SwimSense

Boid Algorithm Implementation with Obstacle Avoidance

AE4350

Quentin Missinne

Delft University of Technology

SwimSense

Boid Algorithm Implementation with Obstacle Avoidance

by

Quentin Missinne

Student Name	Student Number
Quentin Missinne	5941962

Instructor:	Dr.ir. E. van Kampen
Project Duration:	Q4 - 2024
Faculty:	Faculty of Aerospace Engineering, Delft
GitHub Link:	https://github.com/QMissinne/SwimSense/tree/main

Cover: Shark swimming through a school of fish - Photographer unknown
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
F_COUNT	Amount of fish spawned
F_RANGE	Minimal distance required to detect neighboring fish
F_SPEED	Speed with which fish swim
O_COUNT	Amount of obstacles spawned
O_RADIUS	Size of the obstacle
O_RANGE	Minimum distance required for fish to perceive obstacle

1 Introduction

Swarm intelligence is the collective behaviour observed in decentralized, self-organized systems which is often seen in natural phenomena. This behaviour emerges from local interactions between individuals and builds into complex, adaptive and efficient group dynamics. Notable examples of such behaviour can be seen in bird flocking [4], ant swarms [3] and fish schooling [7]. Due to the decentralized nature of these structures, complex tasks can be achieved through individuals with limited computational power.

Consider a school of fish which swims in a dynamic phalanx structure, whilst continuously varying in length, width and shape [7]. Although these structures are visually appeasing, it has been found that when fish are in large shoals they are not only safer from predators [8], but also find food more efficiently [6]. Although much research has been conducted into how such emergent intelligence occurs, the pursuit of algorithms which mimic their behavior is at an all time high. One such algorithm is known as the boid algorithm [5]. This algorithm essentially aims to simulate a generic flocking system where-in each individual 'boid' behaviour is dictated by the following three laws:

1. **Separation:** Steering to avoid crowding local flockmates.
2. **Alignment:** Steering towards the average heading of local flockmates.
3. **Cohesion:** steering to move towards the average position of local flockmates.

In an attempt to simulate a swarming school of fish, this study will aim to implement a version of the boid algorithm in a static environment with varying degrees of complexity. Initially a school of fish will be generated at random positions where-in each fish is given the three basic boid laws. The implementation of this algorithm will be briefly described in Chapter 2. Next the performance of the algorithm will be evaluated in order to see how well it copes with varying detection ranges (both fish-to-fish detection and fish-to-obstacle detection) described in Chapter 2.2. There-after the performance of the algorithm will be analysed in Chapter 3 followed by a brief discussion and conclusion in Chapters 4 and 5, respectively.

2

Method

This Chapter covers the method used to implement the boid algorithm as well as how it will be evaluated. First, a brief overview of the environment will be given in Chapter 2.1. There-after the evaluation metrics will be explained in Chapter 2.2.

2.1. Implementation

In this sub-Chapter an overview of the key implementations will be briefly covered. First in Chapter 2.1.1 the visual environment will be described. Following this, the obstacle class and fish class will be covered in Chapters 2.1.2 and 2.1.3, respectively.

2.1.1. The Environment

The algorithm was deployed in PyGame (a set of python modules designed for writing games [2]). The environment consists of a 1200-by-800 pixel light blue background where-in two PyGame objects co-exist. The black circles represent the obstacles, for which the size and influence range can be varied. The dark blue arrow-heads represent the fish. The orientation of the fish is determined by the direction in which the arrow-head is pointing.

