Spike: Task 10

Title: Tactical Steering

Author: Sam Huffer, 101633177

Goals / Deliverables

- A hunter-prey simulation involving multiple agents, with the prey hiding behind obstacles to evade the hunter. The simulation must:
    - Include objects (simple circles) for the prey to hide behind.
    - Distinguish between the hunter and prey's appearance and abilities.
    - Indicate suitable hiding spots with an "x" or similar.
    - Prey agents must pick a good hiding spot using tactical evaluation.
- Instructions on how to operate the code in the spike report.

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

- I copied the Autonomous Steering project from task 9 into the task 10 folder, and then went through the code to comment out any code or methods that seemed like they would not be necessary for this variant of the project. I didn't delete the code outright in case I needed it later.
- First, I developed the code for the base requirements of the task:
    - I added the code for generating obstacles in random positions and rearranged the code to just spawn one hunter and one evader, the former being made to wander randomly with the wander() method developed in previous tasks.
    - Next task was to adapt the lecture's notes on calculating hiding spots based on the hunter's position and implementing that code in the agent class, as well as the code to select the best hiding spot. At this point, I just had the agent arrive at the closest hiding spot. Each spot's position was marked by an "X", which was red unless it was the agent's chosen hiding spot, which was orange.
- The coding of the base requirements was fairly straightforward. The bulk of the time spent on this task was taken up by coding the extensions:
    - First was the ability to avoid objects (fig. 1). To implement this, I took from the lecture notes the ideas of the three feeler lines and the detection box out in front of the agent, but as I couldn't quite figure out how to implement a few methods detailed in either concept's code, adapted them to instead use the agent's position and a point in front of it and an avoidance radius equal to 1.5 times the agent's radius to check if another agent or an obstacle was getting too close and force needed to be applied to avoid it. These circles are rendered around the agent in blue, turning red if they detect an object to avoid. Later, I added some code such that if an agent would collide with an object, it simply didn't move to its new position.
    - Next I developed the code for the agent to process several tactical considerations when choosing a hiding spot (fig. 3) besides the agent's proximity to the hiding spot,

such as the hunter's proximity to it, and if traveling as the crow flies to it would lead it to cross the hunter's field of view. The field of view I implemented, rendered in white, used a point projected out in front of the hunter, and two extra points flanking it on either side, borrowing the offset pursuit's concept. I also limited the depth of the field of view if an obstacle (and later agents too) would obstruct the field of view (fig. 7).

- The next section I completed was the spawning of new hunters and evaders at the press of a key; that required mapping the spawning to a key-press, swapping the fields for the hunter and evader with lists, and changing every mention of either to use the lists and check each agent in those lists as appropriate, a task that was more tedious than difficult. At this stage, I also implemented a hiding spot class rather than just using positions for hiding spots so that I could bundle in a number of other details and include the rendering functionality for the hiding spots in the class, and create the hiding spots with the obstacles and just have their positions changed.
- The last thing that I implemented was the behaviour for the hunters to hunt the evaders (fig. 6). They wander if there are no evaders in their immediate vicinity or if there are none in their field of vision, the former using similar logic to the obstacle avoidance code detecting obstacles to avoid, the latter taking advantage of the limiting of the field of view markers and checking if they were within a certain range of any evaders. Once hunters detect an evader, they would pursue the closest evader, eating it when they get close enough. This can be repeated indefinitely by spawning new evaders.

