Lecture 23

State-Based Applications I

Announcements for This Lecture

Assignments

- A5 is now graded
  - **Mean**: 47.2, **Median**: 49
  - **A**: 46 (80%), **B**: 38 (15%)
- A6 is due **Sunday**
  - Worth 8% of your grade
  - Remember to fill in Survey
- A7 posted **Wednesday**
  - Due **December 7th** (last day)
  - Minor extensions possible

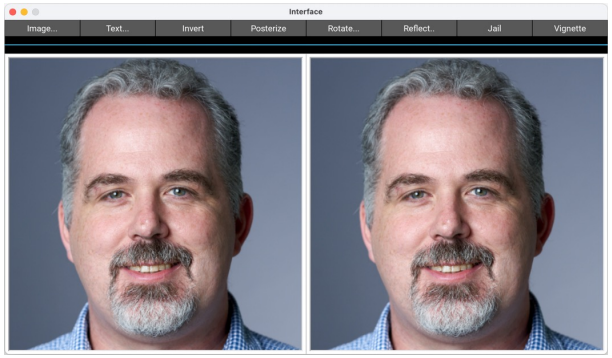
Video Lessons

- **Lesson 25** for today
- **Lessons 26, 27** for Tues

- **Prelim, 11/17 at 7:30 pm**
  - Material up to **11/5**
  - Study guide is posted
  - Review posted
- **Conflicts are now closed**
  - Only if an emergency

Long Running Applications

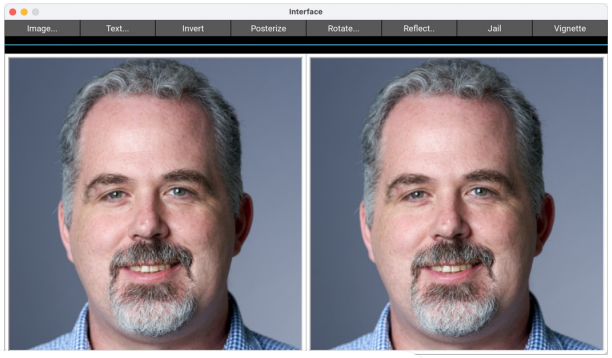
- Think about an application like Assignment 6
  - Displays a window
  - Responds to your input
  - Continues until you quit
- How long until you quit?
  - This is essentially a **while loop**
  - Driven by a **goal condition** or a **bool variable**




```
while program_is_running:
    do something
```

Long Running Applications

- Think about an application like Assignment 6
  - Displays a window
  - Responds to your input
  - Continues until you quit
- How long until you quit?
  - This is essentially a **while loop**
  - Driven by a **goal condition** or a



Effectively while True:

```
while program_is_running:
    do something
```

While-Loops are Fundamental

- They are behind all **long-running applications**
  - Always have a top level while loop
  - And many subtasks are themselves while loops
- The challenge is how these loops fit together
  - Very often you are not writing all of the code
  - Your code has to fit inside "invisible" loops
  - Your code can cause an infinite loop you never saw
- To understand this, we will look at a **case study**

The Game of Pig: A Random Game

- Play progresses clockwise
- On your turn, throw the die:
  - If roll 1: lose turn, score zero
  - Anything else: add it to score
    - Can also roll again (and lose)
    - If stop, score is "banked"
- First person to 100 wins



The Game of Pig: A Random Game

- Play progresses clockwise
- On your turn, throw the die:
  - If roll 1: lose turn, s
  - An
    - Can also roll again (and lose)
    - If stop, score is "banked"
- First person to 100 wins



Easy to write with while-loops

Designing an AI for Opponent is Easy

# Throws	Survial	Expected Gain	Expected Value
1	83%	3.33	3.33
2	69%	2.78	6.11
3	58%	2.32	8.43
4	48%	1.92	10.35
5	40%	1.61	11.96
6	33%	1.34	13.30
7	28%	1.12	14.42
8	23%	.93	15.35
9	19%	.77	16.12
10	16%	.65	16.77
…	…	…	…
50	0.01%	0.0004	19.998

