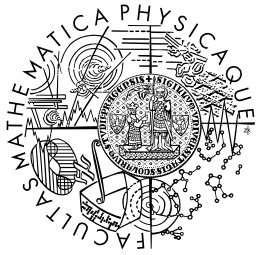

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Ondřej Vaic

Framework for Realtime Realistic and Interactive Simulations of Animal Flocking

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Pavel Ježek, Ph.D.

Study programme: Computer Science

Study branch: Visual Computing and Game Development

Prague 2025

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank the supervisor of this thesis, Mgr. Pavel Ježek, Ph.D, for his feedback and advice. I am very grateful to my wife, BSc. Ji Hyun Kim, for creating large portion of the figures used in this thesis. Lastly, I want to thank my family and friends for supporting me throughout the studies and writing of this thesis.

Title:

Framework for Realtime Realistic
and Interactive Simulations of Animal Flocking

Author: Bc. Ondřej Vaic

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract:

Flocking behavior of groups of animals is a fascinating natural phenomenon. In computer games, flocking simulations can enhance visual realism or create novel mechanics. This thesis presents a framework for flocking in the Unity game engine, designed for simple use by both game designers and programmers. The framework provides a modular architecture, allowing developers to add, remove, modify, or replace individual components, such as behaviors, enabling usage in various scenarios. Furthermore, the thesis introduces several novel theoretical advancements to the conventional methods of modeling flocking behavior.

The thesis first deconstructs a flocking model into modular components. Each component is then analyzed from a theoretical perspective with existing implementations to develop a robust and customizable flocking model. These ideas are then used to create a framework in Unity, structured as a modular parallel pipeline that allows customizability and improves performance. Additionally, a graphical user interface enables users to configure the pipeline without programming knowledge. The framework is later used to create a game scene, with animals using up to even 14 different behaviors. A key theoretical contribution enabling granular control over such complex flocking systems is the introduction of behaviors that return a normalized *desire* value together with a desired velocity. This enabled an improved method of blending between each desired velocity.

Keywords: flocking steering behaviors ECS Unity computer games

Contents

1. Introduction

Computer games often have a goal of creating a realistic and immersive experiences. Virtual worlds often contain simulations of different natural phenomena in order to increase the immersion. One fascinating natural phenomenon is flocking, the collective motion of groups of animals. Observing a large flock of birds, we can see a beautiful complex movement of the group as a whole. The Figure 1.1 highlights this phenomenon with a large flock of starlings.

Figure 1.1: A screenshot from a video on National Geographic's Youtube channel [1]. It captures the flocking of a large number of starlings.

There are different use cases for simulated flocking in computer games. One use case is a simple simulation of birds or fish purely as a visual element of the game. These simulations only serve the purpose of creating an interesting and realistic virtual environment. They do not necessarily need to react to the player or other NPCs. In this case, the birds' paths could be animated by hand instead of using a simulation. However, for a larger number of birds, this becomes very cumbersome.

In other cases, it can be desirable to have the simulated animals react to the environment or the player. An example of such interaction could be a flock of fish dispersing when a player steps into a river, or a flock of birds scattering upon hearing a gunshot. These features could increase the immersion of a game. While birds far up in the sky may not need to interact with the environment, fish must at least avoid running into rocks and the ground. If interaction with the environment is required, a flocking simulation becomes necessary.

Some games may even incorporate flocking as a part of the gameplay, or flocking may even be the core game mechanic. An example of such game is "Flock!" [2]. In this puzzle game, the player controls a UFO, which herds sheep into a specified location. The player's interaction with a simulated flock can create many interesting situations. Obstacles or predators can split the flock into multiple smaller flocks, creating difficult choices for the player. The animals can be pushed off cliffs by their flockmates in narrow paths, as shown in Figure 1.2, or they might wander off the cliff by simply following the animals directly in front of them.

Figure 1.2: A screenshot from the game Flock![2]. The UFO ship herds the sheep towards the end of the level, because of the narrow space, some sheep end up falling into the water.

The interaction between a flock of animals and a player can be an interesting main mechanic. For most games, however, a flocking simulation would be more of an interesting feature of a virtual world. Furthermore, the development of computer games is usually very time consuming and expensive. For these reasons, developing a flocking simulation could be seen as an unnecessary expense in the development a game, especially if it is not the main mechanic, but only a visual element. Large game companies may be able to afford to develop their own flocking solution, in order to slightly improve the immersion of their environment. Smaller studios with more limited resources will likely choose to allocate their effort elsewhere. A framework which solves a large portion of common use cases for flocking simulations in games could be used to reduce the cost and effort in integrating this phenomenon into game environments. It would be especially valuable for those small studios with strict budget constraints.

When looking at which game engines small game companies use, it is clear that Unity [3] by far the most common choice. A survey conducted by Developer Nation indicates that 48% of indie developers use Unity [4]. While researching solutions for this problem that have already been implemented for Unity, we have not found one which would be sufficient for more complex use cases, especially when interaction with a flock is the a main game mechanic. In Section 1.2, we will discuss what we consider to be important requirements for a flocking framework to be sufficient for more complex use cases. Furthermore, we will briefly look at and evaluate some solutions for flocking in Section 1.2.

For the aforementioned reasons, the goal of this thesis will be to create a framework for flocking simulations, which conforms to requirements which will be introduce in Section 1.1. Due to its popularity among small game studios and lack of high quality flocking frameworks, Unity was chosen as the target game engine for this framework.

1.1 Flocking Framework Requirements

Flocking simulations in computer games can have a large number of different usages as discussed earlier. Different use cases will have different requirements, which will now be discussed in detail.

1.1.1 Performance

Flocking is not be the main feature for most games, it is usually just a slight environment enhancement. For this reason, it is crucial that the framework does not consume significant amount of computational resources that could be used for core gameplay features. This poses a challenge as for example murmurations of starlings can contain up to millions of individuals [5] as is the case in the Figure 1.1. However, murmurations are a special case which only happens at certain times of the year, and smaller ones can consist of 500 birds [6]. For sheep or other cattle, the size of a flock is likely to be in tens to hundreds of individuals.

The framework must me able to handle common sizes of flocks without taking up significant portion of computational resources on an average gaming PC. Usually the target frame rate for games is 60 frames per second, which means that each frame can take up to 16 milliseconds. Since the flocking simulation may not even be an important part of the game, we believe that 1 millisecond, a sixteenth of available resources, is a reasonable requirement. The performance will however depend on the number of animals. We have chosen 1000 animals as a reference number because it is sufficient for larger flocks of cattle and still suitable for smaller murmurations.

For concrete performance testing, the author's gaming laptop was chosen as a reference system. It features a 6 core i7-8750H CPU, 32GB of RAM, and NVIDIA GeForce RTX 2060 graphics card. The target is a simulation on the reference system with at most 1 millisecond of run time for 1000 individual animals, which do not interact with the environment in any way.

1.1.2 Modularity

While different animals will likely need to behave differently in some cases, a flock of birds shares many similarities with a school of fish in many ways. It should be possible to share the code for the similar parts across different types of animals when needed, and compose new types of animals from existing code. The target is to create modular "blocks", which can be composed together to create new types of animals.

1.1.3 Extensibility

Extensibility complements the modularity requirement. As discussed, it would be beneficial for the framework to be modular, allowing easy composition of existing modular "blocks" into new types of animals. While this is good for quickly creating new types of animals from the existing "blocks", sometimes it will be necessary to create new "blocks". The framework should be designed with extensibility in mind to allow this.

This is especially important in situations where interaction with a flock of animals is an important game mechanic. It is impossible to predict all different kinds of animals, what data defines them, and how they should react to one another and the environment. The framework should be designed to make it easy for programmers to define new types of animals and behaviors. The main reason to use a framework is to save development time. Therefore, while the framework should offer "plug and play" solutions for common functionality, it should also be easy to extend.

1.1.4 Flexibility

The framework should be flexible enough to support different types of animals with varying behaviors. As discussed, the framework should be divided into modular "blocks" which can be composed together to create new types of animals. The interface of these "blocks" should impose minimal restrictions on what can be done with them.

This allows for cases such as moving in 2D or flying in 3D, or walking on a surface in 3D . Other cases where flexibility is needed is that while in simulations a bird often only needs to know the positions and velocities of other birds to determine its own movement [7], sometimes it might be necessary to take other information into account. For example, a bird might need to know the position of a predator's mouth to better determine how to avoid it. In other words, the framework should not be restrictive in what can be done with it.

1.1.5 User-Friendliness Design for Game Designers

The previous requirements focused on programmers, this one is focused on game designers. In a typical game development team, there are programmers and game designers. Game designers usually want to have some high level control over systems within the game to adjust them for specific needs. While game designers tend to have some technical knowledge, they generally cannot be expected to understand technical details or write code. This is why it is crucial that the framework offers a user friendly high level control to tweak the behaviors. Ideally designers should be able to quickly create and tweak new types of animals without having to write a single line of code.

1.2 Current Solutions in Unity

This section examines selected implementations of flocking in Unity. They are evaluated based on the requirements described in Section 1.1. This evaluation will help identify issues that our framework should address and guide decisions on which Unity technologies to use. While the performance requirement is objectively quantifiable, the other four requirements are not. For this reason, big part of the evaluation will rely on our views of the evaluated solutions.

1.2.1 FlockAI

The first implementation to analyze is FlockAI [8], available on Unity Asset Store for €10 at the time of writing this part of the thesis (January 2024). At the time of submission of this thesis (January 2025) the package has been renamed to GroupAI, price increased to €13.8 and some features were added. Our analysis and decision making is based on the older version. We chose this asset because it is relatively popular and cheaper than most other paid assets. We wanted to explore what was available for a lower price, as our main target is studios with a lower budget.

Performance

First, a scene simulating 1,000 birds with no interaction with the environment was run. Upon analyzing it with a profiler, one iteration of the simulation took around 65 milliseconds on the reference hardware. This is far from the desired goal of 1 millisecond.

Modularity

The whole simulation is written in one large file that tries to cover all cases at the same time. The file also contains long functions with many responsibilities. For this reason, the code is not very modular, because a change in one place can affect many other places.

Extensibility

The main part of the framework is implemented as one large class containing 1950 lines of code. It is implemented as a MonoBehaviour which updates all the currently simulated animals. All interactions are programmed directly in this class, so creating a new interaction would require writing the code directly into it. In other words, the interactions are tied directly to the flocking simulation, rather than to individual types of animals, which is not very extensible

Flexibility

The asset provides many different examples of how to use it. It contains fish, birds, and even cows. It does not provide a way to have different behaviors for different animals, but it does allow modification of properties related to movement, such as maximum speed. There is some flexibility, but we are missing an option to have different behaviors for different animals.

User-Friendliness for Game Designers

The simulation is configured with one `ScriptableObject` which contains all the options. This would make it easy to configure the simulation as everything is in one place. This is straightforward to use, even for non-technical users.

1.2.2 Sebastian Lague's Boids

The second implementation to analyze is Sebastian Lague's Boids [9]. Sebastian Lague is a very popular Unity content creator on Youtube. He created an open source project which implements the Boids model [10], and described it in a video on his channel [9]. The video has 1.5 million views at the time of writing, making it the most popular Youtube video on this topic in Unity, and as far as we know, the most popular video on simulated flocking on Youtube in general. Therefore, we consider it a good implementation to analyze.

The project does not attempt to be a framework, but rather a simple implementation of the algorithm. Lague first experimented with a simulation on the CPU with a spatial partitioning data structure to improve performance. He then improved the performance by moving the simulation to the GPU, disregarding the spatial partitioning. This structure will be discussed in more detail in the Chapter 5. The project mainly consists of one type of boid, behaviors written in a compute shader, and a manager script that binds these parts together and updates the boids.

Performance

Again we ran a simulation of 1000 boids (in this case fish) with no interaction with the environment, after commenting out the parts responsible for obstacle avoidance. The simulation part on the GPU took around 1.5 milliseconds, the rest of the simulation took around 1.2 millisecond on the CPU, or around 2.7 milliseconds in total.

This performance is much better than the previous one, but still not quite at the desired level. The author mentions that the performance could be improved by combining the spatial partitioning data structure with the GPU implementation. While using the GPU for the simulation sounds promising in terms of performance, it is important to note that the GPU could already be fully utilized by the game's graphics, potentially creating a bottleneck.

Modularity

As mentioned earlier, the project can be divided into three main parts: the boid, the GPU-based behaviors, and the manager that binds them together. While the separation between the boid and the behaviors improves modularity, the manager directly binding them does not, as it directly depends on the type of the boid and the specific compute shader. Otherwise, the code is split into smaller functions, many of which could be replaced by different implementations without changing the rest of the code. This is achieved through good coding practices, such as keeping small functions with a single responsibility. However, it lacks any kind of abstraction through interfaces to further decouple the code.

Extensibility

The code is not very extensible, which is understandable as this was not the goal of the project. New behaviors would either have to be directly written into the compute shader, or a new compute shader would have to be written. These changes would need to be reflected in the manager which binds the compute

shader and C# together. The manager would have to be modified for each new behavior. Moreover, new types of animals might require a whole new manager.

Flexibility

The project offers only one type of boid, which is a fish. In the current implementation, it is not possible to have different behaviors for different types of animals, or to have movement in 2D or on a 3D surface. For example, changing the movement to 2D would require either modifying the boid itself, or creating a new type of boid. In case of creating a new type of boid, the rest of the simulation would have to be rewritten to accommodate it, due to the low extensibility.

User-Friendliness for Game Designers

The project offers one way to configure the simulation, which is again through a `ScriptableObject`. This is straighfoward to use, even for non-technical users. However, there are not many options to configure.

1.2.3 Unity's DOTS Boids Sample

Lastly, there is a sample Boids implementation project [11] provided by Unity to showcase their new Data Oriented Technology Stack (DOTS) [12]. The main purpose of the project is to showcase the performance gains when using DOTS.

It assumes one type of a boid (fish). A query is made in a system to find all the entities with a particular tag and required components. Then a series of parallel jobs, compiled into native code with the burst compiler [13], is scheduled to update the boids. On top of parallelization and the benefits of the burst compiler, the implementation also uses a spatial partitioning data structure to improve performance.

Performance

As expected, since the project is trying to showcase the performance benefits of DOTS, the performance is very good. A simulation of 1000 boids with no interactions with the environment was run. The simulation took around 0.2 milliseconds on the reference machine, which is well within the desired target of 1 millisecond.

Modularity

The simulation itself is split into multiple jobs, some of which could be replaced by different implementations or reused for other simulations. The code defines a series of jobs that depend on each other only through the data they read and write to buffers. Furthermore, the jobs do not depend on a specific type of boid, but only on the set of components that the boid has. This is a good design for modularity, but it would be even better if the jobs that are run could be decoupled from the system that schedules them.

Extensibility

The project is not a framework, so it is not built with extensibility in mind. The behaviors are split into multiple jobs, so it would be possible to add new ones, but it would still be needed to schedule them in the main system. This is not very extensible, as this system would grow with each new behavior and would be difficult to maintain. Furthemore, defining new types of animals with different components and behaviors would require creating a new system with a new entity query.

Flexibility

The project offers only one type of boid, a fish. It is not possible to have different behaviors for different types of animals, or to have movement in 2D or on a 3D surface. For example, changing the movement would require creating a new system for each type of animal, which would be almost identical to the current one, but with a different job to update the boids position and a different entity query. Alternatively, branching could be used in the job to determine the movement, but this would have a negative impact on extensibility.

User-Friendliness for Game Designers

The only way to configure the simulation is through properties on the prefab of the boid itself. This is again very simple, but there are not many options to configure (only 6 properties). It is possible to create new types of boids by creating new prefabs with different properties, but there is no option to change which behaviors are used without writing code.

1.2.4 Conclusion

We gave three examples of flocking simulations in Unity. One utilizing MonoBehaviours, one utilizing compute shaders and one utilizing DOTS. They were evaluated in reference to the requirements described in Section 1.1. It was apparent that none of the solutions were sufficient for more complex use cases, and thus there is a need for a framework that would satisfy these requirements. We will now summarize the findings and draw a conclusion as to which technologies will be used for our framework.

The option using MonoBehaviours is not very performant. It could be improved by using better data structures, but it would likely still not come close to the performance of the other two. We believe that running the simulation on the GPU with proper data structures could achieve the best performance. However, compute shaders are very low level and the abstractions needed for a modular and extensible framework would be difficult to implement. Also, the GPU is already being used for the actual graphics, so this could create a bottleneck. The DOTS implementation is performant enough, and because it uses C#, it would be easier, but still challenging, to create a modular and extensible framework. Moreover, the jobs utilize all available CPU cores, which can otherwise be underutilized.

For these reasons, we decided to use DOTS for our framework. The main challenge will be to create a framework that strikes a good balance between all

the specified requirements. DOTS offers great performance, but at the cost of being more low level than traditional MonoBehaviours. This will make it more difficult to achieve modularity and extensibility. It will be important to find a good balance between performance and higher level abstractions, and between flexibility for different use cases and the ease of use for the common cases.

1.3 Flocking and Steering Behaviors

In the previous section, we have discussed three implementations of flocking simulations in Unity. They all derive from the so-called Boids model [7] to simulate flocking. The Boids model is probably the most well known model for simulating flocking behaviors. For this reason, it is a promising model for our framework. We will now discuss the Boids model and how it relates to steering behaviors in more detail.

First described by Craig Reynolds in 1987, the Boids model [7] is perhaps the most important and influential source on simulated flocking behaviors. The paper has since been cited over 14000 times [14] at the time of writing. The model was first used to simulate bats and penguins in the 1992 movie "Batman Returns", as Reynolds mentions on his personal website [15].

Reynolds later described the technique he used to simulate flocking in more detail in a paper titled "Steering Behaviors For Autonomous Characters" [16]. The technique is not limited to flocking simulations, it is more of general technique for determining the desired movement of a reactive agent [16]. Steering behaviors have since found many usages, especially in computer games.

1.3.1 Steering Behaviors

Steering behaviors have been one of techniques for creating various AI behaviors in computer games. One example is the AI spacecrafts in the game "Mace Griffin: Bounty Hunter" [17]. In sections of the game, the player fights against spacecraft controlled by the computer. We know that steering behaviors were used because the author of the AI system for this game describes his approaches in a paper titled "The long and short of steering in computer games" [18].

Further example of the use of steering behaviors is the mobile game "Fieldrunners 2" [19]. In this tower defense game, large numbers of units try to destroy the player's base, while the player has to build structures to defend against them. AI for this game was described by Pentheny in his paper titled "Efficient Crowd Simulation for Mobile Games" [20]. The paper mainly discusses how to efficiently simulate thousands of agents on a mobile device. Pentheny notes that utilizing multiple cores, which are available in modern devices, is important to achieve good performance. He also notes that because steering behaviors are independent for each agent, parallelization of them is trivial. We also observed the benefits of parallelization in Section 1.2, where the DOTS implementation was the most performant. This further supports our decision to use DOTS for our framework, because it offers easy parallelization through jobs.

1.3.2 Boids Model

Looking at open source implementations of the Boids model, it is clear that implementations can be quite simple and still providing a mesmerizing simulation. An example of this is an implementation by a youtuber Ben Eater; his implementation [21], public on GitHub, fits in only 218 lines of code of JavaScript. An interactive simulation can be seen on his personal website [22], here captured in Figures 1.3 and 1.4 taken around a third of a second apart.

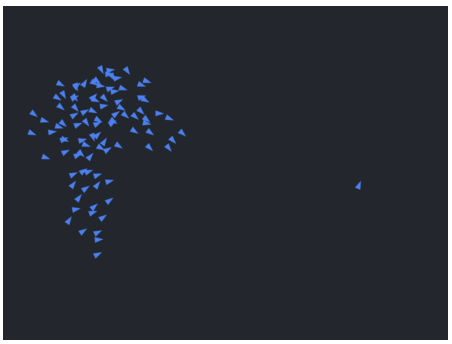

Figure 1.3: First screenshot of flocking simulation by Ben Eater [22].

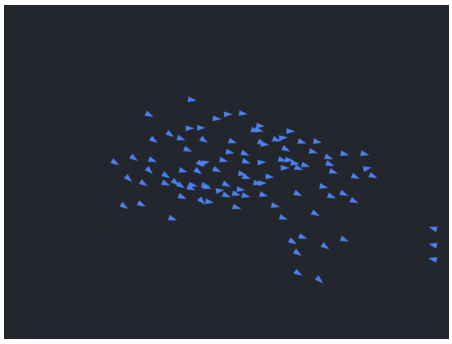

Figure 1.4: Second screenshot of flocking simulation by Ben Eater [22].

We find that while it is trivial to create a simple simulation, there is no standard way to create a Boids model. From the over 14000 citations [14] of the paper that first introduced the Boids model [7], a large number of these papers is concerned with implementating the Boids model. However, Reynolds only described the model in words. Reynolds later published additional materials on his website to further clarify his model [15], and an open source implementation of his steering library [23] was released. However, there is still a lot of room for interpretation and modifications.

We noticed that authors of other papers on flocking usually state that they use the Reynolds' Boids model. Interestingly, many of these implementations differ in subtle details that have different effects on the final simulation. One example discussed in more detail in Section 2.2 is that some authors describe the output of the behaviors as a desired acceleration, while others describe it as a desired velocity. This can be especially problematic when combining implementations of behaviors from different sources. Furthermore, even when using the same output semantics for behaviors, concrete implementations of the behaviors can still differ in how they determine the output.

We believe that these differences are partially caused by the lack of formalization of the Boids model. This is why we believe there is a need for a formalization of it, to better understand it and analyze different implementation choices separately.

1.3.3 Conclusion

In Subsection 1.3.1, we discussed steering behaviors. Given their success in developing models of flocking behaviors, widespread use in AI for computer games, and

potential performance benefits, this thesis will also work with steering behaviors as its technique for modelling flocking simulations. Throughout this thesis, the term "flocking" will refer specifically to the collective motion of animals, and the term "steering behaviors" will refer to the AI technique.

Furthemore, we have stated in the Subsection 1.3.2 that there is no standard way to create a Boids model, and that there is a need to formalize it. The formalization could help in understanding the model, analyzing different implementation choices, and finding the optimal structure for the framework.

1.4 Approach

We found that there is a need for a framework for focking simulations in Unity. We stated requirements for this framework in Section 1.1. Then, analyzed some existing solutions in Section 1.2, and concluded that using DOTS is the best option for our framework. Then, we discussed the Boids model and steering behaviors in Section 1.3, concluding that a model based on steering behaviors is a good option for our framework and that it would be beneficial to formalize it first. This section will now discuss the approach that will be taken to create the framework. Furthermore, the structure of this thesis will be discussed.

This thesis can be split into two parts. The first part analyzes approaches to flocking behaviors used in other sources. First, the high level structure of a flocking algorithm as it was introduced in the Reynolds' seminal paper [7] will be examined. This will be used to create a pseudocode formalization of it. This pseudocode will help to break down the algorithm into distinct parts. This will help with the modular design of the framework, and with the analysis of different implementation choices. The analysis of these choices will be used to find the best "out of the box" defaults for our framework, and to realise which parts of the framework should be easily extensible and flexible.

In the second half of this thesis, the framework will be designed based on the formalization, and with the requirements in mind. It will then be implemented using Unity's latest Data Oriented Technology Stack – DOTS [12]. Lastly it would be beneficial to test the framework by creating a game scene that uses it. This will help evaluate the framework in relation to its requirements, identify possible improvements and limitations, and showcase its use cases.

1.5 Goals

The main goal of this thesis is to create a framework for the Unity game engine that can be used to create flocking simulations. The secondary goal is to test this framework by implementing a simple game scene that uses the framework. The goals for the framework and the game scene will now be discussed.

1.5.1 Framework Goals

The requirements for this framework were discussed in more detail in the Section 1.1. In brief, the framework should have the following qualities:

1. *Extensibility for programmers* – It should be easy to implement new types of animals and behaviors.

2. *Modularity* – It should be possible to reuse parts of the framework between different types of animals.

3. *Flexibility* – The framework should not be restrictive in what can be done with it.

4. *High performance* – The framework should be able to simulate up to lower thousands of animals at interactive framerates.

5. *Usability for game designers* – It should be easy to configure the simulation without writing code.

1.5.2 Game Scene Goals

In order to test the framework thoroughly, a game scene will be developed with the following sample use case in mind. A game scene that has flocking as its main feature. It will be a simulation of relatively large numbers of animals, hundreds or lower thousands. There will be multiple kinds of animals in different environments, water, ground, air. The animals should be able to react to different elements of the environment, not just to other members of the same flock. They should take into account various aspects of the virtual environment, such as different kinds of animals, the player, the sounds, the physical objects in the scene etc.

If the framework meets all the requirements specified in Section 1.1, it should minimize the effort required to create this scene. The scene should run at interactive frame rates on the reference system which was described in the Subsection 1.1.1. Finally, a user without programming knowledge should be able to modify some aspects of the simulation.

2. Formalization of Boids Model

In Section 1.3 of the previous chapter, we discussed that the Boids model lacks a formalization, and why it is important to have one. The problem of insufficient formalization has already been described by Bajec et al. in their paper titled "The computational beauty of flocking: boids revisited" [24]. The paper also provides a formalization of an artificial animal, based on Moore automata, a type of finite state machine. It tries to be general enough to be able to model various animal behaviors. The authors later show that the Boids model can be modeled with their formalization.

The formalization provided by Bajec et al. is rigorous and includes a lot of mathematical notation, which makes it very general and unambiguous, but potentially difficult to work with. For this reason, this thesis will provide its own formalization, focusing only on the Boids model, as it was first described by Reynolds. This formalization will focus on decomposing the Boids model into separate functions, how they are composed together, and determining what is their "contract" [1] and "signature" [2]. Later this formalization will be used to analyze various implementation details and their consequences. This analysis will then help to decide on the structure of the framework and the implementation of the concrete parts of the model.

2.1 Our Formalization

The Boids model will now be formalized in a way to cover all the main ideas mentioned in the first description [7] of it. Implementation details will be intentionally omitted for now, they will be analyzed in next chapters.

The model was initially summarized with the following:

"Stated briefly as rules, and in order of decreasing precedence, the behaviors that lead to simulated flocking are:

1. *Collision Avoidance: avoid collisions with nearby flockmates*
2. *Velocity Matching: attempt to match velocity with nearby flockmates*
3. *Flock Centering: attempt to stay close to nearby flockmates" [7]*

As Reynolds notes, the first two rules can be interpreted as static collision avoidance and dynamic collision avoidance, respectively. The first rule encourages entities to move away from neighbors that are too close, while the second rule promotes matching velocities with nearby neighbors, reducing the likelihood of future collisions. The third rule is crucial for maintaining cohesion within the flock.

Reynolds also emphasizes that the calculation of these three behaviors only considers nearby flockmates. Additionally, he interprets the results of these behaviors as "acceleration requests". *"Each behavior says: 'if I were in charge, I would accelerate in that direction.' "*[7] It is mentioned that all of these requests

[1] Contract of a function are the assumptions about its inputs and outputs.

[2] A signature of a function are the types of its inputs and outputs.

are fed into *navigation module*, which then determines one desired acceleration. Afterwards, the *pilot module* and the *flight module* determine the final movement [7].

Even though the paper does not give a pseudocode or concrete equations, an attempt can be made to formalize this general idea in order to start expanding and analyzing it. The main information for the simulation is a set of boids.

$$Boids = \{boid_0, boid_1, \ldots, boid_{n-1}\}$$

There is some information which should be tracked per boid. It is clear from the three rules that it must be at least a position and a velocity. Exactly what information needs to be tracked per each boid is bound to differ for specific needs.

$$boid_i = (position_i, velocity_i, ...)$$

The paper by Reynolds mentions three main behaviors, commonly referred to as *Separation*, *Alignment*, and *Cohesion*. There have been numerous extensions to the algorithm, many of which introduce new behaviors. One such example was described by Hartman and Beneš in their paper Autonomous Boids [25]. It extended the Boids model to include a behavior where some boids can temporarily become leaders. These leaders aim to escape to areas of lower boid density, while others attempt to follow these temporary leaders.

Therefore, in general, k different behaviors can be assumed. For each boid, k results must be calculated from each of the k behaviors. The result depends only on the given $boid_i$ and a subset of all boids. In the original Boids implementation, the behaviors return desired accelerations [7]. The type of the results is generalized to allow more information to be returned. A single behavior can be thought of as a function of two parameters, a $boid_i$ and a subset of $Boids$. The function returns a single result. Let these functions be referred to as *Neighbor Behaviors*, since they depend on the neighbors.

$$nBehavior_j : (boid_i, neighbors \subseteq Boids) \rightarrow result_{ij}$$

The paper by Reynolds further discusses the importance of considering only nearby flockmates in order to compute a behavior's contribution [7]. It is mentioned that all boids within a certain radius is one way, and the realism can be improved by simulating a limited field of view. This step can also differ, which is why it will be generalized as well. Furthermore, each behavior could also need a different subset of boids, which is why k functions which determine neighborhood for $boid_i$, for behavior $nBehavior_j$, will be assumed. Let these functions be referred to as *Neighbor Queries*.

$$nQuery_j : (boid_i, Boids) \rightarrow neighborhood_{ij} \subseteq Boids$$

A given behavior's $nBehavior_j$ result $result_{ij}$ for a given boid $boid_i$ can now be calculated as follows.

$$neighborhood_{ij} = nQuery_j(boid_i, Boids)$$
$$result_{ij} = nBehavior_j(boid_i, neighborhood_{ij})$$

Once all the results for a given boid are calculated, they must be used to determine one final result. In Reynolds' implementation, the final desired acceleration needs to be decided [7]. No restriction will be provided on this return type. This idea can be represented as a function from a set of results, to some final result value. Let this function be referred to as the *Merger*.

$$merger : (result_{i0}, \ldots, result_{ik-1}) \rightarrow finalResult_i$$

The final step is to take the $finalResult$ and use it to update the position and velocity of the boid [7]. The specifics of the movement are likely to differ. Let this function be referred to as the *Mover*.

$$mover : (boid_i, finalResult_i) \rightarrow boid_{i+1}$$

The next state of the simulation can now simply be expressed as:

$$Boids_{t+1} = \{mover(boid_i, finalResult_i) \mid i = 0, 1, \ldots, N-1\}$$

In this section, the building blocks of the boids model were formalized. We have essentially described which functions need to be implemented and how they are composed together, but we have not provided specific implementations. A specific algorithm **A** will depend on the implementations of these functions. What is needed to describe a specific algorithm **A** are the *Merger* and *Mover*, and a set of pairs of *Neighbor Behaviors* and their corresponding *Neighbor Queries*.

$$\mathbf{A} = \{ \quad merger, \quad mover, \quad \{(nBehavior_0, nQuery_0), \ldots, (nBehavior_{k-1}, nQuery_{k-1})\} \quad \}$$

The complete generalized algorithm can be expressed by the following pseudocode:

Algorithm 1: Boid Simulation Algorithm

Input : Boids, functions **A**.
Output: Updated Boids

```
for each boid_i in Boids do
    results = new array of size k;
    for j = 0 to k − 1 do
        neighborhood_ij = nQuery_j(boid_i, Boids);
        results[j] = nBehavior_j(boid_i, neighborhood_ij);
    finalResult_i = merger(results);
    boid_i = mover(boid_i, finalResult_i);
```

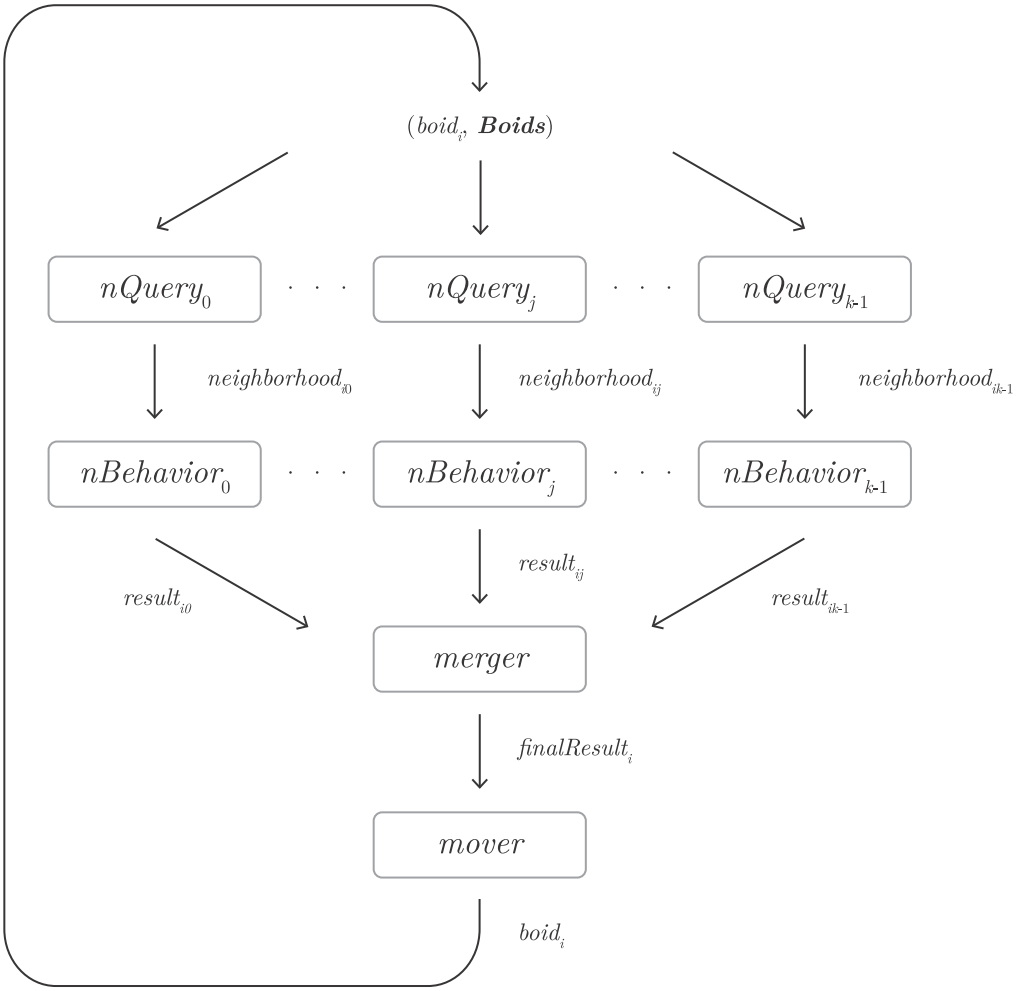

Figure 2.1: Diagram of Algorithm 1. It shows how a single boid $boid_i$ is updated in one frame of the simulation, assuming k behaviors.

