Spike: 16
Title: Soldier on Patrol

Author: Marco Giacoppo, 104071453

Goals / deliverables:
The goal of this spike was to implement and test a simulation where an autonomous soldier agent uses a layered finite state machine (FSM) to patrol a defined area and engage enemies when detected. The system was expected to support:

- A patrolling soldier agent with multiple waypoints.
- Reactive switching between high-level “PATROL” and “ATTACK” modes
- A shooting and reloading mechanic governed by an internal sub-FSM
- The ability to spawn enemy agents using keyboard input
- Health, ammo, and state management with GUI debugging option.

Technologies, Tools, and Resources used:

- Python 3.11
- Pyglet graphics engine
- ChatGPT for planning and debugging

Tasks undertaken:

This section describes the steps to reproduce this spike:

Initialize Project

- Clone or prepare the agent movement starter framework.
- Set up main.py, game.py, world.py, graphics.py, and agent-related files.

Implement Layered FSM (Finite State Machine)

1. Create SoldierAgent class with:

- High-level FSM states: PATROL, ATTACK
- Low-level FSM states: SHOOT, RELOAD

```
class HighLevelState(Enum):
    PATROL = auto()
    ATTACK = auto()

class AttackSubState(Enum):
    SHOOT = auto()
    RELOAD = auto()

class SoldierAgent(Agent):
    def __init__(self, world):
        super().__init__(world, scale=30, mass=1.0, color='BLUE')
        self.weapon = Weapon(world, self)
        self.high_state = HighLevelState.PATROL
        self.attack_state = AttackSubState.SHOOT
        self.reload_timer = 0.0
        self.shoot_cooldown = 0.0
        self.patrol_points = [
            Vector2D(100, 100),
            Vector2D(200, 700),
            Vector2D(300, 100),
            Vector2D(700, 600)
        ]
        self.current_patrol_index = 0

        self.show_debug = False
```

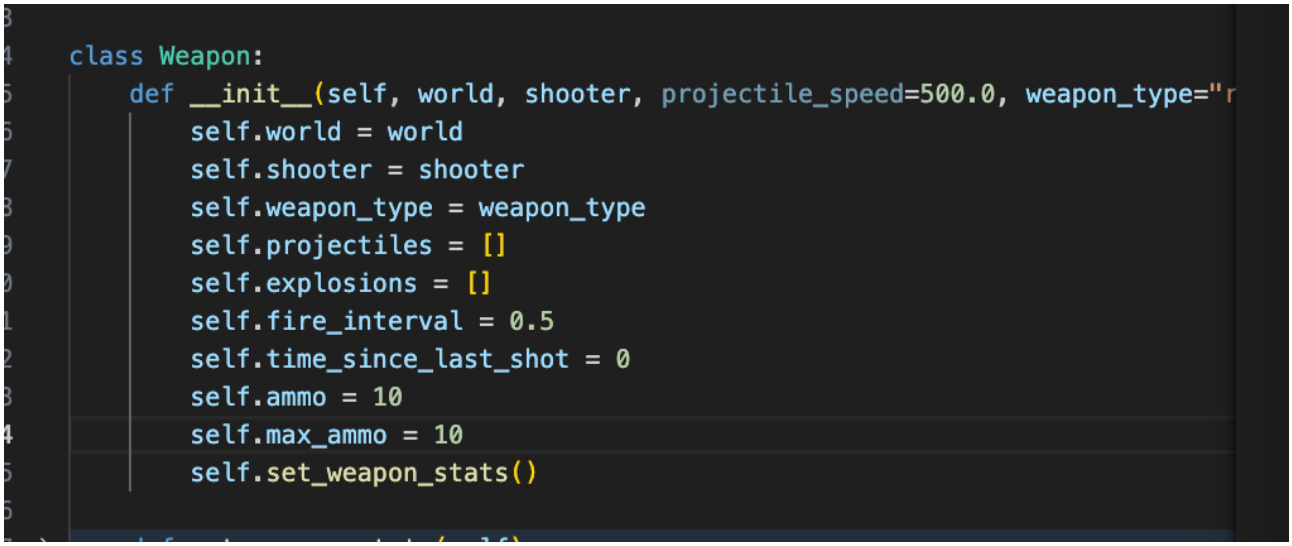
2. PATROL mode:
- Agent follows multiple waypoints using arrive() steering

