Spike: Spike 4
Title: Emergent Group Behaviour

Author: Duy Anh Vuong - 102603197

Goals / deliverables:

Create a group agent steering behaviour simulation that can demonstrate distinct modes of emergent group behaviour. In particular, the simulation must:

• Include cohesion, separation and alignment steering behaviours
• Include basic wandering behaviours
• Use a weighted sum to combine all steering behaviours
• Support the adjustment of parameters for each steering force while running
- Code seen in /11/Spike – Emergent Group Behaviour

Technologies, Tools, and Resources used:

- Given code for Lab 08 and Lab 09
- Canvas Materials
- Visual Studio Code
- Python
- "The Nature of Code" – Chapter 6: Autonomous Agents (https://natureofcode.com/book/chapter-6-autonomous-agents/)

Tasks undertaken:

This task requires repurposing the given Autonomous Moving Agents

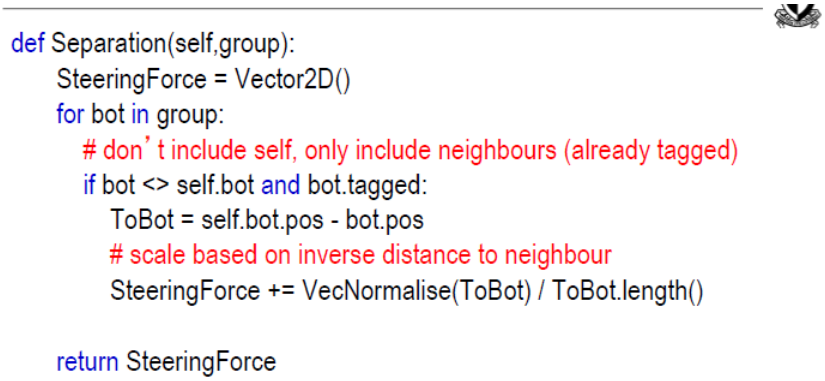
Initially, I cleaned the unnecessary lines for this task to avoid run-time conflicts as well as improve the performance of the program.

After all cleaned, I began to implement the TagNeighbors() method for the agents, based on the suggestion from the given PDF.

```
def tagNeighbors(self):
    for agent in self.world.agents:
        agent.tagged = False
        vt_to = self.pos - agent.pos
        gap = self.tag_rad + agent.bRadius
        if(vt_to.lengthSq()<gap**2):
            agent.tagged = True
```

Moving on to the first mode of emergent group behaviour – Separation(). The idea is to make the agents in one group flee from others. While implementing this behaviour, I noticed that if I write the code as suggested in the document, the behaviour failed to work, which took me several days to conclude that the "return" value in the

document was actually the “desired” vector, not the actual “steering” force.



```
def Separation(self,group):
    SteeringForce = Vector2D()
    for bot in group:
        # don't include self, only include neighbours (already tagged)
        if bot <> self.bot and bot.tagged:
            ToBot = self.bot.pos - bot.pos
            # scale based on inverse distance to neighbour
            SteeringForce += VecNormalise(ToBot) / ToBot.length()

    return SteeringForce
```

Based on that conclusion, I went on alter the code, by normalising the calculated vector, multiplying it by max_speed and finally subtract it by the velocity to get the final steering force.

```
def separation(self):
    force = Vector2D()
    for agent in self.world.agents:
        if(self!=agent and agent.tagged):
            toSelf = self.pos - agent.pos
            force += toSelf.normalise()/float(toSelf.length())
    force.normalise()
    force *= self.max_speed
    return force - self.vel
```

I am not so sure if my implementation was legitimate, but it worked (the result will be shown later 😊).

After that, the Alignment() behaviour would be implemented. The idea is to align the heading of all members of the group to the average heading. Indeed, the document only shows how to get the heading without mentioning how to apply the force:

Alignment

```
def Alignment(self, group):
    AvgHeading = Vector2D()
    AvgCount = 0

    for bot in group:
        if bot <> self.bot and bot.tagged:
            AvgHeading += bot.heading
            AvgCount += 1

    if AvgCount > 0:
        AvgHeading /= float(AvgCount)
        AvgHeading -= self.bot.heading

    return AvgHeading
```

Hence, I decided to normalise the heading (as a vector) and multiply it by the max speed to apply the force:

```
def alignment(self):
    heading = Vector2D()
    count = 0
    for agent in self.world.agents:
        if(self is not agent and agent.tagged):
            heading += agent.heading
    if count > 0:
        heading/=float(count)
        heading -= self.heading
    return (heading.normalise()*self.max_speed)
```

The final key task is to write the Cohesion() behaviour, the idea is to find the center position of a group then let the agent seek the target. I followed the instruction from the document and it worked as expected (again, the result is shown later 😊):

```
def cohesion(self):
    center_mass = Vector2D()
    force = Vector2D()
    count = 0
    for agent in self.world.agents:
        if(self is not agent and agent.tagged):
            center_mass += agent.pos
            count +=1
    if count>0:
        center_mass/=float(count)
        force = self.seek(center_mass)
    return force
```

Combine it with Wander(), we have the final Weighted Sum behaviour (could be considered as Weighted Truncated Sum as the total force will be truncated to max force in the update method):

```
self.tagNeighbors()
force = Vector2D()
force += self.wander(delta)
force += self.separation()*self.separation_amt
force += self.alignment()*self.alignment_amt
force += self.cohesion()*self.cohesion_amt
```

To control and weight how much each of the forces contribute to the resulted force, I multiply them with amount indexes that are controlled by the World Class:

```
class World(object):

    def __init__(self, cx, cy):
        self.cx = cx
        self.cy = cy
        self.target = Vector2D(cx / 2, cy / 2)
        self.hunter = None
        self.agents = []
        self.obstacles = []
        self.paused = True
        self.show_info = True
        self.cohesion = 1.0
        self.alignment = 1.0
        self.separation = 1.0
```

