Checking Input

Keyboard

- `is_key_down(key)`
  - Returns True if key is down
  - `key` is a string (`'a'` or `'space'`)
  - Empty string means *any* key
- `is_key_pressed(key)`
  - Returns True if `key` pressed
  - `key` **not** down prev. frame
- `is_key_released(key)`
  - Returns True if `key` released
  - `key` was down prev. frame

Mouse/Touch

- `touch`
  - **Attribute** giving a position
  - Stored as a Point2 object
  - But None if no touch
- `is_touch_pressed()`
  - True if `touch` pressed
  - `touch` was None prev. frame
- `is_touch_released()`
  - True if `touch` released
  - `touch` **not** None prev. frame

Programming Animation

Intra-Frame

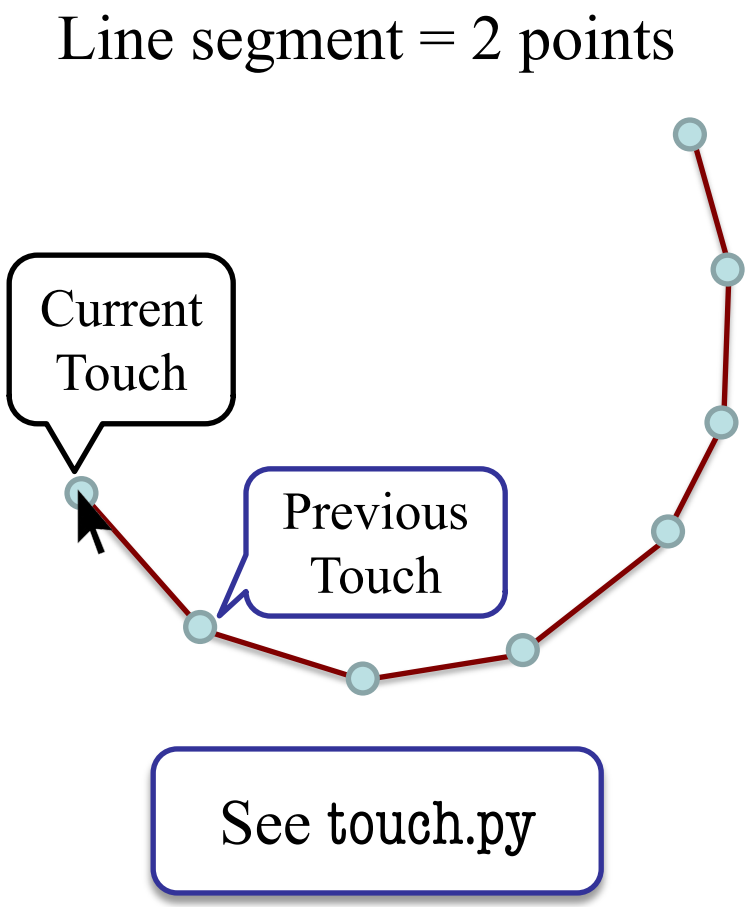
- Computation within frame
  - Only need current frame
- **Example:** Collisions
  - Need current position
  - Use to check for overlap
- Can use **local variables**
  - All lost at `update()` end
  - But no longer need them

Inter-Frame

- Computation across frames
  - Use values from last frame
- **Example:** Movement
  - Need old position/velocity
  - Compute next position
- Requires **attributes**
  - Attributes never deleted
  - Remain after `update()` ends

