CS123

Programming Your Personal Robot

Part 2: Event Driven Behavior

2.2 Event Driven Programming Implementation

Topics

- Threads
  - What are threads?
  - Why use threads?
  - Communication between threads?
- Queues
- Implementing an Event System using Threads and Queue
  - Dispatcher
  - Handlers
- Folder Structure (Behavior Package)
- Home Work Assignment (part 1)

What are Threads

Running several threads is similar to running several different programs concurrently, but with the following benefits:

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they are “cheaper” than processes.

What are Threads For?

- Threads are used in cases where the execution of a task involves some waiting
- So we can execute multiple tasks "at the same time"

Basic Threads

```
#from threading import Thread
import threading
import time

# A thread that produces data
def first_thing():
    data = 0
    while (data < 10):
        data = data + 1
        print data
        time.sleep(0.1)

def main(argv=None):
    # creeating a thread
    t1 = threading.Thread(target=first_thing)
    #starting it
    t1.start()
    #wait until threads finish
    t1.join()
    print "thread 1 done”

if __name__ == "__main__":
    main()
```

Communication Between Threads

- Threads are running asynchronously
- Can communicate through global variables and parameters
- Queue is often used for communication between threads

Queue (in Python)

- The **Queue** module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The **Queue** class in this module implements all the required locking semantics.

Different "types" of Queue

- FIFO queue:
  - *class* Queue.**Queue**(*maxsize=0*): *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue.
- LIFO queue:
  - *class* Queue.**LifoQueue**(*maxsize=0*)¶
- Priority queue:
  - *class* Queue.**PriorityQueue**(*maxsize=0*)¶

