Spike: Task 11

Title: Emergent Group Behaviour

Author: Sam Huffer, 101633177

Goals / Deliverables

- A group agent steering behaviour simulation that can demonstrate several modes of emergent group behaviour. It must:
    - Include cohesion, separation and alignment steering behaviours.
    - Include basic wandering behaviours.
    - Use weighted sum to combine all steering behaviours.
    - Support the adjustment of parameters for steering forces while the code is running.
- As extensions, consider and demonstrate:
    - What happens when agents can't overlap and how does this affect the parameter settings discovered earlier in this task?
    - Adding a predator that other agents avoid.
    - Adding walls for agents to avoid, which will necessitate implementing feelers or another wall avoidance method. Consider what properties are needed to get agents circling around the game space.
    - Create different agent group types and investigate the behaviours that emerge between those groups.

Technologies, Tools, and Resources Used

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

Tasks Undertaken

To set up the project for this task, I copied the code used for Task 10: Tactical Steering to use as the basis for this task. I commented out several methods that, on a read through of the code required for the spike, seemed like they would not be useful at all, and changed the set up to generate one hunter, and allow the creation of prey agents with key input. I also shrank the prey agents to allow many more on the screen.

Next, I changed the key input logic to allow or disable various sets of key input according to keystrokes, and to allow the toggling of the display of various pieces of information pertaining to the prey agents (forces, wandering, avoidance etc.). I also updated the code so that instructions for keyboard input would be displayed on-screen, and that when prey agents existed, the values of their motion variables or associated weightings would be displayed on-screen, and would be changeable with keyboard input.

After that, I implemented the walls and code to agents allow them to avoid them, and to objects to prevent them spawning outside the wall margins.

Once the walls were set up, I implemented the code for the group behaviour methods, adapting the pseudocode in the lecture notes for this, and added a method to the world class to calculate all agents' neighbours so they wouldn't have to all do that themselves and double up on the work. I also changed the code to allow me to switch between the separation code outlined in the lecture

slides, and the agent avoidance code I developed in previous weeks' tasks, as I found that was much more effective in separating agents from each other.

Lastly, I tweaked the variables to achieve different sets of behaviour. To make this easier, I separated the values pertaining to the predator and prey agents in the agent __init__() method. While tweaking the variables, I noticed that the frame rate was dropping into the mid- to low-teens, even below ten sometimes, and I suspected the cause might be the high number of agents on the screen having to check for collisions with everything else. To improve the framerate there, I changed the code so that agents only check for collisions with obstacles, agents or walls that are in their neighbourhood (or within 3 times their radius for the predator), and tightened up the code calculating neighbourhoods somewhat; this did raise the average framerate up to the low- to mid-twenties.

Instructions for Operating the Code

- Key Escape: close the simulation
- Key P: toggle whether the simulation is paused or un-paused
- Key Backspace: exit current menu
- Key 0: spawn 10 new prey agents in random, valid positions, irrespective of which menu is open
- Key A: open prey spawning menu
    - Key [N]: spawn [N] prey agents in random, valid positions (0 spawns 10 agents)
- Key I: open display info menu
    - Key A: toggle displaying of avoidance boundaries
    - Key F: toggle displaying of the force acting on the agent
    - Key N: toggle displaying of prey agents' neighbourhoods
    - Key R: toggle displaying of agents' radii
    - Key W: toggle displaying of wander circles
- Key O: open the obstacle menu
    - Key N: spawn new obstacle
    - Key R: randomise the positions of the obstacles
- Key V: open the variable editing menu
    - Key Up: move selection up
    - Key Down: move selection down
    - Key Left: decrease the selected variable
    - Key Right: increase the selected variable
    - Key Plus/Equals: increase the increment step
    - Key Minus/Underscore: decrease the increment step

