CS 160: Lecture 13

Professor John Canny
Fall 2004
Outline

- Model-view controller
  - Why do we need it?
  - Changing the display
  - Event flow
  - Dragging at interactive speeds

- Interactive application programming
  - Callbacks
  - Multi-threaded programming
Model-View-Controller

- Architecture for interactive apps
  - introduced by Smalltalk developers at PARC
- Partitions application in a way that is
  - scalable
  - maintainable
Example Application

Blue circles: 4
Cardinal squares: 2
- Information the app is trying to manipulate
- Representation of real world objects
  - circuit for a CAD program
    - logic gates and wires connecting them
  - shapes in a drawing program
    - geometry and color
Implements a visual display of the model

May have multiple views
  * e.g., shape view and numerical view
Multiple Views

Blue circles: 4
Cardinal squares: 2
1. Implements a visual display of the model
2. May have multiple views
   * e.g., shape view and numerical view
3. Any time the model is changed, each view must be notified so that it can change later
   * e.g., adding a new shape
Controller

- Receives all input events from the user
- Decides what they mean and what to do
  * communicates with view to determine which objects are being manipulated (e.g., selection)
  * calls model methods to make changes on objects
    + model makes change and notifies views to update
Controller

Blue circles: 4
Cardinal squares: 2
Controller

Blue circles: 4
Cardinal squares: 2
"pattern of behavior in response to user events (controller issues) is independent of visual geometry (view issues)"

Controller must contact view to interpret what user events mean (e.g., selection)
Combining View & Controller

- View and controller are tightly intertwined
  - lots of communication between the two
- Almost always occur in pairs
  - i.e., for each view, need a separate controller
- Many architectures combine into a single class
Why MVC?

- Combining MVC into one class or using global variables will not scale
  - model may have more than one view
    + each is different and needs update when model changes
- Separation eases maintenance
  - easy to add a new view later
  - new model info may be needed, but old views still work
  - can change a view later, e.g., draw shapes in 3-d (recall, view handles selection)
Admin: Any groups who have < 4 members?

Break
Adding Views Later

Blue circles: 4
Cardinal squares: 2
Changing the Display

How do we redraw when shape moves?
Moving Cardinal Square

Blue circles: 4
Cardinal squares: 2
Erase w/ Background Color and Redraw

Blue circles: 4
Cardinal squares: 2
Changing the Display

4 Erase and redraw
   * using background color to erase fails
   * drawing shape in new position loses ordering

4 Move in model and then redraw view
   * change position of shapes in model
   * model keeps shapes in a desired order
   * tell all views to redraw themselves in order
   * slow for large / complex drawings
   + flashing!
Damage / Redraw Method

1. View informs windowing system of areas that need to be updated (i.e., damaged)
   * does not redraw them at this time...

2. Windowing system
   * batches updates
   * clips them to visible portions of window

3. Next time waiting for input
   * windowing system calls *Redraw* method for win.
     * passes region that needs to be updated
Damage old, Change position in model, Damage new

Blue circles: 4
Cardinal squares: 2
Event Flow

Creating a new shape
Event Flow (cont.)

- Assume blue circle selected

Blue circles: 0
Cardinal squares: 0
Event Flow (cont.)

- Press mouse over tentative position
- Windowing system identifies proper window for event
- Controller for drawing area gets mouse click event
- Checks mode and sees “circle”
- Calls models AddCircle method with new position
Event Flow (cont.)

- AddCircle adds new circle to model’s list of objects
- Model then notifies list of views of change
  - drawing area view and text summary view
- Views notifies windowing system of damage
  - both views notify WS without making changes yet!
    + model may override
Event Flow (cont.)

- Views return to model, which returns to controller
- Controller returns to event handler
- Event handler notices damage requests pending and responds
- If one of the views was obscured, it would be ignored

Blue circles: 0
Cardinal squares: 0
Event Flow (cont.)

- Event handler calls view's Redraw methods with damaged area
- Views redraw all objects in model that are in damaged area
Dragging at Interactive Speeds

- Damage old, move, damage new method may be too slow
  - must take less than 200 ms to be smooth

- Solutions
  - don’t draw object, draw an outline (cartoon)
    + use XOR to erase fast (problems w/ color)
    + save portion of frame buffer before dragging
Callbacks

* **Callbacks**: every widget registers itself to receive certain events with the OS.

* **Dangers**: UI code using callbacks is usually single-threaded. Control does not return to the event loop until the callback returns. Therefore in a callback routine, **never**:
  * Wait for another event.
  * Sleep.
  * Perform I/O or other actions that may block.
Threaded code

- Multi-threaded code uses distinct threads for distinct widgets.
- It allows blocking operations (e.g. I/O) to be confined to particular threads that don’t affect interactivity.
- If your program has any time-intensive parts, they should run in different threads from the UI code.
Threaded code

Use separate threads for any operations that can occur asynchronously:

* UI interactions.
* File operations - use separate threads if you need to be updating several files at the same time.
* Inter-process communication (sockets): use one thread for each connection.
* Use a thread for each other I/O device, e.g. one each for reading from or writing to the sound card.
Inter-thread communication

- The window system and running process in the OS communicate using message passing:
  * The event queue and sockets are examples of message-passing primitives.

- Processes in the same address space (i.e. within your program) can use **shared memory** (i.e. shared variables) which is much more efficient.
Shared-memory communication poses challenges. If you rely on “mailbox” primitives, things can go wrong:

- Flag to show this thread is writing new data
- Data

Diagonal arrows indicate the relationship between the flag and the data.
Synchronization

Intuitively, threads that want to write should:

wait until thread_id = 0;
write thread_id;
write data;
Synchronization

But thread switching can happen anytime, e.g.

... wait until thread_id = 0;
write thread_id;
write data;
Synchronization

A switch between checking the flag and setting it allows both threads to (incorrectly) write the flag and their data.

To prevent this, we define critical sections of the code that cannot be interrupted.
Synchronization

e.g. the critical section in the example is:

....
wait until thread_id = 0;
write thread_id;
write data;

{ Critical section, thread can’t be pre-empted.

Flag to show this thread is writing new data

Data

0 <blank>
Semaphores

Rather than many critical sections in your code, you can use a single semaphore primitive.

A semaphore is initialized to an integer \( n \), and has just two operations:

- `test();` wait until semaphore \( > 0 \), then decrement it.
- `increment();` increment the semaphore.
Semaphore example

Then the example code becomes (with sem1 initialized to 1):

```java
....
sem1.test();
write thread_id;
write data;
sem1.increment();
```

This section cannot be pre-empted by the other process that is using this semaphore.
Summary

- Model-view controller
  - Why do we need it?
  - Changing the display
  - Event flow
  - Dragging at interactive speeds

- Interactive application programming
  - Callbacks
  - Multi-threaded programming