SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30002

Tactical Steering (Hiding)

Task 10, Week 6
Spike: Spike_No10

Matthew Coulter
S102573957

23/04/2020

Tutor: Tien Pham

Table of Contents

Goals / Deliverables

The aim of this task is to produce a simulation game consisting of both hunters and prey in a space like environment. This environment shall contain hiding locations where you will program an artificial intelligence for the prey trying to find the most effective hiding location considering a range of factors. Here are a few as an example:

- distance from location
- location of hunter in relation to hiding location
- size of hiding location
- number of prey already at hiding location
- direction that the hunter/s are travelling

Beyond this, this task requires programming all the extra adaptations from the week 9 code to make this simulation function properly. This includes removing a lot of unnecessary code and adding variables that will control the seek, arrive, wander etc. functions that we have al;ready programmed. You will also need to draw a lot of the calculations that are happening to the screen to verify the code is functioning properly. i.e. calculation of the hiding spot in relation to the hunter position etc.

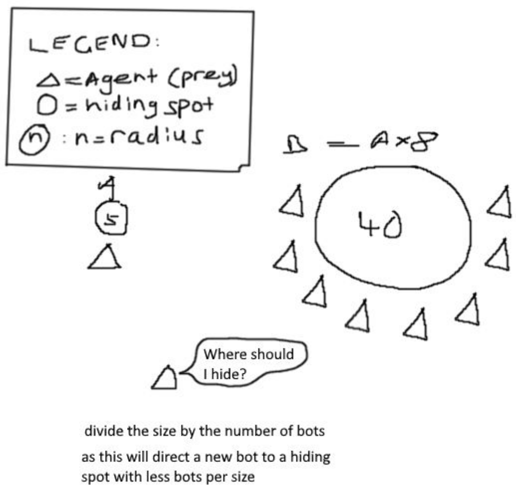
You also have the ability to extend on the program if you wish. Some extension features may include:

- dodging objects on screen
- having the hunter chase the prey
- having the hunter eat the prey
- etc.

My Spike report covers a very thorough analysis and tutorial regarding how to implement all of the aforementioned features. All of the code that I have written is provided and embedded into the structured way of how you could copy this code and learn from it.

Following this implementation of the code, an analysis of how the AI performs will be in the 'What we found out' section.

Besides the code that was created, I made a diagram that helps understand how the number of bots can determine which hiding location is better:



divide the size by the number of bots
as this will direct a new bot to a hiding spot with less bots per size

Technologies, Tools, and Resources used:

If you wish to reproduce my work, here is a list of all the required tools and resources that you will need:

- Visual Studio
- Python language pack
- This spike report
- Lab08 or lab09 code
- Git bash or sourcetree (if you wish to access my code)
    - repo: 'git clone https://mattcoulter7@bitbucket.org/mattcoulter7/cos30002-102573957.git'

Tasks undertaken:

Remove unwanted starting Code

Use the existing lab code and create a new project. Remove what you don't need from the code. (Always keep your code as small/specific as possible.) To start off, you can create a duplicate of your code from task 9, then we will begin to remove parts that we don't need…

We no longer need the path class as we will not be generating random paths for the agents to take, so we can delete that completely.

HideObject Class

Include several "objects" that prey can hide behind (simple circles).We can create a new class for our objects that prey will be hiding behind. The shape of it will only be a simple circle but we will also be able to change its size for different strategic choices of a hiding spot. Here is the code for that class:

```
from random import randrange
from vector2d import Vector2D
from vector2d import Point2D
from graphics import egi, KEY
from math import sin, cos, atan, radians
```

```
class HideObject(object):
    def __init__(self,world = None):
        self.world = world
        self.scale = 1.5
        self.size = randrange(15,100)
        self.color = 'GREEN'
        self.pos = self.randomise_location()
        # List of agents currently hiding at this object
        self.agents = []
```

The self.agents list will store all of the agents that are currently hiding behind this object. This will be useful for later if we want to include it as a factor into our hiding spot calculation.We can calculate the self.agents through the update function:

```
   def update(self):
        # Updates the agents list with current agents hiding here
        self.agents.clear()
        for prey in self.world.preys():
            if (prey.hiding_object is self):
                self.agents.append(prey)
```

When agents are focussing on this hiding location, it should be highlighted, hence we can control this in the render function. This will be called from the world render function.

```
   def render(self):
        egi.set_pen_color(name=self.color)
```

```
        egi.circle(self.pos,self.size)
        if (len(self.agents) > 0):
            self.color = 'GREEN'
        else:
            self.color = 'ORANGE'
```

The calculation to find where the hiding spot should be here. The hiding spot is not the centre of the hiding object, it is in relation to the position of the hunter, this way the prey will hide on the opposite side of the hunter. This calculation works by drawing a line between the hunter and the hiding object, then extending it a little bit further past the hiding object. This is achieved by using the normalise method as it reduces the length down to one. It is then extended by the scale * the size of the object to keep it consistent across all sizes.

```
    def get_hiding_pos(self,hunter_pos):
        # Uses angle to calculate best spot behind the hiding object
        position = Vector2D(hunter_pos.x - self.pos.x,hunter_pos.y - 
self.pos.y)
        position.normalise()
        return self.pos - self.size * self.scale * position
```