Code Snippets

```python
def calculate(self, delta):
    # reset the steering force
    mode = self.mode

    if mode == 'predator':
        force = self.calculate_predator(delta)
    elif mode == 'prey':
        force = self.calculate_prey(delta)
    else:
        force = Vector2D()

    return force

def calculate_predator(self, delta):
    force = self.avoid_obstacles(self.obstacle_neighbours) * self.obstacle_avoidance_multiplier
    force += self.avoid_walls(self.wall_neighbours) * self.obstacle_avoidance_multiplier

    if self.separate_by_avoid:
        force += self.avoid_agents(self.agent_neighbours)

    if force.length() == 0: #self.max_force:
        force = self.wander(delta)

    return force

def calculate_prey(self, delta):
    force = Vector2D()

    if self.add_force(force.length(), 1):
        if len(self.obstacle_neighbours) > 0:
            force += self.avoid_obstacles(self.obstacle_neighbours) * self.obstacle_avoidance_multiplier

        if len(self.wall_neighbours) > 0:
            force += self.avoid_walls(self.wall_neighbours) * self.obstacle_avoidance_multiplier

    if self.add_force(force.length(), self.separation_multiplier) and len(self.agent_neighbours) > 0:
        if self.separate_by_avoid:
            force += self.avoid_agents(self.agent_neighbours) * self.separation_multiplier
        else:
            force += self.separation() * self.separation_multiplier

    if self.add_force(force.length(), self.fleeing_multiplier):
        force += self.flee(self.world.predator.pos, delta) * self.fleeing_multiplier

    if self.add_force(force.length(), self.cohesion_multiplier):
        force += self.cohesion() * self.cohesion_multiplier

    if self.add_force(force.length(), self.alignment_multiplier):
        force += self.alignment() * self.alignment_multiplier

    if self.add_force(force.length(), self.wander_multiplier):
        force += self.wander(delta) * self.wander_multiplier

    return force
```

Figure 1: the updated calculate methods for the predator and prey, prioritising safety behaviours, then group behaviours, then wandering. Add_force() checks if I care if the force is maxed out, and if the force is maxed out if I do care. All forces get truncated in the update method once the total is calculated. The multipliers for each force are stored in the __init__() method and can be altered while the code is running.

Figure 2: the methods required for the emergent group behaviour, adapted from the pseudocode in the lecture slides. Separation didn't seem to work very effectively, even with a high multiplier, so I forwent using it in favour of obstacle avoidance code developed in previous weeks.

```python
def separation(self):
    force = Vector2D()
    avoid = []

    for agent in self.agent_neighbours:
        # force += self.avoid(agent)

        to_agent = self.pos - agent.pos
        force += to_agent.get_normalised() * (1 / to_agent.length())

    return Vector2D()
```

```python
def cohesion(self):
    centre_mass = Vector2D()
    force = Vector2D()
    avg_count = 0

    for agent in self.agent_neighbours:
        centre_mass += agent.pos
        avg_count += 1

    if avg_count > 0:
        centre_mass *= (1 / float(avg_count))
        force = self.seek(centre_mass)

    return force
```

```python
def alignment(self):
    avg_heading = Vector2D()
    avg_count = 0

    for agent in self.agent_neighbours:
        avg_heading += agent.heading
        avg_count += 1

    if avg_count > 0:
        avg_heading *= (1 / float(avg_count))
        avg_heading -= self.heading

    return avg_heading
```

```python
def calculate_neighbours(self):
    self.predator.neighbours = []
    self.predator.wall_neighbours = []
    self.predator.obstacle_neighbours = []

    for wall in self.walls:
        if self.predator.distance(wall.get_pos(self.predator.pos)) < self.predator.avoid_radius * 3:
            self.predator.wall_neighbours.append(wall)

    for obstacle in self.obstacles:
        if self.predator.distance(obstacle.pos) < self.predator.avoid_radius * 3 + obstacle.radius:
            self.predator.obstacle_neighbours.append(obstacle)

    for prey in self.prey:
        if self.predator.distance(prey.pos) < self.predator.avoid_radius * 3:
            self.predator.neighbours.append(prey)

        prey.agent_neighbours = []
        prey.wall_neighbours = []
        prey.obstacle_neighbours = []

        for agent in self.agents:
            if prey is not agent and prey.distance(agent.pos) < self.neighbourhood_radius:
                prey.agent_neighbours.append(agent)

        for wall in self.walls:
            if prey.distance(wall.get_pos(prey.pos)) < self.neighbourhood_radius:
                prey.wall_neighbours.append(wall)

        for obstacle in self.obstacles:
            if prey.distance(obstacle.pos) < self.neighbourhood_radius + obstacle.radius:
                prey.obstacle_neighbours.append(obstacle)
```