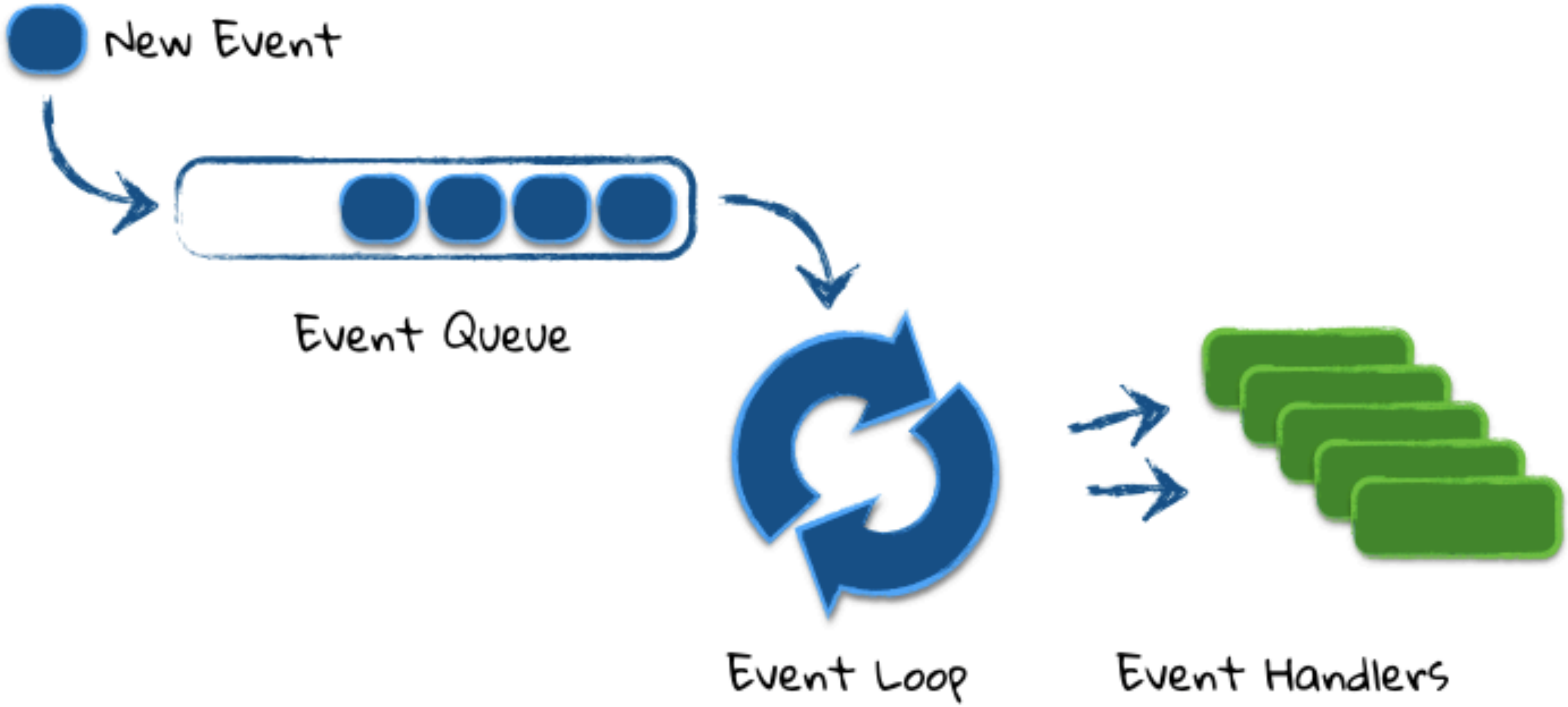
Priority Queue

- *class* Queue.**PriorityQueue**(*maxsize=0*)¶
  - The lowest valued entries are retrieved first (the lowest valued entry is the one returned by sorted(list(entries))[0]). A typical pattern for entries is a tuple in the form: (priority_number, data).