When we spawn the hiding object in, we will make a random location. However, if it is simply random, they will often overlap each other, hence we use the check_location_valid function to ensure they do not spawn over each other.

```
    def randomise_location(self):
        margin = round(self.world.cx/5)
        pos_x = randrange(margin, self.world.cx - margin)
        pos_y = randrange(margin, self.world.cy - margin)
        if (self.check_location_valid(Vector2D(pos_x, pos_y))):
            return Vector2D(pos_x, pos_y)
        return self.randomise_location()

    def check_location_valid(self,pos):
        # Checks if a hiding spot already exists near a new hiding spot
        for obj in self.world.hiding_objects:
            to_existing = obj.pos - pos
            dist = to_existing.length()
            if (dist < 2 * self.size):
                return False
        return True
```

Add circles to the environment that agents can hide behind. We need to create a list in the world that will store all of the hiding objects inside of.

```
def __init__(self, cx, cy):
        self.hiding_objects = []
```

When the game starts, 2 hiding objects will automatically be added. Add this code to main:

```
# add world objects to hide behind
    world.hiding_objects.append(HideObject(world))
    world.hiding_objects.append(HideObject(world))
```

Let’s alter the control so that when you press the ‘D’ key, a new hiding location will come in. Alter this function in main like this:

```
def on_key_press(symbol, modifiers):
    elif symbol == KEY.D:
        world.hiding_objects.append(HideObject(world))
```

Modifying Agent to be either hunter or prey

Show a distinction between the "hunter" and "prey" agent appearance and abilities. Let’s keep this simple, we will identify the hunters as red and the prey as blue. Add this code to the agent constructor:

```
# data for drawing this agent
        if (self.mode == 'hunter'): self.color = 'RED'
        elif (self.mode == 'prey'): self.color = 'BLUE'
```

Add two agents -- one designated as "hunter" and the other the "prey". When we initialise an agent, we want to specify the type it will be: either a hunter or a prey. We will update the calculate function to only have these two types, and update the constructor to accept a string of the type, matching the mode. We can also remove the AGENT_MODES from the agent class and remove those controls in main.

```
def __init__(self, world=None, mode=None, scale=30.0, mass=1.0):
      self.mode = mode
def calculate(self,delta):
        # calculate the current steering force
        mode = self.mode
        if mode == 'hunter':
            # Do Stuff
        elif mode == 'prey':
             # Do Stuff
        else:
            force = Vector2D()
        self.force = force
        return force
```

We can now initialise the agents by type in main like this:

```
# add hunter and pray agents
    world.agents.append(Agent(world,'hunter'))
    world.agents.append(Agent(world,'prey'))
```

Furthermore, when we want to add another agent of either type, we can assign a key for each type:

```
def on_key_press(symbol, modifiers):
    elif symbol == KEY.A:
        world.agents.append(Agent(world,'hunter'))
    elif symbol == KEY.S:
        world.agents.append(Agent(world,'prey'))
```

Because both types are stored in the same list in world called agents, we can make it slightly easier on ourself by creating functions for both types that will return a new list for either type. Add these functions to the world class:

```
def hunters(self):
        ''' Returns all of the hunters in agents '''
        hunters = []
        for agent in self.agents:
            if (agent.mode == 'hunter'):
                hunters.append(agent)
        return hunters

    def preys(self):
        ''' Returns all of the prey in agents '''
        preys = []
        for agent in self.agents:
            if (agent.mode == 'prey'):
                preys.append(agent)
        return preys
```

Keep it mind altering the result of these functions will have no effect on the agents list.

Hunter Behaviour

Make the hunter wander randomly. (It can "chase" as an extension). By default, the hunter will be wandering all the time, hence, under the calculate function, add the following code to the hunter type:

```
def calculate(self,delta):
        # calculate the current steering force
        mode = self.mode
        if mode == 'hunter':
                force = self.wander(delta)
        else:
            force = Vector2D()
        self.force = force
        return force
```

However, when one of the prey is in range of it, it should start to chase that prey. Extend the code to become this.

```
        if mode == 'hunter':
            closest_pray = self.closest(self.world.preys())
            dist = (closest_pray.pos - self.pos).length()
```

```
            if dist <= self.visibility:
                force = self.seek(closest_pray.pos)
            else:
                force = self.wander(delta)
```

In order to make this code work, we need to add another function for calculating the closest object to self in a given list. This will be a flexible function meaning we can use it for both agents and hiding objects. Add this code to the agent class:

```
def closest(self,objects):
        # Returns the closest object to self from a given list of objects
        dist_to_closest = None
        closest = None
        # Gets the closest object to hide behind
        for obj in objects:
            to_obj = obj.pos - self.pos
            dist = to_obj.length()
            if (closest is None or dist < dist_to_closest):
                closest = obj
                dist_to_closest = dist
        return closest
```

