SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30002

Emergent Group Behaviour

Task 11, Week 7
Spike: Spike_No11

Matthew Coulter
S102573957

30/04/2020

Tutor: Tien Pham

Table of Contents

Goals / Deliverables

The aim of this task is to use cohesion, separation and alignment steering behaviours to have multiple agents fly in a group heading the same direction. Subsequently, they will also be trying to maintain distance with each other so they don't overlap often (could be dangerous). First of all, I think it is important two understand what each individual variable represents.

Cohesion generally refers to the size required for an agent to join another agents flock.

Separation refers to two or more agents trying to maintain equal distance with each other

Lastly, alignment will aim to have agents fly in an overall more forward direction.

As part of this implementation, we will implement a way by which the user can control these variables during gameplay to find a good balance of variables and display them to the screen to help with debugging.

Technologies, Tools, and Resources used:

If you wish to reproduce my work, here is a list of all the required tools and resources that you will need:

- Visual Studio
- Python language pack
- This spike report
- lab09 & lab10 code
- Git bash or sourcetree (if you wish to access my code)
    - repo: 'git clone https://mattcoulter7@bitbucket.org/mattcoulter7/cos30002-102573957.git'

Tasks undertaken:

Use the existing lab code, copy and create a new project

For this task, we will copy the code straight from task 9 as this version still has all of our keyboard input controls and we don't need to hunter/prey and hide additions from task 10.

Extend the code to support multiple agents and new keyboard input

From here we can begin to remove unnecessary controls and code

We will only be using the wander control (and seek because wander calls seek), so we will remove all of the modes except for wander. Delete the actual code for what those modes call as well. Remove Deceleration_speeds and have the calculation by default call wander.

Subsequently, we also need to update the main controls.

Because the evidence of our code working depends on having multiple agents, let's start our simulation off with 15 agents. Use this code to replace the old initialisation:

```
# add x agents
    for x in range(15):
        world.agents.append(Agent(world))
```

Create display code that can show the current parameter values you need and that can, for each agent, identify its immediate "neighbours" and gather the average heading, the centre position etc as needed

To start off, we must add our variables that represent radiuses around each agent. Alter constructor of Agent to contain these variables:

```
# Cohesion, separation and alignment steering behaviours
        self.cohesion = 200.0
        self.separation = 80.0
        self.alignment = 400.0
```

We will also be adding two more lists to the constructor. One list to store the neighbours within the cohesion radius and one list to store the agent within the separation radius.

```
        self.neighbours = []
        self.closeby = []
```

We can add a function that will find all of the agent within a given range through this code:

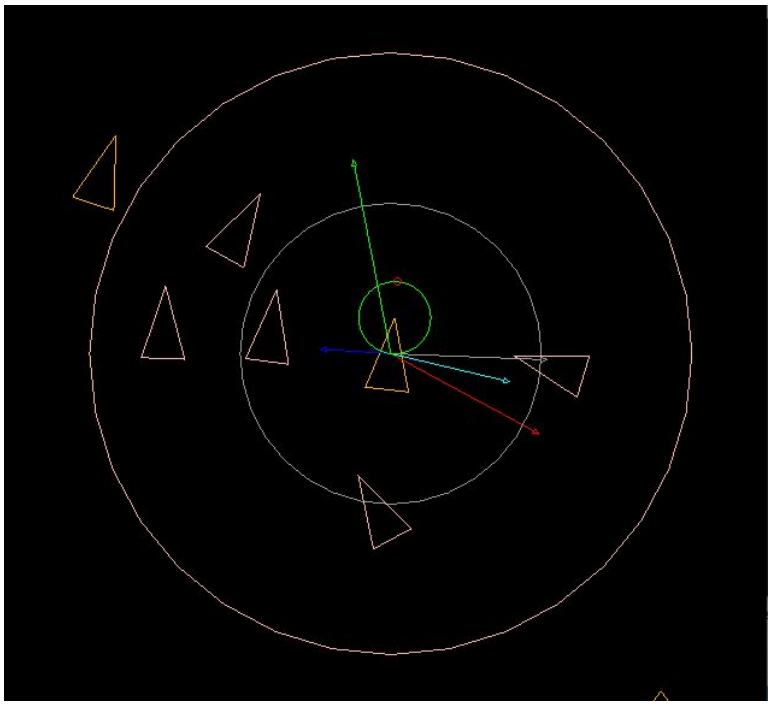
```
def get_agents_in_radius(self,radius):
        # Returns a list of all the agents within a given radius of self
        neighbours = []
        for agent in self.world.agents:
            if (agent != self):
                agent_to_self = self.pos - agent.pos
```

```
                dist = agent_to_self.length()
                if (dist < radius):
                    neighbours.append(agent)
        return neighbours
```

