Input part 3: Implementing Interaction Techniques

Recap: Interaction techniques

- A method for carrying out a specific interactive task
  - Example: enter a number in a range
    - could use… (simulated) slider
    - (simulated) knob
    - type in a number (text edit box)
  - Each is a different interaction technique

Suppose we wanted to implement an interaction for specifying a line

- Could just specify two endpoints
  - click, click
  - not good: no affordance,no feedback
- Better feedback is to use "rubber banding"
  - stretch out the line as you drag
  - at all times, shows where you would end up if you "let go"

Aside

- Rubber banding provides good feedback
- How would we provide better affordance?

Aside

- Rubber banding provides good feedback
- How would we provide better affordance?
  - Changing cursor shape is about all we have to work with

Implementing rubber banding

```
Accept the press for endpoint p1;
P2 = P1;
Draw line P1-P2;
Repeat
  Erase line P1-P2;
  P2 = current_position();
  Draw line P1-P2;
Until release event;
Act on line input;
```

Implementing rubber banding

- Need to get around this loop absolute min of 5 times / sec
  - 10 times better
  - more would be better
- Notice we need "undraw" here

What’s wrong with this code?

```
Accept the press for endpoint p1;
P2 = P1;
Draw line P1-P2;
Repeat
  Erase line P1-P2;
  P2 = current_position();
  Draw line P1-P2;
Until release event;
Act on line input;
```

Not event driven

- Not in the basic event / redraw cycle form
  - don't want to mix event and sampled
  - in many systems, can't ignore events for arbitrary lengths of time
- How do we do this in a normal event / redraw loop?

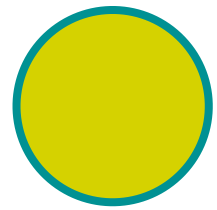
You don’t get to write control flow anymore

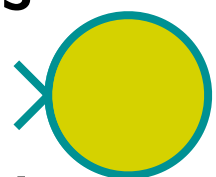
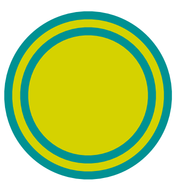
- Basically have to chop up the actions in the code above and redistribute them in event driven form
  - “event driven control flow”
  - need to maintain “state” (where you are) between events and start up “in the state” you were in when you left off

Finite state machine controllers

- One good way to maintain "state" is to use a state machine
  - (deterministic) finite state machine
    - FSM

