Ministerul Educaţiei, Culturii și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei

Lucrare de laborator

Disciplina: *Inteligență Artificială*

Tema: *Flocking Behaviour*

A efectuat: st. gr. SI-221M Postu Ivan

A verificat: Gavrilița Mihail

Chișinău – 2023

Task 1 Implement the Vector class in Python that works on simple Python lists. The Vector class should implement the vector operations:

- Vector norm
- Vector addition
- Vector subtraction
- Multiplication with a scalar
- Division by a scalar
- Vector dot product
- Vector cross product

```
class IVector:
    def __init__(self, values):
        self.values = list(values)

    def norm(self):
        return sum(x**2 for x in self.values) ** 0.5

    def __add__(self, other):
        if len(self.values) != len(other.values):
            raise ValueError("Vectors must be the same length")
        return IVector([x + y for x, y in zip(self.values, other.values)])

    def __sub__(self, other):
        if len(self.values) != len(other.values):
            raise ValueError("Vectors must be the same length")
        return IVector([x - y for x, y in zip(self.values, other.values)])

    def __mul__(self, scalar):
        return IVector([x * scalar for x in self.values])

    def __truediv__(self, scalar):
        return IVector([x / scalar for x in self.values])

    def dot(self, other):
        if len(self.values) != len(other.values):
            raise ValueError("Vectors must be the same length")
        return sum(x * y for x, y in zip(self.values, other.values))

    def __str__(self):
        return "Vector({})".format(self.values)

    def cross(self, other):
        if len(self.values) != 3 or len(other.values) != 3:
            raise ValueError("Cross product is only defined for 3-dimensional vectors")
```

```
        x1, y1, z1 = self.values
        x2, y2, z2 = other.values
        return IVector([y1 * z2 - y2 * z1, z1 * x2 - z2 * x1, x1 * y2 - x2 * y1])

    def get_values(self):
        return self.values.copy()

    def get_value(self, index):
        return self.values[index]

    def set_value(self, index, new_val):
        self.values[index] = new_val


v1 = IVector([1, 2, 3])
v2 = IVector([4, 5, 6])
v3 = v1 + v2
v4 = v2 - v1
v5 = v1 * 2
v6 = v2 / 2
dot_product = v1.dot(v2)
cross_product = v1.cross(v2)

print(v1, v2, v3, v4, v5, v6, dot_product, cross_product)
```

Task 2 - Using the Vector class and the provided paper, implement the Boid class with the steering behaviors:

- Separation
- Alignment
- Cohesion

Task 3 Add the calm flocking behaviour to the Boid class according to the provided paper, using the 3 steering behaviours implemented in the Task 2.

```
import random
from IVector import IVector


class Boid:
    def __init__(self, x, y, width, height, color=255):
        self.color = color
        self.width = width
        self.height = height
        self.max_speed = 10
        self.perception = 100
        self.max_force = 1
```

```
        self.position = IVector([x, y])
        vec = list([(random.random() - 0.5) * 10, (random.random() - 0.5) * 10])
        self.velocity = IVector(vec)
        vec = list([(random.random() - 0.5) * 10, (random.random() - 0.5) * 10])
        self.acceleration = IVector(vec)

    def update(self):
        self.position += self.velocity
        self.velocity += self.acceleration  # limit
        if self.velocity.norm() > self.max_speed:
            self.velocity = self.velocity / self.velocity.norm() * self.max_speed
        self.acceleration = IVector(list([0, 0]))

    def show(self):
        stroke(self.color)
        circle(self.position.get_value(0), self.position.get_value(1), 10)

    def edges(self):
        if self.position.get_value(0) > self.width:
            self.position.set_value(0, 0)
        elif self.position.get_value(0) < 0:
            self.position.set_value(0, self.width)
        if self.position.get_value(1) > self.height:
            self.position.set_value(1, 0)
        elif self.position.get_value(1) < 0:
            self.position.set_value(1, self.height)

    def align(self, boids):
        steering = IVector(list([0, 0]))
        total = 0
        avg_vec = IVector(list([0, 0]))
        for boid in boids:
            if (boid.position - self.position).norm() < self.perception:
                avg_vec += boid.velocity
                total += 1
        if total > 0:
            avg_vec /= total
            avg_vec = IVector(avg_vec.get_values())
            avg_vec = (avg_vec / avg_vec.norm()) * self.max_speed
            steering = avg_vec - self.velocity
        return steering

    def cohesion(self, boids):
        steering = IVector(list([0, 0]))
        total = 0
        center_of_mass = IVector(list([0, 0]))
        for boid in boids:
            if (boid.position - self.position).norm() < self.perception:
```

```
                center_of_mass += boid.position
                total += 1
        if total > 0:
            center_of_mass /= total
            center_of_mass = IVector(center_of_mass.get_values())
            vec_to_com = center_of_mass - self.position
            if vec_to_com.norm() > 0:
                vec_to_com = (vec_to_com / vec_to_com.norm()) * self.max_speed
            steering = vec_to_com - self.velocity
            if steering.norm() > self.max_force:
                steering = (steering / steering.norm()) * self.max_force
        return steering

    def separation(self, boids):
        steering = IVector(list([0, 0]))
        total = 0
        avg_vector = IVector(list([0, 0]))
        for boid in boids:
            distance = (boid.position - self.position).norm()
            if self.position != boid.position and distance < self.perception:
                diff = self.position - boid.position
                diff /= distance
                avg_vector += diff
                total += 1
        if total > 0:
            avg_vector /= total
            avg_vector = IVector(avg_vector.get_values())
            if steering.norm() > 0:
                avg_vector = (avg_vector / steering.norm()) * self.max_speed
            steering = avg_vector - self.velocity
            if steering.norm() > self.max_force:
                steering = (steering / steering.norm()) * self.max_force
        return steering

    def calm_flocking(self, boids):
        alignment = self.align(boids)
        cohesion = self.cohesion(boids)
        separation = self.separation(boids)
        return alignment * 0.04 + cohesion * 0.02 + separation * 0.02
```