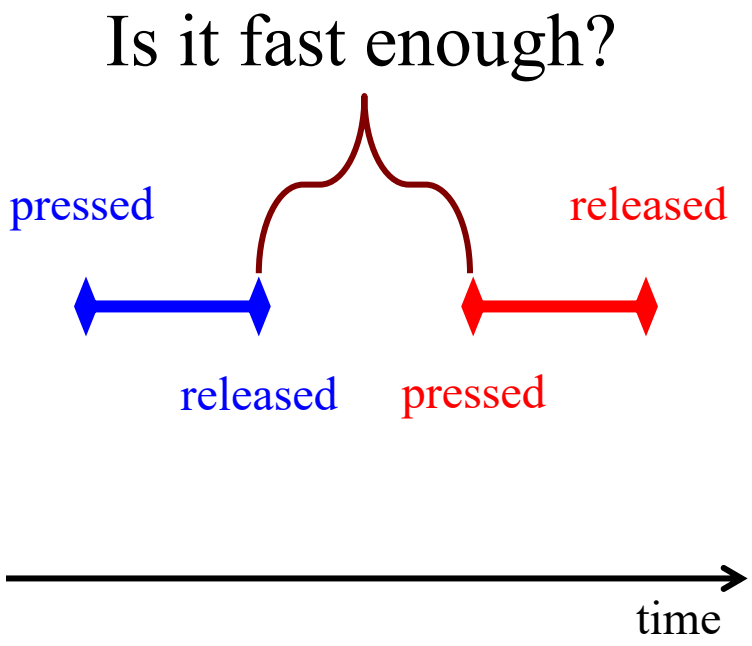
Interframe Computation: Touch

- Works like an Etch-a-Sketch
  - User draws by touching
  - Checks position each frame
  - Draws lines between touches
- Uses attribute `touch` in `GInput`
  - The mouse press position
  - Or **None** if not pressed
  - Access with `self.input.touch`
- But we also need last touch!
  - Forgot if we do not store it
  - Purpose of attribute `last`



Complex Input: Click Types

- Double click = 2 fast clicks
- Count number of fast clicks
  - Add an attribute `clicks`
  - Reset to 0 if not fast enough
- Time click speed
  - Add an attribute `time`
  - Set to 0 when mouse released
  - Increment when not pressed (e.g. in loop method `update()`)
  - Check `time` when next pressed



See `touch.py`

Approaching Complex Input

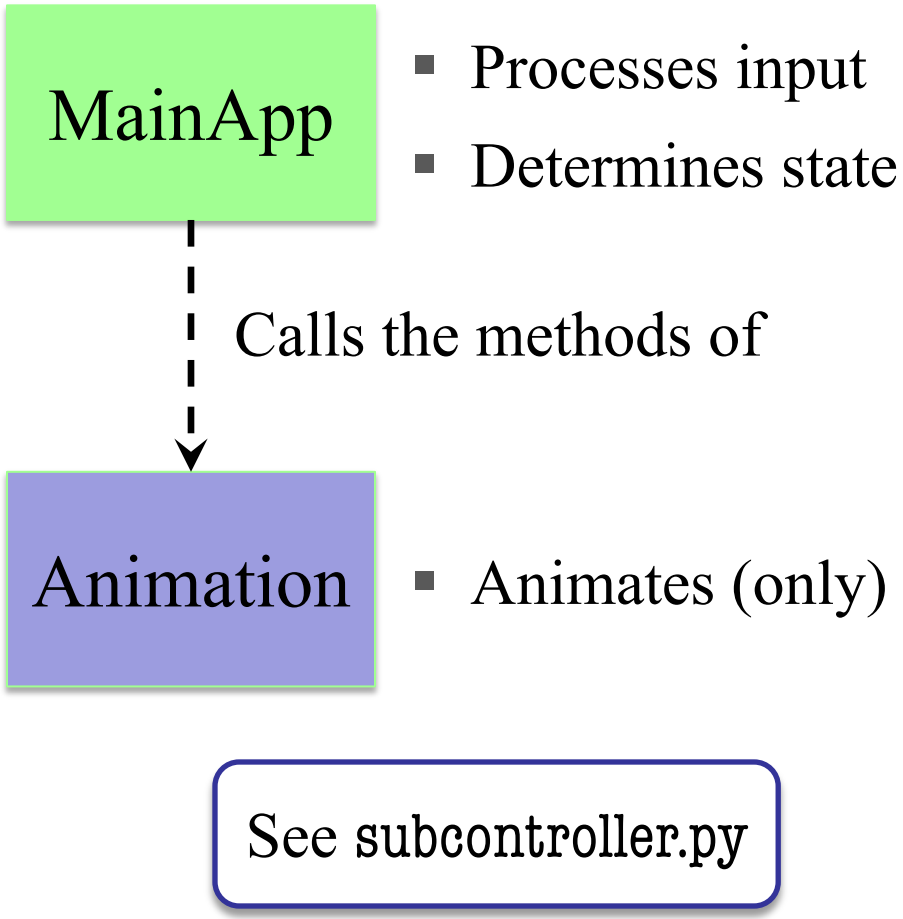
- Look at what you need…
  - From the current animation frame
  - From the previous animation frame
  - From many frames in the past

