Toucan Crossing Assignment

An Object-Oriented Approach

Jasper Day

Thu 10/27/2022

In this paper, we outline an object-oriented, easily extensible approach to modelling traffic light behavior using the Python programming language. At its core, the simulation loops over a function whose output corresponds to one second, or “frame” of real time. Every frame, all of the objects in the scene run their update functions; the timers tick forwards, and the various outputs are created.

In its current iteration, the code only updates the traffic light and illuminates the Blinkstick, but the architecture of the code makes it very simple to extend its functionality to include a 2D representation of the scene and its contents, and to simulate the behaviour of pedestrians and cars (and update the traffic light based on that simulated behaviour).

1 Class Structure

The first classes defined are the `Period` and `TrafficLightAssignments` classes. These classes are simply wrappers to define constants, extending the functionality of Python’s `Enum` class. Their purpose is to make the code simpler and more readable: it’s more convenient to store the state of the traffic light as `Period.I` than as a simple `int`, and the `Enum` wrapper adds little overhead to the actual runtime of the program.

All of the scene information is contained in an object of the `Scene` class. The `Scene` class also implements the function for running a frame of simulation.

The traffic light logic is implemented in the `TrafficLight` class. The `update` method runs different checks depending on the current state of the light. For example, in the first period of operation (`Period.I`), the traffic light is stipulated to change if either:

1. A pedestrian has requested a change (by pressing the cross button) AND at least 6 seconds have elapsed since a car was last detected, OR
2. 20-60 seconds (depending on the size of the road) have elapsed since the beginning of the period.

To implement this logic, the method checks both the time since its state has changed (`window.time_since_update`) and the time since a pedestrian has requested a crossing (`window.time_since_button`, currently unimplemented in the rest of the code). If either of the requisite conditions (1) or (2) are met, then the traffic light changes state to the next period.

2 Simulation Output

Currently, the main output of the simulation is the Blinkstick itself, a matrix of RGB LEDs that changes color to match the current state of the traffic light. All of the interaction with the Blinkstick is sandboxed behind an Application Programming Interface (API) class titled `BlinkstickAPI` in the code.

When the API class is initialized, it attempts to find a Blinkstick. If none is found, an exception is raised and the program unwinds. Once the Blinkstick has been found, it can be interacted with through the functions within the `BlinkstickAPI` class.

`BlinkstickAPI.clear_blinkstick` clears the Blinkstick, and is called every frame of the simulation. `clear_blinkstick` is a solution to the problem otherwise encountered setting the LEDs on the Blinkstick: any LED set high in one state must be set low before switching to the next state. In future iterations of the program, it might be advisable to call this function only when the traffic light changes state. Currently, since the Blinkstick is cleared and re-lit every frame of the simulation, there is a slight but visible flicker in the Blinkstick as the simulation runs.

`BlinkstickAPI.set_light` is a method designed to interact with the `TrafficLightAssignments` class through use of the “splat” operator `**`. This operator unpacks a dictionary of keys and values into the arguments of a function. Using `BlinkstickAPI.set_light` in conjunction with the `TrafficLightAssignments` class avoids unnecessary repeated work in specifying the indices and values of the desired LEDs.

The output of the simulation could easily be extended. Since the `window` object contains a framework for simulating a top-down map of the crossing, it would be easy to implement a function that displays that map every frame. This could be accomplished through the terminal, with a text output, or through the use of a plotting library like `matplotlib`.

3 Further Extensibility

Thanks to the object-oriented paradigm, the written code is easy to extend and change as the engineering requirements of the simulation change over time. A framework is included for scene simulation with realistic pedestrians and cars. These objects would be implemented as subclasses of the `SceneObject` class, which allows the creation of arbitrary rectangular regions within the scene.

When the `SceneObject` is initialized, the program checks whether the coordinates of the corners are valid, raising an exception if given invalid input. The coordinates are stored, enabling simple methods universal to `SceneObject`s, particularly checking whether one `SceneObject` contains another (highly important for pedestrian crossing buttons and car detectors).

