Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Modeling group behaviors for videogames

Vojtěch Nápravník

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
January 2025

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Nápravník Vojtěch** Personal ID number: **516114**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Games and Graphics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Modeling group behavior for videogames

Bachelor’s thesis title in Czech:

Modelování skupinového chování pro videohry

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **08.02.2025** Deadline for bachelor thesis submission: _____

Assignment valid until: **20.09.2026**

_____
Head of department’s signature

_____
prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

_____
Date of assignment receipt

_____
Student’s signature

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Nápravník Vojtěch** Personal ID number: **516114**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Games and Graphics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Modeling group behavior for videogames

Bachelor’s thesis title in Czech:

Modelování skupinového chování pro videohry

Guidelines:

Review methods for modeling group behavior suitable for use in video games. Focus on methods using steering behaviors [1]. Describe these methods in detail, including mathematical formulation of steering force calculation and associated pseudocode.
Implement the selected group behavior modeling method for movement in 2D space containing static and dynamic colliders.
The implementation will support different character types that will directly influence the behavior of the nearby characters (predator and prey behavior).
Use the implementation in a simple game where a scuba diver catches fish (moving in flocks) while occasionally competing with predator fish. Verify the simulated group behaviors, such as fish flocking, by comparing them to videos of real-world flocks.

Bibliography / sources:

[1] Reynolds, C. W. (1999). Steering behaviors for autonomous characters. In Proceedings of Game Developers Conference, 763-782.
[2] Ježowicz, F. (2023). Iteraktivní instalace s podmořskými flokujícími zvířaty pro děti. Bachelor's thesis, CTU FEE.
[3] Popelová, M. (2011). Knihovna steering technik pro virtuální agenty. Bachelor's thesis, MFF UK.
[4] Gargula, M. (2023). Simulation of artificial intelligence for games. Bachelor's thesis, CTU FEE.
[5] Singh, S., Kapadia, M., Faloutsos, P., Reinman, G. (2009). Steerbench: a benchmark suite for evaluating steering behaviors. Computer Animation and Virtual Worlds, 20(5□6), 533-548.
[6] Silveira, R., Dapper, F., Prestes, E., Nedel, L. (2010). Natural steering behaviors for virtual pedestrians. The Visual Computer, 26, 1183-1199.

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s): Nápravník Vojtěch
Personal number: 516114
Programme name: Open Informatics

declare that I have elaborated the bachelor's thesis entitled

Modeling group behavior for videogames

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I did not use any artificial intelligence tools during the preparation and writing of my thesis. I am aware of the consequences if manifestly undeclared use of such tools is determined in the elaboration of any part of my thesis.

In Prague on 16.05.2025

Vojtěch Nápravník
.....
student's signature

Acknowledgement

I would like to thank my supervisor, doc. Ing. Jiří Bittner, Ph.D., for letting me work on a game related topic, his guidance and continuous support throughout the course of this thesis. His insightful feedback and regular consultations helped me stay on track and significantly improved the quality of the final result.

I am also very thankful to my family, friends and girlfriend for their encouragement, support during my studies and taking their time to test the game prototypes.

Abstract

Non-playable characters (NPCs) are important for creating immersive gaming experiences by simulating a world that feels alive at all times. Achieving this often involves implementing artificial intelligence that challenges the player and exhibits natural and realistic behaviors. This project explores the development of such behaviors through the implementation of *steering behaviors* and a *flocking* algorithm. These techniques are designed to ensure that NPCs interact organically with the environment and the player. The result is a prototype of a 2D diving and fishing game, where these behaviors form the foundation of gameplay mechanics, demonstrating their potential to enhance game dynamics and realism.

Keywords: Group behaviors, Steering behaviors, Fish simulation

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Praha 2,
Karlovo náměstí,
E-421

Abstrakt

Nehráčské postavy (NPC) jsou důležité pro vytvoření pohlcujícího herního zážitku tím, že simulují svět, který je neustále živý i když hráč zrovna není aktivní. Dosažení tohoto cíle často zahrnuje implementaci umělé inteligence, která je pro hráče výzvou a projevuje přirozené a realistické chování. Tento projekt se zabývá vývojem takového chování prostřednictvím implementace tzv. *steering behaviors* a tzv. *flocking* algoritmu. Tyto techniky jsou navrženy tak, aby zajistily přirozenou interakci NPC s prostředím a hráčem. Výsledkem je prototyp 2D hry s potápěním a chytáním ryb, kde tato chování tvoří základní herní mechaniky a demonstrují svůj potenciál pro zlepšení herní dynamiky a realističnosti.

Klíčová slova: Skupinové chování, Řízení, Simulace ryb

Překlad názvu: Modelování skupinového chování pro videohry

Contents

Chapter 1

Introduction

Most games nowadays have some form of non-playable characters (NPCs or agents. These are essential to make the player feel that the world around them exists independently, even without them actively interacting with it. This can be achieved through the use of artificial intelligence and well-designed NPCs.

Implementing algorithms that play *optimally* and outperform the user is common, a different question arises: How can I create NPCs that do not simply play perfectly, but behave *naturally*? How can I program complex behavior that appears *organic* and *believable*? How can I ensure that such behaviors will remain effective in different environments?

1.1 Goals

In this thesis, I will investigate how to create behaviors for agents in a 2D game. Specifically, I will focus on complex lifelike movements and strategies for fish in changing environments. I will implement these behaviors using the steering behaviors proposed by Craig W. Reynolds [REY+99].

The goals of this thesis are to:

- Describe and explore methods of group behavior simulation.
- Use steering behaviors to create natural motion for the fish in a procedurally generated world.
- Create a game in which the player will be able to catch the fish in their net.
- Compare the simulation to a video of real movement of fish.

Chapter 2

Related work

In this chapter, we will dive into why it is beneficial for animals to create groups, how ants are great at finding short paths, how hundreds of agents can navigate through complicated terrains, and how simple steering behaviors can be used to create complicated patterns.

2.1 Flocks, herds and schools

Herds, flocks, schools, shoals are all commonly used terms to describe a group of animals. They are often mixed up, below I will try to make the differences clear.

In biology, groups of birds are called *flocks*, a group of mammals is called a *herd*, and a group of fish swimming together in a coordinated way is called a *school*. *Shoal* is a broad term used to describe a group of fish that stay together for social reasons, such as feeding and mating, while *schooling* is a behavior within *shoal*. *Schools* are a specific type of shoal, where the movement of the fish is highly coordinated and synchronized. [PIT98]

In computer graphics, *schools, flocks and herds* are commonly modeled using a flocking algorithm. Therefore, the term flock in computer graphics refers to all types of groups of agents moving together as a single body. In this work, the terms flock and school will be tightly tied together since we will be working with a flocking algorithm and modeling the schooling behavior of fish.

2.1.1 The tendency to create groups

Most animals, have a tendency or a need to create groups. The main reason for this behavior is *survival*. According to Landa [LAN98], living in a group has many advantages such as:

- *Risk dilution:* Each individual is less likely to be picked off by a predator when moving in a large group.
- *Food searching:* Searching for food in a group requires each individual to spend less energy.

- *Ease of movement:* Moving in a fish school is easier thanks to the surrounding increased water flow by the other fish.
- *Camouflage:* Fast-moving individuals of different colors or textures limit the predator's ability to focus on a specific individual.

Each individual acts in a way to increase his own chance of survival, however their individual goal benefits the whole group. This shows how multiple individuals, each acting on their own, create a certain global pattern. This phenomenon is one of the main characteristics of a group behavior.

2.2 Ant colony optimization

In this section we will discuss the Ant colony optimization technique proposed by Dorigo [DBS06].

In nature, ants living in colonies have perfected a method to find optimal pathways to food sources. The ants do this by randomly stumbling into a food source, and once they do, they leave a pheromone trail on their way back to the colony. Other ants can smell this, stop moving at random, and move along the pheromone trail navigating to the previously discovered food source. If they manage to find the food source, they again start leaving the pheromone trail, reinforcing the previously followed trail while offering a slightly different path to the food source. Since each ant moves along its own path before it finds the pheromone trail, multiple paths to the food source are discovered. The pheromone trails evaporate over time and only the continuously reinforced parts of the pheromone trails stay around longer. Effectively "forming" the individual trails into a narrower trail. No single ant would find the optimal path to the food source, but the colony as a group will find the optimal path.

In optimization algorithms, ant colony optimization (ACO) is commonly used to find optimal paths through graphs. Dorigo [DBS06] proposed ACO as a possible optimal solution to the traveling salesman problem. The traveling salesman problem boils down to a simple question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" [CON25c]. The straightforward solution is to find all possible permutations of the towns and pick the one with the smallest sum of distances between the towns. However, this approach runs in O(n!), where n is the number of towns, making this approach practically unusable. Ants, on the other hand, solve this problem every day.

To solve the traveling salesman problem, the issue is represented as finding a sequence of nodes in a fully connected graph with the lowest sum of costs of the edges. The ACO for the traveling salesman problem is described in Algorithm 1.

Algorithm 1 Ant Colony Optimization for TSP

```
    // Use small positive constant on all edges
 1: Initialize pheromone_level on all edges
 2: for each iteration until max_iterations do
 3:     for each ant do
 4:         Place ant at a random starting_vertex
 5:         Initialize tour with starting_vertex
 6:         while not all vertices visited do
    // Calculate the desirability of all unvisited vertices
 7:             for each unvisited_vertex do
 8:                 desirability = Pow(1 / dist_to_vertex, distance_bias) *
    Pow(pheromone, pheromone_bias)
 9:             end for
10:             Pick the next_vertex randomly weighted by the desirability
11:             Append next_vertex to tour
12:             Deposit pheromone on the used edge
13:         end while
14:         Return to the starting_vertex to complete the tour
15:         Save the tour and its tour_length along with the ant
16:     end for
17:     for each edge do
    // Combine newly deposited pheromone with existing value using the
    length of the path constructed by the ant
18:         for each ant that used this edge do
19:             pheromone = pheromone + Q / ant.tour_length
20:         end for
    // Evaporate pheromone
21:         pheromone = (1 - evaporation_rate) * pheromone
22:     end for
23:     Update best_tour if a shorter one was found
24: end for
25: return best_tour
```

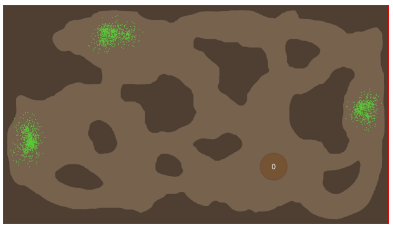
The optimization relies on multiple parameters being set up:

- *distance_bias:* How much do the ants prefer the closer nodes when picking the next node to visit.
- *pheromone_bias:* How much do the ants prefer the nodes, that were reinforced by visits of the other ants when picking the next node to visit.
- *evaporation_rate:* How quickly the pheromones evaporate with each iteration.
- *Q:* How large is the deposit of pheromones by each ant.

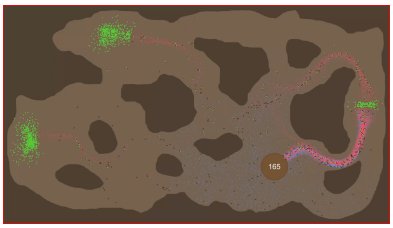
Tinkering with these parameters is an important part of the method; each problem can benefit from different settings.

In the work of Dorigo [DBS06], three variants of the ACO discussed: *Ant System*, *MAX-MIN Ant System* and *Ant Colony Systems*. Each variant specifies how the pheromones at each place should be updated when another pheromone is placed and how the evaporation is handled. In Algorithm 1 the Ant system variant is shown.

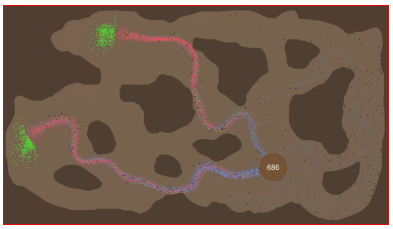
The behavior of the ants form a pleasing group behavior. The ideas can be used in an animation or in a game. In a video by Sebastian Lague [LAG21] a large map was created with multiple ants, their nest, and multiple food sources. The pheromone trail was implemented by emitting a pheromone particle in constant periods of time. The ants then slowly over time steered toward the pheromones. As seen in Figure 2.1, after a short period of time, the ants managed to find an optimal path between food sources.



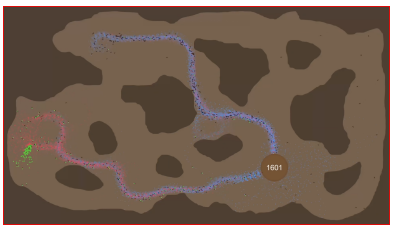
(a) A) Start of the simulation. Green dots represent food sources, the brown circle marks the nest.



(b) B) The right cluster is running out, and the other two are being discovered.



(c) C) The right cluster is depleted, and the other two are the primary clusters.



(d) D) The cluster at the top is depleted, the red pheromone trail there is fading out. There are no pheromones left where the right cluster was previously.

Figure 2.1: Visualization of ant colony optimization in a game-like environment. Green dots represent food sources, blue dots are left behind by ants when searching for food, and the red dots are left behind when they find food source (source: [LAG21].

2.3 Flow fields

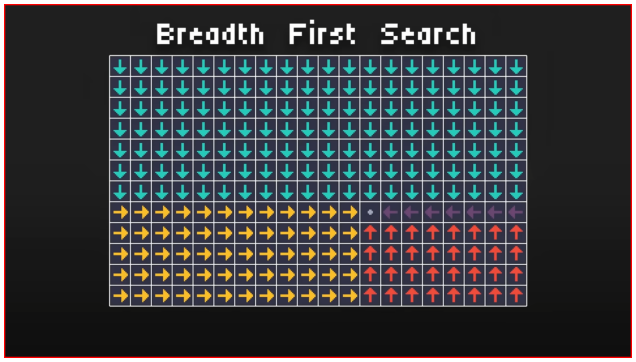
Often, in games, simulating large numbers of NPCs is needed. These NPCs often need to navigate toward a specific target. Finding a path to a target is simple when using one of the pathfinding algorithms mentioned later in

Section 4.2.2. However, these algorithms, when run multiple times by a large number of agents, can slow down the game drastically, moreover, NPCs will often pathfind to the same target from a similar starting position, or at least they will end up using a similar portion of a path as a different agent. This is where flow fields are a great solution.

Imagine a grid, where each cell of the grid contains a vector. This vector points toward a cell that is closer to the target. Then, an agent can, at any time, look up where in the grid its position is represented and use the associated vector to move toward the target without the need to precisely calculate the exact path from its position to the target position.

The flow field calculation typically incorporates a breadth-first search (BFS) algorithm that will visit every cell of the grid. The BFS algorithm is explained and discussed in Section 4.2.2. Then, in each cell, the algorithm visits a vector that points to a cell closer to the target (see Figure 2.2a).

Flow fields also work with obstacles, simply by not allowing any vectors to be stored at the position of the obstacle, making the NPCs using the flow field literally flow around the obstacles. Example of flow field with obstacles can be seen in Figure 2.2b.



(a) Example flow field generated using BFS (source: [DEV25] 6:39).

(b) Example flow field with obstacles (source: [DEV25] 7:42).

Figure 2.2: Visualization of the flow fields.

Flow fields are versatile and can simulate multiple aspects, such as difficult-

to-traverse terrains and avoidance of other NPCs. To better understand flow fields, look at the work of Emerson [EME13].

2.4 Steering

In this section, we will discuss work related to the work of Reynolds on steering behaviors [REY+99] and his flocking behavior [REY87]. The work of Reynolds himself will be discussed in detail later in Chapter 5.

In the work of Ježowicz [JEZ23], the flocking algorithm of Reynolds [REY87] (later discussed in Section 5.3) was implemented and an interactive application was created. The application consisted of an underwater fish simulation projected onto a wall and the tracking of the position of the right hand. The right hand was then cast into the scene, and the fish were either repulsed or attracted by it. In the work, different optimization techniques of the flocking algorithm are discussed, primarily focusing on the space partitioning issue during the neighbor search of each agent. The Voronoi neighborhoods, the K-D tree, and the Cube space partition map were discussed in detail.

In the work of Popelová [POP11], all the common steering behaviors proposed by Reynolds [REY+99] are discussed and explained. Additionally, a way to combine these behaviors and method to allow agents to change their movement speed is discussed. The result of the project was a library of implementations of steering behaviors and a graphical application that allowed users to interact with the different behaviors and to tinker with their parameters.

Chapter 3

The game concept

One of the goals of this project is to create a game that incorporates steering behaviors and flocking. The intention was to develop a game that would be suitable for mobile devices. Since controls for mobile games are typically more intuitive when handling 2D characters, I decided to design the game in 2D.

The game concept revolved around the rogue-lite genre, where a diver explores deep waters and catches fish in a fishing net. These fish exhibit different behaviors and have varying stats, such as size, speed, health, etc. The diver faces limitations, including oxygen capacity, inventory capacity, and the size of the fish net.

3.1 Inspiration: Dave the Diver

Dave the Diver is a game similar to the one I have envisioned. Dave the Diver is a 2D pixel-art game that builds on a story-driven gameplay and various mini-games. The core part of the gameplay revolves around exploring underwater environments and using harpoon to catch fish.

Dave is a *diver* by day and by night he *manages a sushi restaurant and a hotel*. The *diving* part of the game incorporates deep-sea exploration and interaction with fish. The *management of the restaurant and the hotel* part is realized by the player competing in mini-games, such as taking orders, preparing foods, breeding fish, etc. Additionally, some more aspects of the game, such as battles with the fish are also implemented as mini-games.

Some notable features of Dave the Diver include

- *Stylized graphics:* The game uses a mixture of pixel art and a 3D parallax effect to make the background seem as an endless ocean.
- *Procedural generation:* Each time the player dies and dives into the ocean again, the environment is randomly generated. Although the world changes with each dive, some features act as anchor points and are placed at the same location, since they are a part of the narrative or crucial for the player to find with ease.
- *Fishing mechanics and progression:* The gameplay loop relies on grinding

through fishing activities. Players can upgrade their diving gear to allow them to explore deeper locations and discover rarer fish species.