Designing an AI for Opponent is Easy

# Throws	Survial	Expected Gain	Expected Value
1	83%	3.33	3.33
2	69%	2.78	6.11
3	58%	2.32	8.43
4	48%	1.92	10.35
5	40%	1.61	11.96
6	33%	1.34	13.30
7	28%	1.12	14.42
8	23%	.93	15.35
9	19%		16.12
10	16%		16.77
…	…	…	…
50	0.01%	0.0004	19.998

Strategy:
Bank at 20

The Primary Function

```
def play(target):
    """Plays a single game of Pig to target score.

    Precondition: target is an int > 0"""
    # Initialize the scores
    # while no one has reached the target
        # Play a round for the player
        # If the player did not reach the target
            # Play a round for the opponent
    # Display the results
```

The Player Round

```
def player_turn():
    """ Runs a single turn for the player."""
    # while the player has not stopped
        # Roll the die
        # If is a 1
            # Set score to 0 and stop the turn
        # else
            # Add the to the score
            # Ask the player whether to continue
    # Return the score
```

Prompt helper

The Opponent Round

```
def roll_past(goal):
    """Returns: The score from rolling a die until passing goal."""
    loop = True  # Keep looping until this is false
    score = 0
    while loop:
        roll = random.randint(1,6)
        if roll == 1:
            score = 0; loop = False
        else:
            score = score + roll; loop = score < goal
    return score
```

Look familiar?

Can an Application Be a Class?

- We think of programming as parts of speech
  - **Functions** are **verbs**
  - **Classes** are **nouns**
- What is an application?
  - It does things. Does that make it a verb?
  - It is something you run. Is it a noun?
- There is no correct answer
  - Applications can be functions or classes

Advantage of Class-Based Applications

- **Function calls** correspond to **call frames**
  - Any work is lost when the game is complete
  - What if you want to keep statistics?
- **Objects** live in the **heap**
  - Any attributes will continue to persist
  - Still have functions (methods) to do stuff
- **Idea**: **Refactor** our function-based apps
  - Turn some local variables into attributes
  - But not all – only those we want to "save"

The Refactored Pig Game

```
class Pig(object):
    """An application to play the game pig."""
    # IMMUTABLE ATTRIBUTES (GETTERS, BUT NO SETTERS)
    # Attribute _target: The goal target for each player
    # Invariant: _target is an int > 0
    #
    # Attribute _bank: The bank strategy for the AI player
    # Invariant: bank is an int > 0 and <= _target
    #
    # Attribute _playerScore: The current score for the player
    # Invariant: _playerScore is an int >= 0
    #
    # Attribute _aibotScore: The current score for the opponent
    # Invariant: _aibotScore is an int >= 0
```

Classes are Popular in GUIs

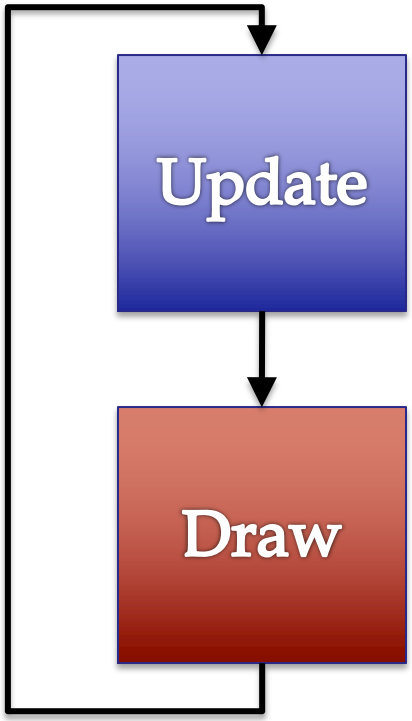
- **G**raphical **U**ser **I**nterface
  - These are normal apps
  - In contrast to text-based (e.g. runs in Terminal)
- These are very complex
  - No one writes from scratch
  - We rely on outside modules
- Often a library of **classes**
  - Visual elements are nouns
  - Subclass them to customize

```
class MyApp(kivy.app.App):
    """
    A window with 'Hello World!' inside
    """
    # HIDDEN INSTANCE ATTRIBUTES
    # Attr _label: text display in window
    # Invar: _label is a Label object

    # Override this method...
    def build(self):
        ...
```