To make this framework fully functional, the following features need to be implemented:

1. A pathfinding function for the pedestrians and cars to update their position towards their destinations during the appropriate periods of the traffic light
2. A factory function (or set of factories) that creates cars and pedestrians at random positions with random destinations throughout the simulation
3. An `update` function for the `Pedestrian` and `Car` classes that changes the simulation state appropriately.

With these simple additions, a fully-fledged traffic light simulation application could easily be created. And thanks to the class architecture, adding those features would be as simple as extending the functionality of the given classes with a few new methods.

4 Conclusion

This software demonstrates a simple simulation of the state of a "toucan crossing" style traffic light. The result is a highly-readable yet extensible codebase, which makes use of Python's object-oriented architecture. The code interfaces with external libraries, creates physical and textual outputs, and accurately and completely implements the logic outlined in national standard LTN2/95 for traffic lights.

A Traffic Light Periods and Timings

Period	Use	Signal for Pedestrians	Signal for Vehicles	Timing	Variation
I	Vehicle Running	Red	Green	20 - 60 (ends at either max time or on pedestrian demand + gap. Vehicle actuation cancels gap for 6 sec)	Traffic volume, pedestrian button
II	Amber to Vehicles	Red	Amber	3	n/a
III	Vehicle Clearance	Red	Red	1 (gap in vehicles) - 3 (vehicle present)	Vehicle actuation
IV	Pedestrian Crossing	Green	Red	4 - 7	n/a
V	Pedestrians keep crossing	Black	Red	3	n/a
VI	Pedestrian clearance	Black	Red	0 - 22 (pedestrian detection adds 2 sec)	Pedestrian detection
VII	Additional Pedestrian Crossing	Black	Red	0 - 3	Pedestrian detection
VIII	All red	Red	Red	1 - 3	n/a
IX	Red / Amber to Vehicles	Red	Red/Amber	2	n/a