The introduced formalization of the Boids model, Algorithm 1, can be visualized with the diagram in Figure 2.1. In the figure, the algorithm is decomposed into 4 different layers, the *Mover*, the *Merger*, the *Neighbor Behaviors*, and the *Neighbor Queries*. This decomposition into layers creates a modular structure, which is one of the requirements for the framework. Each box in the diagram represents a pure function, which means it can be replaced by another function with the same signature and contract without any side effects. This is modular and allows extensibility. In the upcoming chapters, we will analyze options for implementations of the functions in each layer separately.

It is clear from the diagram and the formalization that the algorithm is composed of a series of pure functions. Clearly, the *Neighbor Query* and *Neighbor Behavior* pairs can be run in parallel, and the diagram even indicates where synchronization is needed. This corresponds to parallelizing the inner for loop in Algorithm 1. Not only can the behaviors be run in parallel to each other, but all the boids can be updated in parallel as well. In Algorithm 1 this would correspond to parallelizing the outer for loop.

It is important to note that there are some requirements for the signatures of these functions. The output of all *Neighbor Behavior* must be of the same

type $\boldsymbol{T}$. The input of the *Merger* should be $\boldsymbol{T[]}$, and its output will be of an another type $\boldsymbol{U}$. This type must be the same as the input type of the *Mover*. If these type restrictions are met, the functions can be composed as shown in the diagram. $\boldsymbol{T}$ and $\boldsymbol{U}$ can be polymorphic types, which would allow a more flexible, and modular design. Another type consideration is the type of the boids, and the type of the neighbors. These can also be polymorphic types. This is important for the framework, as it must be able to work with different types of boids.

We have only discussed the signature of the functions, but the contract is equally important. For example, from signature point of view, acceleration and velocity are of the same type, a 3D vector. Mixing behaviors that return accelerations and velocities would not cause a compile error, but it could create a subtle runtime bug. This is because the semantics of acceleration and velocity are different. Commonly used options for behavior output semantics are briefly discussed in Section 2.2.

2.1.1 Other Steering Behaviors

So far, we described a formalization of the Boids model for flocking, the group motion of animals, which is the main focus of this thesis. It uses what we refer to as *Neighbor Behaviors*. However, in context of computer games, it may sometimes be useful to extend flocking with other steering behaviors. For example, steering behaviors to avoid collisions with meshes in the virtual environment. We will now introduce two more types of behaviors, which will occasionally be useful for computer games, and include them in our formalization.

Ray Queries and Ray Behaviors

In computer games, it might be necessary for the boids to not only interact with their neighbors, but also avoid collisions with the virtual environment's geometry. In context of computer games, the virtual environment is usually defined by the game scene's meshes using colliders. In order to react to the colliders, information about them needs to be queried first.

In Unity, information about the game scene's meshes can be gathered using ray casts. This approach was used for example in Sebastian Lague's Boids [9], [10] and FlockAI [8]. These two implementations were briefly discussed in analysis of implementations of flocking in Unity 1.2. Since this approach is common in Unity and other implementations, we will also use ray casting to gather information about the environment.

Functions which cast rays into the environment Env to collect information will be refered to as *Ray Queries*. The concept is similar to *Neighbor Queries* described earlier. *Neighbor Queries* collected information about which boids are neighbors. In general, there could be abitrary number k of *Ray Queries*, same as with *Neighbor Queries*. Formally, *Ray Query* could be expressed as:

$$rQuery_j : (boid_i, Env) \rightarrow rayResults_{ij}$$

where each $rayResults_{ij}$ contain information about the rays' nearest hits. Once information about meshes is queried, it should be used to affect the boid's behavior. This is similar to the concept of *Neighbor Behaviors* described earlier,

where result of a *Neighbor Query* is passed into a *Neighbor Behavior*. For our implementation, we define a second type of behaviors, *Ray Behaviors*, which are influenced by *Ray Queries*. Same as with *Neighbor Behaviors*, we assume each *Ray Behavior* to have its own *Ray Query* which passes information to it.

$$rBehavior_j : (boid_i, rayResults_{ij}) \rightarrow result_{ij}$$

Afterwards, all results $result_{ij}$ from *Ray Behaviors* should be merged together with the results of *Neighbor Behaviors*, using the *Merger*, into one final value, which is then passed into the *Mover*.

Simple Behaviors

So far, there are two types of behaviors: *Ray Behaviors* and *Neighbor Behaviors*. Both depend on result of some queries. However, there are some steering behaviors that only depend on data associated with the current boid $boid_i$. One example can be the wandering behavior, described by Reynolds in his paper about steering behaviors [16]. This behavior makes the boids wander around randomly. We will refer to these behaviors as *Simple Behaviors*. Formally this could be expressed as:

$$sBehavior_j : boid_i \rightarrow result_{ij}$$

As always, there could be an arbitrary number k of these behaviors. Each result $result_{ij}$ would later be merged with other results using the *Merger*.

Extended Formalization

An algorithm $\mathbf{A}'$ including kn number of *Neighbor Behaviors*, kr number of *Ray Behaviors* and ks number of *Simple Behaviors* could be formally described as:

$$\begin{aligned}\mathbf{A}' = \{ & \\ & merger, \\ & mover, \\ & \{(nBehavior_0, nQuery_0), \ldots, (nBehavior_{kn-1}, nQuery_{kn-1})\} \\ & \\ & \{(rBehavior_0, rQuery_0), \ldots, (rBehavior_{kr-1}, rQuery_{kr-1})\} \\ & \\ & \{sBehavior_0, \ldots, sBehavior_{ks-1}\} \\ & \}\end{aligned}$$

Similarly to pseudocode shown in Algorithm 1, one step of the simulation would be expressed with the following pseudocode shown in Algorithm 2 below. In it, *Ray Behaviors* and *Simple Behaviors* are added.

Algorithm 2: Boid Simulation Algorithm Extended

Input : Boids, functions $\mathbf{A}'$.
Output: Updated Boids

```
1  for each boid_i in Boids do
2      results = new array of size kn + kr + ks;
3      for j = 0 to kn − 1 do
4          neighborhood_ij = nQuery_j(boid_i, Boids);
5          results[j] = nBehavior_j(boid_i, neighborhood_ij);
6      for j = 0 to kr − 1 do
7          rayResults_ij = rQuery_j(boid_i, Boids);
8          results[kn + j] = rBehavior_j(boid_i, rayResults_ij);
9      for j = 0 to ks − 1 do
10         results[kn + kr + j] = sBehavior_j(boid_i);
11     finalResult_i = merger(results);
12     boid_i = mover(boid_i, finalResult_i);
```

2.2 Behavior Output Semantics

The previous section mentioned how the *Behaviors* and the *Mergers* can have different return types. We have observed that in most implementations this return type is mainly a 3D vector [3]. Although the return type is the same, we found that it is often the case that the semantics of the return type are different.

In the original Boids model, *Behaviors* and *Mergers* return desired acceleration [7]. In Reynolds' second paper, the semantics for these are desired forces [16]. Desired velocity can be the choice semantics as well, as Fray suggests in his paper introducing a technique called Context Steering [26], which was used to power Formula 1 AI in the 2011 game titled "F1". Later in the paper, he proposes using an array of scalars projected around the circumference of the entity. We have also found some authors interpret the vector as desired direction [27].

We see this as a further potential cause of discrepancies between different implementations. For any implementation, it will be crucial that the semantics are the same across all the behaviors, as we discussed in the previous section. This can become a problem if, for example, a behavior from one source is brought into a model which uses different semantics.

Now, assuming that the type is a 3D vector, we will show how it is possible to convert between different semantics. This can be useful if there is a need to use a behavior from one source in a model which uses different semantics, or to compare them. All the outputs here are 3D vectors, in Chapter 4 we will see that it can be better to use at least a 4D vector, where the last dimension denotes a "desire".

[3] Or a 2D vector in case of a simulation in 2D

2.2.1 Converting Between Workflows

Three main "workflows" were mentioned, force, desired acceleration, desired velocity. We will show how it is possible to convert between them if needed.

Euler integration is a common approach to approximate movement in computer games. Assuming acceleration $\vec{a}$, current velocity $\vec{v_c}$, current position $\vec{p_c}$ and a time step Δt. New velocity $\vec{v_n}$ and new position $\vec{p_n}$ can be calculated as:

$$\vec{v_n} = \vec{v_c} + \vec{a} * \Delta t \tag{2.1}$$

$$\vec{p_n} = \vec{p_c} + \vec{v_n} * \Delta t \tag{2.2}$$

Given a desired acceleration $\vec{a_d}$, the Equation 2.1 can be used to compute a desired velocity $\vec{v_d}$.

$$\vec{v_d} = \vec{v_c} + \vec{a_d} * \Delta t \tag{2.3}$$

The desired acceleration can be computed from desired velocity by rearranging the Equation 2.3.

$$\vec{a_d} = \frac{\vec{v_d} - \vec{v_c}}{\Delta t} \tag{2.4}$$

If the workflow is force, then an equation from the Newtons second law of motion can be used.

$$\vec{F} = m * \vec{a} \tag{2.5}$$

Rearranging the Equation 2.5 for a desired force $\vec{F_d}$ and a desired acceleration $\vec{a_d}$ gives.

$$\vec{a_d} = \frac{\vec{F_d}}{m}$$

Since the mass of an animal will likely stay constant during the simulation, desired force workflow and desired acceleration workflow would essentially be the same.

It is clear that the choice of workflow is arbitrary. For the rest of this thesis, desired velocity will be used as the workflow. We believe it is conceptually the easiest to think about. Each behavior says "I want to go in this direction at this speed". Additionally, for the rest of this thesis, velocity will denote direction and speed, where speed is a scalar value, the magnitude of a velocity.

3. Movers

This is the first chapter which examines one of the distict parts of the model we introduced in Chapter 2, the *Mover*. The function is responsible for updating the boid's current velocity a position, based on the result passed to it from the *Merger*.

First, assumptions about inputs, outputs and goals for the *Mover* will be given. Then, constraints which we believe are necessary for a realistic movement will be stated. This will be used to compare selected implementations, and evaluate them in regards to the constraints. Afterwards, a *Mover* which satisfies all the constraints will be introduced.

Movers are not necessarily specific to flocking simulations. They can be used to move any self propelled agent. This chapter will mainly focus on the linear acceleration and velocity. The last section will briefly discuss the rotations.

3.1 Frame Rate Dependency

Computer games can run at vastly different frame rates, depending on the specific hardware. This can be a cause of certain type of bugs. We will briefly explain how the issue usually arises, and what can be done to account for different frame rates to keep uniform experience across devices. This will help understanding where frame rate can create issues when implementing a flocking simulation, and how these issues can be prevented.

One simple example of a bug caused by not accounting for different frame rate was found in the title Dark Souls II. In the game, a weapon degraded by a constant amount every frame. This caused the weapon to degrade twice as fast when the game was updated from 30 fps to 60 fps [28].

This issue can be called frame rate dependency, since the gameplay is in some way tied to the frame rate. The mentioned bug in Dark Souls II could be fixed, if the degradation amount for each frame was multiplied by the time passed since the last frame Δt.

$$degradation = degradation + degradationSpeed * \Delta t$$

This implementation makes sure that no matter how many times this code is run in one second, after one second the degradation will always increase by *degradationSpeed*. If *degradationSpeed* was not multiplied by Δt, after one second had passed, degradation would increase by $degradationSpeed * fps$, where fps is the number of frames per second. Movement is something where this issue can arise as well. It will be necessary to correctly account for frame rate in the *Mover*.

3.2 Assumptions

Assumptions about the rest of the algorithm will be made, to ease the analysis. Let us assume that *Merger* passes a single desired velocity into the *Mover*, which then returns new boid's position and velocity. Assume these inputs and outputs.

The inputs are:

- $boid = \{\vec{p_c}, \vec{v_c}\}$ the boid's current position and velocity.
- $\vec{v_d}$ the boid's desired velocity.
- Δt the time that the last frame took to complete.

The outputs are:

- $\vec{p_n}$ the boid's new position.
- $\vec{v_n}$ the boid's new velocity.

We will only be concerned with how velocity is updated, from our research, other papers either do not describe how velocity $\vec{v_n}$ is used to update the current position, or they use the Euler Integration (see Equation 2.2 from Section 2.2). Lastly, for the sake of simplicity, vectors are assumed to be from $\mathbb{R}^2$. Generalization to $\mathbb{R}^3$ should be straightforward if needed.

The provided Figure 3.1 illustrates a sample *Mover* with our assumptions. It shows the current velocity $\vec{v_c}$, desired velocity $\vec{v_d}$ and how they are transformed to give a new velocity $\vec{v_n}$.

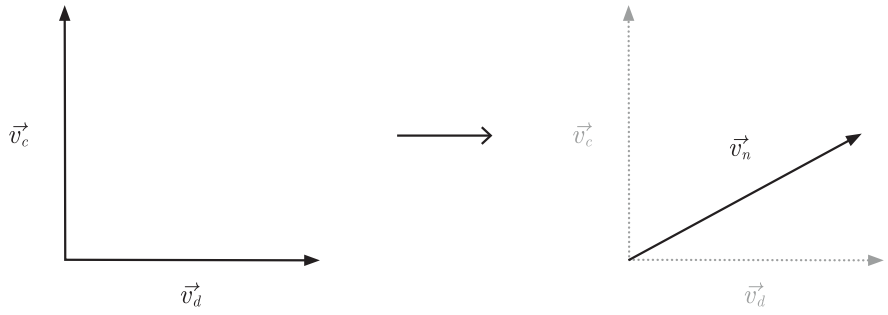

Figure 3.1: A sample illustration of a *Mover*.

3.3 Constraints

We have identified a set of constraints which we believe the *Mover* should satisfy. The constraints are:

1. *The movement must be frame rate independent.*

   The importance of frame rate independency was described previously in Section 3.1.

2. *The magnitude of a boid's acceleration must be bounded by a finite number.*

   This ensures that there is a bound on maximum acceleration, it would be harder to work with system without bounds. Real animals are also bounded by their physical abilities.

3. *The magnitude of a boid's velocity must be bounded by a finite number.*

   This ensures that there is a bound on maximum speed, the reasoning is the same as for the second constraint.

4. *Assuming a given desired velocity, the direction of boid's velocity will converge to the direction of its desired velocity in finite time.*

   This ensures that the boid will eventually walk in its desired direction. No guarantees on reaching the desired speed are given. Consider prey running away from a predator. It would desire to run away at speed larger than the predator, but that might not be within its capabilities.

5. *It must be possible to adjust the boid's physical abilities.*

   This contraint is more loosely defined, it intends to make sure that the game designers can adjust the physical abilities for different types of animals. It might be interesting, for example, to set up prey to have a smaller maximum speed, but higher maneuverability.

3.4 Analysis

Three different implementations will now be analyzed. These implementations were chosen as common examples of what can be found in texts concerned with flocking and steering behaviors.

3.4.1 Implementation 1 – Desired Velocity Assignment

The simplest implementation is to set the desired velocity to the new velocity directly. Formally this can be expressed as:

$$\boxed{\vec{v_n} = \vec{v_d}} \tag{3.1}$$

Figure 3.2 illustrates this idea. In it, $\vec{v_d}$ is assigned to be the new velocity $\vec{v_n}$ directly, ignoring the current velocity $\vec{v_c}$.

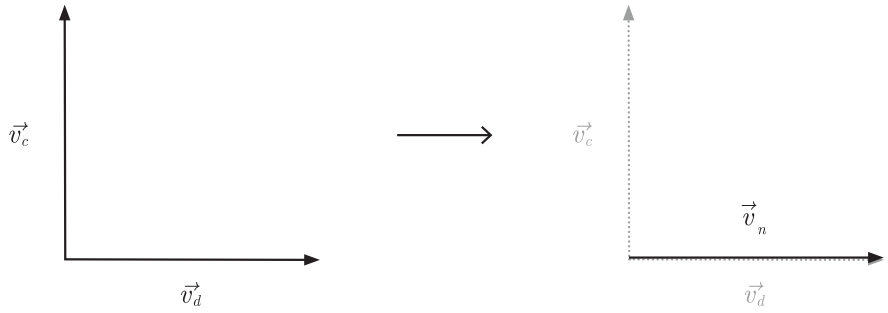

Figure 3.2: Direct assignment of $\vec{v_d}$ to $\vec{v_n}$.

This approach was used for example in a paper titled "Flocking behaviour of group movement in real strategy games" [29] by Fathy et al. The paper explores using flocking and other steering behaviors with A* to control units in an RTS[1] game. This approach was also by Alaliyat et al. [30] in a study comparing two approaches to find optimal values for weighting between behaviors. Lastly it was used in a game where flocking is the main mechanic developed at Chalmers University and University of Gothenburg [31]. In the game, the player controls a dog which herds sheep. Movement using the Equation 3.1 will now be analyzed through the constraints specified in Section 3.3.

Constraint 1

The constraint 1 is satisfied, as the velocity is always the same as the current desired velocity.

Constraint 2

The constraint 2 is not satisfied as it is possible to immediately go from any $\vec{v_c}$ into any $\vec{v_d}$.

[1] Real-time strategy

Constraint 3

The constraint 3 is not explicitly satisfied as there are no bounds on the magnitude of $\vec{v_d}$. This could be easily fixed by clamping the magnitude of $\vec{v_n}$ to some maximum speed.

Constraint 4

The constraint 4 is clearly satisfied.

Constraint 5

Regarding the constraint 5, the only physical ability which could be adjusted would be the maximum speed, as suggested.

3.4.2 Implementation 2 – Sum of Desired and Current Velocity

Second implementation we have commonly found is setting the new velocity to be the sum of current and desired velocity. This is shown in Figure 3.3, and formally can be expressed as:

$$\boxed{\vec{v_n} = \vec{v_c} + \vec{v_d}} \tag{3.2}$$

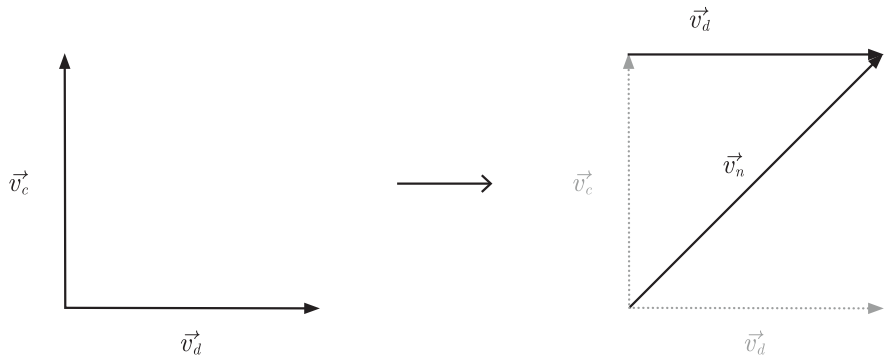

Figure 3.3: Illustration of Equation 3.3, $\vec{v_d}$ added to $\vec{v_c}$.

This implementation was found in a paper by Hartman and Beneš [25], which introduces a change of leadership behavior. The authors handle frame rate dependency when updating the position with Equation 2.2. However, the desired velocity is not multiplied by Δt when adding it to the current velocity. Further example can be found in paper by Mohit Sajwan et. al. [32], which discusses their modifications and enhancements of the Boids model. In their pseudocode, they do not account for Δt. Last example is an open source implementation of flocking, available on GitHub [33]. The author, Suboptimal Engineer, described his solution in his youtube video [34]. There, neither velocity nor position are updated frame rate independently.

For our analysis, we will add a variation which accounts from different frame rates. This means multiplying the desired velocity by Δt before adding it. This can be formally expressed as:

$$\boxed{\vec{v_n} = \vec{v_c} + \Delta t * \vec{v_d}} \tag{3.3}$$

This variation is shown in Figure 3.4, where $\Delta t * \vec{v_d}$ is summed with $\vec{v_c}$ to give new velocity $\vec{v_n}$. While none of the authors used the Equation 3.3, it is given here, because it accounts for frame rate dependency. Further explanation will be provided when analyzing the first constraint.

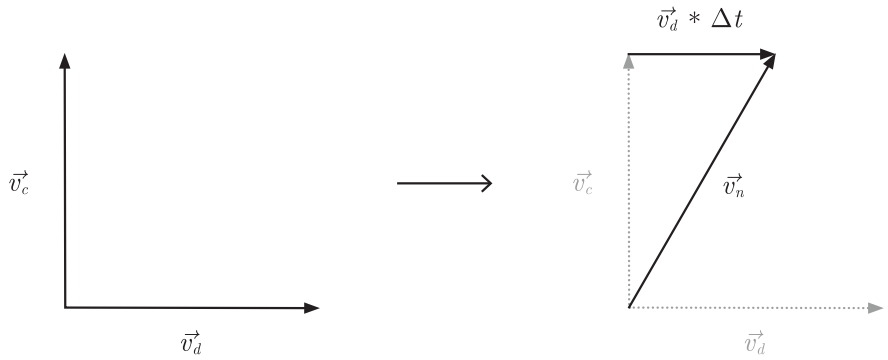

Figure 3.4: Illustration of Equation 3.3, $\Delta t * \vec{v_d}$ added to $\vec{v_c}$.

Constraint 1

Two alternatives were given, the Equations 3.2 and 3.3. From frame rate dependency perspective, it is a very similar situation as discussed earlier in section 3.1. It is easy to see, that in 1 second, using Equation 3.2, $\vec{v_d}$ will be added to the initial velocity fps times, so it is frame rate dependent.

Using Equation 3.3, $\vec{v_d}$ will be added to the initial velocity exactly once in 1 second. For any time t, $\vec{v_d}$ will be added t times, therefore it is frame rate independent.

Constraint 2

In this implementation, the boid is accelerating in direction of the desired velocity. Assuming the frame rate dependent variation (Equation 3.2), the boid's velocity is changed by $fps * \vec{v_d}$ in 1 second. Using the frame rate independent variation (Equation 3.3), the boid's velocity is changed by $\vec{v_d}$ in 1 second.

The constraint 2 is not satisfied in both cases, because $\vec{v_d}$ can be arbitrarily large, so the acceleration can be arbitrarily large. Assuming Equation 3.3, this can be resolved by clamping the magnitude of $\vec{v_d}$. Assuming the frame rate dependent Equation 3.2, even clamping the desired speed will not resolve the issue. The acceleration can still be arbitrarily large as fps gets larger.

Constraint 3

The constraint 3 is not satisfied, but in both cases, v_n could be clamped to some maximum magnitude. Unlike in Subsection 3.4.1 this will always need to be done

when using this approach. $\vec{v_d}$ is always added to $\vec{v_c}$, regardless of the current value of $\vec{v_c}$. The effect this will have is that after some point, speed will only keep increasing if $\vec{v_d}$ does not change. This can be seen in the Figure 3.5.

Constraint 4

The constraint 4 is not satisfied. It can be easily explained with an example illustrated in Figure 3.5. It considers $\vec{v_d} = (v_{dx}, 0)$ and $\vec{v_c} = (0, v_{cy})$. After k iterations of the Equation 3.2, $\vec{v_n} = (k * v_{dx}, v_{cy})$ (same argument would work for Equation 3.3). The y component of $\vec{v_n}$ would never change in this case. The boid would essentially be accelerating in the direction of $\vec{v_d}$ forever, while still keeping the initial velocity in the y direction. It is apparent from the figure, that the magnitude of $\vec{v_n}$ is getting larger, and its y component remains unchanged.

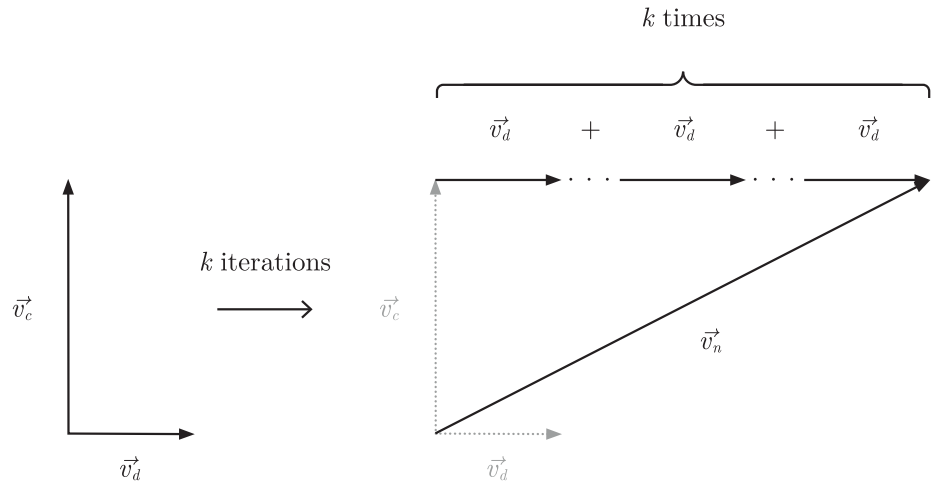

Figure 3.5: Visualization of k iterations of Equation 3.2, assuming constant $\vec{v_d}$. The new velocity $\vec{v_n}$ is shown after k iterations.

Constraint 5

Regarding the constraint 5, $\vec{v_d}$ could be multiplied by some constant to increase or decrease the magnitude of acceleration, and $\vec{v_n}$ could be clamped to some maximum magnitude.

Discussion

While it may seem that this implementation is obviously incorrect, since there are no guarantees on even matching the correct direction. It is still relatively often used. We speculate that the reason for why this issue is not uncovered is the following. Implementations will figure out the need to clamp the $\vec{v_n}$ in order to not have boids acceleration into ever larger speeds. In the previous example, for large k, the direction of the vector $\vec{v_n} = (k * v_{dx}, v_{cy})$ will be getting closer to $(1, 0)$, as the size of the y component will be negligible compared the the size of the x component, so the direction will seem to be approaching the direction of $\vec{v_d}$. Coupled with the fact that the $\vec{v_d}$ will be constantly changing throughout the simulation, this issue may go unnoticed.

This is an example of importance of the semantics of the result from *Merger*, discussed in Section 2.2. It is our belief that the problem boils down to treating a desired velocity as a desired acceleration in the *Mover*.

3.4.3 Implementation 3 – Acceleration Based

The following is perhaps the most common implementation. Reynolds uses this approach in his paper about steering behaviors [16]. A close variation of it is also used in his OpenSteer library [23]. The idea is to use the difference of desired velocity $\vec{v_d}$ and current velocity $\vec{v_c}$ to determine an acceleration $\vec{a}$. This is then added to the current velocity $\vec{v_c}$ to determine the new velocity $\vec{v_n}$. Formally this can be expressed as:

$$\boxed{\vec{v_n} = trunc(\vec{v_c} + \vec{a} * \Delta t, maxSpeed)} \tag{3.4}$$

$$\vec{a} = trunc(\vec{v_d} - \vec{v_c}, maxAcc)$$

where $maxSpeed$ is the maximum speed, $maxAcc$ is the maximum size of acceleration. The idea is illustrated in Figure 3.6. In it, the acceleration $\vec{a}$, determined as difference of $\vec{v_d}$ and $\vec{v_c}$, is multiplied by Δt and added to the current velocity $\vec{v_c}$.

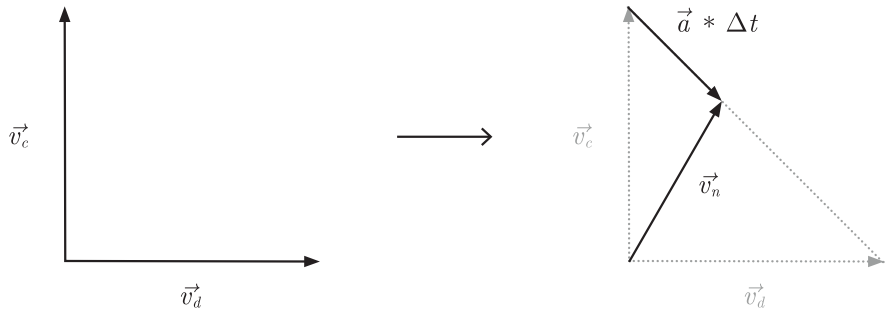

Figure 3.6: Desired acceleration $\vec{a}$ multiplied by Δt and added to $\vec{v_c}$.

Note that in the paper, Reynolds does not specify the multiplication by Δt, while his sample implementation [23] does multiply by Δt. Only the implementation that multiplies by Δt will be considered, for the sake of simplicity. Furthermore, Reynolds uses force as workflow, so in his implementation, $\vec{a}$ is divided by mass first. We omit this for simplicity, as we have shown how to convert between workflows in Section 2.2.1. Also, in Reynolds' description [16], $\vec{a}$ is calculated as part of the *Behaviors*. Here, the responsibility to calculate $\vec{a}$ is passed onto the *Mover*, since we are focused on analyzing how a single desired velocity $\vec{v_d}$ would affect the movement.

An implementation using the Equation 3.4 tries to account for gradual acceleration from current velocity $\vec{v_c}$ toward the desired velocity $\vec{v_d}$. We noticed that usually, implementations based on Reynolds' original papers [7, 16] share the common idea of calculating the acceleration from $\vec{v_c}$ and $\vec{v_d}$ and adding it to the current velocity. Movement based on Equation 3.4 will now be analyzed through the specified constraints.

Constraint 1

The constraint 1 is surprisingly not satisfied in this simple form. It may seems that frame rate is accounted for when velocity is updated, but consider the following scenario.

$$\vec{v_c} = (0, 1), \vec{v_d} = (1, 0), \Delta t = 1$$

In one unit of time, there will be only be one iteration which will result in:

$$\vec{v_n} = (0, 1) + ((1, 0) - (0, 1)) * 1 = (1, 0)$$

.

Figure 3.7 illustrates this case. Assuming $\Delta t = 1$, $\vec{v_d}$ is set directly to $\vec{v_n}$, same as in the first implementation in Subsection 3.4.1.

