Spike: Task 16

Title: Goal-Oriented Action Planning

Author: Sam Huffer, 101633177

Goals / Deliverables

- Basic deliverable: a simulation for agent control demonstrating the effectiveness of goal-oriented action planning in considering long-term outcomes (e.g. side-effects, time delays) of actions, and can plan and act intelligently.

Technologies, Tools, and Resources Used

- SublimeText (for editing, executing and testing the code)
- Learning materials on Canvas (for instructions and sample code)

Tasks Undertaken

- I started by copying the project from Task 15: Soldier on Patrol into the Task 16: Goal-Oriented Action Planning folder, stripping the spike report down to what was needed for this task.
- Next, I consulted my lecture notes and the task instructions document to plan out what I wanted to do for this task, as how students implemented goal-oriented action planning was left up to us. The planned goals I put together for this task were:
    - The agent will have to contend with hunger, which will increase over time and each time it fires a shot. Each update, it must consider whether to continue its current actions, or return to the food station and eat if it would not compromise its current attack against the target. If it gets hungry enough that if it continues its current actions it would not have the stamina to return to a food station to reduce its hunger, it must return to the food station and eat. While patrolling, it should use the maximum distance it might have to cross to calculate when to return to the food station; when attacking, it should use its current distance from the food station with some padding.
    - The soldier will have two weapons at a time, with a limited number of magazines. When patrolling, it must consider if the total remaining ammo would theoretically be sufficient to kill the target. If it is not, the agent must return to an ammunition station to get new weapons. When the soldier goes to attack, if its remaining ammo would become theoretically insufficient to kill the target, it must return to the ammo station to exchange their weapons. If it runs out of ammo completely, it must return to the ammo station to replenish it. The choice of weapons should be random, and all weapons, including the ones just handed back, should be available.
    - When aiming its shot, the agent should consider whether staying with its current weapon or its secondary weapon will be more conducive or sufficient in killing the target in terms of shot impact. For example, if the target has less than full health, the soldier shouldn't use two rifle rounds to take out the target if another weapon would also kill the target in one shot.
- First, I took the code outlining what each weapon was, as well as the code for pooling projectiles, and created a weapon class that could hold the appropriate variables and be passed from world to soldier and back.
- I then created points in the world for the ammo station and the food station, and added logic for the soldier to consider when to swap to the next weapon, and when to go to the ammo

station if its current ammo would theoretically be insufficient to kill the target when encountered, assuming that all rounds hit their target (four out of five pellets in the shotgun's case), and estimating how much damage explosive projectiles would deal on average to the target. With this, I also included a shortcut key for wiping out the soldier's ammo for demonstration purposes.

- I then tried adding logic for handling explosive weapons, such that one would not be fired if the soldier was too close to the target, and that existing explosive projectiles would be avoided so as not to get caught in the blast radius. That code created issues and was ineffective, so instead, due to time constraints and other homework needing to be attended to, I cut that code out and instead beefed up the soldier's health as if it had tough armour or something, so that it would not die to its own explosions so readily during demonstrations.
- Next, I added code for incrementing the soldier's hunger every second, and then for checking if the soldier will starve if they don't go and get food now and consequently for sending them to the food station to satiate its hunger. The code for it moving to the food station works the same as the code for moving to the ammo station. I didn't add a shortcut key for incrementing hunger as with erasing ammo, as it will only take thirty-something seconds for the soldier to be hungry enough that it needs to eat.
- At this point, and having attended the lecture for week 10, I looked at the instructions for this task on Canvas in the instructions PDF again. I became unsure if the goals I planned for the task would meet the task's requirements, so I submitted a draft of this spike report to get feedback from Tien.
- When I got the feedback, it seemed like I was mostly on track, but just needed to tweak the logic for considering if to replenish ammo while attacking so that the soldier went to exchange its weapons if it determined its current ammo would be insufficient to kill the target, rather than doing so only once it ran out of ammo. I made the changes required of choose_weapon(), tweaked the print to screen code slightly to show health as being out of fifty, rather than just N amount of health, and reordered some logic for the shooter so that hunger would always be checked before weapons and ammo, as running out of ammo wouldn't kill the soldier, but starving would.

Instructions for Operating the Code

- A: toggle the display of agents' collision avoidance range.
- C: set the soldier's ammo to 0.
- I: toggle the display of agents' force, velocity and net desired change in position.
- P: pause or un-pause the game.
- S: re-spawn a dead soldier.
- T: re-spawn a dead target.
- W: scroll through soldier weapons.
- Escape: exit the simulation.