In World class

```
    self.cohesion_amt = self.world.cohesion
    self.alignment_amt = self.world.alignment
    self.separation_amt = self.world.separation
```

__init__

```
    def update(self, delta):
        self.separation_amt = self.world.separation
        self.cohesion_amt = self.world.cohesion
        self.alignment_amt = self.world.alignment
```

update()

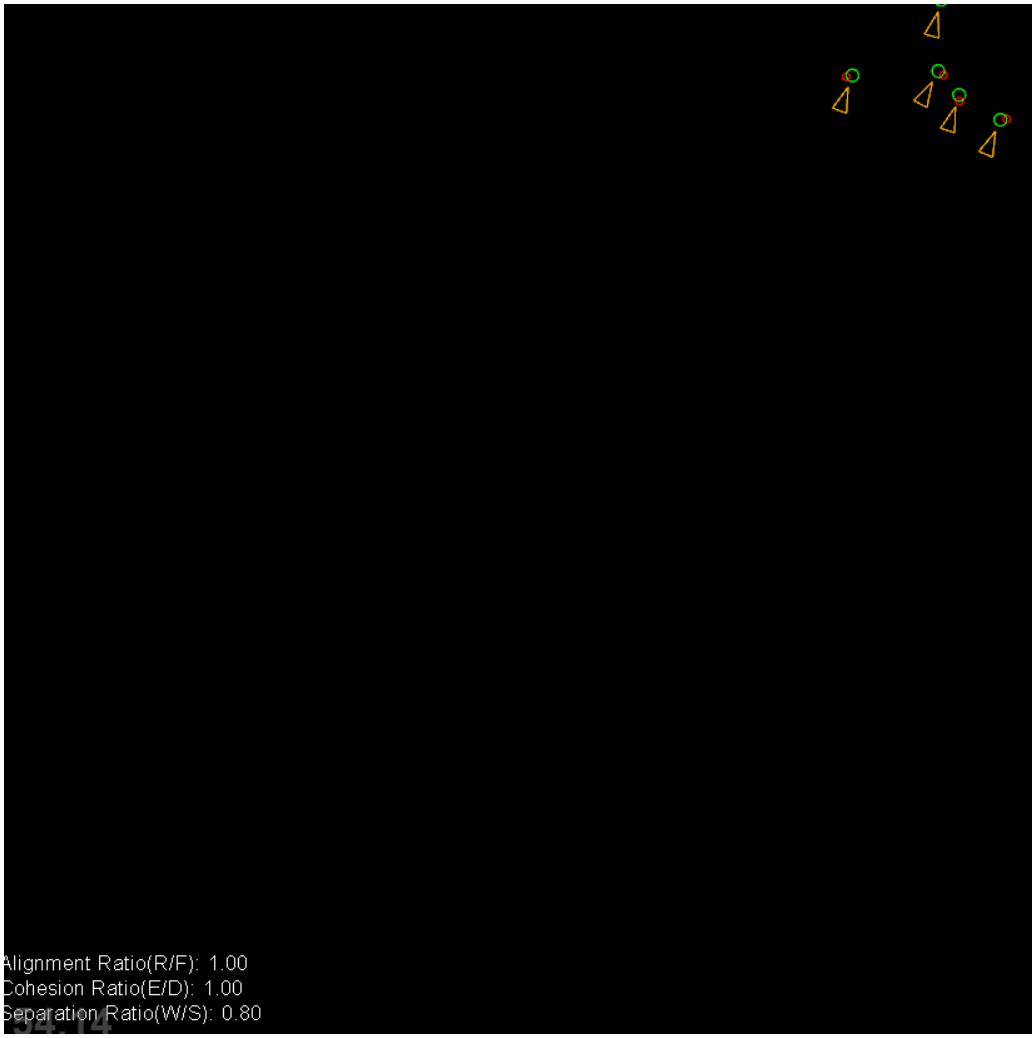
In main.py, I added some keyboard inputs to modify the mentioned parameters:

```
elif symbol == KEY.W:
    world.separation += 0.10
elif symbol == KEY.S:
    world.separation -= 0.10
elif symbol == KEY.E:
    world.cohesion += 0.10
elif symbol == KEY.D:
    world.cohesion -= 0.10
elif symbol == KEY.R:
    world.alignment += 0.10
elif symbol == KEY.F:
    world.alignment -= 0.10
```

All set! The final thing to do is to render the parameters to the terminal:

```
    egi.red_pen()
egi.text_at_pos(0,20,"Separation Ratio(W/S): " + str("%.2f" % self.separation))
egi.text_at_pos(0,40,"Cohesion Ratio(E/D): " + str("%.2f" %self.cohesion))
egi.text_at_pos(0,60,"Alignment Ratio(R/F): " + str("%.2f" %self.alignment))
```