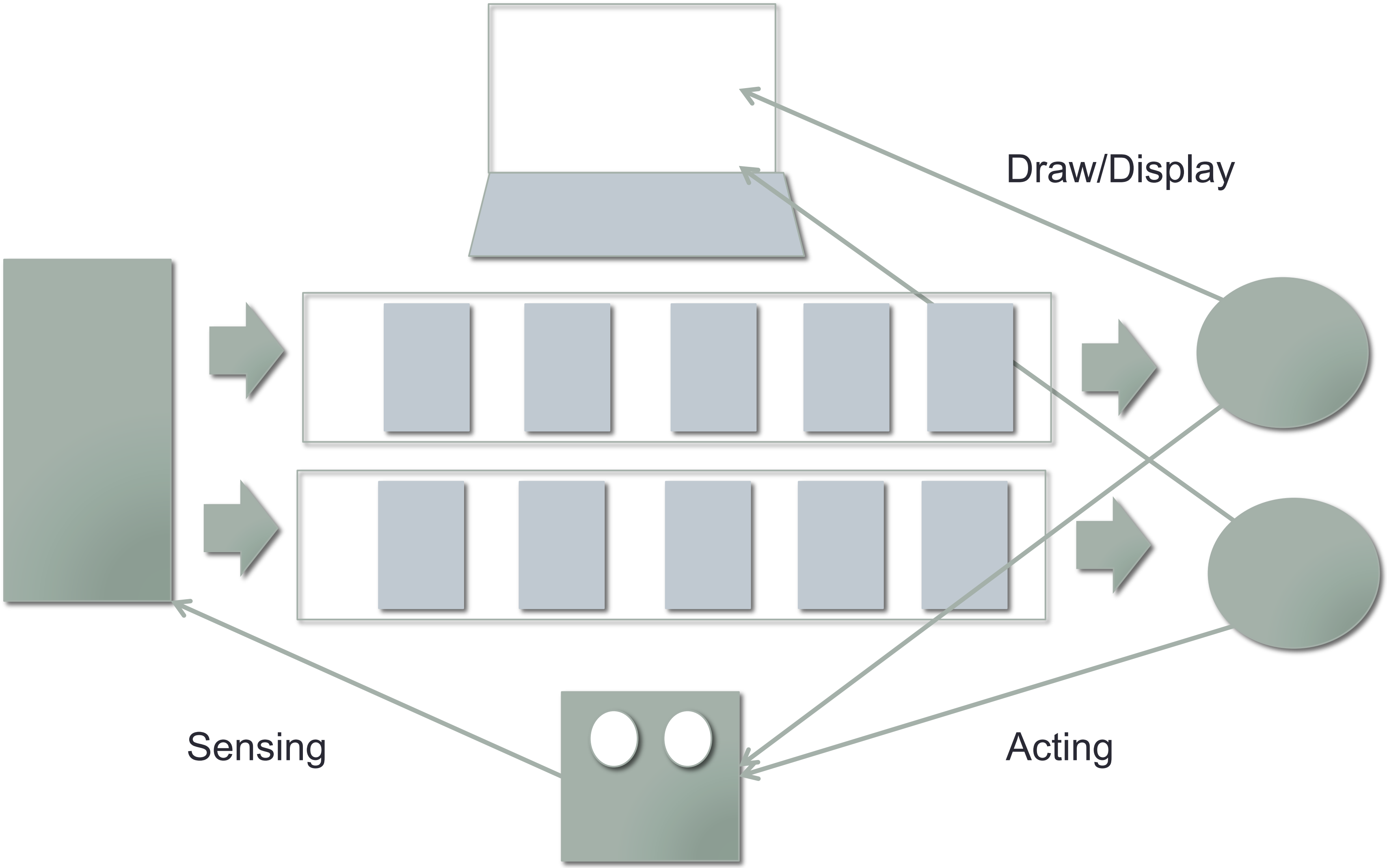
Basic Queue

- Queue.**qsize**()
- Queue.**empty**()
- Queue.**full**()
- Queue.**put**(*item*[, *block*[, *timeout*]])
  - Put *item* into the queue. If optional args *block* is true and *timeout* is None (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the **Full** exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the **Full** exception (*timeout* is ignored in that case).
- Queue.**put_nowait**(*item*)
  - Equivalent to put(item, False).
- Queue.**get**([*block*[, *timeout*]])
  - Remove and return an item from the queue. If optional args *block* is true and *timeout* is None (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the **Empty** exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the **Empty** exception (*timeout* is ignored in that case).
- Queue.**get_nowait**()