Code Snippets

```
class Weapon (object):
    def __init__(self, world=None, owner=None, name='', cooldown=1, effective_range=100, damage=25, reload_time=1,
        self.world = world
        self.owner = owner
        self.name = name
        self.cooldown = cooldown
        self.effective_range = effective_range
        self.speed = speed
        self.damage = damage
        self.damage_factor = damage_factor
        self.reload_time = reload_time
        self.magazine_size = magazine_size
        self.rounds_left_in_magazine = 0
        self.magazines_left = magazines
        self.accuracy_modifier = accuracy_modifier
        self.stamina_drain = stamina_drain
        self.explosive = explosive
        self.explosion_radius = explosion_radius

        self.projectile_pool = []
        i = 0

        while i < 200:
            self.projectile_pool.append(Projectile(world=self.world, weapon=self))
            i += 1
```

Figure 1: the weapon class.

```
def update_shooter(self, delta):
    self.see_target = False
    self.update_fov(self.world.obstacles, self.world.agents)

    if not self.hungry() and self.choose_weapon():
        if self.see_target or (self.world.target is not None and self.distance(self.world.target.pos) < self.hunt_dist + self.world.target.radius):
            self.movement_mode = 'Attack'
        else:
            self.movement_mode = 'Patrol'

        if self.weapons[0].rounds_left_in_magazine == 0 and (datetime.now() - self.last_shot).total_seconds() <= self.weapons[0].reload_time:
            self.combat_mode = 'Reloading'
        else:
            if self.weapons[0].rounds_left_in_magazine == 0:
                self.weapons[0].rounds_left_in_magazine += self.weapons[0].magazine_size
                self.weapons[0].magazines_left -= 1

            if (datetime.now() - self.last_shot).total_seconds() <= self.weapons[0].cooldown:
                self.combat_mode = 'Weapon is Loading Next Round'
            elif self.movement_mode == 'Attack':
                self.combat_mode = 'Aiming'
            elif self.movement_mode == 'Patrol':
                self.combat_mode = 'Ready'

        if self.movement_mode == 'Attack' and self.combat_mode == 'Aiming':
            if len(self.weapons[0].projectile_pool) == 0:
                self.combat_mode = 'No Projectiles Pooled'
            else:
                target = self.aim_shot(self.world.target)

                if target is not None:
                    self.combat_mode = 'Shooting'
                    self.last_shot = datetime.now()
                    self.weapons[0].rounds_left_in_magazine -= 1
                    self.shoot(target)

    elif self.movement_mode == 'Get Food' and self.distance(self.world.food_station) < self.world.station_size:
        self.hunger = 0
        self.movement_mode = 'Patrol'
    elif self.movement_mode == 'Exchange Weapons' and self.distance(self.world.ammo_station) < self.world.station_size:
        self.world.change_weapons(self)
        self.movement_mode = 'Patrol'

    self.move(delta)
```

Figure 2: the update_shooter() method, where the soldier figures out what it's going to do. It updates its field of view, checks it's not hungry or needing to swap its weapons, then sorts out what its going to do. If it is hungry or needs to change its weapons, it goes to the appropriate refilling station instead.

```
def calculate_shooter(self, delta):
    if self.movement_mode == 'Get Food':
        return self.arrive(self.world.food_station, 'slow')
    elif self.movement_mode == 'Exchange Weapons':
        return self.arrive(self.world.ammo_station, 'slow')
    elif self.movement_mode == 'Patrol':
        return self.follow_path()
    elif self.movement_mode == 'Attack':
        if self.world.obstacles_enabled:
            return self.hunt(self.world.target, delta)
        else:
            return self.seek(self.world.target.pos)

    return Vector2D(0,0)
```

Figure 3: the calculate shooter method, where, the soldier having decided how it's going to move, it seeks the result of the relevant movement method.

```
def hungry(self):
    if self.hunger >= 50:
        self.world.destroy_agent(self)

    if self.last_hunger_ping == None:
        self.last_hunger_ping = datetime.now()

    if (datetime.now() - self.last_hunger_ping).total_seconds() >= 1:
        self.last_hunger_ping = datetime.now()
        self.hunger += 1

    if self.movement_mode == 'Get Food':
        return True
    elif self.movement_mode == 'Patrol':
        max_distance = (Vector2D(0,0) - Vector2D(self.world.cx, self.world.cy)).length()
        max_time = max_distance / self.max_speed

        if (self.hunger + max_time) * 1.1 >= 50:
            self.movement_mode = 'Get Food'
            return True

    elif self.movement_mode == 'Attack':
        current_distance = self.distance(self.world.food_station)
        current_time = current_distance / self.max_speed

        if (self.hunger + current_time) * 1.1 >= 50:
            self.movement_mode = 'Get Food'
            return True

    return False
```