Figure 3: the code for sorting out what is in whose neighbourhood. First the predator gets its obstacles and walls, then its prey. Each prey that gets checked if its in the predator's neighbourhood also gets the agents, walls and obstacles in its neighbourhood to save the code from having to loop through the prey list twice.

```python
def get_pos(self, obj_pos):
    if self.side == 'top':
        return Vector2D(obj_pos.x, self.world.cy - self.margin)
    elif self.side == 'bottom':
        return Vector2D(obj_pos.x, self.margin)
    elif self.side == 'left':
        return Vector2D(self.margin, obj_pos.y)
    elif self.side == 'right':
        return Vector2D(self.world.cx - self.margin, obj_pos.y)
    else:
        return Vector2D()

def render(self, color=None):
    # draw the obstacle according to its default colour
    egi.set_pen_color(name=self.color)
    egi.set_stroke(2)
    egi.line_by_pos(self.pos_a, self.pos_b)
    # egi.cross(self.pos_a, 5)
    # egi.cross(self.pos_b, 5)

def set_points(self):
    if self.side == 'top':
        self.pos_a = Vector2D(self.margin, self.world.cy - self.margin)
        self.pos_b = Vector2D(self.world.cx - self.margin, self.world.cy - self.margin)
    elif self.side == 'bottom':
        self.pos_a = Vector2D(self.margin, self.margin)
        self.pos_b = Vector2D(self.world.cx - self.margin, self.margin)
    elif self.side == 'left':
        self.pos_a = Vector2D(self.margin, self.margin)
        self.pos_b = Vector2D(self.margin, self.world.cy - self.margin)
    elif self.side == 'right':
        self.pos_a = Vector2D(self.world.cx - self.margin, self.margin)
        self.pos_b = Vector2D(self.world.cx - self.margin, self.world.cy - self.margin)
```

Figure 4: the major methods of the Wall class. The __init__() method holds the points relevant for rendering and the string dictating what wall it is; set_points() is called when the window is instantiated or whenever the screen size changes, while get_pos() is used whenever an agent needs the wall's position, deriving it from the simulation space's boundaries and the agent's position.

```
egi.white_pen()
egi.text_at_pos(5, 5, 'Current Menu: ' + self.get_menu_text())
egi.text_at_pos(5, 20, 'Agent mode(s): ' + ', '.join(set(agent.mode for agent in self.agents)))

if len(self.prey) > 0:
    if self.change_values:
        egi.red_pen()
        y_top = self.cy - (5 + (15 * self.selected_index))
        y_bottom = self.cy - (20 + (15 * self.selected_index))

        pts = [
                Point2D(5,      y_top),     # top left
                Point2D(400,    y_top),     # top right
                Point2D(400,    y_bottom),  # bottom right
                Point2D(5,      y_bottom)   # bottom left
            ]
        egi.closed_shape(pts)

    egi.white_pen()
    egi.text_at_pos(10, self.cy - (5 + 15), 'Max Speed: ' + str(self.prey[0].max_speed))
    egi.text_at_pos(10, self.cy - (5 + 30), 'Max Force: ' + str(self.prey[0].max_force))
    egi.text_at_pos(10, self.cy - (5 + 45), 'Neighbourhood Radius: ' + str(self.neighbourhood_radius))
    egi.text_at_pos(10, self.cy - (5 + 60), 'Alignment Multiplier: ' + str(self.prey[0].alignment_multiplier))
    egi.text_at_pos(10, self.cy - (5 + 75), 'Cohesion Multiplier: ' + str(self.prey[0].cohesion_multiplier))
    egi.text_at_pos(10, self.cy - (5 + 90), 'Fleeing Multiplier: ' + str(self.prey[0].fleeing_multiplier))
    egi.text_at_pos(10, self.cy - (5 + 105), 'Fleeing Range: ' + str(self.prey[0].flee_range))
    egi.text_at_pos(10, self.cy - (5 + 120), 'Obstacle Avoidance Multiplier: ' + str(self.prey[0].obstacle_avoidar
    egi.text_at_pos(10, self.cy - (5 + 135), 'Separation Multiplier: ' + str(self.prey[0].separation_multiplier))
    egi.text_at_pos(10, self.cy - (5 + 150), 'Wander Multiplier: ' + str(self.prey[0].wander_multiplier))
    egi.text_at_pos(10, self.cy - (5 + 165), 'Separate by Avoid: ' + str(self.prey[0].separate_by_avoid))
    egi.text_at_pos(10, self.cy - (5 + 195), 'Value Step: ' + str(self.value_step))
```

