MINISTRY OF EDUCATION OF REPUBLIC OF MOLDOVA
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software Engineering and Automatics

Report

on Artificial Intelligence Fundamentals

Laboratory Work nr. 2

Flocking Behaviour

Performed by: **Diana Marusic**, FAF-171

Verified by: **Mihail Gavrilița**, asist. univ.

Chișinău, 2021

Contents

Flocking Behaviour

The Task

The task for this laboratory work consisted in implementing the flocking behaviour to the Solanum tuberosum rocks from the starship game.

The simulation was based on a tool developed by Ijon, by using the code that was already provided in the Simulation.py file from the library. The code could run on http://www.codeskulptor.org/ but it also runs perfectly on the Python3 version https://py3.codeskulptor.org/.

The mandatory task was to simulate the calm flocking behavior of rocks and then continue with their evading and attacking behavior, thus yielding a more complete simulation of S. tuberosum's behavior patterns

Solution Description

Note: The code was written in Python3, because Python2 is no longer supported officially, and the ML company would benefit from running the latest technologies, so to run the system, it is necessary to use https://py3.codeskulptor.org/.

Flocking behaviour implementation

Flocking behaviour in this task was simulated using the **boids** model of artificial life presented by Craig Reynolds in the paper [3]. As the paper states, there are 3 basic rules(also named steering behaviours) [3, 4] that control models of flocking behavior:

- Alignment - steer towards the average heading of neighbours
- Cohesion - steer to move towards average position of neighbours (long range attraction)
- Separation - steer to avoid crowding neighbours (short range repulsion)

To model flocking behaviour for the rocks in the game simulations, a new class named ***Boid***, based on the *Sprite* class from the provided code, was created, whith all the required functions.

The function ***flocking_behaviour*** from the *Boid* class is very simple, implementing the 3 behaviours and adding the steer for each behaviour to the acceleration of Boid, as shown in the listing below.

Listing 1: Flocking behaviour method

```

def flocking_behaviour(self, all_boids):
        """
        the function implementing flocking behaviour
        ---------
        parameters:
```

```
            all_boids (set) - the set of all boids
        """
        align_steer = self.alignment(all_boids)
        cohesion_steer = self.cohesion(all_boids)
        sep_steer = self.separation(all_boids)

        self.add_steer(align_steer)
        self.add_steer(cohesion_steer)
        self.add_steer(sep_steer*2)
```

At each step in the game, the boid will calculate the alignment steer, the cohesion steer and the separation steer and add it to the current acceleration. Based on observations that the separation force seemed too small, it was decided to multiply the separation steer by 2, to increase the force of separation between rocks, therefore avoiding colisions between rocks.

The next sections will describe in details the implementations of the alignment, cohesion and separation behaviours.

Alignment

Alignment is one of the behaviours characteristic for boids. The main idea of this behaviour is the fact that all the boids from the neighborhood should align and head in the same direction.

So for one boid the alignment function will have to analyze the direction where the boids from the neighborhood are heading, calculate the average and adjust the direction based on that average. It will have to use a parameter defined called ***"perception"***, that represents the neighborhood area for the boids. In the code, the parameter *self.perception* contains only the radius, or the minimum distance for the boid, from which it can calculate the perception area.

So the general algorithm for the alignment of one boid can be described in pseudocode as the following:

```
    FOR each other_boid in the boids set:
        distance <- calculate distance between current boid and the other_boid
        IF distance < perception:
            add other_boid velocity to the vector of velocity of boids

    IF there is at least one boid in the perception (neighborhood area):
        calculate average velocity of neighborhood boids
        normalize average velocity vector
        steer <- substract from the average vector multiplied by the maximum
    velocity the current                velocity of the boid
        adjust steering force (make it smaller if it exceeds the maximum force)

    return steering
```

A visual representation for better understanding the alignment of the boids can be observed in the Figure 1 below.