Figure 4: the method that checks if the soldier needs to go back to the food station to sate its hunger before it starves.

```
def choose_weapon(self):
    if self.movement_mode == 'Exchange Weapons':
        return False

    if len(self.weapons) <= 1:
        self.movement_mode = 'Exchange Weapons'
        return False

    weapon_0 = self.weapons[0]
    weapon_1 = self.weapons[1]

    weapon_0_ammo = weapon_0.rounds_left_in_magazine + weapon_0.magazine_size * weapon_0.magazines_left
    weapon_1_ammo = weapon_1.rounds_left_in_magazine + weapon_1.magazine_size * weapon_1.magazines_left

    weapon_0_avg_dmg = weapon_0.damage * weapon_0.damage_factor
    weapon_1_avg_dmg = weapon_1.damage * weapon_1.damage_factor

    if self.world.target is not None:
        target_health = self.world.target.health

    # check if out of ammo or if patrolling and have insufficient ammo to kill the target
    if weapon_0_ammo <= 0 and weapon_1_ammo <= 0:
        self.movement_mode = 'Exchange Weapons'
        return False
    elif self.world.target is not None and (self.movement_mode == 'Attack' or self.movement_mode == 'Patrol'):
        if weapon_0_avg_dmg * weapon_0_ammo + weapon_1_avg_dmg * weapon_1_ammo < target_health:
            self.movement_mode = 'Exchange Weapons'
            return False
    elif self.world.target == None and self.movement_mode == 'Patrol':
        if weapon_0_avg_dmg * weapon_0_ammo + weapon_1_avg_dmg * weapon_1_ammo < self.start_health:
            self.movement_mode = 'Exchange Weapons'
            return False

    # check if only current weapon is out of ammo
    if weapon_0_ammo <= 0 and weapon_1_ammo > 0:
        self.next_weapon()
    # check if both weapons have ammo, if both weapons' probable damage dealt (accounting for explosive splash damage and multiple shotgun
    # pellets vs fixed damage rifle and hand gun bullets) would be sufficient to kill the target, and the next weapon deals less damage
    elif self.world.target is not None and weapon_0_ammo > 0 < weapon_1_ammo and weapon_0_avg_dmg > weapon_1_avg_dmg > target_health:
        self.next_weapon()

    return True
```

Figure 5: the method that checks if the soldier needs to go and change it's low-ammo weapons for new weapons, or if it has sufficient ammo, if it should swap its current weapon for the next one.

