Boid Simulation Using Python

Department of Information & Communication Tecnhology

Advanced Programming Lab

ICT – 3166

PRESENTED BY

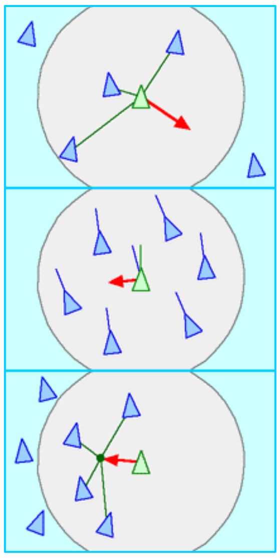
Malab Sankar Barik	190953120	Batch 4
Agastya Gummaraju	190953092	Batch 4
Anushka Bhattacharyya	190953102	Batch 4

Introduction

Every animal that moves in a group has some kind of sense that dictates its movements within the crowd. With each step or each flap of the wings, it's always taking decisions about its movement that might be conscious or unconscious. What direction to go to now, to avoid colliding into another being? What speed to adjust to, to not interrupt the flow of the group? These seemingly simple decisions have some complexity to them that makes them fascinating to study. In flocks of birds for example, think of how each bird monitors its movement without the knowledge of the whole flock. Yet still its flight helps to generate the pattern of the flock.

In 1986, Craig Reynolds picked up this exact problem to see if he could actually simulate this behaviour artificially by using bird-oid objects (or boids). Here, rather than programming the whole flock movement, he decided to program each individual boid with some set of rules to follow. This in turn led to an emergent behaviour of the group of boids that was, to an extent, a believable simulation of an actual flock of birds.

Reynolds stipulated three basic rules for the movement of each boid that would control its movement and steering capabilities with respect to its nearby flockmates.



1. Avoidance: Steering away from local flockmates enough to avoid crowding and collision with nearby flockmates.

2. Adherence: Adhering to the structure or flow of the flock and aligning itself accordingly. They also try to match each other's velocity to maintain the flow of the flock.

3. Coherence: Steering towards the average position of nearby flockmates/immediate neighbours to maintain cohesiveness of the flock.

Reynolds translated the three drives to a set of geometrical equations, where he interpreted the expression 'nearby flockmates' as the boid's immediate surroundings. He found that a boid does not require full knowledge about the position and velocity of every boid in the flock, but only a small subset. The expression 'nearby flockmates', used in the descriptions of the steering behaviours, thus addresses the boid's awareness of another boid and is based on the distance and direction of the offset vector between them. In other words: the boid has a localized perception of the world with a certain perception of distance and field of view and can be visualized as a perception volume shaped like a sphere with a cone removed from the back. It is important to note that, when the boids are in a flock, the individual perception volumes overlap and each individual boid will probably end up in a number of perception volumes.

Avoidance and adherence are complimentary rules, both having the basic purpose of avoiding collisions. The way they work though are different. The avoidance factor solely takes into account the relative positions of the boids and treats them as static objects in any given instance of time. Hence it only monitors the relative positions and helps the boid steer away if it gets too close. On the other hand, adherence is a dynamic factor that only takes their relative velocities and directions in account. This helps the boid to maintain its velocity according to the pack and hence avoid any situation that could cause a collision.

Overall, in light of the boids following the rules with respect to their neighbours, we can notice an emerging pattern of movement in the whole flock. This emergent pattern can be compared to the natural flocks seen around in nature.

Literature Survey

Artificial Life, often called A-Life, is a field of study wherein researchers examine systems related to natural life, its processes, and its evolution, through the use of simulations with computer models, robotics and biochemistry.

It studies the fundamental processes of living systems in artificial environments in order to gain a deeper understanding of the complex information processing that define such systems. These topics are broad, but often include evolutionary dynamics, emergent properties of collective systems (such as our boid simulation program that emulates emergent flock like behaviour observed in birds), biomimicry, as well as related issues about the philosophy of the nature of life and the use of life-like properties in artistic works.

An emergent behaviour or emergent property can appear when a number of simple entities (agents) operate in an environment, forming more complex behaviours as a collective. Emergent behaviours can occur because of intricate causal relations across different scales and feedback, known as interconnectivity.

The emergent property itself may be either very predictable or unpredictable and unprecedented, and represent a new level of the system's evolution. The complex behaviour or properties are not a property of any single such entity, nor can they easily be predicted or deduced from behaviour in the lower-level entities.

