Spike: 6

Title: Navigation with Graphs

Author: Luke Valentino, 103024456

Goals / deliverables:

Expand the Task 5 navigation graph simulation to demonstrate the following:

- A game world that is divided into a larger number of navigation tiles, and corresponding larger navigation graph structure.
- A path-planning system that can create paths for agents, based on the current dynamic environment, using cost-based heuristic algorithms that accounts for at least six types of 'terrain' (i.e. nodes with different costs).
- Demonstrate multiple independent moving agent characters (at least four) that are able to each follow their own independent paths.
- Demonstrate at least two different types of agents that navigate the world differently.

Technologies, Tools, and Resources used:

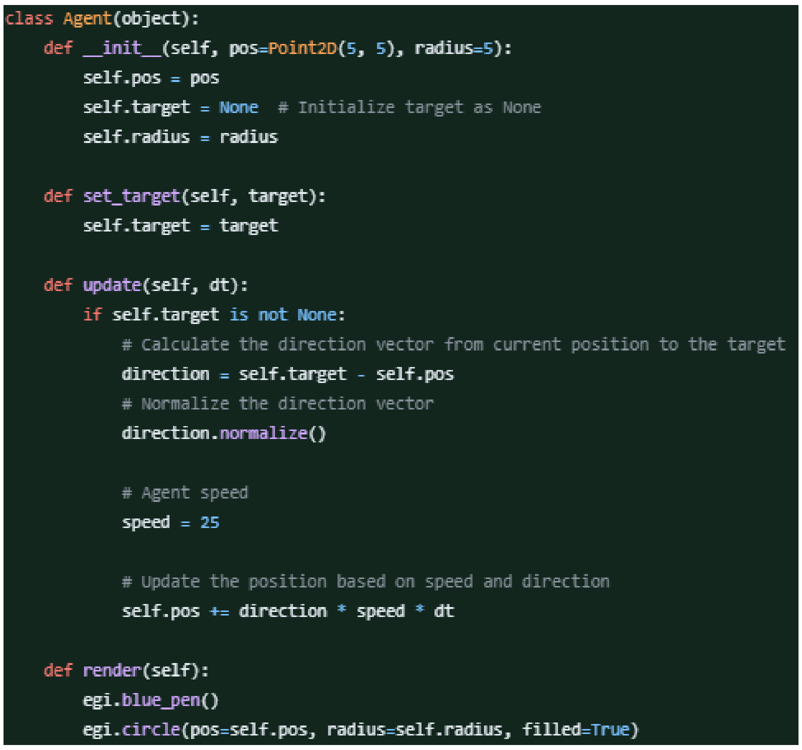
List of information needed by someone trying to reproduce this work.

- Swinburne Lecture Notes
- Swinburne Maths and Physics PDF
- docs.python.org
- Code from Lab 5

Tasks undertaken:

Start with the code from Lab 05. This represents the world as a grid graph and implements search algorithms to create paths from point a to b:

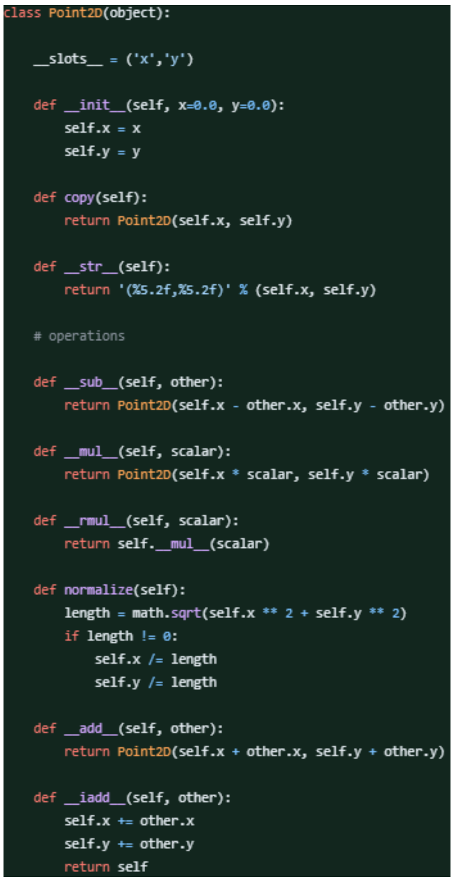
- Create an agent class. This class renders an agent in the window, and makes it move from at a constant speed. Currently it does not move within the graph, instead it moves freely.



```python
class Agent(object):
    def __init__(self, pos=Point2D(5, 5), radius=5):
        self.pos = pos
        self.target = None  # Initialize target as None
        self.radius = radius

    def set_target(self, target):
        self.target = target

    def update(self, dt):
        if self.target is not None:
            # Calculate the direction vector from current position to the target
            direction = self.target - self.pos
            # Normalize the direction vector
            direction.normalize()

            # Agent speed
            speed = 25

            # Update the position based on speed and direction
            self.pos += direction * speed * dt

    def render(self):
        egi.blue_pen()
        egi.circle(pos=self.pos, radius=self.radius, filled=True)
```

