Spike: 15
Title: Agent Marksmanship

Author: Marco Giacoppo, 104071453

Goals / deliverables:
The goal of this spike was to implement and test a simulation where an agent can successfully target and hit a moving target using various types of projectile weapons with different speed and accuracy profiles. The system was expected to support:

- A stationary shooter agent
- A moving target agent
- Four weapon types:
- Rifle (fast, accurate)
- Rocket (slow, accurate)
- Handgun (fast, inaccurate)
- Grenade (slow, inaccurate)

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:

1. Initialize Project

- Clone base agent framework provided

- Set up main.py, game.py, and graphics.py

2. Create Agent Classes

- TargetAgent: moves back and forth between two waypoints

- ShooterAgent: static shooter that faces the target

```python
class TargetAgent(Agent):
    def __init__(self, world, point_a, point_b, speed=100.0, color='YELLOW'):
        super().__init__(world, scale=30, mass=1.0, color=color)
        self.point_a = point_a
        self.point_b = point_b
        self.current_target = point_b
        self.max_speed = speed

    def calculate(self, delta):
        if (self.current_target - self.pos).length() < 10:
            self.current_target = self.point_a if self.current_target == self.point_b else self.point_b
        return self.seek(self.current_target)


class ShooterAgent(Agent):
    def __init__(self, world, pos, target, color='RED'):
        super().__init__(world, scale=30.0, mass=1.0, color=color)
        self.pos = pos.copy()
        self.vel = Vector2D()
        self.max_speed = 0  # Stationary
        self.target = target

    def calculate(self, delta):
        return Vector2D(0, 0)

    def update(self, delta):
        if self.target:
            direction = (self.target.pos - self.pos)
            if direction.lengthSq() > 0.0001:
                self.heading = direction.normalise()
                self.side = self.heading.perp()

        self.world.wrap_around(self.pos)
        self.vehicle.x = self.pos.x + self.vehicle_shape[0].x
        self.vehicle.y = self.pos.y + self.vehicle_shape[0].y
        self.vehicle.rotation = -self.heading.angle_degrees()
```

3. Implement Weapon and Projectile System

- Create Weapon class supporting multiple types via change_weapon()

```python
class Weapon:
    def __init__(self, world, shooter, projectile_speed=500.0, weapon_type="rifle"):
        (parameter) self: Self@Weapon
        self.shooter = shooter
        self.weapon_type = weapon_type
        self.projectiles = []
        self.explosions = []
        self.fire_interval = 0.5
        self.time_since_last_shot = 0
        self.set_weapon_stats()

    def set_weapon_stats(self):
        if self.weapon_type == "rifle":
            self.speed = 600
            self.inaccuracy = 0
            self.color = COLOUR_NAMES['WHITE']
        elif self.weapon_type == "rocket":
            self.speed = 250
            self.inaccuracy = 0
            self.color = COLOUR_NAMES['RED']
        elif self.weapon_type == "handgun":
            self.speed = 600
            self.inaccuracy = 0.2
            self.color = COLOUR_NAMES['AQUA']
        elif self.weapon_type == "grenade":
            self.speed = 300
            self.inaccuracy = 0.4
            self.color = COLOUR_NAMES['YELLOW']

    def change_weapon(self, new_type):
        self.weapon_type = new_type
        self.set_weapon_stats()
        window._update_label("weapon", f"Weapon: {self.weapon_type.title()}")
        print(f"Switched to: {self.weapon_type.title()}")
```

- Create Projectile class for movement and rendering

```
class Projectile:
    def __init__(self, pos, velocity, color):
        self.pos = pos.copy()
        self.velocity = velocity.copy()
        self.radius = 5
        self.color = color
        self.circle = pyglet.shapes.Circle(
            self.pos.x, self.pos.y,
            radius=self.radius,
            color=color,
            batch=window.get_batch("main")
        )

    def update(self, delta):
        self.pos += self.velocity * delta
        self.circle.x = self.pos.x
        self.circle.y = self.pos.y
```