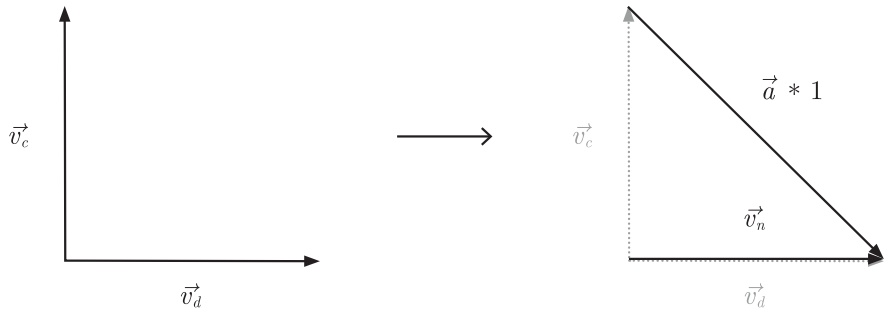

Figure 3.7: Using the Reynolds' movement implementation, for $\vec{v_c} = (0, 1)$, $\vec{v_d} = (1, 0)$, $\Delta t = 1$.

Normally, Δt will likely be smaller than 1, for simplicity assume $\Delta t = 0.5$, and the same scenario. Now, there would be two iterations within one second. Both iterations are visualized in Figures 3.8 and 3.9. It is apparent that in this case, after one second had passed, $\vec{v_n}$ will not be equal to $\vec{v_d}$. The first iteration gives:

$$\vec{v_n} = (0, 1) + ((1, 0) - (0, 1)) * 0.5 = (0.5, 0.5)$$

Here, the new $\vec{v_n}$ is essentially halfway from $\vec{v_c}$ to $\vec{v_d}$, as shown in Figure 3.8.

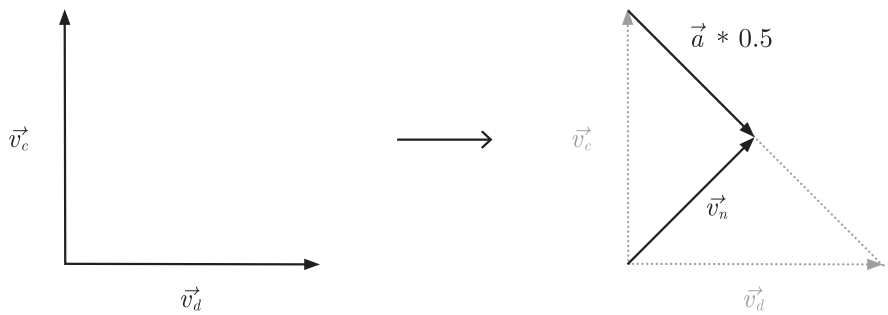

Figure 3.8: Using the Reynolds' movement implementation, for $\vec{v_c} = (0, 1)$, $\vec{v_d} = (1, 0)$, $\Delta t = 0.5$

Second iteration, depicted in Figure 3.9 gives:

$$\vec{v_n} = (0.5, 0.5) + ((0, 1) - (0.5, 0.5)) * 0.5 = (0.75, 0.25)$$

.

Here the new $\vec{v_n}$ is again halfway from $\vec{v_c}$ to $\vec{v_d}$, as shown in Figure 3.9. This means that in the second iteration, the acceleration is only half of what it was in the first one.

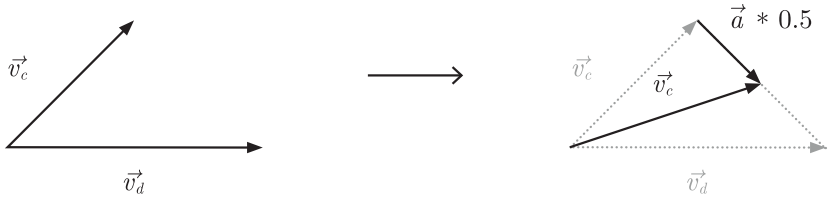

Figure 3.9: Using the Reynolds' movement implementation, for $\vec{v_c} = (0.5, 0.5)$, $\vec{v_d} = (1, 0)$, $\Delta t = 0.5$.

It is clear from this example, that the way in which this algorithm updates the velocity is not frame rate independent – the first constraint is not satisfied. The problem is caused by the fact, that the magnitude of the acceleration depends on the magnitude of $\vec{v_d}$ - $\vec{v_c}$. It is inaccurate in physical sense to calculate the acceleration as $\vec{v_d}$ - $\vec{v_c}$. Both quantities are velocities, so the result is a change in velocity, but not an acceleration, which is a change in velocity over time.

To calculate an acceleration in physical sense, the Equation 2.4 can be used.

$$\vec{a} = \frac{\vec{v_d} - \vec{v_c}}{\Delta t}$$

This was used for example by Colas et al. [35] in their paper, which describes using steering behaviors and flow fields to model crowd simulations. The acceleration is then multiplied by Δt and added to the current velocity:

$$\vec{v_n} = \vec{v_c} + \vec{a} * \Delta t$$

The problem with using this approach, is that it actually equivalent to the first implementation from Subsection 3.4.1, where the $\vec{v_d}$ is assigned to $\vec{v_n}$ directly. The following shows that these two approaches are equivalent:

$$\vec{v_c} + \vec{a} * \Delta t = \vec{v_c} + \frac{\vec{v_d} - \vec{v_c}}{\cancel{\Delta t}} * \cancel{\Delta t} = \cancel{\vec{v_c}} + \vec{v_d} - \cancel{\vec{v_c}} = \vec{v_d}$$

While this would be frame rate independent, it does not result in gradual acceleration from $\vec{v_c}$ to $\vec{v_d}$ as intended.

Constraint 2

The constraint 2 is satisfied, but only because the magnitude of the acceleration is clamped to some maximum magnitude. If it was not clamped, then the size of the acceleration can be arbitrarily large, since it depends on the size of $\vec{v_d}$ - $\vec{v_c}$, which can be arbitrarily large.

Constraint 3

The constraint 3 is satisfied, but again only because $\vec{v_n}$ is clamped to a maximum speed.

Constraint 4

Regarding the constraint 4, direction of $\vec{v_n}$ will be approaching the direction of $\vec{v_d}$ over time. The algorithm essentially in every step modifies the velocity to be Δt of the way from $\vec{v_c}$ to $\vec{v_d}$, so theoretically, in every step, $\vec{v_c}$ is getting closer to $\vec{v_d}$. However, it never fully reaches it (unless $\vec{v_d} = \vec{v_c}$ or $\Delta t = 1$). However, in practical applications, when testing at around 60 fps, we observed that $\vec{v_c}$ generally converges close enough to $\vec{v_d}$ quickly, but it will always be dependent on the actual frame rate.

One edge case is when $\Delta t > 1$, where we would essentially "overshoot" the desired velocity. Computer game would generally not need to worry about $\Delta t > 1$, as that would not be interactive frame rates.

Constraint 5

Regarding the constraint 5, the rate of acceleration can be adjusted by multiplying $\vec{a}$ with some constant. $\vec{v_n}$ and $\vec{a}$ can also be limited by maximum magnitudes.

3.5 Our implementation – Constant Acceleration

Our implementation is based on the Reynolds' implementation described earlier in Subsection 3.4.3. However, it solves the problem that in the aforementioned implementation, the magnitude of acceleration is proportional to the size of $\vec{v_d} - \vec{v_c}$. Here, the idea is to scale the desired acceleration to a constant length, which yields a constant rate of acceleration. The following equations describe this idea concretely, and an illustration is later shown in Figure 3.10.

$$\boxed{\vec{v_n} = \vec{v_c} + \vec{a_{dir}} * min(A * \Delta t, |\vec{v_{diff}}|)} \tag{3.5}$$

$$\vec{a_{dir}} = \frac{\vec{v_{diff}}}{|\vec{v_{diff}}|}$$

$$\vec{v_{diff}} = \vec{v_d} - \vec{v_c}$$

where A is a constant acceleration which can be adjusted for a specific animal, $\vec{a_{dir}}$ is a normalized direction of the desired acceleration and $\vec{v_{diff}}$ is the difference between $\vec{v_d}$ and $\vec{v_c}$. The main variables can be seen in Figure 3.10. Note that the length of the acceleration added to $\vec{v_c}$ is $A * \Delta t$, resulting a constant acceleration rate A, while accounting for frame rate independency.

Also note the usage of *min* function in Equation 3.5 (not apparent from Figure 3.10). It makes sure we do not overshoot $\vec{v_d}$ in a given frame.

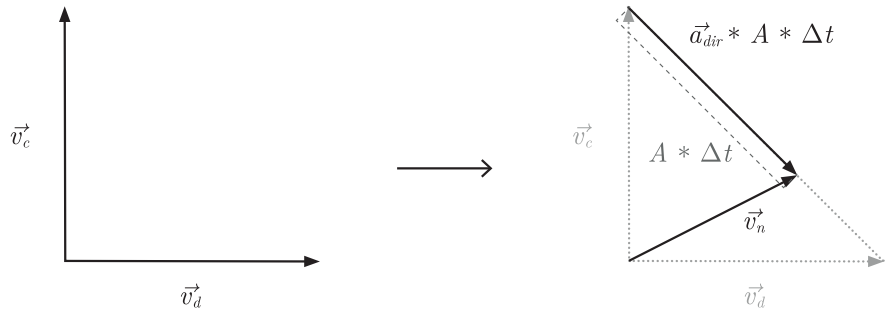

Figure 3.10: Constant and frame rate independent acceleration, the size of acceleration in a given frame is $A * \Delta t$.

Constraint 1

Constraint 1 is satisfied, velocity is updated in frame rate independent way, because the magnitude of acceleration is always some constant A.

Constraint 2

Constraint 2 is satisfied, the magnitude of acceleration is always a constant A.

Constraint 3

Constraint 3 can be satisfied, if size of $\vec{v_d}$ or $\vec{v_n}$ is clamped.

Constraint 4

Constraint 4 is satisfied. For some initial $\vec{v_c}$ and $\vec{v_d}$, the desired velocity $\vec{v_d}$ will be reached in $\frac{|\vec{v_d}-\vec{v_c}|}{A}$ units of time.

Constraint 5

Regarding constraint 5, the rate of acceleration A can be set, and maximum speed could easily be introduced.

3.5.1 Improvements

The main issue with our proposed approach is that the animal would always have a constant acceleration. The movement could be made more dynamic by passing a scalar desired acceleration magnitude a_d to the *Mover* and use it instead of constant A. This would require that the behaviors provide a_d along with $\vec{v_d}$. This can correspond to the idea that sometimes, an animal might want to accelerate as much as possible away from some threat. Other times, accelerating to the desired speed at a slower rate may be more realistic, when not in an immediate danger.

Another improvement was identified already by Reynolds in his paper about steering behaviors [16]. Reynolds proposes that acceleration be split into two components, one in the direction of travel, and the direction perpendicular to it. Let us refer to them parallel component and lateral component respectively.

Then, the lateral component can then be thought of as changing direction, and parallel component as changing speed. This would provide for more control over the animal's abilities, since the components can be scaled by different constant. This could be used to create a type of animal, which can increase its speed quickly, but perhaps cannot make sharp maneuvers that well. This corresponds to the predator-prey interaction discussed in Section 3.3, constraint 5.

Vectors can be split into components by vector projection. Let $proj$ be a vector operation defined as:

$$proj(\vec{v_1}, \vec{v_2}) = (\vec{v_1} \cdot \vec{v_2}) * \vec{v_2}$$

where $\vec{v_2}$ is assumed to be a normalized direction that $\vec{v_1}$ is being projected on. The operation $proj$ gives the projected component of $\vec{v_1}$ onto $\vec{v_2}$. The rest of v_1 is called the rejection, and can be calculated with:

$$rej(\vec{v_1}, \vec{v_2}) = \vec{v_1} - proj(\vec{v_1}, \vec{v_2}) \tag{3.6}$$

The Figure 3.11 illustrates decomposing vector $\vec{v_1}$ into two components, one component in the direction of $\vec{v_2}$, the other perpendicular to it. Note that the sum of $rej(\vec{v_1}, \vec{v_2})$ and $proj(\vec{v_1}, \vec{v_2})$ is equal to the original $\vec{v_1}$.

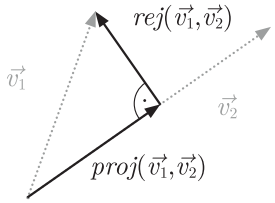

Figure 3.11: $\vec{v_1}$ decomposed into two components, the projection of $\vec{v_1}$ onto $\vec{v_2}$ and the rejection of v_1 from $\vec{v_2}$.

In the previous model, limiting the animals velocity by clamping it to some maximum magnitude was proposed. Also, the acceleration is always constant. In real world it is harder to accelerate, the faster we are moving. We can model this by simulating air drag, a force in the opposite direction of the velocity. In physics, drag force is calculated as a force in the opposite direction of velocity, with the size of F_d, where F_d is given by the following equation:

$$F_d = \frac{1}{2} * \rho * v^2 * C_d * A$$

Where ρ is density of the fluid, v the travelling speed, C_d a drag coefficient, and A the cross sectional area [36]. We can use the Newton's second law of motion to determine the induced acceleration by dividing the resulting force with mass of the object. For this purpose, all the values except for the speed are assumed to be constant. Therefore a single value $D = \frac{1}{2} * \frac{1}{m} * C_d * A * \rho$ can be used as strength of the drag force. Then the acceleration induced by the drag force is given by:

$$\vec{a_{drag}} = -\frac{\vec{v_c}}{|\vec{v_c}|} * |\vec{v_c}|^2 * D * \Delta t \tag{3.7}$$

Figure 3.12 illustrates the acceleration induced by the drag force. The force is in the opposite direction of the current velocity $\vec{v_c}$, and its size is proportional to the square of the speed $|\vec{v_c}|^2$ and the drag strength D.

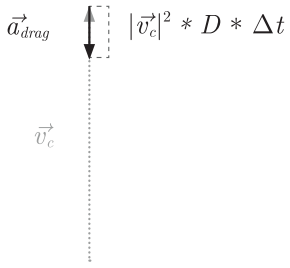

Figure 3.12: Accleration $\vec{a_{drag}}$ induced by drag force on a body traveling at velocity $\vec{v_c}$.

Now, with equations for projection, rejection and drag, we can introduce an improved version of our implementation. Following is the *Mover* with the discussed improvements:

$$\boxed{\vec{v_n} = \vec{v_c} + \vec{a_{proj}} + \vec{a_{rej}} + \vec{a_{drag}}}$$

As usual, this is illustrated in Figure 3.13, where the projected acceleration $\vec{a_{proj}}$, the rejected acceleration $\vec{a_{rej}}$ and acceleration induced by drag $\vec{a_{drag}}$ are added to the current velocity $\vec{v_c}$, to determine the new velocity $\vec{v_n}$.

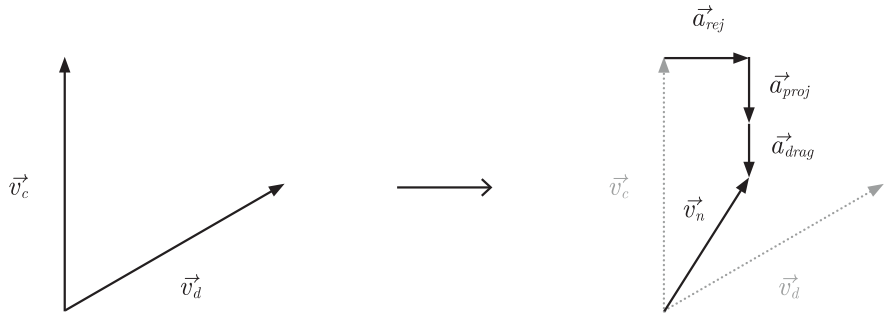

Figure 3.13: All three acceleration components, $\vec{a_{proj}}$, $\vec{a_{rej}}$ and $\vec{a_{drag}}$ are added to the current velocity $\vec{v_c}$, to determine the new velocity $\vec{v_n}$.

The acceleration induced by drag $\vec{a_{drag}}$ was shown in Figure 3.12 and is given by the Equation 3.7. The calculation of projected and rejected accelerations $\vec{a_{proj}}$ and $\vec{a_{rej}}$ is illustrated in Figure 3.14. The figure shows decomposition of desired acceleration into lateral and paralallel components, which are scaled to be of size of $A_p\Delta t$ and $A_r\Delta t$ respectively. To achieve this, first, desired velocity change $\vec{v_{diff}}$ is projected onto the direction of the current velocity. This separates the parallel component of desired velocity change $\vec{v_{diffProj}}$. This is normalized to determine the projected direction of the acceleration $\vec{a_{dirProj}}$. Finally the acceleration in the parallel direction for the current frame $\vec{a_{proj}}$ is calculated, by multiplying $\vec{a_{dirProj}}$ by rate of acceleration A_p in the parallel direction and Δt, to account for frame rate dependency. Note, that while not apparent from the figure, *min* is again

used to make sure the desired velocity $\vec{v_d}$ is not overshot. The calculation of $\vec{a_{proj}}$ and $\vec{a_{rej}}$ can be expressed with the following equations:

$$\boxed{\vec{a_{proj}} = \vec{a_{dirProj}} * min(A_p * \Delta t, |\vec{v_{diffProj}}|)}$$

$$\vec{a_{dirProj}} = \frac{\vec{v_{diffProj}}}{|\vec{v_{diffProj}}|}$$

$$\vec{v_{diffProj}} = proj(\vec{v_{diff}}, \frac{\vec{v_c}}{|\vec{v_c}|})$$

$$\vec{v_{diff}} = \vec{v_d} - \vec{v_c}$$

$$\boxed{\vec{a_{rej}} = \vec{a_{dirRej}} * min(A_r * \Delta t, |\vec{v_{diffRej}}|)}$$

$$\vec{a_{dirRej}} = \frac{\vec{v_{diffRej}}}{|\vec{v_{diffRej}}|}$$

$$\vec{v_{diffRej}} = \vec{v_{diff}} - \vec{v_{diffProj}}$$

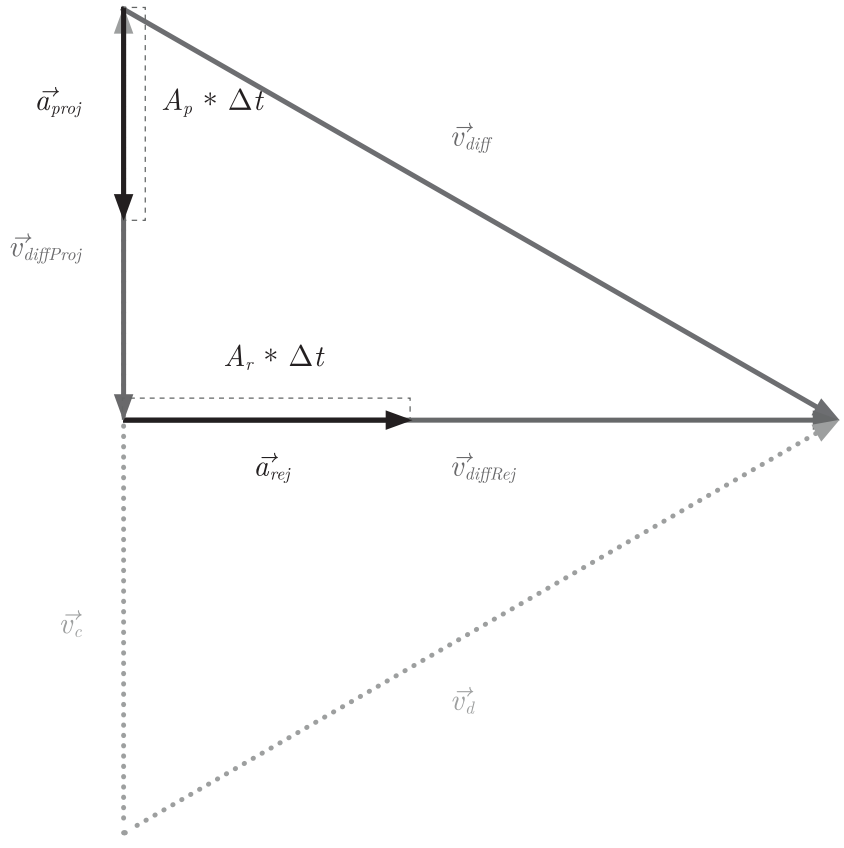

Figure 3.14: Decomposition of desired acceleration into lateral $\vec{a_{proj}}$ and parallel $\vec{a_{rej}}$ components, assuming desired velocity $\vec{v_d}$ and current velocity $\vec{v_c}$.

The described implementation accounts for drag, which can limit the maximum velocity of the animal more naturally. It makes it harder for the animal to keep accelerating in the direction of its velocity as its speed increases. It

also allows adjusting of the maneuvering and forward acceleration abilities of the animals, using A_p and A_r.

There is however still one issue, the implementation does not distinguish between acceleration which increases speed, and that which decreases speed. A real animal would likely be able to decrease its speed faster than increase its speed. This case could be handled by deciding whether $\vec{v_{diffProj}}$ is in the direction of $\vec{v_c}$ or not. Then, a different scalar for the acceleration can be chosen. This could be implemented as:

$$A_p = \begin{cases} A_{speedUp} & \text{if } \frac{\vec{v_c}}{|\vec{v_c}|} \cdot \frac{\vec{v_{diffProj}}}{|\vec{v_{diffProj}}|} = 1 \\ A_{slowDown} & \text{otherwise} \end{cases}$$

3.6 Rotation

So far only linear movement has been discussed. However, the rotation of the animal should also be handled. While for example birds always have to be aligned with the direction of their velocity, four-legged animals can also move sideways or backwards. One approach to handle this can be to return a desired rotation from each of the behaviors. This approach is used in Godot Engine's steering library [37]. While this is an interesting addition, and could be explored further, this thesis will only consider the common approach where the animal's rotation is aligned with the direction of its current velocity. Usually, animals naturally move in the direction they are facing, and considering the fact that most games would likely not need this level realism or control, this feature is not too important.

3.6.1 Approach 1

A simple approach is to set the rotation to align exactly with the velocity every frame. If the velocity changes rapidly throughout the simulation, this can cause jittering.

3.6.2 Approach 2

Second approach is to interpolate between the current rotation and the rotation where the animal would be aligned with its velocity. In 2D, there is only one axis to rotate around, therefore a simple linear interpolation from one angle to another can be used. Interpolation between two rotations in 3D is a more difficult problem. The issues with interpolating rotations in 3D were described well in a paper titled "Animating Rotation with Quaternion Curves" [38] by Shoemaker (1985). In the paper, he proposes using quaternions to represent rotations, and interpolating between them using a function called *slerp*. We suggest using quaternions together with *slerp* to interpolate between the current boid's rotation and the rotation where the boid would be aligned with its velocity. This worked well in our implementations. For details about *slerp*, refer to the original paper [38].

4. Mergers

As outlined in Chapter 2, *Mergers* turn k results $\{r_0 \dots r_k\}$ from k behaviors into one final result, which is then fed into the *Mover*. The *Merger* is responsible for blending and arbitering between the behaviors' results. This chapter covers *Mergers* in a similar fashion as previous chapter covered the *Movers*. First, our constraints and assumptions for *Mergers* are introduced. This is used to analyze some commonly found implementations. Finally an implementation satisfying our constraints is introduced.

4.1 Assumptions

This section introduces assumptions about the rest of the algorithm which we make. Let us again assume desired velocity workflow to interpret the behavior results and merge results. All vectors here are from $\mathbb{R}^2$, generalization to $\mathbb{R}^3$ should be straightforward if needed. Also assume these inputs and outputs:

- $\{r_0 \dots r_k\}$ in general a list of k results from behaviors, usually in case of desired velocities $\{\vec{v_{d0}} \dots \vec{v_{dj}} \dots \vec{v_{dk}}\}$, or in the case of desired accelerations $\{\vec{a_{d0}} \dots \vec{a_{dj}} \dots \vec{a_{dk}}\}$, but more information could be passed in.

The outputs are:

- r a single result, in case of desired velocities $\vec{v_d}$, in case of desired accelerations $\vec{a_d}$

When possible desired velocities are used, for both input results and output result. Some implementations discussed assume results to be desired accelerations rather than desired velocities. In the case where desired accelerations are used, the results could be converted as described in Section 2.2.1.

4.2 Constraints

We have identified a set of constraints which we believe the *Merger* should satisfy. The constraints are:

1. *It should be possible for one behavior to take full lead.*

   This ensures that cases such as running straight into a brick wall can be handled. In this case, a good *Merger* should have some way of completely filtering out behaviors, which are less important in that moment.

2. *A small change in inputs should only cause a small change in the output.*

   This ensures that there are no cases where the desired velocity is vastly different from one frame to another. This intends to eliminate cases where the boid could jitter back and forth between two very different results.

3. *Given only a single behavior result, the behavior's desired velocity (or acceleration) will be returned.*

   This ensures that a merge implementation is sensible, and does not do anything unexpected to a behavior's results. In the case of a single behavior, the *Merger* should only pass on the behavior's result unchanged.

4. *It should be possible to adjust the importance of behaviors relative to each other.*

   This constraints is aimed at game designers, who will want to adjust the flocking simulation to their specific needs.

4.3 Analysis

Three different implementations of *Mergers* will now be analyzed. These implementations were chosen as common examples of what we have found in sources concerned with flocking and steering behaviors.

4.3.1 Implementation 1 – Priority First Non-Zero

The first and perhaps the simplest implementation was proposed in Reynolds' paper about steering behaviors [16]. Reynolds adds a priority p_j to each behavior's desired acceleration $\vec{a_{dj}}$. In this case, the behavior results could be defined by the following tuple:

$$r_j = (\vec{a_{dj}}, p_j)$$

The idea is to sort the results based on their priorities, and return the first non-zero acceleration. No equations or pseudocode is given. An implementation should be straightforward from the description.

Constraint 1

The first constraint is satisfied. The single non-zero highest priority behavior takes lead at any given moment.

Constraint 2

The second constraint is not satisfied. If desired acceleration for a given behavior changes slightly, and is now a zero vector, a result for a different behavior is returned.

Constraint 3

The third constraint is satisfied. Given only one result, it will be returned with no changes.

Constraint 4

The fourth constraint is satisfied to some extend. Priorities allow the possibility to adjust the relative importance of results.

4.3.2 Implementation 2 – Weigthed Sum

In his original paper about Flocking [7], Reynolds suggested using a weighted sum to combine results of behaviors, in his case desired accelerations. From what we have observed in other papers and implementations, this is the most common approach. It could be expressed as:

$$\boxed{\vec{a_d} = \sum_{j=0}^{k} \vec{a_{dj}} * w_j}$$

where w_j is the jth weight. The variation for desired velocity workflow would be similarly be.

$$\boxed{\vec{v_d} = \sum_{j=0}^{k} \vec{v_{dj}} * w_j} \tag{4.1}$$

The Equation 4.1 using the desired velocity workflow is illustrated in the Figure 4.1. It shows a weighted sum of three behavior results $\{\vec{v_{d0}}, \vec{v_{d1}}, \vec{v_{d2}}\}$, which gives the final desired velocity $\vec{v_d}$. The implementation using desired velocity will be discussed in this subsection along with its problems. Using desired accelerations would create similar problems.

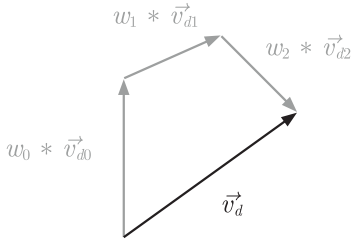

Figure 4.1: Weighted sum of three desired velocities, determining the final desired velocity $\vec{v_d}$.

Constraint 1

The first constraint is not satisfied. The final $\vec{v_d}$ is always a blend of all $\vec{v_{dj}}$.

Constraint 2

The second constraint is satisfied. A small change in any $\vec{v_{dj}}$ does not cause large change in the final $\vec{v_d}$. However, it is interesting to note that the direction of $\vec{v_d}$ can change dramatically from one frame to another. For example a small change in one $\vec{v_{dj}}$ could bring the final desired velocity from $(0, \epsilon)$ to $(0, -\epsilon)$.

Constraint 3

The third constraint is not satisfied. This is because the v_{dj} is multiplied by its weight, which scales its length. Therefore the result is not simply passed on when only one behavior is active.

Constraint 4

The fourth constraint is satisfied. The weights allow the possibility to adjust relative importance of the results.

Discussion

This implementation is very popular, and as Reynolds notes, it is his preferred implementation [16]. Despite its success, the above analysis suggests there are several issues with it. The main issue in uncovered by the third constraint. The problem is that once $\vec{v_{dj}}$ is multiplied by w_j, the information about the intended speed is lost[1]. The weights are, however, needed to satisfy the fourth constraint.

Another consequence of using weighted sum is shown in Figure 4.2. Assume k desired velocities all equal to some vector $\vec{v}$, and $w_j = 1$ for all j. The final $\vec{v_d}$ would in this case be $k * \vec{v}$. This makes little sense conceptually as all behaviors wish to go at the same speed in the same direction, but the final $\vec{v_d}$ is k times larger[2] as illustrated on the right side of the figure.

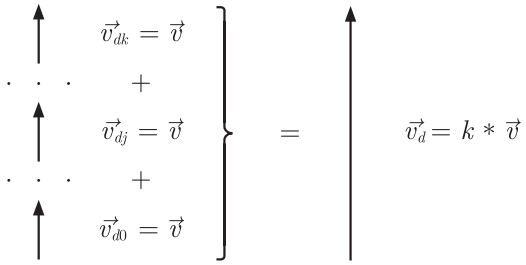

Figure 4.2: Desired velocity $\vec{v_d}$ has k times larger magnitude than what each separate behavior desires.

This specific case can be resolved by dividing the sum by k, the total number of results:

$$\vec{v_d} = \frac{\sum_{j=0}^{k} \vec{v_{dj}}}{k}$$

In a more general sense, to account for the weights, a weighted mean could be used. This is very similar to the original implementation, and it is a quite common approach as well.

$$\vec{v_d} = \frac{\sum_{j=0}^{k} \vec{v_{dj}} * w_j}{\sum_{j=0}^{k} w_j}$$

This change also resolves the issue posed by the third constraint. Given a single behavior, its result is returned unmodified. Using weighted mean might however still have another undesirable effect. Consider that all behaviors except for one find no neighbors. These behavior should likely have no "opinion" on the desired velocity. What value should behaviors return to signify this? If they all

[1]The problem is still present with desired acceleration workflow, there the information about the size of the desired acceleration is lost.

[2]The same argument could be made when working desired acceleration workflow.

return a zero vector, the final $\vec{v_d}$ will be smaller than what the relevant behavior desires (because it will be divided by sum of the weights). Some special value signifying "no opinion" could be used, and these results could be filtered out. Having a special value for "no opinion" is not a good solution, as that could create discontinuities in the desired velocity, which would violate the second constraint. To arrive at velocity desired by the only active behavior, all other behaviors would need to return the same desired velocity as the only active behavior. The behaviors cannot know this in advance. Also it is not clear how this could be generalized to more than one active behavior. Therefore, some issues still remain even with the weighted mean approach.

We believe that the main issue with using a weighted sum or a weighted mean lies in the semantics. Once the desired velocity or acceleration is multiplied by its weight, the information about the desired speed or acceleration rate is lost. To give a concrete example, a desired velocity $\vec{v_{dj}} = (0, 1)$ with weight $w_j = 10$ will have the same effect as $\vec{v_{dj}}\prime = (0, 10)$ with weight $w_j\prime = 1$, eventhough the former has a much higher weight, which should imply it has greater importance. The issue that results with higher speed have higher influence could be resolved by normalizing the desired velocities first. Then a weighted sum of desired directions could be used. However, the ability of a behavior to match a specific speed would be lost. Having the possibility to speed up in certain situations is important when considering, for example, a predator-prey interaction.

