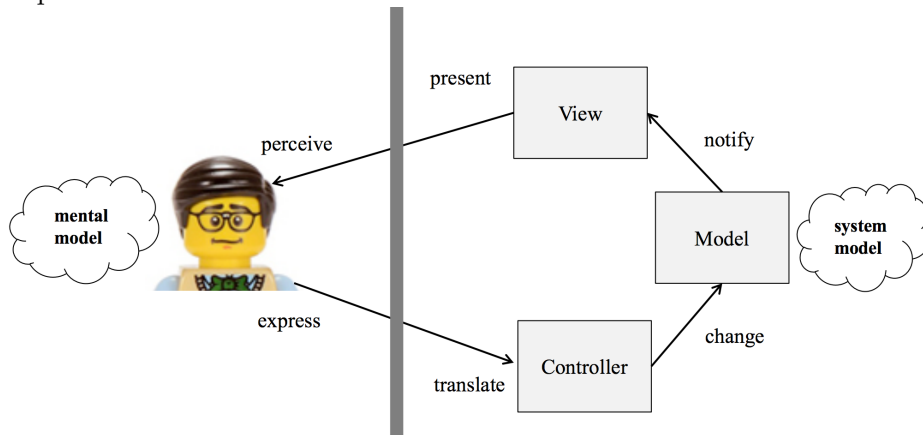


CS 349 Midterm Review

Background & History

- **User interface:**

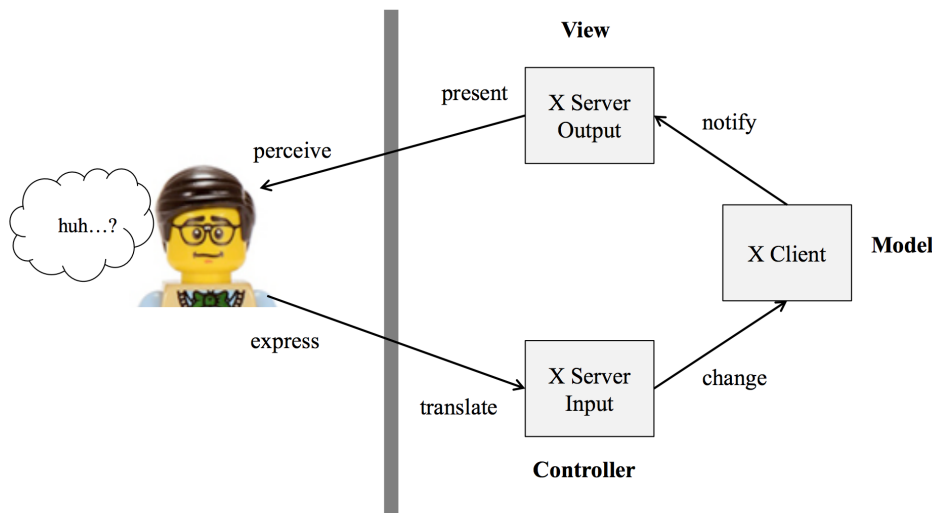
- The place where a person expresses intention to an artifact, and the artifact presents feedback to the person
- The way people (mental model) and technology (system model) interact
- Represented as MVC:



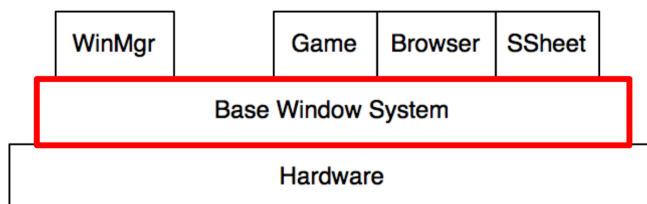
- **Interface:** external presentation (visual, physical, auditory) to the user
 - e.g. controls
- **Interaction:** actions invoked by user and corresponding responses (behaviour)
 - e.g. action and dialog
- Batch interfaces (1945-1965)
 - Sets of instructions fed via punch cards
 - Only used by highly trained individuals
- Conversationalist interface (1965-1985+)
 - Text-based feedback and input
 - I/O is in system language, not task language
 - 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
- Graphic user interface (1984+)
 - Hardware interface: high resolution & refresh graphics display, keyboard, and pointing device
 - WIMP interface: windows, icons, menus, and 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