- Add Explosion class for visual feedback

```
class Explosion:
    def __init__(self, pos, duration=0.3):
        self.pos = pos.copy()
        self.timer = duration
        self.circle = pyglet.shapes.Circle(
            pos.x, pos.y,
            radius=10,
            color=COLOUR_NAMES["RED"],
            batch=window.get_batch("main")
        )
        self.circle.opacity = 200

    def update(self, delta):
        self.timer -= delta
        self.circle.radius += 100 * delta  # expands over time
        self.circle.opacity = max(0, int(255 * (self.timer / 0.3)))  # fades out

    def is_finished(self):
        return self.timer <= 0
```

4. Add Target Prediction Logic

- Predict future position of the moving target using its velocity and projectile speed

```
def predict_target_position(self, target):
    to_target = target.pos - self.shooter.pos
    distance = to_target.length()
    t = distance / self.speed if self.speed != 0 else 0
    return target.pos + (target.vel * t)
```

5. Implement Hit Detection

- Detect collisions between projectile and target using distance()

- Show a red expanding explosion and remove the projectile

```
def update(self, delta, target):
    new_projectiles = []
    for proj in self.projectiles:
        proj.update(delta)
        if proj.pos.distance(target.pos) < (proj.radius + 15):
            print(f"💥 {self.weapon_type.title()} HIT the target!")
            self.explosions.append(Explosion(proj.pos))
            # Don't add it back to the list → it disappears immediately
        else:
            new_projectiles.append(proj)
    self.projectiles = [
        p for p in new_projectiles if 0 <= p.pos.x <= self.world.cx and 0 <= p.pos.y <= self.world.cy
    ]

    # Update explosions
    self.explosions = [e for e in self.explosions if not e.is_finished()]
    for explosion in self.explosions:
        explosion.update(delta)
```

6. Set Manual Firing via Key Press

- Modify world.py to handle SPACE key and call fire() manually

7. Add Weapon Switching

- Keys 1–4 switch between rifle, rocket, handgun, and grenade

8. Debug and Refine

- Prevent multiple explosions from one projectile

- Ensure only one target is visible by removing duplicate agent creation

What we found out:

The spike successfully demonstrated how different projectile configurations affect a shooter agent's effectiveness at hitting a moving target in a 2D simulation environment. Several key insights emerged from the implementation and testing process:

1. Weapon Characteristics Significantly Impact Hit Rate
   Weapons with high speed and low inaccuracy (like the rifle and rocket) consistently hit the target when fired under predicted trajectories. Conversely, weapons with lower speed and higher inaccuracy (like the handgun and grenade) often missed, especially when the target was moving at higher speeds. This clearly highlights the importance of

tuning both speed and precision when designing agent-based weapon systems.

2. Target Prediction Was Essential for Moving Targets
Simply firing at the current position of the moving target caused frequent misses, especially with slower projectiles. Implementing a predictive aiming system that estimated the interception point using the target's velocity and the projectile's speed greatly improved hit accuracy. This reinforced the value of velocity-aware targeting in agent marksmanship AI.

3. Manual Firing Helped Isolate Weapon Behavior
Switching from automatic to manual fire via spacebar key press allowed clearer observation of each shot's behavior. This made it easier to debug the projectile path, confirm prediction accuracy, and distinguish between accurate and inaccurate projectiles.

4. Explosions as Visual Feedback Enhanced User Understanding
Introducing an expanding, fading explosion effect at the hit location offered immediate and intuitive confirmation that a shot had landed. This made the simulation more interactive and easier to validate visually, particularly during rapid iterations of testing.

5. Timing and Collision Handling Are Crucial
Without projectile removal upon impact, the system initially triggered multiple explosions for a single hit. This revealed the importance of managing object lifecycle (removing projectiles after collision) to avoid redundant visual effects or performance issues.