4.3.3 Implementation 3 – Prioritized Allocation

The third sample approach was first also described by Reynolds in his paper about flocking [7], there he calls it "prioritized acceleration allocation". Later in his paper about steering behaviors [16], he notes that over many of his implementations, he found the simpler approach from Subsection 4.3.2 to be sufficient. Reynolds described prioritized allocation in the following way:

"The acceleration requests are considered in priority order and added into an accumulator. The magnitude of each request is measured and added into another accumulator. This process continues until the sum of the accumulated magnitudes gets larger than the maximum acceleration value, which is a parameter of each boid. The last acceleration request is trimmed back to compensate for the excess of accumulated magnitude." [7]

We illustrate this approach in Figure 4.3. It shows how four desired accelerations $\vec{a_{d0}}, \vec{a_{d1}}, \vec{a_{d2}}, \vec{a_{d3}}$, sorted in this order by their priorities, are used to determine final desired acceleration $\vec{a_d}$. The first two accelerations $\vec{a_{d0}}, \vec{a_{d1}}$ are summed fully, then only a part of the third acceleration $\vec{a_{d2}}$ is added, and $\vec{a_{d3}}$ remains unused, as the maximum sum of magnitudes of accelerations is reached. The Algorithm 3 shows a pseudocode of this approach.

While using weights to multiply the desired accelerations is not explicitly specified in the figure or the algorithm, it could easily be done, same as in the previous implementation in Subsection 4.3.2. The weights allow a better control of relative influence between behaviors. For simplicity, we assume the desired accelerations passed into the *Merger* to already be premultiplied by their weights. This brings in the issue discussed in the previous Subsection 4.3.2, namely that information about the acceleration's magnitude is lost.

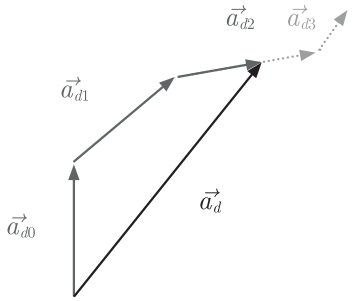

Figure 4.3: Prioritized acceleration allocation, the final desired acceleration $\vec{a_d}$ is the sum of full $\vec{a_{d0}}$ and $\vec{a_{d1}}$, only a part of $\vec{a_{d2}}$ and no part of $\vec{a_{d3}}$

Algorithm 3: Prioritized acceleration allocation

Data: List of results *results* (one result is a desired acceleration and its priority), maximum sum acceleration *maxSumAcceleration*

Result: Desired acceleration

```
// Sort in descending order based on priorities
sort(results);
currentSum ← 0;
accelerationSum ← 0⃗;
for i ← 0 to length(results) − 1 do
    accelerationSize ← ‖results[i].acceleration‖;
    currentSum ← currentSum + accelerationSize;
    if currentSum + accelerationSize >= maxSumAcceleration then
        accDir ← results[i].acceleration/accelerationSize;
        leftOver ← (maxSumAcceleration − currentSum);
        accelerationSum ← accelerationSum + accDir ∗ leftOver;
        break;
    accelerationSum ← accelerationSum + results[i].acceleration;
return accelerationSum;
```

Constraint 1

The first constraint is satisfied. It is possible for a behavior to take the lead by returning the highest priority and an acceleration equal to or larger than $maxSumAcceleration$.

Constraint 2

The second constraint is satisfied. For example, if a small change in one of the $\vec{a_{di}}$ occurs, it can allow accumulating one more result before $maxSumAcceleration$ is reached, but only the small leftover part will be added.

Constraint 3

The third constraint is satisfied, if the desired accelerations are not weighted. That would cause the same issue as the previous implementation in Subsection 4.3.2.

Constraint 4

The fourth constraint can be satisfied, if weights are used, same as in Subsection 4.3.2. Moreover the priorities allow the possibility to adjust the relative importance of results even further.

Discussion

While Reynolds notes that he found that using prioritized acceleration allocation is not necessary [16], we found it to be useful in our implementations. In some cases, it brought us an improvement over the weighted sum from implementation described in Subsection 4.3.2. The following example will illustrate one such case.

Consider the weighted sum is used, and that we are working with desired velocities. Same argument could be made for desired accelerations. Also consider that the separation behavior is the most important, and thus has the largest weight. We would want to guarantee that the separation always has some way to "overpower" other behaviors to prevent collisions. Further consider there are many other behaviors, but with lesser weights. The Figure 4.4 illustrates a possible edge case where all the other results ($\vec{v_{d0}}$, $\vec{v_{d1}}$, $\vec{v_{d2}}$) align such that they are opposite to the separation ($\vec{v_{ds}}$), thus "overpowering" it. This is shown by the final result $\vec{v_d}$ on the right. This happens even though individually, $\vec{v_{ds}} * w_s$ is by far the largest term in the weighted sum. In this case, the separation behavior would not stop a collision.

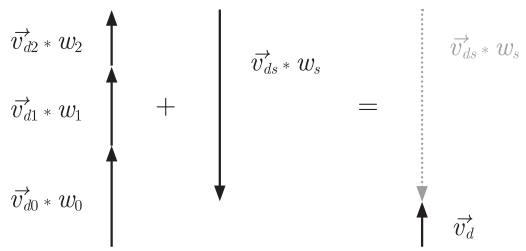

Figure 4.4: Weighted sum edge case. On the left, weighted sum of $\vec{v_{d0}} * w_0$, $\vec{v_{d1}} * w_1$, $\vec{v_{d2}} * w_2$ and a separation term $\vec{v_{ds}} * w_s$. On the right, the resulting final desired velocity $\vec{v_d}$.

Encountering this edge case could lead to increasing the weight of separation further, which we often found the need to do in our experiments. However, this would mean that separation has unnecessarily high influence in the common case, which can lead to erratic movement. Due to the chaotic nature of group behaviors, in common case, the results are randomly distributed, as shown in Figure 4.5. In that case, other behaviors mostly cancel each other out, so the separation does not need such a high weight.

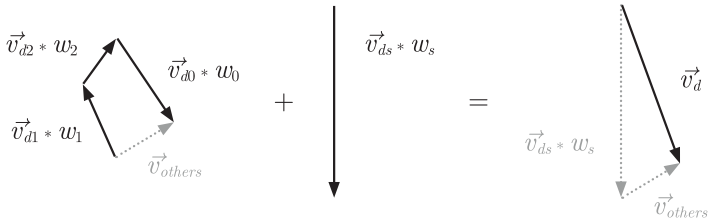

Figure 4.5: Weighted sum common case. On the left, weighted sum of $\vec{v_{d0}} * w_0$, $\vec{v_{d1}} * w_1$, $\vec{v_{d2}} * w_2$ and a separation term $\vec{v_{ds}} * w_s$. On the right, the resulting final desired velocity $\vec{v_d}$.

The issue of needing weights too large for important behaviors can be easily solved when using the prioritized allocation. When the request from the separation behavior is high enough, it leaves no room for other results to be blended with it and overpower its influence. In the common case, the results are smoothly blended together using a weighted sum.

We see the prioritized acceleration allocation as an improvement over the simple weighted sum from implementation described in Subsection 4.3.2, because it does the same in the common case, but it can account for the aforementioned edge cases. However, the issue with using the weighted sum still remains. After multiplying the desired acceleration (or velocity) by its weight, the information about the acceleration rate (or speed) is lost.

4.4 Our Implementation – Desire Weighted Velocities

The main idea behind our implementation is improving on the concept of weighted sum from the implementation described in Subsection 4.3.2. Our implementation

solves the issue that after a desired velocity is multipled by its weight, the information about the speed is lost. This section later suggest some improvements, including the priority allocation from implementation described in Subsection 4.3.3.

A vector represents a direction and a size. The size cannot represent both a speed, and how much it is desirable to use the velocity. This was at the core of the problems when using a weighted sum. It is clear that one more quantity associated with the behavior results is needed. We will call this quantity the desire. Assume that behaviors return results in the following form:

$$r_j = (\vec{v_{dj}}, d_j)$$

where d_j is the current desire for a given behavior, calculated in the behavior. Our proposed *Merger* is:

$$\boxed{\vec{v_d} = \hat{dir} * speed} \tag{4.2}$$

where $\hat{dir}$ is a normalized desired direction, and *speed* is the desired speed. The $\vec{dir}$ and *speed* are calculated as follows:

$$\vec{dir} = \sum_j^k \hat{v_{dj}} * d_j$$

$$speed = \frac{\sum_j^k |\vec{v_{dj}}| * d_j}{\sum_j^k d_j}$$

The desired direction $\hat{dir}$ is calculated using a weighted sum of all desired directions $\hat{v_{dj}}$. Note, here the weights are the desires d_j, returned from the behaviors. The desired speed *speed* is given by weighted mean of all the speeds $|\vec{v_{dj}}|$. Each result has influence on $\vec{direction}$ and *speed* proportional to d_j.

In this implementation, constant weights are no longer needed in the *Merger*. The desires serve this purpose. Calculating the desire in the behavior has the added benefit that the behavior can convey how desirable the behavior's result is in the current moment.

Constraint 1

The first constraint is not satisfied. Just as in Subsection 4.3.2, the final $\vec{v_d}$ is a blend of all the results. This can be resolved using prioritized allocation.

Constraint 2

The second constraint is satisfied. A small change in any result will not cause drastic changes of the final $\vec{v_d}$. However, the direction could change as discussed under Constraint 2 in Subsection 4.3.2.

Constraint 3

The third constraint is satisfied. For a single behavior result, v_{dj} is returned, regardless of the current desire d_j.

Constraint 4

The fourth constraint is satisfied. The desires weigh the relative importance of the results. However, the desires are given by the behaviors, not by the game designer. The desires can however be multiplied by constant weights to adjust the relative importance of the behaviors. For example $d_j = d'_j * w_j$. Where d'_j is determined by the behavior, and w_j is a user-defined constant.

4.4.1 Priority Allocation

The implementation proposed in this section (Equation 4.2) can be enhanced further using the priority allocation described in Subsection 4.3.3. The result type can be extended to include a priority.

$$r_j = (\vec{v_{dj}}, d_j, p_j)$$

While the previously in Algorithm 3, the results were clamped based on sum of magnitudes of accelerations, we clamp them based on sum of desires. For simplicity, the priority allocation step can be done first, to filter out unused results. Afterwards, the remaining results can be merged using the weighted mean described at the beginning of this section[3]. The filtering can be described by pseudocode in Algorithm 4.

In an actual implementation, care should be taken to handle the cases where multiple behaviors return the same priority. In that case, they will be arbitrarily ordered in the sorted list. In that case, there might be behaviors with same priorities, but only some of them might be included in *newResults*.

One solution to this would be the following: Split the *results* into sub-lists according to their priorities. Merge the sub-lists into one result using the weighted mean. Now there are results with unique priorities only, and Algorithm 4 can be used to filter them. Finally merge the filtered *newResults* using the weighted mean.

[3]In an actual implementation, if performance is a concern, it would be more efficient to calculate everything in one pass of the results array.

Algorithm 4: Max sum desire results filtering algorithm

Data: List of results $results$, maximum sum desire $maxSumDesire$
Result: New list of results, with low priority results filtered out

```
// Sort in descending order based on priorities
sort(results);
currentSum ← 0;
maxIndex ← length(results) − 1
for i ← 0 to length(results) − 1 do
    currentSum ← currentSum + results[i].d;
    if currentSum > maxSumDesire then
        // Account for the leftover
        results[i].d ← results[i].d − (currentSum − maxSumDesire);
        maxIndex ← i;
        break;
newResults ← {results[0], ..., results[maxIndex]};
return newResults;
```

4.4.2 Workflow Improvements

Currently, the same desire determines the influence on the final direction as well as the speed. It can, however, also be the case that some behaviors do not care about the boid's speed, only the direction.

One way to try to solve this could be to define a "comfort speed" c for each animal. If a specific behavior does not care about the speed, the desired direction is simply scaled by c. This may somewhat work for most cases, but can cause troubles when a specific speed is needed. Consider a behavior with desired speed equal to zero. Now, even if other behaviors do not care about the speed, their results would bring the final speed closer to c.

Second solution would be to have behaviors return two desires, one for direction, the other one for speed. When discussing potential improvements to our *Mover* in Subsection 3.5.1, we suggested that behaviors could also return the desired size of acceleration. For this, another desire could be introduced to match the desired size of acceleration. Blending can again be done using weighted mean as was the case with *speed*. The desired size of acceleration could then be passed to the *Mover*. The complete result with all the improvements could be as follows:

$$r_j = \Big((\vec{dir}_j, dirDesire_j), (speed_j, speedDesire_j), (acc_j, accDesire_j), p_j\Big)$$

If control over rotation is also needed on per behavior basis, this could be extended further to include desired rotation, desired angular speed and desired angular acceleration and their respective desires. Now every behavior would be responsible for calculating all of these quantities, making the behaviors more complex to implement. To balance complexity and fine control, our framework and behaviors will use the following results by default:

$$r_j = \Big((\vec{dir}_j, dirDesire_j), (speed_j, speedDesire_j), p_j\Big)$$

5. Neighbor Queries

This chapter discusses possible *Neighbor Queries*, similarily as previous Chapters 3 and 4 covered the *Movers* and the *Mergers*. As outlined in Chapter 2, *Neighbor Queries* filter all the boids into a subset that a given behavior uses. Limiting boids' perception has two reasons, first it can increase the realism, because perception of real animals is also limited. Second, it decreases the runtime needed to run behaviors. On the other hand, finding the neighbors can become costly for performance. Reynolds discussed the problem already in his paper which introduces the Boid model [7], he described some solutions to this problem, in his later papers [39] [40], where he uses a spatial partitioning grid, and mentions that Binary Space Partitioning (BSP) trees could also be used. Both approaches are discussed later in this chapter.

This chapter is split into three sections. The first Section 5.1 shows methods that can be used to simulate limited perception. In the second Section 5.2, the choice of *Neighbor Query* is analyzed in regards to runtime of behaviors. The third Section 5.3 explores ways to optimize nearest neighbor search.

5.1 Limited Perception

This section discusses options to determine what a given boid $boid_i$ perceives. The simplest approach would be that every boid considers the whole set of *Boids* when running its behaviors. However, as Reynolds notes, this is unlikely for real birds. He proposes limiting the vision by distance and a field of view. He argues that simulating a limited vision increases perceived realism of the simulation [7]. This can be further supported with paper by Ballerini et al. [41], where the authors show that only around 7 nearest birds have significant influence on the behavior of real birds in a flock. This section will discuss combination of these three conditions: Limiting vision by a radius - *maximum distance condition*, limiting vision by field of vision - *field of view condition*, taking only k nearest neighbors - *k–nn condition*.

As described more formally in the Chapter 2, a *Neighbor Query* takes in the set of all *Boids* and a current $boid_i$ and returns a subset of the *Boids* set.

$$nQuery : (boid_i, Boids) \rightarrow neighborhood_i \subseteq Boids$$

5.1.1 All in Radius

The simplest idea, which implementations of limited perception usually share, is taking all other boids within some radius r, as shown in Figure 5.1. Limiting the neighborhood by distance is natural, because near neighbors are likely to have the most valuable information. For example, it is more important to try avoid collision with a bird 1 meter away, rather than one who is 100 meters away. This can be further accounted for in the *Neighbor Behaviors*, where the influence of boids further can have lesser effect, as discusses by Reynolds [7]. This idea is discussed further in Subsection 6.4.4. Furthemore, Reynolds suggests that when deciding on specific values for r, it can also be considered that, for example, fish in

murky waters have very limited vision, while birds can see much farther through air [7].

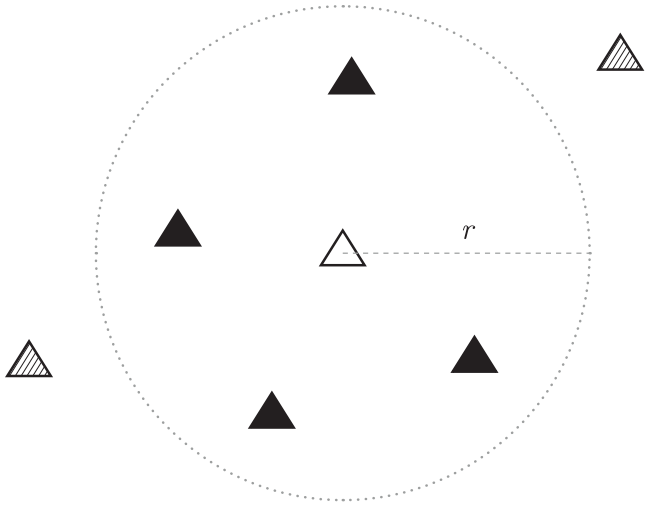

Figure 5.1: All neighbors within radius r. Boid searching for neighbors is white, boids inside its vision are black, the rest is striped.

So far, taking all neighbors within a radius r was considered. However, it can be useful having different radii for different behaviors. For example for separation behavior, using small radius means that only real threats of collision are considered. If the radius is too small however, there might not be enough time to avoid the collision course in time. For cohesion on the other hand, larger radius can prevent boids from loosing their flockmates after being split up. Radius for alignment does not need to be too large, it is most important to be aligned with near neighbors, as that makes collisions with them less likely. Additionally, from our experience, the radius for alignment should be smaller than for cohesion. That ensures that a boid first joins the flock, and only then starts aligning with its flockmates. From our testing, the ordering of $r_{sep} < r_{alignment} < r_{cohesion}$, shown in Figure 5.2, worked best for us. This ordering is quite common, an example can be found for example in a video by the Youtube Channel "Coding Train" [42]. The author also provides an online interactive simulation including the source code [43], which contains the same ordering. The same ordering was also used in a paper by Chiang et al. [44], which explored using the Boids model for crowd simulations.

Limiting the vision only by radius would mean that the boids can essentially see behind themselves. This is not quite realistic. It would be more realistic to also check if the boid is within some field of view. This is discussed in the next Subsection 5.1.2.

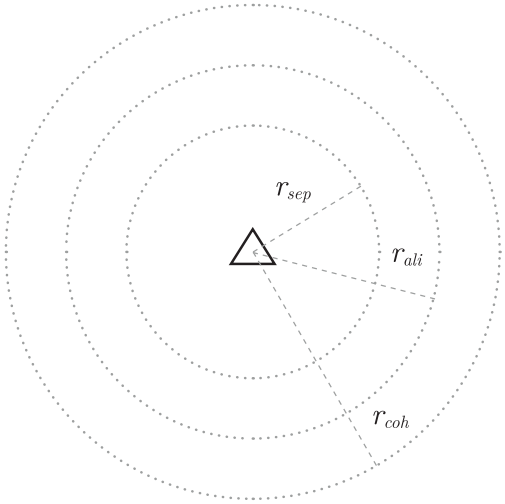

Figure 5.2: Illustration of a boid with three different radii of vision for cohesion, alignment and separation behavior. The radii are denoted with r_{coh}, r_{alig} and r_{sep}.

Note on Optimization of Maximum Distance Condition

To check if a boid is a neighbor according to the maxium distance condition, the distance to it can be compared to the radius of vision r. This condition would have to be checked for many potential neighbors. Therefore, a common optimization technique for this check is worth mentioning. The idea is to avoid expensive square root operation that is normally necessary to determine the distance. The optimization uses the fact that the square root function is strictly increasing on the interval $[0, \infty]$, so if $sqrt(d_{sq}) < r$, then $d_{sq} < r^2$. Calculating only the square distance d_{sq} and comparing it against square of the radius r elimitates the need for calculating square root. In the following, if condition given by Equation 5.1 is satisfied, then boid at position p_2 is inside the radius of a boid at position p_1.

$$\boxed{d_{sq} < r^2} \tag{5.1}$$
$$d_{sq} = v_x^2 + v_y^2$$
$$\vec{v} = \vec{p_2} - \vec{p_1}$$

5.1.2 FOV

In the Subsection 5.1.1, limiting the perception by maximum distance r was dicussed. However, real animals have vision limited not only by distance, but also by their field of view. When simulating limited perception, it would be more realistic to also account for a field of view of an animal, as proposed by Reynolds [7]. The aforementioned maximum distance condition can be combined with a condition that checks whether a potential neighbor is also within a specified field of view. This is illustrated in Figure 5.3, where only boids within radius r and field of view FOV are considered neighbors. In context of computer games, the FOV could be chosen by a game designer to customize an animal's behavior.

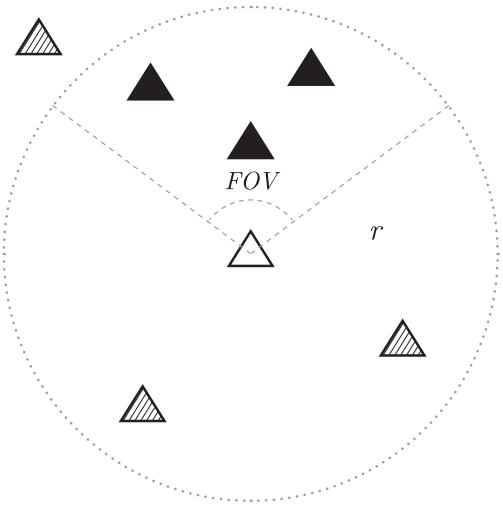

Figure 5.3: Combination of field of view and maximum distance conditions. Black boids are considered neighbors, as they are both within the radius and the field of view.

When deciding what values for FOV to use, nature can be an interesting source of inspiration. For example, predators often have a narrower field of view, while prey animals typically possess a wider one [45]. Narrow field of view gives predators wider area of stereoscopic vision which grants them depth perception crucial to hunting. Prey animals benefit from the wider field of view for detecting predators. For example, the field of view of a sheep is around 320-340 degrees [45], while the field of view of a wolf around 250 degrees [46].

Although limiting the perception by field of view may be more realistic, it can cause some issues. Smaller field of view can result in more collisions between flockmates, because they cannot account for flockmates behind them. In our implementations, using a larger field of view for separation proved to be a simple approach to resolve this issue. Using different FOV for each behavior is similar to using different radii for different behaviors from Subsection 5.1.1. This will be used in our implementations to increase customizability.

Note on Optimization of FOV Condition

Same as with the maximum distance condition, it is worth optimizating the field of view condition. Assume two boids b_1 and b_2. We want to decide whether b_2 is in the field of view of b_1. This can be done by calculating the angle θ between the forward direction of b_1 and the direction from b_1 to b_2. If θ is smaller than $\frac{FOV}{2}$[1], then b_2 is in the field of view of b_1.

A direct approach would be to find the angle θ using the dot product. The dot product of two normalized vectors $\vec{v}$ and $\vec{u}$ gives $cos(\theta)$. From this follows that $arccos(\vec{v} \cdot \vec{u})$ gives θ, which can be used to compare against $\frac{FOV}{2}$. However, since cos is only decreasing on the interval $[0, \pi]$, one can instead check if $cos(\theta) > cos(\frac{FOV}{2})$, which implies that $\theta < \frac{FOV}{2}$. The performance improvement comes from the fact that this way, the expensive $arccos$ can be avoided completely, in

[1]FOV is divided by 2 because the angle is measured from the forward direction to the left and right.

a similar way that square root was avoided in Subsection 5.1.1. Furthermore, $cos(\frac{FOV}{2})$ can be cached, and $cos(\theta)$ is given by an inexpensive dot product. In the following, if condition given by Equation 5.2 is satisfied, a boid at position p_2 is in the field of view of a boid at position p_1 with forward direction $\vec{f_1}$.

$$\boxed{\vec{f_1} \cdot \vec{u} > cos(\frac{FOV}{2})} \tag{5.2}$$

$$\vec{u} = \frac{p_2 - p_1}{|p_2 - p_1|}$$

5.1.3 Topological Distance

One assumption brought into Reynolds' model is that the boids' behaviors depend on all boids within some radius. Experimental study on flocks of birds by Ballerini et al. [41] discovered that the influence of a bird on an individual depends on the topological distance, rather than euclidian distance. Topological distance is defined as the number of birds between the individual and the bird in question, when sorted by euclidian distance.

The authors found that after 7th nearest neighbor, the influence of birds farther is negligible. They argue that unlike in classical models, the attraction to a flockmate is the same regardless of the distance, as long as the topological distance is the same. The authors speculate that limiting the number of neighbors could be beneficial, because it reduces information noise a bird receives, and thus it can make better decisions. This experimental finding suggests that limiting the number of neighbors may increase realism. Based on this idea, only k nearest neighbors can be considered as neighbors, and the rest filtered out. Limiting the maximum number of neighbors to a constant is not only a promising way to create a more realistic model, but it can also increase the run time performance (discussed in Section 5.2). Reynolds also used k nearest neighbors to improve performance in his later paper focused on performant real time flocking, titled "Big Fast Crowds on PS3" [40]. Due to possible increase in realism and better performance, we will use this approach as well.

Taking k nearest neighbors can be combined with the maximum distance condition, and the field of view condition, as illustrated in Figure 5.4 for $k = 3$. Note that some boids in the figure would pass the radius and the field of view conditions, but they are not considered, since they are not within the k nearest neighbors. A *Neighbor Query* can take in a maximum number of neighbors k and return the k nearest neighbors sorted by their distance. The game designers can use k as an additional parameter to adjust behaviors. The maximum number of neighbors k can also be lowered if better performance is required.

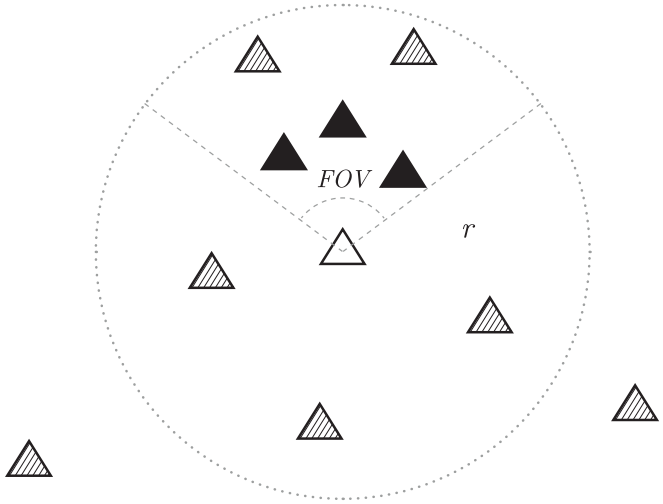

Figure 5.4: Considering only three nearest neighbors restricted by maximum distance and a field of view. Only the black boids are considered neighbors.

5.1.4 Other Possibilities

There are other possible alternatives and additions for *Neighbor Queries*, which we have not experimented with. For example, random noise could be added to simulate imperfect perception. Or vision can be implemented as raycasts, to simulate occlusion by other flock members. We do not use ray casts to query neighbors, but we use them for collision avoidance (Chapter 7). Another idea by Reynolds [39], was to use "dynamically shaped neighborhood". He made a simulation where birds get startled by an incoming car. There, the neighborhood is elongated in the direction of the car's velocity. If a bird finds itself in this shape, it will react to it. This is an interesting addition, which we would like to experiment with in the future.

5.2 Effect of Vision on Performance

The previous section discussed approaches for *Neighbor Queries*. The choice of a *Neighbor Query* has significant implications on the performance. If all boids interacted with all other boids, the time complexity of running the *Neighbor Behaviors* would be $\mathcal{O}(n^2)$, where n is the number of boids. This assumes that the behaviors run in time linearly proportional to the number of neighbors, which we found is usually the case. This does not scale well as size of flock increases. Limiting the vision can be a way to combat this quadratic complexity.

This section will analyze how the choice of the *Neighbor Query* affects the performance cost of running the *Neighbor Behaviors*. The performance cost of finding the neighbors itself is discussed in Section 5.3. The main focus here is on the difference between taking all neighbors within a radius (Subsection 5.1.1), and taking k nearest neighbors (Subsection 5.1.3).

5.2.1 All in Radius

Suppose a behavior should use all neighbors within a radius as described in Subsection 5.1.1. In terms of performance, the worst case would be when the whole flock is contained within one point. In this case, all n boids have n neighbors, so the worst case time complexity of the simulation would be $\mathcal{O}(n^2)$. However, in practice, it can be assumed that there is some minimum distance d_{min} between all boids keeping them separated, as shown in Figure 5.5. This combined with maximum distance r, can give an upper constant bound on the maximum number of neighbors m. This means a linear worst case complexity $\mathcal{O}(n * m)$. This idea was considered in Reynolds' paper titled "Interaction with Group of Autonomous Characters" [39], but relationship between m, r, and d_{min} was not discussed there. It is important to consider the size of the upper bound m, as it could still end up being greater than the number of boids we are interested in simulating.

Assuming some r and d_{min}, a boid cannot have more than m neighbors, where m is the maximum number of circles of radius d_{min} which can fit inside a circle of radius r, when working in 2D. Considering the areas in 2D and volumes in 3D, m cannot be greater than r^2/d_{min}^2 in 2D or r^3/d_{min}^3 in 3D. This indicates that the worst case time complexity is proportional to r and inversely proportional to d_{min}. The minimum separation distance d_{min} is likely implied from the size of the animal, so essentially only r can be reduced if there is a need for better performance.

To get a more concrete sense of what the upper bound on m could be, consider the following convervative scenario. Birds maintaining a minimum separation of 0.5 meters, with vision up 20 meters. In 2D this would give $m < 1600$, in 3D $m < 64000$. This puts into perspective how much worse the situation is in 3D. Moreover, we are mainly concerned with smaller flocks of hundreds or lower thousands of boids, which could mean that $n < m$, therefore our upper bound on the worst case run time could still be quadratic with respect to n.

Further factor affecting the performance in practice is distribution of the boids. In the worst case, each boid has up to m neighbors, in the best case all boids no neighbors. In other words, dense flocks will have worse performance than sparse flocks. Figures below show two extreme of distributions, Figure 5.5 shows worst case where all boids see each other, while Figure 5.6 contains the best case where no boids see each other. While r can be adjusted to improve the worst case performance, the distribution of the boids is unpredictable and can lead to unstable frame rates.

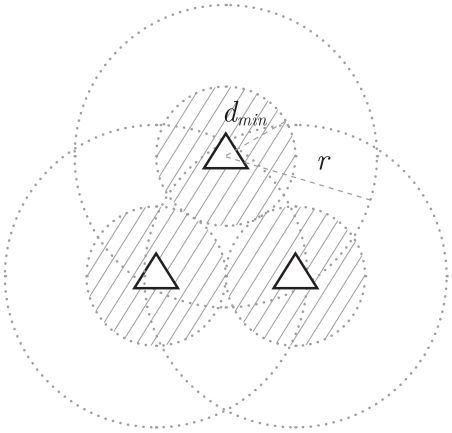

Figure 5.5: A dense flock where all the boids are neighboring each other. Boids have maximum radius of vision r, and minimum separation distance d_{min}.

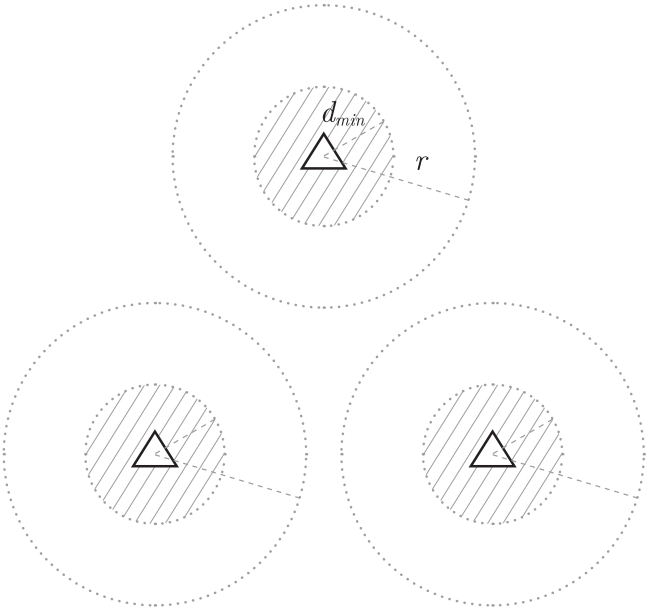

Figure 5.6: A sparse flock where all the boids have no neighbors. Boids have maximum radius of vision r, and minimum separation distance d_{min}.

5.2.2 K Nearest

When using k nearest neighbors, each boid only interacts with at most k neighbors. Thus the worst case time complexity of running the behaviors is linear $\mathcal{O}(n * k)$. While theoretically, the previous approach also has linear worst case time complexity, the multiplicative constant there depends on the radius of vision. Here, the benefit is that if better performance is needed, k can be reduced without sacrificing the maximum distance of the boids' vision.