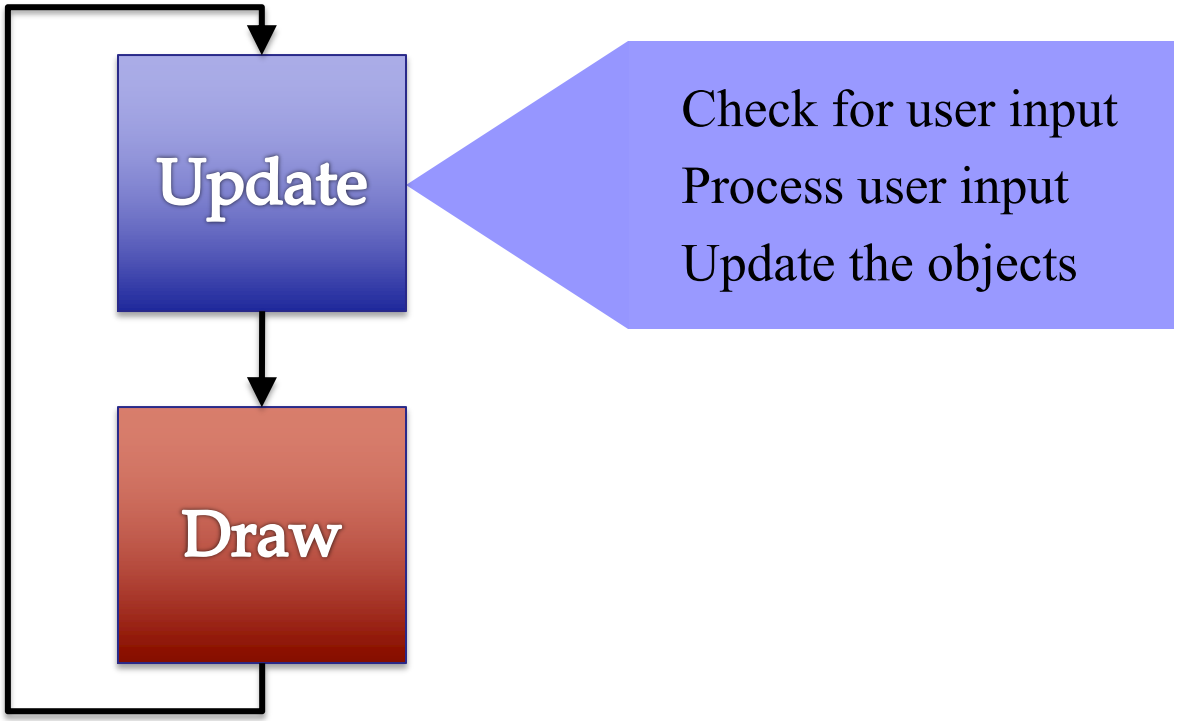
Understanding a GUI Application

Animates the application, like a movie



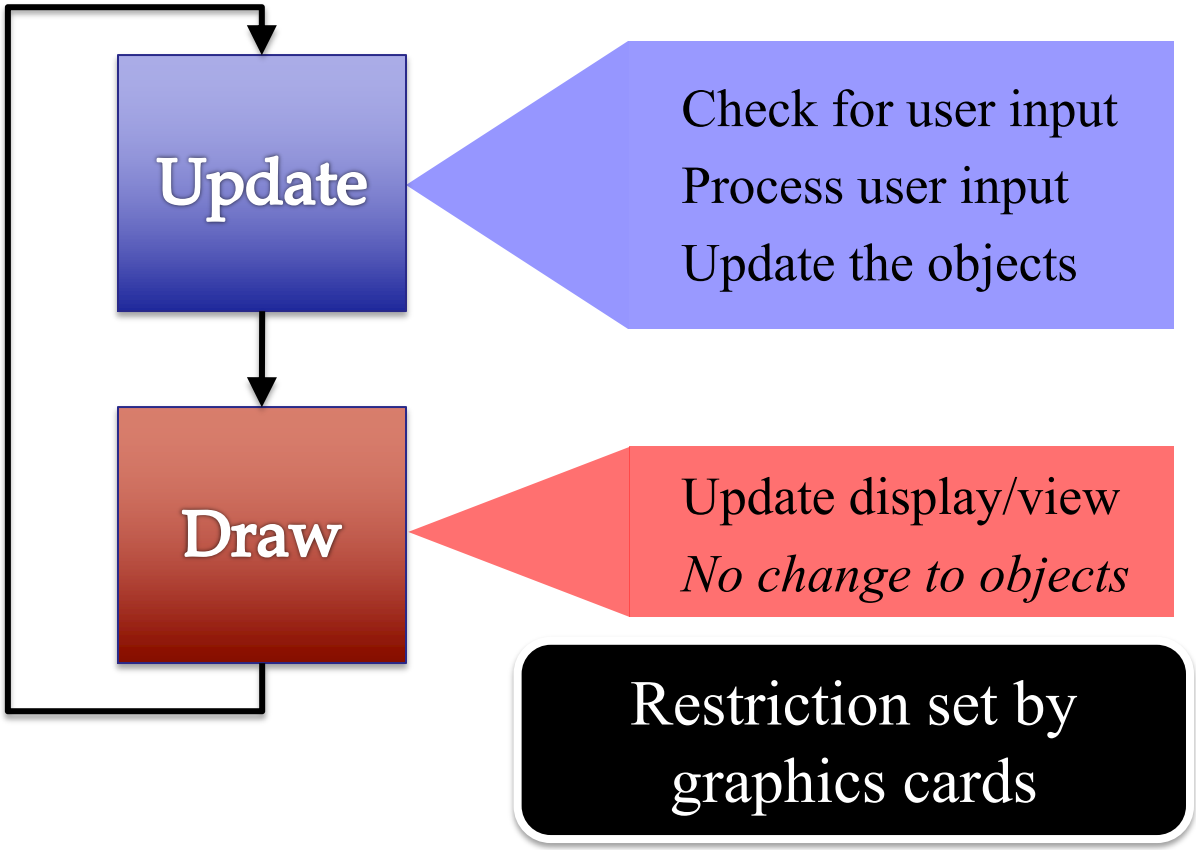
Understanding a GUI Application

Animates the application, like a movie

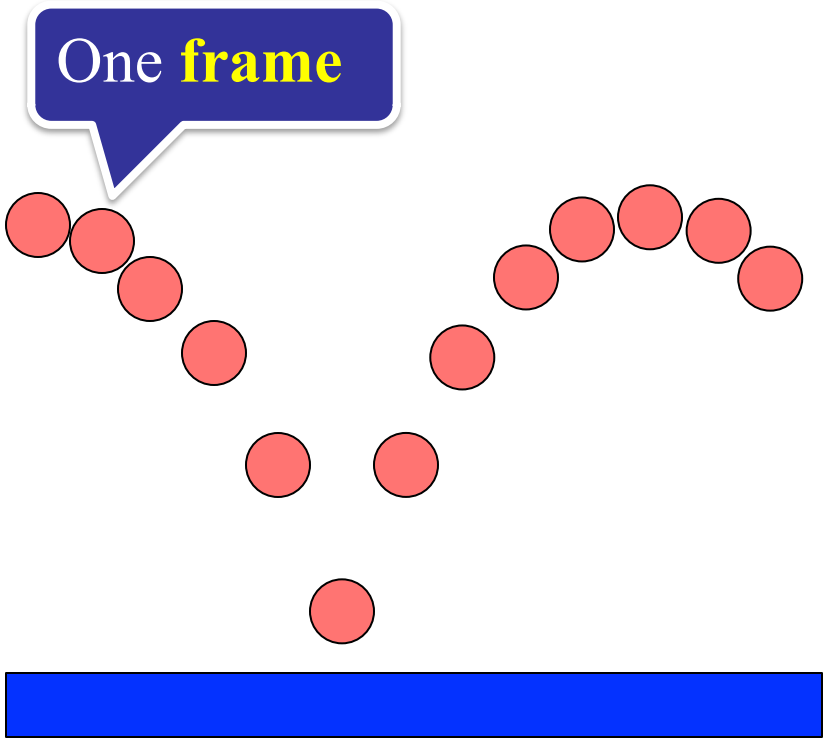


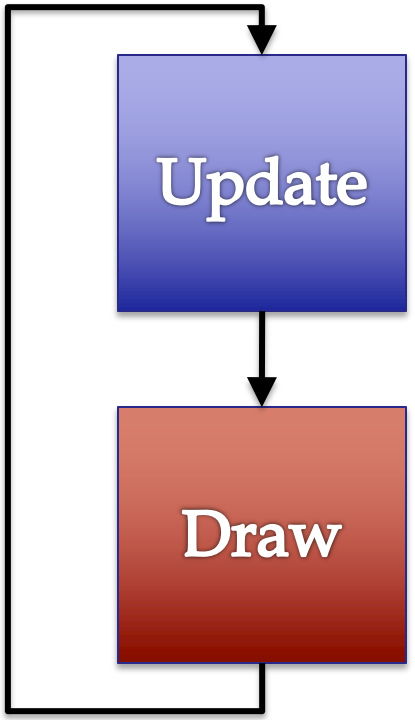
Understanding a GUI Application

Animates the application, like a movie



The Animation Frame





How A Frame is Animated

```
while program_is_running:
    # Get information from mouse/keyboard
    # (Handled by OS/GUI libraries)

    # Your code goes here



    # Draw stuff on the screen
    # (Handled by OS/GUI libraries)
```

How A Frame is Animated

```
while program_is_running:
    # Get information from mouse/keyboard
    # (Handled by OS/GUI libraries)

    # Your code goes here



    # Draw stuff on the screen
    # (Handled by OS/GUI libraries)
```

Would like to "plug in" code

Why do we need to write this each time?

How A Frame is Animated

```
while program_is_running:
    # Get information from mou
    # (Handled by OS/GUI libraries)

    # Your code goes here

    application.update()

    # Draw stuff on the screen
    # (Handled by OS/GUI libraries)
```

The while-loop is **external**

This is the loop **body**

