Spike: 16
Title: Soldier on Patrol

Author: Luke Valentino, 103024456

Goals / deliverables:
The goal of this spike is to create a layered state-machine with high-level modes of behaviour and lower-level modes of behaviour to simulate a soldier on patrol. The simulation must show:

- High level patrol and attack modes
- The patrol mode must use a FSM to control low-level states so that the agent will visit a number of patrol-path way points.
- The attack mode must use a FSM to control low-level fighting states.

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 first step of this task is to copy over our code from a previous spike to use as a base.

- Using the code from the previous spike, give the AttackingAgent a path and the ability to randomise a path.

```
-       def __init__(self, world=None, scale=30.0, mass=1.0):
-           super(AttackingAgent, self).__init__(world, scale, mass,
    mode='patrol')
-           self.color = 'RED'
-           self.pos = Vector2D(450, 250)
-           self.current_weapon = Rifle(self)
-           self.projectiles = []
-           self.randomise_path()
```

- Create base state machine classes. These classes are going to be use to manage our agents states.

```
class State:
    def start(self, agent):
        pass

    def running(self, agent, delta):
        pass
```

```
class StateMachine:
    def __init__(self, agent):
        self.agent = agent
        self.current_state = None

    def change_state(self, new_state):
        self.current_state = new_state
        if self.current_state:  # Ensure the new state is not None
            self.current_state.start(self.agent)

    def update(self, delta):
        if self.current_state:
            self.current_state.running(self.agent, delta)
```

- When this is complete, we can begin to make our specific states.
    - First let's create our PatrolState and its low level states. The role of the high level patrol state is to determine the agent's proximity to waypoints and enemies. If the agent is near an enemy, it will change to the AttackingState. If the agent is near a waypoint use one of its low level states, ArriveState. The purpose of arrive is to increment the agent's path and then change to SeekState. SeekState then continues to move the agent towards the next waypoint.

```
class SeekState(State):
    def start(self, agent):
        print("Entering Seek State")

    def running(self, agent, delta):
        print("Running Seek State")
        agent.force = agent.seek_waypoint()

class ArriveState(State):
    def start(self, agent):
        print("Entering Arrive State")
        agent.path.inc_current_pt()  # Move to the next waypoint
        if agent.path.is_finished():
            return Vector2D()

    def running(self, agent, delta):
        # Directly move to the next waypoint after arriving
        agent.patrol_fsm.change_state(SeekState())

class PatrolState(State):
    def start(self, agent):
        print("Entering Patrol State")
```

```
        agent.patrol_fsm = StateMachine(agent)
        agent.patrol_fsm.change_state(SeekState())

    def running(self, agent, delta):
        print("Running Patrol State")
        agent.patrol_fsm.update(delta)
        if agent.check_waypoint_distance():
            print("Agent Reached Waypoint")
            agent.patrol_fsm.change_state(ArriveState())
        if agent.detect_enemy():
            print("Enemy Detected")
            from attackingState import AttackingState  # Deferred import
            agent.fsm.change_state(AttackingState())
```

- Now we can implement our AttackingState and its low level states. The purpose of the attacking state is to determine if the agent is near an enemy, if he is then we switch to ShootingState, and if he isn't, then we switch back to PatrolState. ShootingState is used to fire the weapon at a detected enemy, when it fires the weapon it will then switch to ReloadingState. ReloadingState will delay any actions for 2 seconds, then switch back to AttackingState.

```
-
-   class ShootingState(State):
-       def __init__(self, enemy):
-           self.enemy = enemy
-
-       def start(self, agent):
-           print("Entering Shooting State")
-           pass
-
-       def running(self, agent, delta):
-           print("Entering shooting running State")
-           agent.fire_weapon(self.enemy)
-           agent.attack_fsm.change_state(ReloadingState())
-
-   class ReloadingState(State):
-       def start(self, agent):
-           self.start_time = time.time()
-
-       def running(self, agent, delta):
-           current_time = time.time()
-           if current_time - self.start_time >= 2.0:  # 2 second delay
-               agent.attack_fsm.change_state(AttackingState())
-
-   class AttackingState(State):
```

```
-       def start(self, agent):
-           print("Entering Attack State")
-           agent.attack_fsm = StateMachine(agent)
-           agent.force = Vector2D()  # Stop the agent
-           agent.vel = Vector2D()
-           self.enemy = agent.detect_enemy()
-           if self.enemy:
-               agent.attack_fsm.change_state(ShootingState(self.enemy))
-           else:
-               from patrolState import PatrolState
-               agent.fsm.change_state(PatrolState())
-
-       def running(self, agent, delta):
-           agent.attack_fsm.update(delta)
-           if not agent.detect_enemy():
-               from patrolState import PatrolState
-               agent.fsm.change_state(PatrolState())
```

- Now that we have our States, we should make sure we define all of the methods used in those states.
    - in AttackingAgent , lets define detect_enemy:

```
-       def detect_enemy(self):
-           for agent in self.world.agents:
-               if agent is not self:  # Don't check against itself
-                   distance = self.pos.distance(agent.pos)
-                   if distance < self.detection_radius:
-                       return agent  # Return the detected enemy
-           return None
```

    - Then get_next_waypoint:

```
    def get_next_waypoint(self):
        self.path.inc_current_pt()
```

    - Then check_waypoint_distance:

```
-       def check_waypoint_distance(self):
-           current_waypoint = self.path.current_pt()
-           if current_waypoint and self.pos.distanceSq(current_waypoint) <
    self.waypoint_threshold ** 2:
-               return True
-           return False
```

    - Then seek_waypoint:

```
    def seek_waypoint(self):
        current_waypoint = self.path.current_pt()
        return self.seek(current_waypoint)
```

- Let’s update some of our existing functions now:
    - Let’s fix the update method in AttackingAgent

```
-   def update(self, delta):
-           self.fsm.update(delta)
-
-           self.force.truncate(self.max_force)
-
-           self.accel = self.force / self.mass
-
-           self.vel += self.accel * delta
-           self.vel.truncate(self.max_speed)
-
-           self.pos += self.vel * delta
-
-           if self.vel.lengthSq() > 0.00000001:
-               self.heading = self.vel.get_normalised()
-               self.side = self.heading.perp()
-
-           self.world.wrap_around(self.pos)
-
-           for projectile in self.projectiles:
-               projectile.update(delta)
```

- We need to make sure we create a state machine in AttackingAgent:

```
-           self.fsm = StateMachine(self)
-           self.fsm.change_state(PatrolState())
```

What we found out:
In this spike I successfully implemented a soldier on patrol style simulation using a finite state machine to control the agent’s behaviour.

Controls:

Key F1 = Rifle
Key F2 = Rocket
Key F3 = Pistol
Key F4 = Grenade
Key T = Add Enemy

This spike relates to multiple ULO's:

ULO 1: Discuss and implement software development techniques to support the creation of AI behaviour in games:

- By developing a layered FSM, I have been able to demonstrate an effective technique to manage complex behaviours.

ULO 2: Understand and utilise a variety of graph and path planning techniques:

- The patrols mode's waypoint system and randomization show this.

ULO 3: Create realistic movement for agents using steering force models:

- The SeekState and ArriveState combine to use steering forces to move the agent towards waypoints.

ULO 4: Create agents that are capable of planning actions in order to achieve goals:

- The high-level patrol and attack states enable the agent to switch goals based on it's environment.

ULO 5: Combine AI techniques to create more advanced game AI:

- Integrating FSMs for both patrol and attack modes, along with steering behaviours shows who I have been able to combine multiple techniques to create one comprehensive simulation.