Event Binding

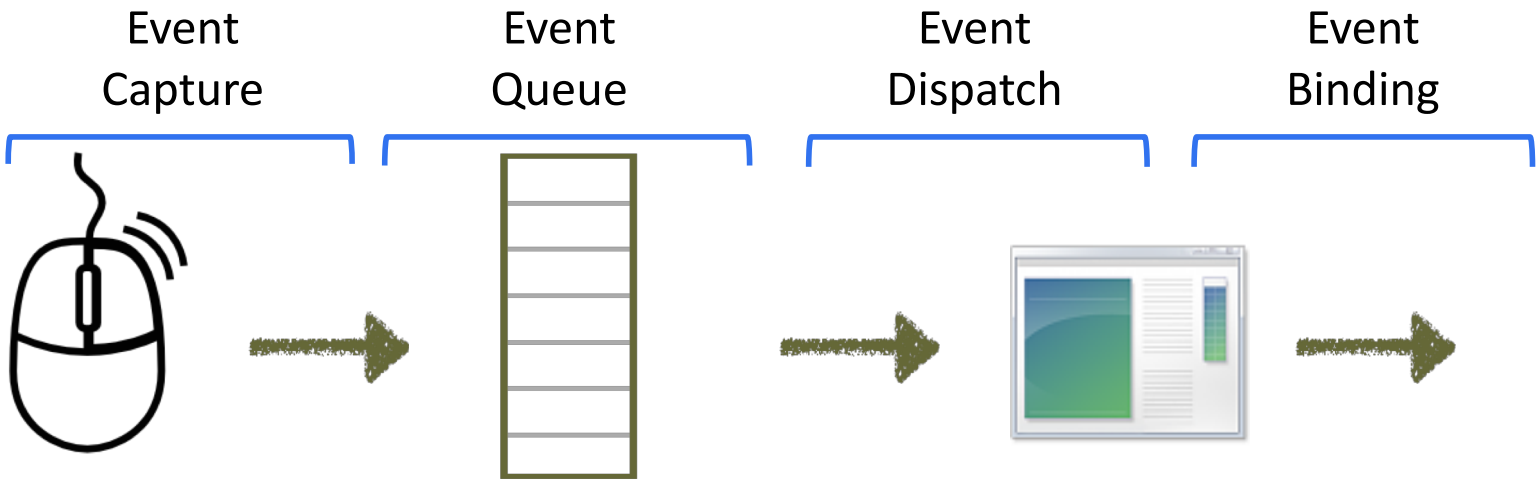
Approaches

Listener model

Global hooks

Event Dispatch vs. Event Binding

- Event Dispatch phase addresses:
  - Which window receives an event?
  - Which widget processes it?
    - Positional dispatch
    - Focus dispatch
- Event Binding answers:
  - After dispatch to a widget, how do we **bind** an event to code?



Event-to-Code Binding

- How do we design our GUI architecture to enable application logic to interpret events once they've arrived at the widget?
- Design Goals:
  - Easy to understand (clear connection between each event and code that will execute)
  - Easy to implement (binding paradigm or API)
  - Easy to debug (how did this event get here?)
  - Good performance

- We'll walkthrough some common implementations of event-binding.

Switch Statement Binding

- All events consumed in one event loop in the application (not by widgets)
  - Switch selects window and code to handle the event
- Used in Xlib and many early systems

```
while( true ) {
    XNextEvent(display, &event); // wait next event
    switch(event.type) {
    case Expose:
       // ... handle expose event ...
       cout << event.xexpose.count << endl;
         break;
    case ButtonPress:
   // ... handle button press event ...
       cout << event.xbutton.x << endl;
         break;
    ...
```