Initial Agent Class

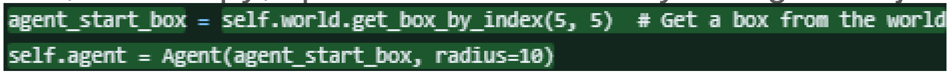
- To aid in the movement of the agent, you will need to expand the point2D class to account for more operations.



```
class Point2D(object):

    __slots__ = ('x','y')

    def __init__(self, x=0.0, y=0.0):
        self.x = x
        self.y = y

    def copy(self):
        return Point2D(self.x, self.y)

    def __str__(self):
        return '(%5.2f,%5.2f)' % (self.x, self.y)

    # operations

    def __sub__(self, other):
        return Point2D(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Point2D(self.x * scalar, self.y * scalar)

    def __rmul__(self, scalar):
        return self.__mul__(scalar)

    def normalize(self):
        length = math.sqrt(self.x ** 2 + self.y ** 2)
        if length != 0:
            self.x /= length
            self.y /= length

    def __add__(self, other):
        return Point2D(self.x + other.x, self.y + other.y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self
```

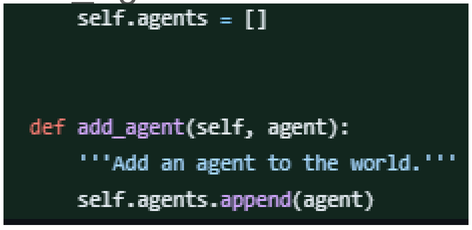
Expanded point2D class

- Next, in main.py, spawn an instance of your agent object.



```
agent_start_box = self.world.get_box_by_index(5, 5)  # Get a box from the world
self.agent = Agent(agent_start_box, radius=10)
```

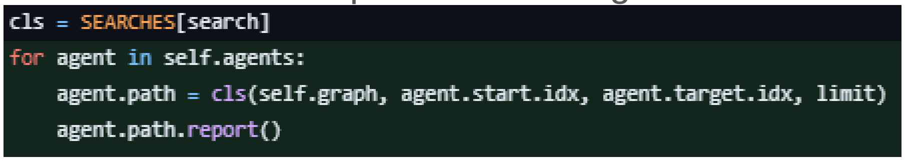
- Your agent should now be spawned in the world.
- The next step is to get the agent moving along a determined path that is calculated from the box world. To do this we will need to edit some existing code.
  - First modify the agent to hold a path object.
  - Next you need to modify the box_world.py file to hold a list of agents, with a subsequent add_agent method.



```
    self.agents = []


def add_agent(self, agent):
    '''Add an agent to the world.'''
    self.agents.append(agent)
```

  - Modify with draw function in box_world.py. Make sure it uses loops through every agent in self.agent and uses agent.path instead of self.path.

- Modify plan_path in box_world.py to loop through the list of agents. This method is to set the path for each agent in our world.



```
cls = SEARCHES[search]
for agent in self.agents:
    agent.path = cls(self.graph, agent.start.idx, agent.target.idx, limit)
    agent.path.report()
```

- Modify main.py to instantiate an agent with a start, a target, and a chosen search algorithm.
- Next we will modify our agent class to move each point at a constant speed. When the agent reaches the target point (the target point is within the proximity radius), the next point in the path is selected and set as the target. When the agent reaches the final target, it stops. Here are the methods:

Seek Method:

```
    def seek(self, target_pos, dt):
        # Moves the agent towards the target position following a direct path
        direction = target_pos - self.pos
        direction.normalize()
        self.pos += direction * self.speed * dt
```

Arrive Method:

```
    def arrive(self, target_pos, dt):
        # Moves the agent towards the target position. If target is in proximity
threshold,
        # sets the position to the target and stops movement
        direction = target_pos - self.pos
        distance = sqrt(direction.x**2 + direction.y**2)
        if distance <= self.waypoint_near_dist:
            self.pos = target_pos
            self.at_final_target = True
```

Path_Finished Method:

```
    def path_finished(self):
        # Checks if the agent has reached the end of the path.
        return self.current_node_index >= len(self.path.path)
```

Next_waypoint:

```
    def next_waypoint(self):
        # Advances the waypoint index to the next one in the path unless it's the last one
        if self.current_node_index < len(self.path.path) - 1:
            self.current_node_index += 1
        else:
            self.at_final_target = True
```

Is_near_waypoint:

```
    def is_near_waypoint(self):
        # Checks if the agent is near the current waypoint based on the defined near
distance threshold
        current_target = self.current_target()
        distance = sqrt((current_target.x - self.pos.x)**2 + (current_target.y -
self.pos.y)**2)
        return distance < self.waypoint_near_dist
```

Update:

```
    def update(self, dt):
        # Updates the agent's state every frame; checks path status and moves towards the
current target or handles arrival
        if self.path and not self.path_finished() and not self.at_final_target:
            if self.is_near_waypoint():
                self.next_waypoint()
            if not self.path_finished():
                self.seek(self.current_target(), dt)
            else:
                self.arrive(self.path.end_point(), dt)
        elif self.path_finished() or self.at_final_target:
            self.at_final_target = True
```

- Now that the agent is full functional, we can create subclasses with different behaviours. For example.

```
class FastAgent(Agent):
    def __init__(self, start, target, world):
        Agent.__init__(self, start, target, world, speed=100, radius=10)
        self.color = (0, 1, 0, 1)
```

What we found out:

This spike was one of the more difficult ones of the semester. Although I found it quite difficult it very much did address learning outcomes 1 and 2. In particular, this task focused on ULO 2.