B Toucan Crossing Python Code

```python
# ++============================================++
# || Programming for Engineers: Toucan Crossing ||
# ||-------------+------------+-----------------||
# || Jasper Day  |  S2265891  |  2022/10/20     ||
# ++============================================++

# Description:
# ============
# `Period` tracks the state of the traffic light as the simulation advances.
# The `window` object contains a grid on which `SceneObjects` are located.
#
# As the simulation advances, every object (the traffic light and the scene
# objects) is updated. The `window` object mutates as these objects are updated.
#
# All of the logic (for traffic light state changes and `SceneObject` updates)
# is contained in the `update` implied functions for their respective members.

from blinkstick import blinkstick
import matplotlib as mpt
import numpy as np
from enum import Enum
import time


class Period(Enum):
    # Enum type allows state comparison of our traffic light variable.
    # `int` enum lets us augment state by addition operator +=
    I = 1
    II = 2
    III = 3
    # I, II, and III are for Mayfield Roads
    # IA, IIA, and IIIA are for Westfield Mains
    IA = 4
    IIA = 5
    IIIA = 6
    # Pedestrian Cycle
    IV = 7
    V = 8
    VI = 9
```

```
    VII = 10
    VIII = 11
    IX = 12
    # red / amber for westfield mains
    IXA = 13

class TrafficLightAssignments(dict, Enum):
    # Traffic Light assignments.
    # Traffic lights can be set with the splat operator **
    MAYFIELD_ROADS_RED = {
        "index": 0,
        "name": "red"
    }
    MAYFIELD_ROADS_YELLOW = {
        "index": 1,
        "name": "yellow"
    }
    MAYFIELD_ROADS_GREEN = {
        "index": 2,
        "name": "green"
    }
    WESTFIELD_MAINS_RED = {
        "index": 5,
        "name": "red"
    }
    WESTFIELD_MAINS_YELLOW = {
        "index": 4,
        "name": "yellow"
    }
    WESTFIELD_MAINS_GREEN = {
        "index": 3,
        "name": "green"
    }
    PEDESTRIAN_LIGHT_GREEN = {
        "index": 6,
        "name": "green"
    }
    PEDESTRIAN_LIGHT_RED = {
        "index": 7,
        "name": "red"
    }
```

```
class Scene():
    def __init__(self, rows, columns):
        scene = np.zeros([rows, columns])

    # Global counters
    time_since_car = 0
    time_since_ped = 0
    time_since_button = 0
    # time since period updated
    time_since_update = 0

    # Advance simulation by one frame
    def simulation_step(self, traffic_light):
        global TIME

        # TODO: broadcast call for all objects to update themselves (position, state, etc)

        traffic_light.update(self)

        # Print current state to terminal
        print(f"State: {traffic_light.state}, Time: {self.time_since_update}")

        # Tick timers forwards
        TIME += 1
        self.time_since_button += 1
        self.time_since_car += 1
        self.time_since_ped += 1
        self.time_since_update += 1
        time.sleep(0.5)

class BlinkstickAPI():
    # interface to easily change traffic lights
    def __init__(self):
        self.bstick = blinkstick.find_first()
        if self.bstick == None:
            raise Exception("No blinkstick found")

    def clear_colors(self):
        for i in range(0,8):
            self.bstick.set_color(index=i, name="black")
```

```python
    def set_light(self, color_dict):
        # For use with TrafficLightAssignments
        self.bstick.set_color(**color_dict)


class TrafficLight():
    def __init__(self, state: Period):
        self.state = state
        self.blinkstick = BlinkstickAPI()

    def update(self, window: Scene):
        # Clear traffic light
        self.blinkstick.clear_colors()

        if self.state == Period.I:
            # Vehicle Running (Mayfield Roads)
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_GREEN)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_RED)

            if (window.time_since_car >= 6 and
                window.time_since_update >= 10 and
                window.time_since_update >= window.time_since_button):
                # Gap condition with pedestrian demand
                self.state = Period.II
                window.time_since_update = 0

            elif window.time_since_update >= 20:
                # Max vehicle running duration
                self.state = Period.II
                window.time_since_update = 0

        elif self.state == Period.II:
            # Amber to Vehicles (Mayfield Roads)
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_YELLOW)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_RED)

            if window.time_since_update >= 3:
                self.state = Period.III
                window.time_since_update = 0
```

```
        elif self.state == Period.III:
            # Vehicle Clearance
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_RED)

            if window.time_since_car >= 6 and window.time_since_update >= 1:
                # Gap condition
                self.state = Period.IXA
                window.time_since_update = 0

            elif window.time_since_update >= 3:
                self.state = Period.IXA
                window.time_since_update = 0

        if self.state == Period.IXA:
            # Amber / Yellow (Westfield Mains)
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_YELLOW)
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_RED)

            if window.time_since_update >= 2:
                self.state = Period.IA
                window.time_since_update = 0


        if self.state == Period.IA:
            # Vehicle Running (Westfield Mains)