- Put loop **BODY** in an app class.
- OS/GUI handles everything else.

We Have to Change Our Thinking

```
while program_is_running:
    # Get information from mouse/keyboard
    # (Handled
    # Your code
    application.update()
                                    aries)
```

Local variables are **erased** each loop step!

If something needs to survive multiple steps it must be an **attribute**

Attributes = Loop Variables

Normal Loops

Variables "external" to the loop body

```
x = 0
i = 2
# x = sum of squares of 2..i-1
while i <= 5:
    x = x + i*i
    i = i + 1
# x = sum of squares of 2..5
```

Application

Attributes are the "external" variables

```
while program_running:
    # Get input
    # Your code called here
    application.update()
    # Draw
```

Variables and the Loop

```
while program_is_running:
    # Get information from mouse/keyboard
    # (Handled by OS/GUI libraries)

    # Your code goes here
    application.update()

    # Draw stuff on the screen
    # (Handled by OS/GUI libraries)
```

Local variables erased.
But **attributes** persist.

The Actual Game Loop

```
# Constructor
game = GameApp(...)
...
game.start() #Loop initialization
while program_running:
    # Get input
    # Your code goes here
    game.update(time_elapsed)
    game.draw()
```

Too *early* to initialize everything

Actual loop initialization

Separate update() and draw() methods

Designing a Game Class: Animation

```
class Animation(game2d.GameApp):
    """App to animate an ellipse in a circle."""

    def start(self):
        """Initializes the game loop."""
        ...

    def update(self,dt):
        """Changes the ellipse position."""
        ...

    def draw(self):
        """Draws the ellipse"""
        ...
```

See animation.py

Designing a Game Class: Animation

```
class Animation(game2d.GameApp):
    """App to animate an ellipse"""

    def start(self):
        """Initializes the game loop."""
        ...

    def update(self,dt):
        """Changes the ellipse position."""
        ...

    def draw(self):
        """Draws the ellipse"""
        ...
```

Parent class that does hard stuff

See animation.py

Designing a Game Class: Animation

```
class Animation(game2d.GameApp):
    """App to animate an ellipse"""

    def start(self):
        """Initializes the game loop."""
        ...

    def update(self,dt):
        """Changes the ellipse position."""
        ...

    def draw(self):
        """Draws the ellipse"""
        ...
```