These two variables will be updated every frame in the update function(). Call get_agents_in_radius in the update function by adding this code:

```
# update emerge seperate lists
        self.neighbours = self.get_agents_in_radius(self.cohesion)
        self.closeby = self.get_agents_in_radius(self.separation)
```

Now, in our render function, we need to draw multiple calculations that are being made to help us visualise what is happening.



For the provided image,

Green line = The average heading from all of the neighbours
Blue line = Line to the centre of the neighbour group (average position)
Cyan line = Line drawing away from the closest agent in the closeby list
Red line = Current combined force being applied
Grey line = Net (desired) change force
Pink circle = Cohesion circle
Grey circle = Separation circle

Add this code to the render code to have these graphics display:

```
if self.show_info:
            s = 0.5 # <-- scaling factor
            ### PHYSICS
            # force
            egi.red_pen()
            egi.line_with_arrow(self.pos, self.pos + self.force * s, 5)
            # net (desired) change
            egi.grey_pen()
            egi.line_with_arrow(self.pos, self.pos+ (self.force+self.vel)
* s, 5)

            ### EMERGE
            # cohesion circle
            egi.pink_pen()
            egi.circle(self.pos,self.cohesion)
            # separation circle
            egi.grey_pen()
            egi.circle(self.pos,self.separation)
            if (self.neighbours):
                # Average Heading Line
                egi.green_pen()
                ave_heading = self.neighbour_average('heading')
                egi.line_with_arrow(self.pos, self.pos + self.alignment *
ave_heading * s,5)
                # Average Position Line
                egi.blue_pen()
                ave_position = self.neighbour_average('pos')
                egi.line_with_arrow(self.pos, ave_position,5)
            if (self.closeby):
                egi.aqua_pen()
                closest_agent_pos = self.closest(self.closeby).pos
                target = 2 * self.pos - closest_agent_pos
                egi.line_with_arrow(self.pos, target,5)
            # Become pink if in neighbourhood of agents[0]
            for agent in self.world.agents:
                if agent in self.neighbours:
                    agent.color = 'PINK'
                else:
                    agent.color = 'ORANGE'
```

However, notice that this will apply to every single agent? We don't want that, modify the main.py code to only have the first agent in world.agent displaying this information

```
elif symbol == KEY.I:
        world.agents[0].show_info = not world.agents[0].show_info
```

Also notice that this code requires one additional function to calculate the middle position and average heading? We can do this in a nice sneaky way by inputting the attribute we want to calculate the average of asd a string into the function. This works well because the middle position is calculated as the average position for all agents in the neighbourhood!

```
    def neighbour_average(self,attr):
        # Gets the average of a given attribute from all of the neighbour
        average = Vector2D(0,0)
        for agent in self.neighbours:
            average += getattr(agent,attr)
        average /= len(self.neighbours)
        return average
```

Include basic wandering behaviours and include cohesion, separation and alignment steering behaviours. (Use weighted-sum to combine all steering behaviours)

There are 4 main forces that are being applied for each agent in a neighbourhood:

The base force is the wander function that we have implemented from the previous labs. This doesn't need to be modified, however the variables may be modified later down the track. Modify your calculate function to look like this:

```
    def calculate(self,delta):
        '''calculate the current steering force'''
        # Wander by default
        force = self.wander(delta)
        return force
```

The first force is the agent heading towards the average heading point. We can call this function emerge()

```
    def emerge(self):
        # Approaches the average heading
        ave_heading = self.neighbour_average('heading')
        target = self.pos + ave_heading * self.alignment
        return self.seek(target)
```

This function works by using the average function I previously introduced to you, and expanding it out by the length of the alignment. The reason this is done is because the further the alignment is, the further the target will be from the agent and hence the straighter they will travel.