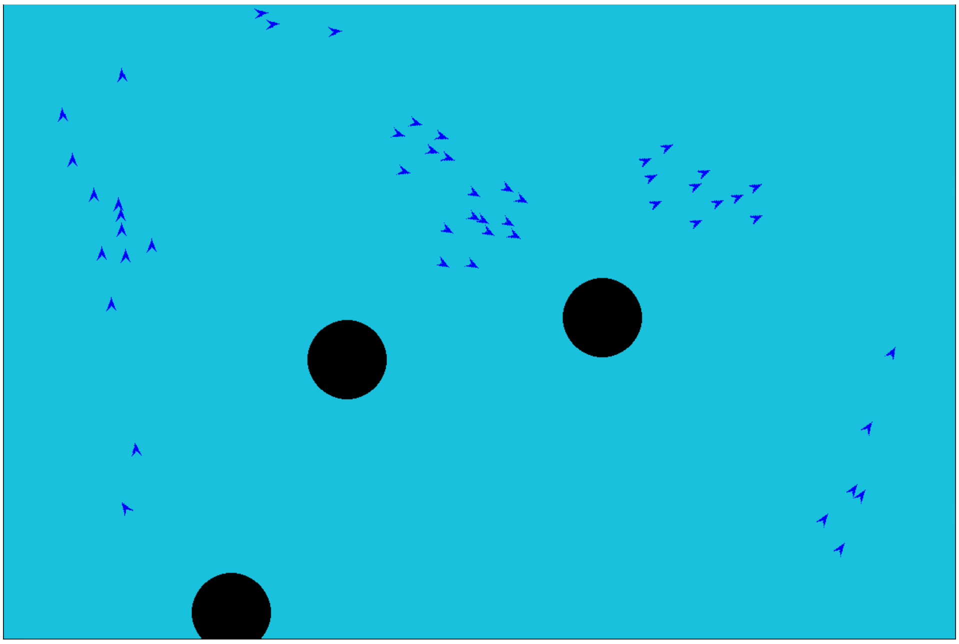

Figure 2.1: Environment in which the fish and obstacles co-exist. the black circles represent obstacles, the dark blue arrowheads represent the fish.

2.1.2. The Obstacles

The obstacles are defined by circles with a fixed radius *O_RADIUS* (a variable the user can change manually). These circles are then generated at a random (x, y) coordinate within the environment window and added to a list and stored alongside their centroid positions. In addition to the obstacle radius, an influence range *O_RANGE* is also used. This allows the user the manually set the distance from which the fish will first identify the obstacle and begin deviating its path.

2.1.3. the Fish

The fish class is built on four main methods. First the basic fish movement is encoded, for which its behaviour (when alone or near an obstacle) is called. Following this, the boid laws the fish will have to follow once a nearby shoal is identified is described. there after a description of the logic the fish must follow in order to avoid the obstacles. Lastly, the method which combines all these behaviors together is explained. Whilst there are other methods within this class, these four are the fundamental methods which encode the boid logic, as well as the obstacle avoidance.

update

The update method is responsible for refreshing the fish's state based on the time that has passed *dt* as well as the speed of the fish *F_SPEED*. Furthermore, it manages the fish's interaction with the environment (flocking and obstacle avoidance). The method first identifies nearby fish and processes their positions and heading, such that cohesion and alignment can be calculated. Following this, an appropriate turning direction is calculated which is then used to update the fish's position and angle. Lastly the position of the fish is determined by applying the movement defined in the move method, as well as a method called *wrap()*[1].

boid logic

This method implements the core logic from the boids algorithm. It first calculates the turn direction necessary for the fish to align with the nearby shoal, whilst maintaining a separation distance to avoid collisions (the fish will re-orient itself if collision is bound to happen). The method first checks the proximity of the nearest fish and then adjusts the target vector accordingly. It then calculates the difference between the fish's current angle and desired angle (through the average angle of nearby fish) and determines whether a turn is needed.

move

This method is responsible for updating the fish's position and orientation based on its calculated velocity (turn direction and speed). It adjusts the fish's angle based on the number of nearby fish and determines the turning direction. The method then calls the obstacle avoidance method (described here-after) ensuring it does not collide with the obstacles. Lastly, the fish's position is updated based on the current direction and speed (which is then used to rotate the fish by it's new angle).

obstacle avoidance

The obstacle avoidance method calculates an appropriate steering angle to avoid collisions with nearby obstacles. The method first checks if the distance between the fish and each obstacle is within the predefined *O_RANGE*. If the fish is within this range, the fish's trajectory is adjusted to steer away from it. The method does this by computing a tangent vector to the obstacle, adjusts its direction based on the fish's current orientation and then calculates a new steering angle to guide the fish away from the obstacle.

2.2. Evaluation Method

In order to evaluate the implementation two key parameters will be varied. In Chapter 2.2.1 an evaluation of the swarming intelligence will be described. Following this in Chapter 2.2.2, a method to evaluate the obstacle avoidance capabilities will be described.

[1] wrap() is a method developed to allow the fish to wrap through the screen. The fish once they pass through the left side of the screen would then appear on the right hand side of the screen (and vice-versa). The same interaction exists with the top of the screen. This was done to allow the fish to swim smoothly through the environment without constantly bouncing off the display edges. [2]

2.2.1. Swarming Evaluation Method

In this study the fish move with Brownian motion until a neighboring fish is found with-which it could form a shoal. In order to evaluate the performance of the algorithm the distance at which the fish will be able to identify neighboring fish will be varied, alongside the distance required for it to swarm with the other fish. Initially *F_RANGE = 50* (see Figure 2.2), and will be gradually increased in steps of size 10 until *F_RANGE = 200* as shown in Figure 2.3. For each step size, the simulation will be run 5 times. A run will continue until all fish have found a neighboring fish (forming a shoal). From this an elapsed time is computed for each individual run, which will then be averaged over each *F_RANGE*.

