COS30002

Agent Marksmanship

Task 14, Week 7
Spike: Spike_No14

Matthew Coulter
S102573957

30/04/2020

Tutor: Tien Pham

Table of Contents

Goals / Deliverables

The aim of this task is to implement a bot that will be able to predict the position of a target based on its velocity, position and also consider the speed of projectiles to shoot the other enemy and kill them. Doing so, we must implement a shooting system that will use both accuracy and speed variables to fire projectiles from an attacker. Essentially, when an attacker has a slower projectile and is further away, they need to predict further ahead of where the target is heading. Comparatively, if the target is closer and the attacker is shooting a fast projectile, they don't need to aim as far ahead of the target.

This spike report will cover the implementation of all of these goals, and also show the results of how effective the chosen technique was.

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:

Create an agent targeting simulation with:

an attacking agent,

First, we will need to either initialise our agents as attacking or being a target. We can set the agent.mode to be either one of these in our constructor.

```
elif symbol == KEY.M:
        world.agents.append(Agent(world,'target'))
elif symbol == KEY.N:
        world.agents.append(Agent(world,'attacking'))
```

To make things easier, as I have done before, we will implement a function in the world that returns a list of all the agents of a given mode. This will allow us to easily check if there are any targets on screen to look for.

```
   def agents_of_type(self,type):
        '''returns all agent of a given mode'''
        agents = []
        for agent in self.agents:
            if agent.mode == type:
                agents.append(agent)
        return agents
```

Our attacking agent will always be chasing the agent. Meaning, it will predict where the agent is going to be and seek that location. If there are no targets, the attacker will wander (use the wander function from previous labs)

```
def calculate(self,delta):
        # reset the steering force
        mode = self.mode
        force = None
        if mode == 'attacking':
            target = None
            if self.world.agents_of_type('target'):
                force =
self.chase(self.closest(self.world.agents_of_type('target')))
            else:
                force = self.wander(delta)
        else:
            force = Vector2D()
        return force
```

```
   def chase(self,target):
        '''seeks the predicted position of an agent'''
        predicted_pos = target.vel.copy()
```

```
        predicted_pos.normalise()
        predicted_pos *= self.get_look_ahead(target)
        target_pos = predicted_pos + target.pos
        return self.seek(target_pos)
```

Notice that chase calls a get_look_ahead function. The calculates how far ahead of the agent it needs to be heading. It will also be used later to calculate what direction it needs to shoot to intersect the target. Hence, I will show you the code at that stage as it is more relevant there.

Our attacker is going to need a weapon to shoot with. For now add this code to the constructor, I will show you the implementation of this class later on.

```
        self.weapon = Weapon(self,world)
```

a moving target agent

We will have our moving target wander by default.

```
elif mode == 'target':
            force = self.wander(delta)
```

a selection of weapons that can fire projectiles with different properties.

Add a new class with the following code for weapon:

```
from projectile import Projectile
from point2d import Point2D
from graphics import egi, KEY
from queue import Queue

PROJECTILE_SPEED = {
    KEY.F: 1000.0, # fast
    KEY.S: 500.0 # slow
}

PROJECTILE_ACCURACY = {
    KEY.A: 0.5, # small margin of error for perfect shot
    KEY.I: 10.0 # big margin of error for perfect shot
}
```

We will use these dictionaries to control the speed and accuracy of the projectiles. There are two settings for both variables.

```
class Weapon(object):
    """description of class"""
    def __init__(self,agent = None,world = None):
        self.projectiles = []
        self.projectiles_queue = Queue(maxsize = 1)
        self.agent = agent
        self.world = world
```

```
        self.color = 'WHITE'
        self.shape = [
            Point2D(0.0,  0.0),
            Point2D( 1.0,  0.0),
            Point2D( 1.0, -0.25),
            Point2D( 0.25, -0.25),
            Point2D( 0.25, -0.5),
            Point2D( 0.0, -0.5),
            Point2D( 0.0, 0.0)
        ]
        self.proj_speed = 1000.0
        self.accuracy = 0.5
        self.initialise_queue(self.projectiles_queue)
```

I felt that it would be appropriate to use the object pool design pattern for the bullets. Essentially rather than using up more memory by having the projectiles removed from the list, the allocating new memory for new projectiles, we can reuse the projectiles. This works by having a list and a queue simultaneously work together. When the attacker is initialised, the queue is filled up to max size, when they fire, the queue puts the item into the list. When the projectile is ready to dispose, it will be put back into the queue for reuse.

```
    def update(self,delta):
        # Update position of all projectiles
        for proj in self.projectiles:
            proj.update(delta)
```