Figure 5: the code for the display of prey agent variables. The current options are displayed at the bottom of the screen and collected from the get_menu_text() method using the values of the bools that signify what menu is open or if none at all are.

```
def select_variable(self, change):
    max_index = 10
    self.selected_index += change

    if self.selected_index < 0:
        self.selected_index = max_index
    elif self.selected_index > max_index:
        self.selected_index = 0

    if self.selected_index == 0:
        self.selected_variable = 'speed'
    elif self.selected_index == 1:
        self.selected_variable = 'force'
    elif self.selected_index == 2:
        self.selected_variable = 'neighbourhood radius'
    elif self.selected_index == 3:
        self.selected_variable = 'alignment'
    elif self.selected_index == 4:
        self.selected_variable = 'cohesion'
    elif self.selected_index == 5:
        self.selected_variable = 'fleeing multiplier'
    elif self.selected_index == 6:
        self.selected_variable = 'fleeing range'
    elif self.selected_index == 7:
        self.selected_variable = 'obstacle avoidance'
    elif self.selected_index == 8:
        self.selected_variable = 'separation'
    elif self.selected_index == 9:
        self.selected_variable = 'wander'
    elif self.selected_index == 10:
        self.selected_variable = 'separate by avoid'
```

Figure 6: the code used to change the selected prey agent variable to a different variable; changes are restricted to incrementing or decrementing selected_index by one and wrapping it at the bounds of the available options.

```
def change_value(self, value, step, sign):
    if value == 'speed':
        for agent in self.prey:
            agent.max_speed += step * sign

            if agent.max_speed < 0:
                agent.max_speed = 0
    elif value == 'force':
        for agent in self.prey:
            agent.max_force += step * sign

            if agent.max_force < 0:
                agent.max_force = 0
    elif value == 'neighbourhood radius':
        self.neighbourhood_radius += step * sign

        if self.neighbourhood_radius < 0:
            self.neighbourhood_radius = 0
    elif value == 'alignment':
        for agent in self.prey:
            agent.alignment_multiplier += step * sign

            if agent.alignment_multiplier < 0:
                agent.alignment_multiplier = 0
```

Figure 7: the method that alters the values of the prey agent variables; this only shows the first few variables for brevity, but the method encompasses all variables detailed in fig. 6.