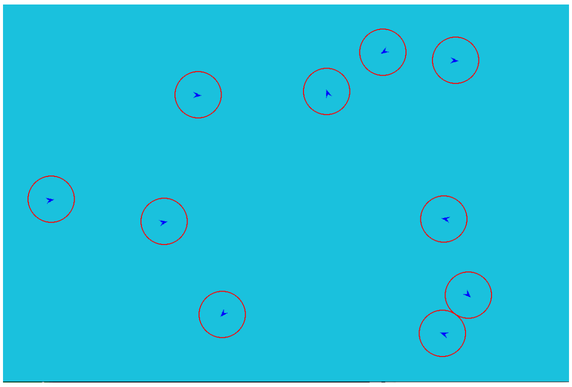

Figure 2.2: Fish with F_RADIUS=50

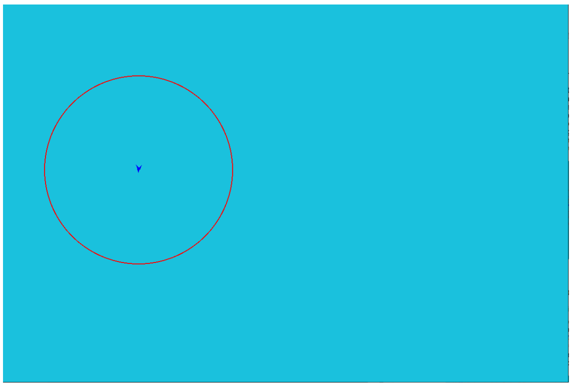

Figure 2.3: Fish with F_RADIUS=100

2.2.2. Obstacle Avoidance Evaluation

The obstacle avoidance method will be evaluated by varying the range at which the fish identify the obstacle, and how well it adjusts accordingly. The centroid of the fish is defined as the center of the 20-by-20 square surrounding it. Once this crosses into the the black of the obstacle, a collision will be detected and documented. Initially three obstacles (randomly located for each iteration) with *O_RANGE=100* are taken with *O_RADIUS=10*. This can be seen in Figure 2.4. Gradually *O_RADIUS* will be increased by 10 until *O_RANGE = O_RADIUS*, as shown in Figure 2.5. For each iteration, the simulation will be run 5 times for a duration of 30 seconds. During this time each collision will be recorded. Afterwards, an average for the amount of collisions per *O_RADIUS* in order to get a more accurate approximation for each *O_RADIUS*.

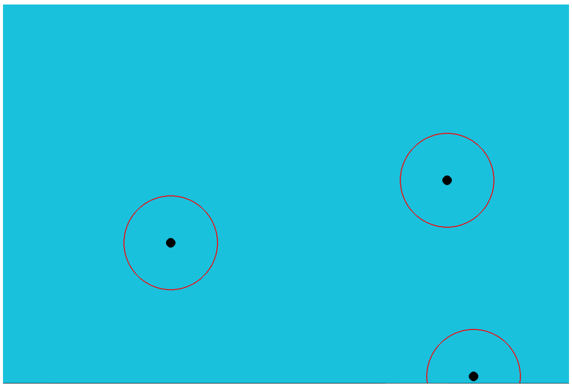

Figure 2.4: Obstacles with O_RADIUS=10

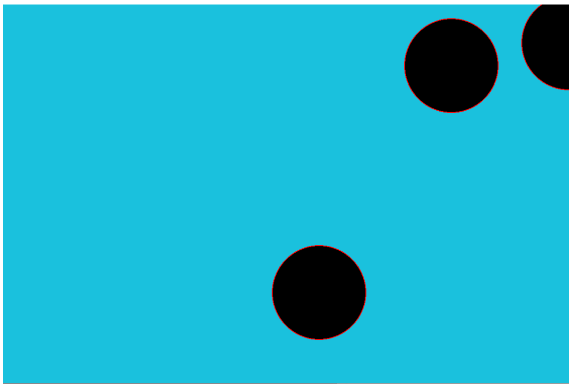

Figure 2.5: Obstacles with O_RADIUS=100

3
Results

The results will be split into two sub-chapters. In Chapter 3.1 the swarming evaluation from Chapter 2.2.1 will be analysed. Following this, in Chapter 3.2, the results from the obstacle avoidance evaluation described in Chapter 3.2, will be reviewed.

3.1. Swarming Evaluation

The fish swarming was evaluated by computing the amount of time required for every fish to associate with a shoal at varying ranges (over several iterations at each range). Figure 3.1 displays the average elapsed time for each *F_RANGE*.