Switch Statement Binding: WindowProc

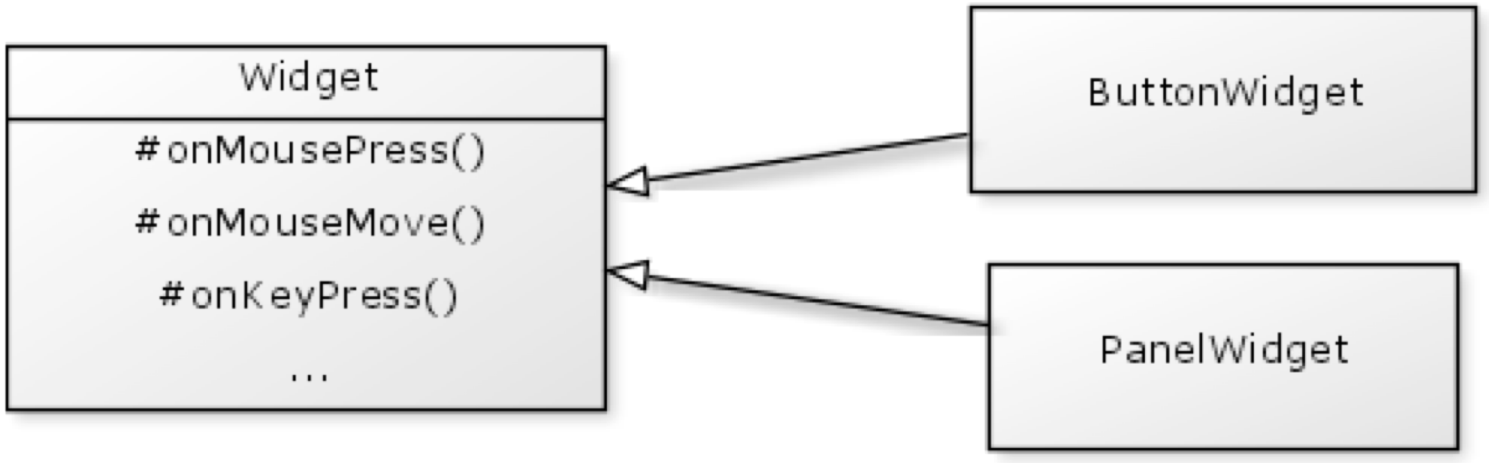
- Each window registers a WindowProc function (Window Procedure) which is called each time an event is dispatched
- The WindowProc uses a switch statement to identify each event that it needs to handle.
  - There are over 100 standard events…

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg,
                            WPARAM wParam, LPARAM lParam) {
   switch (uMsg) {
      case WM_CLOSE:
         DestroyWindow (hWnd);
         break;
      case WM_SIZE:
         …
      case WM_KEYDOWN:
         …
```

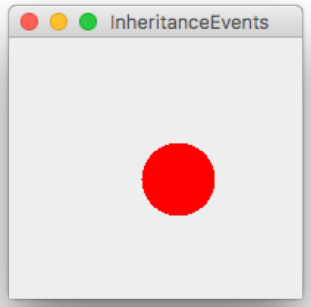
Inheritance Binding

- Event is dispatched to an Object-Oriented (OO) widget
  - OO widget inherits from a base widget class with all event handling methods defined a priori

    onMousePress, onMouseMove, onKeyPress, etc
  - The widget overrides methods for events it wishes to handle
  - Each method handles multiple related events
- Used in Java 1.0



InheritanceEvents.java




```
public class InheritanceEvents extends JPanel  {
   public static void main(String[] args) {
      InheritanceEvents p = new InheritanceEvents();
      ...
      // enable events for this JPanel
      p.enableEvents(MouseEvent.MOUSE_MOTION_EVENT_MASK);
   }