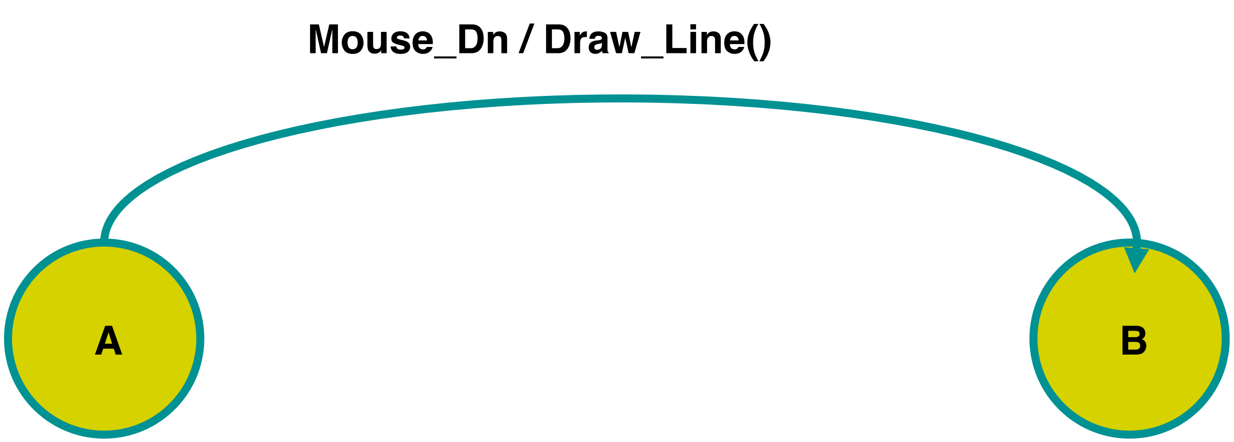
FSM notation



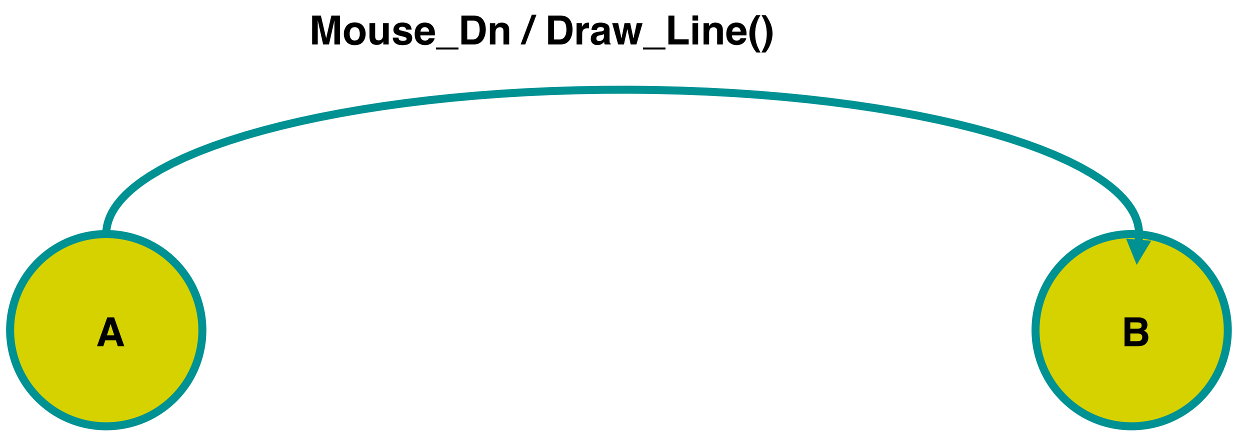
- Circles represent states
  - arrow for start state 
  - double circles for “final states”
    
    - notion of final state is a little off for user interfaces (don’t ever end)
    - but still use this for completed actions
    - generally reset to the start state

FSM notation

- Transitions represented as arcs
  - Labeled with a "symbol"
    - for us an event (can vary)
  - Also optionally labeled with an action



FSM Notation



- Means: when you are in state A and you see a mouse down, do the action (call draw_line), and go to state B

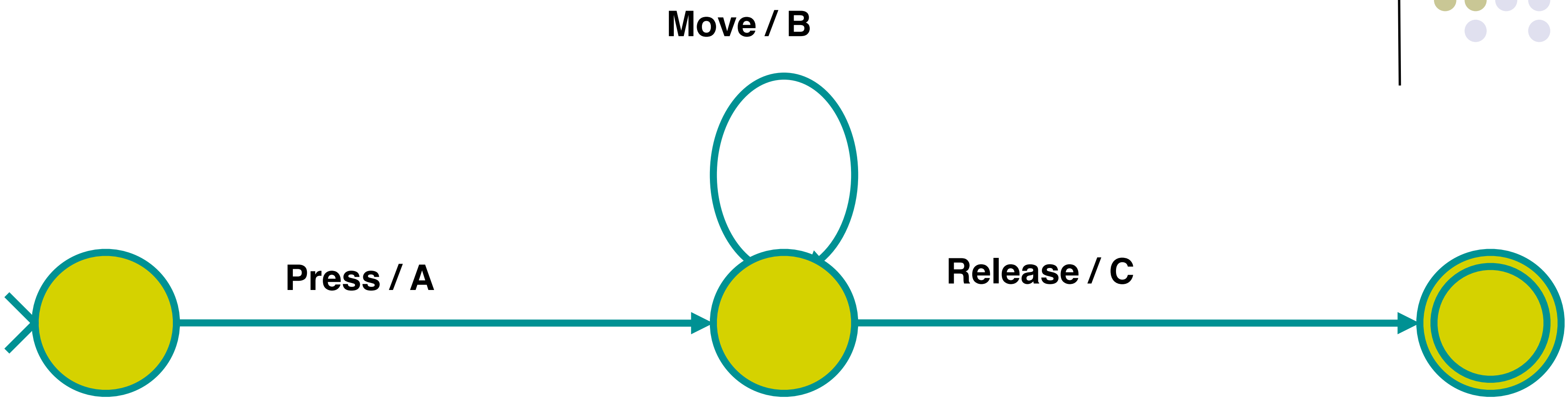
FSM Notation

- Sometimes also put actions on states
  - same as action on all incoming transitions