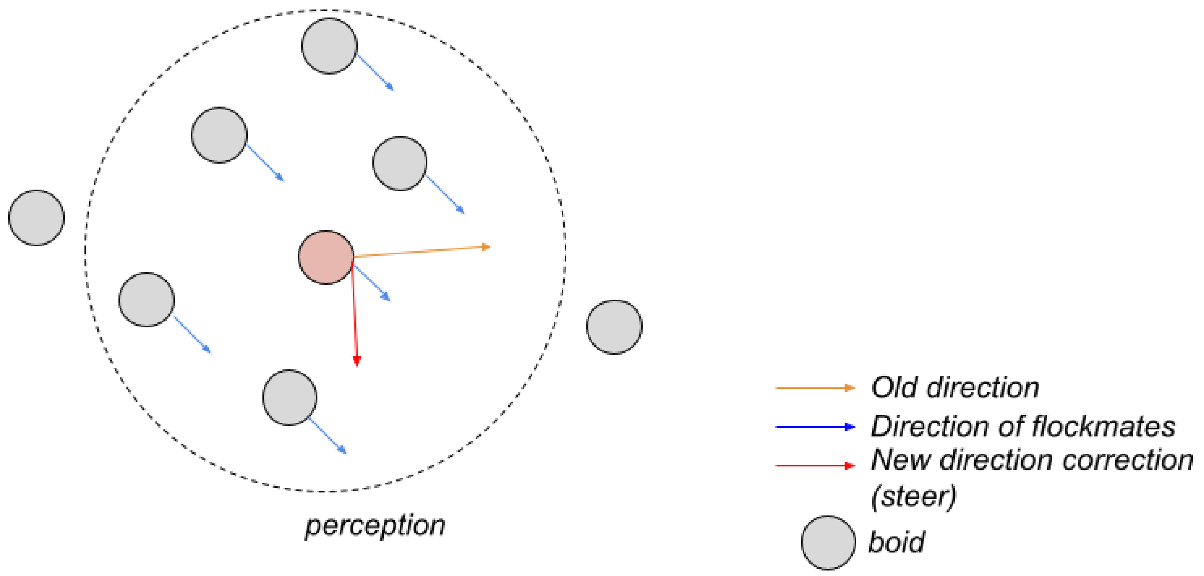

Figure 1: Alignment visual representation

The boid alighment was implemented as a method in the Boid class that was inspired from the Sprite class. The code for the alighment method can be observed in the Listing 2 below.

Listing 2: Code for boid alignment

```

def alignment(self, boids):
        """ steer towards the average heading of neighbours """
        avg_vector = Vector(0,0)
        steering = Vector(0,0)
        cnt_boids_in_perception = 0

        for boid in boids:
            distance = calcDistanceWithRadius(boid.pos, self.pos, self.radius)
            # if (Vector(*boid.pos)-Vector(*self.pos)).norm() < self.perception:
            if (distance < self.perception):

                avg_vector += Vector(*boid.vel)
                cnt_boids_in_perception +=1

        if cnt_boids_in_perception >0:
            avg_vector = avg_vector / cnt_boids_in_perception
            avg_vec_normalized = (avg_vector / avg_vector.norm())
            steering = avg_vec_normalized * (self.max_velocity) - Vector(*self.
    vel)

        steering = self.adjust_steering_force(steering, delta_force=-0.01)

        return steering
```

Cohesion

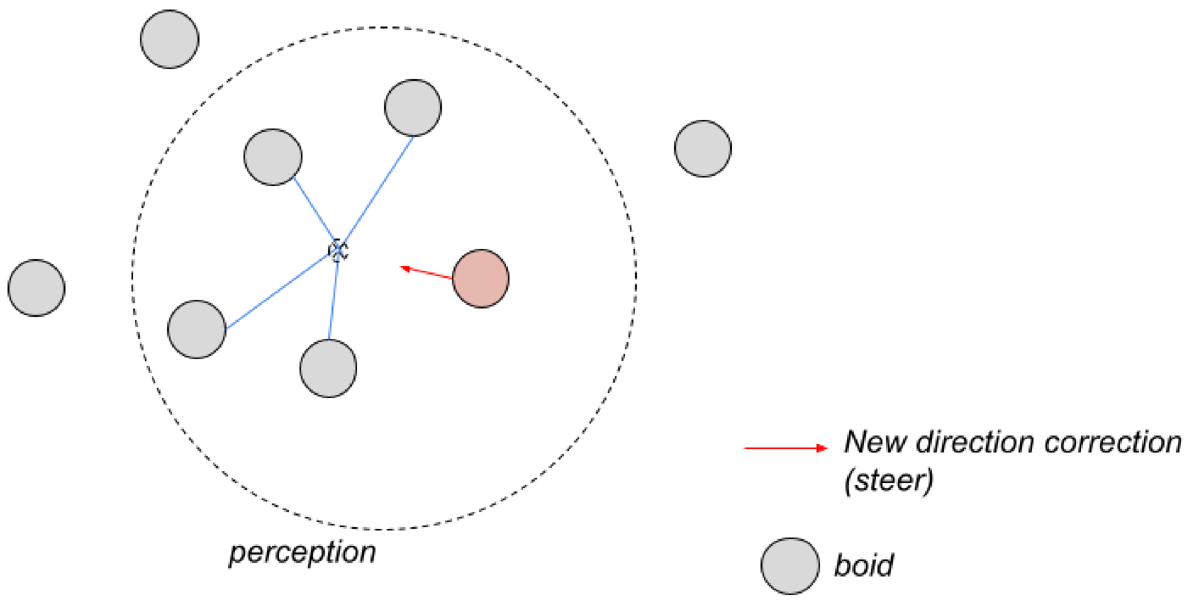

Figure 2: Cohesion visual representation

Cohesion is the second type of behaviour characteristic for the boids. It can be described as the tendency of the boids to move towards the average position of the neighborhood boids, that can also be called the *"center of mass"*, as presented in the Figure 2 also.

