Spike: Spike_17
Title: Spike Extension Report

Author: Tran Duc Anh Dang - 103995439

Task:
Lab 3 - Tic Tac Toe
Extension: Tic Tac Toe A.I Battle

- In this extension task for the Tic Tac Toe game, the existing code base was modified to include an artificial intelligence (AI) opponent that utilises a min-max algorithm for its gameplay.

The min-max algorithm is a recursive algorithm used for decision-making in game theory and AI. It simulates all possible games that might follow from the current position, allowing the AI to make the optimal move. It assumes perfect play from both players, meaning it always considers the best possible move for its opponent when deciding its move.

In our case, the algorithm has been implemented in the get_ai_move() function. This function first checks if the game has ended (a tie or a win), and if not, it calls the minmax() function. The minmax() function calculates the best move by simulating all possible moves recursively.

The AI player uses the minmax() function to decide the best move based on the current state of the game. It simulates all possible moves for the remaining empty positions on the board, predicts the outcome for each move, and then chooses the move with the maximum score.

The minmax() function assigns a score to each end state: 1 for a win, -1 for a loss, and 0 for a tie. By exploring all possible moves and their outcomes, the algorithm chooses the move with the highest value, which means the most favourable outcome for the AI.

Overall, the implementation of the min-max algorithm has significantly improved the gameplay of the AI player in this Tic Tac Toe game. The AI is now capable of choosing the best possible move, making the game more challenging and engaging.

Spike 4: Goal Oriented Behaviour and SGI
Extension: Complex

- This task involved extending a model for goal-oriented AI behaviour in a non-playable character (NPC). The model represents an AI agent that has three goals - 'Hunger', 'Energy', and 'Happiness'. To satisfy these goals, the agent can perform actions like 'get raw food', 'eat food', 'sleep', 'watch movie', 'hang out with friends', and 'exercise'. Each action impacts one or more of the agent's goals.

The object-oriented (OO) version of this model provides multiple benefits. The class-based structure allows for easier management and modification of attributes such as goals, actions, and probabilities. This allows developers and designers to tweak the AI's behaviour easily. With OOP, I can easily create multiple instances of the agent, each with different configurations, to create more diverse NPC behaviour.

However, there are potential downsides to this OO approach. While encapsulation in OOP aids in managing complexity, it might add overhead, especially if multiple NPCs are interacting in real-time, as the class operations require more processing power compared to functional programming.

The next task proposed the creation of a console-based turn-based role-playing game (RPG) simulating two NPCs in combat, each using this model to select actions. This extension highlights the potential of the model in a game environment. Each NPC, behaving as an AI agent, makes decisions based on its set of goals, adding a layer of unpredictability and realism to the combat scenario.

While this simple model does not consider the intricate strategies often present in RPG combat, it does provide a framework that could be expanded with additional goals, actions, and more sophisticated decision-making algorithms.

Overall, this task allowed the exploration of a simple model for goal-oriented behaviour in NPCs and proposed exciting extensions that could contribute to more immersive and unpredictable AI behaviour in games.

Spike 7: Tactical Analysis with PlanetWars

Extension:

- The AI operates with two main strategies - attacking enemy planets and defending its own. When attacking, it chooses the source planet with the most ships and targets enemy planets that are relatively weak and close.

```
src = max(gameinfo.my_planets.values(), key=lambda p: p.num_ships)
...
dest = max(less_ships, default=min(gameinfo.not_my_planets.values(),
key=lambda p: p.distance_to(src)), key=lambda p: (2 * p.num_ships + p.growth_rate) /
p.distance_to(src))
...
gameinfo.planet_order(src, dest, int(src.num_ships * 0.75))
```

In the defense phase, the AI checks for incoming enemy fleets and sends reinforcements from other planets if the targeted planet doesn't have enough ships to fend off the attack.

```
if incoming_fleets:
    total_incoming_ships = sum(fleet.num_ships for fleet in incoming_fleets)
    if planet.num_ships < total_incoming_ships:
        nearest_planet = min(planets_with_enough_ships, default=None, key=lambda p:
p.distance_to(planet))
        if nearest_planet:
            gameinfo.planet_order(nearest_planet, planet, required_ships) # Send reinforcement
```

In terms of tactical information, this AI could benefit from more advanced strategies. For instance, it could analyze the enemy's attack and defense patterns to predict future actions. It could also consider the rate of ship production on each planet to plan ahead more effectively.

Considering the "fog of war" feature in PlanetWars, an AI would have to make decisions based on partial and possibly inaccurate information. This could be exploited by creating deceptive strategies, like feigning weakness on one planet while preparing a large fleet on another.

The symmetry of most game maps could aid in making more accurate predictions about the enemy's position and strategy. However, in an asymmetric map, these predictions would be less accurate. This could lead to a more challenging and unpredictable game, offering more opportunities for unique and creative strategies. The asymmetry might also create a game bias if one position has an inherent advantage, for example, being closer to more planets or having planets with a faster ship production rate.