Task 4 - Combine the Boid class with the behaviours implemented in previous tasks with the provided code for the simulation of S. tuberosum and run it in CodeSkulptor. The rocks should exhibit flocking behaviour as implemented in the Boid class. Note: The NumPy library will not work in CodeSkulptor. If you implemented the Vector class with NumPy, you should also add another implementation with lists or you can use the numeric library.

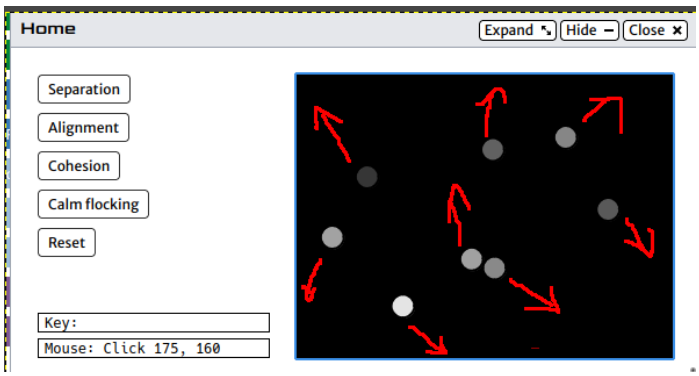

Figura 1 Miscarea im mod aleatoriu

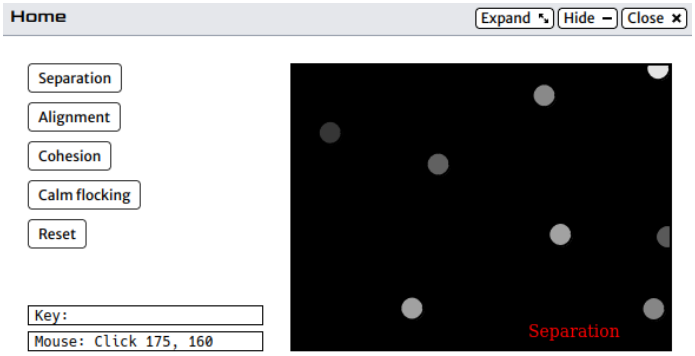

Figura 2 Elementele pastreaza distanța maximală între ele (mod separation)

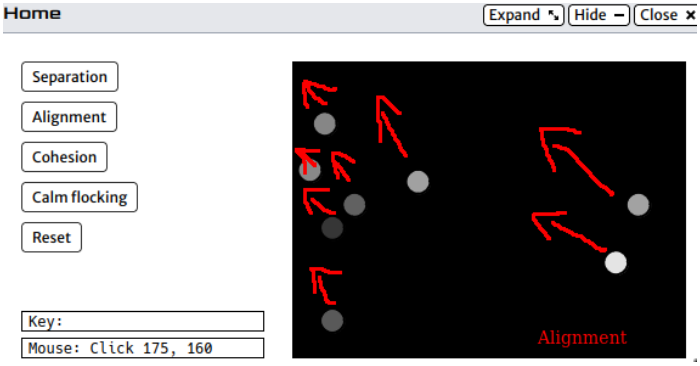

Figura 3 Elementele se mișcă în direcție comună (aligment)