Craig Reynolds, in his research paper titled ‘Flocks, Herds and Schools: A Distributed Behavioural Model, published in *Computer Graphics*, **21**(4), July 1987, pp. 25-34, said,” The motion of a flock of birds is one of nature's delights. Flocks and related synchronized group behaviours such as schools of fish or herds of land animals are both beautiful to watch and intriguing to contemplate.

A flock exhibits many contrasts. It is made up of discrete birds yet overall motion seems fluid; it is simple in concept yet is so visually complex, it seems randomly arrayed and yet is magnificently synchronized. Perhaps most puzzling is the strong impression of intentional, centralized control. Yet all evidence indicates that flock motion must be merely the aggregate result of the actions of individual animals, each acting solely on the basis of its own local perception of the world.

One area of interest within computer animation is the description and control of all types of motion. Computer animators seek both to invent wholly new types of abstract motion and to duplicate (or make variations on) the motions found in the real world.

At first glance, producing an animated, computer graphic portrayal of a flock of birds presents significant difficulties. Scripting the path of a large number of individual objects using traditional computer animation techniques would be tedious. Given the complex paths that birds follow, it is doubtful this specification could be made without error. Even if a reasonable number of suitable paths could be described, it is unlikely that the constraints of flock motion could be maintained (for example, preventing collisions between all birds at each frame).

Finally, a flock scripted in this manner would be hard to edit (for example, to alter the course of all birds for a portion of the animation). It is not impossible to script flock motion, but a better approach is needed for efficient, robust, and believable animation of flocks and related group motions.

This paper describes one such approach. This approach assumes a flock is simply the result of the interaction between the behaviors of individual birds. To simulate a flock we simulate the behavior of an individual bird (or at least that portion of the bird's behavior that allows it to participate in a flock). To support this behavioral "control structure," we must also simulate portions of the bird's perceptual mechanisms and aspects of the physics of aerodynamic flight.

If this simulated bird model has the correct flock-member behavior, all that should be required to create a simulated flock is to create some instances of the simulated bird model and allow them to interact.

Some experiments with this sort of simulated flock are described in more detail in the remainder of this paper. The success and validity of these simulations is difficult to measure objectively. They do seem to agree well with certain criteria and some statistical properties of natural flocks and schools which have been reported by the zoological and behavioral sciences.

Perhaps more significantly, many people who view these animated flocks immediately recognize them as a representation of a natural flock, and find them similarly delightful to watch."

He further stated," The model is based on simulating the behavior of each bird independently. Working independently. The birds try both to stick together and avoid collisions with one another and with other objects in their environment. The animations showing simulated flocks built from this model seem to correspond to the observer's intuitive notion of what constitutes "flock-like motion." However it is difficult to objectively measure how valid these simulations are. By comparing behavioral aspects of the simulated flock with those of natural flocks. We are able improve and refine the model. But having approached a certain level of realism in the model. The parameters of the simulated flock can be altered at will by the animator to achieve many variations on flock-like behavior.

Methodology

For the implementation and python code, we have imported a few modules necessary for the computations and graphical representation. Other than that, we have formulated our own classes and methods for the actual working of the project. We have used the following modules: math, rand, pygame and pygame_gui.

The main class here is 'Boid' which is used to create and program each boid. It inherits some methods from the 'Sprite' class in pygme which is used to define basic entities. It has the 'init' method and the 'update' method in it.
In the 'init' method, we define the grid in which the boids will be present, define its general physical attributes. It also defines the minimum radius around which the boid can have neighbouring boids, direction vectors of the boid, the boid's last known position, etc.

Before defining the update method, we have to understand the 'BoidGrid' class as update method employs some methods from 'BoidGrid' class.

This class is the implementation of a spatial partition grid. It has two variables – grid size and a dictionary which stores the information of each cell in the grid about which boid it contains.

- A boid can call the getcell method giving its own location as a parameter, the method on basis of which the method will discern in which cell is the boid present.

- Add method adds the boid to the cell's mapping in the grid.

- Remove method removes the boid from the present cell's mapping when add method adds the boid to its new cell's mapping.

- A boid calls the getnear method to get all boid objects in the cell in which it is present.

Now that we have clarified the working of the 'BoidGrid' class, we will be looking at the update method. It defines some variables for the boid. We will update our grid where we use the 'BoidGrid' methods to check where the boid is and update the grid accordingly.