   protected void processMouseMotionEvent(MouseEvent e) {
      // only detects button state WHILE moving!
      if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
         x = e.getX();
         y = e.getY();
         repaint();
      }
   }
```

Inheritance Problems

- Each widget handles its own events, or the widget container has to check what widget the event is meant for
- Multiple event types are processed through each event method
  - complex and error-prone, just a switch statement again
- No filtering of events: performance issues
  - consider frequent events like mouse-move
- It doesn't scale well: How to add new events?
  - e.g. penButtonPress, touchGesture, ….
- Muddies separation between GUI and application logic: event handling application code is in the inherited widget
  - Use inheritance for extending functionality, not binding events

Event Interface Binding (Java)

- Define an Interface for event handling
  - Describes method signatures meant for handling events
- Create a class that implements that interface to handle those events
- Simplest case: ActionListeners in Java

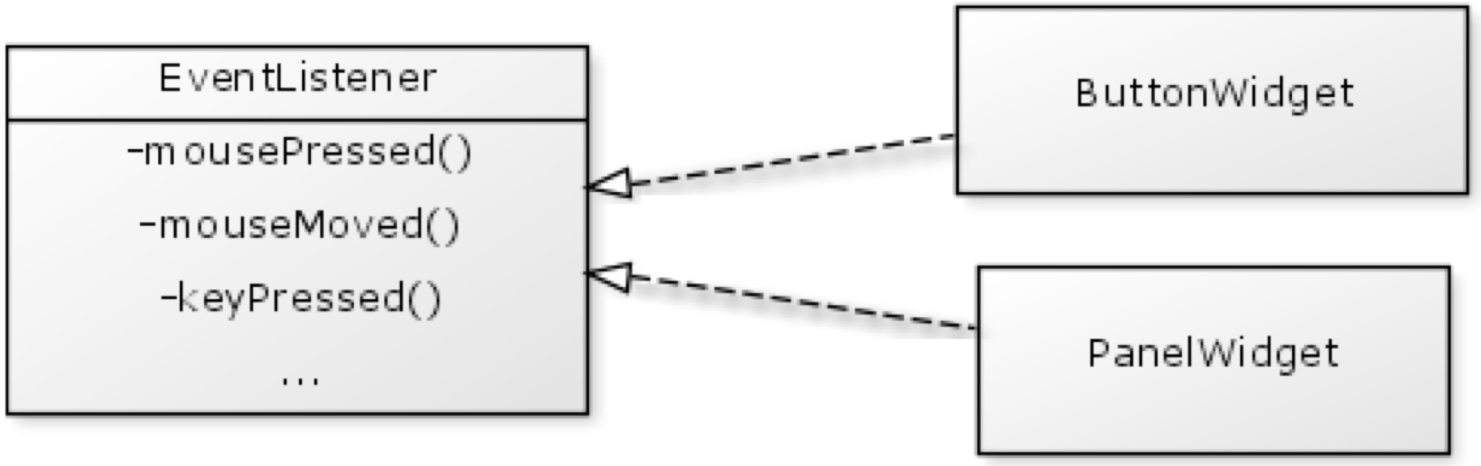
```
public class Beeper extends JPanel
                    implements ActionListener {      // Interface

    public void actionPerformed(ActionEvent e) {     // Method
        Toolkit.getDefaultToolkit().beep();
    }

}
```

Listener Interface Binding (Java)

- Define listener interfaces for *specific event types (or device types)*
  - e.g. MouseListener, MouseMotionListener, KeyListener, WindowListener, …
- Widget object implements event "listener" Interfaces
- When event is dispatched, relevant listener method is called for that widget
  - mousePressed, mouseMoved, …



Java Listener Model

- Java has interfaces specialized by event type.
  - Each interface lists the methods that are needed to support that device's events
  - To use them, write a class that implements this interface, and override the methods for events you care about.
- Because it's an interface, you have to override all of these methods – even for events you don't care about!

```
interface MouseInputListener {
    public void mouseClicked(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mouseDragged(MouseEvent e);
    public void mouseMoved(MouseEvent e)
}
```

Using Listeners

```
// create a custom listener class for this component
static class MyMouseListener implements MouseInputListener  {
    public void mouseClicked(MouseEvent e) {
        System.exit(1);
    }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
    public void mouseDragged(MouseEvent e) { }
    public void mouseMoved(MouseEvent e) { }
}
```

```
// create a panel and add components
JPanel panel = new JPanel();
JButton button = new JButton("Ok");
button.addMouseListener(new MyMouseListener());
panel.add(button);
```

BasicForm2.java

What Listeners Exist?

What's difficult with this approach?

Event Type	Generated By	Listener Interface	Methods
ActionEvent	Button press, Menu select, Key press, Window resize, …	ActionListener	actionPerformed(ActionEvent e)
WindowEvent	Open, close, minimize, maximize, resize	WindowListener	windowOpened(WindowEvent e) windowClosed(WindowEvent e) windowClosing(WindowEvent e) windowActivated(WindowEvent e) windowDeActivated(WindowEvent e) windowIconified(WindowEvent e) windowDeiconified(WindowEvent e)
KeyEvent	Press or release a key	KeyListener	keyPressed(KeyEvent e) keyReleased(KeyEvent e) keyTyped(KeyEvent e)
MouseEvent	Press/release/click mouse button	MouseListener	mouseClicked(MouseEvent e) mouseEntered(MouseEvent e) mouseExited(MouseEvent e) mousePressed(MouseEvent e) mouseReleased(MouseEvent e)
MouseEvent	Move the mouse	MouseMotionListener	mouseDragged(MouseEvent e) mouseMoved(MouseEvent e)

Listener Adapter Binding

- Many listener interfaces have only a single method
  - e.g. ActionListener has only actionPerformed
- Other listener interfaces have several methods
  - e.g. WindowListener has 7 methods, including windowActivated, windowClosed, windowClosing, …
- Typically interested in only a few of these methods. Leads to lots of "boilerplate" code with "no-op" methods, e.g.
  - void windowClosed(WindowEvent e) { }
- Each listener with multiple methods has an **Adapter class** with no-op methods. Simply extend the adapter, overriding only the methods of interest.

Adapters vs. Listeners

- Java also has adapters, which are base classes with empty listeners.
  - Extend the adapter and override the event handlers that you care about; avoids bloat.