Although, *Dave the diver* and my project share the underwater exploration aspect. My project focuses primarily on steering behaviors and group dynamics in fish, with gameplay tailored primarily for mobile devices and a different fishing mechanic.

Figure 3.1: The example visuals of Dave the Diver (source: [MIN23]).

3.2 The game loop

The game loop for the player is built as follows:

- Dive as deep as possible.
- Catch the most valuable fish that fit into the net.
- Fight off or escape predators.
- Explore caves and special locations.
- Tend to oxygen levels, health, inventory capacity, and surrounding pressure.
- Swim back to the surface and sell the collected fish.
- Use earnings to upgrade gear, purchase new tools, and improve the base.

This game loop should balance aspects of exploration, resource management, and progression while still giving the player plenty of opportunities to interact with fish.

3.3 Fish types and their behaviors

The most exciting part of the game should be the interaction and observation of the different fish species. The behaviors of the fish should be unique and lifelike.

The fish should try to school with other fish as they do in nature.

In addition to larger fish schooling with their neighbors, the game should also feature *schools of gobies*, a large number of miniature fish all moving together as one entity, without the need to form a school over time.

Each fish should be able to move on its own, properly interact with the player, interact with the environment, and interact with each other. Individual requirements will be discussed in detail in Chapter 6.3.

3.4 World design

The player is spawned in the base. The base consists of three buildings:

- *The Selling Point:* A place where players can sell everything they have collected during their dive.
- *The Shop with upgrades:* A place where players can buy gear improvements.
- *The fishing hut:* A place where payers can refill their oxygen tank and heal up.

At the edge of the base is the entrance to the ocean. The ocean should contain cave-like structures, such as overhangs and tunnels. Other parts of the ocean should remain mostly clear with smaller obstacles. Each run should be different to improve replayability.

Since the progression system relies on the player trying to dive as deep as possible to find more valuable fish and more interesting places, it is crucial to generate the ocean procedurally. The fish should inhabit the ocean in a predefined way with some species living in the deeper parts of the ocean, while others staying in the more shallow regions.

Chapter 4

Navigation of agents

Navigation is part of an agent's behavior. Agent decides what is the current *goal* - e.g., chase the player, go home, hunt a deer - and a navigational component then figures out how the agent should move. For example, when *hunt a deer* is the goal of the agent, the navigational component then tells the agent how to move towards the deer and how to intersect the deer's movement to align the best attack angle. Navigation can be a complex procedure, since the deer can be far ahead in an obstructed area, finding the shortest path is crucial and once the deer is close enough the method to pursue the deer can be totally different.

When designing a simulation with NPCs the navigational algorithm can be one of the following: precomputed, partially precomputed or real-time. We will discuss these algorithms below. Lastly, we will discuss a way to combine these algorithms.

The navigational component should return a *move step* that satisfies the agent's current *goal* each time it is prompted to.

4.1 Precomputed navigation

Precomputed algorithms have a major advantage in the speed of computation at runtime. At runtime, the program no longer uses a computationally intensive algorithm to find the correct *move step* but uses a precomputed database and can search for the correct *move step* in constant time. Thus, the main part of this algorithm runs before execution, where we are not yet concerned about the computation time because it does not affect the overall sense of the game's flow.

Imagine a game field where an agent can only be in a few limited locations. We, as the game designer, want the agent to be in a specific location at a specific stage. So we can consider manually setting exactly where the agents should be at a certain time, and then the agent will just move in a fixed way as we have defined beforehand. This approach may be appropriate for games with a directed story, where we know exactly how the player will move and how we want the agent to react.

4.2 Partially precomputed navigation

This is a common approach to navigation for agents in games. Agents need a *precomputed structure* to understand the game world, and then the agents can run a *real-time calculation* that finds the move step. In this category, it is most common for agents to find a path between their current position and the desired target and simply follow the previously computed path. In this section, we will focus on *pathfinding* algorithms that *precompute* the *navmesh* first and then run *real-time* calculations to get the path to later follow.

However, not only *pathfinding* algorithms fall into this category. *Flow fields* (Section 2.3) can be another example. The actual *flow field* is precomputed, but the way the agent interprets the vectors from the *flow field* and uses them to move can be far more complex.

4.2.1 Navigational mesh

Pathfinding algorithms require converting the game world into a graph representation, commonly known as a navigational mesh or a so-called *navmesh*. A *navmesh* is a *network* that defines where agents can move within the game world. As shown in the work of Tomek [TOM13], a simple way to create a *navmesh* involves the following steps:

- *Create a grid:* Define a grid with dimensions $M \times N$ and a resolution R, which determines the total number of vertices in the grid. Ensure that all points are evenly spaced.
- *Connect adjacent vertices:* Create edges between all neighboring vertices to establish possible movement paths.
- *Mark walkable and non-walkable areas:* Check each vertex to determine whether it lies on terrain that the agent can move on.
- *Remove non-walkable vertices:* Delete all vertices that fall into non-traversable areas along with their corresponding edges.

With these steps, we have translated the physical game world into a graph representation. However, generating a navmesh can be computationally expensive, especially for large maps or high-resolution grids. Additionally, if the game world changes, for example, a piece of terrain is destroyed, the navmesh must be recalculated, which can lead to lags during gameplay.

4.2.2 Pathfinding

Pathfinding is the process of finding a path from a specific point in space to a destination. This problem is common to various branches of computer science - e.g., navigating robots in space without colliding, navigating using GPS, or finding the shortest path in a graph in more theoretical optimization

problems. In video games, *pathfinding* plays a crucial role in the design of the behavior of autonomous agents [NKO07].

Below, the basic *pathfinding* algorithms are described. All algorithms require the problem to be transformed into the language of finding the cheapest path in the graph, that is, a *navmesh* 4.2.1.

BFS algorithm

A base for more advanced pathfinding algorithms is the breadth-first search (BFS) and its variants, such as Dijkstra's algorithm or A*.

Consider points connected with edges in space, all edges are of equal length. After picking a start and target point, we are trying to find a sequence of edges that is the shortest. That is, we are effectively trying to find a path through the graph that intersects as few points as possible.

Let us imagine that this algorithm is used when sending data between routers in a computer network. We have a starting router and a destination router, but the problem is that there is no direct path between them (no signal reaches them), so we use BFS to find the shortest sequence of routers between which there is a direct path. This will effectively minimize the load on the network because we will be sending data between the smallest possible number of routers. It is essential to note that BFS will find the correct path only if it has a graph (network) in which all edges have the same value (i.e., it does not matter to which router the data is sent to at a given moment because the difference in distance between routers is negligible).

The BFS is shown in described in Algorithm 2, to better understand the steps of the algorithm see Figure 4.1.

Figure 4.1 also indicates an obvious problem with the use of BFS to find the shortest sequence of nodes. The algorithm *visits* too many nodes. Looking at the graph, it becomes obvious that some nodes are much more likely to be in the final sequence of nodes than others. In our example from before, this would cause many *unnecessary packets* to be sent to routers that are too far from the target router. This issue can be solved by using the A* algorithm and a proper heuristic as discussed below in Section 4.2.2, this approach will prefer to send the packets in the general direction of the target router first.

Dijkstra's algorithm

Dijkstra's algorithm is a variant of BFS that allows us to infer the *cost* of edges in the graph. Unlike the pure BFS, we now have the option to define a distance between points that is different from 1. We can also say that BFS is a variant of Dijkstra's algorithm where all the edges have a *cost* of 1.

The algorithm follows the same general steps as BFS but replaces the *FIFO queue* with a *priority queue*. Instead of processing nodes in the order they were discovered, nodes are dequeued based on the *shortest known distance* from the start node. When a neighbor is enqueued to the queue - as in Algorithm 2 - its *priority* is determined by the total *cost* (i.e., distance) from the start node.

Algorithm 2 BFS with path reconstruction

```
1: Record start node
2: Initialize queue with start
3: Initialize visited set
4: Initialize parent map
5: while queue is not empty do
6:     Dequeue a node from queue
7:     if node == target then
8:         // Path reconstruction
9:         Initialize path list
10:        current = target
11:        while current is not start do
12:            Append current to path
13:            current = parent[current]
14:        end while
15:        return reverseOrder(path)
16:    end if
17:    for each neighbor of node do
18:        if neighbor is not in visited then
19:            Enqueue neighbor into queue
20:            Put neighbor into visited
21:            parent[neighbor] = node
22:        end if
23:    end for
24: end while
25: return target is not reachable
```

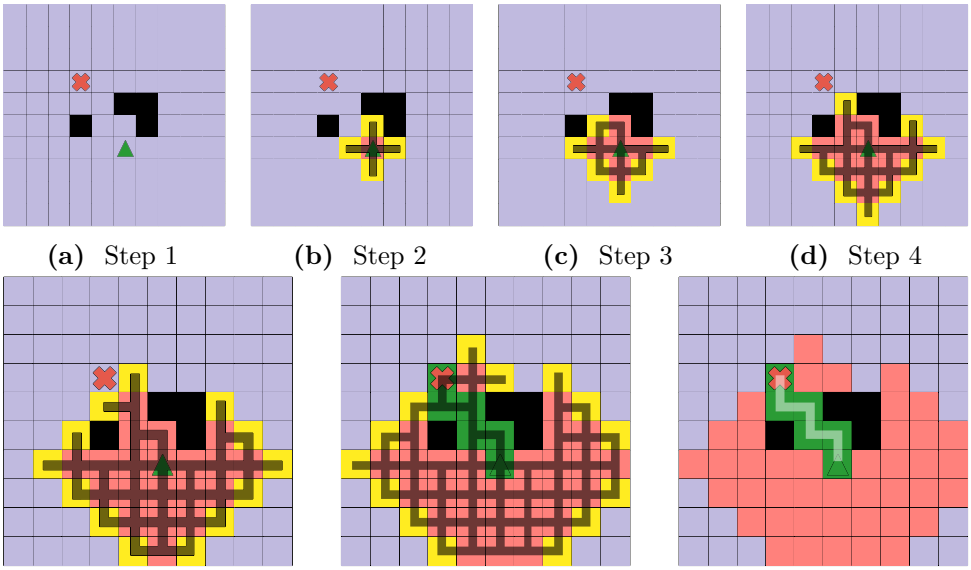
Also, there is a slight change when marking the nodes as visited.

- *In BFS algorithm:* A node is marked as visited when it is *enqueued*.
- *In Dijkstra's algorithm:* A node is only marked as fully visited when it is *dequeued*. If a shorter path to a node is later found, it can be re-added to the queue with the lower cost.

A* algorithm

A* is a variant of Dijkstra's algorithm that improves the efficiency of shortest path search by incorporating heuristics. A heuristic is a rough estimate of the cost required to reach the destination from a given node. By prioritizing nodes that are more likely to be on the optimal path, A* can significantly reduce the number of nodes that need to be processed.

For example, when finding the shortest path between two buildings in a city, we can use the Euclidean distance as a heuristic. This guides the search toward the goal more efficiently compared to exploring all possible paths blindly. The idea is to dequeue the nodes that are in the general direction of the target first and the rest later. The steps of the A* algorithm are the same



(a) Step 1 **(b)** Step 2 **(c)** Step 3 **(d)** Step 4

(e) Step 5 **(f)** Step 6 **(g)** Step 7 - the path reconstruction.

Figure 4.1: Step-by-step visualization of the BFS algorithm. The triangle marks the starting position, the cross marks the goal, black rectangles are obstacles, red blocks mark the visited cells, the yellow cells represent the cells currently in queue.

as Dijkstra's algorithm, but with a modified cost calculation for the nodes in the queue. Instead of prioritizing nodes solely based on their known distance from the start node, A* prioritizes them using the following function:

$$f(n) = g(n) + h(n) \tag{4.1}$$

Where:

- $g(n)$ is the actual cost from the start node to node n.
- $h(n)$ is the heuristic estimate of the remaining cost from node n to the goal (that is, the Euclidean distance from the current building to the target building).

The A* algorithm is a complex tool in optimization techniques that are beyond the scope of this project. For more information on the benefits and pitfalls of heuristics, see [FOE+21; RG20].

4.3 Real-time navigation

The real-time navigation component usually consists of an *environment detection component* that allows the agent to "see" around and an algorithm that interprets and uses what was detected to give precise navigational instructions. The *environment detection component* can be imagined as a

group of sensors that measure data. In games, this is often implemented using *raycasts*. The other part of the algorithm then interprets the measured data and figures out how to navigate the agent. This is similar to how robots navigate in real-life. However, in games, this can be easier to implement because the agent can ask the game for additional data that they might not be able to gather themselves.

We will work on a game where the environment is generated procedurally, can change rapidly and agents do not have a predetermined path nor predefined points of interest. These algorithms are also suitable for games with procedurally generated environments, since they do not rely on any previous knowledge of the map layout.

For our project we will use steering behaviors discussed in Chapter 5.

4.4 Combining algorithms

Typically, agents in most video games do not utilize a single algorithm, but rather combine them in different ways to achieve complex behaviors.

A common pattern occurring in video games, especially stealth games such as Assassin's creed series [UBI23], is the behavior of a Guard. The guard has a predetermined behavior of patrolling between several guarding points and stopping the patrol when the player approaches.

The guard patrolling can be fully precomputed, and the patrolling path can be static. However, the moment the player approaches the guard and the guard notices the player, the real-time component of the behavior is activated. Currently, the guard must navigate the environment in pursuit of the player. Typically, the game world offers a precomputed navmesh enabling the agent to run a pathfinding calculation and perfectly navigate through the environment directly to the player.

However, finding the shortest path to the player at all moments in time is vastly different from real-life. The player would perceive the guard to be cheating, causing an unpleasant experience for the player.

To make the agent's behavior feel more natural, simulating a real-time environment detection is crucial, as discussed in Section 4.3. In the case of the guard, when the player hides behind a corner, the guard should not be able to pathfind toward the player directly, but only to the position where the player was last seen by the guard. Once the guard reaches the last-seen position of the player and is not able to directly see the player after arrival, the guard starts exploring the surrounding environment. For example, checking behind close corners, looking into bushes and aimlessly wandering around. This can go on until the guard either finds the player or loses interest.

This is a common pattern with which players are familiar and expect this type of behavior. However, guard's behavior can be much more complex and challenging depending on the difficulty of the game and desired experience. Games such as Kingdom Come: Deliverance [BOC23] let the guards slightly "cheat" to further challenge the players and make the guards seem smarter. The way Kingdom Come does this is by letting the guard *scan* the surrounding

environment after losing the player and if the *scan* finds the player in proximity, the guard can be hinted to pathfind directly to the player (see Figure 4.2). This prevents players from hiding few steps away from the guard which the guard would realize in real life.

(a) The guard sees the player and navigates toward them.

(b) The player hides and the guard navigates to the player's last known position.

(c) The guard scans the terrain and finds the player.

(d) The guard calculates the exact path to the hidden player.

Figure 4.2: Storyboard explaining the behavior of the *"cheating"* guard.

Separate parts of behaviors, such as patrolling, chasing the player, or returning to base, and the transitions between them, can quickly grow in complexity. Therefore, a structure called a *behavior tree* (see Figure 4.3) can be used to better visualize, understand, and edit the behaviors of agents. The creation, visualization, and usage of *behavior trees* is discussed in the work of Gargula [GAR23].

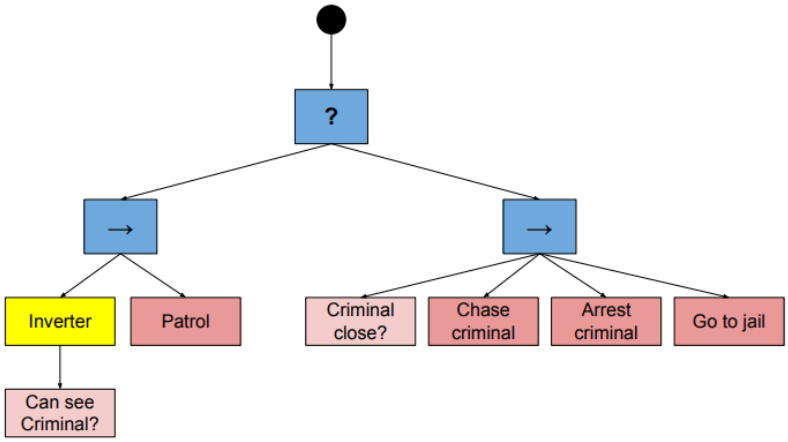


Figure 4.3: A simple behavior tree example (source: [GAR23]). The tree describes the behavior of a guard using predefined nodes. The blue nodes control the flow of behavior: the sequence (arrow) returns true if all child nodes, processed from left to right, return true. The selector (question mark) returns true if at least one of the child nodes returns true. Each leaf node represents an action, and each action returns whether it was successful. The left part of the tree is interpreted as follows: "If *can see criminal* is *false*, start/continue *patrolling*."

Chapter 5

Steering behaviors

In this chapter, we will look at the steering behaviors discussed in the work of Reynolds [REY+99]. We will also discuss issues with the behaviors and how to combine multiple behaviors.

5.1 Terminology

Reynolds focused on the movement and behavior of boids - bird-like creatures that were featured in the Boids program developed by Reynolds in 1986. Boids are typically modeled as vehicles that are free to move in a 2D plane without any obstacles however, there is no reason why the behaviors could not be expanded into 3D. There are three stages to navigation of the agent according to Reynolds [REY+99].

- *Action selection:* The agent decides what they want to do next.
- *Steering:* Calculation of the *desired_velocity* and a *corrective_steering* force to help fulfill the selected action.
- *Locomotion:* The actual execution of the movement of the agent.