3. ATTACK mode:
- Seeks the closest enemy
- If ammo > 0, enters SHOOT substate
- If ammo == 0, switches to RELOAD, waits 2 seconds, then returns to SHOOT

```
def _update_attack(self, delta, target):
    if not target or target.health <= 0:
        return

    # Reduce cooldown timer
    if self.shoot_cooldown > 0:
        self.shoot_cooldown -= delta

    if self.attack_state == AttackSubState.SHOOT:
        if self.weapon.ammo > 0:
            if self.shoot_cooldown <= 0:
                self.weapon.shoot(target)
                self.shoot_cooldown = 0.5  # wait 0.5 seconds before next
        else:
            self.attack_state = AttackSubState.RELOAD
            self.reload_timer = 2.0

    elif self.attack_state == AttackSubState.RELOAD:
        self.reload_timer -= delta
        if self.reload_timer <= 0:
            self.weapon.reload()
            self.attack_state = AttackSubState.SHOOT
```

Implement Combat and Weapon Logic

1. Implement Weapon class with:
- Ammo, max_ammo, and reload() behaviour
- Shoot() that decreases ammo and applies damage
2. Add cooldown between shots to avoid overfiring
3. Enemies (TargetAgent) move between 2 points and has health
4. When health<=0, enemy is removed (died)



```
class Weapon:
    def __init__(self, world, shooter, projectile_speed=500.0, weapon_type="r
        self.world = world
        self.shooter = shooter
        self.weapon_type = weapon_type
        self.projectiles = []
        self.explosions = []
        self.fire_interval = 0.5
        self.time_since_last_shot = 0
        self.ammo = 10
        self.max_ammo = 10
        self.set_weapon_stats()
```

Add input handling

1. P: Pause/unpause the program
2. A: Spawn a new enemy at a random position
3. D: Toggle GUI debug labels (state, ammo, enemy count)

What we found out:

- FSM Structure Enables Clean Behavior Transitions

  Using a layered FSM made it easy to separate movement (patrol) from combat logic. The high-level and sub-state separation kept code readable and expandable. Transitions occurred smoothly based on the presence of enemies and ammo status. This structure also makes future additions (like hiding, charging, or fleeing) very easy to integrate.

- GUI Labels Greatly Improved Visibility and Debugging

  Adding on-screen labels for agent state, ammo count, and number of enemies provided clear real-time feedback and was significantly more intuitive than using terminal logs. Toggling labels with the keyboard also helped reduce screen clutter during testing. It also highlighted bugs such as ammo misreporting, which might have gone unnoticed with print statements alone.

- Reloading Mechanics Require Careful Timing

  Initially, the agent overfired due to missing a cooldown mechanic — bullets were fired on every frame (60 times per second). This caused ammo to drain instantly and skipped the reload state. Introducing a

shoot_cooldown timer and checking for sub-state transitions corrected this, producing smooth and believable attack behavior.

- Clean Architecture Avoids Circular Imports

  I encountered a circular import issue when trying to call game.spawn_enemy() from within world.py. The fix was to inject the Game instance into the World object during initialization. This pattern of dependency injection kept the system modular and prevented module re-import errors.

- GUI-Based Debugging is More Intuitive Than Console Output

  One major takeaway is how much easier it is to debug state transitions and ammo logic through on-screen elements instead of console prints. During testing, labels visually confirmed when the soldier was stuck in a state or if a reload never triggered — problems that would’ve taken longer to catch with print-based debugging.