After this, we update the position of the boid accounting for the normal movement of the boid after processing its neighbours. We also code for avoidance if too close, for adherence to adjust its alignment with its neighbours. After this we code for the updating of position to avoid the edges of the screen by turning according or wrap around the screen if wrap mode is on. In the last line we update the centre of boid which is to be displayed on screen.

Finally, we have the main method of the code. Here we prepare the main simulation window using pygame_gui, create the sliders to control the four factors: adherence, coherence, avoidance and speed.

We add the wrap around and FPS toggle buttons as well. We create the main loop that connects the sliders to the main program and accordingly updates the simulation.

Implementation

Import the necessary modules, and set the GUI parameters to the aforementioned values.

```
from math import pi, sin, cos, atan2, radians, degrees
from random import randint
import pygame as pg
import pygame_gui as pgui


FLLSCRN = True          #True for Fullscreen, or False for Window
BOIDZ = 200             #How many boids to spawn, too many may slow fps
WRAP = True             #False avoids edges, True wraps to other side
SPEED = 150             #Movement speed
WIDTH = 1200            #Window Width (1200)
HEIGHT = 800            #Window Height (800)
BGCOLOR = (0, 0, 0)     #Background color in RGB
FPS = 60                #30-90
SHOWFPS = True          #Frame rate debug
```

Create the ‘Boid’ class. This class contains all of the information of each individual boid, and is derived from the ‘Sprite’ class contained within the Pygame module. Then, define the class’s __init__ method, where we define the properties of all of the boids.

```
class Boid(pg.sprite.Sprite):

    def __init__(self, grid, drawSurf, isFish=False):  #cHSV=None

        super().__init__()  #Calling the Sprite class’s __init__ method
```

Initialise all of the object variables.

```

self.grid = grid #Associating the boid with the spatial grid on the screen

self.drawSurf = drawSurf #The layer on which the boid is drawn, the screen

self.image = pg.Surface((15, 15)).convert()
#15x15 pixel surface is created, and is converted to an image

self.image.set_colorkey(0) #Giving color to each sprite

self.color = pg.Color(0) #Preps color so we can use HSVA

self.color.hsva = (randint(0,360), 90, 90) #Setting the color

pg.draw.polygon(self.image, self.color, ((7,0), (13,14), (7,11), (1,14),
(7,0)))
#Creating the arrow like shape of each individual boid
#The coordinates are of the vertices

self.bSize = 17
#The closest that the neighbouring boids can get to the current boid
without it moving away from its neighbours

```

```
self.orig_image = pg.transform.rotate(self.image.copy(), -90)
#Mirroring the half arrow that we earlier created to form a full image

self.dir = pg.Vector2(1, 0)  #Makes all boids spawn facing forwards

maxW, maxH = self.drawSurf.get_size()

self.rect = self.image.get_rect(center=(randint(50, maxW - 50), randint(50,
maxH - 50)))
#Creating a rectangle using maxW and maxH
#Although the boids look like arrows, it is this rectangle that is actually
dealing with all of the boid collisions and movements

self.ang = randint(0, 360)  #Random start angle and position

self.pos = pg.Vector2(self.rect.center)  #Stores the centre of the
rectangle

self.grid_lastpos = self.grid.getcell(self.pos)  #Last known position of
the boid

self.grid.add(self, self.grid_lastpos)  #Adding the last known position to
the grid
```

The ‘Boid’ class contains another method, ‘update’, which is the most essential section of our program. It contains all of the mathematical expressions that make the boids react to their surroundings the way that they do. But before we define the ‘update’ method, we need to understand the working of another class, the ‘BoidGrid’ class, since the ‘update’ method requires the ‘BoidGrid’ class for a lot of its functions.

```
class BoidGrid():  #Tracks boids in the spatial partition grid


    def __init__(self):

        self.grid_size = 100  #Screen partitioned

        self.dict = {}

    #Finds the grid cell corresponding to given position

    def getcell(self, pos):

        return (pos[0]//self.grid_size, pos[1]//self.grid_size)

    #Boids add themselves to cells when crossing into new cell

    def add(self, boid, key):

        if key in self.dict:

            self.dict[key].append(boid)

        else:

            self.dict[key] = [boid]
    #They also remove themselves from the previous cell

    def remove(self, boid, key):

        if key in self.dict and boid in self.dict[key]:

            self.dict[key].remove(boid)

    #Returns a list of nearby boids within all  surrounding 9 cells

    def getnear(self, boid, key):

        if key in self.dict:

            nearby = []

            for x in (-1, 0, 1):

                for y in (-1, 0, 1):

                    nearby += self.dict.get((key[0] + x, key[1] + y), [])

            nearby.remove(boid)

        return nearby
```