The general algorithm can be described high-level as following:

1. Find the center of mass of the neighboring boids
2. Calculate the difference of the current boid from the center of mass
3. Use the difference from the center of mass to steer to the center of mass with the maximul velocity
4. Adjusts the steering force not to exceed the maximum allowed force max_force

The code for the ***cohesion method*** added to the *Boid* class is presented in the next listing (Listing 3).

Listing 3: Code for boid cohesion

```
    def cohesion(self, all_boids, delta_force=-0.01):
        """ steer to move towards average position of neighbours (long range
    attraction) """
        steering = Vector(0,0)
        cnt_boids_in_perception = 0
        center_mass = Vector(0,0)

```

```
        for boid in all_boids:
            distance = calcDistanceWithRadius(boid.pos, self.pos, self.radius)
            # if (Vector(*boid.pos)-Vector(*self.pos)).norm() < self.perception:
            if (distance < self.perception):

                center_mass += Vector(*boid.pos)
                cnt_boids_in_perception +=1

        if cnt_boids_in_perception >0:
            center_mass = center_mass / cnt_boids_in_perception
            diff_to_center_vect = center_mass - Vector(*self.pos) - self.radius * Vector(1,1)

            if diff_to_center_vect.norm() >0:
                diff_to_center_vect = (diff_to_center_vect/diff_to_center_vect.norm()) * self.max_velocity

            steering = diff_to_center_vect - Vector(*self.vel)

        steering = self.adjust_steering_force(steering, delta_force=delta_force)

        return steering
```

Separation

Separation is the third behaviour characteristic to boids. It can be described as the force which tries to avoid crowding the neighborhood and avoid collision of the objects. It makes the boids that are too close move in opposite directions, so that they do not collide and accumulate in the same point.

This behaviour is very important, because if the boids follow only cohesion and alignment rules, then they tend to accumulate in the same point, collide and become like one single rock. So separation is the force that contra-balances the alignment and cohesion forces.

Separation behaviour is presented visually in the Figure 3 below.

The algorithm implemented for the separation behaviour is based on the inverse-square law: the force which tries to separate the boids is inversely proportional to the distance between boids, so that the boids that are closer will be separated with a greater force than the boids that have a bigger distance between them:

$$separation\_intensity \propto \frac{1}{distance^2}$$

The initial algorithm that was implemented did not take into consideration the fact that Rocks in the game are big objects, not just points, so the radius of the boids should also be taken into consideration. Therefore, a new function named *calcDistanceWithRadius* was added, which can calculate the distance between objects, taking into consideration their radiuses as