Hence we will call this in our calculate function as so:

```
def calculate(self,delta):
        '''calculate the current steering force'''
        # Wander by default
        force = self.wander(delta)
        # Update steering force to neighbours average force if they exist
```

```
        if (self.neighbours):
            self.force += self.emerge()
        return force
```

Notice that we add this force rather than overwriting it. This is because we create an overall weighted-sum force. The result of this is a more fluid movement and the agents in a flock of agents will move together.

Nice, the agents are starting to fly in packs. However, we can improve this… the only problem of this is that the agents towards the edge of the cohesion range won't stay in for long.

Hence the next force is to try and approach the centre of the neighborhood. As previously mentioned, we use the neighbour_average function to calculate the centre.

```
    def approach_centre(self):
        # Approaches the centre
        ave_position = self.neighbour_average('pos')
        return self.seek(ave_position)
```

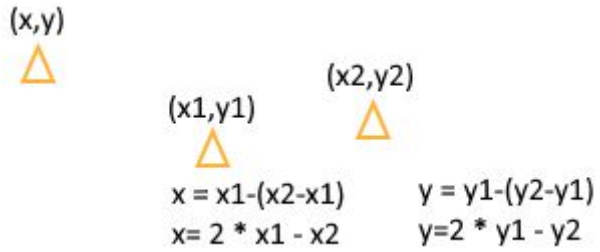
Again, update the calculate function too.

```
    def calculate(self,delta):
        '''calculate the current steering force'''
        # Wander by default
        force = self.wander(delta)
        # Update steering force to neighbours average force if they exist
        if (self.neighbours):
            self.force += self.emerge()
            self.force += self.approach_centre()
        return force
```

Whilst this works, they all begin to ride over the top of each other because they are all steering towards the centre of the pack.

This is where our last force is applied. This force is what our separation variable is for. It will aim to maintain distance between each agent. We can calculate the direction of force through the vector connecting an agent and the closest agent inside of the separation radius.

The calculation is as such:

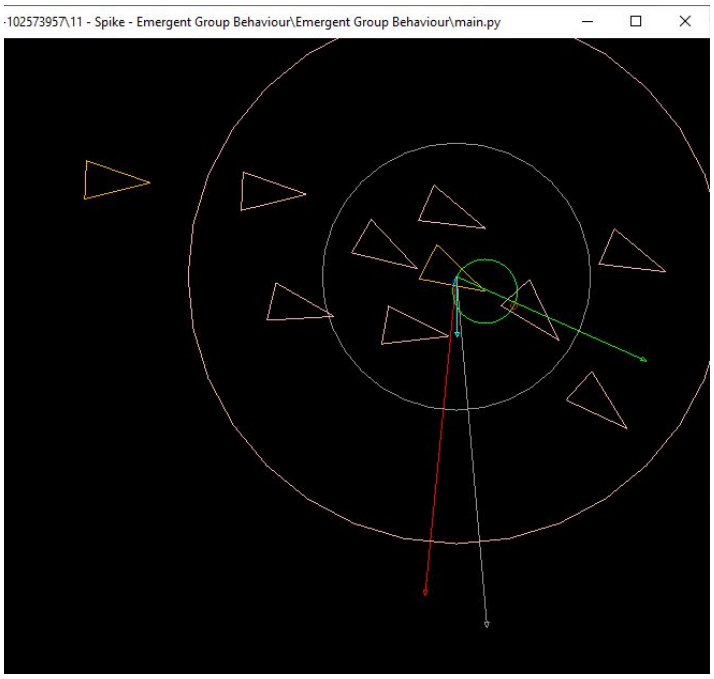



```
    def separate(self):
        # Moves away from nearby agent
        closest_agent_pos = self.closest(self.closeby).pos
        target = (2 * self.pos - closest_agent_pos)
        return self.seek(target)
```