Spike 10: Tactical Steering (Hide!)
Extension:

- Hunter Cannot Enter Hiding Spot: In the updated version of our code, I have explicitly prevented the Hunter AI from entering the designated hiding spots. This prevents the scenario in which the Hunter occupies the hiding spot, making it inaccessible to the Prey. As a result, the hiding spots serve their intended purpose more efficiently, providing a safe haven exclusively for the Prey. This change was made in the update method of the Agent class, ensuring that if the hunter gets into a hiding spot, it immediately gets moved out.

```
def avoid_objects(self):
    avoid_force = Vector2D()
    for obj in self.world.hide_objects:
        # Calculate vector from agent to object
        to_object = self.pos - obj.pos
        # Check if object is near
        if to_object.length() < obj.radius + self.bRadius:
            # Calculate a force to push the agent away from the object
            avoid_force += to_object.normalise() / to_object.length()
    return avoid_force
```

- Prey Finding the Optimal Hiding Spot: The Prey AI has been improved to not just find any hiding spot, but rather, the optimal one. The Prey AI now takes into consideration the current location of the Hunter and selects a hiding spot that is furthest from the Hunter's position. By increasing the distance between the Hunter and the hiding spot, providing the Prey with a higher chance of reaching the hiding spot without getting caught. This strategy, though straightforward, adds a layer of sophistication to the Prey's decision-making process, making the game more dynamic and engaging.

```
# find the closest hunter agent
        closest_hunter = self.get_closest_agent('hunter')
```

```
            # Find the furthest hide object from the hunter
            furthest_hide_obj = self.find_furthest_hide_object_from_agent(closest_hunter)
            if furthest_hide_obj:
                hiding_position, _ = self.get_hiding_position_and_distance(furthest_hide_obj,
self.pos)
            else:
                hiding_position = None

            if hiding_position and closest_hunter:
                hiding_force = self.hide_behind_object(closest_hunter.pos)
                fleeing_force = self.flee(closest_hunter.pos)
                force = hiding_force + fleeing_force + self.seek(hiding_position)
```

Spike 12 - 16:

Extension: Weapon Reload Time

- I have added functionality to incorporate weapon reload time into our game mechanics. This introduces a realistic delay between the time a weapon exhausts its ammunition and when it's ready to fire again. The specifics of implementation are not covered in the provided code snippets, but such a feature usually involves setting a reload duration and tracking time since the weapon was last fired to determine when it's ready again.

```
    # Update reload timer
        if self.reloading:
            self.reload_timer -= delta
            if self.reload_timer <= 0:
                self.current_ammo = self.max_ammo
                self.reloading = False
```

Extension: Weapon Current Ammo

- Closely tied to the reload time is the tracking of current ammunition. This feature keeps a count of how many rounds a weapon has left before it needs to be reloaded. This count decreases with each shot fired and resets to the weapon's capacity after reloading. As with reload time, the specifics of this feature are not covered in the snippets, but it would typically involve an ammunition count variable that gets decremented with each shot and reset after the reload time has passed.

```
if self.current_ammo > 0 and not self.reloading:
            proj = Projectile(self.world, self, target_enemy, self.damage)
            proj.calculate()
            self.projectiles.append(proj)
            self.current_ammo -= 1
        elif self.current_ammo == 0 and not self.reloading:
            self.reloading = True
            self.reload_timer = self.reload_time
```

Extension: Picking up shield

- One of the new power-ups introduced in the game is a shield. The Shield class generates shield power-ups at random locations on the game map. When picked up by a character, presumably, it grants them extra protection. The shield object tracks its own position and re-spawns at a new random location after a set cooldown period once picked up. The shield object also follows the position of the character who picked it up until it's consumed or dropped.

Extension: Healing

- Another power-up introduced in the game is the health pickup, managed by the Health class. Similar to the shield power-up, the health object spawns at random locations on the map and restores health points to the character who picks it up. The health object tracks its own position, follows the character who picks it up, and re-spawns at a new random location after a set cooldown period once it's picked up.

Extension: Pick Up Shield

- This extension is concerned with the implementation of a mechanism that allows characters to pick up a shield power-up. This power-up provides the character with additional defence, enhancing their survivability during gameplay. The code provided is part of a function that is presumably called when a character comes into contact with a shield power-up. It assigns the shield to the character, sets the shield's enemy attribute to the character who picked it up, and updates the character's shield strength to match the shield's value. It also flags the shield as picked up. A debug message indicating the shield strength is printed, which could be helpful for testing and balancing gameplay.

```
def pick_up_shield(self, shield):
    self.shield = shield
    shield.enemy = self
    self.shield_strength = shield.shield_value
    shield.picked_up = True
    print(self.shield_strength)
```

Extension: Seeking For Health for Healing

- This extension introduces a new level of AI sophistication to the game. It describes a behavior where an entity seeks out health pickups when they're in need of healing. This mechanic adds to the realism and challenge of the game, as enemies can now actively restore their health, making them tougher opponents. The provided code appears to be a part of a larger function where entities scan the game world for health objects. It calculates the distance between the entity and each health object, ultimately identifying the closest one. If a health object is within range, the entity will seek it, presumably moving towards its location.

```
for health in self.world.healths:
                    distance = self.pos.distance(health.pos)
                    if distance < closest_distance:
                        closest_distance = distance
                        closest_health = health

                if closest_health is not None:
                    force = self.seek(closest_health.pos)
```