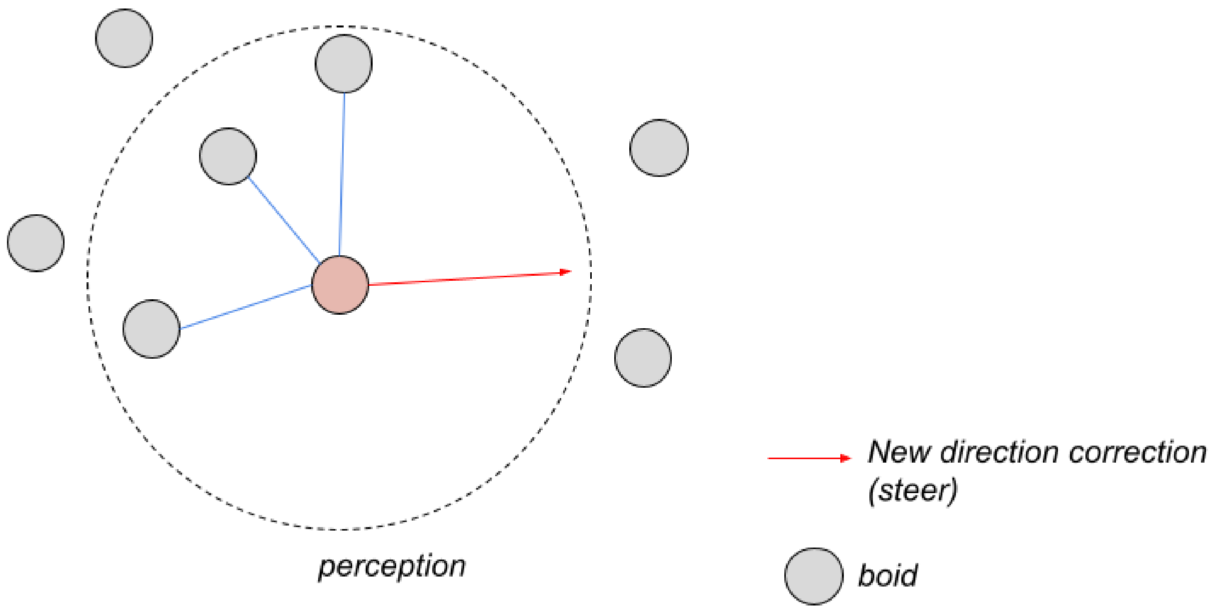

Figure 3: Separation visual representation

well. This improved dramatically the algorithm and made the rocks avoid collisions.

It was also observed that the separation force seemed to be too small, therefore in the *flocking_behaviour* the separation steer is multiplied by 2.

The code for the ***separation*** behaviour method on the *Boid class* is presented in the Listing 4 below.

Listing 4: Code for boid separation

```
def separation(self, all_boids):
        """ steer to avoid crowding neighbours (short range repulsion) """
        #min distance between rocks required
        delta_distance = self.perception
        steer_vector = Vector(0, 0) # must be 2D vector
        avg_steer = Vector(0,0)
        steering = Vector(0,0)

        cnt_close_neigh = 0
        all_positions = get_all_positions(all_boids)

        for rock_i_pos in all_positions:
            dist_i_j = calcDistanceWithRadius(rock_i_pos, self.pos, self.radius)


            if self.pos!=rock_i_pos and dist_i_j < delta_distance:
                Vector(Vector(*rock_i_pos) - Vector(*self.pos))

                #If rocks are intersecting, we consider the distance to be very very small, while the "diff" will still be the distance btw centers
                # so that the final "force" will be big
```

```
                if dist_i_j==0: #means rocks are intersecting
                    small_dist = 0.000001
                    dist_i_j = small_dist
                    diff = Vector(*self.pos) - Vector(*rock_i_pos)
                else:
                    diff = Vector(*self.pos) - Vector(*rock_i_pos) - self.radius * Vector(2,2)

                diff /= dist_i_j
                avg_steer += diff

                cnt_close_neigh +=1

        if cnt_close_neigh > 0:
            avg_steer /= cnt_close_neigh

            steering = avg_steer - Vector(*self.vel)

        steering = self.adjust_steering_force(steering)
        return steering
```

Attacking behaviour

The attacking behaviour shows a type of behaviour where all the asteroids move towards a starship, aiming to collide and destroy them, while still evading collision with other asteroids (similar to a bunch of people trying to fit in a bus).

For this purpose, the function ***attacking_behaviour*** was defined, which is presented in the listing below.

For the program to look more interesting, and the asteroids to "attack" with a greater force, the cohesion_steer from other objects is multiplied by a coefficient, named *"coef_steer"*, which works good with the value of *2*. So the *cohesion_steer* is multiplied by *2*.

The visual representation of the attacking behaviour can be observed in the figure 4 below.

The ***attacking_behaviour*** was added as a method to the *Boid* class. The method is presented in the listing 5 below.

Listing 5: Code for attacking behaviour

```
def attacking_behaviour(self, other_objects, coef_steer=2):
        cohesion_steer = self.cohesion(other_objects, delta_force=4)
        self.add_steer(cohesion_steer*coef_steer)
```

Evading behaviour

The evading behaviour is type of behaviour when the rocks are trying to evade any unknown objects, like starships and their missiles, while still flocking (much like a school of fish attacked