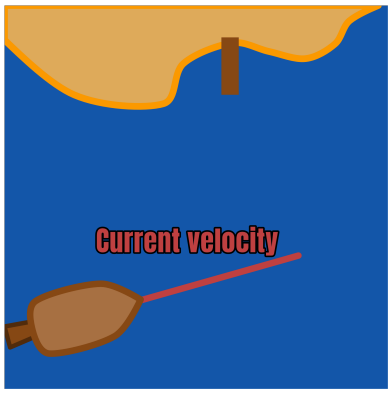
These layers were originally mentioned in the work of Blumberg [BG95], but their purpose remains the same.

To better understand the separate layers I will explain them using an example with a fisherman navigating a boat (scenario shown on Figure 5.1). Imagine a fishing boat with a fisherman. The fisherman's goal is to get to the nearest shore and dock. The fisherman pulls out a map, calculates which shore is the closest, and determines the ideal heading of the boat, i.e., the *desired velocity*. By adjusting the rudder, a *corrective steering force* is applied. However, setting the rudder on its own does not change the heading of the boat. The boat needs to start moving so that the flowing water can push against the offset rudder and gradually change the heading. This is where *locomotion* comes in - the boat's propeller provides the actual movement.

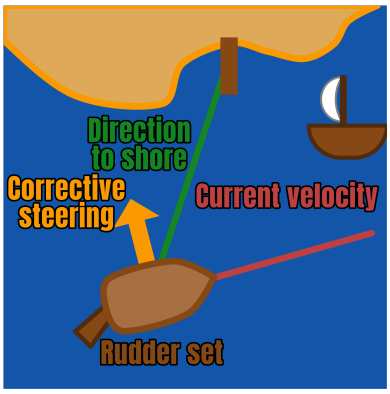
The combination of the rudder (steering force) and the propeller (locomotion) will change the direction of the boat over time until it aligns precisely with the direction of the nearest shore, that is, the original *desired velocity*.

Imagine that as the boat is moving toward the shore, another boat comes in front of the fisherman. The fisherman needs to set a new goal - avoid the boat - whilst ideally making a small enough adjustment to miss the boat and still move in the general direction of the shore.

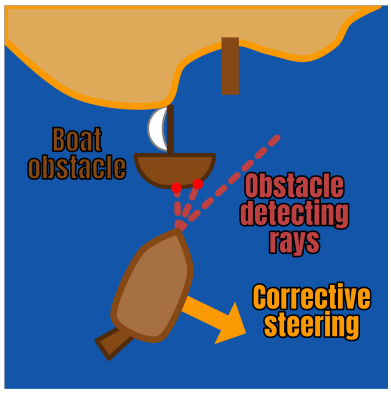
In an ideal scenario, the fisherman would just *set* the heading of the boat instantly to align with the *desired velocity*. In the real world, this is not possible, since the boat has a certain mass and the steering force would need to be much larger than what the rudder can deliver.



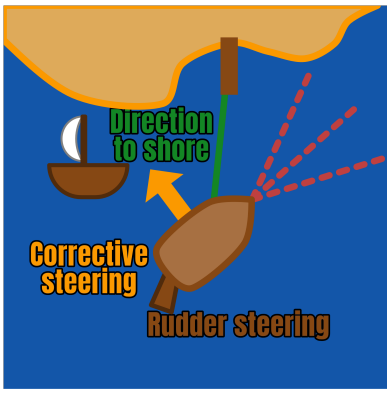
(a) The starting state of the boat.



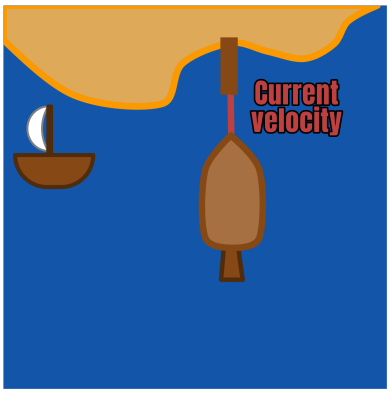
(b) The boat steers to the shore.



(c) The boat detects the approaching boat and tries to avoid it.



(d) The boat steers back toward the shore.



(e) No need for corrective steering anymore.

Figure 5.1: Storyboard explaining the boat scene.

5.2 Common steering behaviors

In this section, we will discuss and explain the steering behaviors presented in the work of Reynolds [REY+99]. The result of each steering behavior should satisfy a particular goal of the boid. The goal is achieved by applying a particular *steering_force*, the *steering_force* is calculated using the *desired_velocity*. The calculation and application of the two components is discussed in detail in Section 5.2.1.

To help clarify the terminology and better understand the commonly used behaviors, below is a breakdown of the individual variables used throughout steering behaviors:

- *position:* current location of the agent
- *velocity:* agent's current movement vector, representing direction and speed
- *desired_velocity:* ideal velocity satisfying agent's desired goal
- *steering_force:* corrective force applied to adjust the *velocity* closer to the *desired_velocity*, calculated as: $desired\_velocity - velocity$; limited by *max_force*
- *acceleration:* result of the applied *steering_force*, divided by the mass
- *mass:* used to simulate inertia

5.2.1 Seek and pursuit behavior

This behavior is useful for agents to help them move toward a target. In our example above 5.1, the fisherman could use the seek behavior to move toward the shore. In that case the target would be stationary; however, the behavior does not require the target to remain stationary. Meaning, the same behavior could be used to navigate toward a moving fish during fisherman's hunting session.

I will use the seek behavior show in Figure 5.2 to explain the simple physical model behind the Reynold's steering behaviors.

At the core of the steering layer is the calculation of the steering force:

$$\mathbf{force}_{\text{steering}} = \text{truncate}(\mathbf{vel}_{\text{desired}} - \mathbf{vel}_{\text{current}}, force_{\text{max}})$$

The steering force calculation 5.2.1 requires a *desired_velocity*. In the case of seek behavior, the *desired_velocity* is a vector pointing from the fisherman's boat toward the fish with a magnitude of the maximum speed of the boat:

$$\mathbf{vel}_{\text{desired}} = \text{set\_magnitude}(\mathbf{pos}_{\text{target}} - \mathbf{pos}_{\text{agent}}, vel_{\text{max}})$$

In an ideal world, the fisherman would immediately set the boat's *current velocity* equal to the *desired_velocity*. This approach does not seem natural.

In real life, the only way to change the velocity of an object is to apply an acceleration. In our case, the acceleration is achieved by applying the *steering_force* to an object, i.e., the boat. The calculation of the acceleration:

$$\mathbf{acceleration} = \frac{\mathbf{force}_{\text{steering}}}{mass}$$

The acceleration then updates the current velocity:

$$\mathbf{vel}_{\text{current}} = \mathbf{vel}_{\text{current}} + \mathbf{acceleration}$$

The velocity is then applied to the position the boat:

$$\mathbf{pos}_{\text{agent}} = \mathbf{agent}_{\text{current}} + \mathbf{vel}_{\text{current}}$$

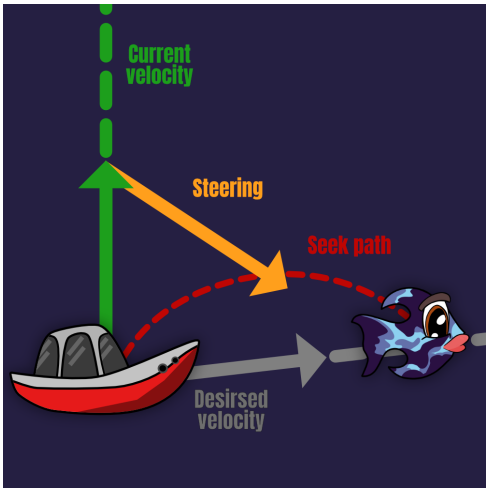


Figure 5.2: Seek steering force calculation diagram.

This approach will ensure gradual acceleration and turning toward the target position, i.e., the fish's position . The rate of these changes is controlled by the *max_force* and *mass*.

The *pursuit* behavior shown in Figure 5.3 is very similar to the *seek* behavior. While the seek behavior calculates the *desired_velocity* using the exact *target_position*, the *pursuit* behavior uses the *expected_position* of the *target* in the *near future*:

$$\mathbf{pos}_{\text{expected}} = \mathbf{pos}_{\text{target}} + \mathbf{velocity}_{\text{target}} * lookahead$$
$$\mathbf{vel}_{\text{desired}} = \text{set\_magnitude}(\mathbf{pos}_{\text{expected}} - \mathbf{pos}_{\text{agent}}, vel_{\text{max}})$$

The target's velocity, $\mathbf{velocity}_{\text{target}}$, can be replaced with a simple heading direction of the target to simulate limited prediction in the pursuit behavior. The *lookahead* parameter determines how far into the future the agent attempts to predict and intersect the target's position.

Usage of *expected_position* in the pursuit behavior improves the likelihood that the agent will intersect the target's trajectory, rather than simple moving toward the *current_position* of the target as in the seek behavior.

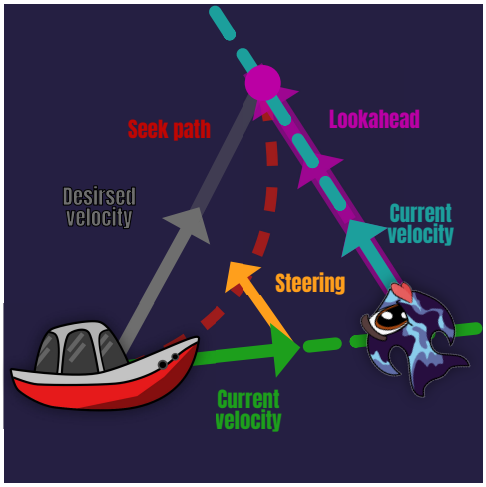


Figure 5.3: Pursuit steering force calculation diagram where the $lookahead = 3$, the purple circle marks the *expected_position*.

5.2.2 Flee and evade

The flee behavior is great to model behavior of a small creature running away from the predator. In a sense, the flee behavior is the opposite of the seek behavior. Basically, $flee\_desired\_velocity = -seek\_desired\_velocity$, that is:

$$\mathbf{vel}_{\text{desired}} = \text{set\_magnitude}(\mathbf{pos}_{\text{agent}} - \mathbf{pos}_{\text{target}}, vel_{\text{max}})$$

Where the pos_{target} is the position of the target the agent flees from, i.e., the position of the predator. The calculated *desired_velocity* is used to determine the steering force as discussed above 5.2.1.

The evade behavior uses the same principle as the pursuit behavior.It is a bit more advanced than the flee behavior allowing the agent to guess where the target will be and flee from that position. The calculation of the desired velocity builds on the flee *desired_velocity* calculation is as follows:

$$\mathbf{pos}_{\text{expected}} = \mathbf{pos}_{\text{target}} + \mathbf{velocity}_{\text{target}} * lookahead$$
$$\mathbf{vel}_{\text{desired}} = \text{set\_magnitude}(\mathbf{pos}_{\text{agent}} - \mathbf{pos}_{\text{expected}}, vel_{\text{max}})$$

5.2.3 Obstacle avoid

As one of the main goals for this thesis I set out to create a steering behavior that will help our agents to avoid obstacles. Craig Reynolds already proposed

such behavior in his paper. We will try to expand on this behavior in upcoming chapters. First, let us break down the original obstacle avoidance steering behavior shown in Figure 5.4.

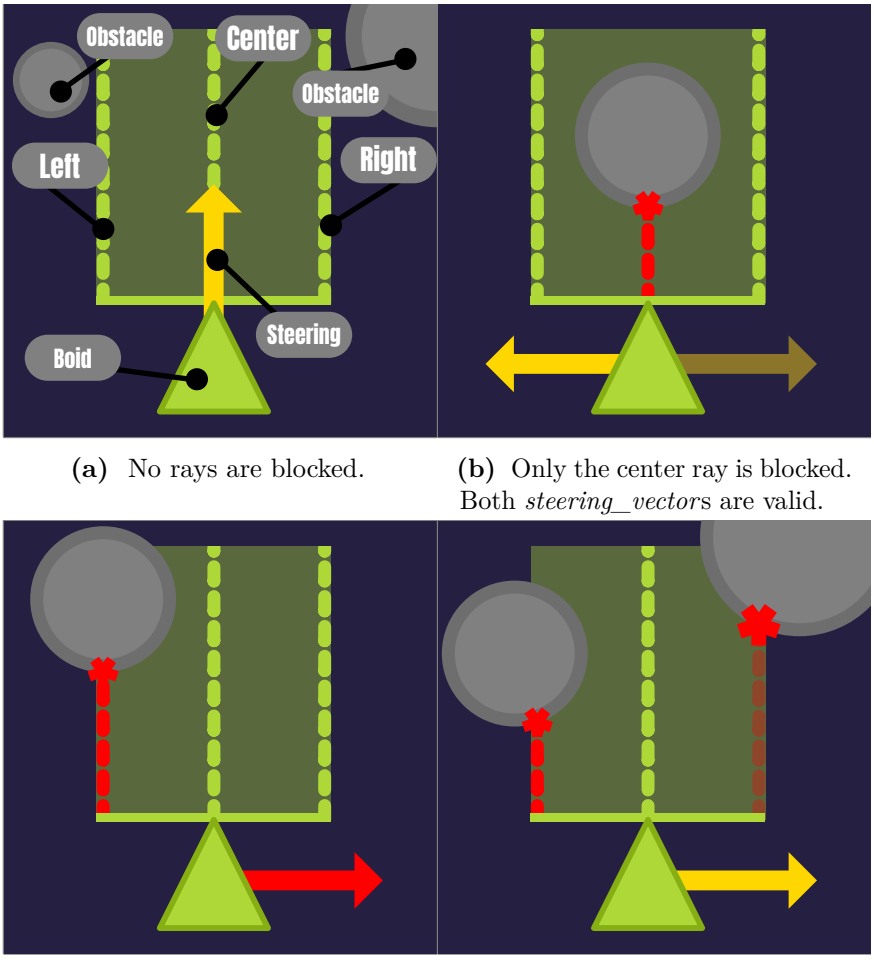
The algorithm is often simplified and approximates all the obstacles with spheres. The general idea is to cast three rays: center, left, and right. The steering behavior evaluates the following cases to calculate the appropriate *steering_vector*:

- *No rays are blocked:* The *steering_vector* is set to the zero vector (no steering is required). See Figure 5.4a.
- *Only the center ray is blocked: steering_vector* is set to the left or right, depending on the clearer path. See Figure 5.4b.
- *The left (or right) ray is blocked:* The *steering_vector* is set to the opposite side (right or left, respectively). See Figure 5.4c.
- *Both the left and right rays are blocked: steering_vector* is set to the side with the least obstruction (the more distant obstruction). See Figure 5.4d.

The *steering_vector* is then used to calculate the *desired_velocity*:

$$\mathbf{vel}_{\text{desired}} = \text{set\_magnitude}(vector_{\text{steering}}, vel_{\text{max}})$$

The *desired_velocity* is then used in the steering force calculation discussed in Section 5.2.1.



(a) No rays are blocked.

(b) Only the center ray is blocked. Both *steering_vector*s are valid.

(c) The left ray is blocked.

(d) Both rays are blocked. Steer away from the closer obstacle.

Figure 5.4: Cases and appropriate *steering_vectors*(yellow arrow) of the wander behavior.

5.2.4 Wander

In some case we might need an agent to appear as if they are moving "randomly" around the environment. We can imagine such behavior as exploring the surroundings. Reynolds presented a behavior that will achieve such effect - the Wander behavior show in Figure 5.5. One of the challenges for this behavior is to make the movement feel fluent and smooth while allow in the agent to change direction of its movement. We might think of rolling dice every few frames and switching the movement direction at random. Such naive behavior does not look right. In case we roll the dice every second the change in direction is too abrupt, also during the interval between dice rolls the agent moves uniformly in a straight line. We could combat this by rolling the dice more often, but that results in jittery movement. The issue is caused by discrete changes in the direction.

We need a smoother transition between the directions to create nice smooth

turns. This can be achieved by sampling a *perlin noise* function and using the result to change direction every couple frames. This results in smooth and fluid turns. The issue with this implementation is the lack of control for each individual boid. It is difficult to control how big the changes in directions are.

The original wander behavior discussed in the paper by Reynolds [REY+99] offers an elegant, simple, and visually intuitive solution.

We begin by creating a sphere at a fixed distance in front of the boid. The sphere represents the *wandering space* for the boid, from which the boid selects a new movement direction. Within this sphere a *point* is chosen as the *target*. Each frame the sphere moves with the boid to preserve the fixed distance, the *target point* is slightly offset to achieve an organic change in direction. The *random offset* of the *target* is achieved by slight random increments, also the *perlin noise* function can be used to determine the increments. The boid then calculates its steering toward the *target_point*.

The *desired_velocity* is calculated as follows:

$$\mathbf{vel}_{\text{desired}} = \text{set\_magnitude}(\mathbf{point}_{\text{target}} - \mathbf{pos}_{\text{current}}, vel_{\text{max}})$$

The *desired_velocity* is then used to calculate the steering force as discussed in Section 5.2.1.

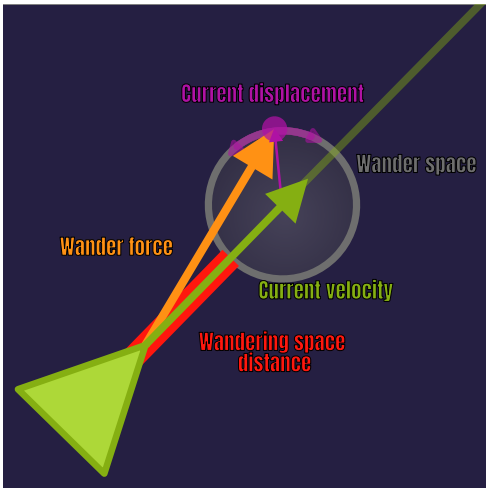


Figure 5.5: Wander behavior visualization.

This approach offers improved control over the behavior by adjusting these parameters.

- The distance between the boid and the sphere influences how smooth and forward oriented the movement feels. The larger the distance between the sphere and the boid, the more forward-oriented and smooth the movement feels.
- The radius of the sphere will affect how sharp the turns are.
- The shape of the wandering is usually a sphere, but it is also possible to create a steering bias by stretching the sphere into capsule.