Setting k can serve as an upper bound on the resources the behaviors are allowed to use. This way, even if a flock becomes temporarily very dense, it does not cause a lag spike, as we identified would be the case when taking all neighbors within radius in Subsection 5.2.1.

Achieving stable frame rate is especially important in games. It would be undesirable if a flock of birds in the background causes random frame rate spikes based on its density. All this indicates that taking k nearest neighbors is a promising approach from the perspective of how performance heavy it is to run the behaviors. However, finding the k nearest neighbors efficiently could still prove to be performance heavy. This is discussed further in Subsections 5.3.2 and 5.3.3.

5.3 Neighbor Search

This section is focused on different choices when implementing neighbor search, with the main focus on performance. First, two options to structure the algorithm of a flocking simulation in regards to the neighbor search are discussed. Then, two data structures for neighbor search, K–d trees and spatial partitioning, are considered and compared. For both data structures, we will focus on using them to find all neighbors within a radius r and using them to find k nearest neighbors.

5.3.1 Algorithm Structure

In Chapter 2, we formalized the *Neighbor Behaviors* as functions that take in a boid and a set of its neighbors. However, what we found in other implementations was a little different. Usually, there is only one pass that calculates the behaviors while searching for the neighbors, as shown in Algorithm 5. For simplicity, the neighbors here are identified by iterating over all other boids naively. They are then used to calculate a cohesion behavior. Other behaviors like separation or alignment could be calculated in the same pass.

Let us consider only a simple implementation of a cohesion behavior for now. Usually a centroid of all neighbors is calculated, and velocity towards it is returned. There are two options, either the cohesion behavior does the neighbor search itself, or the neighbor search is done before the cohesion behavior is called. Algorithms 6, 5 show implementations of a simple cohesion behavior using the two alternatives.

Algorithm 5: Neighbor search in behaviors

Input : Boid $boid_i$, other Boids $Boids$, maximum distance $maxDistance$
Output: Desired velocity v_d

```
1 sumPosition = (0, 0, 0);
2 numNeighbors = 0;
3 for boid_j in Boids do
4     if distance(boid_i, boid_j) < maxDistance then
5         sumPosition = sumPosition + boid_j.position;
6         numNeighbors = numNeighbors + 1;
7 centroid = sumPosition/numNeighbors;
8 v_d = centroid − boid_i.position;
9 return v_d;
```

Algorithm 6: Neighbor search before behaviors

Input : Boid $boid_i$, neighbors of $boid_i$ $Neighbors$
Output: Desired velocity v_d

```
1 ...
2 for boid_j in Neighbors do
3     sumPosition = sumPosition + boid_j.position;
4     numNeighbors = numNeighbors + 1;
5 ...
6 return v_d;
```

It is clear that the second approach (Algorithm 6) is more modular. The cohesion function is only concerned with calculating the centroid, and not with finding the neighbors. This lends itself better to swapping out the neighbor search algorithm or sharing an already calculated neighborhood between multiple behaviors.

While not being very modular, the first approach (Algorithm 5) can be more efficient. The behavior is implemented as a single loop over all boids, while using the seconds approach would require two loops – one for finding the neighbors and a second one for calculating the centroid. Moreover, the second approach requires storing the neighbors in memory. When searching all neighbors within radius, this could be up to $\mathcal{O}(m)$ per boid in the worst case, where m is the maximum number of neighbors from Subsection 5.2.1. If the whole neighbor adjencency matrix was precomputed, $\mathcal{O}(n * m)$ would be required.

Memory Cost

To see how problematic allocating memory for the whole adjencency matrix would be, we conducted a quick test reference hardware described in Subsection 1.1.1. As discussed in Subsection 5.2.1, m can easily be larger than n, so assuming a larger simulation where $m = n = 5000$, an array of $5000 * 5000$ booleans using Unity's `NativeArray` [47] was allocated and deallocated. This took 12 to 20 ms in Editor, far from acceptable for a real time simulation. This potential issue can be resolved by using only k nearest neighbors. That way, the entire adjencency matrix requires $\mathcal{O}(n * k)$ of memory. Another test on the reference hardware, assuming $k = 7$ and $n = 5000$, was conducted. We assume $k = 7$ based on the paper discussed in Subsection 5.1.3. The allocation and deallocation of a `NativeArray` of $7 * 5000$ booleans took only around 0.007 ms in Editor.

Conclusion

Using the second approach (Algorithm 6) with k neighbors provides a good balance between performance and modularity. Additionally, note that when using k nearest neighbors, it needs to be used with second approach. It would not possible to use k nearest neighbors with the first approach, because it is impossible to know beforehand, which boid will be one of the k nearest neighbors, while iterating over them for the first time. Furthemore, as discussed in Subsection 5.2, using k nearest neighbors can guarantee more stable frame rates. For the afor-

mentioned reasons, we chose to structure framework according to Algorithm 6 and limit the number of neighbors to a user given constast k.

5.3.2 Spatial Partition Grid

Finding the k nearest neighbors or all neighbors within a radius can be very performance costly. This subsection discusess one approach to optimize spatial queries, using a spatial partition grid data structure. This was used, for example, by Reynolds for his simulations [39] [40]. The main idea is to divide space into a uniform grid. Then, each boid is assigned to the cell it currently occupies in space. When querying for neighbors, only the cells within the radius r need to be searched.

Dense Grid

There are two options when implementing this structure: a dense and a sparse grid. Dense grid is implemented as a 2D array of cells, where each cell contains a list of boids. The indexes into the array correspond to the coordinates of the cell. Figure 5.7 shows a dense grid of size 3 by 3, where width of each cell $w_{cell} = 5$. In the figure, the boids occupy the cells at indexes $[1, 1]$ and $[2, 2]$. Note that memory needs to be allocated for all cells, even if they are not unoccupied. Also, the case where a boid moves outside of the allocated grid would need to be accounted for, either by growing the grid or by limiting the boids' positions to a bounding box.

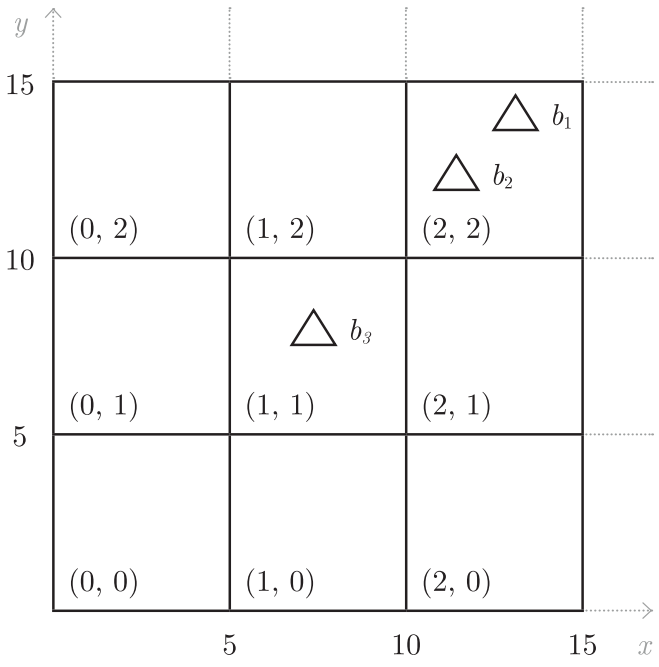

Figure 5.7: A dense grid covering a square section of space from $[0, 0]$ to $[15, 15]$. The grid is split into a 3 by 3 matrix, width and height of one cell is 5. Each cell represents one element of a 2D array, and can contain a reference to multiple boids.

Sparse Grid

Sparse grid uses the same idea as dense grid, but it is represented by a hash map. The key is a hash of coordinates of the cell, and the value is a list of boids. The same situation as in Figure 5.7 is shown in Figure 5.8. Note that here, memory only needs to be allocated for the used cells. In the figure below, that is cells at indexes $[1, 1]$ and $[2, 2]$.

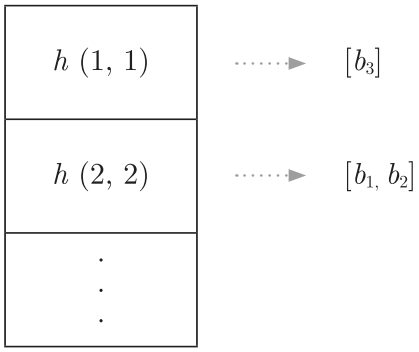

Figure 5.8: A sparse grid implemented as a hash map. Key is a hash of index which the boid would have in dense grid. Each element holds an array of boids.

Comparing Sparse and Dense Grid

Dense grid can have better performance, because working with a contiguous array can be more cache friendly. Furthermore, it avoids overhead of hashing the coordinates. The main disadvantage is that the case when boids move out of the grid has to be handled. Additionally, memory is wasted on unoccupied cells. The main weakness we see with the sparse grid is that the hash map look up is likely to be slower than indexing an array, and the memory access is less cache friendly.

Because the framework aims to be useful in many different scenarios and to be user-friendly, we see sparse grid as the better option. This way, the user of the framework will not have to think about maintaining a bounding box around the dense grid, or dense grid taking up too much memory.

Note on Performance of Construction of Spatial Partitioning Grid

To construct the grid, each boid is simply added to its corresponding cell. In case of sparse grid, we assume a hashmap with $\mathcal{O}(1)$ insertion. Therefore, in both cases the time complexity of both approaches is $\mathcal{O}(n)$, where n is the number of boids.

From memory perspective, sparse grid only allocates what it needs, so only $\mathcal{O}(n)$ memory is needed. In case of the dense grid, our main concern is allocating memory for all the cells inside its bounding box. Let us assume a bounding box in 3D, for simplicity a cube with sides of length w_{bb} and thus volume of w_{bb}^3. Then the number of cells needed is w_{bb}^3/w_{cell}^3, where w_{cell} is a cell's width. Thus the memory complexity would be $\mathcal{O}(n + w_{bb}^3/w_{cell}^3)$. While the extra term is a constant, in practice this constant may be very high, as it scales with the square (in 2D) or cube (in 3D) of the bounding box's side length.

All in Radius Search

To find all neighbors of a given boid, boids in all cells within its radius need to be considered. A common approach to simplify this, is to set $w_{cell} = 2r$. This was described, for example, in book titled *Game Programming Design Patterns* [48]. This way, only the current cell and all adjacent to it need to be checked. In that case, the search needs to visit 9 cells in 2D and 27 in 3D.

We will consider the upper bound on the number of flockmates that need to be visited. Same as in earlier sections of this chapter, we assume all boids to have a minimum separation distance of d_{min}. This means that each boid occupies area of πd_{min}^2. Then in the worst case, each cell can contain up to $w_{cell}^2/(\pi d_{min}^2)$ boids in 2D and $w_{cell}^3/(\pi d_{min}^3)$ in 3D. Thus, the maximum number of neighbors a boid needs to visit can be bounded by constant $m_{grid} = 9w_{cell}^2/(\pi d_{min}^2)$. While having a constant worst case look up time $\mathcal{O}(m_{grid})$ may seem good, it is not as important if m_{grid} is larger than the number of boids n. As we saw in Section 5.2.1, even an upper bound on all neighbors within radius r can easily be larger than n for our purposes, and it grows quadratically in 2D and cubically in 3D with the radius. From perspective of worst case analysis, spatial partitioning grids can theoretically be used to find full adjencency matrix in linear $\mathcal{O}(nm_{grid})$ time. This may provide a huge advantage for cases where $n >> m_{grid}$. Meaning either the radius of vision has to be small, which implies small m_{grid}, or the total number of boids n needs to large.

Until now, we considered $w_{cell} = 2r$. However, this is not necessary. Implementing the search such that w_{cell} does not depend on r would be a bit more difficult, but it can provide greater flexibility in regards to performance, since w_{cell} can be adjusted independently of r. Figures 5.9, 5.10 show all cells that would need to be searched, for different choices of w_{cell}. Smaller cells result in lesser area to search, but more overhead as more cells need to be visited and have memory allocated. We have not experimented with this approach, but it would be an interesting addition. For our implementations, we only used the clasical approach where $w_{cell} = 2r$, due its simplicity.

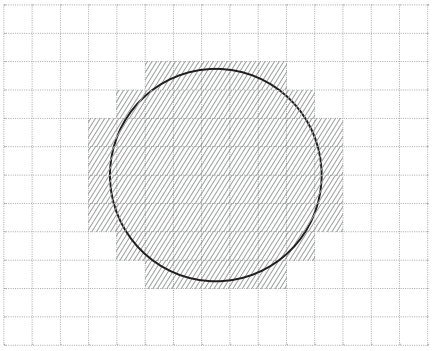

Figure 5.9: A spatial partition grid, with a small width of cells w_{cell} relative to the search radius.

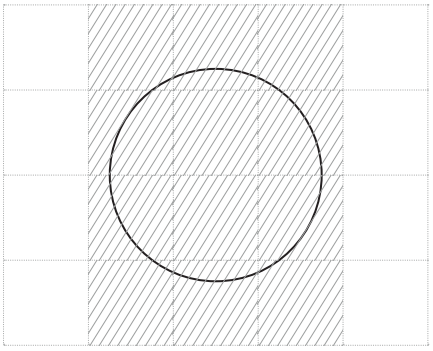

Figure 5.10: A spatial partition grid, with a large width of cells w_{cell} relative to the search radius.

K–Nearest Neighbors

In case of k nearest neighbors search, the same cells as when searching all neighbors within radius need to be visited. To find the nearest neighbors, the neighbors need to be sorted by their distance to the boid. An efficient way to achieve this is having a priority queue with k best guesses and adding all potential neighbors to it as they are being found. Insertion into a priority queue is $\mathcal{O}(log(k))$, thus the time complexity of one neighbor search is $\mathcal{O}(m_{grid} * log(k))$ in the worst case. Note that despite finding only k nearest neighbors, the worst case time complexity is higher than to find all in a radius.

The search could be optimized further by searching the cells in order of their distance to the boid and stopping once the priority queue contains k neighbors, with no unvisited cell capable of containing a neighbor closer than the current farthest neighbor. This optimization could be especially beneficial for a grid with small w_{cell}, as the one shown in Figure 5.9. For a boid inside a densely packed flock, only a few nearest cells would need to be considered.

5.3.3 K–d Trees

The second data structure that can accelerate the neighbor search is a K–d tree. It was first described by Friedman, Bentley, and Finkel in 1977 [49]. A more modern introduction to this data structure was written in 2019 by Martin Skrodzki [50]. A K–d tree, shown in Figure 5.11, is a type of binary tree, where each node represents a region of space. In each node, space is split by each spacial dimension alternately. For example, in 2D, the root node would split the space on the x axis, its children split it on the y axis, and that node's children split it on the x axis again [50].

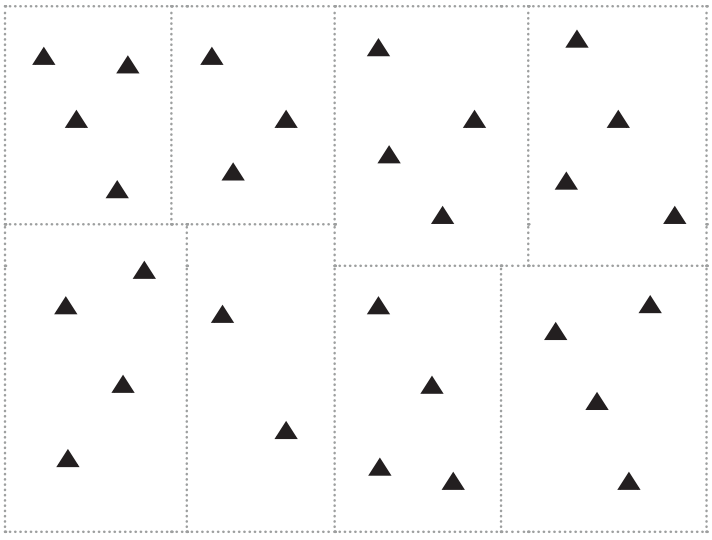

Figure 5.11: A K–d tree containing several boids in its leaf nodes. The whole space was first split once vertically, these two halves were then split horizontally and lastly the four regions were again split vertically.

Note on Performance of Construction of a K–d Grid

Building a balanced K–d tree can be done in $\Theta(n * log(n))$ time, and the tree will use $\Theta(n)$ memory [50]. To balance the tree, the median of the points in the region is chosen as the splitting axis. From algorithmic complexity perspective, the amount of memory needed is the same as when using spatial partitioning grid. However, the construction cost is $log(n)$ times larger. This additional initialization cost might be balanced out by potential performance improvement of the search.

Nearest neighbor

K–d tree enables efficient look up of a nearest neighbor of a point p. The algorithm recursively goes down the tree, choosing regions of space containing the point p. When the leaf node is reached, the minimum distance to the points in that region is calculated. This represent the currently best candidate for the nearest neighbor. Then, the recursion goes back up the tree, and at each level, the algorithm checks whether other regions of space could contain a closer point [2]. If so, that region of space is searched as well, and if any point is closer, the current minimum distance is updated. This algorithm is $\mathcal{O}(log(n))$ [50].

K–Nearest Neighbors

The algorithm to find k nearest neighbors is similar to a single nearest search, with a small difference. Instead of a single best guess for minimum distance, a priority queue of best guesses is maintained. At the top is the farthest of the nearest guesses, it is used to decide whether a region of space can contain a closer point. Friedman et al. [49] who invented the K–d tree show that finding k nearest neighbors is also $\mathcal{O}(log(n))$, and provide empirical evidence that while the number of points examined increases with k, it increases "slightly slower" than linearly. This implies that an upper bound on the time complexity which accounts for k is $\mathcal{O}(k * log(n))$.

All in Radius

K–d trees can also be used to find all neighbors within a radius r. K–d trees support efficient range search, meaning finding all points within a given bounding box. Range search was discussed by Bentley and Friedman in their "Survey of Algorithms and Data Structures for Range Searching" [51]. Range search could be used to find all neighbors within a radius r, by finding all points within a bounding box of width $w = r * 2$ first. Range search with K–d trees is $\mathcal{O}(n^{1-1/d} + m_{kd})$, where d is the number of dimensions, and m_{kd} is the number of points found [51]. Worst case m_{kd} for our use case can be estimated with $m_{kd} = w^2/(\pi d_{min}^2)$ for 2D, similarily as with the spatial partitioning grid. Note that here, the constant m_{kd} is 9 times smaller for 2D and 27 times smaller in 3D than m_{grid}, when using the spatial partitioning grid.

[2] The region cannot contain a closer point if all points in that region are farther than the current best candidate. This can be decided simply by comparing the current minimum distance to the distance between the point p and the splitting axis of the region.

5.3.4 Comparison

We saw that both K–d trees and spatial partitioning grids can be used to find all neighbors within a radius r as well as k nearest neighbors. The two data structures will now be compared.

K Nearest

We know that finding k nearest neighbors cannot be worse than $\mathcal{O}(klog(n))$ for K–d trees, and $\mathcal{O}(m_{grid}log(k))$ for spatial partitioning grids. Theoretically, the time complexity when using K–d trees is worse, because it grows logarithmically with n, while it is constant for spatial partitioning grids. In practice, as we are focused on simulations of lower thousands of boids, even assuming 10000 boids gives $log_2(10000) = 13.2$. On the other hand it is apparent from Subsection 5.2.1 that m_{grid} could easily be much larger for our use case. This suggests that using K–d trees could be more efficient for smaller flocks, especially if r is large.

All in Radius

For finding all neighbors within a radius r, the time complexity of K–d trees is $\mathcal{O}(n^{1-1/d} + m_{kd})$, meaning it scales badly for higher dimensions. For spatial partitioning grids it is $\mathcal{O}(m_{grid})$. As we saw, m_{kd} is smaller than m_{grid}, especially in 3D. This indicates again that for smaller flocks, K–d trees could be a good option, while spatial partitioning grids are better fit for larger flocks, because of their constant time complexity. This is especially true in 3D, where K–d trees have $\mathcal{O}(n^{2/3} + m_{kd})$ time complexity.

Construction

The construction of K–d trees is $\mathcal{O}(n * log(n))$ and the construction of spatial partitioning grids is $\mathcal{O}(n)$. This further indicates that K–d trees will be less efficient for larger flocks.

Conclusion

From the analysis above, it is clear that both data structures have their advantages and disadvantages. The optimal choice of the data structure depends on the specific values of n, r, d_{min}, and whether all neighbors within radius are needed or only the k nearest. The analysis however indicates that for our use case of hundreds to lower thousands of boids, K–d trees are likely to be a better choice in most cases. It will, however, be necessary to profile the actual performance for a specific use case, to find the best option.

5.4 Conclusion

In this chapter, different options for limitting the perception of boids were given. Traditional options included having a limited radius and field of view of vision. A combination of them is a good option to increase realism and performance.

Then, we found that limiting the maximum number of neighbors to k can increase realism [41] and performance even further. Afterwards, we discussed that precomputing the neighborhood first, and passing it to the behaviors as in Algorithm 6 is a good way to increase modularity of the whole simulation.

Lastly, K–d trees and spatial partitioning grids were discussed together with their implications on performance. It was found that for our use case of smaller flocks, K–d trees will likely be a better option. However, due to the numerous tradeoffs between K–d trees and spatial partitioning grids, our framework will include both data structures.

It is important to note, that since the complexity of running the behaviors is $\mathcal{O}(nk)$, and finding k nearest neighbors for all boids is $\mathcal{O}(nk * log(n))$ (using K–d trees), the worst case time complexity of the whole simulation is $\mathcal{O}(nk * log(n))$. This means that the neighbor queries can be expected to be the main bottleneck. This is why it is crucial to choose the right data structure, and implement it efficiently.

6. Flocking Behaviors

The basic Boids model was discussed in the Chapter 2. The core model needs three *Neighbor Behaviors*: *cohesion*, *alignment* and *separation*. In this chapter, only the three core *Neighbor Behaviors* responsible for flocking will be discussed in detail, to illustrate our main ideas behind implementing new behaviors. However, as mentioned in the formalization, other types of behaviors can be used to extend flocking. For example, wandering (*Simple Behaviors*), briefly discussed at the end of this chapter, or obstacle avoidance (*Ray Behaviors*), discussed in Chapter 7. Our detailed discussion of the three main *Neighbor Behaviors* will later provide useful techniques for both *Simple Behaviors* and *Ray Behaviors*.

As was the case in Chapters 3, 4, we first set some assumptions and constraints about *Neighbor Behaviors*, since they are the main focus. Then, we analyze two implementations of the three *Neighbor Behaviors* responsible for flocking, and evaluate them against the constraints. Lastly, using the analysis, our own implementation of the three flocking behaviors will be proposed.

6.1 Assumptions

Assumptions about the behaviors will now be stated. The *Neighbor Behaviors* accept a boid and a list of its neighbors. There is some data associated with each boid – for simplicity, only their positions and velocities. The behaviors usually return a desired velocity or a desired acceleration. In our case, they will return a desired direction, desired speed and desires for the direction and speed. The idea behind this was described in Subsection 4.4.2, which discussed improvements for the types of results passed into our *Merger*. Furthermore, it is assumed that there is some maximum radius of vision r for each behavior. In case of our behaviors, each behavior also has a field of view *fov* associated with it.

Furthemore, all behaviors will be analyzed in isolation from each other, and independently of the *Mergers* and *Movers*. When possible, it is assumed that there is only one *Neighbor Behavior*, which returns a desired velocity. The desired velocity would then be passed directly to a *Mover*, which gradually updates the boid's velocity and position. Considering each behavior in isolation will simplify the analysis, and make sure that each behavior is sensible on its own.

The inputs are:

- $b_c = \{\vec{p_c}, \vec{v_c}\}$, the current boid's position and velocity
- $N_c = \{b_1, b_2, \ldots, b_m\}$ the current boid's neighborhood
  where $b_j = (\vec{p_j}, \vec{v_j})$ is the position and velocity of the j-th neighbor
- r maximum radius of vision

The outputs are:

- $\vec{v_d}$ or $\vec{a_d}$ the boid's desired velocity or desired acceleration. In case of our behaviors, q, which contains a desired velocity with its desires.

6.2 Constraints

We have identified a set of constraints, which we believe the *Neighbor Behaviors* should satisfy. The constraints are:

1. *The functions should be "smooth" functions of their inputs.*

   This makes sure that there are no sudden changes when the inputs change slightly, similarily as with *Mergers* in Section 4.2. This is to ensure smooth behavior without jittering.

2. *It must be possible to adjust some weights or other parameters to determine strength of the behavior.*

   This is targeted at designers. Adjusting weights gives them an easy way to adjust the behavior of the boids, without touching the code directly.

3. *A neighbor should have no influence on a behavior's result, as distance to the neighbor goes to* $\boldsymbol{r}$.

   This makes sure that the *Neighbor Behaviors* remain smooth even as a neighbor leaves the radius of vision. This builds on the idea that after some distance r, the neighbors leave the boids vision as discussed in Subsection 5.2.1. Together with constraint 1, this suggests that the strength of influence of a neighbor should be a smooth decreasing function of distance to the neighbor, giving zero as distance goes to r.

4. *The maximum possible size of the result should be finite.*

   This makes sure that there is an upper bound on the size of the result. If the result has no upper bound, then it is more difficult to reason about the behavior, and adjust it by designers. Having no upper bound could create edge cases where the influence of a behavior becomes unexpectedly high, and the behavior overpowers all other behaviors, even when it normally should not.

5. *The maximum possible size of the result should be independent of the number of neighbors.*

   This partially follows from the fourth constraint. For example, if the maximum size of the result was proportional to the number of neighbors, then the result could be arbitrarily large. Furthemore, keeping the maximum size of the result independent of the number of neighbors makes it easier to reason about the behaviors. It will also make it easier to reuse the same behaviors with the same parameters for different sizes of flocks.

6. *The maximum possible size of the result should be independent of the radius of vision.*

   This is similar to the fifth constraint. The main issue for designers would be that after changing the radius of vision, they would likely have to adjust some parameters of the behavior as well.

6.3 Analysis of Other Implementations

We chose two implementations of the three *Neighbor Behaviors*, which we will analyze. The first implementation is simpler, and captures the most basic ideas. The second implementation is similar to the first one, but it conforms better to the specified constraints. The behaviors will be analyzed against the constraints and discussed. In Section 6.4, this analysis is used to propose our own implementation of the behaviors.

6.3.1 Implementation 1 – Simple Flocking

An example of a simple implementation of the three rules can be found in paper by Alaliyat et al. [30]. The paper is mainly focused on the finding optimal weights for the three behaviors[1].

Behavior Output Semantincs

In this paper, it is difficult to determine if the semantics of their output is desired acceleration, or desired velocity. They calculate three components $\vec{coh}$, $\vec{ali}$ and $\vec{sep}$, and then set their weighted sum to be the boid's new velocity $\vec{v_n}$. This would imply that all the three behaviors are desired velocities, because setting a sum of accelerations to velocity would not make sense. The following equation is given:

$$\boxed{\vec{v_n} = w_{coh}\vec{coh} + w_{ali}\vec{ali} + w_{sep}\vec{sep}} \tag{6.1}$$

Where w_{coh}, w_{ali} and w_{sep} are weights of the three behaviors.

From perspective of our formalization in Chapter 2, the weighted sum would be the *Merger*. Merging using a weighted sum was desribed earlier in Subsection 4.3.2. The *Mover* would correspond to the simplest possible *Mover* (Subsection 3.4.1), that simply sets the new velocity to be the desired velocity. This indicates that the results are desired velocities. However, when looking at their alignment function, it seems that it uses a desired acceleration semantics. Results for cohesion and separation might be either, if we consider looking at them in isolation, without knowing which *Merger* and *Mover* is used. Therefore, both options will be discussed. However, when considering desired acceleration, assume that an appropriate *Mover* based on Euler integration is used.

Output Semantics Issue

The authors' implementation illustrates an important issue that we discussed in context of weighted sum *Merger* (Subsection 4.3.2). Consider the cohesion behavior. The desired velocity is multiplied by w_{coh}. Adjusting the weight should increase or decrease the influence this behavior has relative to the other behaviors. However, after multiplying by w_{coh} the information about the desired speed is essentially lost. Consider that only one behavior is active. Then, the boids will want to travel at speed proportional to w_{coh}. A designer changing the weight would likely not expect this. They would only expect the behavior to have a stronger influence relative to other behaviors.

[1] In fact the authors use 5 behaviors, but we will only consider the main three.

Cohesion

The authors (Alaliyat et al. [30]) use the following equations for cohesion:

$$\boxed{\vec{coh} = \vec{p_{avg}} - \vec{p_c}} \tag{6.2}$$

$$\vec{p_{avg}} = \frac{\sum_{\vec{p_j} \in N} \vec{p_j}}{m}$$

This equation calculates the centroid[2] of the neighborhood, $\vec{p_{avg}}$, and returns a vector $\vec{coh}$ pointing from a boid at position p_c to $\vec{p_{avg}}$, as shown in Figure 6.1. Finding the centroid is commonly done for cohesion, it was also used by Reynolds in his original paper [7].

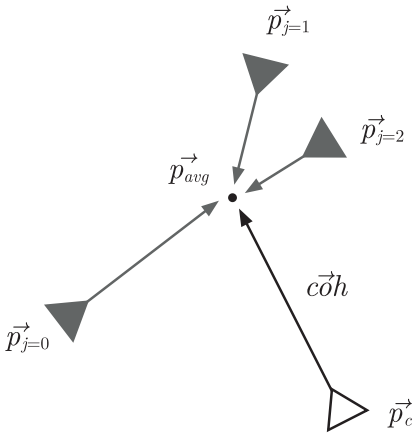

Figure 6.1: Illustration of a cohesion behavior. Centroid $\vec{p_{avg}}$ is calculated from neighbors' positions. The behavior returns vector $\vec{coh}$ towards it.

Consider the size of the resulting vector, $d_c = |\vec{coh}|$, the distance to the centroid. Here, d_c would be the desired speed or desired size of acceleration based on the semantics. In the first case, this would mean decreasing speed as the centroid is reached, in the second case, the boid would decrease acceleration to the centroid as it is reached. This would be similar to a force pulling the boid towards the centroid. From perspective of merging the behaviors, it also means that the behavior has higher relative influence to other behaviors, as the boid gets further from the centroid.

The difference between the two semantics can be seen in Figures 6.2 and 6.3. On the left, a boid travelling at $\vec{v_c}$ is accelerated by $\vec{coh}$ towards the centroid $\vec{p_{avg}}$. Its new velocity is $\vec{v_n}$. Note that while the direction of $\vec{v_n}$ points more to $\vec{p_{avg}}$ than $\vec{v_c}$, the speed increased, meaning that the boid might overshoot its target. On the right, the boid has desired velocity $\vec{coh}$, pointing towards $\vec{p_{avg}}$. Based on that and its $\vec{v_c}$, a desired acceleration $\vec{a_d}$ is determined. Adding $\vec{a_d}$ to $\vec{v_c}$ would make the boid have velocity of $\vec{coh}$.

Based on this, it seems it would be best to give desired velocity towards the centroid, and then determine a desired acceleration based on that. The conversion into desired acceleration could be part of the *Mover*, if all behaviors

[2] Centroid is the "average position".

are expected to return a desired velocity, or it could be done inside the behavior, if the behavior is supposed to return a desired acceleration. In the authors' implementation, determining the desired acceleration is not necessary, because they set $\vec{coh}$ directly as the new velocity $\vec{v_n}$.

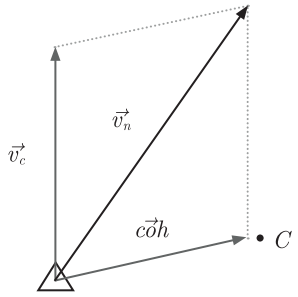

Figure 6.2: Boid with velocity $\vec{v_c}$ accelerated towards centroid $\vec{p_{avg}}$ by $\vec{coh}$.

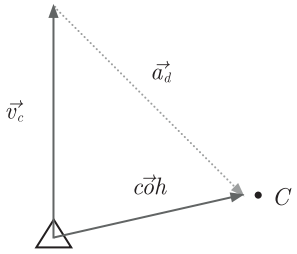

Figure 6.3: Boid with velocity $\vec{v_c}$, desired acceleration $\vec{a_d}$ is calculated such that $\vec{coh} = \vec{v_c} + \vec{a_d}$.