Now, we define the ‘Boid’ class’s ‘update’ method.

```
def update(self, dt, speed, coherence, avoidance, adherence, ejWrap=False):

    maxW, maxH = self.drawSurf.get_size()

    selfCenter = pg.Vector2(self.rect.center)  #Centre of boid saved

    turnDir = xvt = yvt = yat = xat = 0

    turnRate = 120 * dt  #Responsible for how smoothly the boids turn

    margin = 42

    self.ang = self.ang + randint(-4, 4)
    #Makes it look as though the boid is looking around before it moves

    self.grid_pos = self.grid.getcell(self.pos)  #Grid position returned

    if self.grid_pos != self.grid_lastpos:
    #If they are equal, updation not required

        self.grid.add(self, self.grid_pos)  #Added to new position

        self.grid.remove(self, self.grid_lastpos)  #Removed from old
position

        self.grid_lastpos = self.grid_pos  #Last known position updated

    #Get nearby boids and sort them by order of their distances to the
current boid
    near_boids = self.grid.getnear(self, self.grid_pos)
    neiboids = sorted(near_boids, key=lambda i:
pg.Vector2(i.rect.center).distance_to(selfCenter))

    del neiboids[7:]  #Keep 7 closest, dump the rest

    #This is because we only want our boid to be influenced by the 7
closest boids to it, and not EVERY boid on the screen
    #When boid has neighbors

    if (ncount := len(neiboids)) > 1:

        nearestBoid = pg.Vector2(neiboids[0].rect.center)

        for nBoid in neiboids:
        #Adds up neighbor vectors and angles for averaging

            xvt += nBoid.rect.centerx

            yvt += nBoid.rect.centery

            yat += sin(radians(nBoid.ang))

            xat += cos(radians(nBoid.ang))

        tAvejAng = degrees(atan2(yat, xat))

        targetV = (xvt / ncount, yvt / ncount)
        #Position boid wants to go to, average position of all boids

        #If too close, move away from closest neighbor

        if selfCenter.distance_to(nearestBoid) < self.bSize + avoidance :
targetV = nearestBoid

        tDiff = targetV - selfCenter  #Get angle differences for steering

        tDistance, tAngle = pg.math.Vector2.as_polar(tDiff)
        #Distance and angle to the target coordinate returned by the polar
function

        #If boid is close enough to neighbors, match their average angle

        if tDistance < self.bSize*5 + coherence*2 : tAngle = tAvejAng

        #Computes the difference to reach target angle, for smooth steering

        angleDiff = (tAngle - self.ang) + 180

        if abs(tAngle - self.ang) > .5 + ((adherence * 5) / 1000): turnDir
= (angleDiff / 360 - (angleDiff // 360)) * 360 - 180

        #If boid gets too close to target, steer away

        if tDistance < self.bSize + avoidance and targetV == nearestBoid :
turnDir = -turnDir
    #We take the negative of the current turnDir so the boid moves away
from its nearest neighbour (^)

    #Avoid edges of screen by turning toward the edge normal-angle

    sc_x, sc_y = self.rect.centerx, self.rect.centery
    #If wrap is disabled, we need to make the boids stay within the frame

    if not ejWrap and min(sc_x, sc_y, maxW - sc_x, maxH - sc_y) < margin:

        if sc_x < margin : tAngle = 0  #If on the left side, go right

        elif sc_x > maxW - margin : tAngle = 180  #If on the right, go left

        if sc_y < margin : tAngle = 90  #If at the bottom, go up

        elif sc_y > maxH - margin : tAngle = 270 #If at the top, go down

        angleDiff = (tAngle - self.ang) + 180
        #Increase turnRate to keep boids on screen

        turnDir = (angleDiff / 360 - (angleDiff // 360)) * 360 - 180

        edgeDist = min(sc_x, sc_y, maxW - sc_x, maxH - sc_y)

        turnRate = turnRate + (1 - edgeDist / margin) * (20 - turnRate)

    #turnRate=minRate, 20=maxRate
    if turnDir != 0:  #Steers based on turnDir, handles left or right

        self.ang += turnRate * abs(turnDir) / turnDir

    self.ang %= 360  #Ensures that the angle stays within 0-360

    #Now we make the image of the boid match all the updates that we made
to the boid container rectangle

    self.image = pg.transform.rotate(self.orig_image, -self.ang)

    self.rect = self.image.get_rect(center=self.rect.center)
    #Recentering fix

    self.dir = pg.Vector2(1, 0).rotate(self.ang).normalize()

    self.pos += self.dir * dt * (speed + (7 - ncount) * 5) #Movement speed

    #Optional screen wrap

    #We now update the boids reappear at the opposite side of the screen
    if ejWrap and not self.drawSurf.get_rect().contains(self.rect):

        if self.rect.bottom < 0 : self.pos.y = maxH

        elif self.rect.top > maxH : self.pos.y = 0

        if self.rect.right < 0 : self.pos.x = maxW

        elif self.rect.left > maxW : self.pos.x = 0

    #Actually update position of boid

    self.rect.center = self.pos
```