Notice it uses a new variable called self.visibility. This is the range of how far the hunter can see. We will reuse this for the prey detecting when they are seen and drawing the visibility to the screen. Add the following code to the render method in agent to draw the visibility circle.

```
def render(self, color=None):
        if self.world.show_info:
            # Visibility Radius
            if (self.mode == 'hunter'):
                egi.red_pen()
                egi.circle(self.pos,self.visibility)
```

Based on the hunter, draw a line from the hunter to the target object and then extend it through the object (by the object size plus a bit more). This is the hiding spot (as described in lecture notes). Show an indicator ("x" or similar) to indicate suitable "hide" locations for prey to select from

```
# Draws all hiding spots as an x from all hunters with a line
                for obj in self.world.hiding_objects:
                    hiding_pos = obj.get_hiding_pos(self.pos)
                    egi.line_by_pos(self.pos,hiding_pos)
                    egi.cross(hiding_pos,10)
```

Prey Behaviour

The prey should always be aiming to hide behind the objects UNLESS the hunter can see them. In that case, the prey should flee. Fortunately, we can reuse our arrive and flee functions to calculate the required force. Firstly, because we have designed our program to have multiple hunters, we need to

provide the flee function with a specific hunter to avoid. Sensibly, this should be the closest hunter to the prey. We will again use the closest function to calculate this. The new calculate function should include this:

```
elif mode == 'prey':
            closest_hunter = self.closest(self.world.hunters())
            force = self.flee(closest_hunter.pos)
```

Hence, we need to add the position argument into the flee function.

```
def flee(self, hunter_pos):
        ''' move away from hunter position '''
        ## add panic distance (second)
        panic_distance = self.visibility
        ## add flee calculations (first)
        dist_to_hunter = hunter_pos.distance(self.pos)
        # fly as fast as possible to escape hunter
        if dist_to_hunter <= panic_distance:
            # Goes quickly to get out of panic distance
            desired_vel = (hunter_pos + self.pos).normalise() *
self.max_speed
            return (desired_vel + self.vel)
        # Approaches hiding spot for the closest hiding spot calculated in
HideObject class
        return
self.arrive(self.hiding_object.get_hiding_pos(hunter_pos),'slow')
```

You may have noticed that this function has been slightly modified since the last lab. The panic_distance is set to the self.visibility because it represents how far the hunter can see. (For now all agents have the same visibility). Hence, there is a conditional statement whereby if the prey is within that panic distance, it will return the original result from flee. However, if it is not in that range, there is no reason to flee, hence it will try and reach the best location.

Calculating the Hide location

Prey agents must select a "good" location, and head to it, based on tactical evaluation.

There are two main variables to consider when choosing the hide location. These are:

1. Distance from the hide location and
2. Size of the hide location

To create a relationship between these two variables, we can simply divide the distance by the size. To do this pythonically, we can use a lambda function as a key for a min() function. Here is what that looks like:

```
(h.pos - prey.pos).length()/h.size
```

In order to implement this, we need to implement a good structured way of choosing the bot we want to use, and have our bots use that code. We will employ a method provided from the planetwars code. Essentially, we enter the name of the file that has the class for our bot into the initialisation for our world. The file must be in a subfolder called ‘bots’. This is done in main like this:

```
world = World(1000, 1000, 'MyNewBot')
```

Hence, we update our world constructor with this code

```
def __init__(self, cx, cy,bot):
        self.name = bot.replace('.py', '')  # accept both "Dumbo" or
"Dumbo.py"
        # Create a controller object based on the name
        # - Look for a ./bots/BotName.py module (file) we need
        mod = __import__('bots.' + bot)  # ... the top level bots mod
(dir)
        mod = getattr(mod, bot)       # ... then the bot mod (file)
        cls = getattr(mod, bot)      # ... the class (eg DumBo.py contains
DumBo class)
        self.controller = cls()
```

The self.controller simply refers to the class in the file name we specified. Hence, we should add that class and name it *‘MyNewBot.py’*.

```
from vector2d import Vector2D
from point2d import Point2D
from random import random, randrange, uniform
from agent import Agent

class MyNewBot(object):
    def update(self,world):
        for prey in world.preys():
            # Create ratio between distance and size
            closest_hunter_pos = prey.closest(world.hunters()).pos
            hiding_object = min(world.hiding_objects,key = lambda h:
(h.pos - prey.pos).length()/h.size)
            prey.hiding_object = hiding_object
```

Notice that this is where we inserted our formula. No constructor is needed as the world passed into the update function. We can call this from the world update method like so:

```
   def update(self, delta):
        if not self.paused and self.preys():
            # Updates the AI calculations
            self.controller.update(self)
```

All the main code is done now and should be working. Additionally if you like, you can make the hunters actually eat the pray. Modify the agent code with this.

```
def intersect_hunter(self):
        # Detects collision between hunter and self
        for hunter in self.world.hunters():
            to_hunter = self.pos - hunter.pos
            dist = to_hunter.length()
            if (dist < 20):
                return True
        return False
```