Constraint 1

The first constraint is satisfied. The function is smooth, the result depends on the centroid, which changes smoothly with the positions of the neighbors.

Constraint 2

The second constraint is satisfied. There is a single weight w_{coh}, which can be adjusted.

Constraint 3

The third constraint is **not** satisfied. Neighbors influence the result regardless of distance to them.

Constraint 4

The fourth constraint is satisfied. The maximum size of the result is r, the radius of vision, which is finite.

Constraint 5

The fifth constraint is satisfied. The maximum size of the result is r, which is independent of the number of neighbors.

Constraint 6

The sixth constraint is **not** satisfied. The maximum size of the result is r, so it is exactly the radius of vision.

Alignment

The authors (Alaliyat et al. [30]) use the following equations for alignment:

$$\boxed{\vec{ali} = \vec{v_{avg}} - \vec{v_c}} \tag{6.3}$$

$$\vec{v_{avg}} = \frac{\sum_{\vec{v_j} \in N} \vec{v_j}}{m}$$

As is usually the case when calculating alignment, the average velocity of the neighborhood, $\vec{v_{avg}}$, is calculated. However, in this case, the current boid's velocity $\vec{v_c}$ is subtracted from it. This would indicate that $\vec{ali}$ is a correction vector to be added to the current velocity. In other words, it would be a desired acceleration. However, consider that only this behavior is active. Then, using Equation 6.3 together with Equation 6.1, the boid's new velocity would be equal to $w_{ali}\vec{ali}$, not $\vec{v_{avg}}$ as one would expect. This again highlights the importance of semantics. We believe that in this case, the authors should have set $\vec{ali} = \vec{v_{avg}}$ directly.

Assume the corrected version, $\vec{ali} = \vec{v_{avg}}$, with desired velocity semantics. It is interesting to note that the larger the speed of a neighbor, the bigger influence it will have on the direction of $\vec{v_{avg}}$. This may not be desirable. Further consider a case with two neighbors, both travelling in the exact opposite direction. One travels at $10\,\mathrm{m\,s^{-1}}$, the other at $9\,\mathrm{m\,s^{-1}}$. If both vectors are summed together, the result will be in the direction of the first boid, and its size will be $1\,\mathrm{m\,s^{-1}}$. It may be better for the desired speed to be $9.5\,\mathrm{m\,s^{-1}}$, the average of the two speeds.

Constraint 1

The first constraint is satisfied. The function is smooth because its result depends on the average velocity, which changes smoothly when the neighbors' velocities change smoothly.

Constraint 2

The second constraint is satisfied. There is a single weight w_{ali}, which can be adjusted.

Constraint 3

The third constraint is **not** satisfied. Neighbors influence the result regardless of distance to them.

Constraint 4

The fourth constraint is satisfied. The maximum size of the result is s_{max}, the neighbors' maximum speed, which we assume to be finite.

Constraint 5

The fifth constraint is satisfied. The maximum size of the result is s_{max}, which is independent of the number of neighbors.

Constraint 6

The sixth constraint is satisfied. The maximum size of the result is s_{max}, which is independent of the radius of vision.

Separation

The authors (Alaliyat et al. [30]) use the following equation for separation:

$$\boxed{\vec{sep} = \sum_{\vec{p_j} \in N} (\vec{p_c} - \vec{p_j})} \tag{6.4}$$

The authors' separation is given by the sum of vectors pointing from each of the neighbors position $\vec{p_j}$, to the current boid's position $\vec{p_c}$. In this case, both semantics could work, either the boid accelerates away from collisions, or the boid wants to move in a direction away from collisions. However we believe the second option makes better sense intuitively.

The function sums up vectors from each neighbor to the current boid. Each vector is longer the bigger the distance d to the neighbor. Consider only a single neighbor. The closer it gets to the boid, the smaller $\vec{sep}$ will be. Therefore, $\vec{sep}$ will have less effect when merging the behaviors with Equation 6.1. This is exact opposite of what one might expect. The size of $\vec{sep}$ should grow, as the neighbor gets closer, in order to overpower the other two behaviors. Now consider that there is more neighbors. The closer a neighbor is, the less influence it has on direction of $\vec{sep}$. Again this is unexpected, because separating from the closest neighbors should be the most important. The maximum size of $\vec{sep}$ depends on the number of neighbors m, and it is at most $m * r$ (assuming all neighbors are at the same point at distance r).

Constraint 1

The first constraint is satisfied. The function is smooth – small change in position of one neighbor means small change in the result.

Constraint 2

The second constraint is satisfied. There is a single weight w_{sep}, which can be adjusted.

Constraint 3

The third constraint is **not** satisfied. Neighbors influence the result more, the further away they are.

Constraint 4

The fourth constraint is **mostly** satisfied. The maximum size of the result is $m * r$. If there is some bound on the number of neighbors m, then $m * r$ will be finite. If we consider all neighbors within a radius, the upper bound on m can be very large, as discussed in Subsection 5.2.1.

Constraint 5

The fifth constraint is **not** satisfied. The maximum size of the result is directly proportional to the number of neighbors m.

Constraint 6

The sixth constraint is **not** satisfied. The maximum size of the result is directly proportional to the radius of vision r.

6.3.2 Implementation 2 – UAV Flocking

As a second example, we chose an implementation which is focused on a very different use case than ours. The paper by Hoang et al. [52] uses the Reynolds' boids model to control swarms of Unmanned Aerial Vehicles (UAVs). The interesting aspect is that because of the nature of having a swarm of real UAVs, the authors needed more robust guarantees about the swarm.

Note, when analyzing the behaviors, the authors handle the special cases where the number of neighbors m is zero. In those cases they return $\vec{0}$ to avoid divisions by zero. We describe their implementation without this detail for the sake of simplicity, but it would be an important detail in an actual implementation.

Behavior Output Semantincs

Same as in the previous example, the authors use the weighted sum of the three behaviors. This time, however, the result is added to the current velocity, rather than setting it directly. The authors interpret the results as desired velocity. We consider the addition of desired velocity to the current velocity as *Mover* described in Subsection 3.4.2 together with its issues.

The authors give the following equation:

$$\vec{v_n} = \vec{v_c} + \sum_{i=1}^{m} w_i \vec{r_i}$$

where w_i is the weight of the i-th behavior, $\vec{r_i}$ is the result of the i-th behavior and $\vec{v_c}$ and $\vec{v_n}$ are the current and new velocity of the boid respectively.

Looking closer at the authors' behaviors, it is clear that each function computes some normalized direction, $\vec{r_i}$, where the boid should travel, and multiply it by w_i. Unlike in the previous example, the weights are not constant. They are calculated as part of the behavior. We believe that multiplying normalized directions by weights which specify their influence would correspond more to semantics of desired directions, rather than desired velocities. In the previous example (Subsection 6.3.1), we discussed the difficulties when the size of the result has two meanings. Assuming that $\vec{r_i}$ is the desired direction, and that the weight indicates the level of influence the behavior should have in the weighted sum, the size of $w_i\vec{r_i}$ has only one interpretation. To control speed, they clamp $\vec{v_n}$ to a maxium length. For the sake of simplicty, the semantics of desired directions will be assumed throughout the analysis.

Cohesion

The authors (Hoang et al. [52]) use the following equations for cohesion[3]:

$$\boxed{\vec{coh} = w_{coh} \frac{\vec{c}}{||\vec{c}||}} \tag{6.5}$$

$$\vec{c} = \vec{p_{avg}} - \vec{p_c}$$

$$\vec{p_{avg}} = \frac{\sum_{\vec{p_j} \in N} \vec{p_j}}{m}$$

$$w_{coh} = \frac{M}{m}$$

where M is the "optimal number of neighbors", the authors set this to be 6.

The authors find direction towards centroid of the neighborhood $\vec{c}$, and multiply it by a weight w_{coh}. The weight is proportional to M, and inversely proportional to m. In other words, the weight is smaller, the more neighbors there are. We understand this as the boid wanting to go towards the centroid more, when it has fewer neighbors. This relationship is graphed in Figure 6.4 for different choices of M. When the number of neighbors m is equal to the optimal number M, the weight is 1, and it goes to zero as m grows larger. As the number of neighbors goes to 1, the weight goes to M. Unlike in the previous example of a cohesion function, the maximum size of $\vec{coh}$ does not depend on the distance to the centroid. We consider this an improvement.

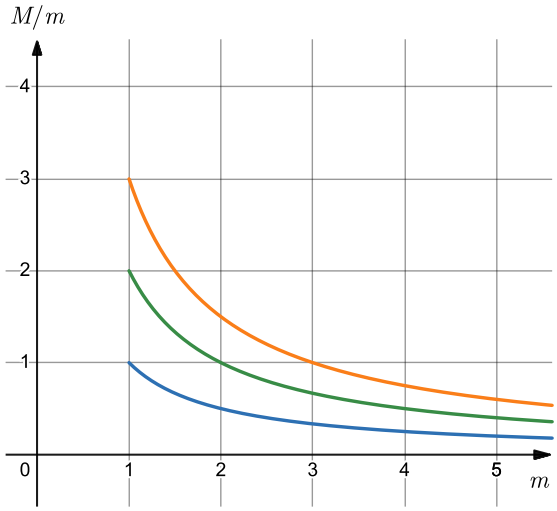

Figure 6.4: Function $\frac{M}{m}$ used by the authors for weighting the cohesion behavior, graphed for different values of M. Orange: $M = 3$, Green: $M = 2$, Blue: $M = 1$. The number of neighbors m is on the horizontal axis.

[3]Upon closer inspection, it seems that the authors forgot to subtract the current position from the centroid. We assume that this is a mistake, which we corrected in the description here.

Constraint 1

The first constraint is satisfied. The function is smooth – the result depends on the centroid, which changes smoothly with the positions of the neighbors.

Constraint 2

The second constraint is satisfied. The optimal number of neighbors M can be adjusted.

Constraint 3

The third constraint is **not** satisfied. Neighbors influence the result, regardless of distance to them.

Constraint 4

The fourth constraint is satisfied. The maximum size of the result is $\frac{M}{m}$, which is finite.

Constraint 5

The fifth constraint is **not** satisfied. The maximum size of the result is $\frac{M}{m}$, which is indirectly proportional to the number of neighbors.

Constraint 6

The sixth constraint is satisfied. The maximum size of the result is $\frac{M}{m}$, which is independent of the radius of vision.

Alignment

The authors (Hoang et al. [52]) use the following equations for alignment:

$$\vec{ali} = w_{ali} \frac{\vec{v_{avg}}}{||\vec{v_{avg}}||} \tag{6.6}$$

$$w_{ali} = \frac{M}{m}$$

$$\vec{v_{avg}} = \frac{\sum_{\vec{v_j} \in N_c} \vec{v_j}}{m}$$

This function is analogous to their cohesion function. The authors find the average velocity $\vec{v_{avg}}$ of the neighborhood, normalize it, and scale it by w_{ali}. The weight itself is also calculated in the same way as for their cohesion behavior. Same as in the previous alignment implementation (Equation 6.3), neighbors with higher speeds have larger influence on direction of $\vec{v_{avg}}$, which may not be intended. Since w_{ali} is indirectly proprotional to number of neighbors, it is interesting to consider that the more neighbors a boid has, the less it will want to align with them. Our concern with this scaling is that it may cause dense flocks to become unaligned and chaotic. The constraints are satisfied in the same way as with cohesion.

Constraint 1

The first constraint is satisfied. The function is smooth – the result depends on average velocity, which changes smoothly with velocities of the neighbors.

Constraint 2

The second constraint is satisfied. The optimal number of neighbors M can be adjusted.

Constraint 3

The third constraint is **not** satisfied. Neighbors influence the result regardless of distance to them.

Constraint 4

The fourth constraint is satisfied. The maximum size of the result is $\frac{M}{m}$, which is finite.

Constraint 5

The fifth constraint is **not** satisfied. The maximum size of the result is $\frac{M}{m}$, which is indirectly proportional to the number of neighbors.

Constraint 6

The sixth constraint is satisfied. The maximum size of the result is $\frac{M}{m}$, which is independent of the radius of vision.

Separation

The authors (Hoang et al. [52]) use the following equations for separation:

$$\boxed{\vec{sep} = w_{sep}\frac{\vec{s}}{||\vec{s}||}} \tag{6.7}$$

$$\vec{s} = \sum_{\vec{p_j} \in N} f_w(||\vec{p_c} - \vec{p_j}||)(\vec{p_c} - \vec{p_j})$$

$$w_{sep} = T_s \max_{\vec{p_j} \in N_c} f_w(||\vec{p_c} - \vec{p_j}||)$$

$$f_w(d) = r - \frac{d^2}{r}$$

where T_s is a scaling factor. The authors give an in depth explanation how to determine the scaling factor, such that the weight w_{sep} can be larger than then sum of all other weights, thus it can overpower all other behaviors. They determine it based on M from previous behaviors, maximum distance of vision r, and intended shortest possible distance between the boids d_{min}. The details are not be discussed here, as the method is specific to their use case.

To determine $\vec{sep}$, the authors take a weighted sum of vectors pointing from each neighbor's position $\vec{p_j}$ to the current boid's position $\vec{p_c}$. The weights are given by function f_w, which takes in distance between the two boids d. The function,

an upside down parabola graphed in Figure 6.5, is zero at the maximum distance r, and r at distance 0. It is apparent that $0 < f_w(d) < r \quad \forall d \in (0, r)$, and the function grows the closer the neighbor is. We see this as an improvement over Equation 6.4, where closer neighbors had lesser influence.

The weight w_{sep} is the maximum value of all the weights given by f_w, multiplied ty T_s. Thus, w_{sep} is at most rT_s. The fact that there is an upper bound on the weight regardless of the number of neighbors provides a guarantee that the boids will not separate too much, as could be the case in the previous example's separation, where the weight was essentially unbounded. We also see this as an improvement.

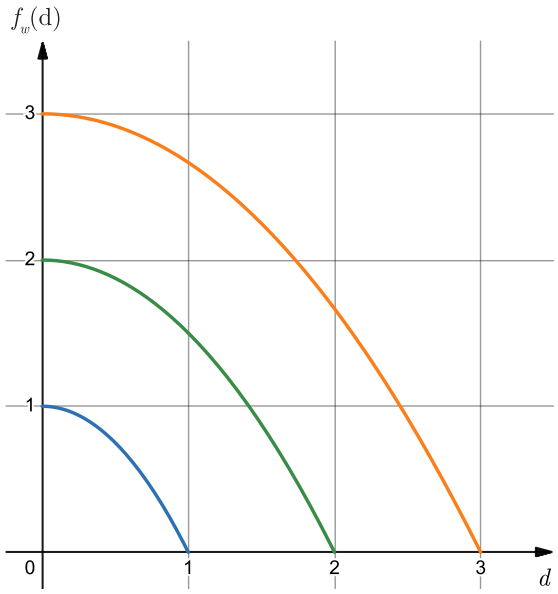

Figure 6.5: Function f_w used by the authors for weighting the separation behavior, graphed for different values of r. Orange: $r = 3$, Green: $r = 2$, Blue: $r = 1$. The distance to the neighbor d is on the horizontal axis.

Constraint 1

The first contraint is satisfied. The function is smooth – small change in position of one neighbor will mean small change in result.

Constraint 2

The second constraint is satisfied. The maximum and minimum distances r and d_{min} can be adjusted.

Constraint 3

The third constraint is satisfied. Neighbors loose influence on the result, as distance to them goes to r.

Constraint 4

The fourth constraint is satisfied. The maximum size of the result is rT_s, which is finite.

Constraint 5

The fifth constraint is satisfied. The maximum size of the result is rT_s, which is independent of the number of neighbors.

Constraint 6

The sixth constraint is **not** satisfied. The maximum size of the result is rT_s, which depends on the radius of vision. This is likely not an issue here, as they determine T_s using r such that it satisfies their criteria.

6.4 Our Implementation

This section will describe our *Neighbor Behaviors* used for flocking, and concepts used to derive them. First, similarities between behaviors from the reference implementations are discussed. Based on these similarities, we give a generalization of the behaviors. Then, we discuss easing functions, and how they can be useful when creating new behaviors. Afterwards, this generalization, together with the easing functions, is used to develop our behaviors that satisfy the constraints from Section 6.2. Finally, our entire flocking model is summarized through all the implementation choices from Chapters 3, 4, 5.

6.4.1 Semantics

When introducing our *Merger* in Section 4.4, we discussed why it is beneficial for behaviors to return more information than one 2D or 3D vector. In our implementation, each behavior returns five values:

$$q = \Big((\vec{dir}, des_{dir}), (spd, des_{spd}), p\Big)$$

where $\vec{dir}$ is the desired direction, des_{dir} is the desire of the direction, spd is the desired speed, and des_{spd} is the desire of the speed. The priority p is used by the *Merger* for prioritized allocation (see Subsection 4.3.3). In our implementation, we assume the priority to be a constant set by a game designer according their needs.

Note, when analyzing other behaviors through our constraints, we were often interested in the maximum size of a behavior's result, since it had a key role in how influential the result will be, when it is merged together with results of other behaviors. Because all the aforementioned behaviors returned a 2D or a 3D vector, we considered the size to be the length of the vector. Here, our results, q, have a more complicated type. For example, the amount of influence a result has on the final direction produced by our *Merger* from Section 4.4 is given by des_{dir}. For this reason, when discussing the constraints, q will have two sizes, des_{dir} and des_{spd}.

6.4.2 Generalization of Neighbor Behavior Functions

After reviewing the three behaviors in the previous sections, it becomes clear that they share some similarities. Identifying these similarities is useful to create a

general description of *Neighbor Behaviors*. This generalization is especially useful for the framework design. It can provide the users with a unified approach for creating new *Neighbor Behaviors*. All behaviors discussed until now first sum up some values over all the neighbors, and then use this to determine the final result. Generally, this can be expressed as:

$$q = f(b_c, q')$$
$$q' = \oplus_{b_j \in N} g(b_c, b_j)$$

Where q is the result of the behavior, given by function $f(b_c, q')$, which receives the current boid b_c and an accumulator q'. The accumulator q' is given by accumulating some intermediate results $g(b_c, b_j)$ over all neighbors b_j of the current boid b_c. The accumulation operation is given in general by some operation $\oplus$. While $\oplus$ is often a summation ($\sum$), it can more generally, represent other operations, such as finding the position of the nearest neighbor. To illustrate the generalization, the cohesion function from the first example (Equation 6.2) could be described as follows:

$$\vec{q} = \left(\frac{\vec{q'}}{m} - \vec{p_c} \right) w_{coh}$$
$$\vec{q'} = \sum_{\vec{p_j} \in N} \vec{p_j}$$

In this example:

- $\oplus = \sum$
- $g(b_c, b_j) = p_j$
- $f(b_c, q') = \left(\frac{\vec{q'}}{m} - \vec{p_c} \right) w_{coh}$

Activation Functions

In the sample cohesion behavior above, the size of the result $|\vec{q}|$ linearly decreases as boid b_c gets closer to the centroid. This is important to realize, because the size of the vector determines the influence of a behavior relative to other behaviors when they are merged. In general, the relationship does not have to be linear. Non-linear relationships could provide more interesting behaviors. For example, in the second separation behavior (Equation 6.7), analyzed in Subsection 6.3.2, the size of the result was a quadratic function of distance to the nearest neighbor.

We will call functions which shape the size of the result *activation functions*. In general, an activation function does not always need to depend on distance to something, but it will often be the case. For example, in the cohesion behavior above, the weight w_{coh} could be multiplied by a function $a(d_{centroid})$, where $d_{centroid}$ is distance to the centroid. This could be expressed as:

$$\vec{q} = \left(\frac{\vec{q'}}{m} - \vec{p_c} \right) w_{coh} a(d_{centroid})$$

In this example, the vector towards the centroid is not normalized. We saw this as problematic when analyzing the constraints of the first cohesion implementation (Subsection 6.3.1), because it meant that the maximum size of the result was proportional to the radius of vision r.

In our implementations, all behaviors will work with normalized directions, much like the second set of example behaviors from Subsection 6.3.2. In our case, a normalized direction $\vec{dir}$ will be returned, together with a desire for direction des_{dir}. The des_{dir} will be given by product of a weight w and a result of an activation function $a(x)$. To ensure a predictable maximum size of the desire, all of our activation functions will return values in the range of $[0, 1]$, regardless of the number of neighbors m, or radius of vision r. This way, each behavior will have a predictable maximum desire equal to its weight.

Having a predictable maximum desire will make it easier to assign a weight to a behavior, to adjust how influential it is relative to other behaviors when they are merged together. Furthermore, the choice of activation function will be another point where the behavior can be customized to specific needs. The high level of customizability and predictability will be useful to game designers who would want high level control over the behaviors.

Observability Functions

We introduced the concept of activation functions, which shape the influence of a behavior relative to other behaviors. A second important concept that we introduce is *observability functions*. They will be used in our behaviors to shape the influence each neighbor has on the behavior's result. This is inspired by Reynolds' idea of simulated perception, where sensitivity to other neighbors is a function of distance to them [7]. For example, in the cohesion behavior above, the influence each neighbor has on position of the centroid could be given by a function $o(d)$, where d is the distance to the neighbor. This could be expressed as follows:

$$\vec{q'} = \sum_{\vec{p_j} \in N} \vec{p_j} o(|\vec{p_j} - p_c|)$$

It is important to note that in this case, $\vec{q'}$ would have to be divided by sum of all $o(|\vec{p_j} - p_c|)$ to give a centroid. In this case, the centroid would no longer be a simple average of positions of the neighbors, but a weighted average of positions of the neighbors.

The text above only serves as a short introduction to the concept of observability functions, and where they fit into our generalization. Subsection 6.4.4 will explore this idea in more detail, including a specific implementation, which will be shared among our cohesion, alignment and separation behaviors. However, some easing functions useful for observability functions as well as activation functions need to be discussed first.

6.4.3 Easing Functions

The concept of activation and observability functions was introduced in the previous Subsection 6.4.2. However, no concrete functions were described so far.

Some functions to consider can be found among easing functions, commonly used in computer games to transitions between values. They are functions which take in a value from 0 to 1 range, and return a value from 0 to 1 range. Due to their popularity in computer games, and the ease of working with normalized values, they are very useful for our use case. A good resource covering many different easing functions is the website easings.net [53]. Below are some examples of easing functions that we experimented with for activation and observability functions for our behaviors.

Power Functions

Perhaps the simplest easing function is a quadratic $f(x) = x^2$, usually referred to as "quadratic ease in" [53]. In the same sense, "cubic ease in" is $f(x) = x^3$ [53]. Here, we generalized this concept to an arbitrary power: $f(x) = x^l$, where l is some constant. The following Figures 6.6 – 6.9 show the functions and their transformations for different values of l. The function $f(x) = x^l$ has a useful property, the derivative at $x = 0$ is 0 for $l > 1$, giving it a smooth start. The parameter l is useful since it provides one more point of customizability.

$f_{q1}(x) = x^l$

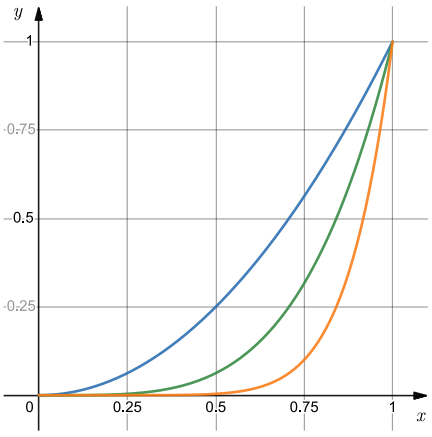

Figure 6.6: Easing function f_{q1} for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 8$, Green $l = 4$, Blue: $l = 2$.

$f_{q2}(x) = 1 - x^l$

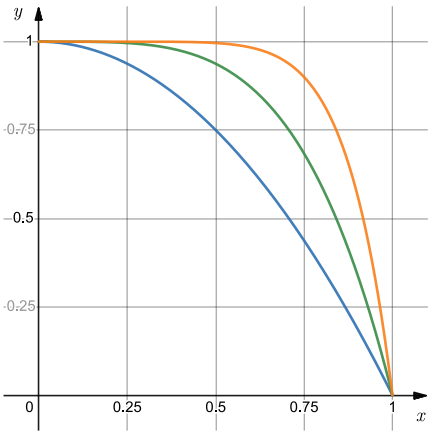

Figure 6.7: Easing function f_{q2} for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 8$, Green $l = 4$, Blue: $l = 2$.

$$f_{q3}(x) = (1-x)^l$$

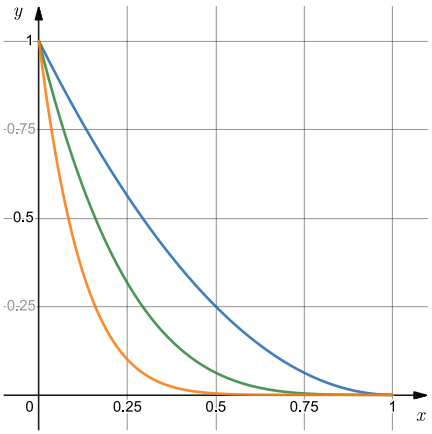

Figure 6.8: Easing function f_{q3} for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 8$, Green $l = 4$, Blue: $l = 2$.

$$f_{q4}(x) = 1 - (1-x)^l$$

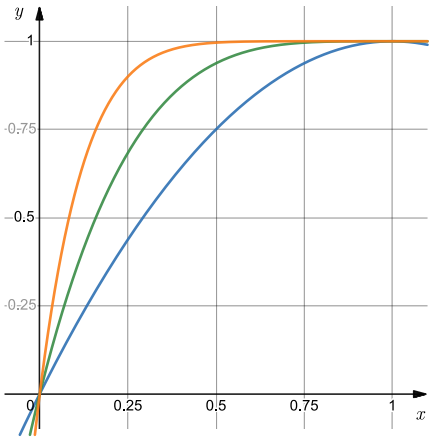

Figure 6.9: Easing function f_{q4} for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 8$, Green $l = 4$, Blue: $l = 2$.

S–Curves

Second class of useful easing functions are s-shaped curves. They are useful when both the start and the end should be smooth. The following Figures 6.10 – 6.14 show some s-shaped curves we experimented with. The functions f_{s1} and f_{s2} are so-called smoothstep functions, which are commonly used in computer graphics [54]. The function f_{s3} is a combination of $f_{q1}(x)$ and $f_{q4}(x)$. For second power, f_{s3} is usually referred to as "quadratic ease in out" [53]. The function f_{s4} is an s-shaped curve which we derived. It has a smooth start and end, and grows in between, but is not symmetric like f_{s3}. The function f_{s5} is a segment of *sin*.

$$f_{s1}(x) = 3x^2 - 2x^3$$

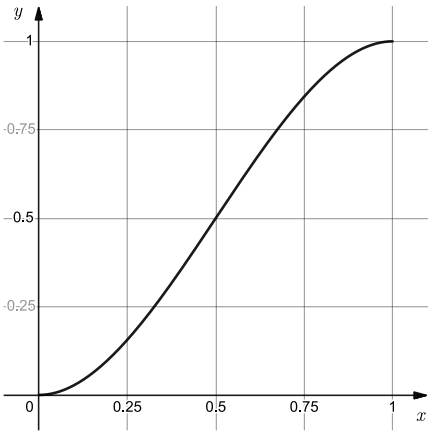

Figure 6.10: Easing function f_{s1} for $x \in [0, 1]$

$$f_{s2}(x) = 6x^5 - 15x^4 + 10x^3$$

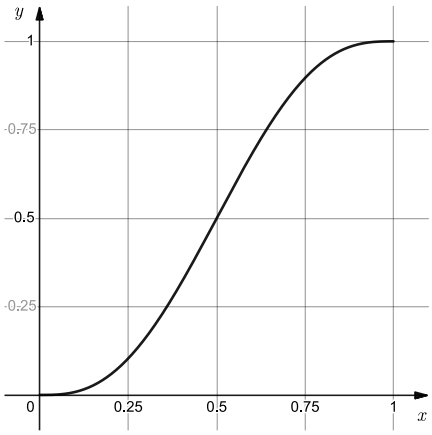

Figure 6.11: Easing function f_{s2} for $x \in [0, 1]$

$$f_{s3}(x) = \begin{cases} \frac{f_{q1}(2x)}{2} & \text{if } x \leq 0.5 \\ \frac{f_{q4}(2x-1)+1}{2} & \text{if } x \geq 0.5 \end{cases}$$

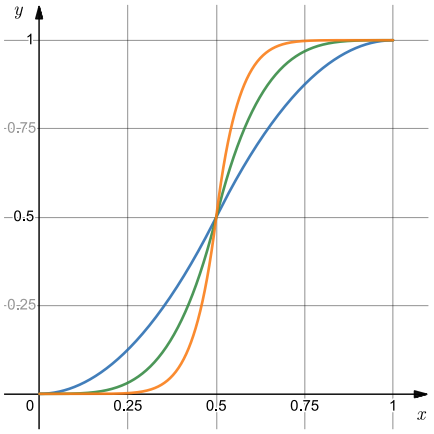

Figure 6.12: Easing function f_{s3} for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 8$, Green $l = 4$, Blue: $l = 2$.

$$f_{s4}(x) = |x-1|^{2l} - 2|x-1|^{l} + 1$$

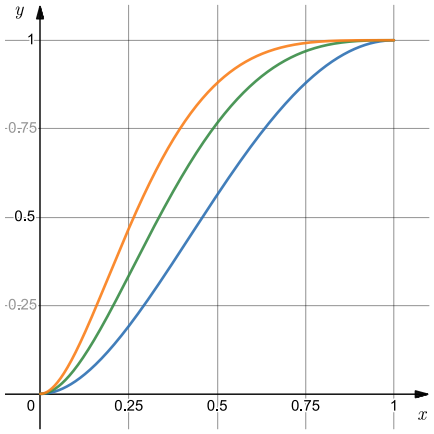

Figure 6.13: Easing function f_{s4} for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 4$, Green $l = 3$, Blue: $l = 2$.

$$f_{s5}(x) = \tfrac{1}{2} sin((x - \tfrac{1}{2}\pi))$$

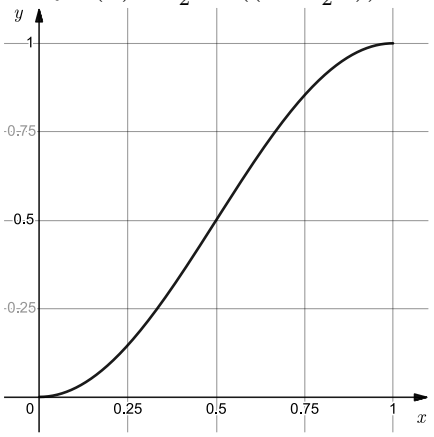

Figure 6.14: Easing function f_{s5} for $x \in [0, 1]$.

