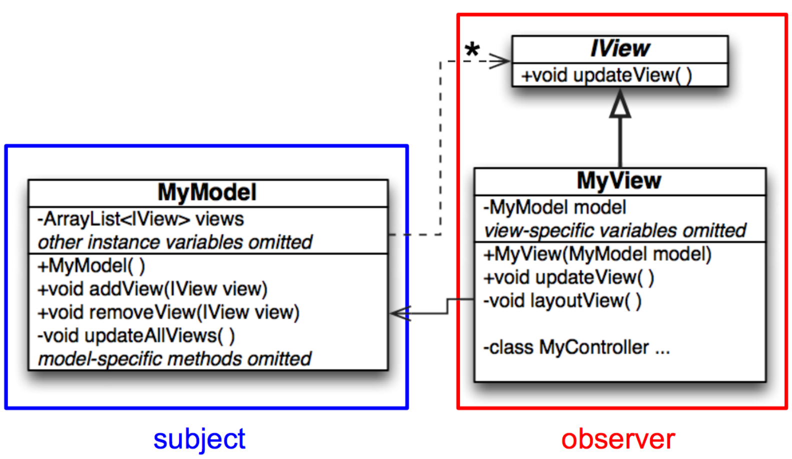
    - Allows specification of object in its own coordinate system (e.g. relative to parent)
    - Simplifies repositioning of object after change (e.g. moving an object in animation)
- Coordinates given by events need to be transformed as they traverse the interactor tree
    - e.g. for <u>inside tests/hit detection</u>, mouse event coordinates must be transformed into a model's local coordinates
    - Transforming <u>mouse → model</u> coordinates
        - Only one transformation (of the mouse event) – take the inverse of model's affine matrix
    - Transforming <u>model → mouse</u> 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
- Devices are mostly focused on text & positional input
- **Text input**
  - QWERTY has many *perceived* problems:
    - Many common combinations require inefficient finger movements
    - Most typing is done with left hand
    - Most typing is *not* done on the home row
  - 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 aethestics – good for when the amount of input is limited
- **Positional input**
  - <u>Isometric</u> (force) vs. <u>isotonic</u> (displacement) sensing
    - Device senses displacement (mouse) or force (joystick)
  - <u>Position</u> vs. <u>rate</u> control
    - Change in input device maps to change in position (mouse) or speed (joystick)
    - Usually, isometric → rate, isotonic → position
  - <u>Absolute</u> vs. <u>relative</u> position
    - 1:1 mapping between input & output position (touchscreen) or non-1:1 mapping (mouse)
  - <u>Direct</u> vs. <u>indirect</u> 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)