```
// create a custom adapter from MouseAdapter base class
static class MyMouseAdapter extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        System.exit(1);
    }
}
```

BasicForm3.java

```
// create a panel and add components
JPanel panel = new JPanel();
JButton button = new JButton("Ok");
button.addMouseListener(new MyMouseAdapter());
panel.add(button);
```

What's wrong with this approach?

Anonymous Inner Classes

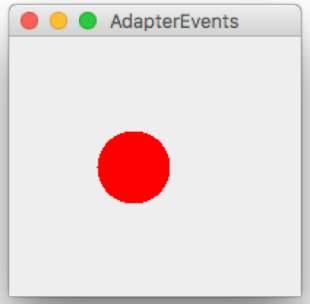
- We really, really don't want to create custom adapters for every component.
  - Solution? Anonymous inner class.

BasicForm4.java

```
public static void main(String[] args) {
    // create a window
    JFrame frame = new JFrame("Window Demo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // create a panel and add components
    JPanel panel = new JPanel();
    JButton button = new JButton("Ok");
    button.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
        System.exit(1);
    }});
    panel.add(button);
```

AdapterEvents.java




```
public class AdapterEvents extends JPanel  {
   public static void main(String[] args) {
      AdapterEvents panel = new AdapterEvents();
      ...
   }

   AdapterEvents() {
      this.addMouseMotionListener(new MyListener());
   }

   class MyListener extends MouseMotionAdapter {
      public void mouseDragged(MouseEvent e) {
         x = e.getX();
         y = e.getY();
         repaint();
      }
   }
```

Delegate Binding (.NET)

- Interface architecture can be a bit heavyweight
- Can instead have something closer to a simple function callback (a function called when a specific event occurs)
- Delegates in Microsoft's .NET are like a C/C++ function pointer for methods, but they:
  - Are object oriented
  - Are completely type checked
  - Are more secure
  - Support multicasting (able to "point" to more than one method)
- Using delegates is a way to broadcast and subscribe to events
- .NET has special delegates called "events"

Using Delegates

1. Declare a delegate using a method signature

```
public delegate void Del(string message);
```

2. Declare a delegate object

```
Del handler;
```

3. Instantiate the delegate with a method

```
// method to delegate (in MyClass)
public static void MyMethod(string message) {
            System.Console.WriteLine(message); }
handler = myClassObject.MyMethod;
```

4. Invoke the delegate

```
handler("Hello World");
```

Multicasting

- Instantiate more than one method for a delegate object

```
handler = MyMethod1 + MyMethod2;
handler += MyMethod3;
```

- Invoke the delegate, calling all the methods

```
handler("Hello World");
```

- Remove method from a delegate object

```
handler -= MyMethod1;
```

- What about this?

```
handler = MyMethod4;
```

Events in .NET

- Events are a delegate with restricted access
- Declare an event object instead of a delegate object:

```
public delegate void Del(Object o, EventArgs e);
event Del handler;
```

- "event" keyword allows enclosing class to use delegate as normal, but outside code can only use the -= and += features of the delegate
- Gives enclosing class exclusive control over the delegate
- Outside code can't wipe out delegate list, can't do this:

```
handler = MyMethod4;
```

- Can have anonymous delegate events (similar to Java style):