5.2.5 Follow path

In the previous Section 4.2.2, we discussed the importance of finding a path between two points. The path is commonly represented as a list of points or nodes. Such representation allows us to interpolate the "shape" of the path between the points. The easiest way is to linearly interpolate between the points, creating a path where each point is connected to the next with a straight line. There are different ways to interpolate the path, for example, using a Catmull-Rom spline or a different type of curve.

Having an agent follow such path is a different issue often refereed to as animating the gent along a path. The *follow path* steering behavior (see Figure 5.6) presented in the work of Reynolds [REY+99] offers a solution. The behavior follows these steps:

Algorithm 3 Path Following with Corrective Steering

Define *pathRadius*
Calculate *futurePosition* = *currentPosition* + *currentVelocity* * *lookahead*
Project the *futurePosition* onto the path to get *projectedPoint*
Compute *distance* between *futurePosition* and *projectedPoint*
if *distance* > *pathRadius* **then**
    Compute a *targetPoint*:
    targetPoint = *projectedPoint* + *currentVelocity* * *lookahead*
    Apply corrective steering:
    desiredVelocity = *targetPoint* - *currentPosition*
else
    No steering needed; agent is within acceptable path bounds
end if

The *path_radius* controls how closely the agent follows the path. The desired velocity calculated with Algorithm 3 is then used to apply the steering force as discussed previously in Section 5.2.1.

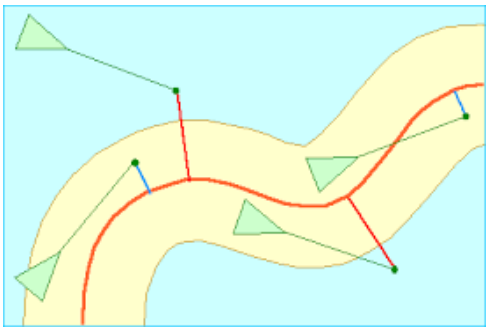


Figure 5.6: Visual representation of the path follow steering behavior by Reynolds (source: [REY+99]. The red curve represents the path, the yellow marks the *pathRadius*, the dots in front of the boids represent the *targetPoint*

5.3 Flocking

In nature, organisms have a tendency to move in groups as discussed previously in Section 2.1. Each member of the flock moves in its own way, but when observing the flock as a whole a certain pattern emerges.

For us, the goal is to make all the agents act as members of a *flock*. We will use a popular flocking algorithm described in the Boids program and discussed in the work of Reynolds Flocks, herds and schools [REY87].

Reynolds's flocking behavior is the result of three rules, e.g., steering vectors:

- *Separation:* Calculate the sum of vectors pointing from each flockmate toward the boid. Basically flee from each flockmate (see Figure 5.7).

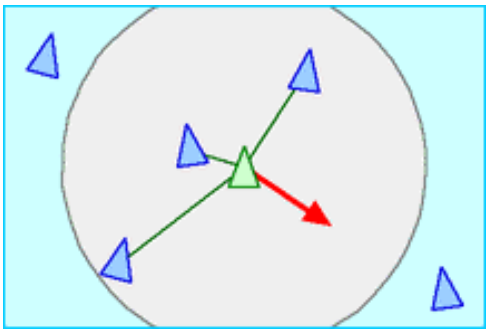


Figure 5.7: Steer to avoid crowding of the flockmates (source: [REY25]) The boids inside the gray area are the flockmates..

- *Cohesion:* Calculate the average position of the flockmates and use the seek behavior to move toward this position (see Figure 5.8).

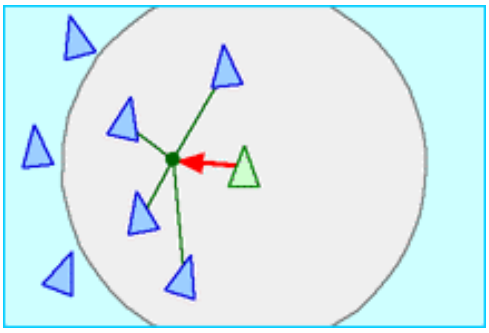


Figure 5.8: Steer toward the average position of the flockmates (source: [REY25]).

- *Alignment:* Calculate the average heading of the flockmates and steer toward the average (see Figure 5.9).

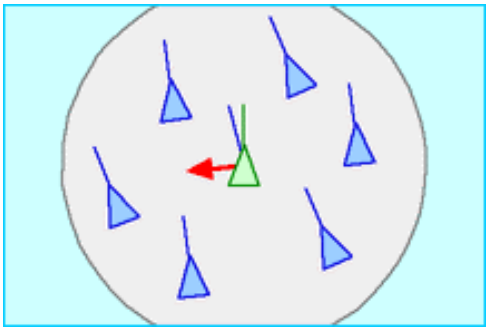


Figure 5.9: Steer to align the heading to the average heading of the flockmates (source: [REY25]).

Each rule results in a desired velocity. Their sum can be used as final *desired_velocity* to determine the steering force as mentioned previously in Section 5.2.1. Calculating and applying the steering force to each individual boid results in a flocking behavior.

To gain a finer control of the flocking behavior a weighted sum of the desired velocities can be used to determine the final *desired_velocity*.

- Increasing the weight of the separation rule will result in boids keeping a larger distance between each other.
- Increasing the weight of the cohesion rule will result in boids sticking more toward the center of the flock.
- Increasing the weight of the alignment rule will result in boids quickly changing their heading to move in the same general direction.
- Decreasing the weights will have an opposite effect.

All rules are computed in a neighborhood of the individual boid. A neighborhood is typically a group of boids in a certain radius around the individual boid. The radius is an important parameter of the behavior and represents the limited vision of each boid. A large radius results in larger flocks and vice versa.

Since the algorithm needs all the boids to adjust their velocities according to the velocity and position of every other boid, the algorithm runs at $O(n^2)$. This can cause performance issues with a growing number of boids. We can fight this issue using space partitioning structures such as Voronoi Neighborhoods, K-D Trees, and the Cube Partitioning Map mentioned in the work of Ježowicz [JEZ23].

Chapter 6

Implementation

In this chapter we will discuss the implementation of the individual components needed to bring the game concept from Chapter 3 to life.

6.1 Software and tools used

I used the Unity game engine since it is a great, versatile engine for mobile game development. It includes a complete 2D physics engine, crucial for tasks like ray casting, a built-in profiler to identify performance bottlenecks, and robust cross-platform support, allowing builds for mobile, Windows or WebGL.

I used Procreate and Krita for raster graphics and Affinity Designer for vector graphics.

6.2 Player

The player has two types of movements: on land, in water. Movement in water is handled with the use of the same principles of steering behavior. Where the desired velocity is the current device input of the user. Usage of the steering force grants slow turns and natural acceleration by tweaking the *maxSteeringFroce* parameter.

6.2.1 The tools

The player is equipped with a fishing net. The net has a certain size, which naturally sets a maximum size of the fish the player can catch. The net size is an important stat that the player can upgrade during gameplay. The fishing net is used by pressing the action button. The net moves in an arc above the player's head.

To make the act of catching fish feel more natural and challenging, multiple points were placed along the shape of each fish as in thee Figure 6.1. Each point acted as the center of a circle with a defined radius. When the net entered the fish's main collider, it checked whether all of these circles were overlapped during the collision. If so, the catch was considered successful.

This approach worked better than simply checking whether the net touched the fish, as it required the player to properly align the net and move it along the entire fish in a sweeping motion. The simpler method felt unconvincing, since just barely touching the fish was enough to catch it.

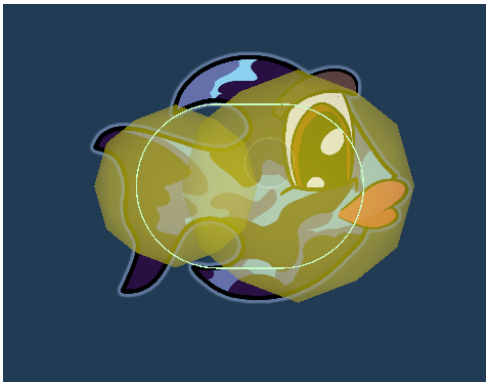


Figure 6.1: The fish collider visualized in Unity. The yellow circles represent the collider used for catching the fish. Each circle needs to be overlapped by the net during a single sweep to catch the fish. The green capsule is the collider used for Unity collision detection.

The player is also equipped with a spear. The spear can help the player to weaken the predators or kill them and put them into the backpack. The spear has a cool-down to prevent players from spamming the attack. The player can switch between the net and the spear at will as show in Figure 6.2.

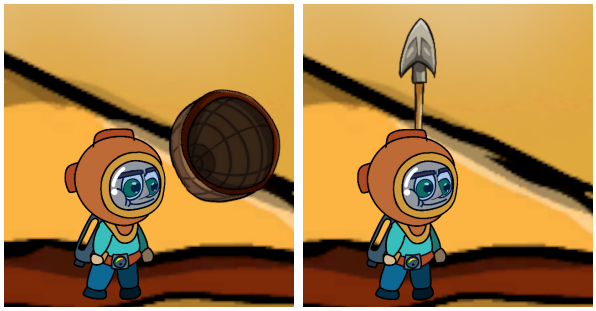


Figure 6.2: Net and the spear on the player. The spear is shown during movement.

The fisherman's abilities are constrained by multiple stats, such as maximum speed, swimming agility, maximum health, maximum oxygen capacity, and pressure resistance. All the stats can be upgraded in the shop.

6.3 Modeling group behaviors for video games

In this chapter we will discuss all the individual components required to simulate behaviors of grouping agents in a procedurally generated world. The goal of the project is to create a game that simulates the behaviors of fish as

mentioned in Section 3.3. The behaviors will be discussed regarding fish in a 2D however, these concepts can be transferred to 3D or completely different scenarios and contexts.

There are three types of behaviors each fish expresses:

- Interaction with the environment
- Interaction with the player
- Interaction with other fish

These behaviors are achieved by combining multiple steering behaviors 5, each designed to satisfy a certain need or a goal of the fish. We will also later discuss how to combine the individual behaviors.

6.3.1 Herbivores and carnivores

The fish species were split into two categories:

- *Herbivores:* Neutral fish type that search for plants as the food source, are scared of the player and will flee when the player gets close. This type has a tendency to school with other fish that share the same goal.
- *Carnivores:* Hostile fish type that hunts smaller fish for food or even the player. This fish type will not be aggressive unless it is hungry or attacked. They prefer to hunt alone and do not school.

6.3.2 Environment interaction

In this section, we will focus on all behaviors related to the interaction of fish with the game world.

Avoiding collisions

Each fish needs to be able to avoid unnecessary collisions with the environment. Using the Obstacle avoid behavior 5.2.3 is a great start. However, the basic behavior can have some limitations when the agent gets stuck near the corner of an obstacle similarly as in Figure 6.3. When the center of the obstacle avoidance rays is near a sharp corner of an obstacle. The obstacle avoid behavior interprets all the rays as blocked and forces the fish to spin endlessly, since the fish will turn to a side without moving away from the obstacle fast enough.

This can be solved by detecting all obstacles inside a circle around the fish and using the flee behavior with the closest intersection as a *flee target*. The circular check forces the fish to keep distance between the fish and the obstacles.

The circular check can cause issues in narrow corridors by forcing the fish to repeatedly move away from one edge of the corridor to the other. It is important to make the weight of this circular check low to gently nudge the fish in the correct direction without creating a too big of a disturbance.

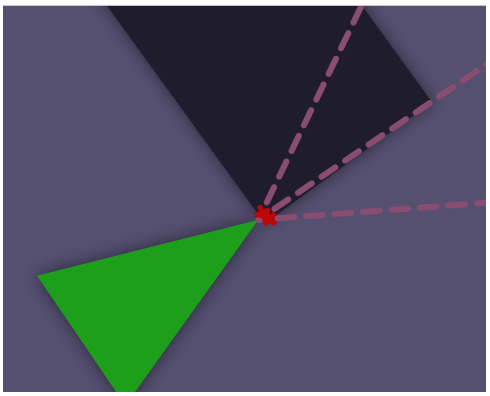


Figure 6.3: The diagram showing the problematic case for the obstacle avoid behavior when approaching sharp corners. Since all the rays start inside the collider.

Moreover, the basic obstacle avoidance behavior looks slightly unnatural because the fish steer away from the obstacle just enough to stop detecting the collision, which causes the fish to swim along the walls.

Exploration

As previously stated, the fish will follow along the walls forever until they reach another obstacle, then just steer away slightly and keep moving ahead. It is important to give the fish a reason to not move in a straight line forever. Using the wander behavior is a great start. The wander behavior will make the fish move randomly around the world.

Giving fish long term goals is a great approach to make the fish move as if they have a purpose. It is possible to randomly pick spots near the fish and make the fish move toward that point. Once they reach the point a new one is picked. These points should be easily reachable for the fish and picking unreachable points should be avoided entirely. This is was implemented in the early prototypes of the game but it was replaced with the food hunting discussed next.

Food hunting

Simulating a fish's need to search for food is crucial. The fish can be *full*, *satisfied*, or *hungry*. Once the fish is *full* they should not even consider moving toward a food source. Once the fish is satisfied, it will seek a food source when it is close enough or passes directly by it. When the fish is hungry, its top priority is to find a food source, otherwise it will die. In case of hunger, the fish should be able to locate the food source directly, using the pathfinding algorithm.

Following the path is solved by using the path follow algorithm, discussed in Section 5.2.5, to move toward the food. A different method of path following could be used to follow the path, such as the one discussed in the work of

Silveria et al. [SIL+10] using harmonic functions. Otherwise, when the fish is not pathfinding for food it just uses the *seek behavior* 5.2.1 and the nearest food source as the *seek target*.

The navmesh (see Figure 6.4) used for the pathfinding algorithm was generated using the algorithm discussed previously in Section 4.2.1.

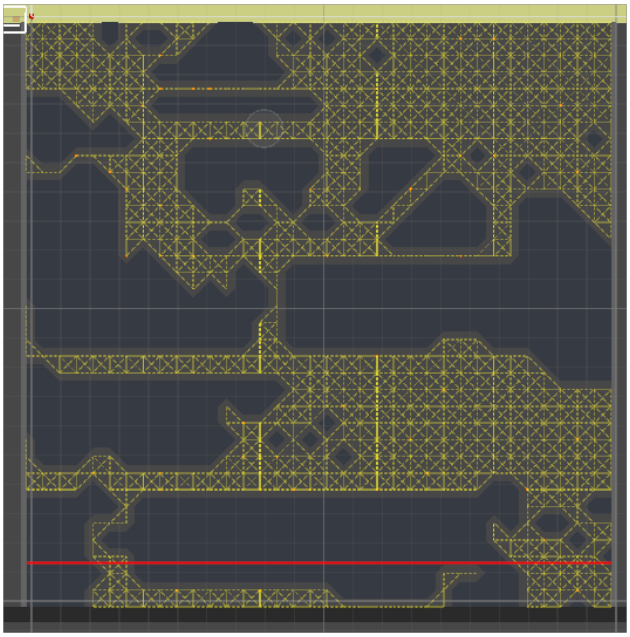


Figure 6.4: The generated navmesh of the world visualized using Unity gizmos.

6.3.3 Interaction with the player

In this section, behaviors of fish when the player gets close will be discussed.

Fleeing from player

Herbivores, *injured fish* or *smaller fish* will always *flee* from the player when it gets close. Fleeing from the player is a high priority and the fish will not be able to eat from the food source at peace.

Attacking the player

Aggressive fish when *threatened* or *hungry* will attack the player. However, when the fish feels like losing the battle it will try to survive and possibly *flee* from the player. Bitting the player or a fish will give the fish the necessary nutrition. Individual bites of the fish have a cool-down. During the cool-down the fish will try to gain some distance from the player and get back close when it is ready to bite the player again. When trying to catch the player, the fish will mix the *pursuit* and the *seek* behavior 5.2.1 depending on its specie to catch up with them.

6.3.4 Interaction between the fish

In this section we will discuss how the fish should behave when they encounter another fish. These behaviors will differ depending on whether the fish is a herbivore or a carnivore.

Flocking

The flocking algorithm, discussed in Section 5.3 by Reynolds [REY87], is the base of the interaction between the fish. Schooling with other fish is a natural tendency for all the herbivore fish. This allows them to move together as one large entity. It does not make sense for fish to school with others if their current *purpose* does not match. This can be handled by each fish *broadcasting* their *purpose* to the other *flockmates*. Fish will only school with others if their purposes match. For example, *fleeing* fish will school with other *fleeing* fish, fish that are *food hunting* will school with other *food hunting* fish. Fish that do not have any current purpose will typically just wander around and school with all other fish.

The fish decides its purpose by analyzing its current situation, e.g., when it is hungry it is seeking for food. However when a predator is close its purpose is to flee even when it is hungry.

Predator-prey relation

Fish will always flee from predators. This creates nice effect when a fish school is gathered around a food source and suddenly a predator approaches. The school can either flee as a whole or split into smaller groups and flee individually. This is discussed in later in Section 7.2.2 and compared to a video of real-life fish.

Hierarchy

There is a *hierarchy* in the fish schools. When a fish does not have any other leader in its neighborhood it becomes the leader. This is specifically useful when seeking for food. Only the leader fish will be allowed to run the pathfinding algorithm to the nearest food. Other, follower fish, will just school, i.e., basically copy the leaders' movement. *Follower* fish will typically just run the flocking algorithm and a wander behavior. This allows for a more structured movement of the school and allows for performance improvement by running the pathfinding once per school and not for all the individual fish. The leader fish do not use the flocking algorithm since their movement is usually along a path to food source.

In nature, the system of leaders is similar. They use a rule of *any-one leader* [LAN98] typically when fleeing from predator, one fish decides the direction of the whole school.