world.render() method



Result

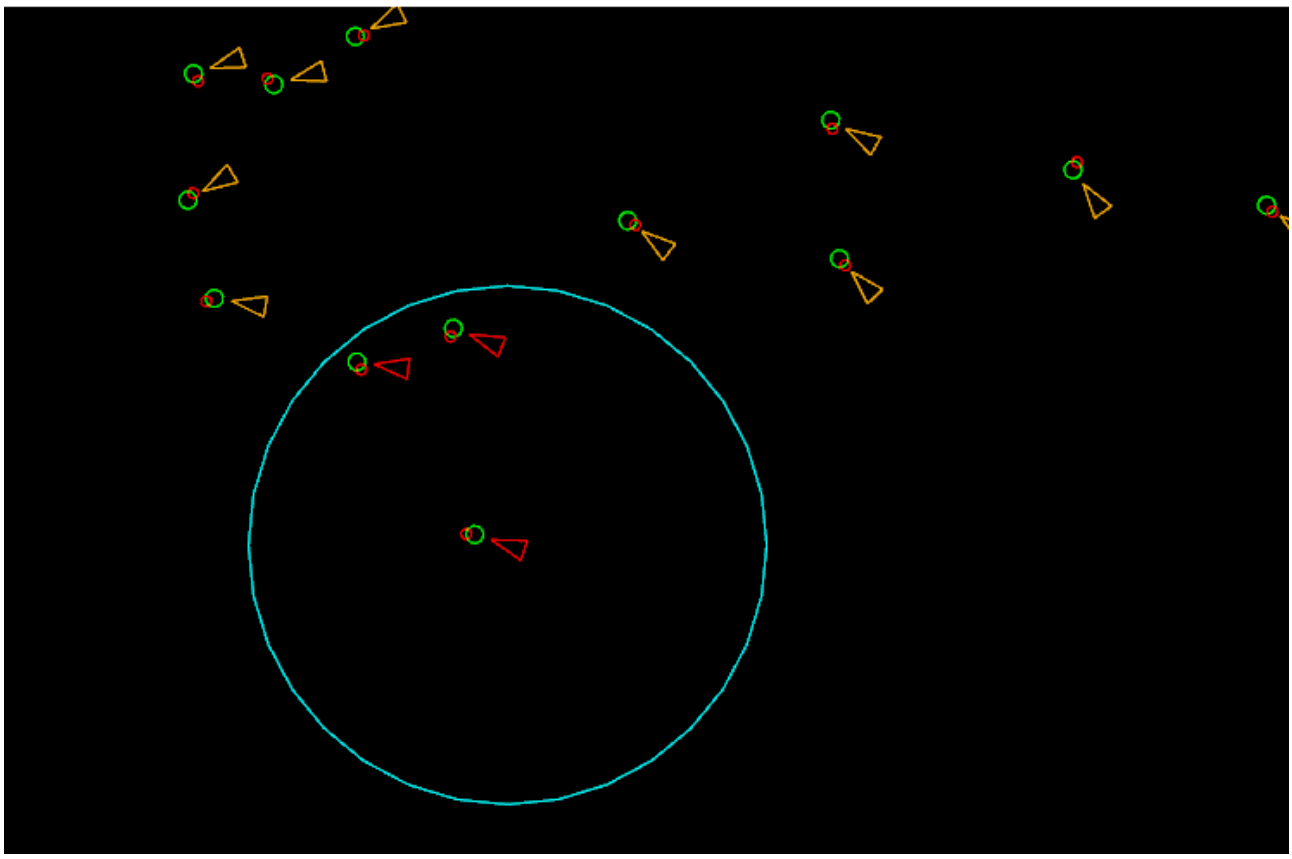
Also, I drew the tag radius around the final agent in the list for observation. The reason is that this agent will always be considered tagged at the end of each rendered frame, and as I set the “tagged” agents’ color to red, its tagged agents will also be colored the same:

```
    def render(self, color=None):
        ''' Draw the triangle agent with color'''
        # draw the path if it exists and the mode is follow
        # draw the ship
        egi.set_pen_color(name=self.color)
        if(self.tagged):
            egi.set_pen_color(name='RED')
```

agent.render() method

```
        egi.aqua_pen()
        last_a = self.agents[len(self.agents)-1]
        egi.circle(last_a.pos,last_a.tag_rad)
```

world.render() method

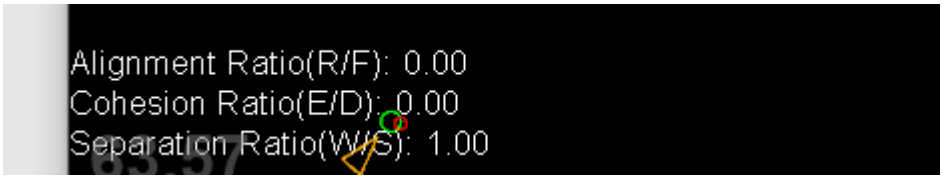


Terminal

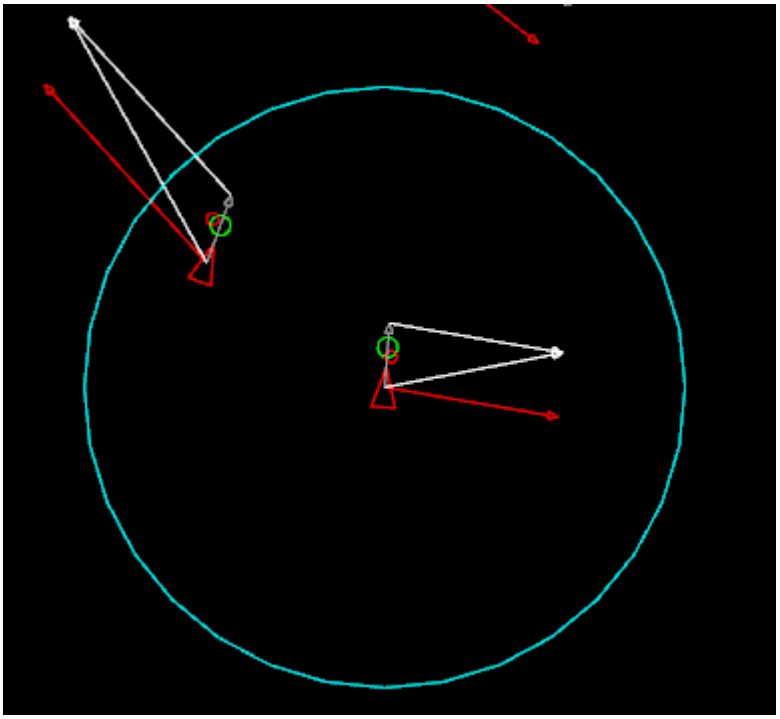
What we found out:

1. Separation:

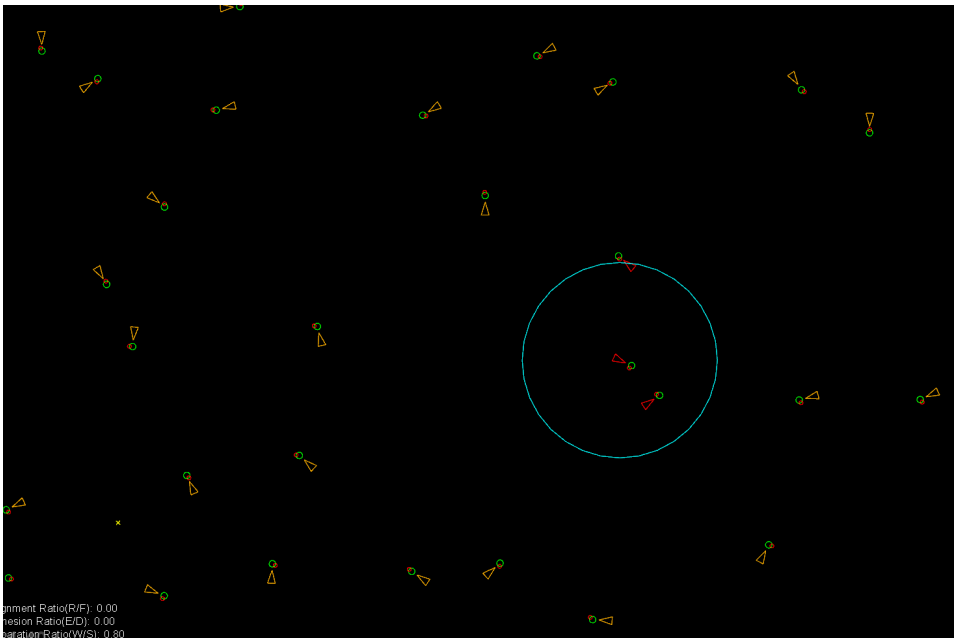
   To observe this behaviour, I zeroed **alignment** and **cohesion** amount:



   As can be seen in the following figure, the agents will keep a certain distance from each other by steering away from its group members:



2 agents are steering away from others in its group



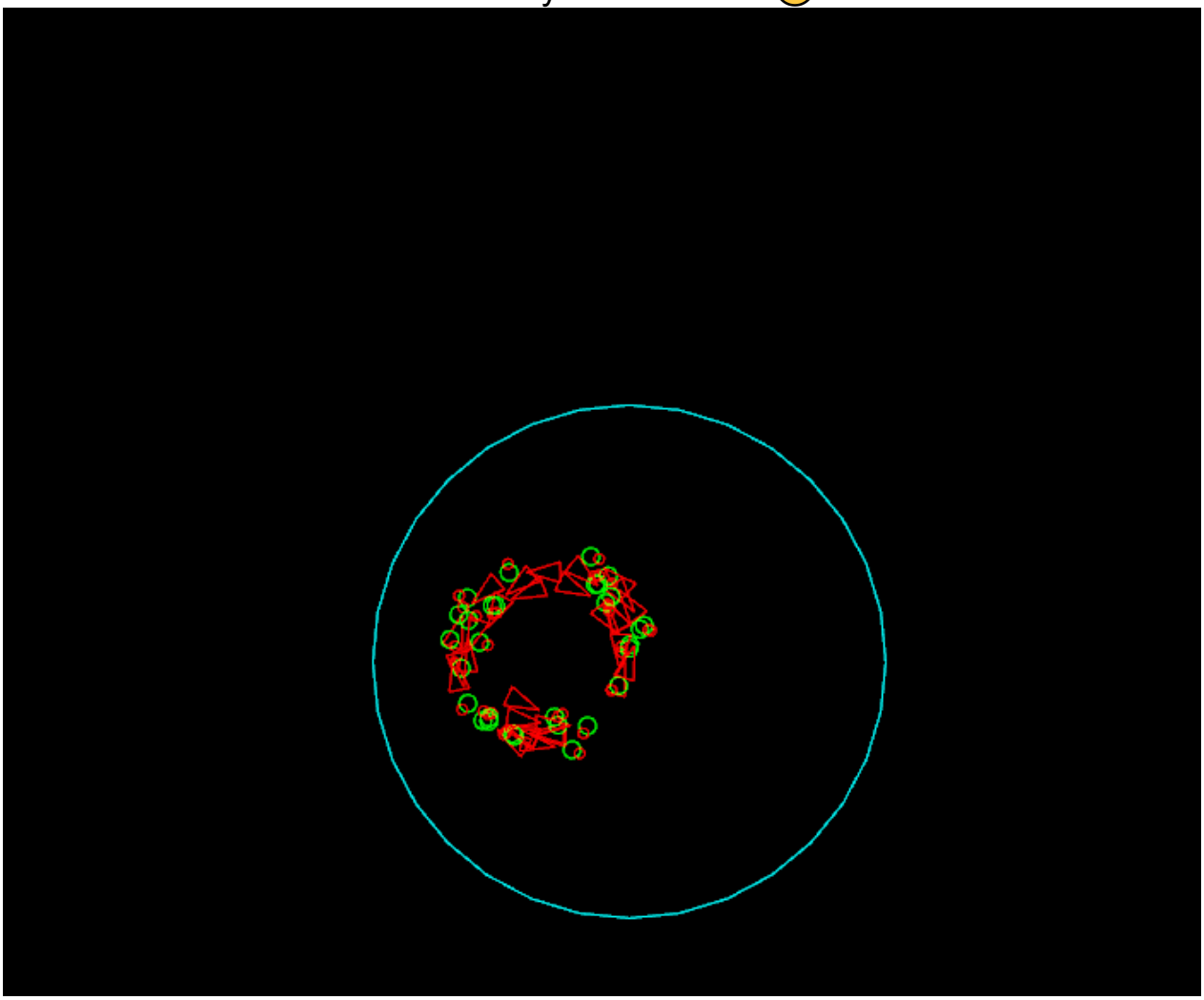
The agents are “distancing” (Covid-19 flashback 😊)

In my view, separation is used in group behaviour for avoiding collision among agents, as it will generate a force to “pull” the agent out of the crowd.