Parent class that does hard stuff

See animation.py

Loop initialization
Do NOT use __init__

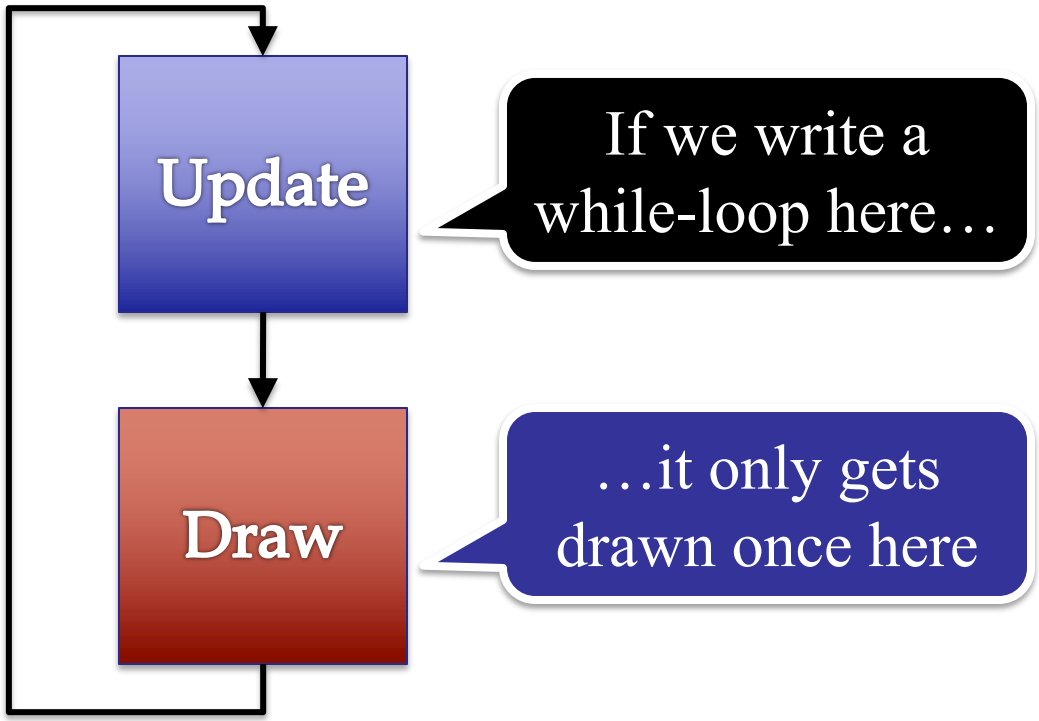
Loop body

Use method draw() defined in GObject

How Do We Change Animations?

- Remember the Pig game application
  - There were two turns: player and opponent
  - Each of those had its own while loop
- Same for animations: each is its own loop
- But we cannot write our own loops!
  - We did not write the loop; it is hidden
  - We only write the **body** of the loop
  - Writing our own loop "blocks" drawing

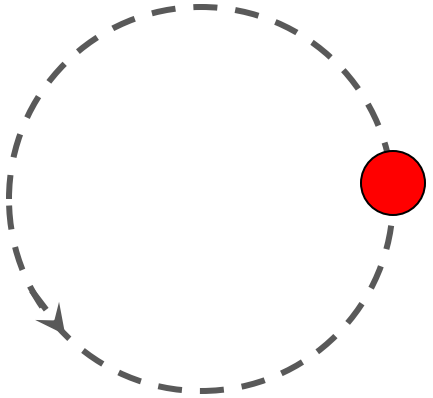
Understanding a GUI Application



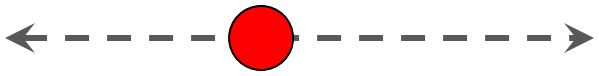
Solution: Application State

- **Idea:** *Interleave* loops
  - Write body for each loop
  - Pick right body in `update`
- **State**: Current loop body
  - Playing game vs. pausing
  - Choice of animation
- Add an attribute `state`
  - Method `update()` checks it
  - Executes correct helper