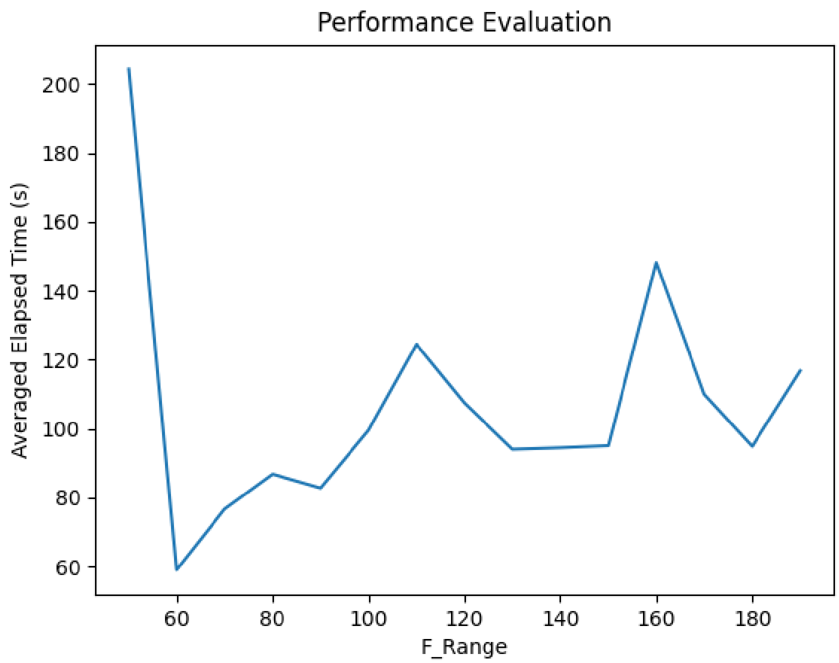

Figure 3.1: Graph showing the average time elapsed before all fish found a shoal to follow for 50 < F_RADIUS < 200

From this graph it is clear that the best performance is when *F_RANGE=60*, with the worst performance being with *F_RANGE=50*. This progression however does not align with the rest of the readings generated for the *F_RANGE* values that follow. What can be seen is an increase in the average elapsed time as the detection radius increases. This seems converse to what may be initially hypothesised.

3.2. Obstacle Evaluation

As previously described the behaviour of the fish around obstacles was evaluated by varying the ratio of obstacle size to the distance from which fish could react to the the obstacle. This was done by keeping *O_RANGE = 10* and gradually increasing *O_RADIUS* from 10 to 100 with a step size of 10. for each *O_RANGE* the simulation was run 5 times for 30 seconds, and the collisions were accumulated. Below in Figure 3.2 the average behaviour for each *O_RANGE* is visualised.

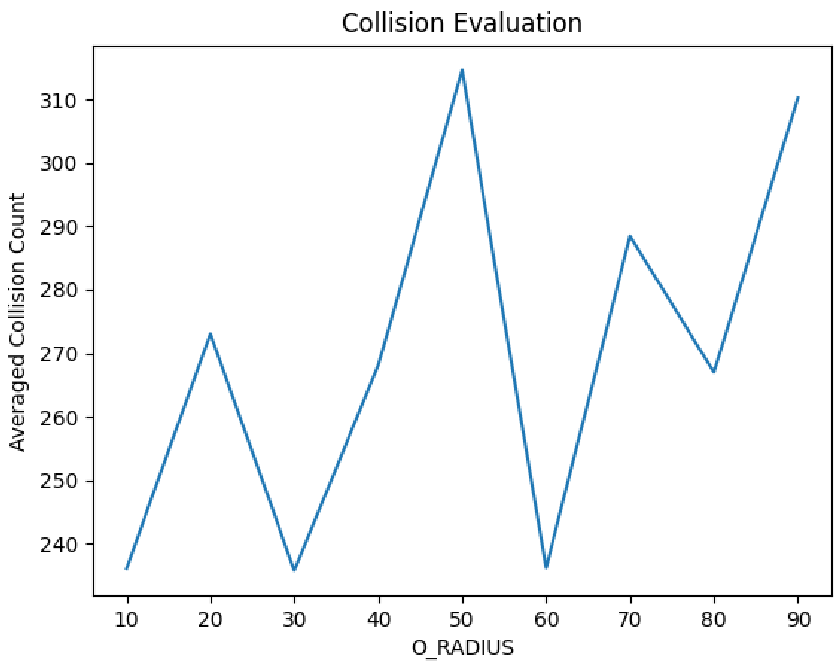

Figure 3.2: Graph showing the average collision rate for varying obstacle dimensions and range to radius ratio.