Bell Curves

Lastly, we experimented with bell shaped curves. For us, that is curves that give 0 for $x = 0$ and $x = 1$, and give 1 for $x = 0.5$. We have not found resources for this type of curves, so we built them from the previously discussed ones. The first option, $f_{b1}(x)$, is a segment of $sin()$ function. The second option $f_{b2}(x)$ is built from $f_{q4}(x)$. The third and fourth options $f_{b3}(x)$, $f_{b4}(x)$ are built from $f_{s3}(x)$ and $f_{s4}(x)$ respectively. Lastly $f_{b5}(x)$ and $f_{b6}(x)$ are built from $f_{s1}(x)$ and $f_{s2}(x)$ respectively. All the functions except for $f_{b2}(x)$ have derivatives at $x = 0$ and $x = 1$ equal to 0, and they all have derivative equal to 0 at $x = 0.5$. This means

that (except for $f_{b2}(x)$) they all have a smooth start and end, and a smooth peak at $x = 0.5$.

$$f_{b1}(x) = 0.5sin(2x\pi - 0.5\pi) + 0.5$$

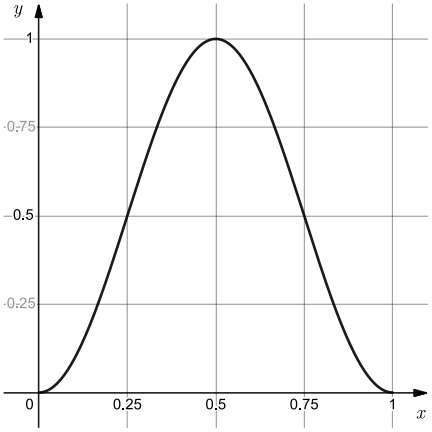

Figure 6.15: Easing function f_{b1} for $x \in [0, 1]$.

$$f_{b2}(x) = f_{q4}(1 - |2x - 1|)$$

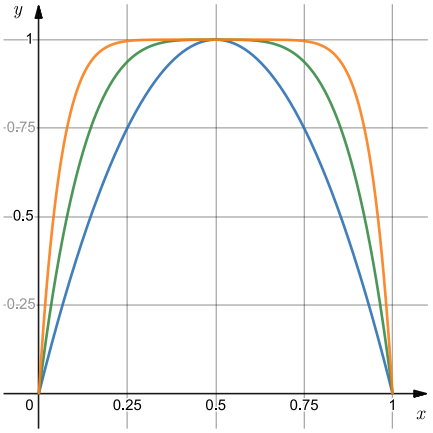

Figure 6.16: Easing function f_{b2} for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 8$, Green $l = 4$, Blue: $l = 2$.

$$f_{b3}(x) = f_{s3}(1 - |2x - 1|)$$

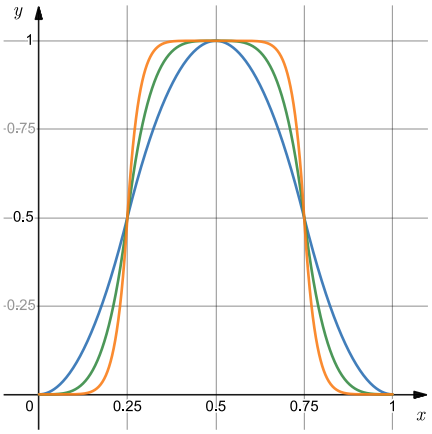

Figure 6.17: Easing function f_{b3} for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 8$, Green $l = 4$, Blue: $l = 2$.

$$f_{b4}(x) = f_{s4}(1 - |2x - 1|)$$

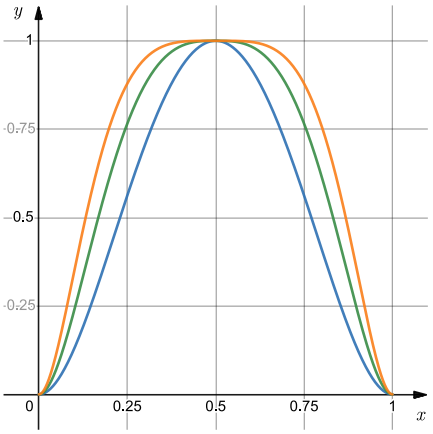

Figure 6.18: Easing function f_{b4} for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 4$, Green $l = 3$, Blue: $l = 2$.

$$f_{b5}(x) = f_{s1}(1 - |2x - 1|)$$

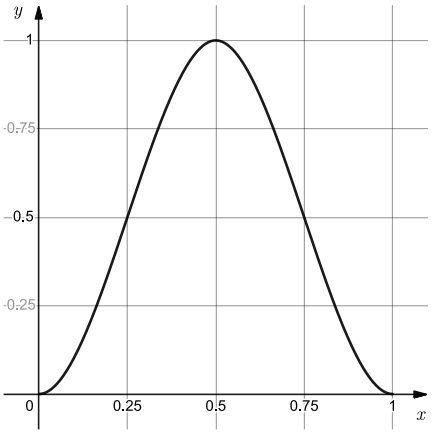

Figure 6.19: Easing function f_{b5} for $x \in [0, 1]$.

$$f_{b6}(x) = f_{s2}(1 - |2x - 1|)$$

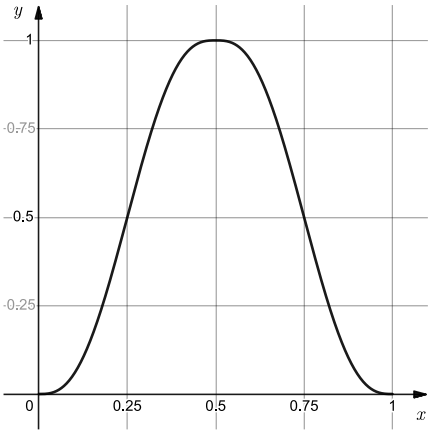

Figure 6.20: Easing function f_{b6} for $x \in [0, 1]$.

Range Remapping

All functions given above are from range $[0, 1]$ to range $[0, 1]$. It can sometimes be useful to remap them to other ranges. For example, remapping the output from $[0, 1]$ to $[y_{min}, y_{min}]$. Additionally, it can be useful to move the function to the right, or left. For example, if the function should start at x_{min}, and end at x_{max}. Given a function $f(x)$, its remapped version $f'(x)$ can be calculated as:

$$f'(x) = f\left(clamp\left(\frac{x - x_{min}}{x_{max} - x_{min}}\right)\right)(y_{max} - y_{min}) + y_{min} \tag{6.8}$$

where

$$clamp(x) = min(max(x, 0), 1)$$

Figures 6.21 – 6.23 show examples of remapping some of the previously introduced functions into different ranges.

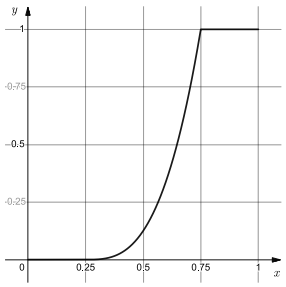

Figure 6.21: Easing function f_{q1} for $x \in [0, 1]$. Input range is remapped into $[0.25, 0.75]$

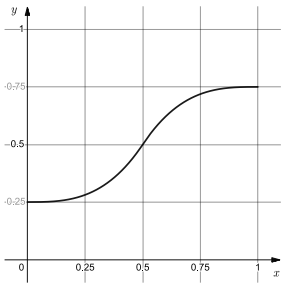

Figure 6.22: Easing function f_{s3} for $x \in [0, 1]$. Output range is remapped into $[0.25, 0.75]$

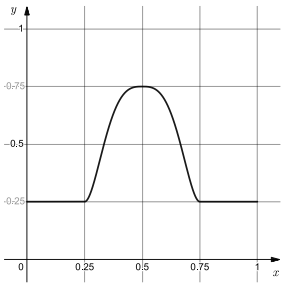

Figure 6.23: Easing function f_{b3} for $x \in [0, 1]$. Input and output ranges are both remapped into $[0.25, 0.75]$

6.4.4 Observability Functions

The concept of observability functions was introduced in the generalization in Subsection 6.4.2. The inspiration comes from Reynolds' idea that not only should the boids' perception be limited, as discussed in Chapter 5, but also different importance should be given to neighbors based on distance to them.

For us, the two main factors limiting a boid's perception are the maximum distance, r, at which the boid can observe its neighbors, and the boid's field of view *fov*. It is one of our constraints, that neighbors at distance r should have no influence on a behavior's result, to ensure smoothness. This indicates that the influence of a neighbor should be some function of distance to the boid, which gives 0 at distance r. In our model, we also assume the boids to have a field of view *fov*. The same concept can be applied here as well. The observability function should give 0 as neighbor reaches the edge of the boid's field of view *fov*.

In this sense, an observability function is a scalar field in radius r around the boid. A neighbor's position within the field gives a weight specifying how influential the neighbor should be. There are two components to this idea. We will call one *distance based observability functions*, they give a neighbor's weight based on distance. The second one, where the weight depends on the angle between the boid and its neighbor will be called *angle based observability functions*. These two components will now be analyzed separately, and then combined together to arrive at the final observability function that will be used for our behaviors.

Distance Based Observability Functions

As discussed, a distance based observability functions is a scalar field around the boid, returning a weight based on distance to its neighbor. To illustrate this idea, consider Figure 6.24. It shows a circular scalar field with radius r_{max} around a boid at position p_i. The weights are given by $o_{dist}\left(\frac{d_{ij}}{r_{max}}\right)$, where d_{ij} is the distance from p_i to a neighbor at p_j. The distance d_{ij} is divided by r_{max}, to normalize it to range $[0, 1]$. This ensures that the weight is independent of r_{max}, so that r_{max} can easily be adjusted, while preserving the relationship. The weight is visualized using a gradient at the bottom of the figure, going from blue at 0 to yellow at 1. The observability function $o_{dist}(x)$, used for this figure, gives 1 for a neighbor at distance 0, and 0 for a neighbor at distance r_{max}. This means that the closer the neighbor is, the more important it will be. To create this figure, a simple linear relationship, $o_{dist}(x) = 1 - x$, was used. In general, the relationship does not need to be linear. Other options will now be considered.

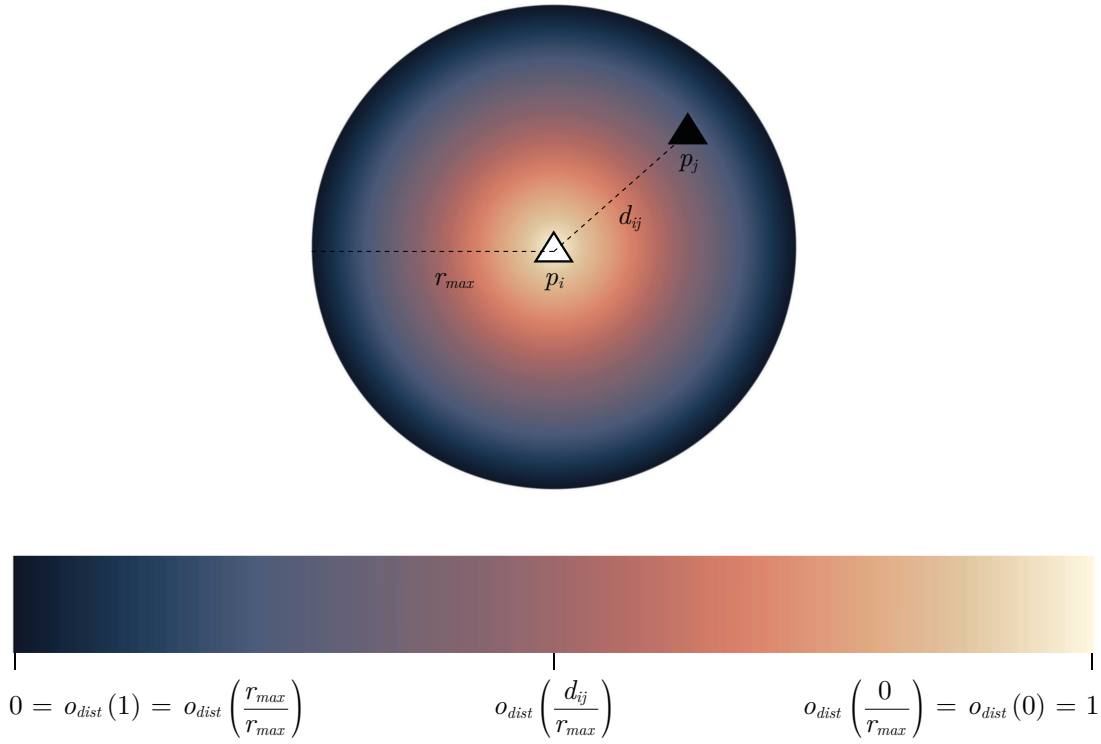

Figure 6.24: Visualization of distance based observability function as a scalar field with gradient from blue at 0 to yellow at 1. Position of current boid – p_i, neighbor's position – p_j, distance between them – d_{ij}, maximum radius – r_{max}.

The first option for a distance based observability function we will consider is based on Reynolds' first paper about flocking [7]. In the paper, he mentions he was inspired by experimental findings of Brian Partridge [55], who studied behavior of schools of fish. He suggests that the influence of neighbors is proportional to the inverse square or cube of the distance. Based on this idea, our first suggestion is the function $f_{ip}(x)$ given below:

$$f_{ip}(x) = \frac{1}{x^l} - 1$$

The function $f_{ip}(x)$ is graphed in Figure 6.25 for different choices of l. The parameter l can be used to adjust steepness of the curve. For example, $l = 2$ would correspond to inverse square relationship. One assumption here is that same as before, the distance passed in is normalized, and therefore $x \in [0, 1]$. The function $f_{ip}(x)$ gives 0 for $x = 1$, and goes to infinity as x goes to 0. For observability, this means that the influence will be larger, the closer the neighbor is. The fact that $f_{ip}(1) = 0$ is important in order to satisfy Constraint 3, which says that neighbors at distance r should have no effect.

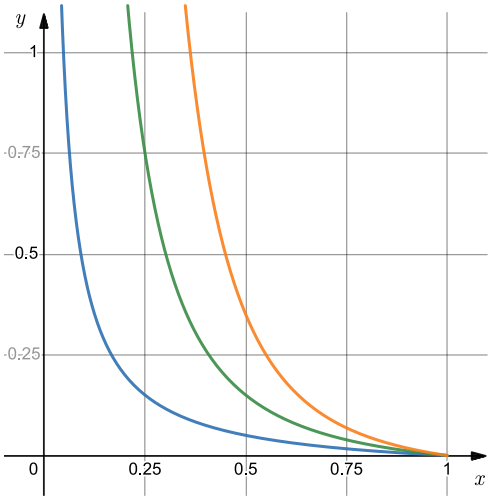

Figure 6.25: Observability function $0.05 f_{ip}$ for $x \in [0, 1]$. Graphed for different values of l. Orange: $l = 3$, Green $l = 2$, Blue: $l = 1$.

Furthemore, for illustrative purposes, Figures 6.26 – 6.28 show the visualization of $f_{ip}(x)$ for the same choices of l used in Figure 6.25. When looking at the figures, note that the function cannot be fully visualized as it goes to infinity as x goes to 0. Also note that in those figures, the function's output is scaled by 0.05 for visualization purposes. For our purpose, only the relative influence of the neighbors is important, so the actual multiplication constant does not matter. Lastly, it is important to remember that the function is not defined at 0, so it is necessary to handle this case separately, possibly by clamping all inputs below some threshold to some ϵ. This should not be an issue, since two boids occupying the exact same position should likely never happen.

Figure 6.26: Distance observability function f_{ip} for $x \in [0, 1]$ and $l = 1$.

Figure 6.27: Distance observability function f_{ip} for $x \in [0, 1]$ and $l = 2$.

Figure 6.28: Distance observability function f_{ip} for $x \in [0, 1]$ and $l = 3$.

Second option for distance based observability function that we experimented with was $f_{q2}(x) = 1 - x^l$ or $f_{q3}(x) = (1 - x)^l$, described in Subsection 6.4.3. They are both 0 for $x = 1$, as required, and they go to 1 as x goes to 0. We find it preferable to have larger differences in influence for closer neighbors, so $f_{q3}(x)$ would be the better option, because it is steeper around 0. Its visualization is shown in Figures 6.29 – 6.31 for varying options of l.

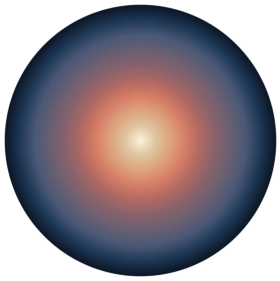

Figure 6.29: Distance observability function f_{q3} for $x \in [0, 1]$ and $l = 1$.

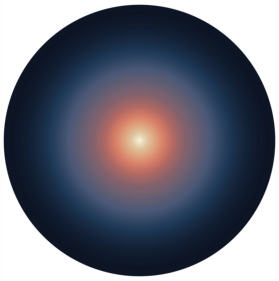

Figure 6.30: Distance observability function f_{q3} for $x \in [0, 1]$ and $l = 2$.

Figure 6.31: Distance observability function f_{q3} for $x \in [0, 1]$ and $l = 3$.

Comparing $f_{ip}(x)$ to $f_{q3}(x)$, one advantage of $f_{q3}(x)$ is that it is defined at 0. The disadvantage is that the maximum of $f_{q3}(x)$ is limited to 1, so the relative difference in influence of neighbors cannot be as large. For this reason, and because inverse square or cube should be realistic based on experimental findings, we decided to use $f_{ip}(x)$ for our behaviors.

Angle Based Observability Functions

So far, only observability based on distance was described. However, the boids' vision is limited by a field of view as well. This will be handled by an angle based observability function, in a similar manner to the distance based observability function. The idea is analogous. Here, neighbors right in front of the boid will have maximum weight, which goes to 0 as the the neighbor leaves the field of view.

Figure 6.32 illustrates this idea. In the figure, a boid at position p_i has a field of view of $fov_{max} = 2\pi$ around its current direction of movement, given by its velocity $\vec{v_i}$. The angle between the boid at p_i and its neighbor at p_j is denoted by α_{ij}. The same gradient as before is used, and again, the angles are normalized into $[0, 1]$ range before being passed in. The weight of a neighbor is therefore given by $o_{fov}\left(\frac{2\alpha_{ij}}{fov_{max}}\right)$. Again, $o_{fov}(1) = 0$ should hold, and the maximum should be for $o_{fov}(0)$. Same as before, the relationship in this figure is linear, therefore $o_{fov}(x) = 1 - x$. Other non-linear relationships will be discussed next.

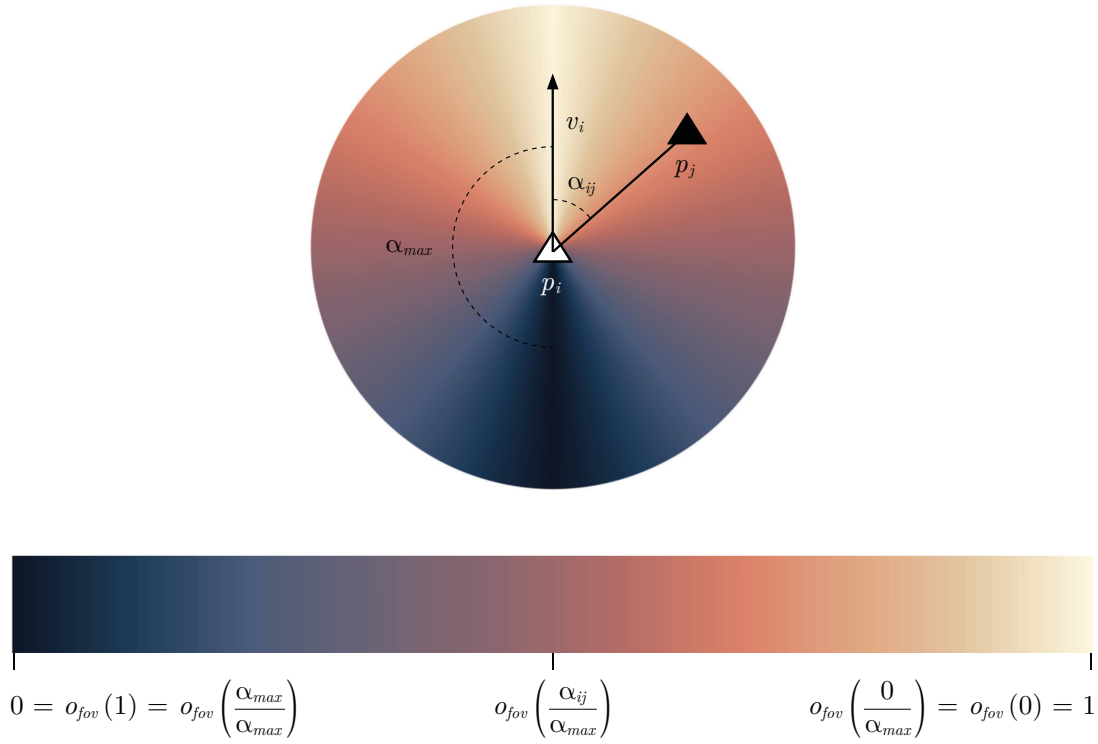

Figure 6.32: Visualization of angle based observability function as scalar field with gradient from blue at 0 to yellow at 1. Position and velocity of current boid $\vec{p_i}$, $\vec{v_i}$, neighbor's position $\vec{p_j}$, angle between them α_{ij}, maximum angle α_{max}.

For distance based observability, the function $f_{ip}(x)$ was suggested. We assumed that the case where the distance is 0 would likely never happen. Here, the case where the angle is 0 a normal situation, one where a boid is facing directly one of its neighbors. From our experimentation, we found that $f_{q2}(x) = 1 - x^l$ worked well for us. It was chosen over $f_{q3}(x) = (1-x)^l$, because this way, even angles relatively far from the center still have high influence. We think of it as the boid having almost equally good vision for close angles, which quickly falls off at the edge of the field of view. This effect can be observed in Figures 6.33 and 6.34, which visualize $f_{q2}(x)$ for different choices of l, assuming $fov_{max} = \pi$. Note that the influence of neighbors at angles close to 0 is higher, as l grows. Also, note that in the figures, if the angle α_{ij} is above $\frac{\pi}{2}$, the observability is 0.

Figure 6.33: Angle observability function f_{q2} for $l = 2$ and $fov_{max} = \pi$.

Figure 6.34: Angle observability function f_{q2} for $l = 3$ and $fov_{max} = \pi$.

Combining Observability Functions

Two types of observability functions were introduced, one for distance, and one for field of view. We want to combine the two observability functions into one, $o(d_{ij}, \alpha_{ij})$, which depends on both the distance d_{ij} and angle α_{ij}. The main idea behind observability functions was that they should go to zero as a neighbor leaves the boid's vision, either by getting too far, or out of the field of view. Therefore, $o(d_{ij}, \alpha_{ij})$ should give 0 when either of the functions gives 0. Naturally, the final observability should be higher, the more a neighbor is in front of the boid, and the closer it is. To satisfy these conditions, we define $o(d_{ij}, \alpha_{ij})$ as product of o_{fov} and o_{dist}. This is analogous to Boolean *AND* operator. This way, neighbors who are both close, and right in front of the boid, will have high influence. Furthemore neighbors have no influence, if either d_{ij}, or α_{ij}, is too high. The combined observability function we will use is the following:

$$o(d_{ij}, \alpha_{ij}) = o_{dist}\left(\frac{d_{ij}}{r_{max}}\right) o_{fov}\left(\frac{2\alpha_{ij}}{fov_{max}}\right)$$

For our behaviors, $o_{dist}(x) = f_{ip}(x)$, $o_{fov}(x) = f_{q2}(x)$. Both functions are illustrated in Figures 6.35, 6.36, and their product is shown in Figure 6.37. In the last figure, see that the product indeed has highest values for low d_{ij} and α_{ij}, and it is 0 when either d_{ij} or α_{ij} reaches its maximum.

Both of the chosen functions allow us to specify a parameter l, which shapes them. This constant can be set by the game designer to customize the behaviors. Based on our testing, setting $l = 2$ for both functions seemed to be a good default. We chose default value of 2 based on the inverse square or cube relationship suggested by Reynolds [7]. The motivation was mentioned when discussing distance based observability functions.

However, it is possible to try different values for different purposes. For example, to increase difference in influence for close neighbors, one would increase l of the $o_{dist}(x)$. On the other hand, increasing l of the $o_{fov}(x)$ would reduce the difference in influence for small α_{ij}.

Figure 6.35: Distance based observability function using f_{ip} for $l = 2$.

Figure 6.36: Angle based observability function using f_{q2} for $l = 2$ and $fov_{max} = 2\pi$.

Figure 6.37: Product of distance based observability function f_{ip} for $l = 2$ and angle based observability function f_{q2} for $l = 2$.

6.4.5 Neighbor Behaviors

In the previous subsections, the concepts which will help deriving the behaviors were introduced. Now, these concepts will be used to derive our cohesion, alignment and separation functions. The behaviors are based on the generalization from Subsection 6.4.2, and take on the following pattern: First some information weighted by observability is accumulated over all neighbors. This is used to determine one final desired direction $\vec{dir}$ and speed spd. Then, easing functions from Subsection 6.4.3 are used to assign direction desire des_{dir} and speed desire des_{spd}.

All behaviors below are defined from the perspective of the current boid b_c. For example, the current boid's position is p_c, and the distance between the current boid and its j-th neighbor is d_{cj}. Furthermore, for the sake of conciseness, normalized vectors are denoted using the "hat" notation, defined as: $\hat{v} = \frac{\vec{v}}{||\vec{v}||}$. Lastly, when analyzing the constraints and discussing the size of a result, we mean the result's desires as discussed previously in Subsection 6.4.1.

Cohesion

As is traditionally the case, our implementation also builds on the concept of centroid. However, here, it is calculated as a weighted mean of the neighbors' positions. The weights are given by $o(d_{cj}, \alpha_{cj})$. The desired direction is then direction towards this centroid. The direction desire grows with distance to the centroid using $f_{q1}(x) = x^l$. There is "no opinion" on the desired speed, so the speed and its desire are 0. The cohesion behavior is given by:

$$\boxed{\vec{dir} = \hat{a}, \qquad des_{dir} = w_{coh} f_{q1}\left(\frac{||\vec{a}||}{r}\right)} \tag{6.9}$$

$$\boxed{\vec{spd} = 0, \qquad des_{spd} = 0} \tag{6.10}$$

$$\vec{a} = \vec{p_{avg}} - \vec{p_c}$$

$$\vec{p_{avg}} = \frac{\sum_{\vec{b_j} \in N} o(d_{cj}, \alpha_{cj}) \vec{p_j}}{\sum_{\vec{b_j} \in N} o(d_{cj}, \alpha_{cj})}$$

where w_{coh} is an adjustable weight.

After a lot of experimentation, we settled on using f_{q1} for the activation function. This makes the desire smoothly fall off to zero as the boid gets closer to the centroid. Note, this implementation does not entirely satisfy what we wanted to achieve with the third constraint. If there is only one neighbor, then as it leaves the radius of vision r, the desire suddenly drops from maximum w_{coh} to 0. This can result in sharp change of the boid's movement, which we wanted to avoid. We tried to solve this issue by using one of the bell shaped functions for activation, then the activation function would give 0. However, this edge case rather rare (since the number of neighbors is usually larger than 1), and we found no success with this approach.

Constraint 1

The first constraint is satisfied. The function is smooth, small change in position of one neighbor will mean small change in result.

Constraint 2

The second constraint is satisfied. The value of w_{coh} and power l of f_{q1} can be adjusted. It is also possible to experiment with settings of the observability functions.

Constraint 3

The third constraint is satisfied. Neighbors loose influence on the result, as distance to them goes to r. However, there will be a discontinuity in desire if there is only a single neighbor leaving the radius of vision.

Constraint 4

The fourth constraint is satisfied. The maximum size of the result is w_{coh}, which is finite.

Constraint 5

The fifth constraint is satisfied. The maximum size of the result is w_{coh}, which is independent of the number of neighbors.

Constraint 6

The sixth constraint is satisfied. The maximum size of the result is w_{coh}, which is independent of the radius of vision.

Alignment

Same as with other implementations of alignment functions we explored earlier, our alignment is also based around the concept of average velocity. However, as with our cohesion, a weighted mean is used. Additionally, we calculate a weighted mean of directions and speeds separately. This avoids the issue where neighbors with higher speeds have more influence on the final direction. This issue was discussed when analyzing the first example implementation (Subsection 6.3.1). The desire for direction increases, the more the boid is misaligned with the desired direction. The desire for speed grows with magnitude of difference between the current and desired speed. The alignment behavior is given by:

$$\boxed{\vec{dir} = \hat{a}, \quad des_{dir} = w_{aliDir}\, f_{q1}\left(\frac{\theta(\hat{v_c}, \hat{a})}{\pi}\right)} \tag{6.11}$$

$$\boxed{spd = b, \quad des_{spd} = w_{aliSpd} f_{q1}\left(\frac{|b - \|\vec{v_c}\|\,|}{s_{max}}\right)} \tag{6.12}$$

$$\vec{a} = \sum_{b_j \in N} o(d_{cj}, \alpha_{cj})\hat{v_j}$$

$$b = \frac{\sum_{\vec{b_j} \in N} o(d_{cj}, \alpha_{cj})\, \|\vec{v_j}\|}{\sum_{\vec{b_j} \in N} o(d_{cj}, \alpha_{cj})}$$

where w_{aliDir} and w_{aliSpd} are adjustable constants, s_{max} is the maximum possible speed, and $\theta(\vec{v}, \vec{u})$ gives angle between two vectors. Activation function for both desires is f_{q1}. We also experimented with s-shaped curves, but found no large difference. As usual, the values passed into the activation functions are normalized to $[0, 1]$ range.

Constraint 1

The first constraint is satisfied. The function is smooth, small change in velocity of one neighbor will mean small change in result.

Constraint 2

The second constraint is satisfied. The values of w_{aliDir} and w_{aliSpd}, and powers l of the f_{q1} activation functions can be adjusted. It is also possible to experiment with settings of the observability functions.

Constraint 3

The third constraint is satisfied. Neighbors loose influence on the result, as distance to them goes to r. However, same issue as with cohesion applies.

Constraint 4

The fourth constraint is satisfied. The maximum sizes of the result are w_{aliDir} and w_{aliSpd}, both of which are finite.

Constraint 5

The fifth constraint is satisfied. The maximum sizes of the result are w_{aliDir} and w_{aliSpd}, both of which are independent of the number of neighbors.

Constraint 6

The sixth constraint is satisfied. The maximum sizes of the result are w_{aliDir} and w_{aliSpd}, both of which are independent of the radius of vision.

Separation

Our implementation of separation is quite similar to the one discussed earlier in Subsection 6.3.2. The desired direction is also determined as a weighted sum of directions from neighbors to the boid. As usual in our behaviors, the summed directions are weighted by result of the observability function. This makes sense conceptually, since neighbors that are closer, and positioned more directly in front of the boid, should be seen as a bigger threat of causing a collision.

The idea of seeing the observability as a measure of threat can be used for our activation function. The direction desire, des_{dir}, should grow with observability of the neighbors. However, there is one observability value per each neighbor. We considered working with the sum of the observabilities, but that sum depends on the total number of neighbors, so that could lead to unexpected results. Taking the average observability would avoid this issue, but the average would be too low in situations where one neighbor is very close, and the rest is very far. For this reason, we got inspired by implementation from Subsection 6.3.2, where the authors use the nearest neighbor to determine the behavior's weight. Similarly, we choose the highest observability out of all neighbors, denoted by u. However, the observability has no upper bound, but we want our activation function to give a normalized result in the $[0, 1]$ range, as discussed in Subsection 6.4.2. Therefore, an activation functions based on easing functions (Subsection 6.4.3) cannot be used here as before, since their inputs should be from $[0, 1]$ range.

The solution is to use an activation function with an asymptote at $y = 1$, as x goes to infinity. One such option is $f_{sep}(x)$ defined below and graphed in Figures 6.38, 6.39. It is a monotonically increasing function, with an asymptote $y = 1$ as x goes to infinity. Moreover, it gives 0 for $x = 0$, to make sure neighbors at the edge of vision have no influence. The function $f_{sep}(x)$ can be customized with two parameters, l and k. The effect of these parameters can be observed in the figures. Essentially, l makes the curve grow faster, and k "squashes" it vertically. Using the activation function $f_{sep}(x)$, our separation behavior is given by:

$$\boxed{\vec{dir} = \hat{a}, \qquad des_{dir} = w_{sep} f_{sep}(u)} \tag{6.13}$$

$$\boxed{spd = 0, \qquad des_{spd} = 0} \tag{6.14}$$

$$u = \max_{b_j \in N} o(d_{cj}, \alpha_{cj})$$

$$\vec{a} = \sum_{b_j \in N} o(d_{cj}, \alpha_{cj}) \frac{\vec{p_c} - \vec{p_j}}{||\vec{p_c} - \vec{p_j}||}$$

$$f_{sep}(x) = -\frac{1}{(kx+1)^l} + 1$$

where w_{sep} is an adjustable constant for weight, and k and l affect the shape of the activation function $f_{sep}(x)$.