All of the above mathematical expressions were used to calculate self.pos, and we can finally update the centre of the rectangle so that we can observe the boid move on screen. We now get to the final section of the program, the __main__ function.

In the main method, we create and define the interactive GUI used to simulate the boids. We finally implement our main program through the GUI.

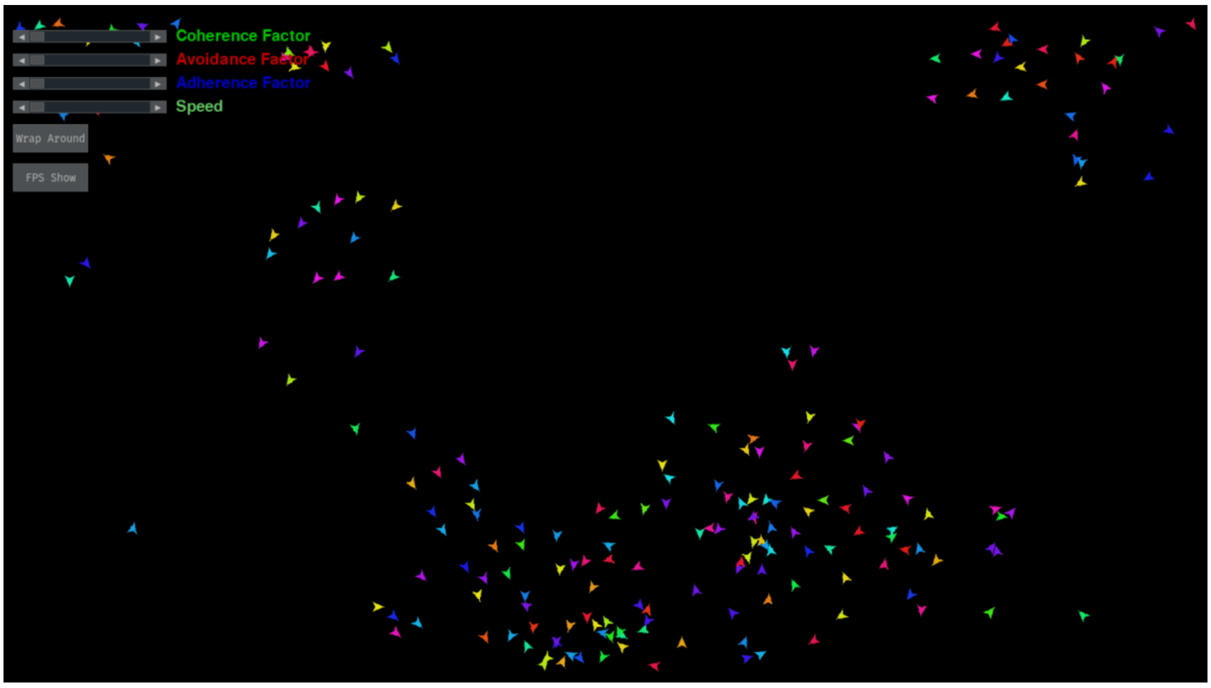
```
def main():

    pg.init()  #Preparing the window

    pg.display.set_caption("PyNBoids")

    if FLLSCRN:

        #Getting the display size
        currentRez = (pg.display.Info().current_w,
pg.display.Info().current_h)

        screen = pg.display.set_mode(currentRez, pg.SCALED | pg.NOFRAME |
pg.FULLSCREEN, vsync=1)

        # pg.mouse.set_visible(False)

    else: screen = pg.display.set_mode((WIDTH, HEIGHT), pg.RESIZABLE |
pg.SCALED, vsync=1)

    max_width, max_height = screen.get_size()



    #Handles all of the UI rendering from pygame_gui
    manager = pgui.UIManager((WIDTH, HEIGHT))



    #Slider to control cohesion
    cohesion_slider = pgui.elements.UIHorizontalSlider(

        pg.Rect((10, 30), (200, 20)),

        0,

        (0, 100),

        manager=manager

    )

    cohesion_slider.enable()  #To make it interactive



    #Slider to control avoidance
    avoidance_slider = pgui.elements.UIHorizontalSlider(

        pg.Rect((10, 60), (200, 20)),

        0,

        (0, 100),
        manager=manager

    )

    avoidance_slider.enable()  #To make it interactive



    #Slider to control adherence
    adherence_slider = pgui.elements.UIHorizontalSlider(

        pg.Rect((10, 90), (200, 20)),

        0,

        (0, 100),

        manager=manager

    )

    adherence_slider.enable()  #To make it interactive



    #Slider to control speed of the boids
    speed_slider = pgui.elements.UIHorizontalSlider(

        pg.Rect((10, 120), (200, 20)),

        0,

        (0, 100),

        manager=manager

    )

    speed_slider.enable()  #To make it interactive



    #Used to either turn wrap on or off
    wrap_toggle = pgui.elements.UIButton(

        pg.Rect((10, 150), (100, 40)),

        text="Wrap Around",

        manager=manager

    )



    #Used to display the number of frames per second
    fps_toggle = pgui.elements.UIButton(

        pg.Rect((10, 200), (100, 40)),  #Location and dimensions on screen
```

```
        text="Show FPS",

        manager=manager

    )


    boidTracker = BoidGrid()

    nBoids = pg.sprite.Group()

    #Creating the group of boids to be rendered in the UI
    #Spawns the desired number of boids

    for n in range(BOIDZ) : nBoids.add(Boid(boidTracker, screen))



    if SHOWFPS: font = pg.font.Font(None, 30)

    clock = pg.time.Clock()

    coherence = avoidance = adherence = speed = 0

    wrap = fps = False

    #Main loop
    while True:

        for e in pg.event.get():
            #Exit condition to break out of the loop

            if e.type == pg.QUIT or e.type == pg.KEYDOWN and (e.key ==
pg.K_ESCAPE or e.key == pg.K_q or e.key==pg.K_SPACE):

                return
            #The main user event to control the various factors using the
sliders

            if e.type == pg.USEREVENT:

                if e.user_type == pgui.UI_HORIZONTAL_SLIDER_MOVED:

                    if e.ui_element == cohesion_slider:

                        coherence = e.value  #Changing the value of
coherence

                    elif e.ui_element == avoidance_slider:

                        avoidance = e.value  #Changing the value of
avoidance

                    elif e.ui_element == adherence_slider:

                        adherence = e.value  #Changing the value of
adherence

                    elif e.ui_element == speed_slider:
```