Code Snippets

```
def avoid(self, obj_pos):
    desired_vel = (self.pos - obj_pos).normalise() * self.max_speed
    return (desired_vel - self.vel)

def avoid_agents(self, agents):
    agts = agents.copy()
    agts.remove(self)
    return self.avoid_obstacles(agts)

def avoid_obstacles(self, obstacles):
    sns_pos = Vector2D(min(self.avoid_radius * 2, self.vel.length()), 0) # factor in current speed
    self.sensor_pos = self.world.transform_point(sns_pos, self.pos, self.heading, self.side)

    closest_obst = None
    closest_dist = 9999999999999

    closest_obst_sns = None
    closest_dist_sns = 9999999999999

    result = Vector2D(0, 0)

    for obstacle in obstacles:
        dist_to_self = self.distance(obstacle.pos)
        dist_to_avoid = (obstacle.pos - self.sensor_pos).length()

        # if dist_to_avoid < self.avoid_radius + obstacle.radius or dist_to_self < self.avoid_radius +
        #     if self.mode == 'hide':
        #         self.best_hiding_spot.collisions += 1

        if dist_to_self < self.avoid_radius + obstacle.radius and dist_to_self < closest_dist:
            closest_obst = obstacle
            closest_dist = dist_to_self

        if dist_to_avoid < self.avoid_radius + obstacle.radius and dist_to_avoid < closest_dist_sns:
            closest_obst_sns = obstacle
            closest_dist_sns = dist_to_avoid

    if closest_obst is not None:
        self.obst_detected = True
        result += self.avoid(closest_obst.pos)

    if closest_obst_sns is not None:
        self.sensor_obst_detected = True
        result += self.avoid(closest_obst_sns.pos)

    return result
```

Figure 1: The code for calculating the force for avoiding obstacles.

```python
def get_hiding_spots(self, hunters, obstacles):
    hiding_spots = []

    # check for possible hiding spots
    for hunter in hunters:
        for obstacle in obstacles:
            spot = obstacle.hiding_spot
            spot.pos = self.get_hiding_spot_position(hunter, obstacle)
            spot.valid = True
            spot.rank = 0
            total = 0

            # check if spot is in the bounds of the screen
            if spot.pos.x < 0 or spot.pos.x > self.cx or spot.pos.y < 0 or spot.pos.y > self.cy:
                spot.valid = False
            else:
                # check if spot is inside the bounds of an object
                for o in obstacles:
                    if spot.distance(o.pos) < o.radius:
                        spot.valid = False

            # update spot data
            for hunter in hunters:
                total += spot.distance(hunter.pos)

            spot.avg_dist_to_hunter = total / len(hunters)
            hiding_spots.append(spot)

    return hiding_spots

def get_hiding_spot_position(self, hunter, obstacle):
    # set the distance between the obstacle and the hiding point
    dist_from_boundary = self.agent_avoid_radius + 5.0
    dist_away = obstacle.radius + dist_from_boundary

    # get the normal vector from the hunter to the hiding point
    to_obstacle = obstacle.pos - hunter.pos
    to_obstacle.normalise()

    # scale size past the obstacle to the hiding location
    return (to_obstacle * dist_away) + obstacle.pos
```

Figure 2: The code in the World class for determining the hiding spots.

```python
def get_best_hiding_spot(self, spots, hunters):
    # sort and / or rank the hiding spots
    prioritise_close = False
    prioritise_far_from_hunter = False

    for spot in spots:
        spot.rank = 0

    if prioritise_close:
        # get closest
        for spot in spots:
            spot.dist_to_evader = self.distance(spot.pos)
        spots.sort(key=lambda x: x.dist_to_evader)
    elif prioritise_far_from_hunter:
        # get furthest from hunter
        spots.sort(key=lambda x: x.avg_dist_to_hunter, reverse=True)
    else:
        # sort and increment spots' rank by closeness, lowest (index = 0) to highest (ind
        for spot in spots:
            spot.dist_to_evader = self.distance(spot.pos)
        spots.sort(key=lambda x: x.dist_to_evader)
        for i in range(len(spots) - 1):
            spots[i].rank += i

        # sort and increment spots' rank by distance from hunter, highest (index = 0) to
        spots.sort(key=lambda x: x.avg_dist_to_hunter, reverse=True)
        for i in range(len(spots) - 1):
            spots[i].rank += i

        # account for the number of collisions that have been experienced going to each s
        # got agent stuck in a loop where it kept changing the hiding spot it was going t
        # if False:
        #     for spot in spots:
        #         spot.rank += spot.collisions

        # increment spots' rank if moving to them would require crossing the hunter's lin
        for spot in spots:
            for hunter in hunters:
                if self.intersect(self.pos, spot.pos, hunter.pos, hunter.fov_markers[1]):
                    spot.rank += 999999999

        # sort spots by rank, lowest (index = 0) to highest (index = len - 1)
        spots.sort(key=lambda x: x.rank)

    for i in range(0, len(spots) - 1):
        if spots[i].valid:
            return spots[i]

    return None
```

