Keyboard and mouse input

- What and why?
  - Mapping keys and mouse events to game actions
  - Want the user to be able to configure their input control
  - Essential part of any game
- Book reference: chapter 3
  - Chapter 3 also includes a section about creating user interfaces, both design tips and Java

Java input

- Our game code is all built on AWT
  - Java's core standard graphics library with Frames etc.
- When an input event occurs, AWT notifies all listeners of that event
  - Essentially: "Yo, someone just pressed the space bar"
  - The notification is done by the AWT event dispatch thread; so keep synchronization in mind
    - e.g. don't change the game state in the middle of a draw
- Any object can be a listener of certain events by implementing the corresponding listener interfaces
  - Implement a function which is called every time the event occurs

Keyboard input

- If you have programmed Java before, you might be used to pop-up windows which take text
- We will do lower level stuff: handling the actual keys being pressed
- To capture key events you need to:
  - Implement a KeyListener
  - Register the listener to listen for events on a certain object
    - e.g. register with the game Window

KeyListener

- A.k.a. interface java.awt.event.KeyListener
- You might want to check out Sun's tutorial on KeyListener
  - It is linked to from Sun's Java doc of KeyListener (http://java.sun.com/j2se/1.4.2/docs/api/)
- To implement a KeyListener, you need to implement three methods:
  - keyPressed(e)
  - keyReleased(e)
  - keyTyped(e)
  - where e is a KeyEvent

KeyListener

- *keyTyped* is a higher level event (than the other two), which is called when a Unicode character is called
- *keyPressed* and *keyReleased* are simply called whenever a key is pressed and released respectively
  - See the documentation for exact definitions
- A *KeyEvent* contains information about which key was pressed, represented as a virtual key code
  - The virtual key codes are defined in KeyEvent
  - Note that the info is about which key was pressed; not the character
    - E.g. characters q and Q have the same virtual key code

KeyListener

- Let's look at KeyTest.java (p. 96)
  - It just prints when keys are pressed and released, and the name of them
- Catching a pressed key (in pseudo-Java):

```
void keyPressed(KeyEvent e) {
    int keycode = e.getKeyCode();
    if (keyCode == KeyEvent.VK_ESCAPE) stop();
    else
        addMessage("Pressed: " + KeyEvent.getKeyText(keyCode));
}
```

Mouse input

- The mouse can do these things:
  - Mouse button clicks
  - Mouse motion
  - Mouse wheel scrolls (possibly)
- Each event has its own listener; in the same order:
  - MouseListener, MouseMotionListener, and MouseWheelListener
  - Each take a MouseEvent as parameter

Mouse listeners

- The *MouseListener* interface has methods for
  - mouse presses, releases and clicks
    - clicks are higher level combinations of presses and releases
  - the pressed button is available via getButton()
- The *MouseMotionListener* can detect regular motion and drag motion
  - a drag motion is motion with a button pressed
  - the current position of the mouse is available via getX() and getY()
- The *MouseWheelListener* can detect wheel scrolls
  - getWheelRotation() gives the 'size' and direction of the scroll

A MouseListener

- A test program: MouseTest.java (p. 102)
  - Shows "Hello world" as a trail after the mouse pointer
- Catch mouse movements and store visited points:

```
void mouseMoved(MouseEvent e) {
    Point p = new Point(e.getX(), e.getY());
    trailList.addFirst(p);
    while (trailList.size() > TRAIL_SIZE)
        trailList.removeLast();
}
```

A MouseListener

- Draw the trail:
  - In the *draw* method, do:

```
for (int i = 0; i < trailList.size(); i++) {
    Point p = trailList.get(i);
    g.drawString("Hello World!", p.x, p.y);
}
```

- That's it :)

Game input

- Structured input handler:
  - Handles all key and mouse events
  - Saves the events so you can process them when you want to, instead of when the event dispatch thread wants to
  - Detects the initial press for some keys and whether the key is held down for others
    - e.g., you typically want to be able to hold down a key to keep moving, not having to tap it – but maybe opposite for jumping
  - Maps keys to game actions
  - Can change the key mapping in run-time
    - So the user can reconfigure controls

Game input

- Let us look at an implementation of an input manager
- *GameAction.java* is used to keep track of input events relevant to a game action
  - such as whether it was triggered at all; "was the jump key pressed?"
  - GameActions can be mapped to virtual key codes, to enable dynamic reconfiguration of keys

Game input

- How do we map keys to game actions? And how do we do it dynamically?
- Well, we can just store a map of virtual key codes to game actions, and update this when the user wants to reconfigure control
  - Create GameAction objects, and index them by key codes

Game input

- InputManager.java (p. 118)
  - has code for mapping game actions to key codes and mouse events
- Uses an array for the mapping:
  - GameAction[] keyActions = new GameAction[K];

```
void mapToKey(GameAction gameAction, int keyCode) {
    keyActions[keyCode] = gameAction;
}
```

- Test program: InputManagerTest.java (p. 134)
  - Tests the input manager with a jumping figure

Summary

- Keyboard and mouse input
  - Handle key and mouse events
- Mapping keys to game actions
  - Dynamically change it
    - User interface (a menu for binding keys)
      - See chapter 3 for a refresher on graphical user interfaces
- Many good examples in the book
  - Explore...

Wednesday 22 June

- Guest lecture followed by project lecture!
  - 13.15 in room 1211: guest lecture by Starbreeze
  - Followed by project lecture by Jim