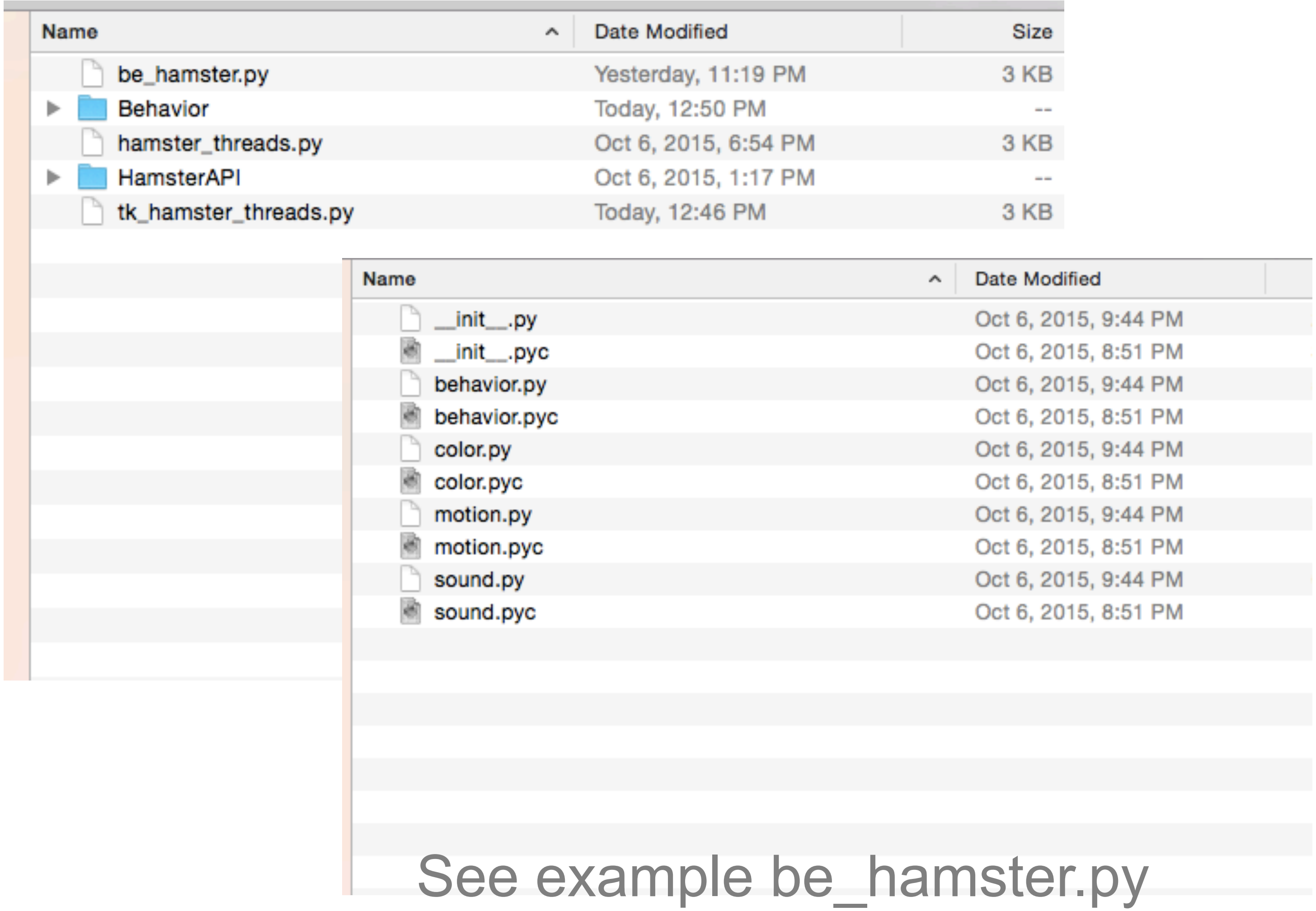
Event Queue



A Simple Structure Using Queues



Folder Structure – Behavior Package

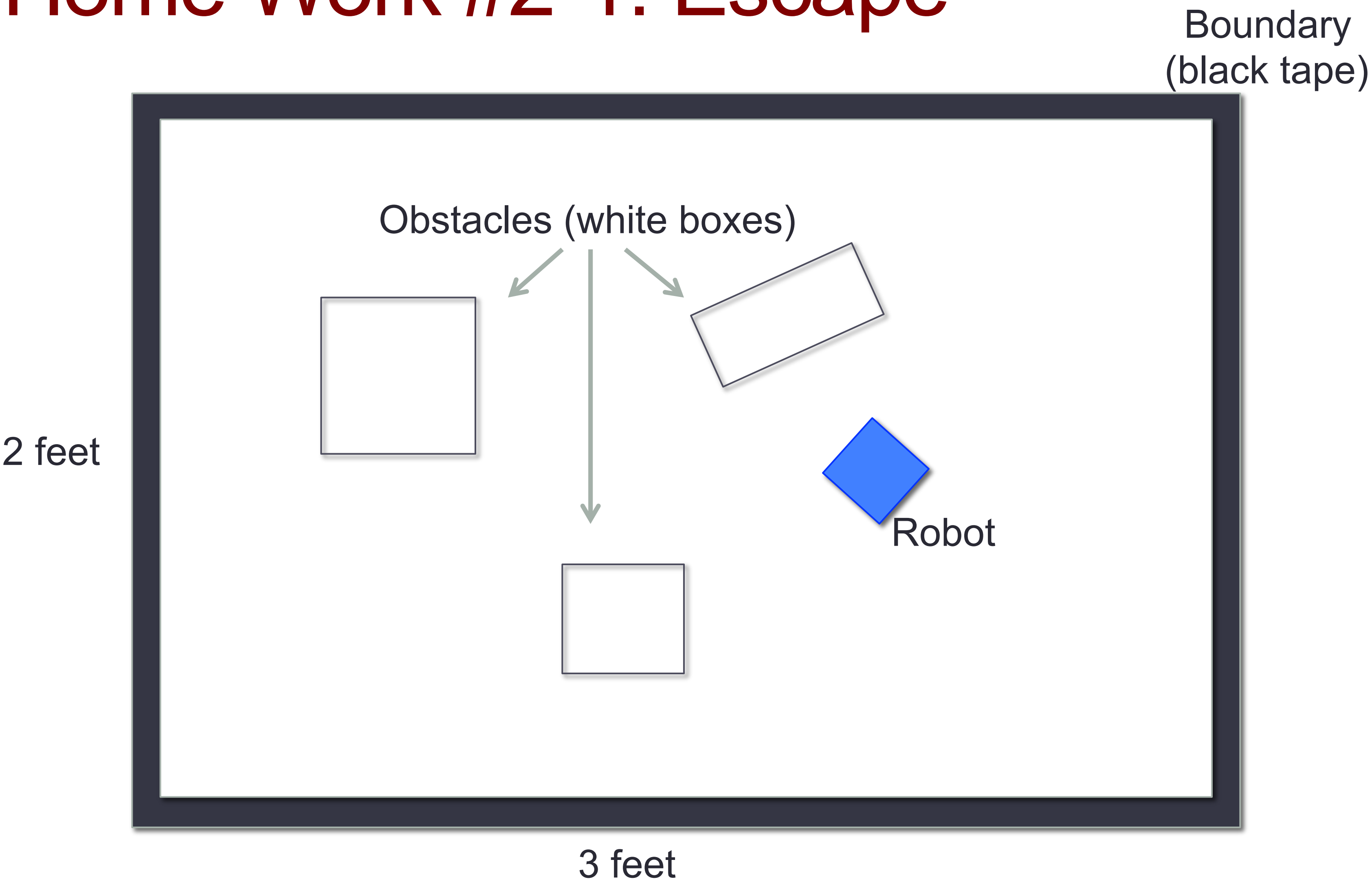


Name	Date Modified	Size
be_hamster.py	Yesterday, 11:19 PM	3 KB
▶ Behavior	Today, 12:50 PM	--
hamster_threads.py	Oct 6, 2015, 6:54 PM	3 KB
▶ HamsterAPI	Oct 6, 2015, 1:17 PM	--
tk_hamster_threads.py	Today, 12:46 PM	3 KB