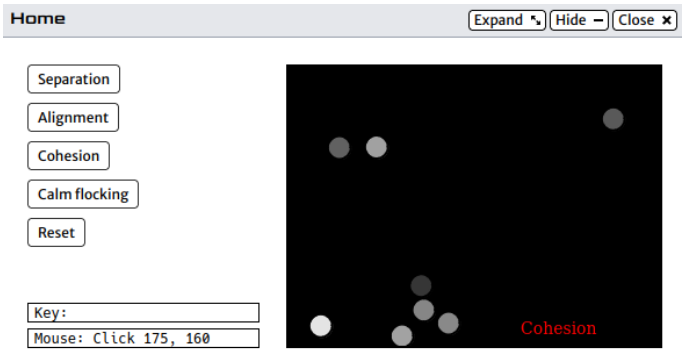

Figura 4 Elementele tind să fie mai aproape între ele (Cohesion)

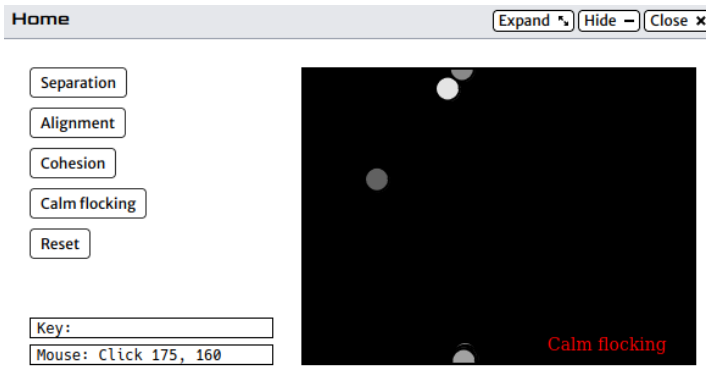

Figura 5 Elementele emulează mișcare în stol

```
import simplegui
import random


def arr_len(arr):
    if isinstance(arr, list):
        return len(arr)
    size, _ = arr.buffer_info()
    element_size = arr.itemsize
    length = size // element_size
    return length


class IVector:
    def __init__(self, values):
        self.values = list(values)

    def norm(self):
        return sum(x**2 for x in self.values) ** 0.5

    def __add__(self, other):
        if len(self.values) != len(other.values):
            raise ValueError("Vectors must be the same length")
        return IVector([x + y for x, y in zip(self.values, other.values)])

    def __sub__(self, other):
        if len(self.values) != len(other.values):
            raise ValueError("Vectors must be the same length")
        return IVector([x - y for x, y in zip(self.values, other.values)])

    def __mul__(self, scalar):
        return IVector([x * scalar for x in self.values])

    def __truediv__(self, scalar):
```

```
        return IVector([x / scalar for x in self.values])

    def dot(self, other):
        if len(self.values) != len(other.values):
            raise ValueError("Vectors must be the same length")
        return sum(x * y for x, y in zip(self.values, other.values))

    def __str__(self):
        return "Vector({})".format(self.values)

    def cross(self, other):
        if len(self.values) != 3 or len(other.values) != 3:
            raise ValueError("Cross product is only defined for 3-dimensional vectors")
        x1, y1, z1 = self.values
        x2, y2, z2 = other.values
        return IVector([y1 * z2 - y2 * z1, z1 * x2 - z2 * x1, x1 * y2 - x2 * y1])

    def get_values(self):
        return self.values.copy()

    def get_value(self, index):
        return self.values[index]

    def set_value(self, index, new_val):
        self.values[index] = new_val


class Boid:
    def __init__(self, x, y, width, height, color=255):
        self.color = color
        self.width = width
        self.height = height
        self.max_speed = 10
        self.perception = 100
        self.max_force = 1
        self.position = IVector([x, y])
        vec = list([(random.random() - 0.5) * 10, (random.random() - 0.5) * 10])
        self.velocity = IVector(vec)
        vec = list([(random.random() - 0.5) * 10, (random.random() - 0.5) * 10])
        self.acceleration = IVector(vec)

    def update(self):
        self.position += self.velocity
        self.velocity += self.acceleration  # limit
        if self.velocity.norm() > self.max_speed:
            self.velocity = self.velocity / self.velocity.norm() * self.max_speed
        self.acceleration = IVector(list([0, 0]))
```

```
def show(self, canvas):
    color = (
        "rgb("
        + str(self.color)
        + ","
        + str(self.color)
        + ","
        + str(self.color)
        + ")"
    )
    canvas.draw_circle(
        [self.position.get_value(0), self.position.get_value(1)],
        10,
        2,
        "Black",
        color,
    )
    #  stroke(self.color)
    # circle(, 10)

def edges(self):
    if self.position.get_value(0) > self.width:
        self.position.set_value(0, 0)
    elif self.position.get_value(0) < 0:
        self.position.set_value(0, self.width)
    if self.position.get_value(1) > self.height:
        self.position.set_value(1, 0)
    elif self.position.get_value(1) < 0:
        self.position.set_value(1, self.height)

def align(self, boids):
    steering = IVector(list([0, 0]))
    total = 0
    avg_vec = IVector(list([0, 0]))
    for boid in boids:
        if (boid.position - self.position).norm() < self.perception:
            avg_vec += boid.velocity
            total += 1
    if total > 0:
        avg_vec /= total
        avg_vec = IVector(avg_vec.get_values())
        avg_vec = (avg_vec / avg_vec.norm()) * self.max_speed
        steering = avg_vec - self.velocity
    return steering

def cohesion(self, boids):
    steering = IVector(list([0, 0]))
    total = 0
```

```
        center_of_mass = IVector(list([0, 0]))