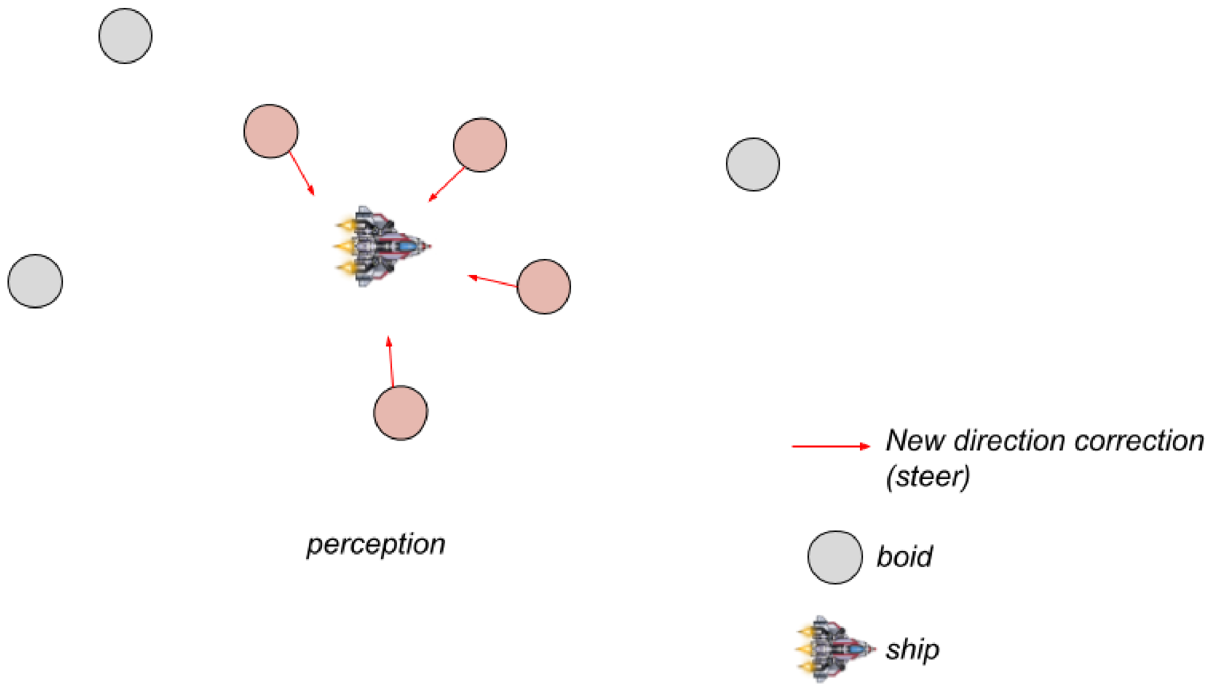

Figure 4: Attacking behaviour visual representation

by a predator). It was modeled by using the separation function that was already defined for the flocking behaviour, however changing the input parameter from *boids* to *other_objects*, so that the rocks will try to separate from other_objects, as well as keeping the previous flocking behaviour. A visual representation of the evading behaviour can be observed in the Figure 5.

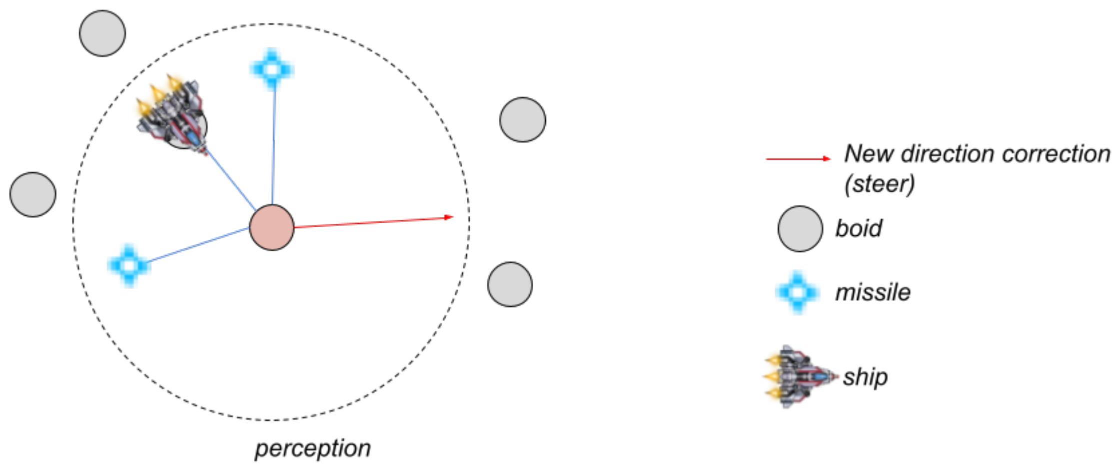

Figure 5: Evading behaviour visual representation

For this purpose, the function ***evading_behaviour*** was defined, which is presented in the listing below.

For the program to look more interesting, the *separation_steer* from other objects is multiplied by a coefficient, named *"coef_steer"*, which works good with the value of *4*. So the *sepration_steer* is multiplied by *4*.

Listing 6: Code for evading behaviour

```
def evading_behaviour(self, other_objects, coef_steer=4):
        separation_steer = self.separation(other_objects)
        self.add_steer(separation_steer*coef_steer)
```

The Vector Class

This project is based a lot on vectors and operations for vectors, therefor the class Vector was implemented, together with the operations characteristic for vectors: *addition (sum)*, *subtraction (difference)*, *division by a coefficient*, *multiplication with a coefficient*, *normalization*, *checking if the vector is negative*, *conversion to list*.