We can use this function for collision detection in the update function:

```
       # Eats prey
        if (self.mode == 'prey' and self.intersect_hunter()):
            self.world.agents.remove(self)
```

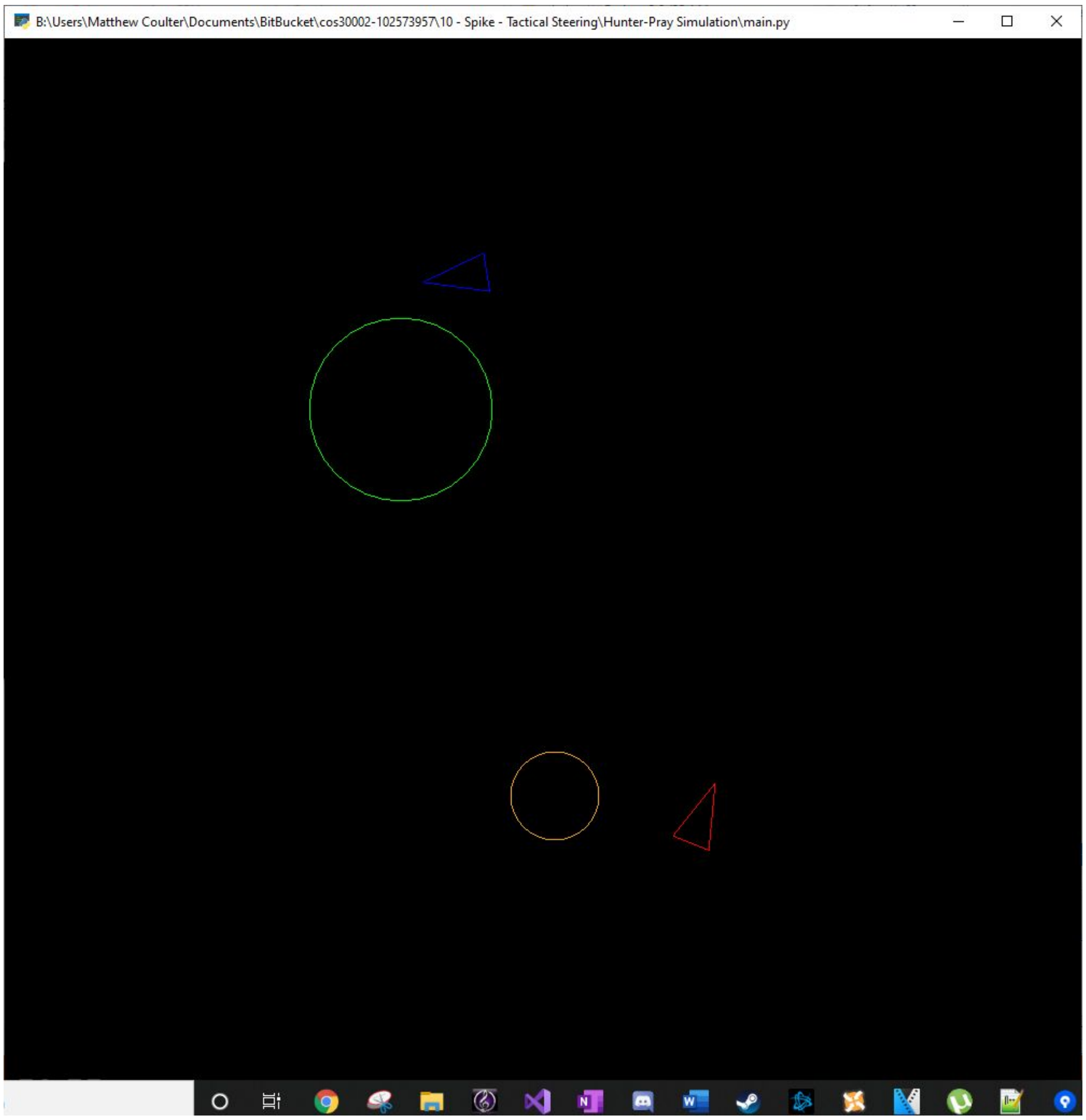
However, this causes one problem when there are no prey left on the map as the hunter will try to calculate the closest prey from an empty list. Hence modify the update function in the world class to stop the game if all the prey are dead.

```
   def update(self, delta):
        # Game stops when paused or no prey left
        if not self.paused and self.preys():
            # Updates the AI calculations
            self.controller.update(self)
            # Updates all agents
            for agent in self.agents:
                agent.update(delta)
            # Updates all all hiding positions, (checking for agents
hiding at it)
            for obj in self.hiding_objects:
                obj.update()
```