2. Cohesion:

   Again, to see how cohesion individually works, I zeroed other amounts:

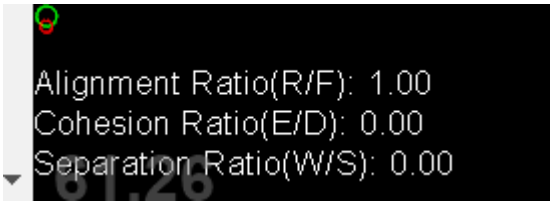
The observed behaviour is, well, very interesting. The agents will keep seeking the centre point, yet as the centre point changes over time, it results in a circulation which is very fun to watch 😊:



Agents circulating the centre mass

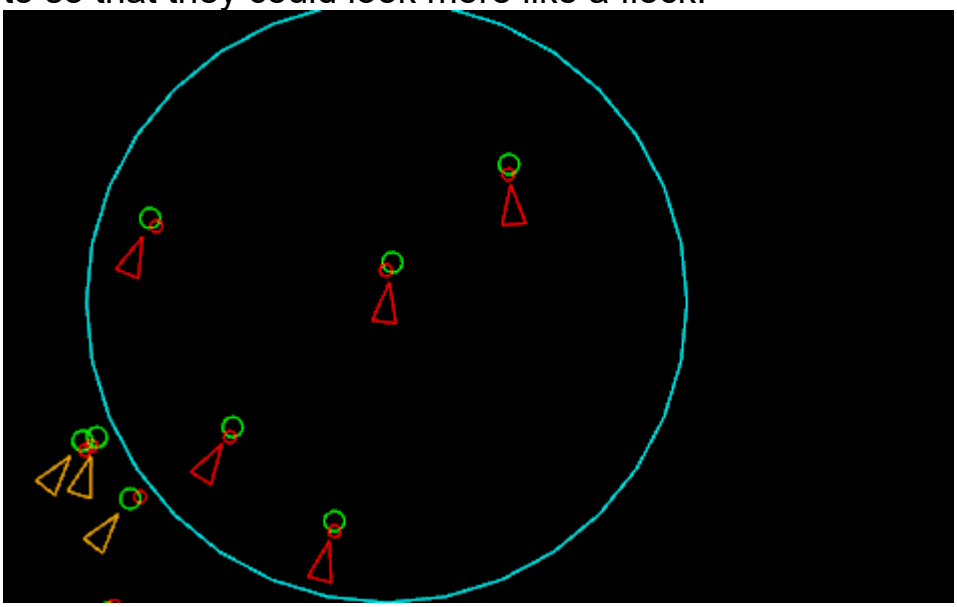
From my perspective, Cohesion is used to keep the members inside the radius of the group, as it will “pull” the agents into the centre of the crowd.

3. Alignment:



Repeat the same step to observe

Alignment alone gives the agents the same direction for them to head to so that they could look more like a flock:

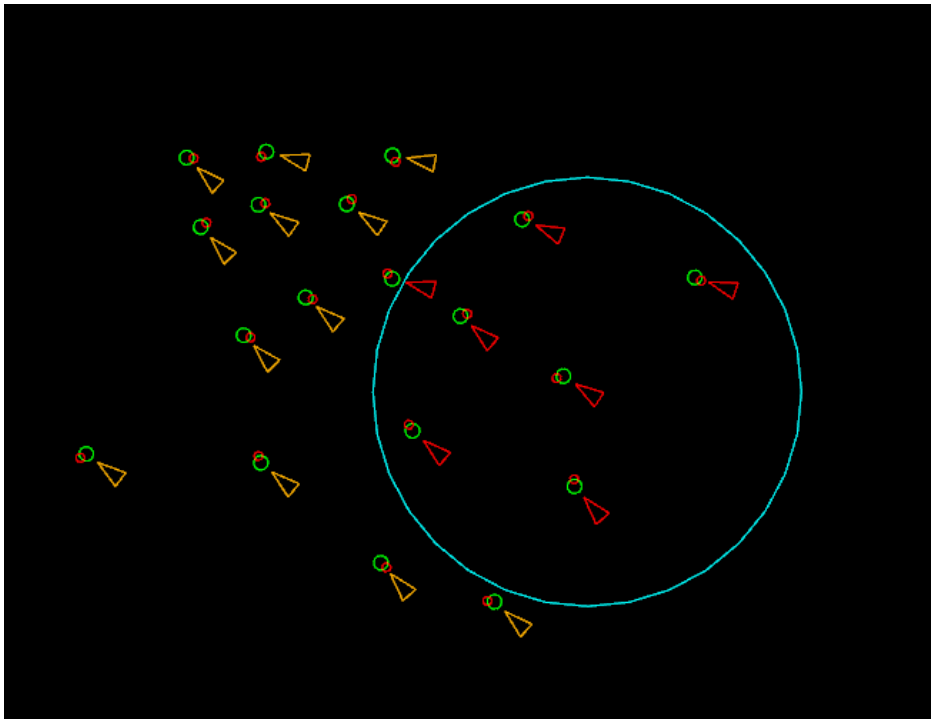


Agents trying to move in the same direction

From the observation, I suppose that Alignment is used to neutralize cohesion and separation, as it will prevent the collision of agents and keep them in a group by driving them along a uniform heading.