I will show you the update class code soon

```
    def render(self):
        '''Renders gun to screen'''
        egi.set_pen_color(name=self.color)
        pts = self.world.transform_points(self.shape, self.agent.pos,
self.agent.heading, self.agent.side, self.agent.scale)
        egi.closed_shape(pts)
```

```
    def shoot(self):
        '''Moves obj from queue into list'''
        # Shoot if queue still has obj to move
        if not self.projectiles_queue.empty():
            # Get obj at front of queue
            proj = self.projectiles_queue.get()
            # Prepare for shooting
            proj.calculate()
            # Put obj in list so it is updated
            self.projectiles.append(proj)
```

It calls a calculate() function at the time of shooting so the position is reset again. This way, we can also update the speed for any new bullet that will be fired after the player has changed the speed setting.

```
    def initialise_queue(self,queue):
        '''Fills up a queue to its max size'''
        for i in range(queue.maxsize):
            queue.put(Projectile(self.world,self))
```

As a bonus to the object pooling, if we set the max size to 1, this means that the attacking agent will only shoot one bullet at a time, until that bullet has either hit a target or gone out of the screen, then they will again be able to fire a projectile.

Here is the projectile class code:

```
from graphics import egi
from vector2d import Vector2D

class Projectile(object):
    """description of class"""
    def __init__(self,world,weapon):
        self.world = world
        self.weapon = weapon
        self.pos = Vector2D()
        self.vel = Vector2D()
        self.max_speed = None

    def update(self,delta):
        # check for limits of new velocity
        self.vel.normalise()
        self.vel *= self.max_speed
        # update position
        self.pos += self.vel * delta
        # Recycle self back into queue if outside of map
        if self.intersect_edge():
            self.recycle()

        agent_hit = self.intersect_agent()
        if agent_hit:
            self.world.agents.remove(agent_hit)
            self.recycle()

    def render(self):
        egi.circle(self.pos,5)

#------------------------------------------------------------------------

    def intersect_edge(self):
        '''check if projectile goes out of the map'''
```

```
        if self.pos.x < 0:
            return True
        elif self.pos.x > self.world.cx:
            return True
        elif self.pos.y < 0:
            return True
        elif self.pos.y > self.world.cy:
            return True
        return False

    def intersect_agent(self):
        '''check if projectile hits another agent'''
        for agent in self.world.agents:
            if agent is not self.weapon.agent:
                to_agent = agent.pos - self.pos
                dist = to_agent.length()
                if dist < 30:
                    return agent
        return False

    def recycle(self):
        '''remove projectile from list and put it at the end of the queue
for reuse'''
        self.weapon.projectiles.remove(self)
        self.weapon.projectiles_queue.put(self)

    def calculate(self):
        '''prepare to be put back onto the screen'''
        self.max_speed = self.weapon.proj_speed
        self.pos = self.weapon.agent.pos.copy()
        self.vel = self.weapon.agent.vel.copy()
```

You can read the comments to understand the purpose of each function. The only real understanding that you need to know about the projectile is that it has an initial velocity and that does not change. For the purpose of this simulation, we don't need to add any resultant force on it to slow down. Hence, a copy of the agent velocity at the time of shooting is grabbed to prepare for shooting.

Be able to demonstrate that the attacking agent that can successfully target (hit) with different weapon properties:

The shooting agent will need to use the target's position and velocity, together with the projectile speed, to work out an interception point to shoot at.

We don't want our agent to be shooting all the time… we only want it to shoot when it will predict the path of the target and land a shot on the target. Hence we need to consider many variables. These variables include the target's position and velocity alongside and projectile speed and how far away the attacker is from the target. We will have a boolean function that returns true if the AI thinks it is a good time to shoot. We will call this function should_shoot().

```
   def should_shoot(self):
        '''uses distance, proj velocity and target agent velocity to
predict which direction to shoot'''
        if self.world.agents_of_type('target'):
            for agent in self.world.agents_of_type('target'):
                # Predicts where the
                predicted_pos = agent.vel.copy()
                predicted_pos.normalise()
                predicted_pos *= self.get_look_ahead(agent)
                to_predicted = agent.pos + predicted_pos - self.pos
                angle = self.vel.angle_with(to_predicted) * 180 / pi
                if angle < self.weapon.accuracy and angle >
-self.weapon.accuracy:
                    return True
        return False
```

It loops through every target on screen and determines whether if it were to shoot now, would it hit a target? It achieves this through two considerations.