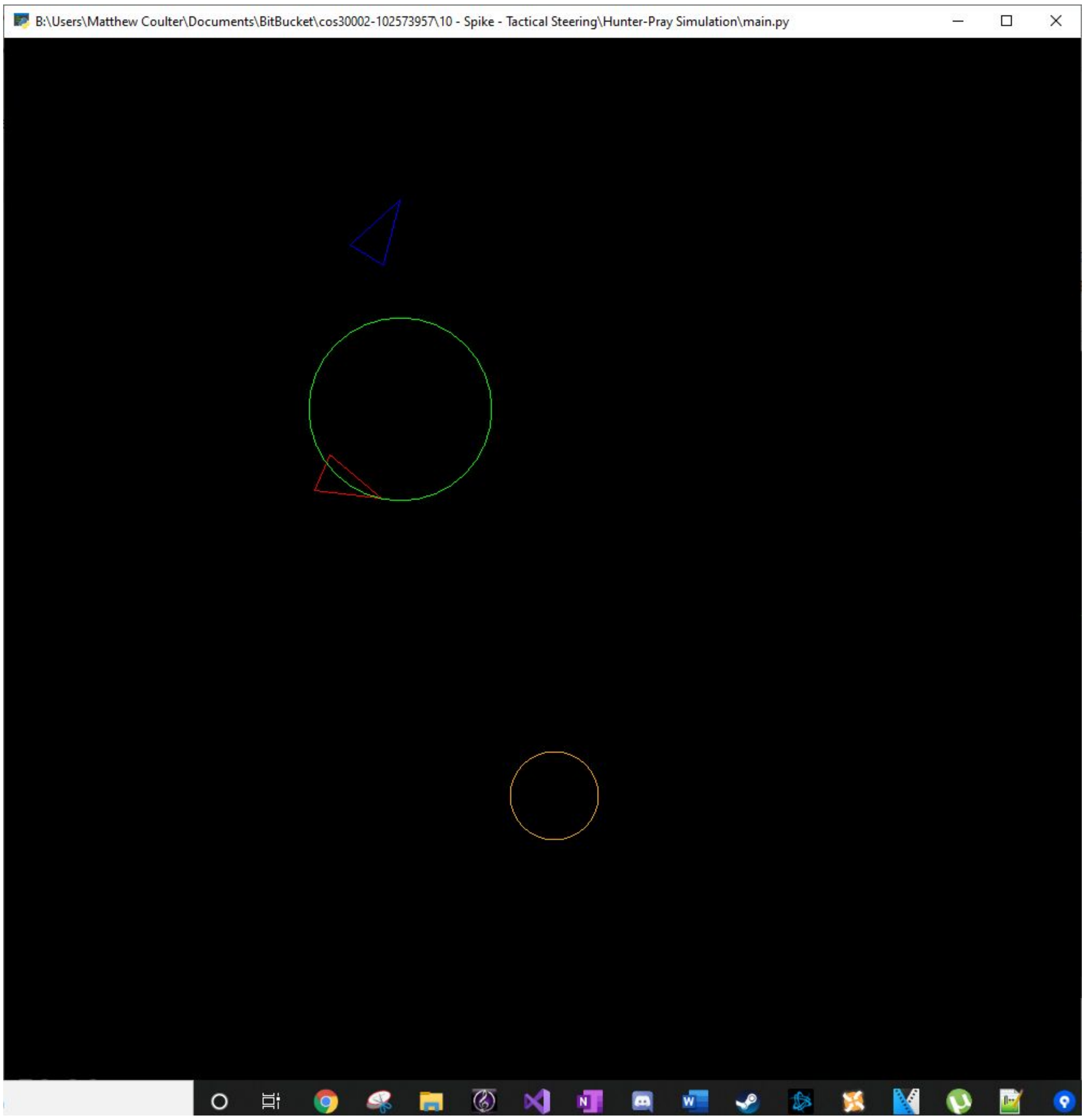
Remember, you can start the game again by pressing s to spawn another prey!

What we found out:

After all of the code is implemented, your game starting off should look like this:

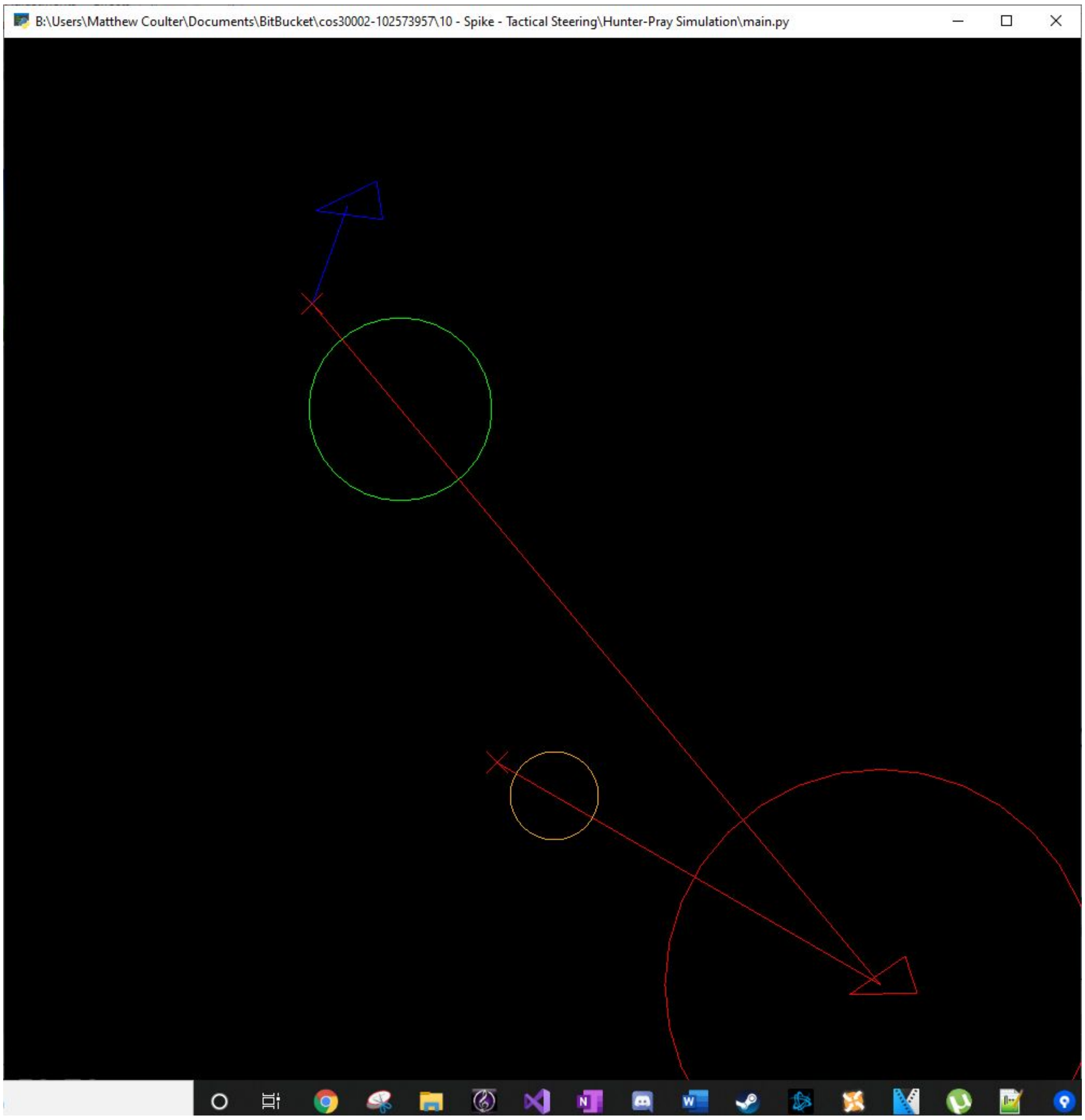

B:\Users\Matthew Coulter\Documents\BitBucket\cos30002-102573957\10 - Spike - Tactical Steering\Hunter-Pray Simulation\main.py

When the hunter (red) gets close to the prey, the prey will flee away:


B:\Users\Matthew Coulter\Documents\BitBucket\cos30002-102573957\10 - Spike - Tactical Steering\Hunter-Pray Simulation\main.py

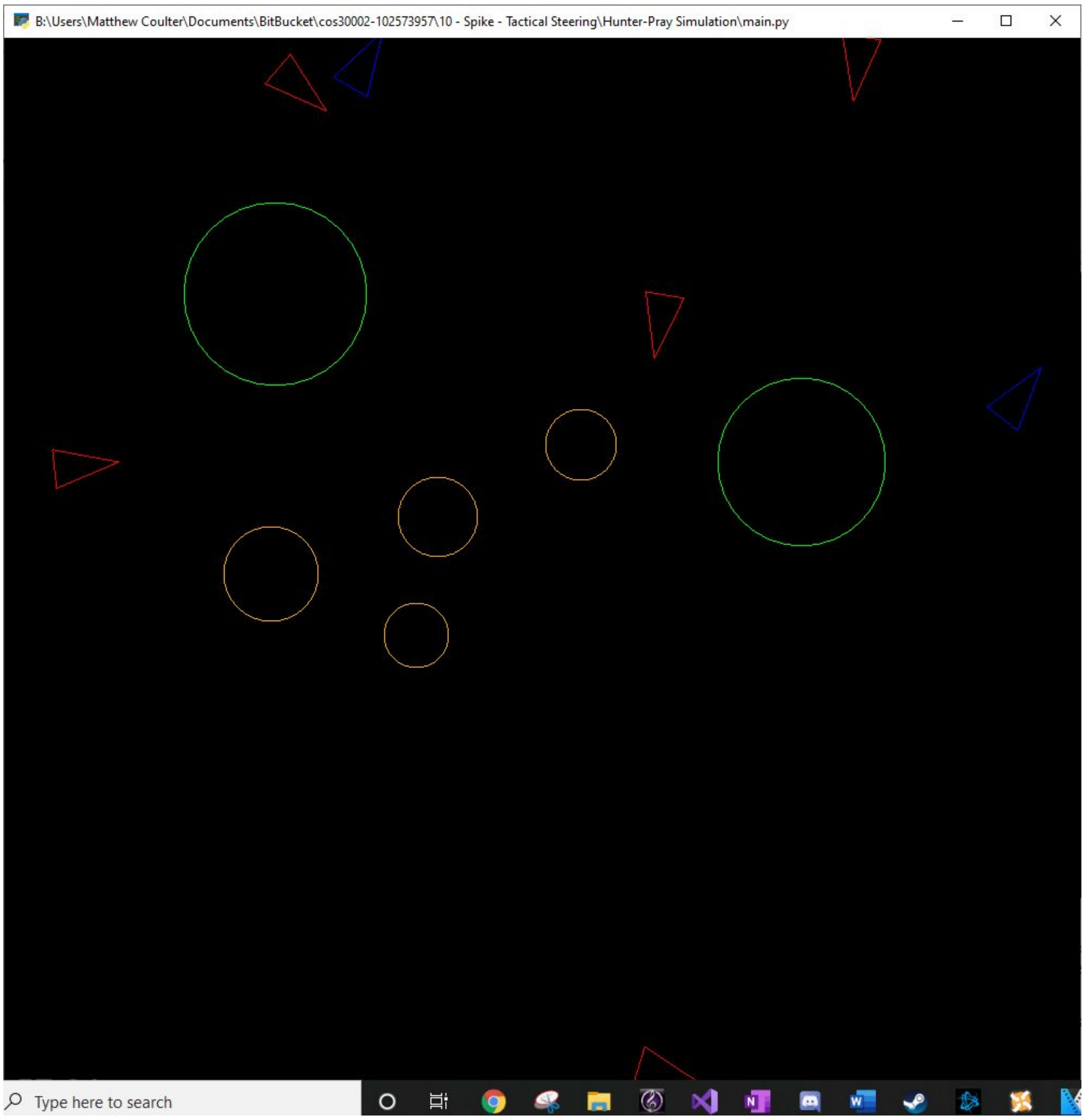
The green circle is highlighted as the one the prey is currently hiding behind.