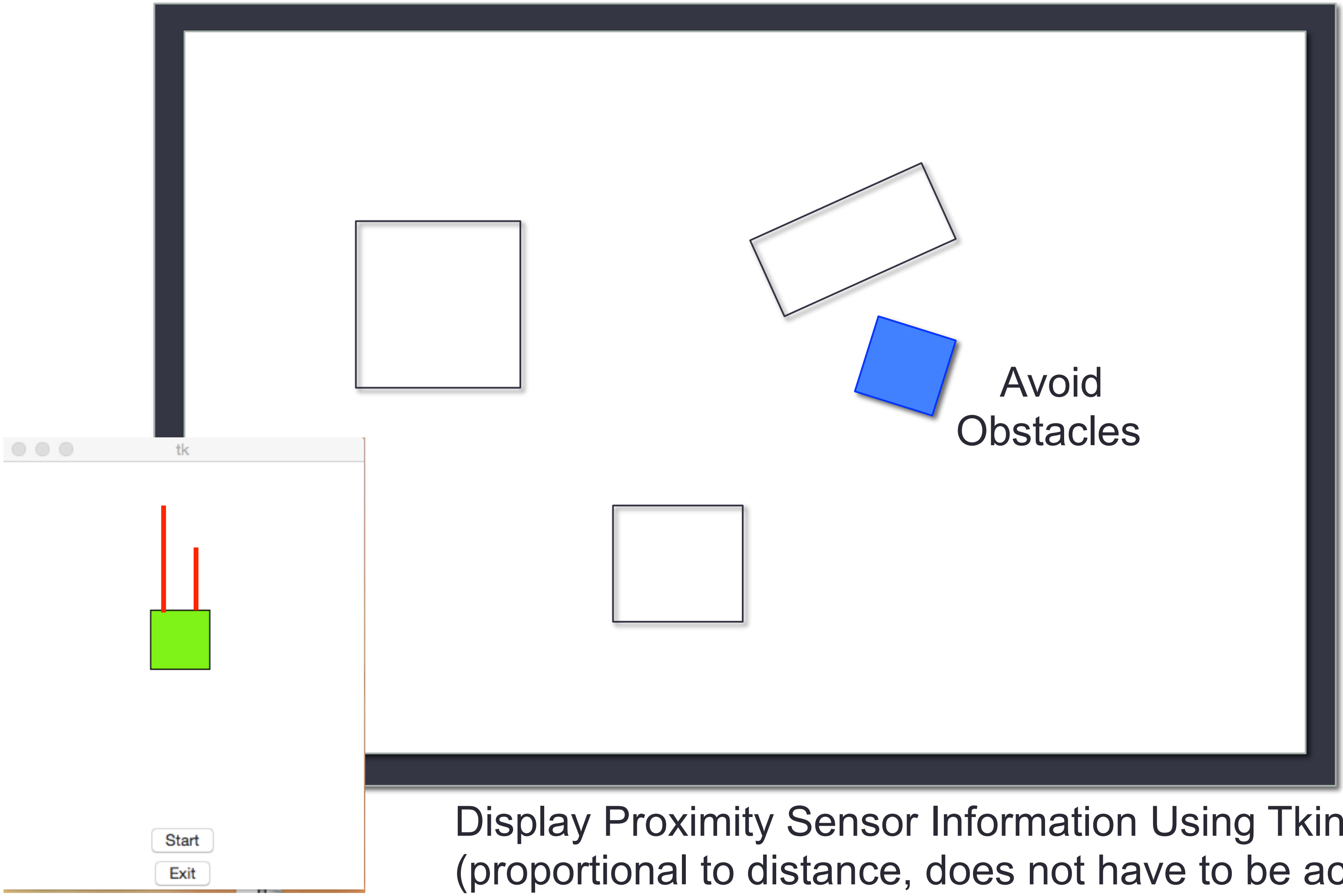
Name	Date Modified
__init__.py	Oct 6, 2015, 9:44 PM
__init__.pyc	Oct 6, 2015, 8:51 PM
behavior.py	Oct 6, 2015, 9:44 PM
behavior.pyc	Oct 6, 2015, 8:51 PM
color.py	Oct 6, 2015, 9:44 PM
color.pyc	Oct 6, 2015, 8:51 PM
motion.py	Oct 6, 2015, 9:44 PM
motion.pyc	Oct 6, 2015, 8:51 PM
sound.py	Oct 6, 2015, 9:44 PM
sound.pyc	Oct 6, 2015, 8:51 PM