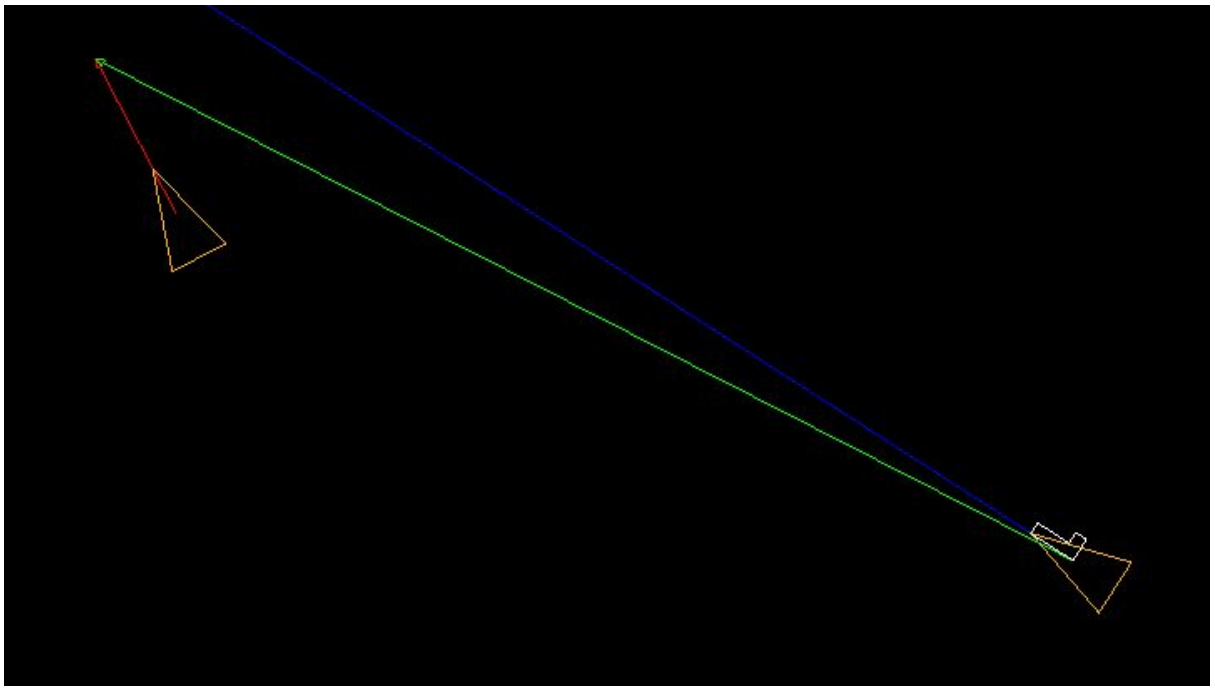
The first consideration is the get_look_ahead function that I previously mentioned. This function considers position, velocities and distance to calculate how far it would need to shoot ahead.

```
   def get_look_ahead(self,agent):
        to_target = agent.pos - self.pos
        dist = to_target.length()
        look_ahead = dist * agent.speed() / self.weapon.proj_speed
        return look_ahead
```

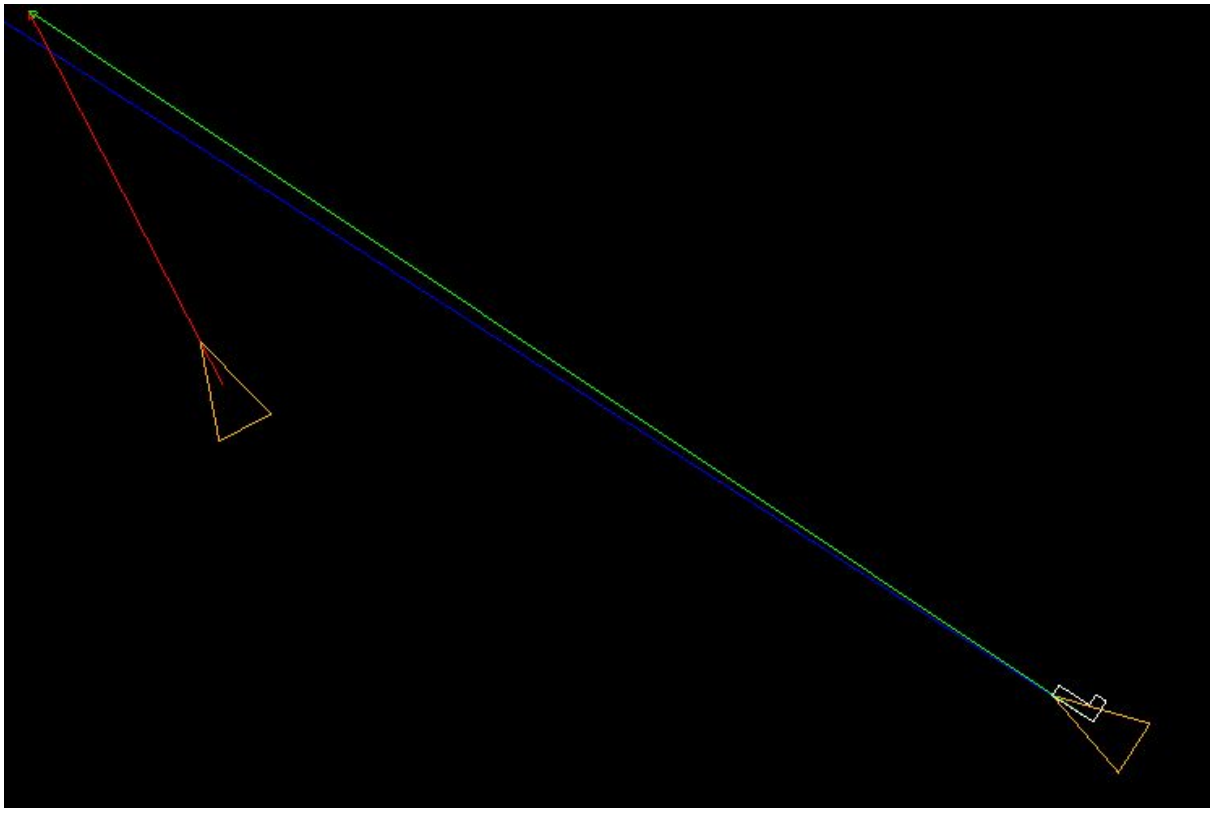
Essentially, as our attacker is further away it will need to look further ahead, and if the projectile is slow, it will need to look even further. However, if it is closer it can look closer ahead of the target and a faster projectile will also shorten that distance. It also considers the speed of the target as if it is moving faster, it will also need to project further in front.