Rubber banding again (cutting up the code)

```
   Accept the press for endpoint p1;
A: P2 = P1;
   Draw line P1-P2;
   Repeat
B:    Erase line P1-P2;
      P2 = current_position();
      Draw line P1-P2;
   Until release event;
C: Act on line input;
```

FSM control for rubber banding



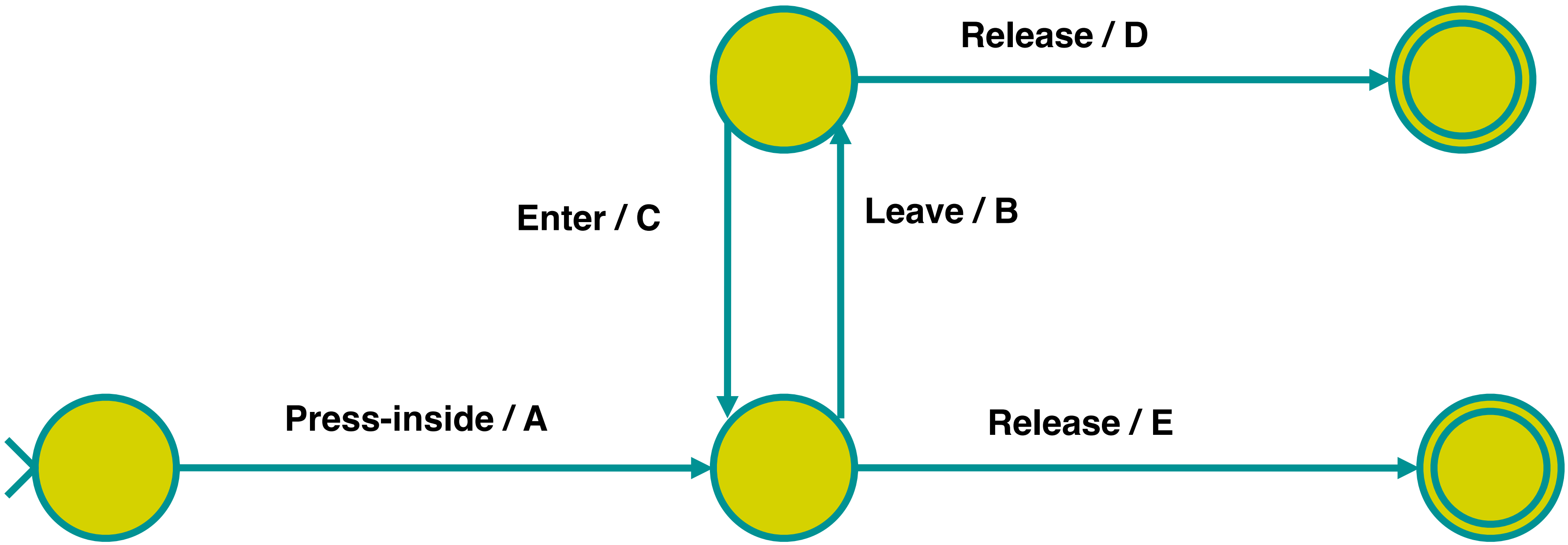

```
A:  P2 = P1;
    Draw line P1-P2;
B:  Erase line P1-P2;
    P2 = current_position();
    Draw line P1-P2;
```