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_GREEN)
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_RED)

            if (window.time_since_car >= 6 and
                window.time_since_update >= 10 and
                window.time_since_update <= window.time_since_button):
                # Gap condition with pedestrian demand
                self.state = Period.IIA
                window.time_since_update = 0

            elif window.time_since_update >= 20:
```

```python
                # Max vehicle running duration
                self.state = Period.IIA
                window.time_since_update = 0

        elif self.state == Period.IIA:
            # Amber to Vehicles (Westfield Mains)
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_YELLOW)
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_RED)

            if window.time_since_update >= 3:
                self.state = Period.IIIA
                window.time_since_update = 0

        elif self.state == Period.IIIA:
            # Vehicle Clearance
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_RED)

            if window.time_since_car >= 6 and window.time_since_update >= 1:
                # Gap condition
                self.state = Period.IV
                window.time_since_update = 0

            elif window.time_since_update >= 3:
                self.state = Period.IV
                window.time_since_update = 0

        elif self.state == Period.IV:
            # Pedestrian Crossing
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_GREEN)

            if window.time_since_update >= 6:
                # Appropriate time for 10+ m road
                self.state = Period.V
                window.time_since_update = 0

        elif self.state == Period.V:
            # Fixed black-out
```

```
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)

            if window.time_since_update >= 3:
                self.state = Period.VI
                window.time_since_update = 0

        elif self.state == Period.VI:
            # Pedestrian Clearance
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)
            if window.time_since_ped >= 2 or window.time_since_update >= 22:
                # Pedestrians extend clearance by 2s
                self.state = Period.VII
                window.time_since_update = 0

        elif self.state == Period.VII:
            # Additional Pedestrian Clearance
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)

            if window.time_since_ped >= 2 or window.time_since_update >= 3:
                self.state = Period.VIII
                window.time_since_update = 0


        elif self.state == Period.VIII:
            # All-Red for 2s
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_RED)

            if window.time_since_update >= 2:
                self.state = Period.IX
                window.time_since_update = 0

        elif self.state == Period.IX:
            # Red/Amber Period
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_RED)
            self.blinkstick.set_light(TrafficLightAssignments.MAYFIELD_ROADS_YELLOW)
            self.blinkstick.set_light(TrafficLightAssignments.WESTFIELD_MAINS_RED)
```

```python
            self.blinkstick.set_light(TrafficLightAssignments.PEDESTRIAN_LIGHT_RED)

            if window.time_since_update >= 2:
                self.state = Period.I
                window.time_since_update = 0

class SceneObject():
    def __init__(self, window:Scene, row_start:int, row_end:int, col_start:int, col_end:in

        # Check if bounds are valid
        if not (row_start <= row_end < window.scene.shape[0] and
                col_start <= col_end < window.scene.shape[1]):
            raise Exception("Incorrect bounds for object")

        self.row_start = row_start
        self.row_end = row_end
        self.col_start = col_start
        self.col_end = col_end
        self.letter = letter

    def contains(self, scene_object):
        return (self.row_start <= scene_object.row_start and
                self.row_end >= scene_object.row_end and
                self.col_start <= scene_object.col_start and
                self.col_end >= scene_object.col_end)

    def centroid(self) -> (int, int):
        return (np.floor((row_start + row_end)/2), np.floor((col_start + col_end)/2))

class Sidewalk(SceneObject):
    pass

class Road(SceneObject):
    pass

class MobileObject(SceneObject):
    def __init__(self,x:int,y:int,speed,destination: SceneObject, letter):
        super().init(row_start=x,row_end=x,col_start=y,col_end=y, letter=letter)
        self.destination = destination
        self.speed = speed
        self.dest_coords = destination.centroid()
```

```
    def update_coords(self,new_x,new_y):
        self.row_start = new_x
        self.row_end = new_y
        self.col_start = new_y
        self.col_end = new_y

    def time_step(self) -> (int, int):
        pass




class Car(MobileObject): # Contents of Address Register. just kidding.
    def __init__(self, x, y, destination: SceneObject):
        super().init(x=x, y=y, destination=destination, letter='C')

class Pedestrian(MobileObject): # Contents of Address Register. just kidding.
    def __init__(self, x, y, destination: SceneObject):
        super().init(x=x, y=y, destination=destination, letter='P')

# CONTROL FLOW STARTS HERE

# Size of the crosswalk simulation matrix. One square is 1 meter on a side.
ROWS = 70
COLUMNS = 30
TIME = 0 # seconds

window = Scene(rows=ROWS, columns=COLUMNS)

traffic_light = TrafficLight(Period.I)

while True:
    window.simulation_step(traffic_light)
```