6.3.5 Combining strategies

In this section, we will discuss solutions to the execution of multiple steering behaviors. It is common for a fish to react to multiple things at once, e.g., avoiding an obstacle while trying to dodge the player. Designing a method that supports growing number of behaviors, while maintaining a "natural" pyramid of the needs is challenging. Some possible solutions will be offered below.

State Machine

An easy way control the execution of behaviors is to use a *state machine*. The first step is to define *states*, each state represents a reaction to a situation. Each state is represented by a *steering behavior*. Each state returns a steering force. An example of the state machine (see Figure 6.5) can look like this:

- *Idle state:* Use wander steering behavior mentioned in Section 5.2.4.
- *Flee from player state:* Use a flee steering behavior 5.2.2. with the player as the flee target.
- *Avoid walls state:* Use the obstacle avoid steering behavior 5.2.3 or flee from the nearest wall.

The transition between states is handled by simple conditions:

- *Idle to flee from player:* Switch if distance between the player and the fish is smaller than an arbitrary threshold.
- *Any state to avoid walls state:* Switch if distance between a player and some obstacle ahead is smaller than an arbitrary threshold.
- *To idle state:* Switch if no other transition is satisfied.

A state machine provides a precise and deterministic way to manage behavior of an agent. Transitions between these states are triggered by specific conditions (represented by arrows in Figure 6.5). This pattern offers full and simple control over the behaviors, but abrupt transitions between states can occur. Improving the smoothness transition is discussed below (see 6.3.5).

A state can also be represented by a list of steering behaviors each with its own weight. The sum of the forces can be used to do the final steering - similar approach will be discussed in the next Section 6.3.5.

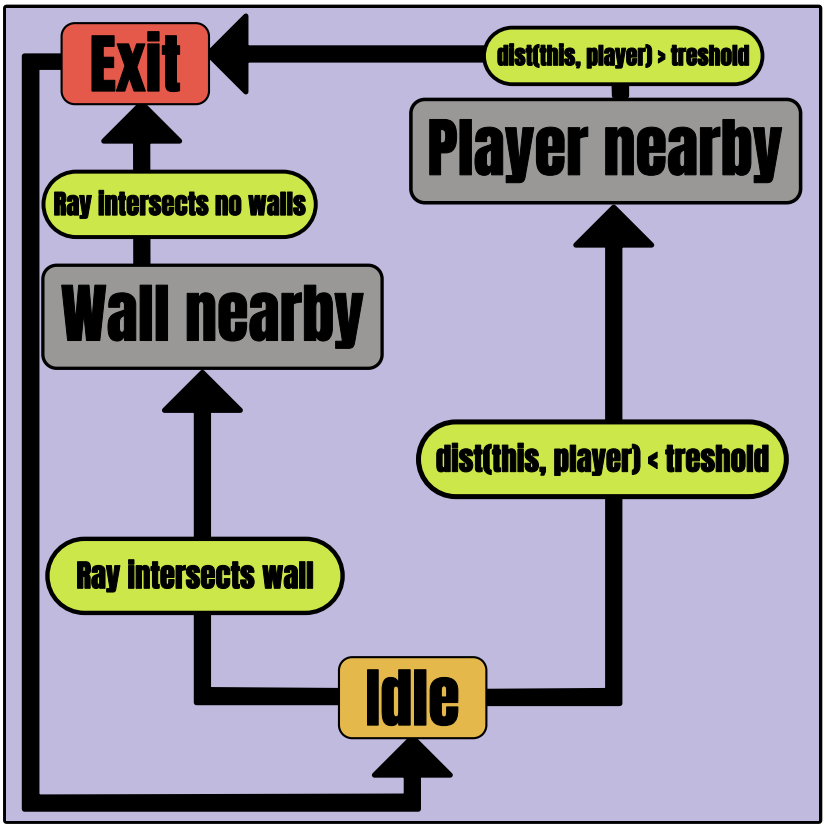


Figure 6.5: Example state machine visualized as a graph. The capsules represents the transition conditions and the rectangles represent the individual states.

Using the state machine alone can have potential issues when transitioning between states. Typically, when the Idle state is active and an obstacle is approached the avoid walls state is triggered. The obstacle is successfully avoided and back to the idle state is transitioned. However, the idle state can often wander back to the previously avoided obstacle, the avoid walls state is triggered again, the obstacle is dodged, the idle state is triggered and into obstacle is wandered again. This can cause a loop of switching between the individual states and can cause a jittering movement of the fish. This looks unnatural and can happen quite often.

The issue can be fixed by putting in a delay between the transitions of the states (i.e., exit times) or by blending the *current desired velocity* with the *desired velocity* from the previous step.

Utility based weighting

This approach is different from the state machine. Rather than setting exact conditions under which a behavior should run, all behaviors can run at the same time and blend between them according to the current utility of the behavior.

Each behavior requires a utility function. The utility function is a function with range [0,1] the higher the utility the more useful is the function at the

moment. The calculation of the final steering force weighted by utilities is shown in Algorithm 4.

Algorithm 4 Utility-weighted steering force calculation

```
Define list behaviors
Initialize array utilities of length behaviors.count
Initialize array desired_velocities of length behaviors.count
total_utility = 0
for i = 0 to behaviors.count do
    utilities[i] = behaviors[i].getUtility()
    desired_velocities[i] = behaviors[i].getDesiredVelocity()
    total_utility += utilities[i]
end for
desired_velocity = (0, 0)
if total_utility > 0 then                    ▷ Normalization of the utilities
    for i = 0 to behaviors.count do
        normalized_utility = utilities[i] / total_utility
        desired_velocity += normalized_utility * desired_velocities[i]
    end for
else
    desired_velocity = fallback direction (e.g., agent's forward)
end if
desired_velocity = set_magnitude(desired_velocity, max_speed)
steering_force = truncate(desired_velocity - current_velocity, max_force)
```

The normalization of the utilities keeps all the behaviors contribution as expected. Each *behavior_desired_velocity* is the normalized *desired_velocity* calculated by the steering behavior, in a sense it is the *desired direction* of the individual steering behaviors.

This approach is the best for the case of our game. It allows the fish to be part of school while still slightly jittering their movement with the wander behavior and they will always avoid obstacles when approaching one. The disadvantage of this approach is the need to define the utility functions. It is a great practice for each utility function to be normalized in the range [0,1].

Utility functions for behaviors like seek, flee are easy to set up. The utility should rely on the distance from the target as in Algorithm 5. However, different behaviors require often much more difficult calculation and logic.

Algorithm 5 Flee behavior utility

```
d = distance(position, flee_target_position)
clamped_distance = Clamp(d, 0, danger_distance)
return 1 - (clamped_distance / danger_distance)
```

The *danger_distance* is the maximum distance at which the danger from the flee target is relevant. If the distance is larger than the *danger_distance* the behavior is not relevant to the fish.

The utility function (shown in Algorithm 5) works great. However, it is beneficial to add a threshold for the distance, so that once the distance is smaller than a threshold value, the utility of the behavior is equal to 1. Using this approach, the fish can try to flee from the player as the top priority and react faster.

6.3.6 Deciding the speed of movement

The calculation of the steering force from Section 5.2.1 relies on setting the magnitude of the desired velocity to the *max_speed.* However, it would be nice to allow the fish to change their *max_speed* according to the situation. For example, when the fish is wandering around it should move at a leisure pace, but once a predator approaches and tries to attack the fish, it should try to move away with all its energy and power.

An approach discussed in the work of Popelová [POP11] added an option for each behavior to ask for a speed-up or a slow-down. If all behaviors want to speed up or slow down collectively, the fish changes its speed. The approach used in the game is similar to this one. However, rather than simply deciding whether to speed up or slow down, each behavior contributes a vote for a desired speed, along with an associated utility value calculated by the steering behavior. The final speed is then calculated as a weighted average of all suggested speeds, where the weights are given by the utility values. An identical approach is used to allow the fish to momentarily increase their maximum steering force.

6.3.7 Implementations of the fish object

In this section, the implementation of the fish object will be discussed. Each fish uses some of the classes explained below.

Fish Movement base class

A fundamental component of the implementation is the *Fish Movement abstract base class.* This class is responsible for calculation of the *desired_velocity* for all the types of steering behaviors,

By abstracting this functionality, the base class simplifies the overall structure and ensures consistency across the different movement behaviors.

```
public abstract class FishMovement : MonoBehaviour
{
    protected Vector2 desired;

    public abstract Vector2 MoveStep();

    public abstract void Mutate(Fish fish);

    public abstract float GetUtility();

}
```

Figure 6.6: Declaration of the base class for all fish movement types. Each specific movement behavior inherits from this class. Implementations may range from simple steering behaviors to more complex logic such as food-seeking, which combines pathfinding with seek steering behavior.

- *MoveStep:* Returns the *desired direction* of the behavior, that is, the normalized desired velocity.
- *Mutate:* Handles mutation of behavior according to fish specie, such as size and speed, or steering behavior parameters.
- *GetUtility:* Calculates the utility of the behavior as discussed in Section 6.3.5.

Steering behavior manager

The steering behavior manager class serves as the core component for integrating multiple steering behaviors. It calculates the final steering force by combining the desired velocities from individual behaviors and then applies the resulting force to the fish. The class handles all the *FishMovement* (see 6.3.7) scripts their *MoveStep* and uses the utility based weighting as discussed in Section 6.3.5.

When a utility of the behavior is zero the desired velocity is not event calculated, this can save us from some extra calculations.

This component can be also reworked to use the state machine approach discussed earlier in Section 6.3.5.

School of gobies

A school of gobies is an entity that uses all the group behavior components while maintaining a large group of miniature fish or gobies as shown in Figure 6.7.

Each goby is a single sprite and its position is managed by a *school manager* script which calculates the flocking rules, discussed in Section 5.3, and handles its steering. When a goby moves outside of the radius of the school it uses seek behavior to get closer to the middle of the school.

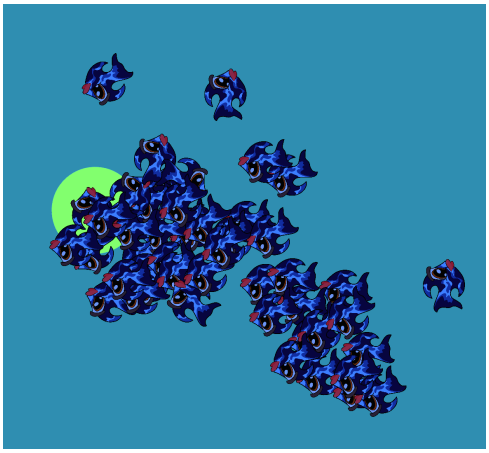


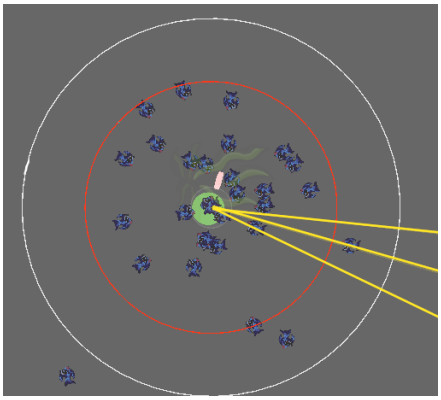
Figure 6.7: A school of ∼45 gobies in the game. The green circle marks the center of the school. All gobies use the flocking rules and created 2 smaller distinct flocks. Some gobies are away from the flock but will move back once they reach the outside border of the school.

The catching of the individual gobies is handled using a *CapsuleCollider2D* which is scaled to fit most of the gobies. When the net enters the collider all the gobies overlapped by the net are removed and put into the fisherman's bag.

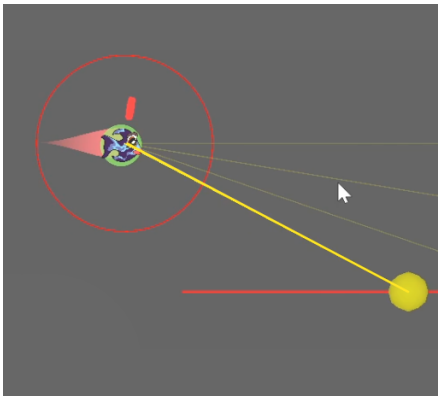
The addition of the gobies improved the overall feel from the game. However, there were performance issue with the naive implementation as discussed in Section 7.1.

6.3.8 Debug view

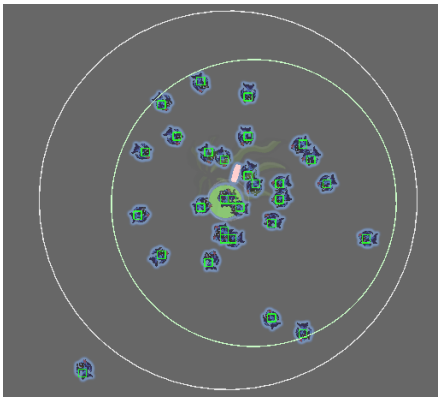
An important tool during development was a custom debug view, which used Unity's Gizmos and Handles [UNI24]. The debug view was especially useful when certain behaviors did not seem to work as expected, since it provided a visual insight into what the fish perceives. Examples of these debug views are shown in Figure 6.8.



(a) Obstacle avoidance visualization. The red circle checks for nearby collisions and the yellow rays detect upcoming obstacles. The white circle marks the radius of the school, if gobies were to move behind the border, they would seek back towards center.



(b) Food-seeking behavior using a follow-path method. The red line represents the path to follow, yellow circle visualizes the projected position onto the path. In the figure also the obstacle avoidance is visualized.



(c) Space partitioning visualization in the school of gobies. The green circle shows the actual capsule collider. The green rectangles show the used the cells of the space partition grid. Yellow circle represents the shool radius.

Figure 6.8: Steering behaviors visualized in the Unity editor.

6.4 Terrain generation discussion

In this section we will discuss, how the underwater terrain can be generated, and how the terrain is generated in the game. The terrain generation method needs to satisfy these features:

- *Travers-ability of the terrrain:* The player needs to be able to reach all important locations.

- *Tile-ability:* The world should be composed of individual tiles to allow unloading a loading the tiles only when needed.
- *Randomness with control:* We need to keep the terrain different in all runs while still having a certain control over the overall look and difficulty.

All the discussed algorithms, when altered slightly, can be used to generate a 3D terrain.

6.4.1 Randomized obstacle placement

The simplest approach to quickly generate the terrain is to use pre-made obstacles and put them randomly on a grid. The terrain is generated layer by layer, where each layer is composed of blocks arranged in a row and each block contains a grid of points. The grid can have any dimensions but using a square grid is the simplest approach.

To ensure that each generated level remains traversable, it is important that the radius of each obstacle follows the condition 6.1:

$$r_{\text{obstacle}} < d - \frac{r_{\text{player}}}{2} \tag{6.1}$$

Where:

- r_{obstacle} is the radius of the smallest circle that fully encloses the obstacle.
- r_{player} is the radius of the player's bounding circle.
- d is the spacing between grid points where obstacles can be placed.

Satisfying the condition grants the player the ability to always squeeze between the obstacles, ensuring that every level is traversable.

When generating the obstacles, the algorithm picks a few spots at random and spawns a random pre-made obstacle at the place. Similarly, as in Section 6.4.2, 2D perlin noise function can be sampled and only when the sampled value reaches a certain threshold the point spawns a random obstacle. This can incline toward creation of continuous obstacles and larger obstacle-free areas.

6.4.2 Marching Squares

Marching squares algorithm requires a grid of points marked as either terrain or air for input (Boolean value). The algorithm than uses a predefined set of rules to connect points of terrain into triangles as shown in Figure 6.9. It is important to notice that the algorithm connects only the *mid-points* of each edge.

The real issue is to figure out which points should be marked as air or terrain. Often, 2D perlin noise function is sampled and points with a value above some arbitrary threshold are marked as air and vice versa. This approach grants organic-looking cell-like mesh as seen in Figure 6.10a.

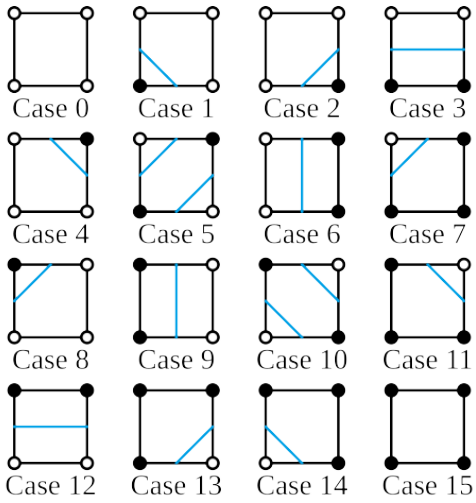
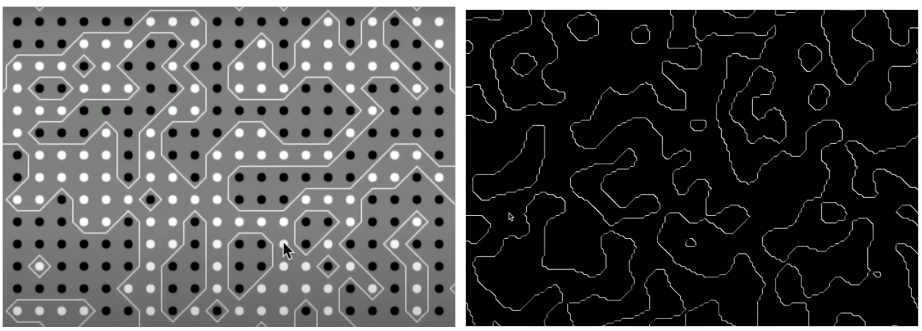


Figure 6.9: Rules of triangulation for the marching squares (source: [CON24]). The black circles represent solid terrain, the white circles represent air. The blue line connects mid-points and creates the final mesh.

The algorithm can be tweaked to create more rounded edges. Instead of using the binary approach and deciding whether a sampled perlin noise value is above certain threshold, we can use the sampled value directly. The sampled value is then used to interpolate the *mid-point* of each edge toward a vertex with the higher value resulting in slightly more curved contours as in Figure 6.10b.