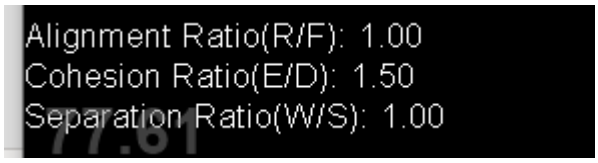
4. Weighted Sum and Behaviour Combination:

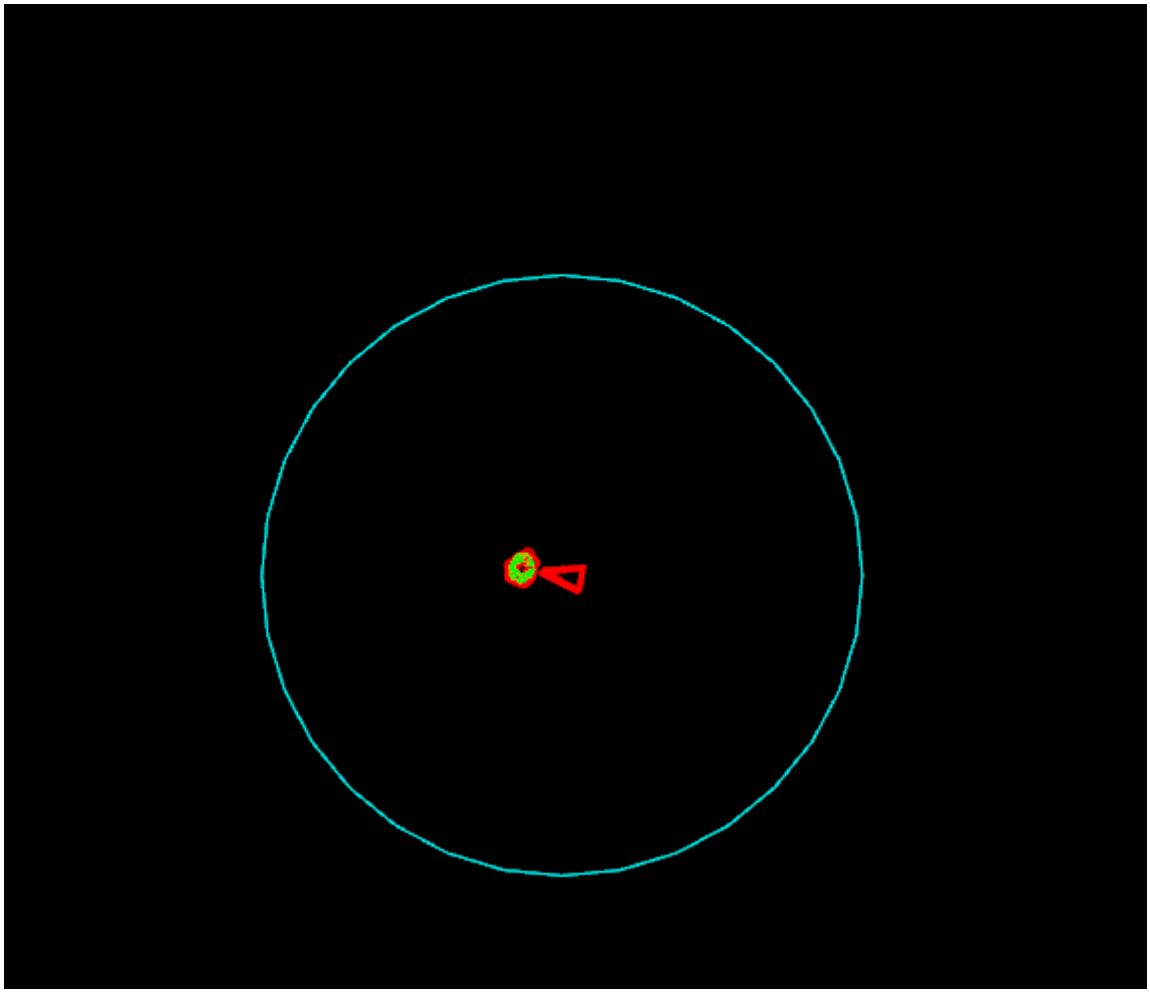
When combined with the Wander() behaviour, we have a group of randomly moving agents. When they are weighted the same in the calculation, the result shows a rather neat moving group of agents:



But what happened when the parameters are altered? I will explain shortly with a few words for each (as the report has been too wordy).

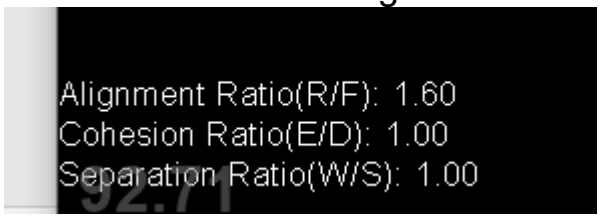
a. Heavier-weighted Cohesion: The crowd will collide at the center and behave the same.