The following images capture this behaviour:\

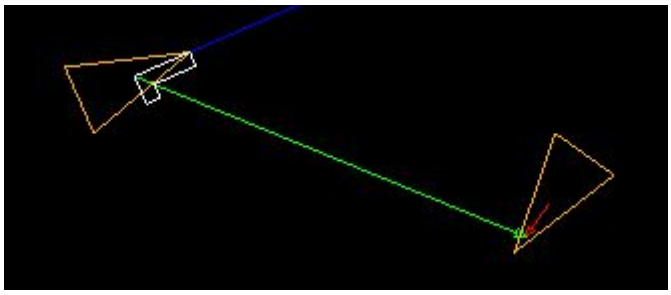
Far away fast projectile



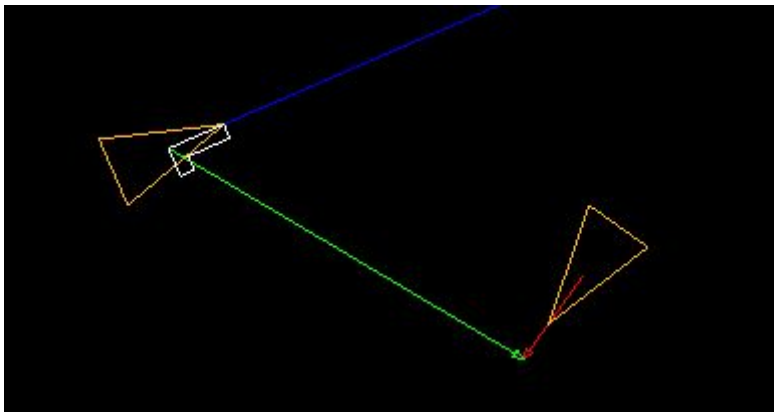
far away slow projectile



close fast projectile



close slow projectile



The key part of calculating that distance is in this formula.

$$look\_ahead = dist * agent.speed() / self.weapon.proj\_speed$$

It simply creates a valid ratio between all three variables. Now keep in mind, this is a prediction based off of one 2d vector. Hence, it does not consider that randomness of the wander behaviour as it is… unpredictable.

The second consideration is the angle between the blue line and the green line. The blue line is the current shooting direction (if the attacker was to shoot now, what direction would it head?). The green arrow points to the predicted end of the look_ahead position. We want to consider the angle between these two lines to determine when they should shoot.

Add this code to the vector2D class to calculate angles.

```
    def angle_with(self,v2):
        return atan2(v2.y, v2.x) - atan2(self.y, self.x)
```

This returns an angle in radians, so you multiply it my 180/pi to get the degree version. (don't forget to import math.pi). This calculation will have a domain of -180deg to 180 deg.

Unfortunately, the angle between these two lines can never be 0 because of how precise they are. We could round them, but there is a better way. By giving a domain for: if the angle is in this domain, return true, we can actually implement our accuracy control here. For our accurate mode, we will have

an angled margin of error of 0.5 degrees. For our inaccurate mode, we can have a margin of error of 10-20 degrees.

```
                if angle < self.weapon.accuracy and angle >
-self.weapon.accuracy:
                    return True
```

This line of code will grab the current accuracy setting to test domains.

