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

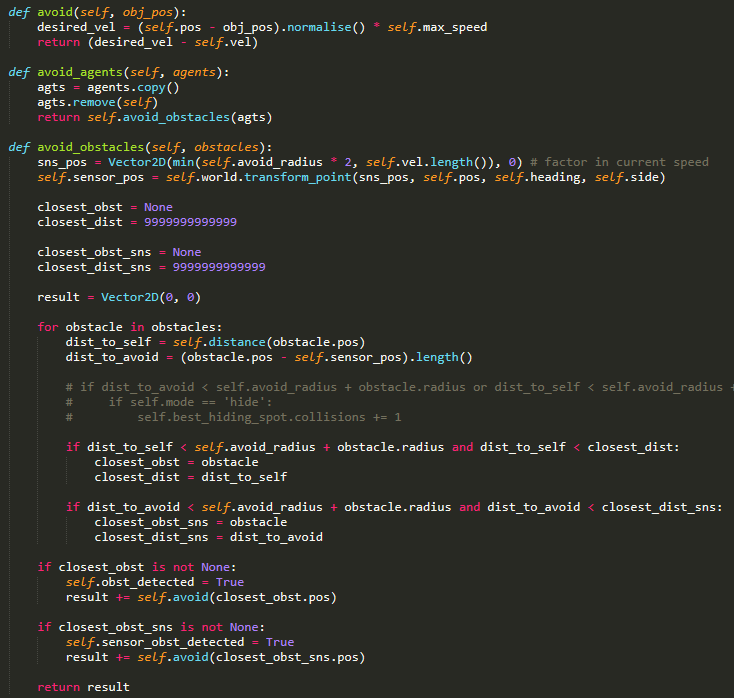


Figure 1: The code for calculating the force for avoiding obstacles.

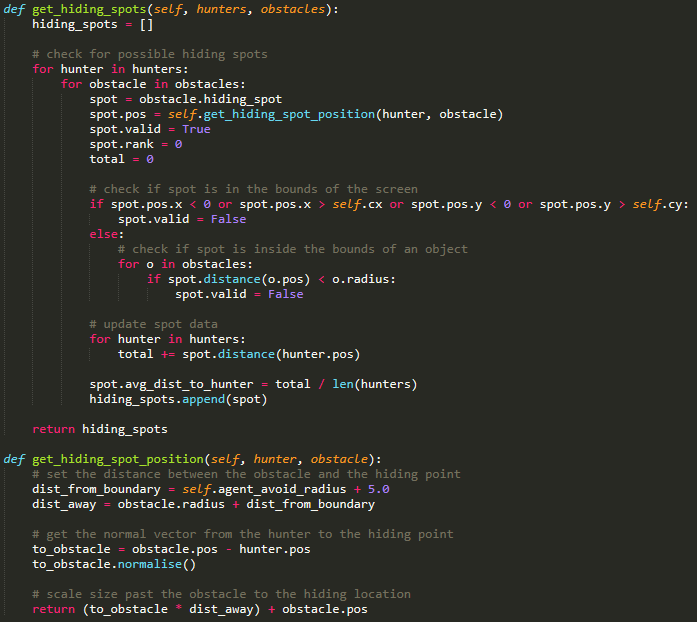


Figure 2: The code in the World class for determining the hiding spots.



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

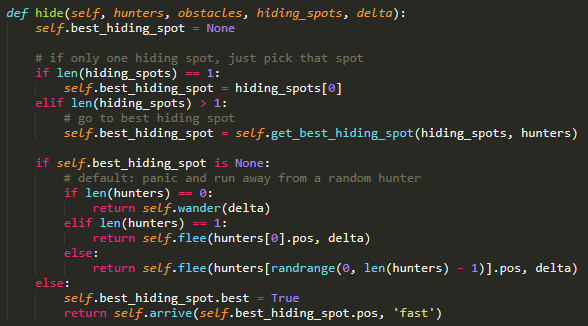


Figure 4: The code for evader agents to hide from hunter agents.

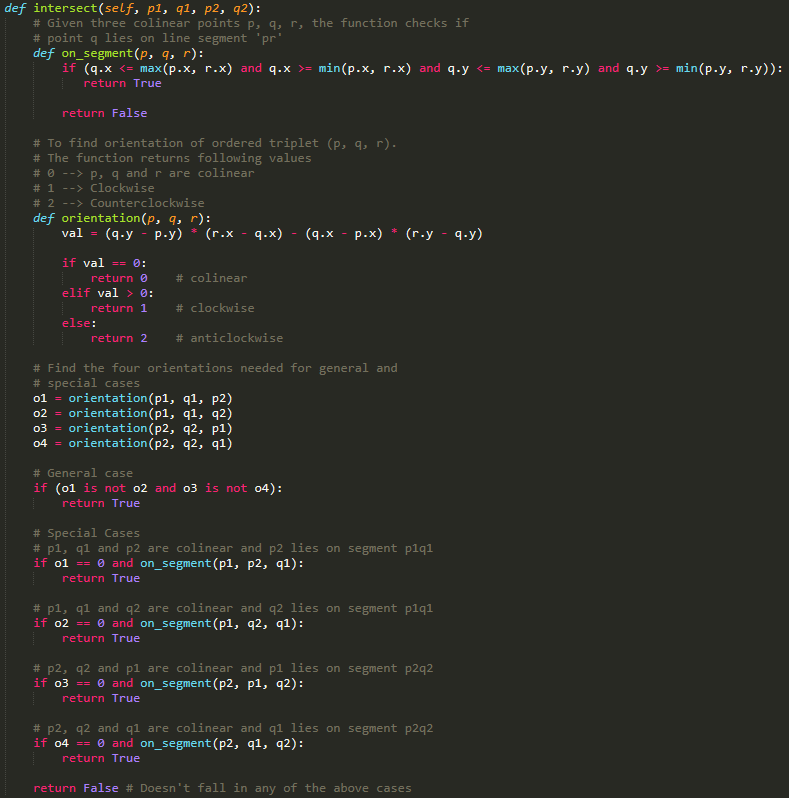


Figure 5: Code borrowed for determining if two lines intersect; used to check if an agent’s path would cross the hunters’ field of vision when moving to a hiding spot.



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

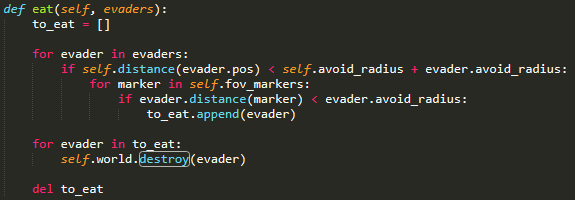


Figure 7: The code for checking if any evaders are in eating range, and then removing them from the game.

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.

# 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.

# In-Game Screenshot

Figure : How the game renders now that the code is complete. Note: there is a hiding spot that is selected and is rendered as orange, but for some reason it flickers between red and orange sometimes and I couldn’t manage to pause it for a screenshot with the hiding spot rendered in orange. This detail I’ll have to demonstrate by running the code in person.

# 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.