Spike: 15
Title: Agent Marksmanship

Author: Luke Valentino, 103024456

Goals / deliverables:
The goal of this spike is to create an agent targeting simulation with:
- An attacking agent
- A moving target agent
- A selection of weapons that can fire projectiles with different properties.

Be able to demonstrate that the attacking agent can hit the target with different weapon properties:
- Fast moving accurate projectile (Rifle)
- Slow Moving accurate projectile (Rocket)
- Fast Moving low accuracy projectile (Handgun)
- Slow moving low accuracy projectile (Hand Grenade)

Technologies, Tools, and Resources used:
List of information needed by someone trying to reproduce this work.
- Swinburne Lecture Materials
- Swinburne Maths and Physics PDF
- Docs.python.org

Tasks undertaken:
The key tasks undertaken in this spike were as follows:
- Copy over base code from another task.
- Create an attacking agent subclass:

```
class AttackingAgent(Agent):
    def __init__(self, world=None, scale=30.0, mass=1.0):
        super(AttackingAgent, self).__init__(world, scale, mass, mode='stationary')
        self.color = 'RED'
        self.pos = Vector2D(100,250)

    WEAPON_MODES = {
    KEY.F1: 'rifle',
    KEY.F2: 'rocket',
    KEY.F3: 'pistol',
    KEY.F4: 'grenade',
    }

    def calculate(self, delta):
        mode = self.mode
        if mode == 'stationary':
            force = Vector2D()
        else:
            force = super(AttackingAgent, self).calculate(delta)
        self.force = force
        return force
```

- Update the targetAgent subclass from the imported code to render itself as a circle. You will need to override render for this.

```
-   egi.circle(self.pos, self.bRadius)
```

- Add a mode to agent_modes called stationary.

```
-   KEY._1: 'stationary',
```

- Next we want to create our weapons. To do this we need to create a base weapons class and then use this to create subclasses.

```
-   class Weapon:
-       def __init__(self, owner):
-           self.owner = owner
-
-       def fire(self, target_pos):
-           raise NotImplementedError("This method should be overridden by
    subclasses")
-
```

    - Here are our subclasses.

```
-   class Rifle(Weapon):
-       def fire(self, target_pos):
-           return Projectile(self.owner.pos, target_pos, speed=500,
    accuracy=0.99)
-
-   class Rocket(Weapon):
-       def fire(self, target_pos):
-           return Projectile(self.owner.pos, target_pos, speed=200,
    accuracy=0.80)
-
-   class Pistol(Weapon):
-       def fire(self, target_pos):
-           return Projectile(self.owner.pos, target_pos, speed=450,
    accuracy=0.25)
-
-   class Grenade(Weapon):
-       def fire(self, target_pos):
-           return Projectile(self.owner.pos, target_pos, speed=130,
    accuracy=0.1)
-
```

- Now that we have created our weapons, we need to create our projectile that will fire from the weapons.

```
- class Projectile:
-     def __init__(self, start_pos, target, speed, accuracy):
-         self.pos = start_pos.copy()
-         self.target = target
-         self.speed = speed
-         self.accuracy = accuracy
-         self.vel = self.calculate_velocity(start_pos, target, speed,
  accuracy)
-
-     def calculate_velocity(self, start_pos, target, speed, accuracy):
-         target_pos = self.predict_target(start_pos, target)
-         desired_velocity = (target_pos - start_pos).normalise() * speed
-         if accuracy < 1.0:
-             # apply inaccuracy
-             max_deviation = (1 - accuracy) * radians(15)  # Max deviation
-             deviation_angle = uniform(-max_deviation, max_deviation)
-             cos_angle = cos(deviation_angle)
-             sin_angle = sin(deviation_angle)
-             # Rotate the desired_velocity by the deviation_angle
-             rotated_velocity = Vector2D(
-                 desired_velocity.x * cos_angle - desired_velocity.y *
  sin_angle,
-                 desired_velocity.x * sin_angle + desired_velocity.y *
  cos_angle
-             )
-             return rotated_velocity.normalise() * speed
-         return desired_velocity
-
-     def predict_target(self, start_pos, target):
-         to_target = target.pos - start_pos
-         target_speed = target.vel.length()
-         closing_speed = self.speed - target_speed
-         if closing_speed != 0:
-             time_to_target = to_target.length() / closing_speed
-             return target.pos + target.vel * time_to_target
-         else:
-             return target.pos
-
-     def update(self, delta):
-         self.pos += self.vel * delta
-
-     def render(self):
-         egi.set_pen_color(name='WHITE')
-         egi.circle(self.pos, 2)
-
```