(a) Example of a mesh generated by the marching squares. The black dots represent the terrain and the white dots the air (source [TRA20] 17:31).

(b) Example of a mesh created by the marching squares with interpolation and the perlin noise function setting the value for individual points (source: [TRA20] 23:40).

Figure 6.10: Examples of marching squares output.

The 3D variant of this algorithm is the Marching Cubes algorithm; see [CON25b] to better understand the differences.

6.4.3 Room based generation

This method operates with pre-made rooms with defined exits and entrances. The level is split into a grid of cells and a clear path through the level is constructed first. The remaining cells are then filled with random rooms that make the level feel more organic. Similar techniques are used in games such as Spellunky [TOO16] and Wizard of Legend.

The first step of the algorithm is to construct a clear path through the level. After the construction of the clear path, the rest of the level can be filled with random fitting rooms using a similar approach to that in Section 6.4.1.

To find a clear path, we can use one of the pathfinding algorithms mentioned in Section 4.2.2. However, the shortest path may not be the best approach. Creating the shortest possible path through the level can be too streamlined of an experience for the player and may prevent the desire to explore.

Using a random walker or depth-first-search (DFS) algorithm will create a more tangled path. The random walker approach will be discussed next. The DFS is the same algorithm as the BFS shown in Algorithm 2, with the only difference being the use of stack (LIFO queue) instead of the queue (FIFO).

The Random walker is a simple algorithm that can be imagined as drill making path through a cave without knowing which direction the end is. The random walker generates the clear path using Algorithm 6.

Algorithm 6 Random-walker-based path generation (Lévy flight)

```
1: Choose start position for the drill
2: while drill has not reached the end position do
3:     Select random direction
4:     Determine step size using Lévy flight
5:     Move drill in chosen direction by step size
6:     Mark all visited points as on-path
7: end while
```

The random walker usually takes steps of magnitude equal to one. However, introducing a varying step size, also called Levy's flight, is a great way to create distant clusters of rooms joined with long corridors (see Figure 6.11). By giving a small chance to the random walker to increase its step size by a large number while using a unit step the rest of the time, rooms connected with "corridors" can appear. The unit steps will create clusters and the rare occurrence of a flight will join the clusters with a corridor.

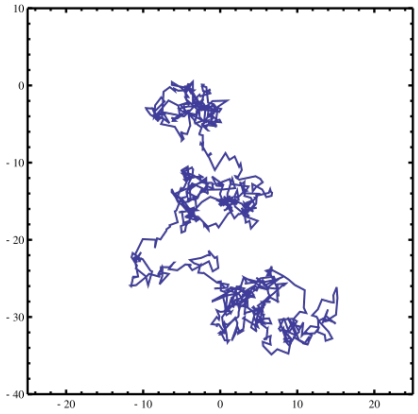


Figure 6.11: The path generated by the random walker. Clusters of rooms joined by corridors created by random walker with Levy's flight can be seen (source: [CON25a]).

The construction of the level is realized by instancing a room at each cell of the clear path. The room can be picked randomly but has to have an entrance or exit at each edge between the two cells of the path. After finishing the clear path, the rest of the grid is filled with random rooms until no entrances remain unused.

This method works great for creating intricate levels that contain complex and well-designed elements while maintaining variety in the play-throughs. Expanding the level generation is as simple as adding new room templates to the algorithm. It is crucial that each room has predefined exits and entrances, but whether the actual content of the room is generated at random during run-time or is completely hand-crafted is entirely up to the developer.

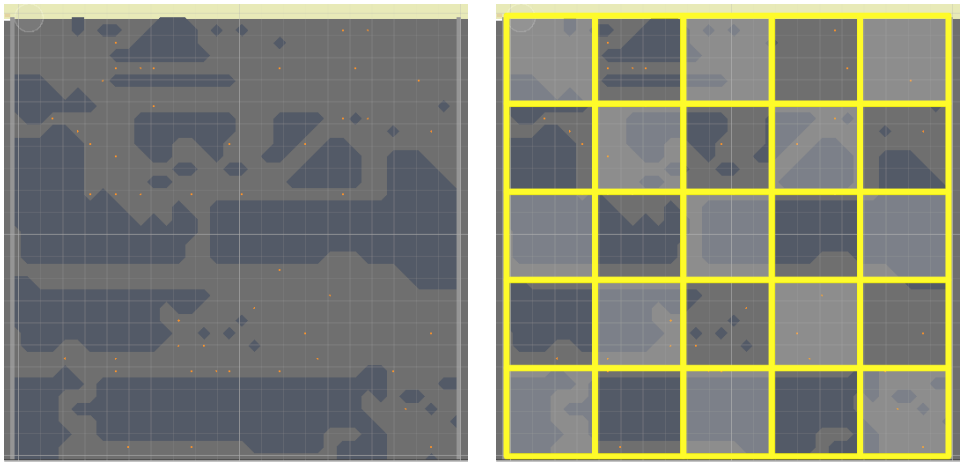
6.4.4 Chosen approach

For the game the marching squares algorithm, discussed previously in section 6.4.2, was implemented. To ensure a smooth path through the level, the random walker from Algorithm 6 was used. In Figure 6.12, the generated environment can be seen.

For the game it is important to have the world generated procedurally as individual tiles. The tiles can be unloaded when the player is far enough. In the game, the first layer of the ocean is generated as follows:

- Create a layer containing 5 blocks in a row.
- Start the random walker at the entrance to the level.
- Count how many times the walker touched the bottom row of each block.
- When the bottom row of each block was visited at least once - Stop the drill.

Tinkering with the ratio of how many blocks had to be visited or how many times the walker touched the bottom row allows for more versatile terrains.



(a) Generated terrain by the marching squares and the random walk in Unity.

(b) Generated terrain with the underlying blocks highlighted.

Figure 6.12: Screenshots of the generated terrain in game by the marching squares and the random walk.

Once the player approaches the end of the generated layer, a new layer is generated. The only difference is now the drill continues where it last touched the bottom row of any block.

When running the marching squares algorithm, the mesh was generated using the grid of each block and the corresponding *visited-by-drill* Booleans. *Visited-by-drill* points were interpreted as air. To ensure a continuity between the blocks, the *visited-by-drill* boolean values of the last column of each block were copied to the next neighboring block as the first column. The same goes for layers, the last row of each layer was copied to the next layer as the first row. This ensured continuity and the ability to keep all the tiles separated.

Each tile has a corresponding *2D Polygon collider* component. The contour of the mesh acted as the collider. The contour extraction was achieved using Algorithm 7.

Algorithm 7 The contour extraction algorithm

```
1: Initialize dictionary
   Add all boundary edges to the dictionary
2: for each triangle of mesh do
3:     for each edge of triangle do
4:         if reversed edge is in dictionary then
5:             Remove edge from dictionary
6:             continue next iteration
7:         end if
8:         if dictionary does not contain edge then
9:             Add edge to the dictionary
10:        end if
11:    end for
12: end for
13: Initialize list of loops
14: while dictionary is not empty do
15:    Pick first edge from the dictionary as start_edge
16:    current_edge = start_edge
17:    repeat
18:        Remove current_edge from dictionary
19:        Find next_edge in dictionary that connects to current_edge, the
   last vertex of current_edge equals to the first vertex of the next_edge
20:        current_edge = next_edge
21:    until current_edge is the start_edge
22: end while
23: Pass each loop as a path to the polygon collider
```

Spawning fish

The spawning of fish is determined by the specie of the fish. Each fish species has these attributes:

- *Commonness:* How likely is the fish to spawn when given the chance.
- *MinY and maxY:* Range of depths in which the fish has the ability to spawn.
- *Fish prefab:* The object associated with the specie. Typically, herbivore fish, piranha or goby school.
- *Stats:* An object containing all the attributes of the fish, such as speed, steering force, flocking radius or scale (see Figure 6.13).

The fish specie is implemented using the Unity Scriptable Object. The use of the scriptable objects allows for creation of many species with different attributes quickly.

When a new fish is needed to spawn. Each *air* point of the world block becomes a valid spawn point for fish. A single point is picked at random.

For the point a group of candidates is gathered by going through all the species and checking whether the point's Y position is within the $minY$ and $maxY$ o the specie. The final candidate is then picked randomly from the commonness-weighted list of the candidates and the associated fish prefab is instantiated.

Each specie has its own average stats, and upon spawning the fish generates an offset creating its own individual values. This allows each fish to be similar while sometimes creating complete outliers - such as huge fast fish that will not even try to flee from predator but will try its best to perfectly flee from the player. Upon applying the stats to the spawned fish, diameter of the fish measured and used as the mass in the steering force calculation 5.2.1. Keeping track of the fish's diameter allows the game to make sure that at least some fish will always fit into the player's net.

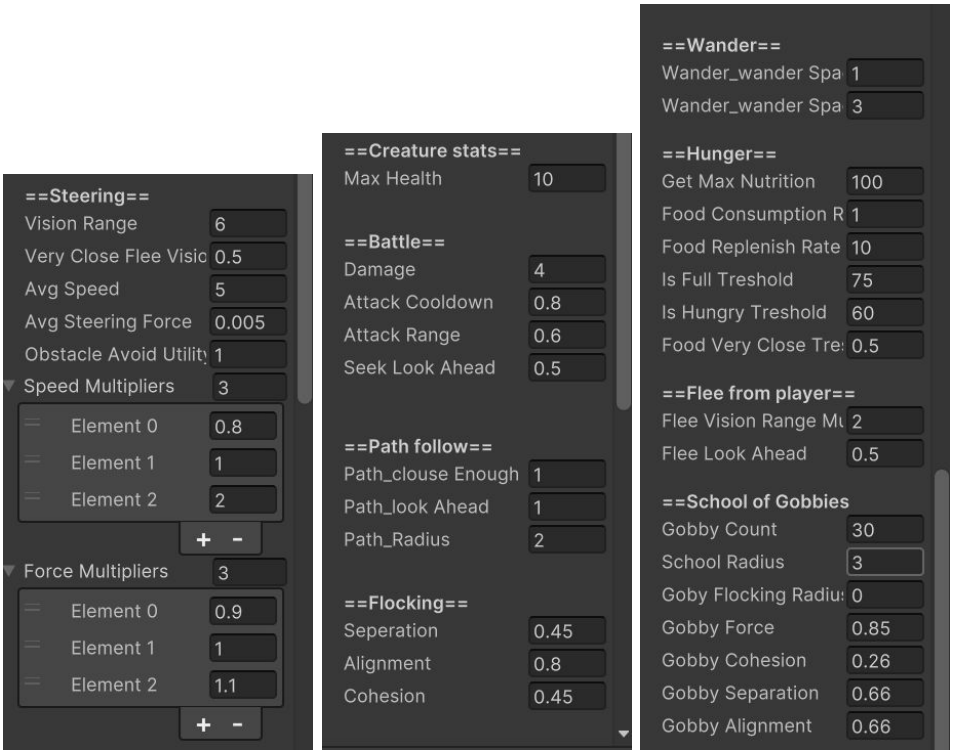


Figure 6.13: All the stas stored inside a fish specie. The fish create a copy of these stats and mutate them. This is viewed from inside the Unity inspector as Scriptable Object.

Chapter 7

Results

7.1 Performance

In this section, performance issues and bottlenecks of the game will be discussed. I have created a testing framework discussed below and used Unity Profiler to locate bottlenecks. Some optimizations were made and are compared with the unoptimized version.

7.1.1 Testing framework

To evaluate performance, frames per second (FPS) and frame time statistics were recorded under increasing fish counts. The following procedure was used for each test run:

- The game is built and launched.
- An empty scene is loaded to initialize the environment.
- A static seed is set for Unity's Random to ensure consistent results across tests.
- Spawner parameters are configured.
- The main game scene is loaded.
- Five world layers are generated.
- A fixed number of fish is spawned across the layers.
- The game is left running for 5 seconds to allow for warm-up and stabilization.
- Performance data is collected over the next 10 seconds, with 5 samples recorded per second.
- The game returns to the empty scene and prepares for the next test, where the number of fish is increased.

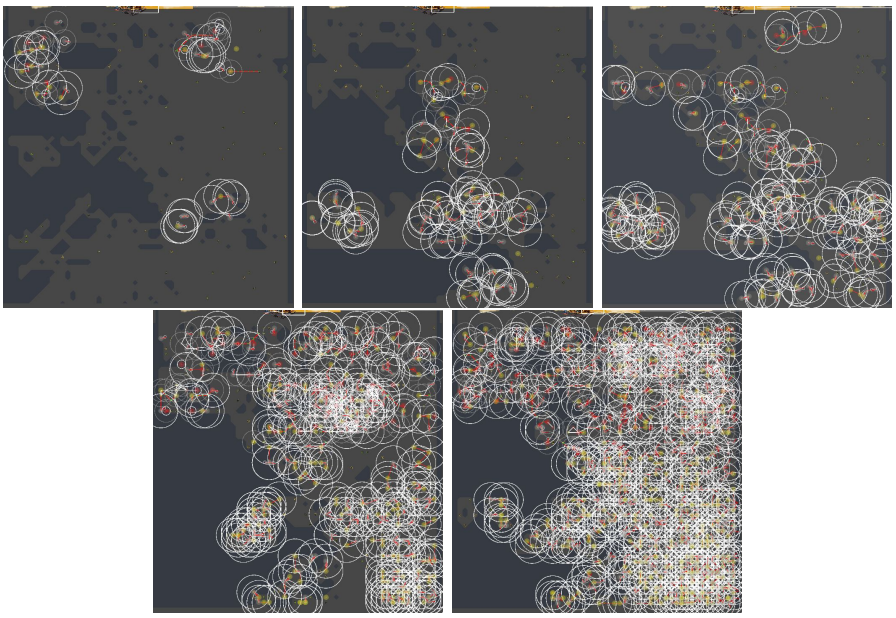


Figure 7.1: The state of the world right after the test starts. The circles display the flocking radius of a fish. The fish counts are 50, 100, 200, 500, 1000.

The tests were carried out with fish counts of 50, 100, 200, 500, and 1000. Each test recorded detailed statistics, including total frame time, time spent on flocking computations, and time consumed by steering behavior calculations.

The tests were run on my desktop computer, with specifications in Table 7.1, as a .exe build.

CPU	Intel Core i7-13700KF, 3.4 GHz
GPU	ASUS GeForce DUAL-RTX3060-O12G-V2, LHR, 12 GB GDDR6
RAM	32 GB DDR5 5200 MHz CL40
Operating System	Microsoft Windows 11 Pro (x64-based)
Unity Version	2022.3.10f1

Table 7.1: Hardware and Software Specifications of the computer that ran the tests.

7.1.2 The neighbor search

After inspecting the test results and profiler, it was found that the flocking algorithm, discussed in Section 5.3, significantly slows down the performance. The issue lies within the $O(n^2)$ complexity of the neighborhood search.

Improving the neighborhood search procedure is a common optimization technique when it comes to the flocking algorithm. The issue is commonly solved using a space partitioning map as discussed by Ježowicz [JEZ23].

A 2D grid partition map was implemented for the game. Each fish now

only calculates the distance between itself and the fish in the neighboring cells of the grid, significantly reducing the number of distance calculations. A same grid was used for each school of gobies itself to speed up the neighborhood search.

Version without the space partitioning grid uses *Physics2D.OverlapCircleAll* to get the flockmates of the fish. As seen in Figure 7.2 and Table 7.2, the space partitioning grid was not helpful when used with fully simulated fish objects.

Unity documentation does not state whether internal optimization techniques are used to query nearby objects. However, the result of the test may hint at the use of a space partitioning grid, quad-trees or other optimization techniques, since it is a common practice for game engines to use advanced and highly optimized algorithms to search for possible collisions.

Results summarized in Table 7.3 and Figure 7.3 show a significant improvement in the speed of the flocking algorithm calculation. Since the gobies do not use any Unity Physics 2D components such as colliders or rigidbodies, the neighborhood was originally gathered by calculating the distance of every fish with each other. The later use of the space partitioning grid lowered the calculation time significantly.

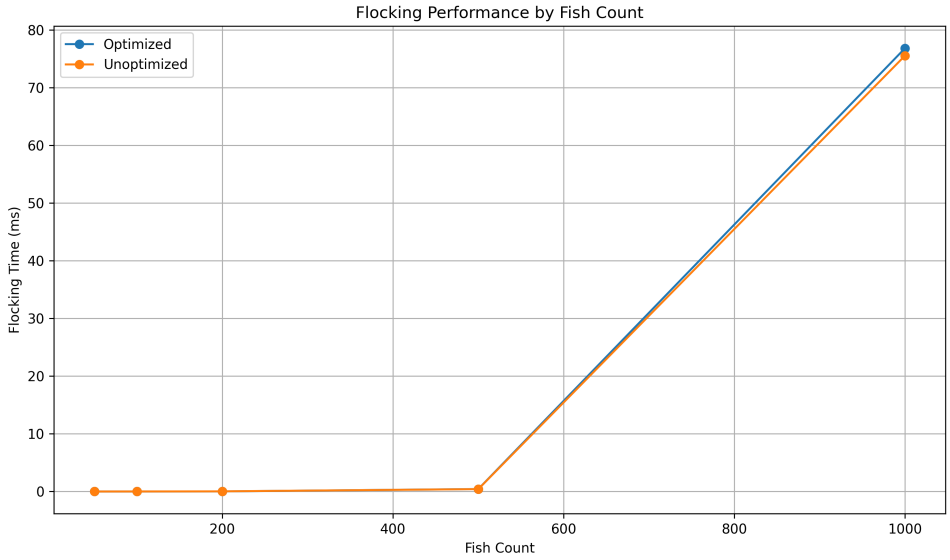


Figure 7.2: Comparison of the flocking algorithm computation time of the fish object. The optimized version uses a space partition grid and the unoptimized version uses *Physics2D.OverlapCircleAll* to gather the neighborhood. The first 4 values seem very low since the fish did not always have many neighbors as can be seen in Figure 7.1.