```
b.Click += delegate(Object o, EventArgs e) {
        Windows.Forms.MessageBox.Show("Click!");  };
```

Global Event Queue “Hooks”

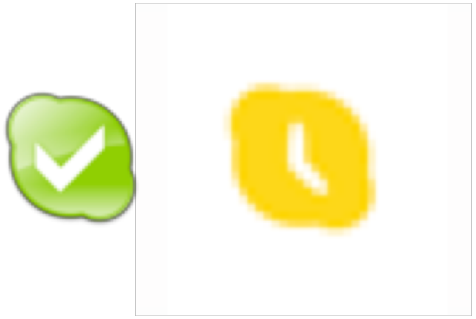
In specific situations, we can monitor events outside of an application

- An application can monitor BWS events **across all applications**
- Can also inject events **to another application**
- This can be a very useful technique
  - examples?
- This can be a security issue
  - examples?

- Take a look at jnativehook
  - library to provide global keyboard and mouse listeners for Java.
  - https://github.com/kwhat/jnativehook/

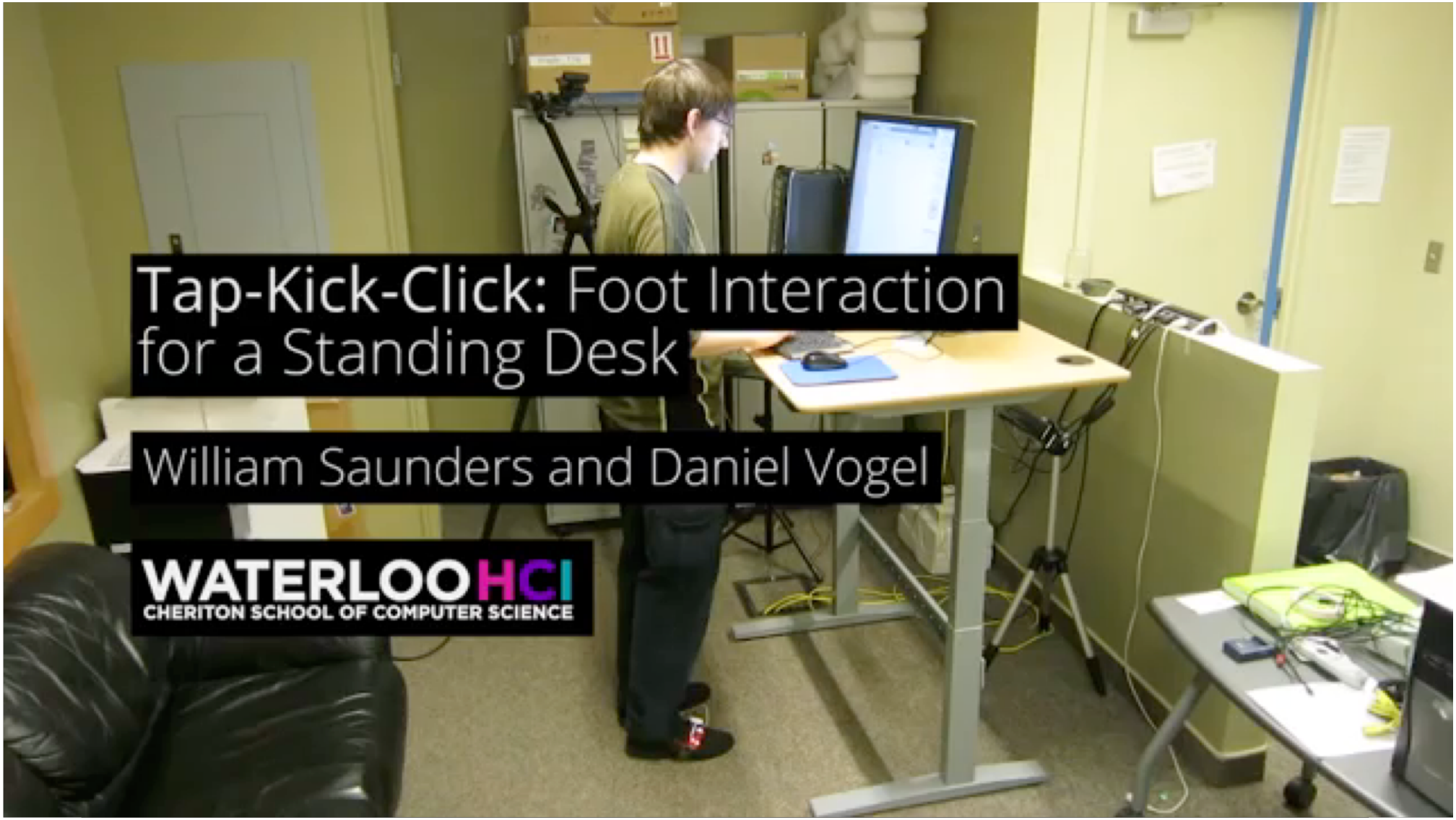
Global Hooks for Awareness

- Some application monitor level of "activity" using global hooks
- When activity drops, can do something
  - IM client: set state to "away"