        for boid in boids:
            if (boid.position - self.position).norm() < self.perception:
                center_of_mass += boid.position
                total += 1
        if total > 0:
            center_of_mass /= total
            center_of_mass = IVector(center_of_mass.get_values())
            vec_to_com = center_of_mass - self.position
            if vec_to_com.norm() > 0:
                vec_to_com = (vec_to_com / vec_to_com.norm()) * self.max_speed
            steering = vec_to_com - self.velocity
            if steering.norm() > self.max_force:
                steering = (steering / steering.norm()) * self.max_force
        return steering

    def separation(self, boids):
        steering = IVector(list([0, 0]))
        total = 0
        avg_vector = IVector(list([0, 0]))
        for boid in boids:
            distance = (boid.position - self.position).norm()
            if self.position != boid.position and distance < self.perception:
                diff = self.position - boid.position
                diff /= distance
                avg_vector += diff
                total += 1
        if total > 0:
            avg_vector /= total
            avg_vector = IVector(avg_vector.get_values())
            if steering.norm() > 0:
                avg_vector = (avg_vector / steering.norm()) * self.max_speed
            steering = avg_vector - self.velocity
            if steering.norm() > self.max_force:
                steering = (steering / steering.norm()) * self.max_force
        return steering

    def calm_flocking(self, boids):
        alignment = self.align(boids)
        cohesion = self.cohesion(boids)
        separation = self.separation(boids)
        return alignment * 0.04 + cohesion * 0.02 + separation * 0.02


boids = list()

MAX_WIDTH = 320
MAX_HEIGHT = 240
```

```
text_msg = "_"


# Handler for mouse click
def click_separation():
    global text_msg
    text_msg = "Separation"
    pass
def click_alignment():
    global text_msg
    text_msg = "Alignment"
    pass
def click_cohesion():
    global text_msg
    text_msg = "Cohesion"
    pass
def click_calm_flocking():
    global text_msg
    text_msg = "Calm flocking"
    pass
def click_reset():
    global text_msg
    text_msg = "_"
    pass

# Handler to draw on canvas
def draw(canvas):
    canvas.draw_text(text_msg, [MAX_WIDTH - 120, MAX_HEIGHT - 10], 14, "Red")
    for i in range(len(boids)):
        boids[i].update()

        if text_msg == "Separation":
            separation = boids[i].separation(boids)
            boids[i].acceleration += separation * 0.03
        elif text_msg == "Alignment":
            alignment = boids[i].align(boids)
            boids[i].acceleration += alignment * 0.03
        elif text_msg == "Cohesion":
            cohesion = boids[i].cohesion(boids)
            boids[i].acceleration += cohesion * 0.03
        elif text_msg == "Calm flocking":
            calm_flocking = boids[i].calm_flocking(boids)
            boids[i].acceleration += calm_flocking

        boids[i].edges()
        boids[i].show(canvas)
```

```
def mouse_handler(pos):
    print("Mouse clicked at", pos[0], pos[1])
    boids.append(Boid(pos[0], pos[1], MAX_WIDTH, MAX_HEIGHT, random.randint(50, 255)))


frame = simplegui.create_frame("Home", MAX_WIDTH, MAX_HEIGHT)
frame.add_button("Separation", click_separation)
frame.add_button("Alignment", click_alignment)
frame.add_button("Cohesion", click_cohesion)
frame.add_button("Calm flocking", click_calm_flocking)
frame.add_button("Reset", click_reset)
frame.set_mouseclick_handler(mouse_handler)
frame.set_draw_handler(draw)

# Start the frame animation
frame.start()
```

Concluzie

În urma realizării lucrării de laborator cu succes au fost atinse obiectivele de bază de a crea implementarea proprie a structurii de date Vector, de a emula comportamentul obiectelor spațiale utilizînd librarii de vizualizare 2d și structura de date creată la pasul 1. A fost implementat comportamentul obiectelor din spațiu de mărire a distanței între ele, mișcare în direcție comună, micșorarea distanței între ele și mișcarea în stol. Cu succes logica a fost adaptată pentru emulatorul din browser CodeSkulptor3.