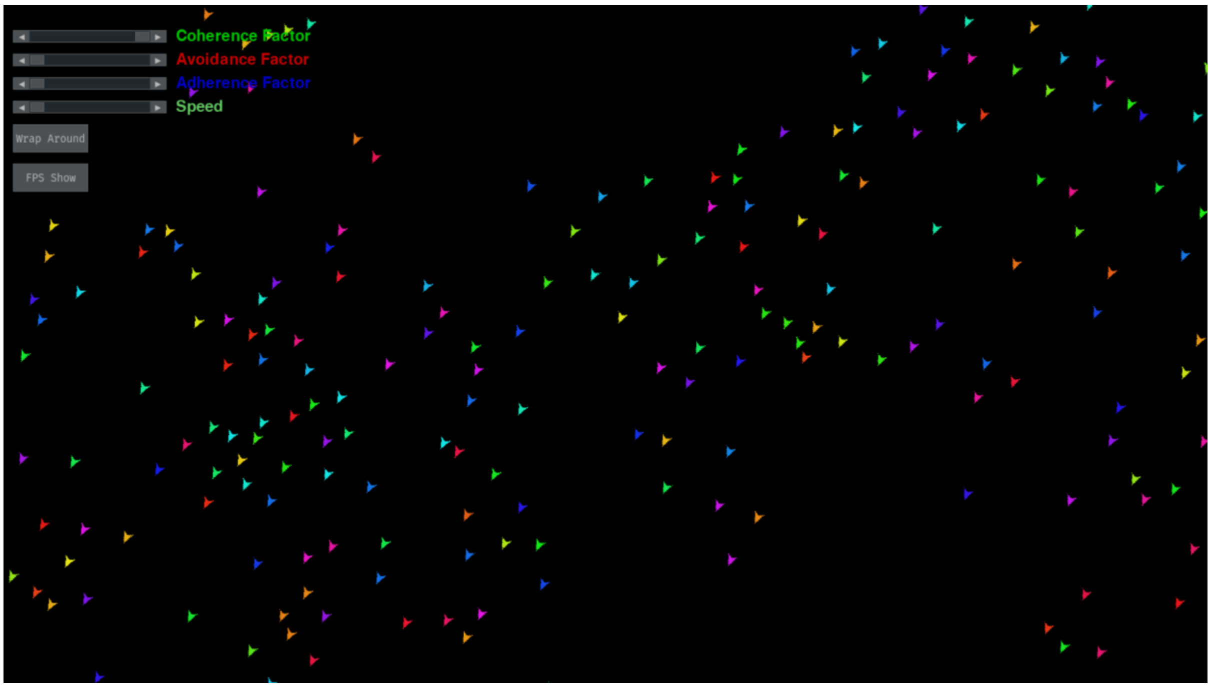
```python
                        speed = e.value  #Changing the value of speed

                elif e.user_type == pgui.UI_BUTTON_PRESSED:

                    if e.ui_element == wrap_toggle:

                        wrap = not wrap  #Toggling wrap value based on previous state

                    if e.ui_element == fps_toggle:

                        fps = not fps



            manager.process_events(e)  #Processing e and implementing it in UI



        dt = clock.tick(FPS) / 1000

        #Keep checking after dt intervals to detect changes in UI to update the boids
        manager.update(dt)



        screen.fill(BGCOLOR)
        #Update boid logic, then draw them

        nBoids.update(dt, SPEED + speed, coherence, avoidance, adherence, wrap)
        nBoids.draw(screen)
        manager.draw_ui(screen)

        #If true, displays the FPS in the upper left corner for debugging

        if fps : screen.blit(font.render(str(int(clock.get_fps())), True, [0, 200, 0]), (8, 8))
        screen.blit(font.render("Coherence Factor", True, [0, 200, 0]), (220, 30))
        screen.blit(font.render("Avoidance Factor", True, [200, 0, 0]), (220, 60)
        screen.blit(font.render("Adherence Factor", True, [0, 0, 200]), (220, 90))
        screen.blit(font.render("Speed", True, [100, 200, 100]), (220, 120))

        pg.display.update()

if __name__ == '__main__':

    main()

    pg.quit()
```