Figure 3: The code for selecting the best hiding spot for an evader.

```python
def hide(self, hunters, obstacles, hiding_spots, delta):
    self.best_hiding_spot = None

    # if only one hiding spot, just pick that spot
    if len(hiding_spots) == 1:
        self.best_hiding_spot = hiding_spots[0]
    elif len(hiding_spots) > 1:
        # go to best hiding spot
        self.best_hiding_spot = self.get_best_hiding_spot(hiding_spots, hunters)

    if self.best_hiding_spot is None:
        # default: panic and run away from a random hunter
        if len(hunters) == 0:
            return self.wander(delta)
        elif len(hunters) == 1:
            return self.flee(hunters[0].pos, delta)
        else:
            return self.flee(hunters[randrange(0, len(hunters) - 1)].pos, delta)
    else:
        self.best_hiding_spot.best = True
        return self.arrive(self.best_hiding_spot.pos, 'fast')
```

Figure 4: The code for evader agents to hide from hunter agents.

```python
def intersect(self, p1, q1, p2, q2):
    # Given three colinear points p, q, r, the function checks if
    # point q lies on line segment 'pr'
    def on_segment(p, q, r):
        if (q.x <= max(p.x, r.x) and q.x >= min(p.x, r.x) and q.y <= max(p.y, r.y) and q.y >= min(p.y, r.y)):
            return True

        return False

    # To find orientation of ordered triplet (p, q, r).
    # The function returns following values
    # 0 --> p, q and r are colinear
    # 1 --> Clockwise
    # 2 --> Counterclockwise
    def orientation(p, q, r):
        val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y)

        if val == 0:
            return 0    # colinear
        elif val > 0:
            return 1    # clockwise
        else:
            return 2    # anticlockwise

    # Find the four orientations needed for general and
    # special cases
    o1 = orientation(p1, q1, p2)
    o2 = orientation(p1, q1, q2)
    o3 = orientation(p2, q2, p1)
    o4 = orientation(p2, q2, q1)

    # General case
    if (o1 is not o2 and o3 is not o4):
        return True

    # Special Cases
    # p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if o1 == 0 and on_segment(p1, p2, q1):
        return True

    # p1, q1 and q2 are colinear and q2 lies on segment p1q1
    if o2 == 0 and on_segment(p1, q2, q1):
        return True

    # p2, q2 and p1 are colinear and p1 lies on segment p2q2
    if o3 == 0 and on_segment(p2, p1, q2):
        return True

    # p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if o4 == 0 and on_segment(p2, q1, q2):
        return True

    return False # Doesn't fall in any of the above cases
```

Figure 5: Code borrowed for determining if two lines intersect; used to check if an agent's path would cross the hunters' field of vision when moving to a hiding spot.

```python
def hunt(self, evaders, delta):
    prioritise_visible = False
    prioritise_close = False

    hunting = []
    visible = []
    close = []
    dist_multiplier = 1.25

    for evader in evaders:
        if self.distance(evader.pos) < (self.avoid_radius + evader.avoid_radius) * dist_multiplier:
            close.append(evader)

        for marker in self.fov_markers:
            if evader.distance(marker) < evader.avoid_radius: #* dist_multiplier
                visible.append(evader)

    hunting = list(set(visible) and set(close))

    visible.sort(key=lambda x: x.distance(self.pos))

    if len(hunting) > 0:
        hunting.sort(key=lambda x: x.distance(self.pos))
        return self.pursuit(hunting[0])
    elif prioritise_close and len(close) > 0:
        if len(close) > 1:
            close.sort(key=lambda x: x.distance(self.pos))
        return self.pursuit(close[0])
    elif prioritise_visible and len(visible) > 0:
        if len(visible) > 1:
            visible.sort(key=lambda x: x.distance(self.pos))
        return self.pursuit(visible[0])
    else:
        hunting = close + visible
        if len(hunting) == 0:
            return self.wander(delta)
        else:
            if len(hunting) > 1:
                hunting.sort(key=lambda x: x.distance(self.pos))
            return self.pursuit(hunting[0])
```