The Vector class can be found on github in the ***vector.py*** file: https://github.com/mdiannna/Labs_UTM_AI/blob/main/Lab2/vector.py.

Useful functions in utils.py

The ***utils.py*** file contains some util functions for calculating distances and getting the positions of the rocks:

- **calcDistance(pos1, pos2, distance="euclidean")** - calculates the distance between 2 objects (now only euclidean distance)
- **calcDistanceWithRadius(pos1, pos2, radius, distance="euclidean")** - calculates the distance between 2 objects (now only euclidean distance), but takes into account the radius of the object as well, assuming that both objects have the same radius

They are also presented on github in the ***utils.py*** file: https://github.com/mdiannna/Labs_UTM_AI/blob/main/Lab2/utils.py.

Other code changes

The first change to existing simulation was to duplicate the *Sprite* class, rename it to ***Boid*** and add the required methods for flocking behaviour. It is present in the ***boid.py*** file, as well as the Annex 1.

It is also needed to change the **last 2 lines of the rock_spawner() function**, so that an asteroid/rock is no longer a standard *Sprite*, but a *Boid*, as described in the next listing:

Listing 7: Changes to the rock_spawner function

```
    # last 2 lines of the rock_spawner() function
    a_rock = Boid(newpos, newvel, newang, newangvel, asteroid_image, asteroid_info)

    rock_group.add(a_rock)
```

One of the most substantial changes to the existing simulation code represents the changing of process_sprite_group function, in the update part. The code for the updated function is presented in the Listing 8.

Listing 8: The updated process_sprite_group function

```
def process_sprite_group(sprite_group, canvas):
    """Function to draw sprites on canvas, update them and delete those who
    became old"""
    remove_sprites = set([])
    is_boid = False

    # checks if object is a boid
    if len(sprite_group)>0 and type(list(sprite_group)[0])==Boid:
        is_boid = True

    for sprite in sprite_group:
        sprite.draw(canvas)

        if is_boid:
            other_objects = explosion_group.copy()
            other_objects = other_objects.union(missile_group)
            other_objects.add(my_ship)

            upd_sprite = sprite.update(sprite_group, my_ship, other_objects) #
    sprite_group will contain all boids (rocks)
        else:
            upd_sprite = sprite.update()

        if upd_sprite: # update returns True if the sprite became old, else
    False
            remove_sprites.add(sprite)

    if len(remove_sprites): # if something needs to be deleted..
        sprite_group.difference_update(remove_sprites)
```

Conclusions

This report presents an implementation of the Flocking behaviour based on the Boids [3] paper written by Craig Reynolds, as well as other behaviours based on this model, which was added to a game simulation.

The algorithms that were implemented are mostly based on vectors and operations with vectors, so the linear algebra operations proved to be very useful in simulating flocking behaviour and calculating the directions, velocities, forces and other parameters.

The implementation of the simulation will help colleagues at "LemML" to solve their problem by simulating the intelligent rocks' behaviour and finding strategies to attack them.

The addition of the flocking behaviour to the game simulation with starship and rocks makes the game become more interesting and more similar to how the objects would move in the real life. The evading behaviour and the attacking behaviour, which change from time to time, make the game even more interesting (and harder), because at some point the rocks might behave in an evading way, and afterwards start attacking the ship.

Finally, it can be concluded that complex behaviours added to games such as flocking behaviour, evading and attacking behaviour can increase the level of similarity to the real world movements and make the games more interesting, and make the objects or enemies in the game seem more "intelligent". They can also be used in simulating real life behaviours to some extent and help find strategies based on simulations.

References

[1] Diana Marusic. Source code for the laboratory work. Accessed February 19, 2021. https://github.com/mdiannna/Labs_UTM_AI/tree/main/Lab2.

[2] UTM Fundamentals of Artificial Intelligence Course

[3] Boids by Craig Reynolds. https://www.red3d.com/cwr/boids/ Accessed February 19, 2021.

[4] Steering behaviours by Craig Reynolds. https://www.red3d.com/cwr/steer/ Accesed February 19, 2021.

[5] Simulating Bird Flock Behavior in Python Using Boids. https://medium.com/better-programming/boids-simulating-birds-flock-behavior-in-python-9fff99375118 Accessed February 19, 2021.