Requires new attributes

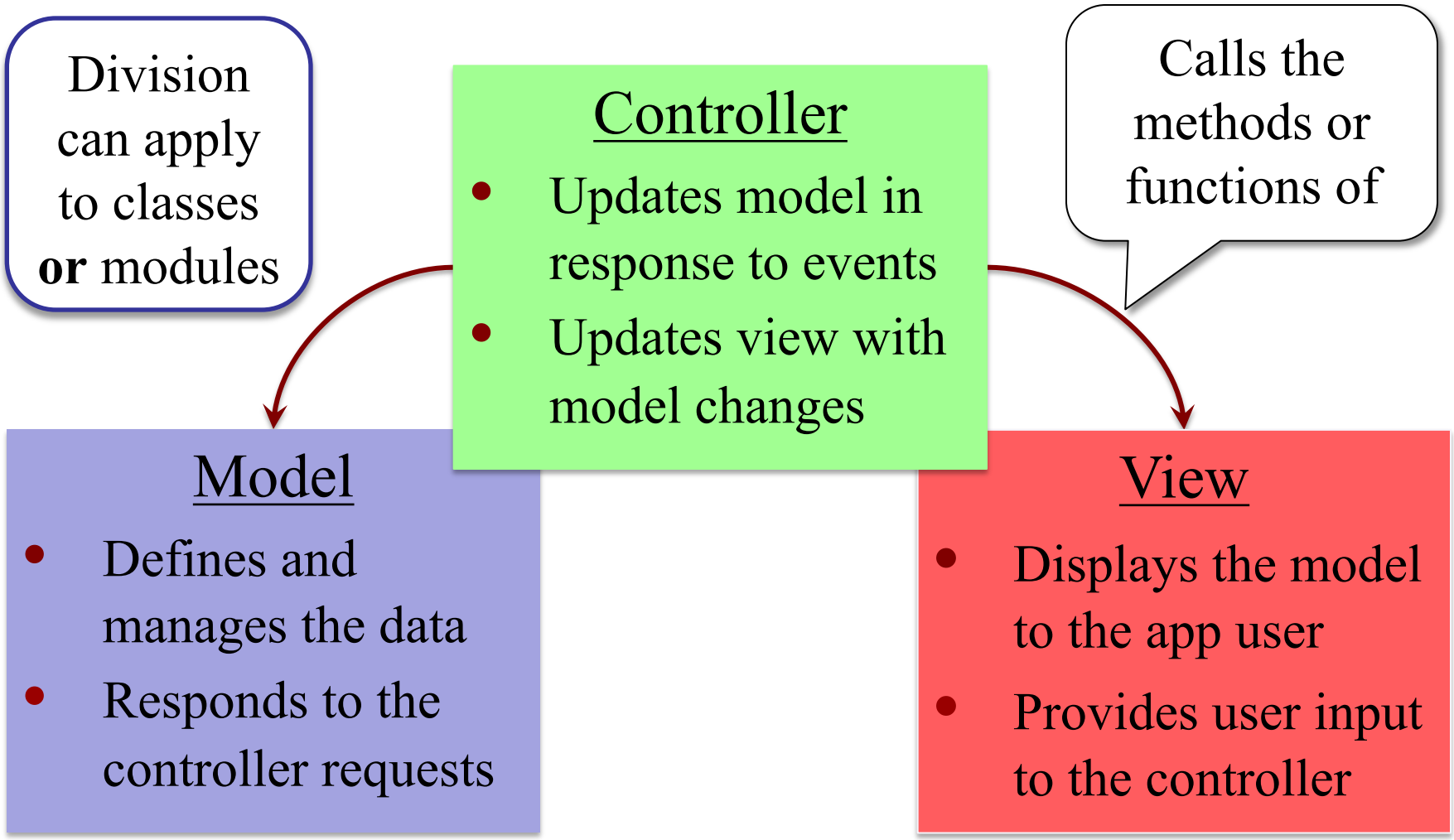
- **Previous frame** is the easiest
  - Set them at the end of `update()` method
  - Just like updating a loop variable!
- **Longer term** depends on the problem
  - See today's **optional lab**