Figure 6: The code for the hunters to check if there is something visible or close enough to pursue.

```python
def update_fov(self, obstacles, agents):
    crossed_obj = False
    max_fov_length = self.avoid_radius * 10
    fov_length = self.radius
    fovm = Vector2D()
    fov_marker = Vector2D()
    markers = []
    objects = obstacles + agents
    objects.remove(self)

    while not crossed_obj and fov_length < max_fov_length:
        fovm = Vector2D(fov_length, 0)
        fov_marker = self.world.transform_point(fovm, self.pos, self.heading, self.side)

        for obj in objects:
            if obj.distance(fov_marker) < obj.radius:
                crossed_obj = True
                overshoot = obj.radius - obj.distance(fov_marker)

        if not crossed_obj:
            fov_length += self.radius
        else:
            fov_length -= overshoot

    fov_length = min(fov_length, max_fov_length)
    fovm = Vector2D(fov_length, 0)
    offset = 2 * self.scale_scalar
    m_left = Vector2D(fov_length, -offset)
    m_right = Vector2D(fov_length, offset)
    markers.append(self.world.transform_point(m_left, self.pos, self.heading, self.side))
    markers.append(self.world.transform_point(fovm, self.pos, self.heading, self.side))
    markers.append(self.world.transform_point(m_right, self.pos, self.heading, self.side))
    self.fov_markers = markers
```

Figure 8: The code for updating hunters' field of view, moving the centre marker forward until it reaches its max length or is obstructed by an object.

```python
def eat(self, evaders):
    to_eat = []

    for evader in evaders:
        if self.distance(evader.pos) < self.avoid_radius + evader.avoid_radius:
            for marker in self.fov_markers:
                if evader.distance(marker) < evader.avoid_radius:
                    to_eat.append(evader)

    for evader in to_eat:
        self.world.destroy(evader)

    del to_eat
```

Figure 7: The code for checking if any evaders are in eating range, and then removing them from the game.

Instructions for Operating the Code

- E: create a new evader agent.
- H: create a new hunter agent.
- O: create a new obstacle in a random but valid position.
- R: reposition all obstacles in random but valid positions. Obstacles are automatically repositioned when the window changes size.
- P: pause or un-pause the game.
- Escape: exit the game.
- F: toggle friction method on and off.
- I: toggle the display of agents' force, velocity and net desired change in position.

What I Found Out

- Obstacle avoidance based solely on breaking force doesn't mix with high speeds; when an agent was travelling at a high velocity and would encounter an obstacle that they had to avoid, if they were travelling too fast, the breaking force would be insufficient to prevent the agent from coming into contact with the obstacle. Indeed, once the agent's centre point had passed over the obstacle's, the breaking force would turn into acceleration out of the obstacle, making the agent travel faster and making it move through other obstacles as well. To get around this, as noted above, I eventually implemented some code to revert the agent back to its original position if moving would result in a collision.
- Force-based obstacle avoidance doesn't account for wrapping an agent's position to the other side of the screen if it leaves the screen, as the position change there is not force based but merely teleportation to the other side of the screen. The only way to avoid teleporting into obstacles on the edge of the screen was to use the position reversion as with the failure of force-based obstacle avoidance described above.
- The selection of hiding spots also needed to take into account the position of a hiding spot relative to all the obstacles in the game space, as hiding spots were often prone to passing into nearby obstacles, yet the evader would still seek out the hiding spot regardless.
- As part of the tactical analysis going into selecting a hiding spot, at one stage I implemented a counter for hiding spots to record the number of collisions experienced while moving to a hiding spot. This didn't end up working so well when there were a number of obstacles positioned close together and the evader could move between them, as the evader would move towards one hiding spot, notice it was getting too close to an obstacle, update it's collision counter, choose a new hiding spot, and repeat, getting stuck bouncing from obstacle to obstacle like a pinball.
- Although the evaders were spawned at random positions, eventually they would select the same hiding spot to hide at. Once this happened and they got close enough to each other, they would make identical decisions and seek out the same hiding spots.