Appendix1 - The Boid class implemented in the boid.py file

Listing 9: The final full Boid class

```python
class Boid:
    def __init__(self, pos, vel, ang, ang_vel, image, info, sound = None):
        self.pos = [pos[0],pos[1]]
        self.vel = [vel[0],vel[1]]
        self.acceleration = [0,0]
        self.max_velocity = 2
        self.max_force = 0.25
        self.perception = 250
        self.behaviour_driver = BehaviourChangeDriver('normal')
        self.delta_s_change_behaviour = 7 #will change bbehaviour every 7 seconds

        self.angle = ang
        self.angle_vel = ang_vel
        self.image = image
        self.image_center = info.get_center()
        self.image_size = info.get_size()
        self.radius = info.get_radius()
        self.lifespan = info.get_lifespan()
        self.animated = info.get_animated()
        self.age = 0

        if sound:
            sound.rewind()
            sound.play()

    def get_pos(self):
        """
        get position of boid
        returns:
            list[2] - current position of boid
        """
        return self.pos

    def get_radius(self):
        """ get radius of boid """
        return self.radius

    def draw(self, canvas):
        """ draw boid on canvas """
        if self.animated:
            new_image_center = [self.image_center[0] + self.age * self.image_size[0], self.image_center[1]]
```

```python
            canvas.draw_image(self.image, new_image_center, self.image_size, self.pos, self.image_size, self.angle)
        else:
            canvas.draw_image(self.image, self.image_center, self.image_size, self.pos, self.image_size, self.angle)

    def separation(self, all_boids):
        """
        steer to avoid crowding neighbours (short range repulsion)
        ----------
        parameters:
            all_boids(set) - the set of all boids
        ----------
        returns:
            steering(Vector) - the steering vector to add to acceleration of boid
        """

        #min distance between rocks required
        delta_distance = self.perception
        steer_vector = Vector(0, 0) # must be 2D vector
        avg_steer = Vector(0,0)
        steering = Vector(0,0)

        cnt_close_neigh = 0
        all_positions = get_all_positions(all_boids)

        for rock_i_pos in all_positions:

            dist_i_j = calcDistanceWithRadius(rock_i_pos, self.pos, self.radius)


            if self.pos!=rock_i_pos and dist_i_j < delta_distance:
                #If rocks are intersecting, we consider the distance to be very very small, while the "diff" will still be the distance btw centers
                # so that the final "force" will be big
                if dist_i_j==0: #means rocks are intersecting
                    small_dist = 0.000001
                    dist_i_j = small_dist
                    diff = Vector(*self.pos) - Vector(*rock_i_pos)
                else:
                    diff = Vector(*self.pos) - Vector(*rock_i_pos) - self.radius * Vector(2,2)

                diff /= dist_i_j
                avg_steer += diff
                cnt_close_neigh +=1

        if cnt_close_neigh > 0:
```

```
            avg_steer /= cnt_close_neigh

            steering = avg_steer - Vector(*self.vel)

        steering = self.adjust_steering_force(steering)

        return steering


    def alignment(self, boids):
        """
        steer towards the average heading of neighbours
        ----------
        parameters:
            boids(set) - the set of all boids
        ----------
        returns:
            steering(Vector) - the steering vector to add to acceleration of boid
        """
        avg_vector = Vector(0,0)
        steering = Vector(0,0)
        cnt_boids_in_perception = 0

        for boid in boids:
            distance = calcDistanceWithRadius(boid.pos, self.pos, self.radius)

            if (distance < self.perception):
                avg_vector += Vector(*boid.vel)
                cnt_boids_in_perception +=1

        if cnt_boids_in_perception>0:
            avg_vector = avg_vector / cnt_boids_in_perception
            avg_vec_normalized = (avg_vector / avg_vector.norm())
            steering = avg_vec_normalized * (self.max_velocity) - Vector(*self.vel)

        steering = self.adjust_steering_force(steering, delta_force=-0.01)

        return steering


    def cohesion(self, all_boids, delta_force=-0.01):
        """
        steer to move towards average position of neighbours (long range attraction)
        ----------
        parameters:
            boids(set) - the set of all boids
```

```
            delta_force(float) - the max_force adjustment to add to increase or decrease max force
        ----------
        returns:
            steering(Vector) - the steering vector to add to acceleration of boid
        """
        steering = Vector(0,0)
        cnt_boids_in_perception = 0
        center_mass = Vector(0,0)

        for boid in all_boids:
            distance = calcDistanceWithRadius(boid.pos, self.pos, self.radius)
            if (distance < self.perception):

                center_mass += Vector(*boid.pos)
                cnt_boids_in_perception +=1

        if cnt_boids_in_perception >0:
            center_mass = center_mass / cnt_boids_in_perception
            diff_to_center_vect = center_mass - Vector(*self.pos)  - self.radius * Vector(1,1)

            if diff_to_center_vect.norm() >0:
                diff_to_center_vect = (diff_to_center_vect/diff_to_center_vect.norm()) * self.max_velocity