Fish counts	50	100	200	500	1 000
Unoptimized time (ms)	0.000	0.006	0.026	0.413	75.5
Optimized time (ms)	0.001	0.004	0.015	0.428	76.8

Table 7.2: Measured values during the tests, visualized in Figure 7.3.

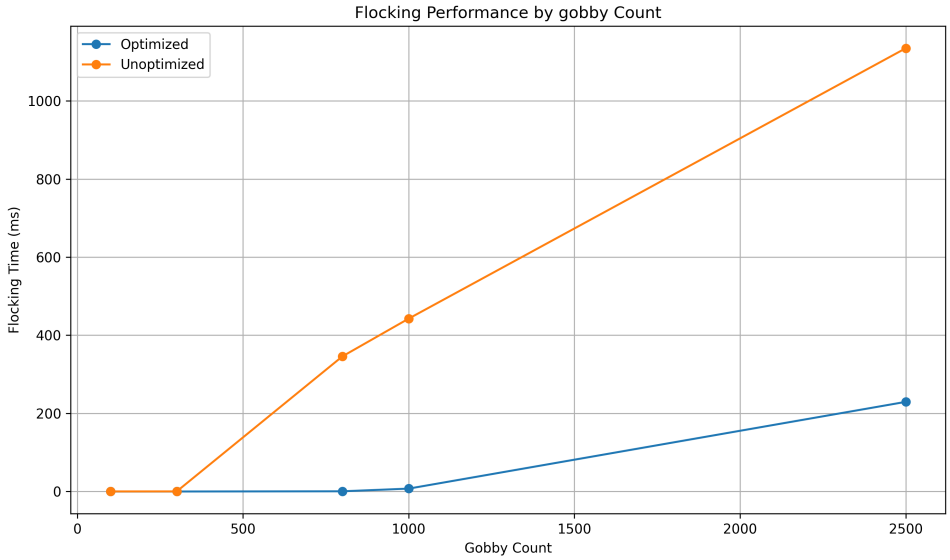


Figure 7.3: The graph shows a great difference in computation time between the goby calculation using the distance check (unoptimized) and the space partition grid (optimized). The goby count refers to the total number of simulated gobies, where each school consists of 100 gobies.

Gobby counts	100	300	800	1 000	2 500
Unoptimized time (ms)	0.229	0.305	346	443	1135
Optimized time (ms)	0.035	0.038	0.545	7.42	229

Table 7.3: Measured values during tests, visualized in Figure 7.3.

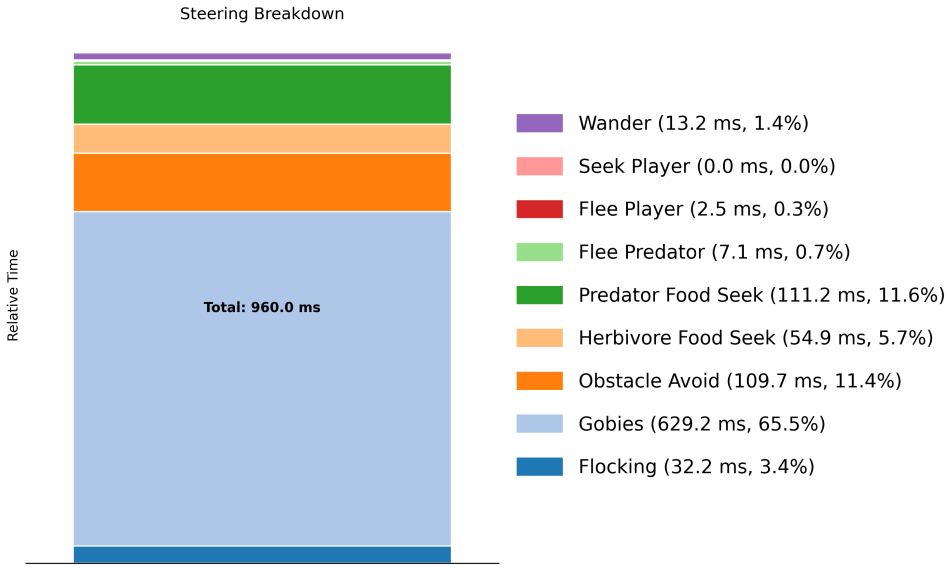
In the figures, we can see that the computation time of the flocking algorithm in the first few test scenarios is very low. This is caused by the size of the world across which the fish are spawned (as seen in Figure 7.1). They often may not even encounter any potential flockmates.

Deciding the size of the cells in the space partitioning grid is an important parameter. Using a 1–1.5 multiple of the flocking radius resulted in the best results. The same tests with different multiples of the flocking radius were performed and the averaged values were used in Figure 7.3.

7.1.3 Reducing the count fo gobies

In Figure 7.4 a complete breakdown of performance is shown. The graph shows a relative computation time of the individual steering behaviors. The

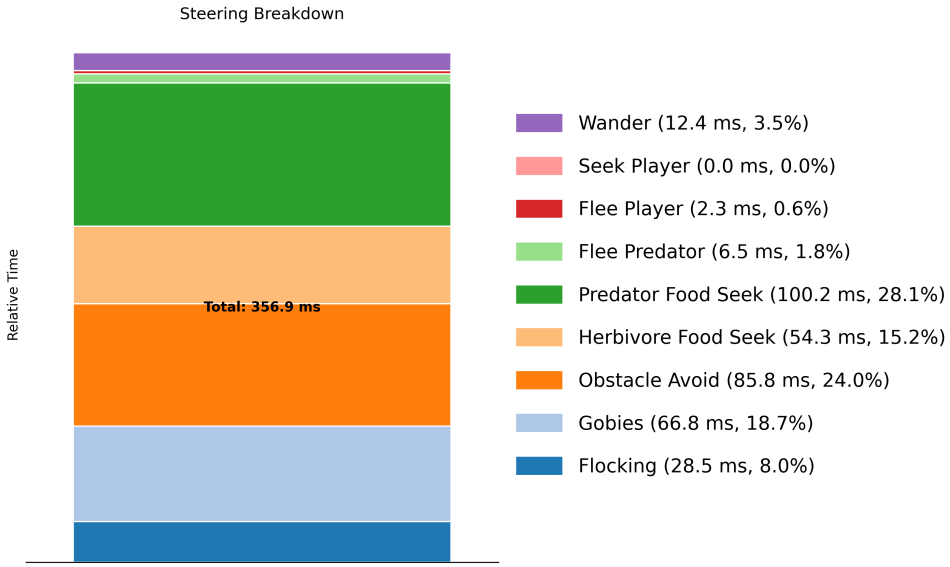
data used to construct the graph comes from the test case with the starting fish count being 1000.



Starting fish count	50	100	200	500	1 000
Fish alive	46	89	162	448	907
Gobies	700	500	1 000	4 100	7 200
School of Gobies	7	50	10	41	72
Frame (ms)	0.65	0.84	2.37	722.91	1 142.61
FPS	1 547.9	1 191.9	422.5	1.4	0.9

Figure 7.4: Visualization and performance table for original goby spawning setup with 100 gobies per school. The major part of the calculation is spent on flocking for gobies.

The major part of the calculation is the Gobies. Reducing the count of gobies to 15 per school significantly reduced the computation time while keeping the total number of gobies high. The Figure 7.5 shows the performance breakdown with the lowered goby count.



Starting fish count	50	100	200	500	1 000
Fish alive	48	87	177	445	894
Gobies	75	135	210	495	885
School of Gobies	5	9	14	33	59
Frame (ms)	0,55	0,68	1,22	7,57	494,46
FPS	1 824,8	1 462,0	818,3	132,0	2,0

Figure 7.5: Visualization and performance table when the goby count per school was reduced to 15.

7.1.4 The final performance of the game

To better fit the gameplay, the spawn rates of the fish were slightly tweaked. The biggest change was lowering the average count of gobies in school from 100 to 50 and increasing their scale for better clarity in the game.

Table 7.4 shows the final average fps during the tests after the tweaks.

Starting fish count	50	100	200	500	1 000
Average fish count	50	100	198	497	978
Goby count	116	417	1 182	1 985	4 477
Frame (ms)	0,56	0,77	2,36	414,54	1 039,73
FPS	1 776,2	1 300,4	423,7	2,4	0,9

Table 7.4: The average performance of the final game with tweaked spawn rates,goby counts and all in-game fish species - 25 gobies per school on average.

In the extreme case with 1000 start fish, we can see that each frame of the simulation takes approximately 1039 ms to compute. A significant portion of 830 ms is spent evaluating the various steering behaviors of the fish. Figure 7.6 illustrates the relative computation time of each individual steering behavior. Behaviors such as gobies, flocking or obstacle avoidance are more computationally expensive than others. This breakdown helps to

identify which parts of the overall behavior are the most demanding and may benefit from further optimization.

The very last optimization was chunk loading. Since the world generation is split into layers, only closest 3 layers to the player active at all times. When the player moves past a certain threshold the layers are either turned on or off and all the fish in the layers as well. A collider is then constructed around the loaded area to prevent fish from moving into the unloaded chunks.

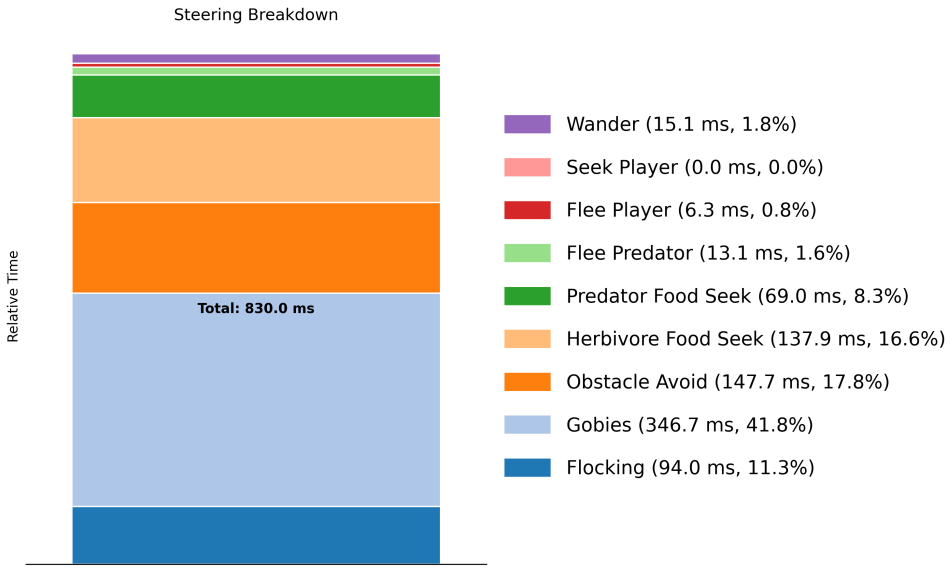


Figure 7.6: Distribution of computation time among individual steering behaviors. The chart highlights the relative cost of each behavior such as flocking, obstacle avoidance, and food seeking, measured over the duration of a frame. The absolute computation times are high, however thanks to chunk loading and the spawn rates, player will usually only need to simulate up to 150 fish during gameplay.

7.2 Current state of the game

The game currently offers two play modes: *Survival* and *Creative*. The Creative mode being functionally identical to the Survival mode, except that the player's stats are boosted significantly. This allows player to explore the depths without the need to worry about their survival (see Figure 7.7).

Figure 7.7: The game's main menu, allowing the player to choose a play mode or open the help window.

A contextual pop-up is displayed in the top right corner of the screen providing in-game tips and explanations (see Figure 7.9). Additionally, a help button opens a quick help window (see Figure 7.8) explaining all the game core mechanics with visual aid.

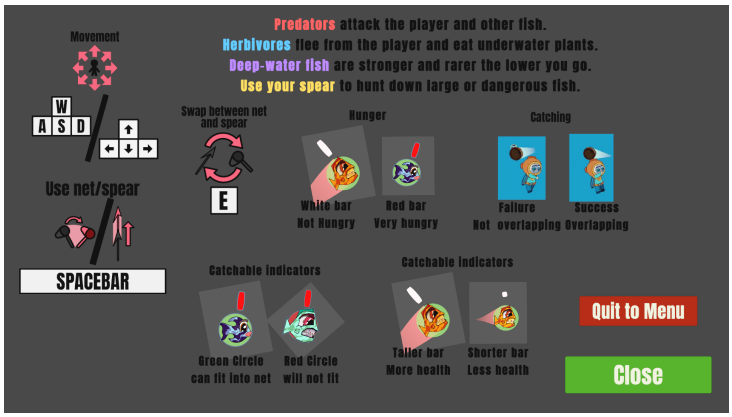
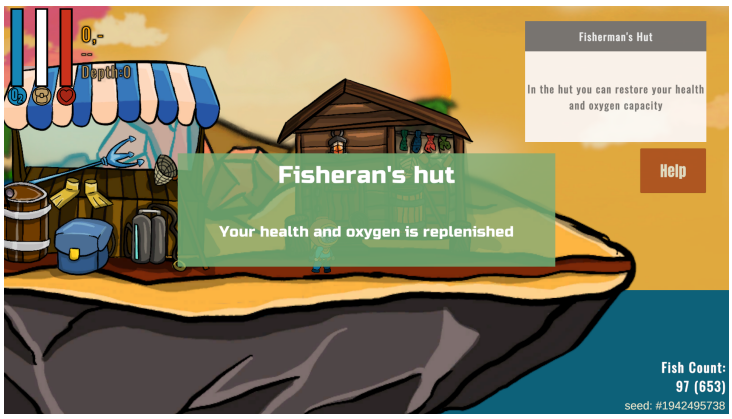


Figure 7.8: The quick help window explaining the basic game mechanics to the player with simple visuals.

The player's progression involves diving into the sea, catching fish, returning to the island, selling the fish, and purchasing upgrades. Players can level up individual statistics, such as speed, oxygen, or net size. The upgrade cost scales linearly with the current level (see Figures 7.9a, 7.9b, and 7.9c).

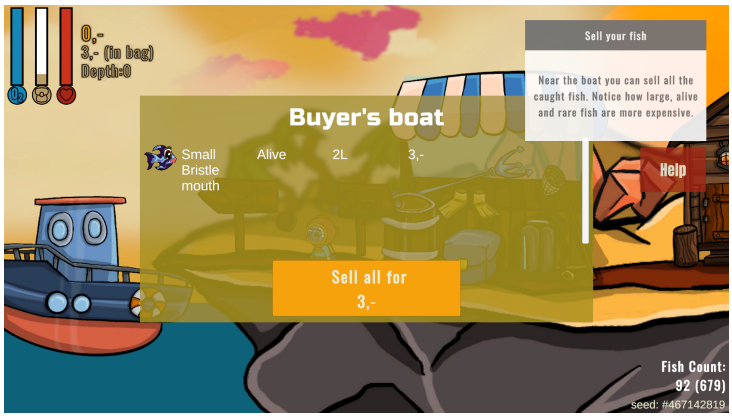
The game features a total of three different fish types: a herbivore, a predator, and a school of gobies. Each species includes three variants representing different difficulty tiers: weak, medium, and hard. These variants differ in attributes such as speed, size, strength, and behavior.

Fish display their status using visual indicators (see Figure 7.10). A bar above a fish shows its health (the size of the bar) and the its hunger (color of the bar). The circle behind the fish is green when the fish can potentially fit into the player's net. Additionally, the color of the trail behind the fish shows its current speed.



(a) Fisherman's hut menu. Place for player to instantly replenish health and oxygen.

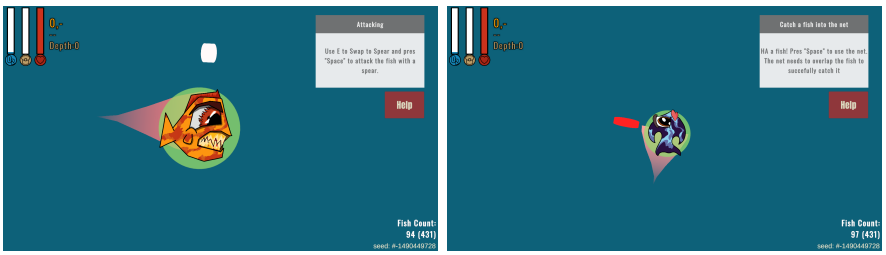
(b) Upgrades menu. Here player can pick player stats to upgrade.



(c) Selling menu. Here player can sell content of its inventory. The shop shows fish name, icon, cost and whether it was caught dead or alive.

Figure 7.9: Functional buildings in the game. In the top-right corner the contextual pop-up can be seen.

All game visuals were personally illustrated by the author using Procreate and Affinity Designer.



(a) Predator fish with low health, not hungry and will fit into the net, moving with high speed.

(b) Herbivore fish with full health, Very hungry and will fit into the net, moving with high speed.

Figure 7.10: Fish indicators explained.

7.2.1 Player feedback

The game was tested by 12 players. Some filled a short form with questions and multiple choice answers, while other provided their insights during an open discussion. Several key points emerged from their feedback and will be discussed here.

A common issue encountered by players was the difficulty of catching the fish during their first few attempts. Initially, players found the fish-catching mechanic frustrating, too difficult and their successful catches were not consistent. This was fixed by giving players a slight room for error. In the first version, the requirement for the 100% overlap between the net and the fish's body was too strict - the overlap was adjusted to 90% which made the overall experience much more pleasant.

The players often drowned or were eaten by predators. The game did not sufficiently warn the player when loosing health. This was later improved with an effect of blinking red screen when the player took damage.

Numerous smaller bugs were also reported. The labels in the shop menu not refreshing at time, issues with collisions between player and the main island and issues with scaling the instantiated corpse of the fish.

Overall, the players enjoyed the interaction with the fish and found the basic gameplay loop engaging. However, the lack of the long-term content, the lack of variability between the fish types and very simple upgrades were not enough to maintain the player's interest. The possible additions in terms of content will be discussed later in Section 8.

Some players never used the fisherman's spear, since they upgraded the size of the fishing net to an excessive extent, allowing them to catch all the fish from a large distance without being threatened. This can be fixed by setting a maximum size of the fish net and forcing the player to weaken the fish before letting the player catch them.