In this simulation there is a large peak in collisions at *O_RADIUS=50*, where as the least collisions happen when *O_RADIUS=30*. There seems to be a trend that the larger the obstacle becomes (and the closer the ratio of *O_RADIUS : O_RANGE* comes to 1:1) the worse the obstacle avoidance performs.

4 Discussion and Future Work

4.1. Discussion

4.1.1. Swarming

From the results, it is safe to assume that for a smaller *F_RADIUS* performs better than a larger one. This could be because of several reasons. First and foremost the fish stick to those that are closest to them, and are less likely to be influenced by another large shoal passing by. Furthermore, *F_RADIUS* influences all three boid laws, which means that it fewer fish will be perceived and only those very close will be aligned with. Similarly, cohesion is affected as for a smaller *F_RANGE* There are fewer perceived fish, which means a smaller perceived center of mass (faster cohesion).

Additionally it is important to consider the effect of the wrap method. Although this method does take away from fish bouncing off all the walls it does make the grouping harder. For instance, consider a fish trailing the end of a shoal as the rest of the fish pass through the top of the screen and appear at the bottom. This will now leave the trailing fish all alone at the top, without any nearby fish to socialise with. If at this moment a new heading is calculated that sends the fish away from general heading of the shoal then it is now isolated once more. This phenomena occurred several times when watching the environment.

4.1.2. Obstacle avoidance

Evidently, the obstacle avoidance could use some serious improvement, as it does not fully work (there are still many collisions). Although there are simpler implementations that exist (such as simply reversing the direction when approaching the obstacle, so that it just bounces off), the existing method was used as it is more accurate to the actual motion of fish [1]. For this reason the tangent vector was calculated in order to try and make fish circumvent the obstacle and retain their general heading. This definitely needs some tuning and could be improved.

It is also extremely important to mention that in some cases, the obstacle algorithm does struggle with the combination of obstacle avoidance and the wrap method. Consider the case where an obstacle is located on the left edge of the window, and a shoal is swimming through the right edge of the display at an equal height as the obstacle. Due to the way the algorithm is written, the fish do not consider what could be on the opposite side of the screen, and therefore a collision is bound to happen. Although the fish do swim out, there is still a collision that is registered.

4.2. Future Work

This project has a lot of potential in terms of future additions. Whilst there are the more obvious improvements which this project could benefit from (a more refined obstacle avoidance method, better swarm implementation so that the algorithm can handle having 200+ agents, etc), there are entire projects which can benefit from such an implementation.

An interesting concept to consider would be to add a shark class, who's movements would be dictated entirely by the schools of fish. For instance, it could be modelled with the following three laws:

- seek: find a school of fish
- pursuit: pursue the nearest fish in the school of fish
- avoid: avoid obstacles

These three laws can be taken a step further where the shark could identify lone fish (those who are not part of a shoal) and attempt to catch them before they rejoin a shoal. The fish would also need to model their interaction with the sharks, where they would have to avoid collisions with the shark (similar to obstacles), although in this case collision would mean fatality. This can be further augmented by making the shark a reinforcement learning agent which learns to isolate one fish in order to eat it. Such an agent could be modelled in several ways. One of which would be with a reward function based purely on the isolated fish. Alternatively, a reef shark's behavior could be mimicked, where a pack of sharks would have to work together to try and trap fish within A reef where they could easily pick them out.

Alternatively, the project could be further improved by adding non-convex obstacles such that the fish would need to learn actual obstacle avoidance for situations where simply circumventing a circular object no longer holds. Lastly I would suggest also adding an option to control whether the algorithm wraps through the screen, as it could be interesting to consider cases where that is not an option. As stated previously, the effects of wrapping through the screen do have some negative effects on the performance of the algorithm.

5 Conclusion

In conclusion, SwimSense is a boid algorithm implementation designed to be customize. There are many variables which can be adjusted in order to vary the behaviour of the fish in the environment. the parameters that can be modified in order to change the behaviour can be found in the the README.md file in the GitHub repository.

In this particular study, the swarming behaviour and obstacle avoidance were analysed. In both cases surprising results were established. For swarming the fish seemed to swarm better when their influence radius was smaller. After analysing the results it became clear that this is because both alignment and cohesion are positively influenced by smaller influence radii. As for Obstacle avoidance, it was found that higher collisions occurred when the obstacle radius was closer to the obstacle range. This means that when the fish had less time to perceive the obstacle, they were more likely to hit it. Lastly a brief overview of future work was provided, where-in specific improvements were mentioned as well as a few interesting projects which could originate from the groundwork that SwimSense provides.