Lastly, we need to make sure we have control over the accuracy and speed so our should_shoot function will update accordingly. Update main on_key_press to look like this:

```
def on_key_press(symbol, modifiers):
    if symbol == KEY.P:
        world.paused = not world.paused
    elif symbol == KEY.M:
        world.agents.append(Agent(world,'target'))
    elif symbol == KEY.N:
        world.agents.append(Agent(world,'attacking'))
    elif symbol == KEY.L:
        for agent in world.agents:
            agent.show_info = not agent.show_info
    elif symbol in PROJECTILE_SPEED:
        for agent in world.agents:
            agent.weapon.proj_speed = PROJECTILE_SPEED[symbol]
    elif symbol in PROJECTILE_ACCURACY:
        for agent in world.agents:
            agent.weapon.accuracy = PROJECTILE_ACCURACY[symbol]
```

Drawing useful debug line to the screen

Along the way of this implementation, it can become overwhelmingly difficult when working with multiple vectors, combining them and what not. Hence, the best method I found for progressing was drawing the vector I was trying to calculate to the screen. Add this code to our render function in agent and it will help.

```
if self.mode == 'attacking':
                # middle = Vector2D(self.world.cx/2,self.world.cy/2)
                # Blue line for bullet path
                egi.blue_pen()
                shoot_path = self.vel.copy()
                shoot_path.normalise()
                shoot_path *= self.weapon.proj_speed
                egi.line_with_arrow(self.pos,self.pos + shoot_path,5)
                # egi.line_with_arrow(middle, middle + shoot_path,5)

                # Red line for predicted location of closest target
                if self.world.agents_of_type('target'):
                    # Necessary render from attacking agent because of
look_ahead
                    closest_target =
self.closest(self.world.agents_of_type('target'))
                    egi.red_pen()
                    predicted_pos = closest_target.vel.copy()
                    predicted_pos.normalise()
                    predicted_pos *= self.get_look_ahead(closest_target)
egi.line_with_arrow(closest_target.pos,closest_target.pos +
predicted_pos,5)
                    # Green line for line from attacking to predicted
target location
                    egi.green_pen()
                    to_predicted = closest_target.pos + predicted_pos -
self.pos
                    egi.line_with_arrow(self.pos,closest_target.pos +
predicted_pos,5)
                    # egi.line_with_arrow(middle, middle + to_predicted,5)
                    angle = self.vel.angle_with(to_predicted) * 180 / pi
                    #print(angle)
```

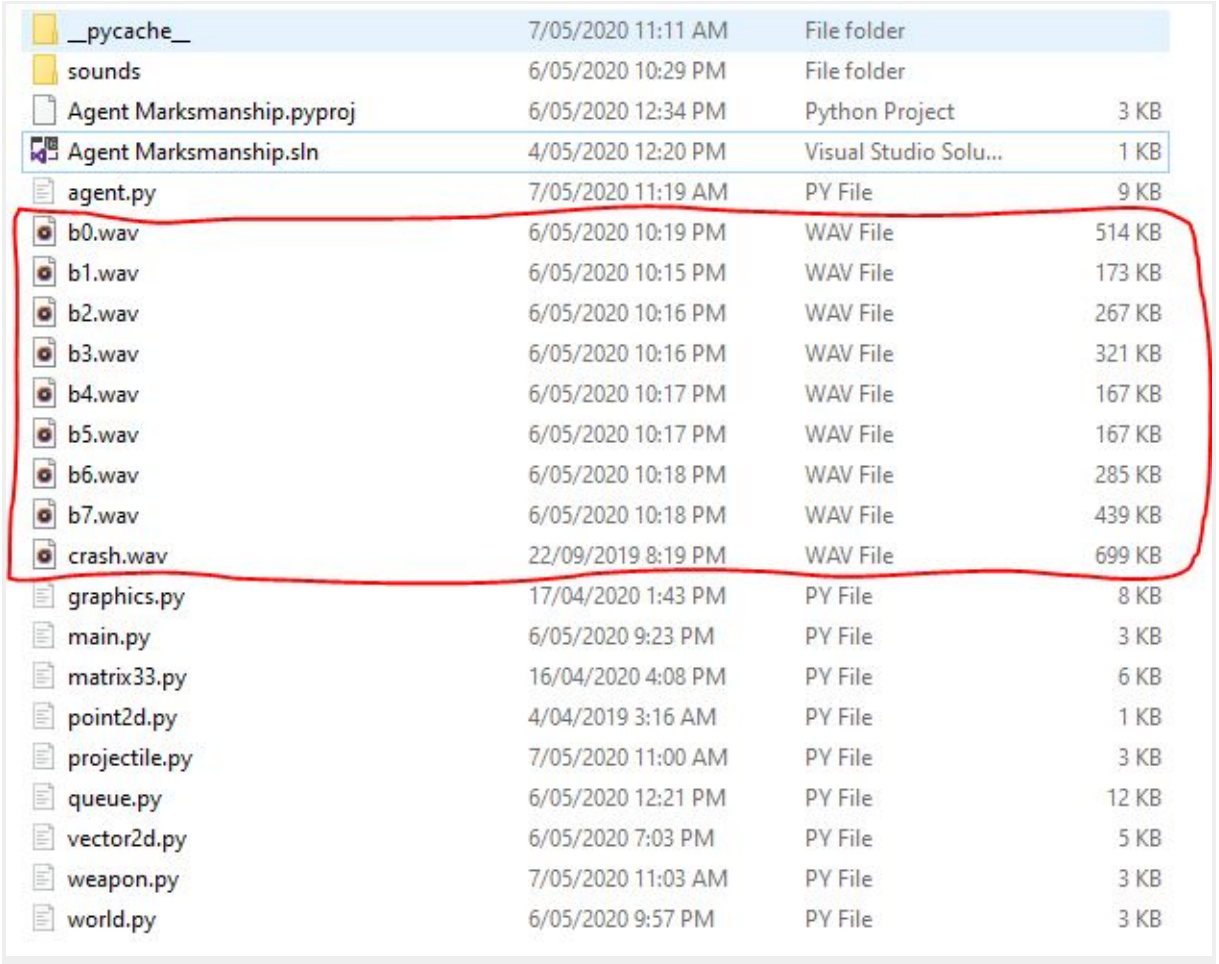
I also drew the angle in the middle of the screen and printed the angle to the console to debug the vector angle function we implemented.