Designing Complex Applications

- Applications can become extremely complex
  - Large classes doing a lot
  - Many states & invariants
  - Specification unreadable
- **Idea**: Break application up into several classes
  - Start with a "main" class
  - Other classes have roles
  - Main class delegates work

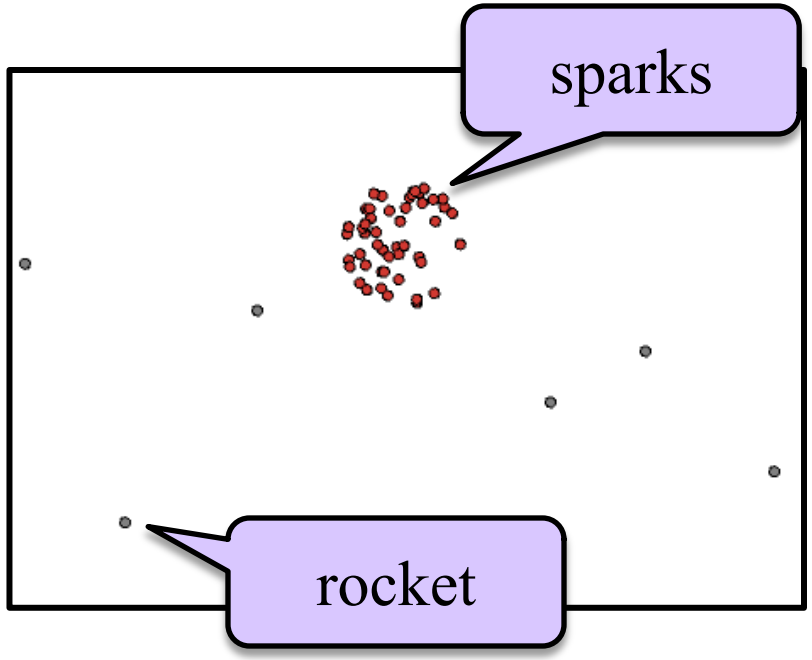


Model-View-Controller Pattern



Models in Assignment 7

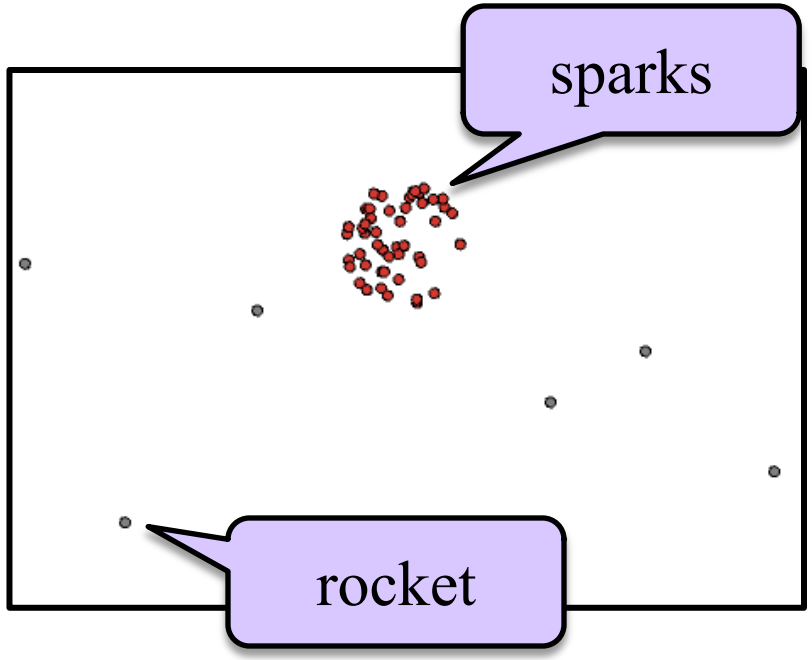
- Often subclass of `GObject`
  - Has built-in draw method
- Includes groups of models
  - **Example**: rockets in `pyro.py`
  - Each rocket is a model
  - But so is the entire list!
  - `update()` will change both
- **A7**: Several model classes
  - `Ship` to animate the player
  - `Alien` to represent an alien