  - This projectile class contains two important functions. The first is calculate_velocity. This uses the attributes of the weapon to determine the speed and direction of the projectile.
  - The other important method is predict_target. This method is responsible for making sure the bullet hits a moving target.
- Now we can update our attacking agent class to be able to switch and shoot the weapons.

```
def select_weapon(self, weapon_mode):
        if weapon_mode == 'rifle':
            self.current_weapon = Rifle(self)
        elif weapon_mode == 'rocket':
            self.current_weapon = Rocket(self)
        elif weapon_mode == 'pistol':
            self.current_weapon = Pistol(self)
        elif weapon_mode == 'grenade':
            self.current_weapon = Grenade(self)

    def fire_weapon(self, target_pos):
        projectile = self.current_weapon.fire(target_pos)
        self.projectiles.append(projectile)

    def update(self, delta):
        # use inherited update + addon
        super().update(delta)
        for projectile in self.projectiles:
            projectile.update(delta)

    def render(self):
        # use inherited render + addon
        super().render()
        for projectile in self.projectiles:
            projectile.render()
```

- We also need to make sure if can store its own weapon and its own list of projectiles.

```
-           self.current_weapon = Rifle(self)
-           self.projectiles = []
```

- Now that our agent can use a weapon to shoot projectiles at a target, we need to update our target to detect when it is hit.
  - First let's add some new methods.

```
-   def update(self, delta):
-           super().update(delta)
```

```
-           self.check_collision()
-           if self.hit_timer > 0:
-               self.hit_timer -= delta
-               if self.hit_timer <= 0:
-                   self.hit_timer = 0  # stops the timer from being negative
-
-       def check_collision(self):
-           for projectile in self.world.attackingAgent.projectiles:
-               if (self.pos - projectile.pos).length() < self.radius:
-                   self.handle_hit(projectile)
-
-       def handle_hit(self, projectile):
-           # remove the projectile
-           self.world.attackingAgent.projectiles.remove(projectile)
-           print("Hit detected!")
-           # set hit timer
-           self.hit_timer = self.hit_duration
```

- Then let's update the render method to show when we are hit.

```
        current_color = self.hit_color if self.hit_timer > 0 else self.color

        egi.set_pen_color(name=current_color)
        egi.circle(self.pos, self.bRadius)
```

What we found out:

In this spike I successfully implemented an agent marksmanship simulation where an attacking agent can shoot at a moving target using different weapons.

Controls:

Key F1 = Rifle
Key F2 = Rocket
Key F3 = Pistol
Key F4 = Grenade
Key SPACE = Shoot Projectile

Key Outcomes:

AttackingAgent: Created an AttackingAgent that can cycle through multiple weapons and shoot the target, even if it is moving.

TargetAgent: Created a TargetAgent that has hit detection for projectiles.

Weapons: Created a modular weapons system that allows for easy addition of new weapons.

This spike relates to multiple ULO's:

ULO 1: Discuss and implement software development techniques to support the creation of AI behaviour in Games.
ULO 2: Understand and utilise a variety of graph and path planning techniques.
ULO 3: Create realistic movement for agents using steering force models.
ULO 4: Create agents that are capable of planning actions in order to achieve goals.
ULO 5: Combine AI techniques to create more advanced game AI.