See example be_hamster.py

Home Work #2-1: Escape



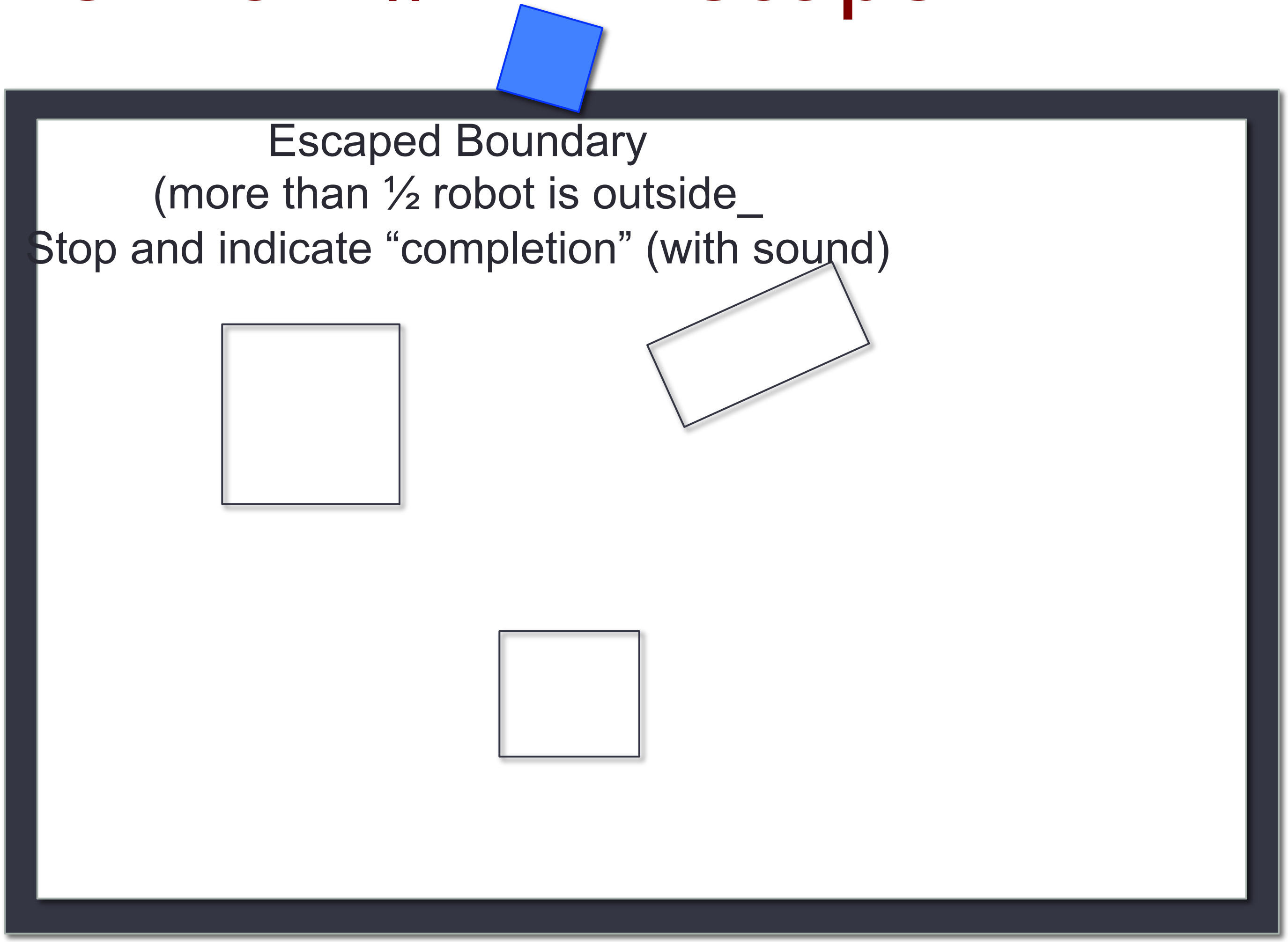
Home Work #2-1: Escape



Avoid Obstacles

Display Proximity Sensor Information Using Tkinter
(proportional to distance, does not have to be accurate)

Home Work #2-1: Escape



Escaped Boundary
(more than ½ robot is outside_
Stop and indicate “completion” (with sound)

Home Work #2-1: Escape

- Implement 2 or more “handlers” (each running on its own thread)
- Using 1 or more queues (for storing events)
- There are different ways you can implement “dispatcher” as discussed in class. Please put enough comments to make it clear how you implement it (or you can write up a description and submit with your homework)

2.3 Finite State Machine (FSM)