Adding sound effects (just for fun)

step 1: import playsound library using pip

```
C:\Users\Matthew Cuulter>pip install playsound
Collecting playsound
  Using cached playsound-1.2.2-py2.py3-none-any.whl (6.0 kB)
Installing collected packages: playsound
Successfully installed playsound-1.2.2
```

step 2: add your sound files:



__pycache__	7/05/2020 11:11 AM	File folder	
sounds	6/05/2020 10:29 PM	File folder	
Agent Marksmanship.pyproj	6/05/2020 12:34 PM	Python Project	3 KB
Agent Marksmanship.sln	4/05/2020 12:20 PM	Visual Studio Solu...	1 KB
agent.py	7/05/2020 11:19 AM	PY File	9 KB
b0.wav	6/05/2020 10:19 PM	WAV File	514 KB
b1.wav	6/05/2020 10:15 PM	WAV File	173 KB
b2.wav	6/05/2020 10:16 PM	WAV File	267 KB
b3.wav	6/05/2020 10:16 PM	WAV File	321 KB
b4.wav	6/05/2020 10:17 PM	WAV File	167 KB
b5.wav	6/05/2020 10:17 PM	WAV File	167 KB
b6.wav	6/05/2020 10:18 PM	WAV File	285 KB
b7.wav	6/05/2020 10:18 PM	WAV File	439 KB
crash.wav	22/09/2019 8:19 PM	WAV File	699 KB
graphics.py	17/04/2020 1:43 PM	PY File	8 KB
main.py	6/05/2020 9:23 PM	PY File	3 KB
matrix33.py	16/04/2020 4:08 PM	PY File	6 KB
point2d.py	4/04/2019 3:16 AM	PY File	1 KB
projectile.py	7/05/2020 11:00 AM	PY File	3 KB
queue.py	6/05/2020 12:21 PM	PY File	12 KB
vector2d.py	6/05/2020 7:03 PM	PY File	5 KB
weapon.py	7/05/2020 11:03 AM	PY File	3 KB
world.py	6/05/2020 9:57 PM	PY File	3 KB

step 3: add header to .py file

```
from playsound import playsound
```

add this to projectile and weapon

step 4: call sound

Sound when an agent is hit

```
if agent_hit:
            self.world.agents.remove(agent_hit)
            playsound('crash.wav',block=False)
            self.recycle()
```

Sound when shooting (gets random shooting sound)

```
   def shoot(self):
        '''Moves obj from queue into list'''
        # Shoot if queue still has obj to move
        if not self.projectiles_queue.empty():
            # Get obj at front of queue
            proj = self.projectiles_queue.get()
            # Prepare for shooting
            proj.calculate()
            # Put obj in list so it is updated
            self.projectiles.append(proj)
        sound = randrange(0,7)
        playsound('b%s.wav' % sound,block = False)
```