References

[1] Theodore Castro-Santos and Alex Haro. “Fish guidance and passage at barriers”. In: *Fish locomotion: An eco-ethological perspective* (2010), pp. 62–89.

[2] Pygame Community. *Pygame Documentation*. https://www.pygame.org/docs/. 2024.

[3] Chrysostomos Fountas. “Swarm Intelligence: The Ant Paradigm”. In: vol. 2. Sept. 2010, pp. 137–157. ISBN: 978-3-642-13354-1. DOI: 10.1007/978-3-642-13355-8_9.

[4] Ashish Girdhar. “Swarm Intelligence and Flocking Behavior”. In: Mar. 2015.

[5] Iztok Lebar Bajec, Nikolaj Zimic, and Miha Mraz. “The computational beauty of flocking: Boids revisited”. In: *Mathematical and Computer Modelling of Dynamical Systems - MATH COMPUT MODEL DYNAM SYST* 13 (Aug. 2007), pp. 331–347. DOI: 10.1080/13873950600883485.

[6] Tony J. Pitcher, Anne E. Magurran, and I. J. Winfield. *Fish in larger shoals find food faster*. URL: https://link.springer.com/article/10.1007/BF00300175#citeas.

[7] Ying Tan and Zhong-yang Zheng. “Research Advance in Swarm Robotics”. In: *Defence Technology* 9.1 (2013), pp. 18–39. ISSN: 2214-9147. DOI: https://doi.org/10.1016/j.dt.2013.03.001. URL: https://www.sciencedirect.com/science/article/pii/S221491471300024X.

[8] M. Zheng et al. “Behavior pattern (innate action) of individuals in fish schools generating efficient collective evasion from predation”. In: *Journal of Theoretical Biology* 235.2 (2005), pp. 153–167. ISSN: 0022-5193. DOI: https://doi.org/10.1016/j.jtbi.2004.12.025. URL: https://www.sciencedirect.com/science/article/pii/S0022519305000056.