Result

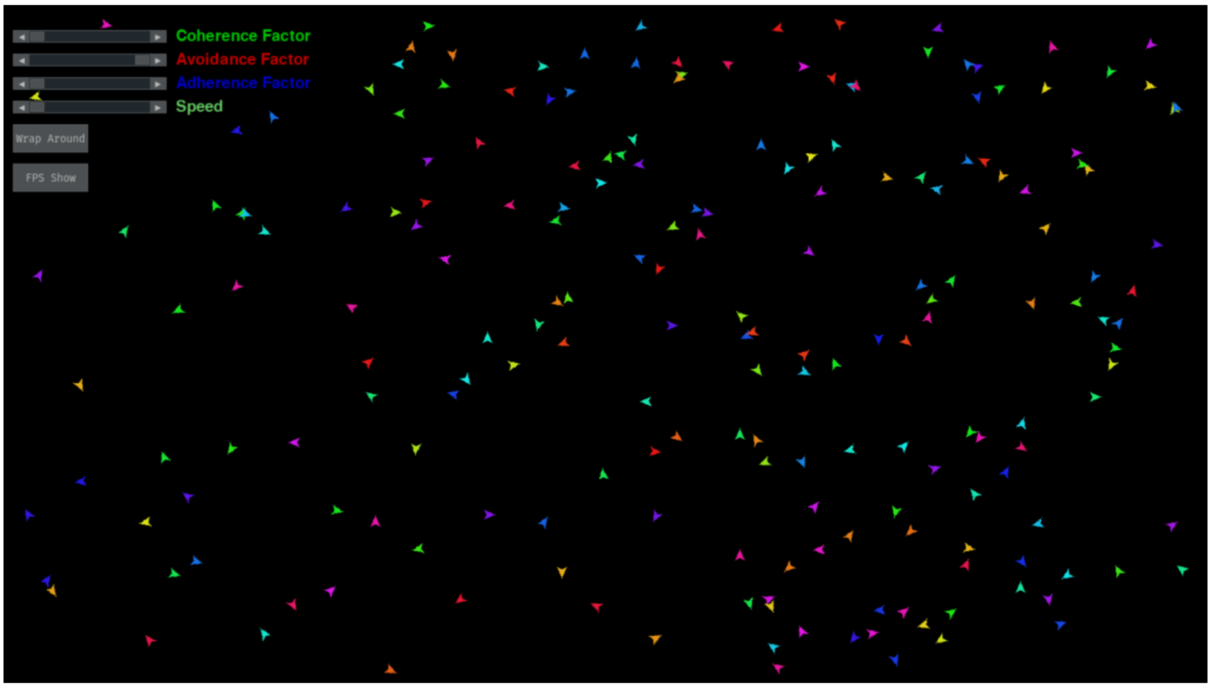
We first see the initial simulation.