In-Simulation Screenshots

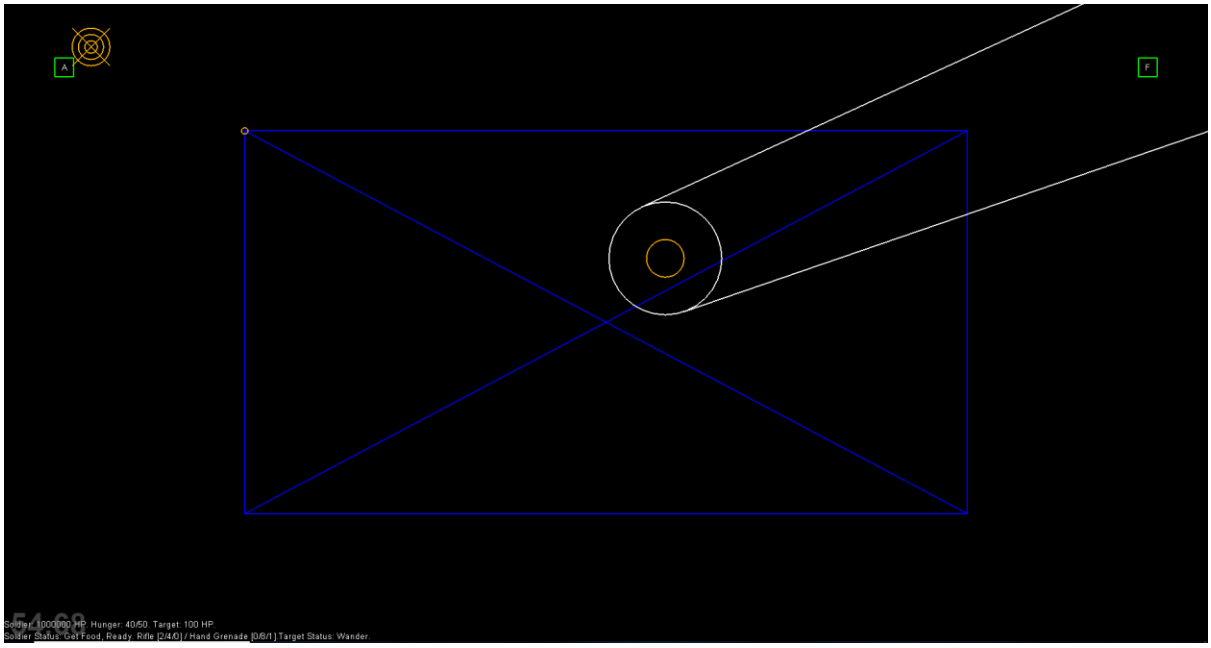

Figure 6: soldier agent is hungry and needs to eat.

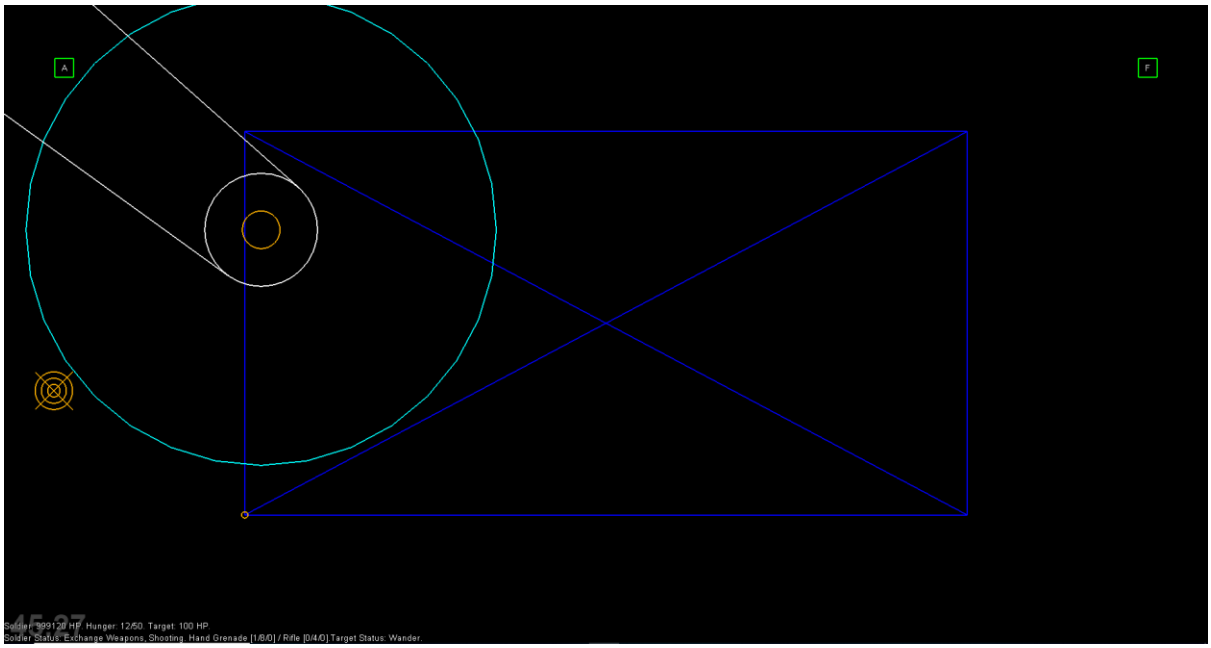

Figure 7: soldier agent is out of ammo and needs to swap its weapons.

What I Found Out

- The original idea I had of the soldier thinking ahead about its stocks of ammo and hunger was fairly sound, but my original implementation plan just needed to be tweaked to have the soldier think ahead about ammo to when it went to attack the target, rather than having it merely react to having no ammo left. Fixing that up was simple enough.
- Explosion avoidance was weirdly fiddly for some reason when I tried it, which I didn't think it would be considering that I'd already managed to get agents avoiding obstacles and colliding with each other, and (at least) trying to not run into walls and to dodge projectiles. But

apparently staying out of the range of an explosion was trickier than those, enough that I decided to leave it out and finish this task so I could instead attend to other homework.