Spike: 13
Title: Emergent Group Behaviour

Author: Marco Giacoppo, 104071453

Goals / deliverables:
This spike aimed to create a dynamic simulation where multiple agents demonstrate emergent flocking behavior through the combination of four core steering forces: wander, cohesion, separation, and alignment.

The agents' behavior is controlled using real-time adjustable weight values, allowing interactive exploration of how each component contributes to group dynamics.

Deliverables:
- Functional Python simulation with 20+ autonomous agents
- Flocking behavior via combined steering logic
- Real-time adjustment of behavior weights using keyboard input
- On-screen display of selected weight type
- Arrow-key-based intuitive tuning interface
- Visual agent rendering using Pyglet

Technologies, Tools, and Resources used:
- Python 3.11
- Pyglet (for game visualisation)
- Lab 13 code for foundation
- ChatGPT for planning and debugging

Tasks undertaken:

1. Base Setup and Agent Spawning
- Extended the Agent class to support multiple autonomous FlockAgent instances.

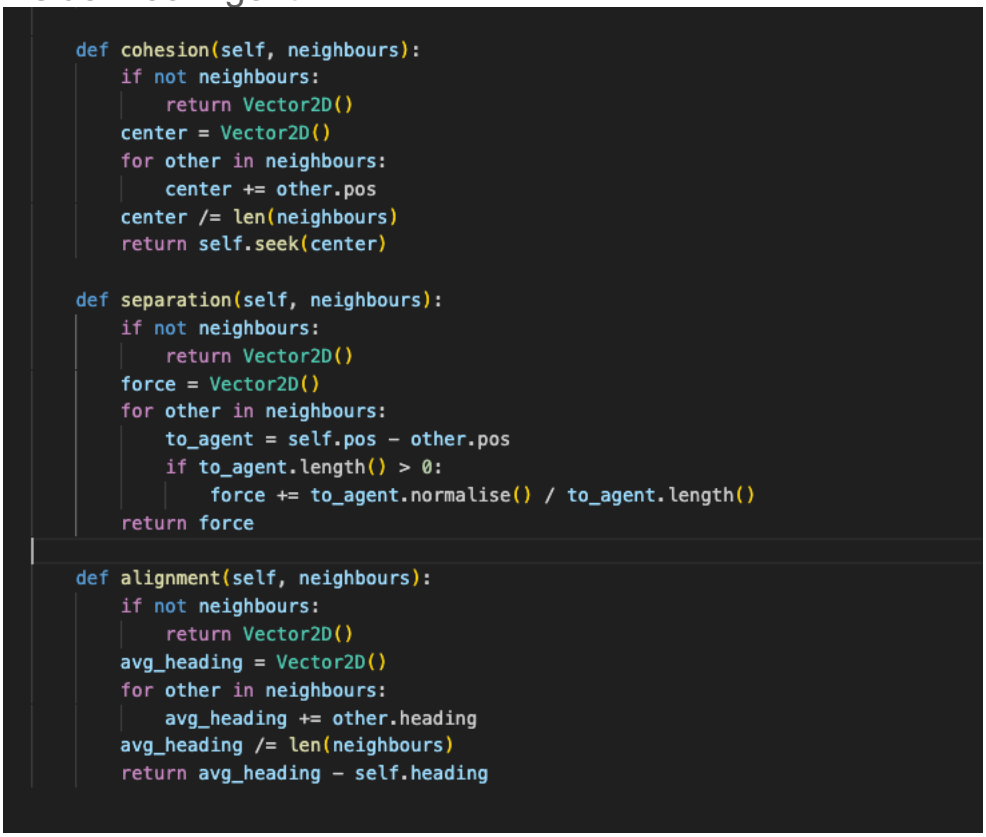
```
class FlockAgent(Agent):
    def __init__(self, world, scale=20.0, mass=1.0, color='AQUA'):
        super().__init__(world, scale, mass, color)
        self.wander_weight = 0.3
        self.cohesion_weight = 0.5
        self.separation_weight = 0.8
        self.alignment_weight = 0.5

        self.wander_target = Vector2D(1, 0)
        self.wander_dist = 1.0 * scale
        self.wander_radius = 1.0 * scale
        self.wander_jitter = 10.0 * scale
```

- Spawned 20 flocking agents using a loop in the World constructor.