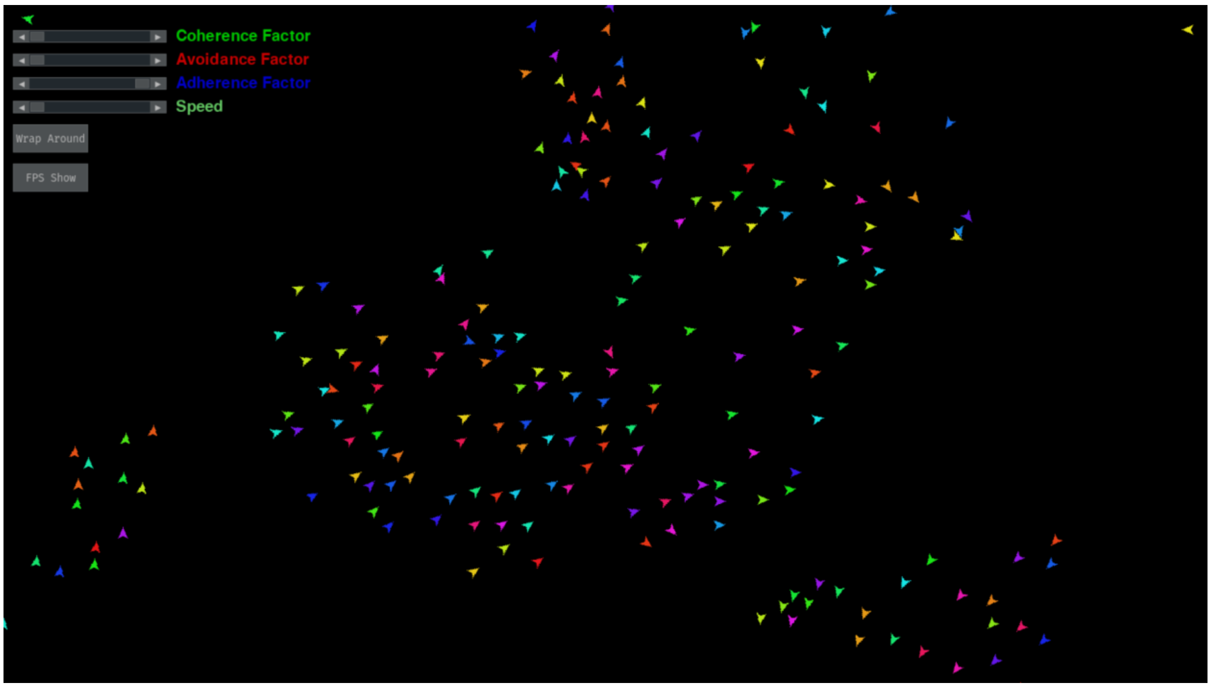
Increasing COHERENCE :-



Increasing AVOIDANCE :-



Increasing ADHERENCE :-



Conclusion

As we can observe from the result, changing the different factors for each individual boid changes how the flock behaves. This shows how emergent behaviours depend on each boid’s actions. The study of boids is an interesting part of Artificial Life, which is a field of study that observes living beings and simulates their likeliness in computer models.

Boid simulations can be used in movies, games, and other media content to simulate flocks of birds, group of bats or a school of fishes. It can also be used to study behaviours of individual creatures that in turn contribute to the emergent pattern.

Extra components can be added to the environment according to the needs of the simulation. For example, we can introduce obstacles and predators in the environment and code each boid to act accordingly. We can also add extra behaviours like perching at the bottom of the screen for a few seconds before taking flight and re-joining the flock.