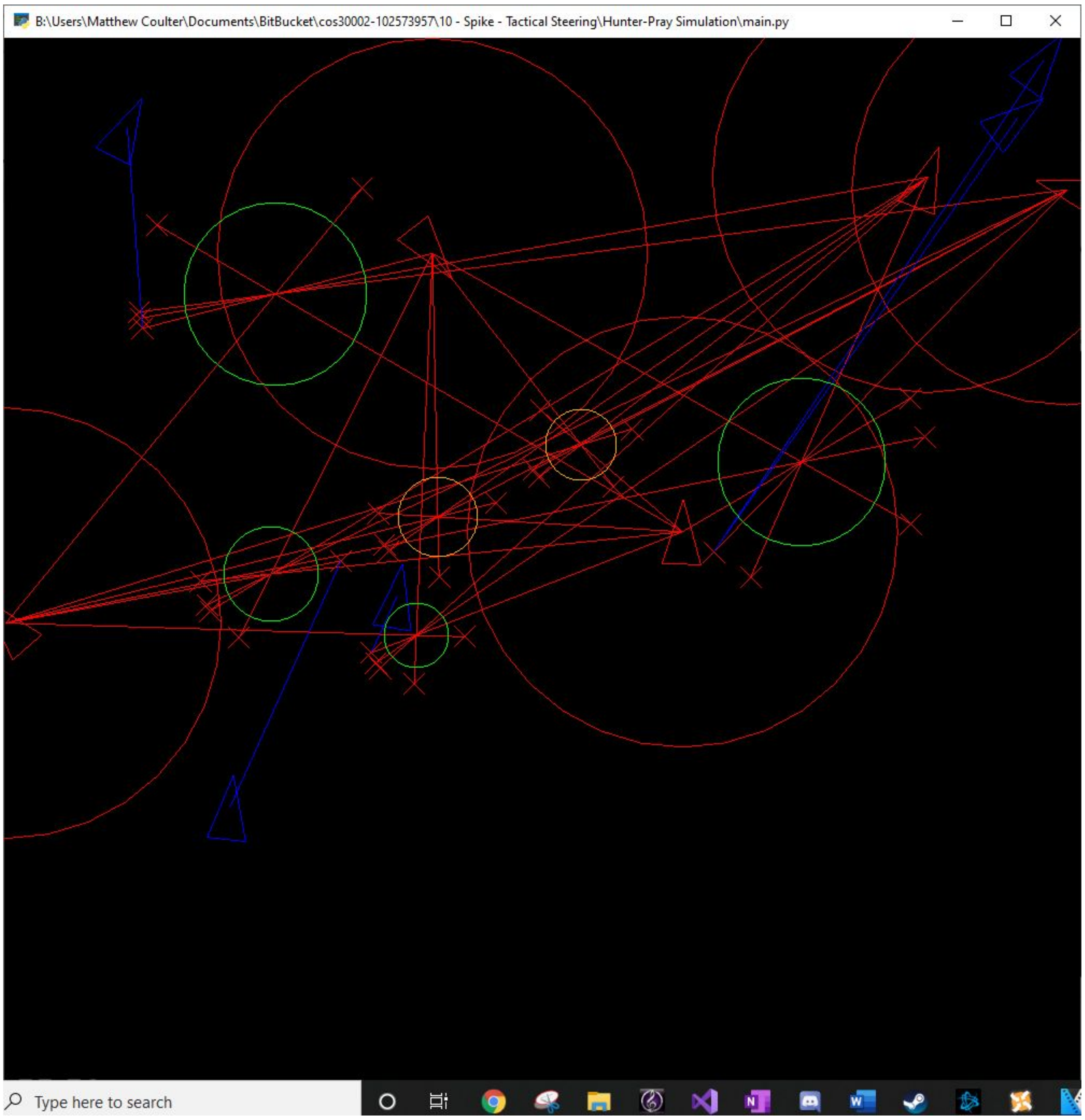
Presing 'I' will toggle the extra drawing data:


B:\Users\Matthew Coulter\Documents\BitBucket\cos30002-102573957\10 - Spike - Tactical Steering\Hunter-Pray Simulation\main.py

Notice there red crosses indicating where the prey can hide. It will hide at the cross on the hiding location calculated by MyNewBot. The red circle indicates the visibility of the hunter.

New prey can be added through the 'S' key, new hunters can be added through the 'A' key and new hiding location can be added through the 'D' key:


B:\Users\Matthew Coulter\Documents\BitBucket\cos30002-102573957\10 - Spike - Tactical Steering\Hunter-Pray Simulation\main.py
Type here to search


B:\Users\Matthew Coulter\Documents\BitBucket\cos30002-102573957\10 - Spike - Tactical Steering\Hunter-Pray Simulation\main.py
Type here to search

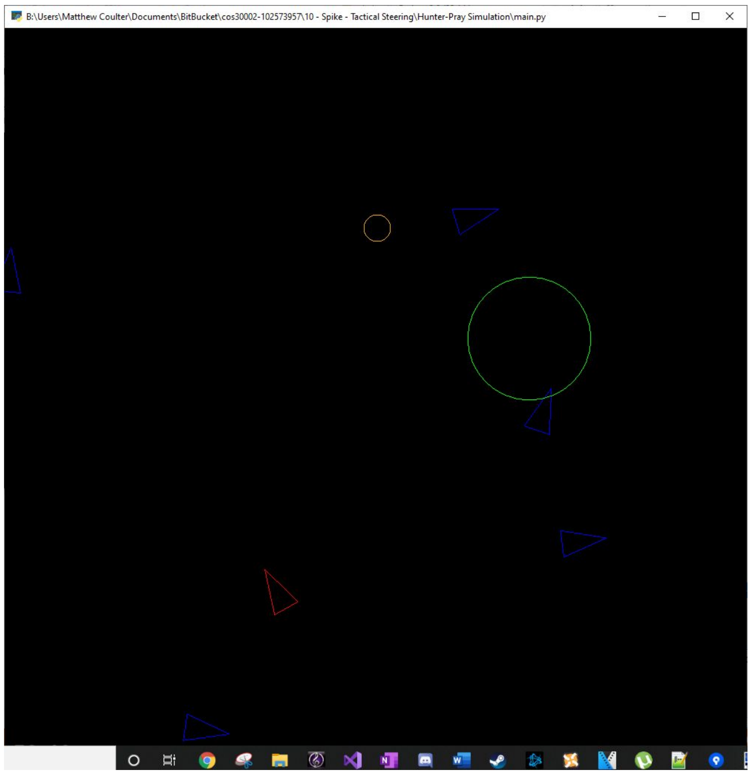
(Yes, it gets messy when there are lots of hunter on screen)

The performance of our bot can be measured through samples of how long it takes for our hunter to eat a number of prey. We will compare the survival times of:

5 prey which simply go to the closest hiding spot with 5 bots which uses our more advanced calculation.

The environment has 2 randomly generated hiding locations and one hunter. The world size is 1000x1000

For each both, the time 5 sample recording will be taken.


B:\Users\Matthew Coulter\Documents\BitBucket\cos30002-102573957\10 - Spike - Tactical Steering\Hunter-Pray Simulation\main.py

Record 1	Survival Time
Closest Hiding Spot	26.86s
MyNewBot	38.52s

Record 2	Survival Time
Closest Hiding Spot	69.15s
MyNewBot	85.36s

Record 3	Survival Time
Closest Hiding Spot	40.64s
MyNewBot	32.83s

Record 4	Survival Time
Closest Hiding Spot	3.08s
MyNewBot	41.31s

Record 5	Survival Time
Closest Hiding Spot	60.92s
MyNewBot	88.41s

Results:
Closest Hiding Spot (ave) = 40.13s
MyNewBot (ave) = 57.29s

Based on average, MyNewBot had a 17 second average better survival time. However, there are common denominators that both AIs suffer from. These include:

- grouping: prey will often follow a very similar path and multiple will in turn be eaten at the same time.
- The direction they try to flee in

Because there is a lot of chance with survival time, it is hard to confidently say that MyNewBot is a lot better than the simple Closest Hiding Spot bot.
In conclusion, a slightly more complex bot was implemented, however its results were not much better than a very basic bot. In order to improve this bot, they would need to be able to avoid running over each other then they will group less often.