Second example: button

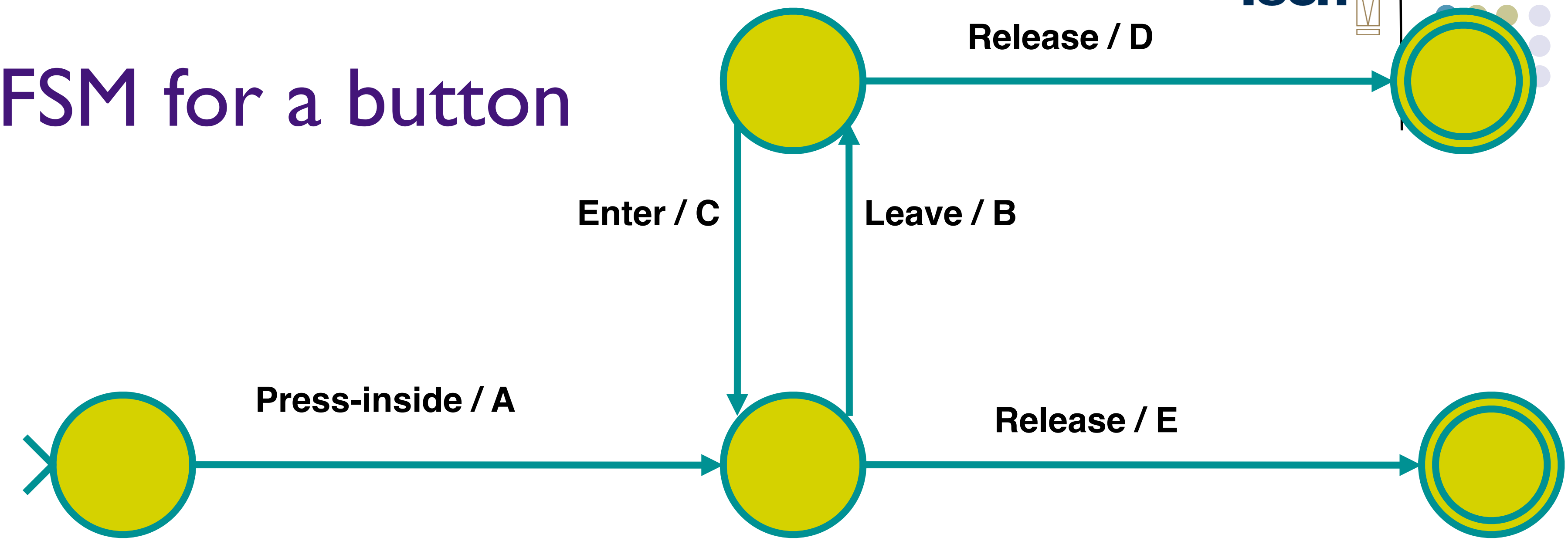
Press inside	→	highlight
Move in/out	→	change highlight
Release inside	→	act
Release outside	→	do nothing

FSM for a button?

FSM for a button



FSM for a button



A: highlight button

B: unhighlight button

C: highlight button

D: <do nothing>

E: do button action

In general...

- Machine states represent context of interaction
  - "where you are" in control flow
- Transitions indicate how to respond to various events
  - what to do in each context

“Events” in FSMs

- What constitutes an “event” varies
  - may be just low level events, or
  - higher level (synthesized) events
    - e.g. region-enter, press-inside
  - Example: Swing ActionEvents
    - Generated from a range of *different* low-level events
      - Completion of button activation FSM
      - Hitting *enter* in a text field

Guards on transitions

- Sometimes also use "guards"
  - predicate (boolean expression) before event
  - adds extra conditions req to fire
  - typical notation: pred: event / action
    - e.g. button.enabled: press-inside / A

  - Note: FSM augmented with guards is Turing complete

FSM are a good way to do control flow in event driven systems

- Can do (formal or informal) analysis
  - are all possible inputs (e.g. errors) handled from each state
  - what are next legal inputs
    - can use to enable / disable
- Can be automated based on higher level specification

Implementing FSMs

```
state = start_state;
for (;;) {
  raw_evt = wait_for_event();
  evt = transform_event(raw_evt);
   state = fsm_transition(state, evt);
}
```

- Note that this is basically the normal event loop

Implementing FSMs

```
fsm_transition(state, evt)
    switch (state)
        case 0:   // case for each state



        case 1: // case for next state
```

Implementing FSMs

```
fsm_transition(state, evt)
   switch (state)
      case 0:   // case for each state
         switch (evt.kind)
            case loc_move:   // trans evt
               … action …    // trans action
               state = 42;   // trans target
            case loc_dn:
               ...
      case 1: // case for next state
         switch (evt.kind) …
return state;
```

Implementing FSMs

```
fsm_transition(state, evt)
   switch (state)
      case 0:   // case for each state
         switch (evt.kind)
            case loc_move:  // trans evt
               … action …   // trans action
               state = 42;  // trans target
            case loc_dn:
               ...
      case 1: // case for next state
         switch (evt.kind) …
return state;
```

Table driven implementation

- Very stylized code
- Can be replaced with fixed code + table that represents FSM
  - only have to write the fixed code once
  - can have a tool that generates table from something else

Table driven implementation

- Table consists of array of states
- Each state has list of transitions
- Each transition has
  - event match method
  - list of actions (or action method)
  - target state

Table driven implementation

```
fsm_transition(state, evt)
  for each transition TR in table[state]
    if TR.match(evt)
      TR.action();
      state = TR.to_state();
      break;
  return state
```

- Simpler: now just fill in table