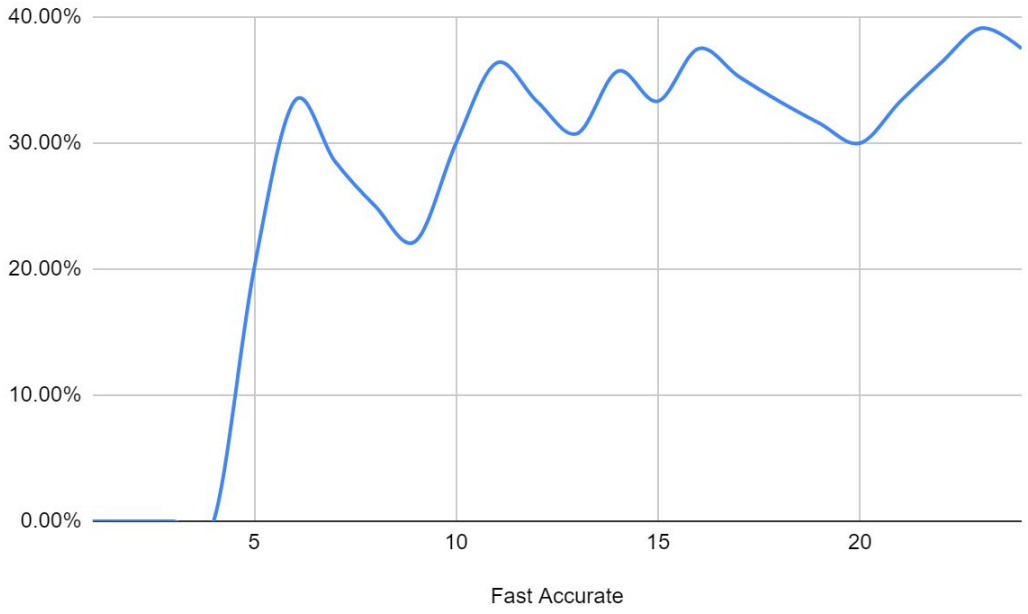
What we found out:

This implementation works really well. It requires a lot of fine tuning in the formula for look_ahead. I believe the look_ahead formula I have provided is the best possible option for considering straight lines.

The real world result is reflective of what we predicted. As the distance expands and the projectile speed slows down, the length of look_ahead increases.

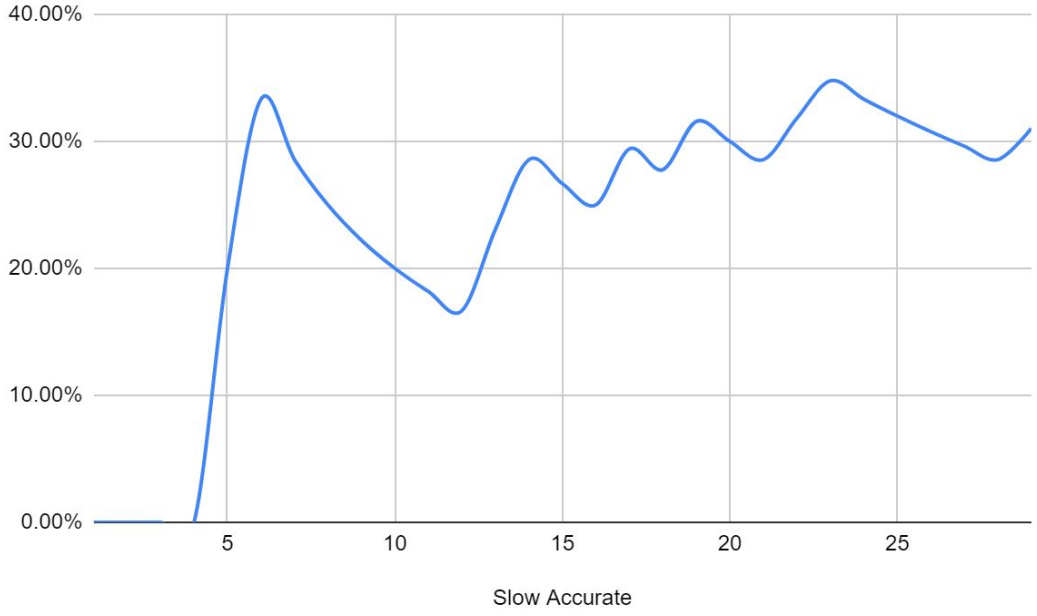
I will run a few tests to measure the overall accuracy of the accurate mode. We will set the max size of the projectile to 1 so they can only shoot one bullet at a time. The game will initialise with 50 wandering targets and 1 attacker. We will count the total amount fired and the total shots hit and then gather a percentage of accuracy based on these two values.