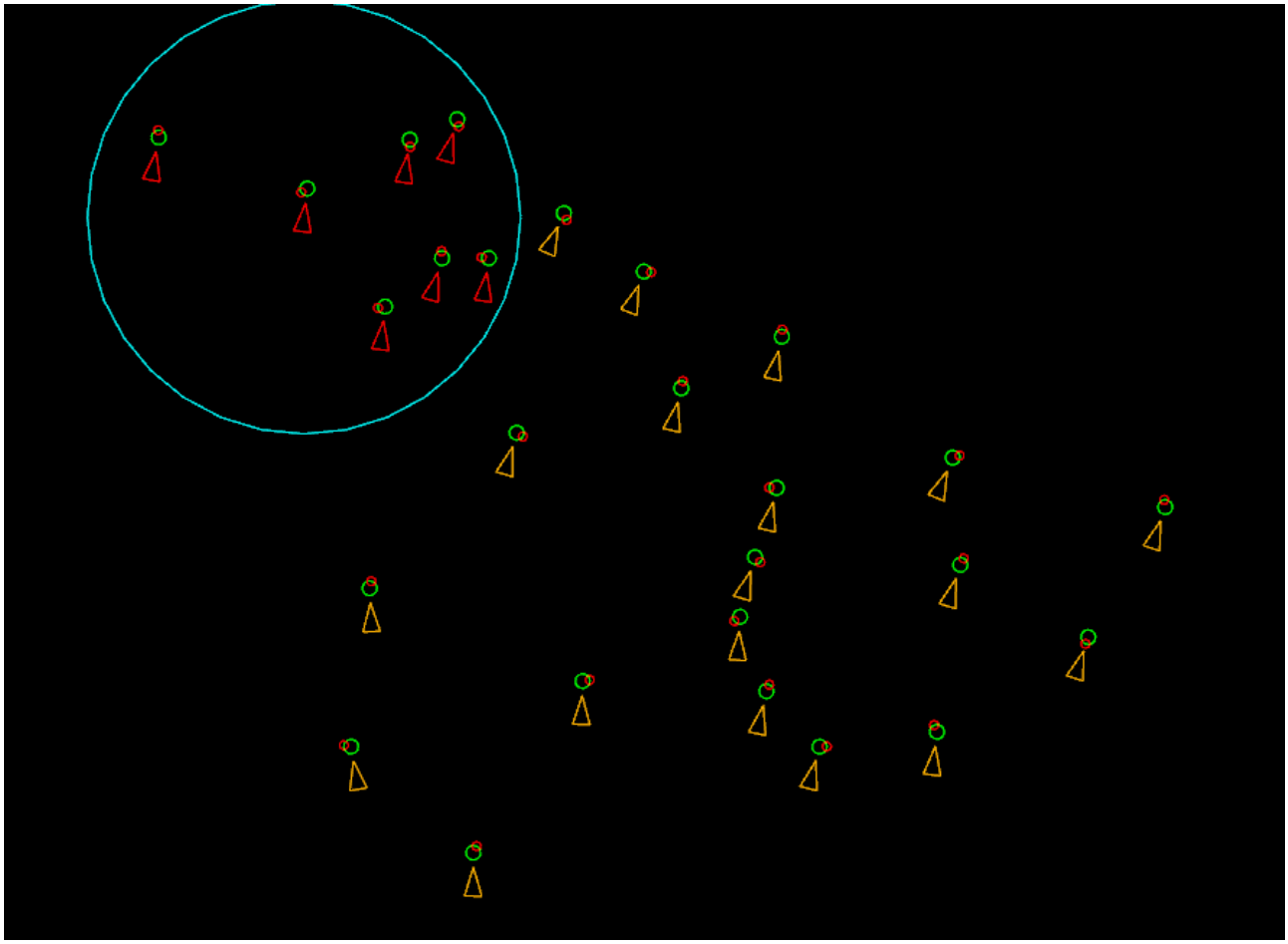




Agents overlapping each other

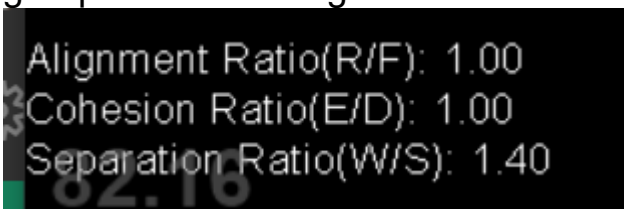
b. Heavier-weighted Alignment: The agents will behave more consistently with less “steering”