Some player reported unusual behavior when the fish was cornered by the player and the environment. The fish exhibited a jittery movement, likely

due to inability of the fish to surrender or stop moving when cornered.

Lastly, the players were asked to report the seed used to generate the world if they encountered lag spikes or crashes. The lag spikes could be caused by issues in world generation. However, no such problems occurred during the testing sessions.

7.2.2 Comparison with real life examples

To compare the simulated behavior with real-life examples, I visited Mořský svět Praha and recorded videos of fish in their aquariums.

It is difficult to compare the similarities and differences directly, since the game is in 2D while real fish move in 3D. However, some similarities are clearly visible. The overall schooling behavior of smaller fish in the game seems similar to its real-life model. In both the real footage and the simulation (see Figures 7.12 7.11), fish tend to stay close together and maintain a similar heading. While moving, they avoid collisions with each other. More specifically, they become startled when another fish gets too close and react quickly. These components are roughly modeled using the flocking rules discussed earlier.

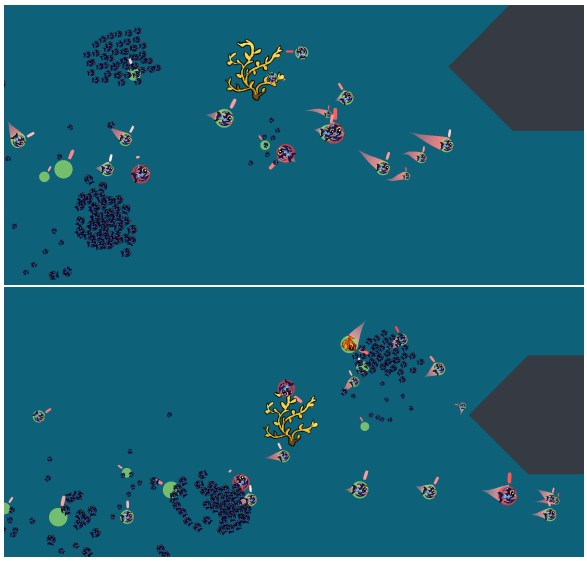
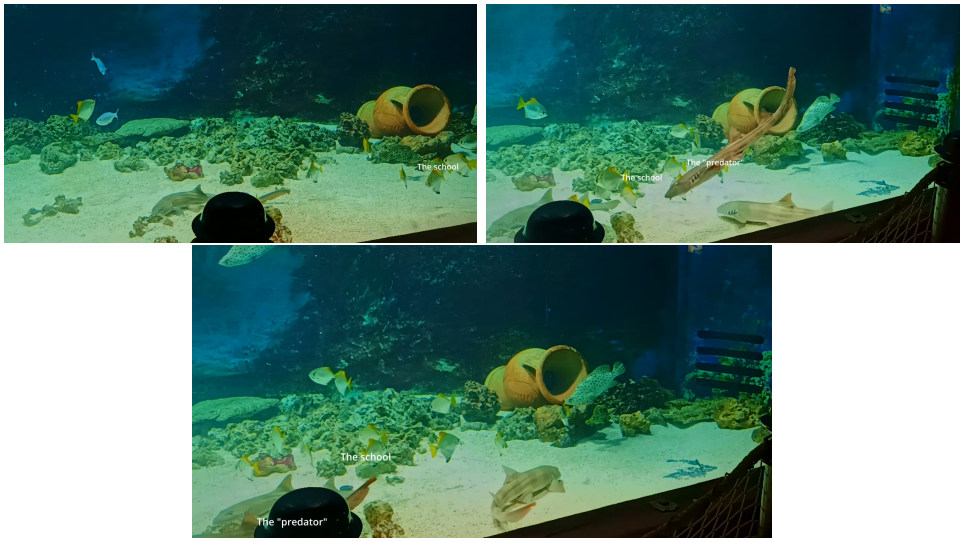


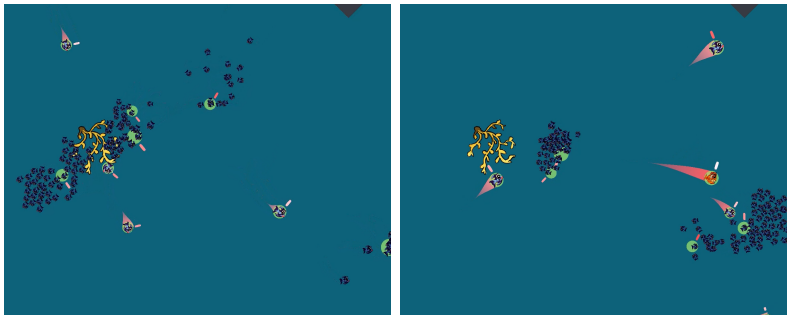
Figure 7.11: Schooling fish in the game. We can see the fish being aligned together, while eating the yellow plant. We can also see multiple smaller schools.

As can be seen in the attached video (see Appendix A.2.1), an interesting moment occurred when a large fish swam through a school of smaller fish: the school split and moved away from the larger individual. A very similar behavior can be observed in the game (see Appendix A.2.2) when a predator disrupts a group of fish (see Figure 7.12). We can observe similar behaviors; however, the real-life fish react much slower than those simulated.

A major difference from real life is that, in the aquarium, fish often remain stationary for longer periods. This does not happen in the game, since the



(a) Large fish startling a group of smaller fish (see Appendix A.2.1).



(b) School of fish before and after predator approaches in the game (see Appendix A.2.2.

Figure 7.12: Comparison of real and simulated group behavior when a school is startled by a large fish or predator.

fish are constantly moving, even if without any reason. The simulation does attempt to approximate this behavior by reducing movement speed when a fish has no reason to move.

An area for improvement in the simulation is the fish movement animation. In real life, fish move by bending their bodies sideways. This could be simulated in the game using sprite deformations to create a more lifelike swimming animation.

Chapter 8

Conclusion and future work

In this project, I have explored various algorithms that exhibit group behaviors. I have examined steering behaviors in depth and used them to create a fishing and diving game. The game's terrain is procedurally generated, and multiple fish species were designed. Herbivores seek plant-based food sources, while predators hunt herbivores. In addition to larger fish, I have implemented a school of gobies, which controls a large group of miniature fish.

To ensure smooth performance, I have analyzed performance of the game and improved it using a space-partitioning grid and chunk loading. I have compared the movement and group behavior of the fish to real-life footage of fish swimming in aquariums. Lastly, multiple players tested the game and provided valuable feedback, which I took into account and used to adjust and improve the game accordingly.

Although the game successfully demonstrates group behavior and real-time interaction, there are some limitations to address. For example, fish sometimes fail to recognize that a food source is accessible if it is too close to a wall, due to their simplistic collision avoidance logic. This is fine-tuned with the utility functions; however, it can still cause issues in particular cases. Additionally, fish do not yet have the ability to stop moving or remain idle when appropriate, which can lead to unnatural motion when cornered by player. On the technical side, while chunk loading currently works well on a per-layer basis, switching to a block-based loading system could further improve performance. Currently, the fish in the unloaded chunks cannot move into the loaded chunks and vice versa, which could be improved by using a harshly simulated movement of fish when unloaded.

The design and implementation of these behaviors was a valuable learning experience. It was particularly interesting to balance the complexity of the fish AI, while showcasing interesting and behaviors, that are fun to engage with. Navigating fish through a dynamically changing environment required a careful combination of algorithms and tuning of parameters. The project offered insights into both game development and the challenges of simulating lifelike group behavior in real-time systems.

While the current version of the game serves as a solid foundation, multiple improvements are planned to enhance gameplay, some of which were proposed by players in their feedback:

- *Interesting locations:* Add areas containing treasures or maze-like caves to improve the exploration aspect of the game.
- *Encouraging spear usage:* Currently, players can upgrade the net to catch any fish, so they do not have to learn how to use the spear. A better approach could involve limiting the net to only catch weakened fish, making the spear a necessary tool.
- *Fish mating system:* Right now, fish only spawn at the beginning and when a different fish dies. Introducing a breeding system where fish can reproduce would add a new layer of depth and realism to the simulation.
- *Genetic mutations:* Allowing newly spawned fish to mutate based on a selection of the *best* existing fish (e.g., using a basic genetic algorithm) would increase the challenge for the player. Poorly mutated fish that cannot navigate terrain effectively could die off over time, simulating a natural selection.
- *New underwater creatures:* Introducing more species with unique behavior patterns would enrich gameplay. For example, sharks that relentlessly pursue prey, piranhas that hunt cooperatively in flocks, or pufferfish that inflate to damage the player's net. Unique creatures like squids could shoot ink to temporarily disable the player's abilities.
- *Skills and upgrades:* Currently, players can only improve base stats like movement speed and damage. Adding active skills such as a temporary swim "burst" for speed could make gameplay more dynamic and strategic.
- *Island buildings:* The fisherman's hut is currently only used to restore health and oxygen. Expanding its functionality to allow decoration or customization could give players more to do outside of fishing.
- *Quests and economy system:* Right now, the value of fish is based solely on size and species. Introducing dynamic pricing, by reducing the prices for commonly sold fish, could encourage players to seek out rare species. Adding time-limited quests, such as catching specific fish under certain conditions, could also increase engagement.

Bibliography

[BG95] Bruce Blumberg and Tinsley Galyean. "Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments". In: *Proceedings of SIGGRAPH 95, in Computer Graphics Proceedings, Annual Conference Series*. ACM SIGGRAPH. 1995. DOI: https://doi.org/10.1145/218380.218405.

[BOC23] Viktor Bocan. "Design a produkce hry". Osobní účast. Přednáška v rámci předmětu B4B39HRY - Počítačové hry, ČVUT FEL, Město Praha. Sept. 2023.

[CON24] Wikipedia contributors. *Marching squares*. [Accessed 14-04-2025]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Marching_squares&oldid=1230396469.

[CON25a] Wikipedia contributors. *Lévy flight*. [Accessed 14-04-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=L%C3%A9vy_flight&oldid=1277421664.

[CON25b] Wikipedia contributors. *Marching cubes*. [Accessed 16-04-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=Marching_cubes&oldid=1270693065.

[CON25c] Wikipedia contributors. *Travelling salesman problem*. [Accessed 11-04-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1283057172.

[DBS06] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. "Ant colony optimization". In: *IEEE Computational Intelligence Magazine* 1.4 (2006), pp. 28–39. DOI: 10.1109/MCI.2006.329691.

[DEV25] Deep Dive Dev. *Pathfinding Hordes of Enemies with Flow Fields*. Accessed: 2025-04-15. Feb. 2025. URL: https://www.youtube.com/watch?v=tVGixG_N_Pg.

[EME13] Elijah Emerson. "Crowd Pathfinding and Steering Using Flow Field Tiles". In: *Game AI Pro: Collected Wisdom of Game AI Professionals*. CRC Press, 2013. Chap. 23.

[FOE+21] Daniel Foead et al. “A Systematic Literature Review of A* Pathfinding”. In: *Procedia Computer Science* 179 (2021). 5th International Conference on Computer Science and Computational Intelligence 2020, pp. 507–514. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2021.01.034. URL: https://www.sciencedirect.com/science/article/pii/S1877050921000399.

[GAR23] Matěj Gargula. “Simulation of artificial inteligence for games”. Bachelor’s Thesis. Czech Technical University in Prague, 2023.

[JEZ23] Filip Jezowicz. “Interactive Installation with Underwater Flocking Animals for Children”. Bachelor’s Thesis. Czech Technical University in Prague, 2023.

[LAG21] Sebastian Lague. *Coding Adventure: Ant and Slime Simulations*. Accessed: 2025-04-13. Mar. 2021. URL: https://www.youtube.com/watch?v=X-iSQQgOd1A&.

[LAN98] Janet T. Landa. “Bioeconomics of schooling fishes: selfish fish, quasi-free riders, and other fishy tales”. In: *Environmental Biology of Fishes* 53.4 (Dec. 1998), pp. 353–364. ISSN: 1573-5133. DOI: 10.1023/A:1007414603324. URL: https://doi.org/10.1023/A:1007414603324.

[MIN23] MINTROCKET. *DAVE THE DIVER*. Video game. 2023. URL: https://store.steampowered.com/app/1868140/DAVE_THE_DIVER/.

[NKO07] D. Nieuwenhuisen, A. Kamphuis, and M.H. Overmars. “High quality navigation in computer games”. In: *Science of Computer Programming* 67.1 (2007). Special Issue on Aspects of Game Programming, pp. 91–104. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2006.06.008. URL: https://www.sciencedirect.com/science/article/pii/S0167642307000561.

[PIT98] Tony Pitcher. “Shoaling and Schooling in Fishes”. In: Garland, New York, June 1998, pp. 748–760.

[POP11] Markéta Popelová. “Steering techniques library for virtual agents”. Bachelor’s Thesis. Charles University in Prague, 2011.

[REY+99] Craig W Reynolds et al. “Steering behaviors for autonomous characters”. In: *Game developers conference*. Vol. 1999. Citeseer. 1999, pp. 763–782.

[REY25] Craig Reynolds. *Boids: Background and Update*. https://www.red3d.com/cwr/boids/. Accessed: 2025-01-13. 2025.

[REY87] Craig W. Reynolds. “Flocks, herds and schools: A distributed behavioral model”. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’87. New York, NY, USA: Association for Computing Machinery,

1987, pp. 25–34. ISBN: 0897912276. DOI: 10.1145/37401.37406. URL: https://doi.org/10.1145/37401.37406.

[RG20] Dian Rachmawati and Lysander Gustin. "Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem". In: *Journal of Physics: Conference Series* 1566.1 (June 2020), p. 012061. DOI: 10.1088/1742-6596/1566/1/012061. URL: https://dx.doi.org/10.1088/1742-6596/1566/1/012061.

[SIL+10] R. Silveira et al. "Natural steering behaviors for virtual pedestrians". In: *The Visual Computer* (2010), pp. 1183–1199.

[TOM13] Jakub Tomek. "Navigation Mesh for the Virtual Environment of Unreal Tournament 2004". MA thesis. Charles University in Prague, 2013.

[TOO16] Game Maker's Toolkit. *How (and Why) Spelunky Makes its Own Levels*. Accessed: 2025-04-13. Apr. 2016. URL: https://www.youtube.com/watch?v=Uqk5Zf0tw3o.

[TRA20] The Coding Train. *Coding Marching Squares*. Accessed: 2025-04-13. July 2020. URL: https://www.youtube.com/watch?v=0ZONMNUKTfU.

[UBI23] Ubisoft. *Assassin's Creed IV: Black Flag*. Accessed: 2025-04-26. 2023. URL: https://www.ubisoft.com/en-gb/game/assassins-creed/iv-black-flag.

[UNI24] Unity Technologies. *Gizmos and Handles*. Accessed: 2025-05-14. 2024. URL: https://docs.unity3d.com/6000.2/Documentation/Manual/gizmos-and-handles.html.

Figures

Tables

Appendix A
Attached Files

This appendix provides an overview of the supplementary files submitted along with this thesis. These include video recordings, game build, and source code. The files are organized into folders for easier navigation.

A.1 Folder Structure

The attached files are structured as follows:

- **Videos:** Contains video recordings of both real-life and in-game fish schooling behavior used for comparing their behavior.
- **Build:** Contains playable game build for Windows.
- **Source:** Contains the Unity project files necessary to open and modify the game in the Unity Editor (game was develoepd with Unity Editor version 2022.3.10f1).

A.2 Videos

A.2.1 School of Fish Startled by Large Fish in Aquarium

This video shows real footage of a large fish swimming through a school of smaller fish in a controlled environment. As the predator approaches, the school reacts by splitting momentarily.

- **Filename:** *predator_startle.mp4*
- **Location:** Captured in a closed aquarium at Mořský Svět Praha.

A.2.2 School of Fish Startled by Predator in the Game

This video demonstrates the same behavioral concept simulated within the game. A predator entity causes the virtual school to break apart in response.

- **Filename:** *predator_startle_game.mp4*
- **Location:** Captured during a simulated scenario in the game.

Appendix B

User manual

Below, the basic game mechanics, tips and controls for the game are explained.

B.1 Game Overview

- Objective: Dive as deep as possible, discover all fish species, and survive.
- Survival threats: Oxygen depletion, water pressure, predators.
- Core loop: Catch fish, sell them in the shop, upgrade stats, explore deeper.

B.2 Interaction with the fish

- Net: Use to catch fish alive (full overlap in one sweep required). Live fish are worth full price.
- Spear: Give damage to fish. Dead fish are worth half the price.
- Swap tools with E.

B.3 Locations

- Fisherman's Hut: Instantly restores health and oxygen.
- Shop: Sell caught fish (Click button or Enter).
- Upgrade Shop: Improve player stats via mouse clicks.

B.4 Essential Upgradable Stats

- Swimming Agility: Improves turning underwater.
- Movement Speed: Increases swim and move speed.
- Fish Net Size: Increases net radius, essential for progression.

- Pressure Resistance: Allows deeper diving.
- Backpack Capacity: Allows player to carry more fish.
- Max Oxygen: Allows to spend more total time underwater.

B.5 HUD

- Bars in top left corner: Oxygen, backpack capacity, health
- Popup window in top right corner: Contextual hints for current events.
- Help button: Opens the quick help window.
- Fish count in bottom right: Shows currently existing fish count, count of the loaded fish, and count of total gobies isndie the brackets.
- Numbers in top left corner: Current player's currency, the potential income if player was to sell everything in the bag, the current depth.

B.6 Controls

Action	Key
Move	WASD / Arrows
Use Net / Spear	Space
Swap Tool	E
Sell Fish (Shop)	Enter
Upgrade (Shop)	Mouse Click
Focus on Random Fish	Numpad 1
Focus Back on Player	Numpad 2

Table B.1: The game's controls. The focusing on fish and back on player is intended for debugging and creative purposes.

B.7 Game modes

- Survival: Intended way of playing with upgrades, survival mechanics.
- Creative: Intended for testing and exploration, player has increased base stats to an absurd amount.