  - Screensaver: start screensaver



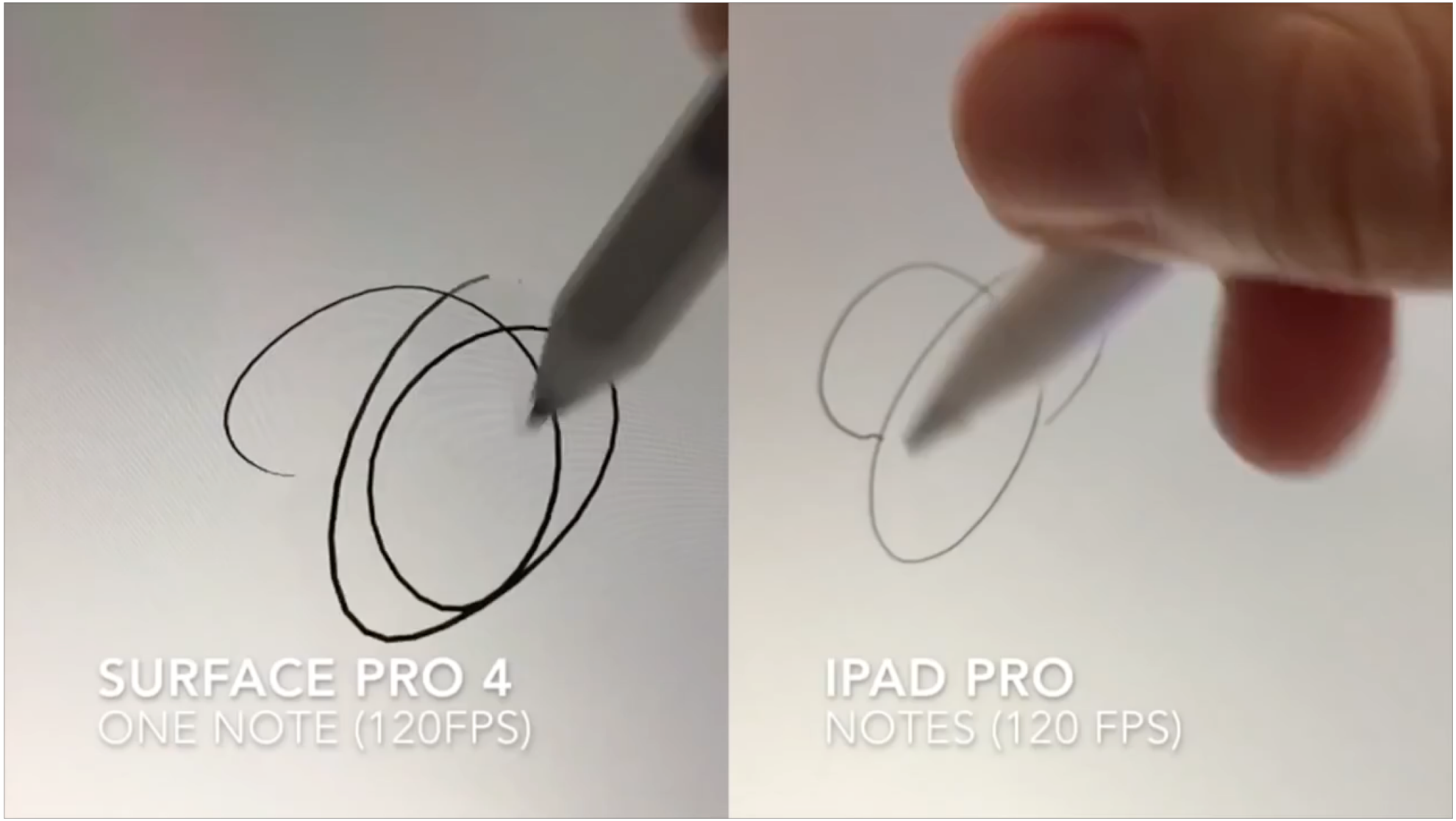
Events for High Frequency Input

- Pen and touch generate many high frequency events
  - pen motion input can be 120Hz or higher
  - pen sensor is much higher resolution than display
  - multi-touch generates simultaneous events for multiple fingers
- **Problem:** These events often too fast for application to handle
- **Solution:** Not all events delivered individually:
  - e.g. all penDown and penUp,
    but may skip some penMove events
  - Event object includes array of “skipped” penMove positions
  - (Android does this for touch input)



Global Hooks in Tap-Kick-Click research prototype

- https://youtu.be/pqycjWHoI2w



Surface Pro 4 vs iPad Pro pencil tracking

- https://www.youtube.com/watch?v=pK41eAYNLu4