See `pyro.py`

Models in Assignment 7

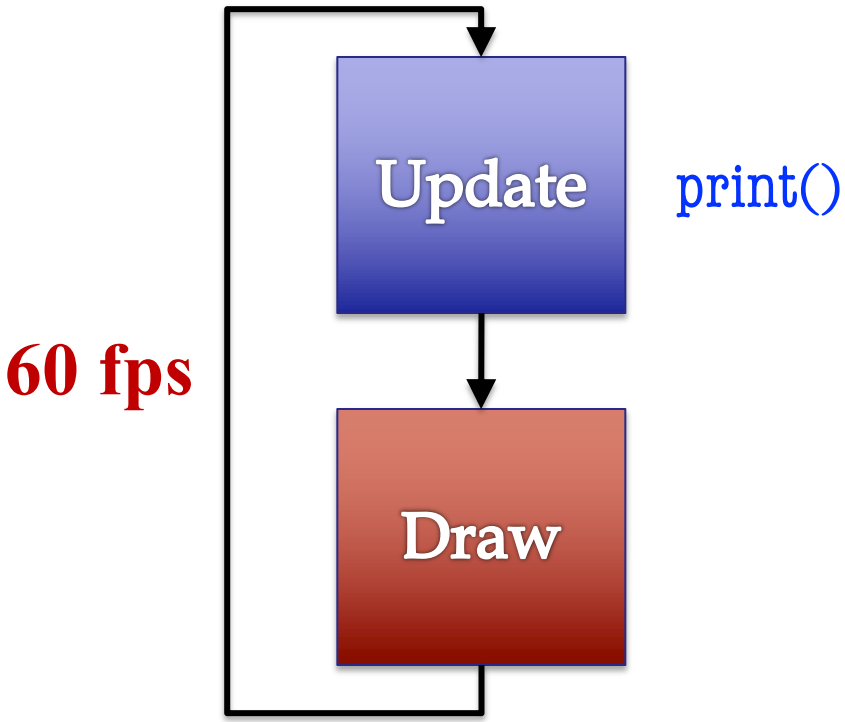
- Often subclass of `GObject`
  - Has built-in draw method
- Includes groups of models
  - **Example**: rockets in `pyro.py`
  - Each rocket is a model
  - But so is the entire list!
  - `update()` will change both
- **A7**: Several model classes
  - `Ship` to animate the player
  - `Alien` to represent an alien



See `pyro.py`

The Debugging Problem

- What if there is a bug?
  - Graphic does not display
  - Input does not work right
- Have to **find** the bug
  - Isolate bad line of code
  - Requires visualization
  - Typically use print()
- But this will **print a lot**!
  - Overwhelmed by output
  - Hard to find anything



Managing Your Diagnostics

- The print() approach has two main problems
  - Prints all the time, even if no problem
  - Lots of prints means it scrolls by very fast
- To control the first problem, **toggle the print**
  - Create **special input** for debugging
  - Use that input to turn on/off as needed
- To control the second, **print to a file**
  - A file can be searched
  - A file can be compared against runs

Logger: A Class to Support Logging

```
import debug
tool = debug.Logger()
tool.setFile('log.txt')

tool.log('This is always printed')
tool.debug('This is not printed')

tool.setVerbose(True)
tool.log('This is still printed')
tool.debug('This is printed this time')
```

Can be None for screen only

Only prints if verbose is True

See firelog.py