We can use our closest() function that I created from last lab:

```
    def closest(self,list):
        # Returns the closest object to self from a given list of objects
        dist_to_closest = None
        closest = None
        # Gets the closest object to hide behind
        for obj in list:
            to_obj = obj.pos - self.pos
            dist = to_obj.length()
            if (closest is None or dist < dist_to_closest):
                closest = obj
                dist_to_closest = dist
        return closest
```

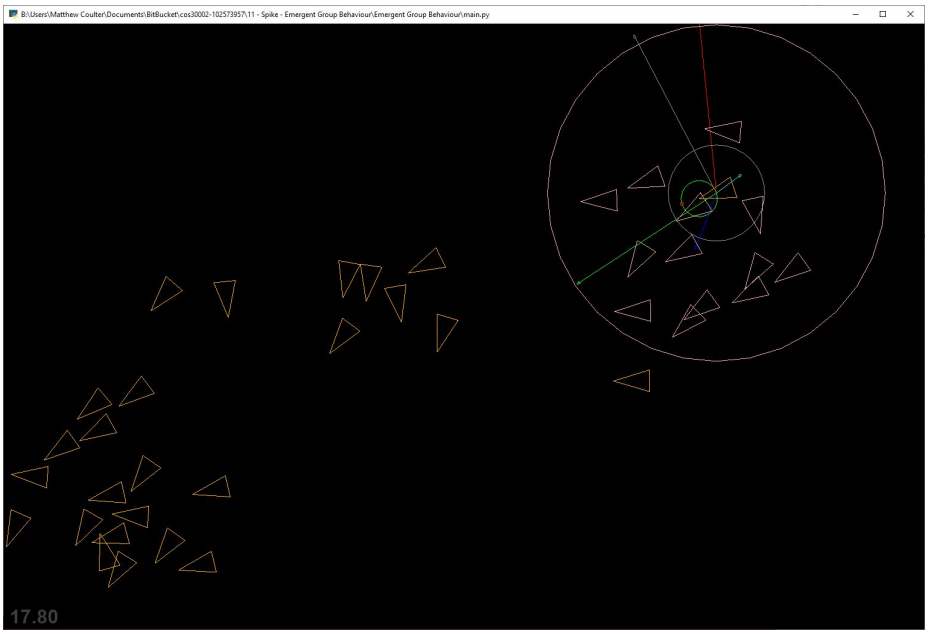
Whilst this works in theory, there is one problem that comes from this which is that they keep moving left and right as the closest agent to evade changes. The result if this is that not much separation happens.



They seperate enough, but not enough to be the desired value that we need.

This is because of a conflict between the force that is pulling them in and the force that is pulling them away from each other; they basically cancel eachother out. To fix this, we can have a condition that only applies to centre pulling force when it won't bring them into other agents. Fortunately, there is a simple fix to this: we can modify the calculate function to only pull them towards the centre when they don't have any agents within their separation radius. Modify the calculate function to look like this:

```
    def calculate(self,delta):
        '''calculate the current steering force'''
        # Wander by default
        force = self.wander(delta)
        # Update steering force to neighbours average force if they exist
        if (self.neighbours):
            self.force += self.emerge()
        if (self.closeby):
            self.force += self.separate()
        if (self.neighbours and not self.closeby):
            self.force += self.approach_centre()
        return force
```



Support the adjustment of parameters for each steering force while running

Because there are a lot of attributes to control, we can implement a similar system to the one used for AGENT_MODES. We will call it MODIFIERS, we will also have a CURRENT MODIFIER global variable that stores the result of which key was pressed. Add them to main as such:

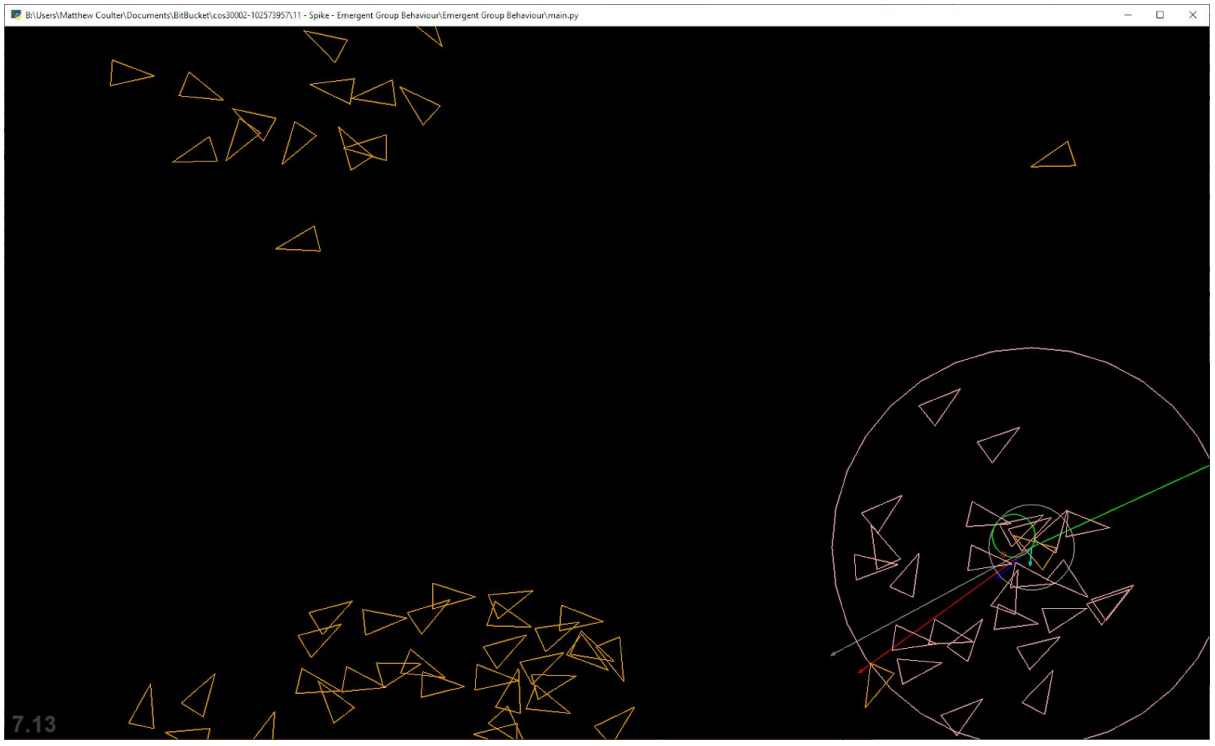
```
MODIFIERS = {
    KEY.C: 'cohesion',
    KEY.S: 'separation',
    KEY.A: 'alignment'
}

CURRENT_MODIFIER = 'cohesion'
```

Now we can loop through the dictionary to check for key presses

```
def on_key_press(symbol, modifiers):
    global CURRENT_MODIFIER
    elif symbol in MODIFIERS:
        CURRENT_MODIFIER = MODIFIERS[symbol]
    elif symbol == KEY.NUM_ADD:
        for agent in world.agents:
            attr = getattr(agent,CURRENT_MODIFIER)
            setattr(agent, CURRENT_MODIFIER, attr + 20.0)
    elif symbol == KEY.NUM_SUBTRACT:
        for agent in world.agents:
            attr = getattr(agent,CURRENT_MODIFIER)
            setattr(agent, CURRENT_MODIFIER, attr - 20.0)
```

What we found out:

The results work rather well. As we begin to initialise more agents at the beginning, they immediately begin forming their flocks and flying around. Their alignment is pretty good however I think there is a bit too much swerving that happens from the agents trying not to hit each other. Another disadvantage is that as the alignment variable is increased, the agents don't begin to straight out as much as i expected, especially when there are more agent in the world.



Other than that they definitely form groups well and respond accurately in accordance to the separation and cohesion variables we modified.
Something that helps with this is reduces the wander_jitter variable increasing the wander_distance as this in turn reduces the amount of left and right movement that happens for each agent. However it does not completely solve it.