State `STATE_CIRCLE`



State `STATE_HORIZONTAL`



See `state.py`

Solution: Application State

- **Idea:** *Interleave* loops
  - Write body for each loop
  - Pick right body in update
- **State**: Current loop body
  - Playing game vs. pausing
  - Choice of animation
- Add an attribute state
  - Method update() checks it
  - Executes correct helper

```
def update(self,dt):
    """Animates current state"""

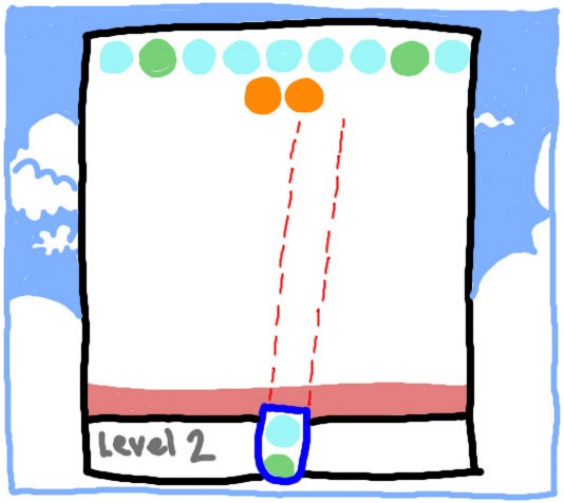
    # Determine the current state
    self._determineState()

    # Process the states
    if self.state==STATE_CIRCLE:
        self._animateCircle()
    elif self.state==STATE_HORIZONTAL:
        self._animateHorizontal()
    elif self.state==STATE_VERTICAL:
        self._animateVertical()
```

Aside: Storyboarding

- App design uses **storyboards**
  - Illustrations of different views
  - Diagrams how views switch
  - Important for UI design
- Each storyboard is a **state**
  - They explain state transition
  - But each state picture is static
  - (But often that Is enough)
- Purpose of tools like Figma
  - Taught in INFO courses





How Do We Represent State?

- Use an **enumeration**
  - List of fixed values
  - Each value a variable
  - Variable is a state name
- In Java/C this is a type
  - But we use int in Python
  - Achieves the same result
- Affects state **invariant**
  - Should not be any int
  - Should be in the list

```
# state before the game has started
STATE_INACTIVE = 0

# state when loading a new wave
STATE_LOADING  = 1

# state when the wave is in play
STATE_ACTIVE   = 2

# state when we pause the game
STATE_PAUSED   = 3

# state when the game is complete
STATE_COMPLETE = 5
```

The Refactored Pig Game (Again)

```
# The state variables
STATE_PLAYER = 1
STATE_AIBOT  = 2
STATE_COMPLETE = 3

class Pig(object):
    """An application to play the game pig."""

    def playMove(self):              # The update() method
        """Plays a single game move in the game of Pig"""
        if self._state == STATE_PLAYER:
            self._do_player()
        elif self._state == STATE_AIBOT:
            self._do_aibot()
```

The Refactored Pig Game (Again)

```
# The state variables
STATE_PLAYER = 1
STATE_AIBOT  = 2
STATE_COMPLETE = 3

class Pig(object):
    """An application to play the game pig."""

    def playMove(self):              # The update() method
        """Plays a single game move in the game of Pig"""
        if self._state == STATE_PLAYER:
            self._do_player()
        elif self._state == STATE_AIBOT:
            self._do_aibot()
```

Picks loop body

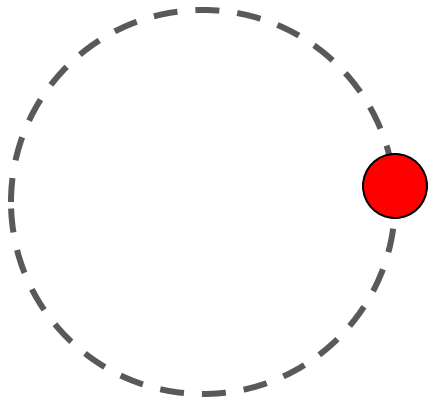
Stops entire app

Loop body

Loop body

When Do We Switch State?

- We switch for one of two reasons
  - Can be the result of an *event* (e.g. game over)
  - Can be the result of an *input* (e.g. a key press)
- Checks for this at the start or end of `update()`
  - At the **beginning** if **input driven**
  - At the **end** if **event driven**
- **Examples:**
  - Refactored `pig3.py` is it at the end
  - Animated `state.py` at the beginning



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

Checking Input

Keyboard

- is_key_down(key)
  - Returns True if key is down
  - key is a string ('a' or 'space')
  - Empty string mean
- is_key_pressed(key)
  - Returns True if key
  - 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
  - None if no touch
- h_pressed()
  - e if touch pressed
  - touch was None prev. frame
- is_touch_released()
  - True if touch released
  - touch **not** None prev. frame

All accessed from self.input in App

Next Time: Complex User Input