In-Simulation Screenshot

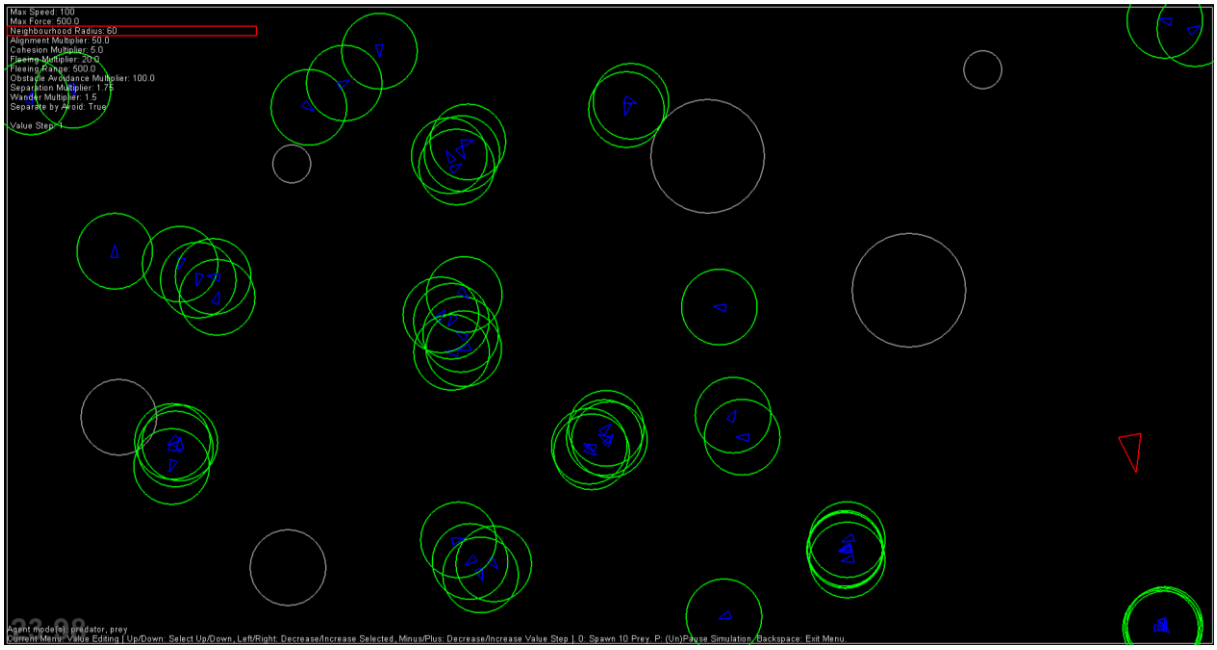

Figure 8: a screenshot of the simulation running. Prey agents (dark blue) have their neighbourhoods being displayed (green) and are cohering together into a number of loose groups. The predator (red) is wandering about, while a group of prey in the bottom right corner panic to get away from it, packing tightly together; another group to their left are fleeing from the predator in a more orderly manner. The top left corner shows the display of the prey agents' variables, with the currently selected variable bounded in red. The lists of current agent types and available controls are displayed in the bottom left corner.