Fast Accurate



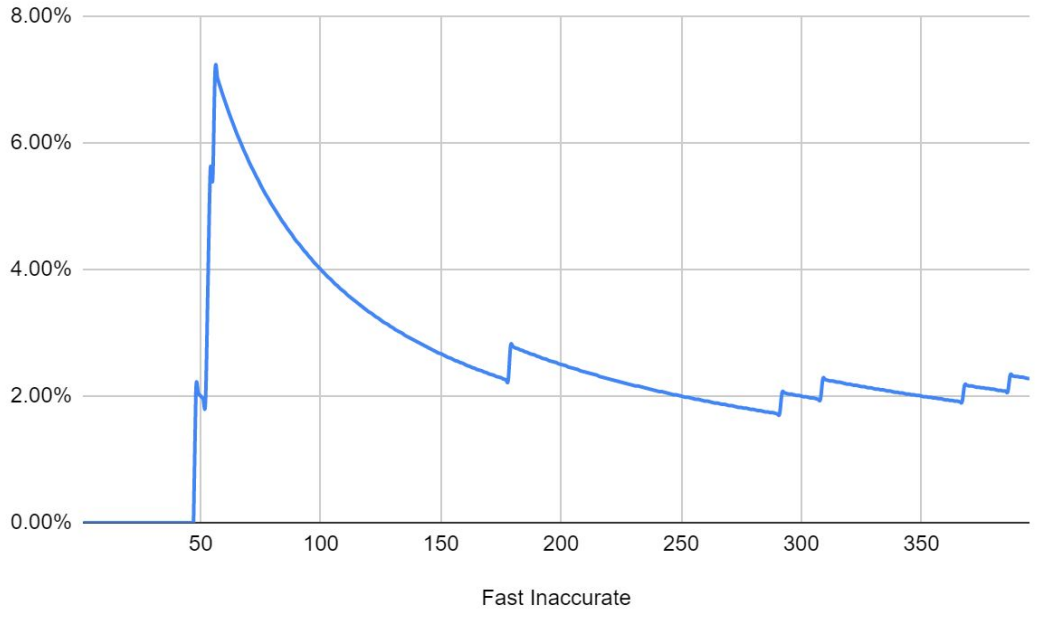
Final accuracy: 37.5%

Slow Accurate



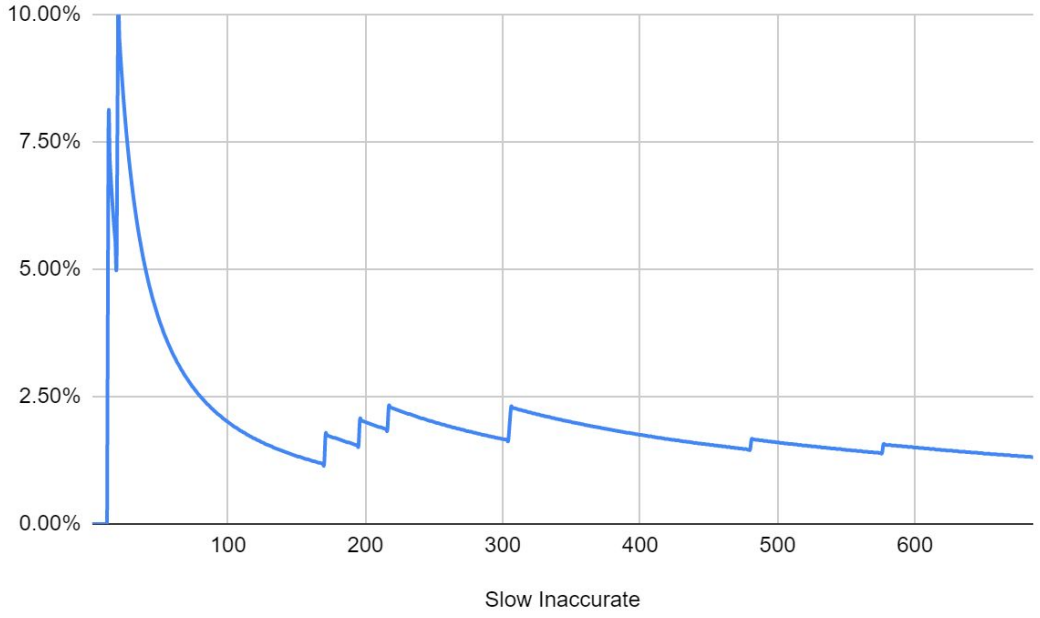
Final accuracy: 31.03%

Fast Inaccurate



Final accuracy: 2.28%

Slow Inaccurate



Final accuracy: 1.31%

In summary, the accurate mode works much more accurately than the inaccurate mode as expected. The faster the projectile is, the more accurate the shot is which is evident in both the accurate and inaccurate mode with differences of 6.47% and 0.97% respectively. However, this was also expected. THis is because when we implemented the design, we discussed how we cannot predict the randomness of the wander function, hence we predict the intersect point in assumption if it were to travel in a perfect straight line.

To explain why the accurate mode is not at 100%, the above reason is also valid. However, since the projectiles do not wrap around the screen like the agents do, occasionally it will shoot and miss because the agent has teleported to the other side of the screen.

Link to data is here