2. Steering Behaviours Implementation

- Implemented cohesion, separation, alignment, and wander methods inside FlockAgent.



```
def cohesion(self, neighbours):
    if not neighbours:
        return Vector2D()
    center = Vector2D()
    for other in neighbours:
        center += other.pos
    center /= len(neighbours)
    return self.seek(center)

def separation(self, neighbours):
    if not neighbours:
        return Vector2D()
    force = Vector2D()
    for other in neighbours:
        to_agent = self.pos - other.pos
        if to_agent.length() > 0:
            force += to_agent.normalise() / to_agent.length()
    return force

def alignment(self, neighbours):
    if not neighbours:
        return Vector2D()
    avg_heading = Vector2D()
    for other in neighbours:
        avg_heading += other.heading
    avg_heading /= len(neighbours)
    return avg_heading - self.heading
```

- Each behavior returns a force vector steering the agent in a specific direction.

3. Weighted Sum Combination
- Each behavior's influence is scaled using a weight.
- Final steering force is computed using:
  total_force = wander * w + cohesion * c + separation * s + alignment * a

4. Real-Time Parameter Control
- Replaced individual keys with a behavior selector (1–4) and **arrow keys (↑ / ↓)` to adjust the selected weight.
- Current selection is displayed in the top-left using a pyglet.text.Label.

5. Neighbourhood Detection
- Implemented get_neighbours(agent, radius) in World.py
- This returns a list of other flock agents within a certain distance (default 100 units)
- Each agent uses its neighbours to compute cohesion, separation, and alignment forces.
- Ensures each agent responds only to nearby agents – enabling realistic, local emergent behaviour instead of global syncing.

```
        return world_pt

    def get_neighbours(self, agent, radius=100):
        return [other for other in self.flock_agents if other is not agent and
          (other.pos - agent.pos).length() < radius]
```

6. Visualization & Feedback

- Agents are rendered as triangles with heading direction.
- Parameter values can be printed to the console using the T key for debugging.

```
def input_keyboard(self, symbol, modifiers):
    if symbol == pyglet.window.key.P:
        self.paused = not self.paused

    elif symbol == pyglet.window.key._1:
        self.selected_weight = "cohesion"
        self.update_selected_label()
        print("Selected: Cohesion")

    elif symbol == pyglet.window.key._2:
        self.selected_weight = "separation"
        self.update_selected_label()
        print("Selected: Separation")

    elif symbol == pyglet.window.key._3:
        self.selected_weight = "alignment"
        self.update_selected_label()
        print("Selected: Alignment")

    elif symbol == pyglet.window.key._4:
        self.selected_weight = "wander"
        self.update_selected_label()
        print("Selected: Wander")

    elif symbol == pyglet.window.key.UP:
        for a in self.flock_agents:
            if self.selected_weight == "cohesion":
                a.cohesion_weight += 0.3
            if self.selected_weight == "separation":
                a.separation_weight += 0.3
            if self.selected_weight == "alignment":
                a.alignment_weight += 0.3
            if self.selected_weight == "wander":
                a.wander_weight += 0.3
        print(f"Increased {self.selected_weight} weight")

    elif symbol == pyglet.window.key.DOWN:
        for a in self.flock_agents:
            if self.selected_weight == "cohesion":
                a.cohesion_weight = max(0, a.cohesion_weight - 0.3)
            if self.selected_weight == "separation":
                a.separation_weight = max(0, a.separation_weight - 0.3)
            if self.selected_weight == "alignment":
                a.alignment_weight = max(0, a.alignment_weight - 0.3)
            if self.selected_weight == "wander":
                a.wander_weight = max(0, a.wander_weight - 0.3)
        print(f"Decreased {self.selected_weight} weight")

    elif symbol == pyglet.window.key.T:
        for a in self.flock_agents:
            print(f"W: {a.wander_weight:.2f}, C: {a.cohesion_weight:.2f}, S: {a.separation_weight:.2f}, A: {a.alignment_w
```

What we found out:

- Cohesion encourages agents to move toward the group center. When too high, agents clump too tightly.

- Separation prevents overlap and crowding. Essential for natural-looking spacing.

- Alignment leads to synchronized movement. With high alignment, agents appear to “flock” smoothly.

- Wander maintains variability and prevents rigid motion. Too much wander breaks flocking.

- Balancing weights creates emergent behavior — realistic flocks form when forces are tuned together.

- Interactive control is extremely useful — real-time tuning makes it easy to visualize each behavior’s role.

- Local neighbourhoods are key to emergence. Each agent does not know about the whole group — it only reacts to nearby agents.

- Changing the neighbourhood radius (e.g., from 50 to 150) significantly alters behavior:

- Smaller radius → tighter, more fragmented groups.

- Larger radius → smoother, more unified motion.

- The decentralized nature of neighbor detection is what allows emergence — global patterns from simple local rules.