A

Source Code SwimSense - Generic

```python
import pygame as pg
from pygame.locals import *
import random
from random import random, randint
import math
from math import pi
from math import cos, sin, radians, degrees, atan2

import time
import matplotlib.pyplot as plt

# initialise pygame:
pg.init()

# ---------------------------------------------------------------------------
# Setup:
# ---------------------------------------------------------------------------
vec = pg.math.Vector2
clock = pg.time.Clock()

# Display:
GRID = False
OBSTACLES = False
WIDTH = 1200
HEIGHT = 800
FPS = 60
BACKGROUND = (26, 193, 221)

# Obstacle Parameters:
O_RANGE = 100
O_RADIUS = 25
O_COUNT = 1

# Fish setup:
RECT_FISH = True
RADIUS = True
CLOSEST_OBSTACLE = False
F_COUNT = 50
F_SPEED = 100
F_RANGE = 100

FramePerSec = pg.time.Clock()

displaysurface = pg.display.set_mode((WIDTH, HEIGHT), pg.DOUBLEBUF | pg.HWSURFACE)
displaysurface.fill(BACKGROUND) #(26, 193, 221)
pg.display.set_caption("The Reef")
# ---------------------------------------------------------------------------
# Grid Class:
# ---------------------------------------------------------------------------
```

```python
class Grid():
    """
    The grid class is used to seperate the environment into a grid of cells
    purely for visualisation purposes.
    """
    def __init__(self):
        self.cell_size = 50
        self.grid = {}

    def get_cell(self, pos):
        return (int(pos.x / self.cell_size), int(pos.y / self.cell_size))

    def visualize_grid(self):
        for x in range(0, WIDTH, self.cell_size):
            pg.draw.line(displaysurface, (0, 0, 0), (x, 0), (x, HEIGHT))
        for y in range(0, HEIGHT, self.cell_size):
            pg.draw.line(displaysurface, (0, 0, 0), (0, y), (WIDTH, y))

grid = Grid()

# ---------------------------------------------------------------------------
# Obstacle Class:
# ---------------------------------------------------------------------------
class Obstacle(pg.sprite.Sprite):
    """
    The obstacle class is used to define the obstacles within the reef.
    This will create a barrier for the fish to navigate around. The obstacles
    will be defined as black circles with a radius of O_Radius (variable).
    """
    def __init__(self, x , y, radius=O_RADIUS):
        super().__init__()
        self.surf = pg.Surface((radius*2, radius*2), pg.SRCALPHA).convert_alpha()
        pg.draw.circle(self.surf, (0, 0, 0), (radius, radius), radius)
        self.rect = self.surf.get_rect(center=(x, y))
        self.pos = vec(x, y)

    def create_obstacles():
            obstacles = []
            obstacle_positions = []
            for _ in range(O_COUNT):
                x = randint(0, WIDTH)
                y = randint(0, HEIGHT)
                obstacle = Obstacle(x, y)
                obstacles.append(obstacle)
                obstacle_positions.append((x, y))
            return obstacles, obstacle_positions


# ---------------------------------------------------------------------------
# Fish Class
# ---------------------------------------------------------------------------
class Fish(pg.sprite.Sprite):
    """
    Fish class used to describe the fish swarming behaviour according to
    the boids algorithm. the fish will move according to the following rules:
    - seperation: steer to avoid crowding local flockmates;
    - alignment: steer towards the average heading of local flockmates;
    - cohesion: steer to move towards the average position of local flockmates;
    The fish will also need to avoid the obstacles presented within the reef.
    """
    def __init__(self):
        super().__init__()
        self.surf = pg.Surface((20, 20), pg.SRCALPHA).convert_alpha()
        self.original_surf = self.surf
        self.color = (0, 0, 255)
        points = ((4, 18), (10, 2), (16, 18), (10, 12), (4, 18))
        pg.draw.polygon(self.surf, self.color, points)
        self.rect = self.surf.get_rect()
        self.rect.center = (10, 10)
        self.angle = randint(-180, 180)
        self.vel = vec(F_SPEED, 0).rotate(-self.angle)
```

```
        self.in_obstacle = [False for _ in range(O_COUNT)]
        self.school_count = 0


    def update(self, dt, speed, F_RANGE=F_RANGE):
        self.wrap()
        turn_direction = 0
        x_position = 0
        y_position = 0
        sin_angle = 0
        cos_angle = 0
        nearby_fishes = sorted([fish for fish in fishes if vec(fish.rect.center).distance_to(self.rect.center) < F_RANGE and fish != self],
                               key=lambda i: vec(i.rect.center).distance_to(self.rect.center))
        del nearby_fishes[5:]

        self.school_count = len(nearby_fishes)
        if self.school_count > 1:
            nearest_fish = vec(nearby_fishes[0].rect.center)
            for fish in nearby_fishes:
                x_position += fish.rect.centerx
                y_position += fish.rect.centery
                sin_angle += sin(radians(fish.angle))
                cos_angle += cos(radians(fish.angle))
            target_vector = (x_position / self.school_count, y_position / self.school_count)
            average_angle = degrees(atan2(sin_angle, cos_angle))
            self.boid_logic(nearest_fish, average_angle, target_vector)
            turn_direction = self.boid_logic(nearest_fish, average_angle, target_vector)
        self.move(dt, self.school_count, turn_direction, speed)
        self.rect.center = self.pos

    def boid_logic(self, nearest_fish, average_angle, target_vector):
        threshold_nearest_fish = 10
        if vec(self.rect.center).distance_to(vec(nearest_fish)) < threshold_nearest_fish:
            target_vector = nearest_fish
        else:
            target_vector = self.rect.center

        difference = vec(target_vector[0] - self.rect.center[0], target_vector[1] - self.rect.center[1])
        target_distance, target_angle = difference.as_polar()

        if target_distance < F_RANGE:
            target_angle = average_angle
        else:
            target_angle = self.angle

        angle_difference = (target_angle - self.angle)
        if abs(angle_difference) > 1:
            turn_direction = angle_difference
        else:
            turn_direction = 0

        if target_distance < F_RANGE and target_vector == nearest_fish:
            if turn_direction < 0:
                turn_direction = turn_direction + 180
            else:
                turn_direction = turn_direction - 180
        return turn_direction

    def move(self, dt, school_count, turn_direction, speed=F_SPEED):
        turn_rate = 2
        if school_count == 0:
            self.angle += randint(-5, 5)

        if turn_direction != 0:
            self.angle += turn_rate * abs(turn_direction) / turn_direction

        steering_angle = self.obstacle_avoidance()
        diff_angle =  self.angle - steering_angle
```

```
        if abs(diff_angle) > 70 and abs(diff_angle) < 90:
            scaling_factor = 1.75
        else:
            scaling_factor = 1.2
        if self.angle > 90  and self.angle < -90:
            self.angle += diff_angle*scaling_factor
        else:
            self.angle -= diff_angle*scaling_factor

        self.rect = self.surf.get_rect(center=self.rect.center)
        self.dir = vec(1, 0).rotate(-self.angle).normalize()
        new_pos = self.pos + self.dir * (speed + (5 - school_count)**2) * dt
        self.pos = new_pos
        self.angle = degrees(atan2(-self.dir.y, self.dir.x))
        self.surf = pg.transform.rotate(self.original_surf, self.angle - 90)
        self.rect = self.surf.get_rect(center=self.rect.center)

    def obstacle_avoidance(self, OBSTACLE_RANGE=O_RANGE):
        steering_angle = self.angle
        sin_obstacle = 0
        cos_obstacle = 0
        if OBSTACLES == True:
            for i_obstacle, obstacle in enumerate(obstacles):
                distance = obstacle.pos.distance_to(self.rect.center)
                if distance < O_RANGE:

                    direction_to_obstacles = obstacle.pos - self.pos
                    tangent_vector = vec(- direction_to_obstacles.y, direction_to_obstacles.x
                        ) # counterclockwise
                    scalar_prod = tangent_vector.x * cos(radians(self.angle)) +
                        tangent_vector.y * sin(radians(self.angle))
                    if scalar_prod < 0:
                        tangent_vector = - tangent_vector
                    obstacle_angle = tangent_vector.as_polar()[1]
                    sin_obstacle += sin(radians(obstacle_angle))
                    cos_obstacle += cos(radians(obstacle_angle))
                    steering_angle = degrees(atan2(sin_obstacle, cos_obstacle))

        return steering_angle

    def find_nearest_obstacle(self, obstacles):
        min_distance = float('inf')
        nearest_obstacle = None

        for obstacle in obstacles:
            distance = vec(obstacle.pos).distance_to(self.pos)
            if distance < min_distance:
                min_distance = distance
                nearest_obstacle = obstacle
        return nearest_obstacle.pos if nearest_obstacle else None

    def lerp(self, a, b, t):
        return a + (b - a) * t

    def wrap(self):
        if self.rect.left < 0:
            self.rect.right = WIDTH
            self.pos.x = self.rect.centerx
        if self.rect.right > WIDTH:
            self.rect.left = 0
            self.pos.x = self.rect.centerx
        if self.rect.top < 0:
            self.rect.bottom = HEIGHT
            self.pos.y = self.rect.centery
        if self.rect.bottom > HEIGHT:
            self.rect.top = 0
            self.pos.y = self.rect.centery

    def draw(self, surface=displaysurface):
        surface.blit(self.surf, self.rect)
```

```
fish = Fish()

# ----------------------------------------------------------------------------
# Spawn Fish:
# ----------------------------------------------------------------------------
def is_position_in_obstacle(pos, obstacles):
    for obstacle in obstacles:
        if pos [0] > obstacle.rect.left - 20 and pos[0] < obstacle.rect.right + 20:
            if pos[1] > obstacle.rect.top - 20 and pos[1] < obstacle.rect.bottom + 20:
                return True
    return False

def spawn_fish(obstacles):
    while True:
        pos = randint(0, WIDTH), randint(0, HEIGHT)
        if not is_position_in_obstacle(pos, obstacles):
            fish = Fish()
            fish.rect.center = pos
            fish.pos = vec(pos)
            return fish

# ----------------------------------------------------------------------------
# Game Loop:
# ----------------------------------------------------------------------------
for _ in range(O_COUNT):
    obstacles, obstacle_positions = Obstacle.create_obstacles()

fishes = []

for _ in range(F_COUNT):
    new_fish = spawn_fish(obstacles)
    fishes.append(new_fish)

start_time = time.time()

while True:
    dt = clock.tick(FPS) / 1000
    for event in pg.event.get():
        if event.type == QUIT or (event.type == KEYDOWN and event.key == K_ESCAPE):
            pg.quit()
            pg.exit()

    displaysurface.fill(BACKGROUND)

    if OBSTACLES == True:
        for obstacle in obstacles:
            displaysurface.blit(obstacle.surf, obstacle.rect)
            pg.draw.circle(displaysurface, (255, 0, 0), obstacle.rect.center, O_RANGE, 2)

    for fish in fishes:
        fish.draw(displaysurface)
        fish.update(dt, F_SPEED)
        if RECT_FISH == True:
            pg.draw.rect(displaysurface, (0, 255, 0), fish.rect, 2)
        if RADIUS == True:
            pg.draw.circle(displaysurface, (255, 0, 0), (int(fish.pos.x), int(fish.pos.y)),
                F_RANGE, 2)
        if CLOSEST_OBSTACLE == True:
            if OBSTACLES == True:
                pg.draw.line(displaysurface, (0, 255, 0), fish.pos, fish.
                    find_nearest_obstacle(obstacles), 2)

    if GRID == True:
        grid.visualize_grid()

    print(time.time() - start_time)

    pg.display.update()
    FramePerSec.tick(FPS)
```