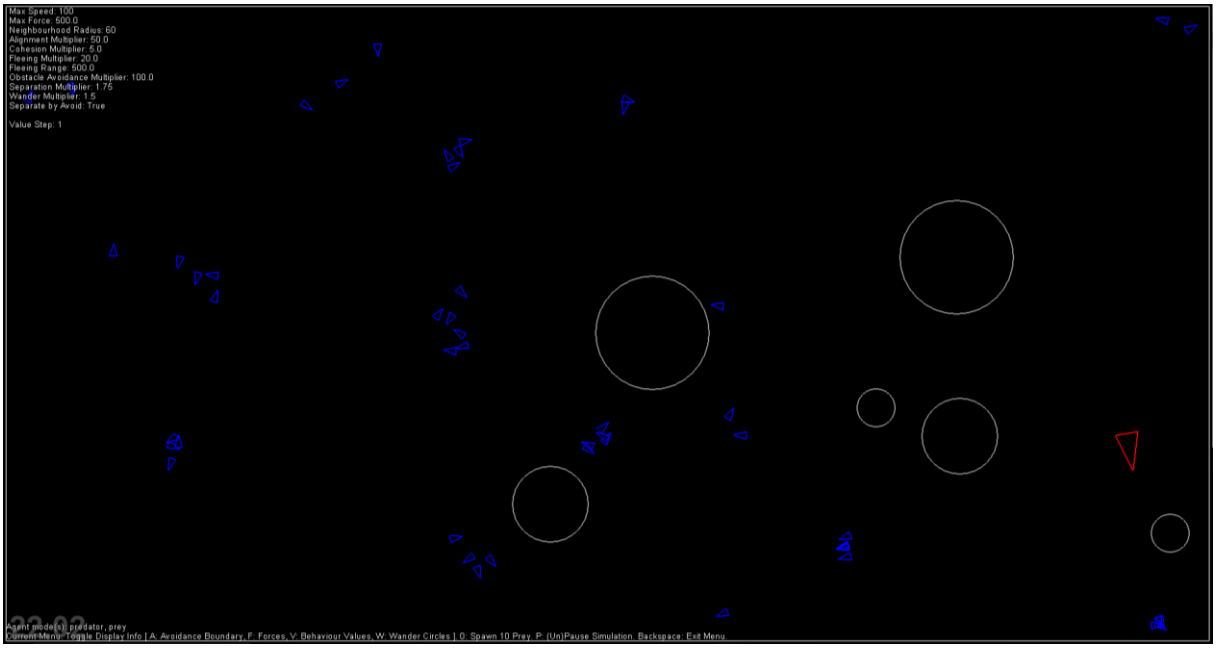

Figure 9: the simulation-state from fig. 8 with the neighbourhoods turned off and the variable editing menu exited.

What I Found Out

To keep prey from colliding with walls or obstacles, I needed to keep the obstacle avoidance multiplier quite high and position the checks for wall and obstacle avoidance at the top of the list of gathered forces checks.

After tweaking the variables, I arrived at a balance of parameters I was reasonably happy with. And saved it to the agent file for prey agents to use as their default behaviour. As a result of the proportions of values that I arrived at, the prey exhibited the following behaviour:

- They would congregate together in schools.
- If the predator was wandering off somewhere else, the prey agents just milled about each other and the group meandered a little bit, more so if they managed to align to each other.
- If the predator was within their fleeing range, the prey would pack together tightly and flee to a corner.

From there, I also played with the values of the parameters further, and observed the following:

- Putting alignment through the roof and raising the max force resulted in some small groups aligning better and wandering together while the predator was off doing something else. Larger groups just packed more closely together.
- Increasing the separation multiplier, I could get the agents to stop themselves from overlapping. However, I could only achieve this with the obstacle avoidance code I had already written for previous tasks (the separation code based on the lecture content did little to prevent agents overlapping), and once the predator came near, they all tried to pile into a corner but couldn't all fit and ended up shaking in place.
- Increasing the cohesion multiplier resulted in fewer and very tightly packed groups, groups that might wander a bit if they all aligned well, which the predator would prompt if it came nearby and then wandered off again without getting too close.

While adjusting the parameters, I couldn't get the prey agents circling around the boundaries of the simulation. Perhaps a higher speed and walls that more closely resemble a circle (octagonal walls) than a rectangle would be more conducive to circling behaviour. I should note that I didn't delve too deeply into investigating circling behaviour due to other units also needing attention.

Similarly, I didn't attempt the extension for different types of agents. For the latter, I imagine I would need to maintain several sets of parameters for each type, the same as how I've separated the predator's and prey's parameters, and be able to switch between the agent types for spawning and variable viewing and editing purposes. With the agent types suggested:

- Zombies would probably want to include seeking or pursuit behaviours covered in previous weeks to seek out prey agents and have a much lower maximum speed.
- Soldiers would probably want to use the path methods from previous weeks and patrol the simulation space, perhaps also using seek or pursuit behaviours to chase prey agents at a normal top speed.
- Perhaps the prey agents as they are now could be recoloured green and made into aliens; I'm not sure what further behaviours I would add to them beyond the prey's current behaviour (not that I've considered it too much though).