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

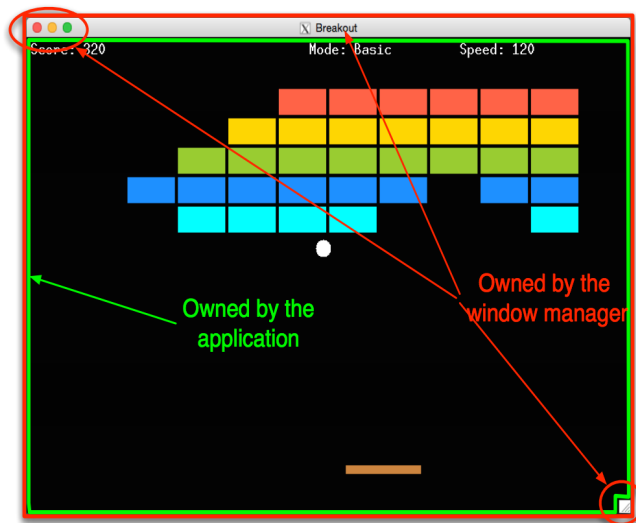


- Structure of an X program (application is run on the X client):
 - Perform client initialization
 - Connect to X server (e.g. `XOpenDisplay()`, `XCreateWindow()`)
 - Perform X related initialization (e.g. create graphic contexts with `XCreateGC()`; put window on the screen with `XMapRaised()`)
 - Event 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, etc.
- Some design goals of X11/BWS:
 - Supports multiple overlapping & resizable windows
 - A display may have multiple screens (e.g. 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
 - Implementation in X11:
 - `XSetClipMask()`, `XSetClipRectangles()`
 - Only exposed area is repainted
 - **Painter's Algorithm**: draw shapes in layers from back to front to create composite shapes
 - Implementation in X11:
 - `Drawable` class with abstract `paint()` method
 - Implement `paint()` in each subclass

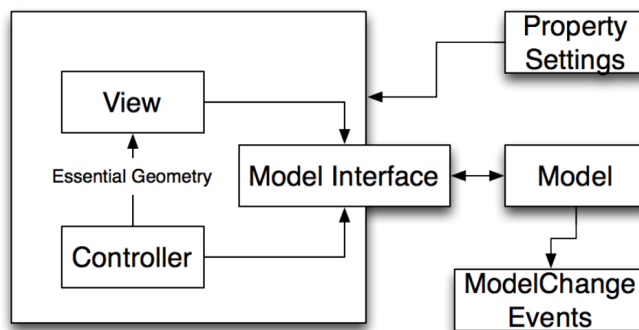
- Draw list of `Displayables` from back to front, clear screen on every repaint
- **Events & animation**
 - Objective: need to map input from real-world devices to actions within a system
 - Event-driven programming: flow of program is determined by events such as user input (key press, mouse click, input focus change) or sensor/timer events
 - Implementation in X11:
 - Use `XSelectInput()` and event masks (e.g. `KeyPressMask` etc.) to register for types of 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)

Widgets, Events & Layout

- **Widgets:** parts of an interface that have their own behaviour
 - Control their own appearance; receive 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: <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• Allows for highly optimized widgets
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)



- 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
- Examples of widgets and their characteristics:
 - e.g. button

- ◊ Model = *none*; events = *push*; properties = *label, size, colour etc.*
 - e.g. radio button
 - ◊ Model = *Boolean*; events = *changed*; properties = *size, colour etc.*
 - e.g. text field
 - ◊ Model = *string*; events = *changed, selection*; properties = *optional formatters, font etc.*
- Special value widgets: colour picker, calendar etc.
- Event dispatch → event handling → notifying view & windowing system
- **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 positional dispatch:
 - ◊ Event routed to leaf 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
 - Top-down positional dispatch:
 - ◊ Event routed to highest-level node that contains mouse cursor
 - ◊ Widget can process the event or pass to child component
 - ◊ e.g. parent widget can enforce policies
 - ◊ e.g. easy logging of events (as it traverses down through the tree)
 - 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
 - Accelerator keys can bypass focus dispatch

Heavyweight toolkits:

- BWS has visibility to all widgets
- Can use top-down or bottom-up dispatch

Lightweight toolkits:

- BWS only has visibility to application window
- Toolkit dispatches event to widget
- Can only use top-down dispatch

- **Event handling:** interpreting events in widget's application code
 - Event loop & switch statement (X11):
 - All events are consumed in one event loop
 - Switch statement selects the appropriate code for each event (`switch (event.type) ...`)
 - 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 (default behaviour)
 - Child widget overrides methods with custom behaviour
 - Downsides:
 - ◊ Multiple events are handled by each method – becomes switch statement again
 - ◊ 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:

Graphics & Transformations

-