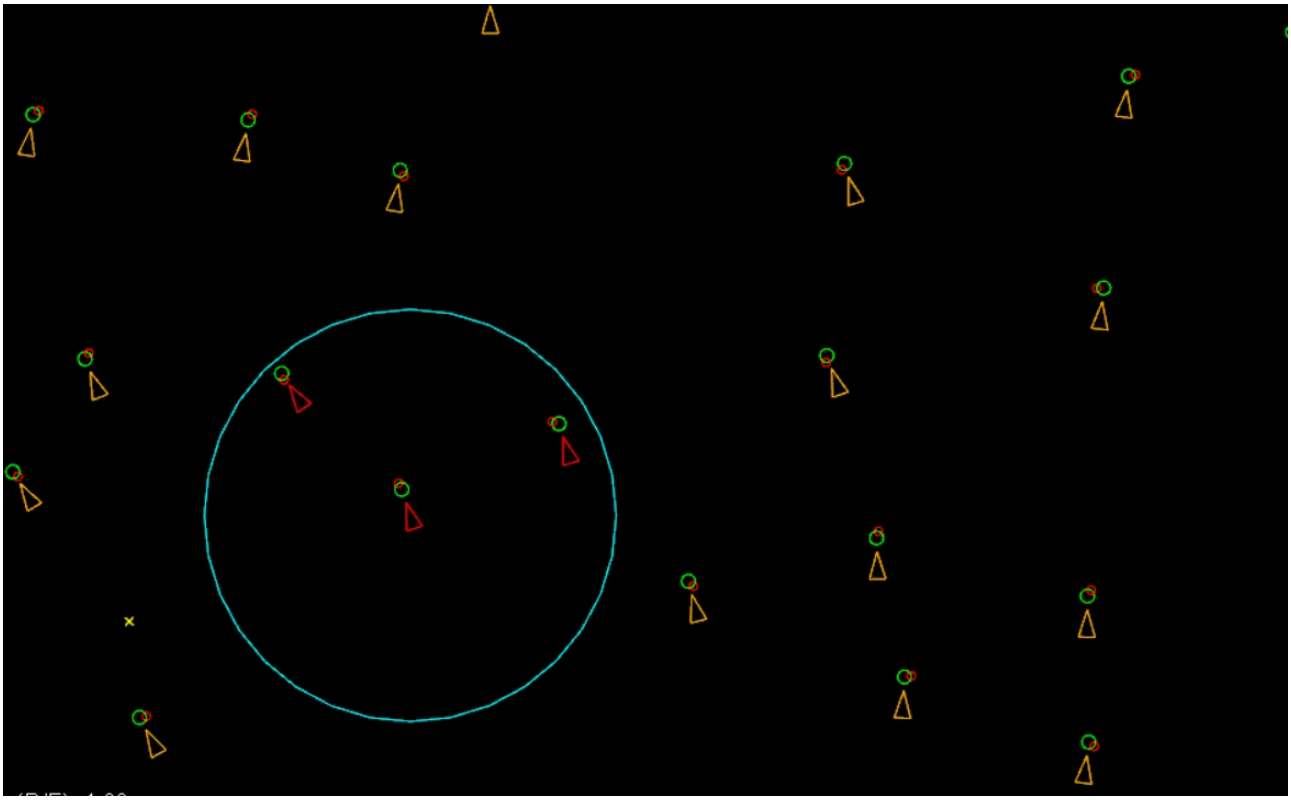




Agents' headings are nearly the same

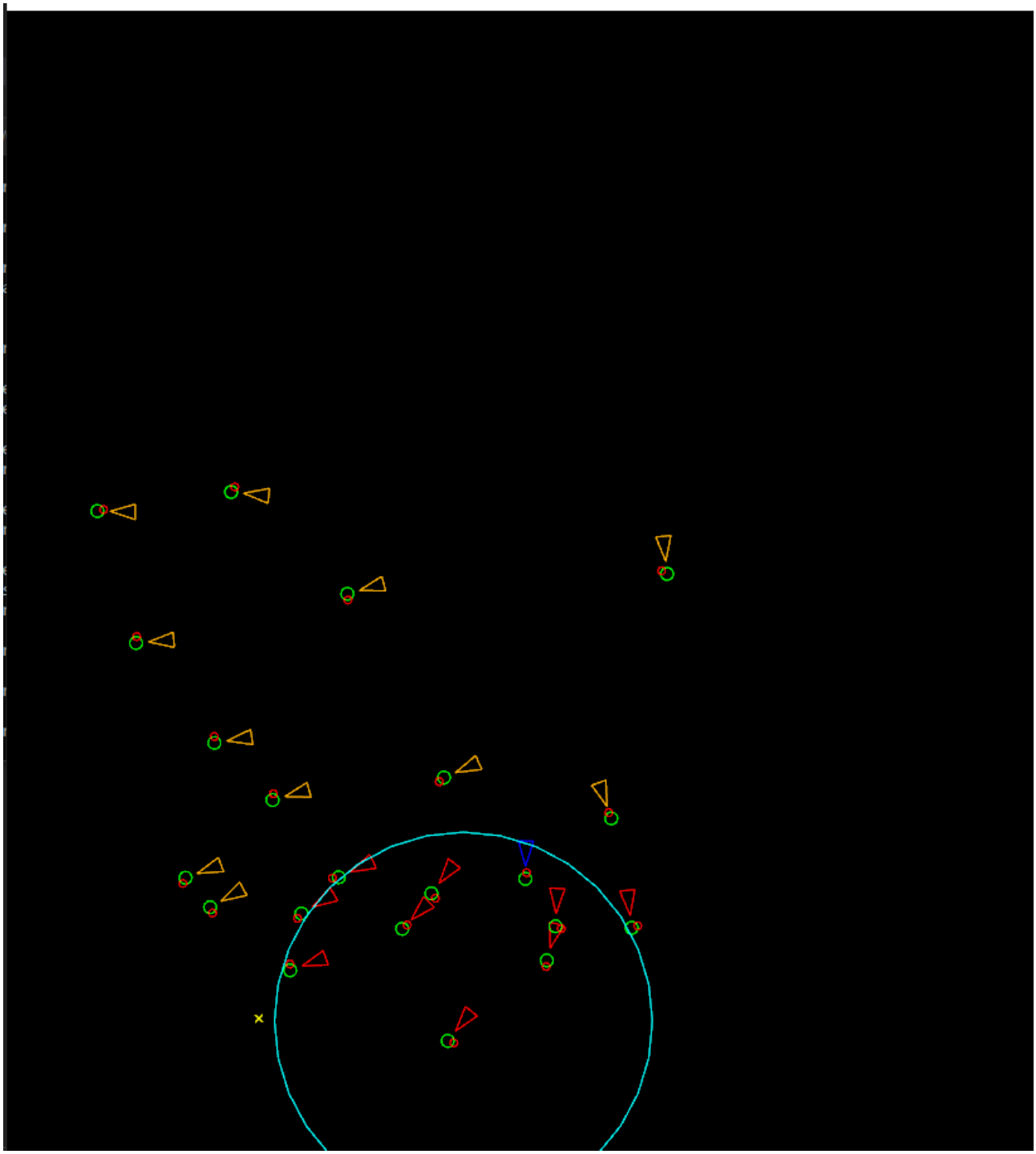
c. Heavier-weighted Separation: The agents will behave in a more separated way propotional to the ratio, as well as there will be more groups with fewer agents





More groups with fewer agents

Extensions



Hunter agent that other agents flee from

Open issues/risks:

- There are extensions that could be added to the program, yet as I invested too much time into bug-spotting, I failed to extend it
- If the cohesion is weighted too heavily, agents will surely overlap, yet on the other hand if separation is weighted in the same way, the agents' movement will not be uniform. Hence, I failed to find the perfect amount to erase the mentioned weaknesses.