            steering = diff_to_center_vect - Vector(*self.vel)

        steering = self.adjust_steering_force(steering, delta_force=delta_force)

        return steering


    def adjust_steering_force(self, steering, delta_force=0):
        """
        function to adjust steering force - if the force exceeds max fore, then it is decreased
        -------
        parameters:
            steering(Vector) - the steering vector
            delta_force(float) - the max_force adjustment to add to increase or decrease max force
        returns:
            steering(Vector) - the steering vector with force adjusted
        """
        if steering.norm() > self.max_force:
            steering = (steering / steering.norm()) * (self.max_force+delta_force)
```

```python

        return steering


    def keep_on_screen(self):
        """ Keeps the object inside visible screen """
        for i in range(DIMENSIONS):
            self.pos[i] %= CANVAS_RES[i]


    def add_steer(self, steer):
        """
        adds steering to acceleration
        ----------
        parameters:
            steer(Vector) - the steering vector
        """
        if type(steer)==Vector:
            steer = steer.to_list()

        for i in range(DIMENSIONS):
            if steer[i]>0:
                self.acceleration[i] += max(1, int(steer[i]))

            elif steer[i]<0:
                self.acceleration[i] += min(-1, int(steer[i]))


    def add_negative_steer(self, steer):
        """
        adds negative steer to acceleration
        ----------
        parameters:
            steer(Vector) - the steering vector
        """
        if type(steer)==Vector:
            steer = steer.to_list()

        for i in range(DIMENSIONS):
            if steer[i]>0:
                    self.acceleration[i] -= max(1, int(steer[i]))
            elif steer[i]<0:
                self.acceleration[i] -= min(-1, int(steer[i]))



    def flocking_behaviour(self, all_boids):
        """
        the function implementing flocking behaviour
```

```
        ----------
        parameters:
            all_boids (set) - the set of all boids
        """
        align_steer = self.alignment(all_boids)
        cohesion_steer = self.cohesion(all_boids)
        sep_steer = self.separation(all_boids)

        self.add_steer(align_steer)
        self.add_steer(cohesion_steer)
        self.add_steer(sep_steer*2)



    def attacking_behaviour(self, other_objects, coef_steer=3):
        """
        implementing attacking behaviour - thrust asteroids towards a starship, aiming to collide and destroy them
        ----------
        parameters:
            all_boids (set) - the set of all boids,
            coef_steer(int) - the coefficient to multiply the steering
        """
        cohesion_steer = self.cohesion(other_objects, delta_force=4)
        self.add_steer(cohesion_steer*coef_steer)


    def evading_behaviour(self, other_objects, coef_steer=5):
        """
        implementing evading behaviour - trying to evade any unknown objects, like starships and their missiles
        ----------
        parameters:
            all_boids (set) - the set of all boids,
            coef_steer(int) - the coefficient to multiply the steering
        """
        separation_steer = self.separation(other_objects)
        self.add_steer(separation_steer*coef_steer)


    def update(self, all_boids, ship, other_objects):
        """
        update boid position and parameters
        --------
        parameters:
            all_boids (set) - the set of all boids
            ship (Ship) - the ship object
            other_objects(Sprite or obj) - other objects in the game, like ships, missiles etc
```

```
        ----------
        returns:
            _(bool) - True if the sprite is old and needs to be destroyed
        """

        if time.time() - self.behaviour_driver.last_time_behaviour_changed >= self.delta_s_change_behaviour:
            self.behaviour_driver.change_behaviour()
            self.behaviour_driver.last_time_behaviour_changed = time.time()
            print("current_beh:", self.behaviour_driver.behaviour)

        self.flocking_behaviour(all_boids)

        if self.behaviour_driver.behaviour=="attacking":
            self.attacking_behaviour([ship])

        elif self.behaviour_driver.behaviour=="evading":
            self.evading_behaviour(other_objects)


        self.pos = (Vector(*self.pos) + Vector(*self.vel)).to_list()
        self.vel  = (Vector(*self.vel) + Vector(*self.acceleration)).to_list()
        velocity_vect = Vector(*self.vel)

        if velocity_vect.norm() > self.max_velocity:
            self.vel = ((velocity_vect / velocity_vect.norm()) * self.max_velocity).to_list()

        self.acceleration = [0,0]

        self.keep_on_screen()
        self.age   += 1

        # return True if the sprite is old and needs to be destroyed
        if self.age < self.lifespan:
            return False
        else:
            return True

    def collide(self, other_object):
        """
        Method that takes as imput a sprite and another object (e.g. the ship, a sprite)
        and returns True if they collide, else False
        ----------
        parameters:
            other_object(Sprite or object) - other object to check if collision
        """
        distance = dist(self.pos, other_object.get_pos())
```

```python
        sum_radii = self.radius + other_object.get_radius()

        if distance < sum_radii:
            return True
        else:
            return False
```