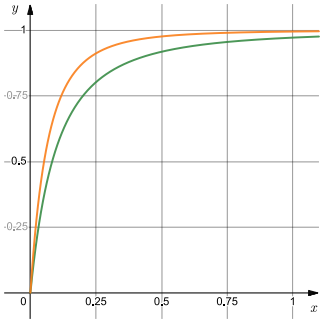

Figure 6.38: Activation function f_{sep} for $x \in [0, 1]$, where $k = 5$. Orange: $l = 3$, Green: $l = 2$

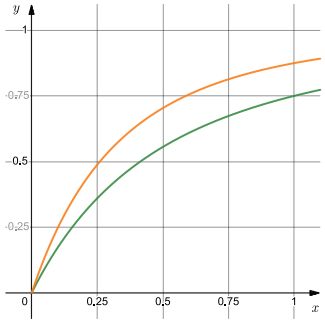

Figure 6.39: Activation function f_{sep} for $x \in [0, 1]$, where $k = 1$. Orange: $l = 3$, Green: $l = 2$

Constraint 1

The first constraint is satisfied. The function is smooth, small change in position of one neighbor will mean small change in result.

Constraint 2

The second constraint is satisfied. The value of w_{sep} and power l and constant k of f_{sep} can be adjusted. It is also possible to experiment with settings of the observability functions.

Constraint 3

The third constraint is satisfied. Neighbors loose influence on the result, as distance to them goes to r.

Constraint 4

The fourth constraint is satisfied. The maximum size of the result is w_{sep}, which is finite.

Constraint 5

The fifth constraint is satisfied. The maximum size of the result is w_{sep}, which is independent of the number of neighbors.

Constraint 6

The sixth constraint is satisfied. The maximum size of the result is w_{sep}, which is independent of the radius of vision.

Notes on Our Flocking Behaviors

The main parameter that can be adjusted on all the behaviors are the weights. Same as Reynolds in his first paper [7], we recommend ordering of $w_{sep} > w_{ali} > w_{coh}$. If the behaviors' results are merged using priority allocation (see Subsections 4.3.3, 4.4.1), the same ordering is recommend for priorities.

The powers l can shape the observability and activation functions. As mentioned earlier, at the end of Subsection 6.4.4, $l = 2$ seemed to generally be a good default. It seemed to be a generally good default for all powers l in the activation functions as well. This can also have a positive effect on performance, as $x * x$ is faster to run than invoking a mathematical library function to compute x^l, for a general floating point number l. For k used in $f_{sep}(x)$, a good default seemed to be $k = 0.01$. However, all of these constants were not chosen based on any objective measures, but rather what worked well for us when testing this algorithm. The behaviors can be customized further, by remapping the ranges of inputs or outputs with Equation 6.8. It will always be necessary to find the ideal combination of all of these hyper-parameters through experimentation. Furthemore, the choice of the activation and observability functions themselves could be another interesting aspect to experiment with.

There is one serious issue, if these three behaviors stood on their own. Consider a starting state, with multiple boids all having no speed. The only behavior that affects speed is alignment, which tries to match the speed of other neighbors. Therefore, the boids would not move anywhere. The wandering behavior in the next Subsection 6.4.6 solves this issue.

6.4.6 Other Behaviors

This section adds two more behaviors to our flocking model. As discussed, the boids would not move anywhere if only our cohesion, alignment and separation were used (end of Subsection 6.4.5). For this reason, a wandering behavior that drives a boid's desired speed is added. Furthemore, a homing behavior is added to keep the flock contained within a defined area. While the previous behaviors were what we refer to as *Neighbor Behaviors*, these two behaviors will not depend on any neighbors. We refer to these behaviors as *Simple Behaviors*. This distinction was discussed when extending the formalization of the Boids model in Subsection 2.1.1.

Wandering

The wandering behavior's main purpose is to drive the boids' desired speeds. Additionally, it makes the boids smoothly change direction over time. That makes sure that even a boid with no neighbors moves around in an interesting way.

Wandering is quite a common steering behavior that can be added to a flocking model. It was described, for example, by Reynolds in his paper on steering behaviors [16]. In the paper, he described an approach where the boid has sphere in front of it, and a point on the surface of the sphere is selected. Then, wandering direction is the direction from the boid to that point. Each frame, a random offset is added to this direction, which is then constrained back to the surface of the

sphere. However, he also mentions using Perlin Noise could be a good alternative. We decided to use Perlin Noise, as it is a commonly used in computer games, so users of the framework are likely to understand this approach easily.

To implement our wandering, two Perlin Noise functions are sampled over time, to give angles α and θ. They determine a random direction in spherical coordinates. Third noise function is used to sample a random speed. Our wandering function is given by:

$$\boxed{\vec{dir} = \vec{a}, \quad des_{dir} = w_{wanDir}} \tag{6.15}$$

$$\boxed{spd = b, \quad des_{spd} = w_{wanSpd}} \tag{6.16}$$

$$\vec{a} = (sin(\theta)cos(\alpha), cos(\theta), sin(\theta)sin(\alpha))$$
$$b = lerp(s_{min}, s_{max}, u)$$

$$\theta = \pi\, noise(2id, tf_{xz})$$
$$\alpha = \frac{1}{2}\pi + \alpha_{max}\, noise(id, tf_y)$$
$$u = \frac{1}{2} + \frac{noise(3id, tf_s)}{2}$$

where w_{wanDir} and w_{wanSpd} are adjustable constants. The variable α_{max} determines maximum angle of $\vec{dir}$ with the xz plane. This is useful to prevent boids from flying up at unnaturally high angle. The frequency for change in direction in the xz plane can be adjusted with constant f_{xz}. The frequency for change in the up and down direction can be adjusted with constant f_y. Frequency for change in speed is given by f_s. The function *noise* is a Perlin Noise function, which outputs values in range of $[-1, 1]$ with uniform probability. The boid's id, id, is used to seed the noise function. The variable t is the current time, and lastly, the function *lerp* is linear interpolation.

Note, the desired direction $\vec{dir}$ is in world space, which is why it is crucial that the noise function has a uniform probability distribution. Usually that is not the case for Perlin Noise implementations, which tend to be more biased towards 0. This would bias all boids towards some world space direction. In our implementation, we used a histogram equalization technique based on cumulative distribution function [56] (usually used for images), to find a function that converts non-uniform Perlin Noise into Perlin Noise with uniform distribution. This technique gives the function in terms of a lookup table. We wanted to avoid using a large lookup table in our implementation, so we fit the table using a polynomial of 5th degree (higher degress provided marginal accuracy improvements). Disclaimer, the core idea to use histogram equalization to find a uniform perlin noise is not our own. It was suggested to us by AI chatbot *ChatGPT* [57].

Homing

Currently, there is nothing stopping the boids from leaving an area where they are initialiazed. Having this feature would be very useful in context of games. For example, one might want to have a flock of birds above a specified area, and ensure they do not leave it. A homing behavior based on the idea that the boids have some "home" area can resolve this problem.

A homing behavior was proposed, for example, by Frank Heppner in his paper titled "A Stochastic Nonlinear Model for Coordinated Bird Flocks" [58], written in 1990. Note, the author uses homing as a core component of flocking, we will use it mainly to restrict the boids to a given area. In the paper, the author uses two radii, minimum and maximum, around a homing position. The "attractivness" of homing (in our case this would be desire), is 0 at either extreme, and reaches a maximum somewhere in between. For us, this would be like using a bell shaped activation function.

For our implementation, we use a similar idea. Desire is 0 when the boid is within some minimum radius, r_{min}, but it only grows as the boid reaches the maximum radius, r_{max}. This could result in a sharp change in the boids behavior, as it leaves the maximum radius. However, for us, the reason to use homing is to prevent the boids from leaving the homing area altogether, so ideally that case should not happen. If it does, the weight or priority should be increased to prevent it from happening.

Our homing is given by:

$$\boxed{\vec{dir} = \hat{a}, \quad des_{dir} = w_{home}\, f_{q1}(d)} \tag{6.17}$$

$$\boxed{\vec{spd} = 0, \quad des_{spd} = 0} \tag{6.18}$$

$$d = remap(r_{min}, r_{max}, 0, 1, ||\vec{a}||)$$

$$\vec{a} = p_h - p_c$$

where w_{home} is an adjustable constant, r_{min} and r_{max} are the minimum and maximum distance around the home at position p_h. When the boid's distance to the home is below r_{min}, it will stop desiring to go towards it.

6.5 Full Model

By now, all the components needed for our flocking model were discussed. That is the *Mover* (Chapter 3), *Merger* (Chapter 4), *Neighbor Query* (Chapter 5) and finally, this chapter discussed three *Neighbor Behaviors* and two *Simple Behaviors*. All the ideas can now be brought together to create a whole flocking model.

To query neighbors, each boid considers k nearest boids within some maximum distance and field of view, as discussed in Section 5.1. Our model then uses the previously proposed 5 behaviors – cohesion, alignment, separation, wandering and homing. Each behavior returns a constant priority, desired direction with a direction desire, and desired speed with speed desire. The behaviors' results are passed into a *Merger* described in Section 4.4, which uses weighted sum and

priority allocation to determine a single desired velocity. Finally, the desired velocity is passed into movement function from Subsection 3.5.1, which finds a new velocity of the boid based on its current velocity.

Naturally, the behaviors do not have to be limited to these five behaviors. A useful addition would be, for example, an obstacle avoidance behavior, using the concept of *Ray Behaviors*. One such obstacle avoidance behavior is described in Chapter 7.

7. Avoiding and Resolving Collisions

In Chapter 6, we introduced our complete model for flocking. It could be used, for example, for a flock of birds in a computer game. Using that model would be sufficient, because birds would likely not need to interact with the game's environment, since they are too high away from it. However, other animals, such as fish or sheep, might need to interact with the game's environment. For this thesis, we assume the game's environment to be defined by arbitrary meshes with colliders, since this is usually the case in Unity.

This chapter will discuss two aspects of interaction with the environment. Collision avoidance will involve *Ray Queries* and *Ray Behaviors*, which can be used to prevent collisions with the environment. Collision resolution will determine how a boid's movement changes, if collision avoidance fails. Since neither collision avoidance nor collision resolution is necessary for all animals, both these aspects will be covered only briefly.

7.1 Collision Avoidance

When adding other types of steering behaviors to our formalization of Boids model in Subsection 2.1.1, we introduced the concept of *Ray Queries* and *Ray Behaviors*, with the intent of using them for collision avoidance. The idea is that a *Ray Query* casts rays into the environment, and a *Ray Behavior* takes the ray hits and returns a result suggesting how to avoid collision.

7.1.1 Assumptions about Ray Queries

Our assumptions about *Ray Queries* will now be stated. The queries accept a boid with some data associated with it, cast several rays intersecting the environment, and return results of those ray casts. As usual, we assume that a boid would have at least a current position $\vec{p_c}$ and a current velocity $\vec{v_c}$. For each cast ray, we assume that commonly used data, such as its direction, as well as hit-related details like the normal, distance, and position, are provided by the physics engine.

The inputs are:

- $b_c = \{\vec{p_c}, \vec{v_c}\}$, the current boid's position and velocity.

The outputs are:

- $R = \{r_0, r_1, \ldots, r_n\}$, a set of rays cast into the environment.
  - If a ray r_i does not hit anything, then $r_i = \emptyset$.
  - If a ray r_i hits an object, it is represented as a tuple containing:
    1. The ray's direction $\vec{u_i}$.
    2. The hit's normal $\vec{n_i}$.
    3. The hit's position $\vec{h_i}$.
    4. The hit's distance d_i.

7.1.2 Assumptions about Ray Behaviors

Assumptions about *Ray Behaviors* will now be stated. They accept a boid which has some data associated with it, and result R of the *Ray Query*. They use this to return a result suggesting how the boid should move. For our behaviors, the return data would be the same as introduced in Subsection 4.4.2, and used for our behaviors in Chapter 6. For other authors' behaviors, the return data is usually a vector representing either a desired acceleration or a desired velocity.

- $b_c = \{\vec{p_c}, \vec{v_c}\}$, the current boid's position and velocity.
- $R = \{r_0, r_1 \dots r_n\}$, result of the *Ray Query*.

The outputs are:

- $q = \Big((\vec{dir}, des_{dir}), (spd, des_{spd}), p\Big)$ for our behaviors, where:
  1. $\vec{dir}$ is the desired direction,
  2. des_{dir} is desire for the direction,
  3. spd is the desired speed,
  4. des_{spd} is desire for the speed, and
  5. p priority.
- Alternatively, $\vec{v_d}$ or $\vec{a_d}$, the boid's desired velocity or desired acceleration, when discussing other authors' solutions.

7.1.3 Discussion of Other Implementations

While reading sources related to flocking, we found many approaches to implement *cohesion*, *alignment* and *separation*. However, we have not found many sources discussing approaches to collision avoidance in detail with their formal descriptions. For this reason a detailed analysis of other authors approaches like in Chapter 6 will not be done here.

Reynold's Implementations

One often cited source is Reynold's obstacle avoidance behavior from his paper about steering behaviors [16]. His illustration of the behavior is shown in Figure 7.1. For simplicity, he assumes all obstacles to be circles, but notes that the technique could be extended to other shapes. The behavior works in the following way. In the figure, the boid (green arrow) selects the first obstacle on its collision course (obstacle B). The behavior then returns a desired acceleration $\vec{a_d}$ (red arrow) in direction perpendicular to the boid's current velocity $\vec{v_c}$. In 2D, there are two directions perpendicular to $\vec{v_c}$. As shown in the figure, the direction pointing away from the sphere's center is selected (red arrow).

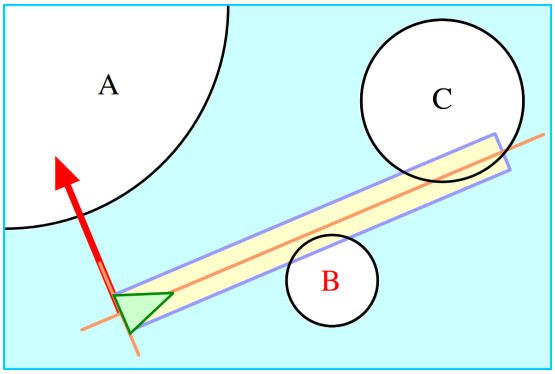

Figure 7.1: Reynolds' obstacle avoidance behavior. Figure from his paper about steering behaviors [16].

While this idea seemed promising at first, we identified a potential issue with accelerating by $\vec{a_d}$ perpendicular to $\vec{v_c}$. To illustrate the issue, consider a boid moving slowly in the direction of an obstacle. In each frame, an acceleration $\vec{a_d}$ is added to $\vec{v_c}$, resulting in a new velocity $\vec{v_n}$. Since $\vec{a_d}$ is perpendicular to $\vec{v_c}$, the magnitude of $\vec{v_n}$ will grow. Therefore, the initially slow-moving boid gains speed as it attempts to avoid the collision, which feels unrealistic. In our view, the growing speed could lead to other collisions.

Lague's Implementation

One approach we experimented with can be found in an open-source implementation of Boids in Unity by Sebastian Lague [10], [9]. His approach involves casting multiple rays into the environment, in a pattern shown in Figure 7.2. The pattern covers certain a maximum field of view FOV_{max} up to a maximum distance d_{max} where obstacles can be detected. For us, the pattern in which the rays are cast represents the *Ray Query*. Figure 7.3 shows some of the rays (red) detecting an obstacle.

The behavior then works in the following way. First, it checks for a potential collision ahead using a sphere cast. If there is a collision ahead, the behavior returns a desired velocity $\vec{v_d}$ in the direction of one of the rays that did not hit any obstacle. Among the rays that did not hit any obstacle, the one closest in angle to the boid's forward direction is selected. This ray is visualized in magenta in Figure 7.3.

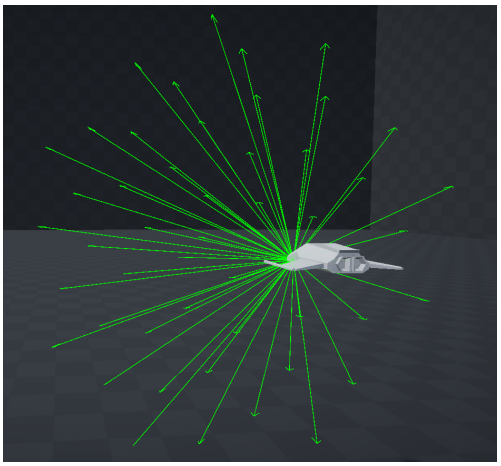

Figure 7.2: Sebastian Lague's pattern of ray casts for $FOV_{max} = \pi$.

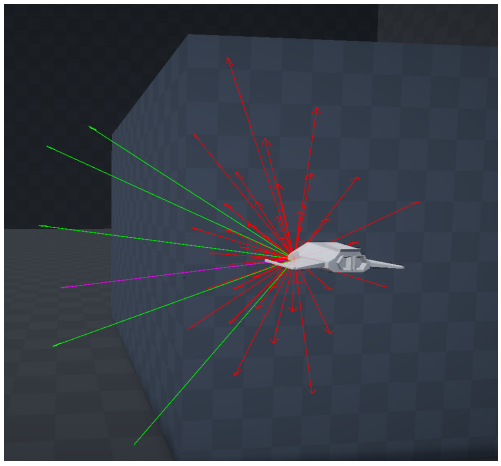

Figure 7.3: Sebastian Lague's pattern of ray casts for $FOV_{max} = \pi$. Red rays – hit, green rays – no hit, magenta ray – no hit and closest in angle to the boid's forward direction.

In our implementation, this approach was quite promising at first, but it was unclear how to handle a situation where all rays hit an obstacle. This meant that in some edge cases, a boid would stop trying to avoid collisions, since all the rays were hits. This could be mitigated by removing the limit on FOV_{max}, and cast rays in all directions around the boid instead. This potentially requires far more rays, which has a negative impact on the performance. Furthermore, it was difficult to decide on the optimal length of the rays d_{max}. Longer rays detect potential collisions sooner, giving the boid more time to react. However, larger d_{max} also increases the likelihood of encountering the edge case where all the rays hit something. Due to these issues, we did not pursue this approach further.

7.1.4 Our Implementation

For our implementation of obstacle avoidance, we use the same *Ray Query* as Lague used in his implementation described in the previous Subsection 7.1.3. His *Ray Query* is similar to the *Neighbor Query* that we assumed in our implementations of *Neighbor Behaviors*. In the same way, the boid's vision is limited by a maximum distance d_{max} and a maximum field of view FOV_{max}. For this reason, the same concept of observability (Subsection 6.4.4) from our *Neighbor Behaviors*, can be applied for our collision avoidance behavior. Here, the observability depends on distance to ray hit d_i, and the angle between the ray and the current boid's forward direction, α_i. For *Neighbor Behaviors*, observability considered distance to a neighbor and an angle between direction to the neighbor and the boid's current forward direction.

The implementation of our obstacle avoidance behavior is almost identical to our implementation of separation (Subsection 6.4.5), since separation's responsibility is essentially obstacle avoidance as well. The main idea in the separation behavior was to take a weighted sum of directions away from each neighbor. Similarly, the obstacle avoidance accumulates a weighted sum of normals $\vec{n}$ pointing away from the obstacles over all ray hits. The weights in this sum are given by

observability of each ray, just as the weights for separation were given by observability of each neighbor. Furthemore, the same activation function, $f_{sep}(x)$, is used to determine des_{dir}. The function $f_{sep}(x)$ receives u as parameter, where u is the highest observability over all rays. For separation, the highest observability over all neighbors was used. More formally, our obstacle avoidance behavior is given by:

$$\boxed{\vec{dir} = \hat{a}, \quad des_{dir} = w_{oa} f_{sep}(u)} \tag{7.1}$$

$$\boxed{spd = 0, \quad des_{spd} = 0} \tag{7.2}$$

$$u = \max_{r_i \in R,\, r_i \neq \emptyset} o(d_i, \alpha_i)$$

$$\vec{a} = \sum_{r_i \in R,\, r_i \neq \emptyset} o(d_i, \alpha_i)\vec{n_i}$$

$$f_{sep}(x) = -\frac{1}{(kx+1)^l} + 1$$

where w_{oa} is an adjustable constant for weight, and k and l affect the shape of the activation function $f_{sep}(x)$. How k and l affect $f_{sep}(x)$ was discussed in Subsection 6.4.5.

7.2 Collision Resolution

While the obstacle avoidance behavior described in Subsection 7.1.4 works quite well for avoiding collisions, it does not guarantee that all collisions will be avoided. For this reason, collisions with the environment need to be handled in some way, to ensure that the boids do not clip through the scene's colliders.

7.2.1 Options

There are some options to consider for collision resolution. One option in Unity or other game engines would be to make the boids dynamic rigidbodies and let the physics engine handle the rest. However, a dynamic body can fall over when hit, slide down on uneven ground, or bounce off walls. Rigidbodies are good for simulating inanimate objects, but a real animal would be able to keep its balance without falling over, stand without moving on steep ground and it would not bounce off a wall upon hitting it.

The second option would be to use a "character controller" which handles collisions. For example in Unity, there is a `CharacterController` [59] component for MonoBehaviors. There is also an experimental character controller for ECS [60]. However, upon testing it, we found it to be unnecessarily complex for our needs. We also found its performance to be insufficient. For these reasons, we decided to implement our own collision resolution algorithm, based on the commonly used "collide and slide" algorithm.

7.2.2 Collide and Slide

Collide and slide is a technique for collision resolution that can be used to build a character controller around. We were inspired to use this approach by a YouTube video [61] by Poke Dev, who used it to build a simple character controller in Unity. His implementation is based on a paper [62] by Kasper Fauerby, who introduced the technique for collision resolution of characters in 3D games. The video also references a paper [63] by Jeff Linahan who improved the technique to be numerically stable.

The main idea behind this algorithm is that when a character collides with a wall, it should slide along it. This is illustrated in Figure 7.4. The entity is represented by a sphere. In the figure, the sphere is supposed to move with velocity $\vec{v_c}$, but that would result in a collision. Therefore, the algorithm finds $\vec{v_1}$, the largest velocity to travel at before colliding. The rest of the unused $\vec{v_c}$ is then projected onto the collision surface, giving $\vec{v_2}$. The final new velocity to use is $\vec{v_n}$, the sum of $\vec{v_1}$ and $\vec{v_2}$. In case that $\vec{v_2}$ would result in a collision, the algorithm recursively again snaps the velocity to the collision surface, and projects the rest. This is repeated until the entire velocity is used up, or until some maximum number of iterations is reached.

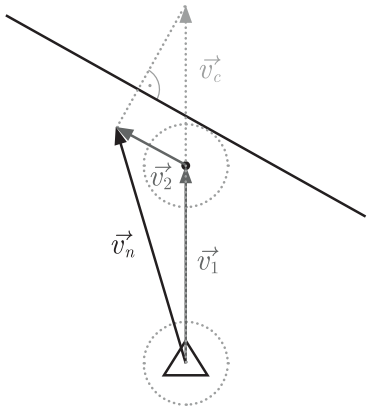

Figure 7.4: One iteration of collide and slide with current velocity $\vec{v_c}$. New velocity $\vec{v_n}$ is composed of $\vec{v_1}$ and $\vec{v_2}$.

Improvements

When experimenting with the basic collide and slide algorithm, we first attempted to reimplement the algorithm from the video [61] that inspired us to use this approach. However, our implementation suffered from edge cases, such as clipping through the walls and jittering. For this reason, we reimplemented the algorithm from Jeff Linahan's paper [63], which aims to solve numerical stability issues. The main idea behind this implementation is the same, with some small changes, and usage of techniques to improve the numerical stability. One notable improvement is that this algorithm does not need a "magic number" for maximum number of iterations. Instead, it uses at most three iterations, guaranteeing that clipping through geometry is not possible. This spares the user of having to balance quality of collision resultion and performance by manually setting a maximum number of iterations. A further benefit is that the performance would be more

unstable and unpredictable if a high maximum number of iterations was used.

The three iterations are the following: The boid first slides along the first wall as before. If there is a second wall, it slides along a vector which is parallel to both the first, and the second wall. If there is a third wall, the movement stops there. Especially the second step is important. Using the classical collide and slide, jittering can happen when moving into a corner. This is because in each iteration, the boid could slide along one wall, onto the second one and then back to the first one and so on. Restricting the movement to a vector parallel to both walls in the second step avoids this.

Additionally, the paper explains how to find the normal of the wall while handling edge cases like colliding with a sharp edge. Furthermore, it describes how to prevent edge cases where the sphere moves too close to the collision point, which could make the next sweep cast detect that position as a collision due to floating-point errors. It uses the concept of a "skin width" to prevent this. For implementation details, see the referenced paper [63].

8. DOTS and ECS Background

In Chapter 1, the Unity's Data Oriented Technology Stack – DOTS [12] was chosen to implement the framework. It was relatively recently introduced into Unity in 2018 and consists of a collection of technologies designed to improve performance. There are three main parts of DOTS, *Burst Compiler* [13], *Job System* [64] and *ECS* [65].

The *Burst Compiler* enables compilation of C# into optimized native code. However, it comes with many restrictions, such as being able to compile only unmanaged types. In practice, this means that only structs, not classes, can be burst-compiled. For this reason, common data structures like `T[]` arrays or `List<T>` lists cannot be used. Instead, Unity provides an unmanaged array, `NativeArray<T>` [47], where all elements must be unmanaged types, and nested arrays are not supported.

Job System enables utilizing all CPU cores through parallelization. Code inside a job's `Execute`[1] method can be scheduled to run in parallel. The parallelization can be twofold, jobs can invoke `Execute` multiple times in parallel and multiple jobs can run in parallel to each other. Additionally, jobs can be burst-compiled. For synchronization, each job, `A`, provides a `JobHandle` that can be used as a dependency, making sure that another job, `B`, waits for `A` to complete, before it begins execution.

ECS, which stands for Entity Component System, is a paradigm for game development, different to the traditional OOP-based approach. Its main concepts relevant to this thesis are explained in the list below.

- **Entity** [66] – Represents a single object in the game, for example a boid. Entities have a set of components associated with them. In practice entity is only an index into arrays of components.
- **Component** [67] – A piece of data associated with an entity. Components are structs that can contain only unmanaged types, and usually contain no logic, only data. They can, for example, be used to configure each boid, or contain a boid's current velocity.
- **Tag Component** [68] – A component with no properties. Tag components can be used to uniquely identify entities. For example, all entities representing birds can be marked with a bird tag.
- **Archetype** [69] – All entities sharing the exact same set of component types are said to belong to the same archetype. For example, all entities with exactly these three components: position, velocity, and a bird tag, would share the same archetype.
- **Entity Query** [70] – A query used to look up all entities with certain restrictions on their components. For example, any entity with these three components: position, velocity, and a bird tag.

[1] Usually the name of the method is `Execute`, but it can be named in a different way.

- **System** [71] – Contains logic which processes and updates entities' components. Usually, a system looks up entities with certain components using an entity query, and then performs operation on them, possibly by running parallel jobs.

9. Framework Implementation Documentation

In Chapter 2, we presented formalization of a flocking algorithm by breaking it down into main four parts, *Movers*, *Mergers*, *Queries*, and *Behaviors*. In the following Chapters 3, 4, 5, 6 7, implementations of these functions were discussed. While the concrete implementations from different sources varied significantly, the core structure of the algorithm, described by the formalization, remained the same.

A framework based on the formalization would allow modular interchaning of concrete implementations, such as a specific *Neighbor Query* or *Neighbor Behavior*, without having to make changes to rest of the algorithm. That would, for example, make it easy to implement and use a new behavior, such as a new cohesion behavior. However, most users would not want to write their own behaviors. Therefore, the framework should also include good default implementations that the user can use. Moreover, the framework should provide a user friendly GUI, that allows the user to set up a flocking system without writing any code. There, the user could configure their own system, for example, by selecting a set of implementations of *Neighbor Behaviors* to use. This is especially useful for computer game development, to allow even non-technical team members to work with the framework.

This chapter will describe the implementation of a framework that has these features. The implementation is done in Unity using their Data Oriented Technology Stack (DOTS) [12]. This technology was chosen in Section 1.2, based on analysis of other implementations of flocking in Unity. For terminology related to DOTS, see Chapter 8.

9.1 Requirements

So far, we outlined what the framework should be able to do. The requirements for the framework were described more concretely in Section 1.1. The following list summarizes these requirements in the context of previous chapters. The framework will be implemented such that the requirements are satisfied as well as possible.

1. **Performance** – Maximum 1 millisecond to run a flocking simulation of 1000 boids on reference system described in Subsection 1.1.1.

2. **Modularity** – Support for swapping different implementations of *Mover*, *Merger*, *Neighbor Queries*, *Neighbor Behaviors*, *Ray Queries*, *Ray Behaviors* and *Simple Behaviors* (Subsection 1.1.2).

3. **Extensibility** – Possibility to easily add new implementations of *Mover*, *Merger*, *Neighbor Queries*, *Neighbor Behaviors*, *Ray Queries*, *Ray Behaviors* and *Simple Behaviors* (Subsection 1.1.3).

4. **Flexibility** – Ability to support different design choices like what information a behavior receives and returns, or how movement is handled (Subsection 1.1.4).

5. **User-Friendliness** – Provide default implementations of *Mover*, *Merger*, *Neighbor Queries*, *Neighbor Behaviors*, *Ray Queries*, *Ray Behaviors* and *Simple Behaviors* that can be used without programming through GUI (Subsection 1.1.5).

9.2 Framework Design

Based on the requirements discussed again in Section 9.1, the framework must essentially be divided into two main parts. The *core* ensures the modular design based on the formalization from Chapter 2. It provides interfaces such as the concept of a *Neighbor Behavior*, together with a system to run their concrete instances. It also contains code that simplifies implementation of these interfaces. The *defaults* provide some concrete implementations of the interfaces, for example a cohesion behavior. The GUI for working with the framework will be described in Chapter 10. The framework's source files are located in `com.o-vaic.steering.ai/Runtime/Core` and `com.o-vaic.steering.ai/Runtime/Defaults` after importing the framework (see Attachment A.1.1).

We will begin with a high level overview of the *core*. The *core* gives a modular structure and manages the data flow, parallelization, and memory management in the background. It is designed to be as general as possible, while still doing large portion of the work for the user. Figure 9.1 shows a diagram of the *Base System*, which integrates all the key interfaces. In terms of DOTS, it is an ECS system that schedules a parallel pipeline of jobs, where jobs pass information between one another.

The *Base System* does the following. First, it queries all relevant entities using `EntityQuery`, named *Main Entity Query* in the figure. Then it passes information about the entities into three pipelines, one per each behavior kind – *Simple Behaviors*, *Neighbor Behaviors* and *Ray Behaviors*. *Simple Behaviors* are for behaviors like wandering, whose result depends only on the current boid. Wandering was discussed in Subsection 6.4.6. *Neighbor Behaviors* are for behaviors like cohesion, alignment and separation, discussed extensively in Chapter 6. Their result depends on the current boid and its neighbors. At last, *Ray Behaviors* can be used for behaviors like obstacle avoidance, discussed in Section 7.1. Their result depends on the current boid and its ray casts.

To avoid having to determine neighborhood or ray casts for each *Neighbor Behavior* and *Ray Behavior*, the behaviors can be grouped by their queries. For example, in Figure 9.1, an array of *Neighbor Behaviors* depends on a single *Neighbor Query*, which in turn depends on an *Entity Query* responsible for querying potential neighbors. To allow having multiple different *Neighbor Queries* or *Ray Queries*, the framework supports having multiple groups of *Neighbor Behaviors* and *Ray Behaviors*.

Once all behaviors are finished, *Merger* merges their results together into one final result (normally a desired velocity), which is used by the *Mover System* to update the boid's current velocity. Optionally a *Collision System* can run at the

end to resolve collisions.

As mentioned earlier, the parts of *Base System* that the user can implement and replace are defined in terms of interfaces. Namely, *Merger*, *Simple Behavior*, *Neighbor Behavior*, *Ray Behavior*, *Neighbor Query* and *Ray Query* are interfaces, and their implementations are parallel jobs. The parallelization is twofold: the jobs run in parallel to other jobs, while each job internally parallelizes computation per each entity. The *Base System* is set up to be agnostic to concrete implementations of the jobs, it only schedules them and passes information between them. The data flow and dependencies between the jobs are indicated by the arrows in Figure 9.1. The user can configure what set of jobs to run using `SteeringSystemAsset`, described in Subsection 9.3.1.

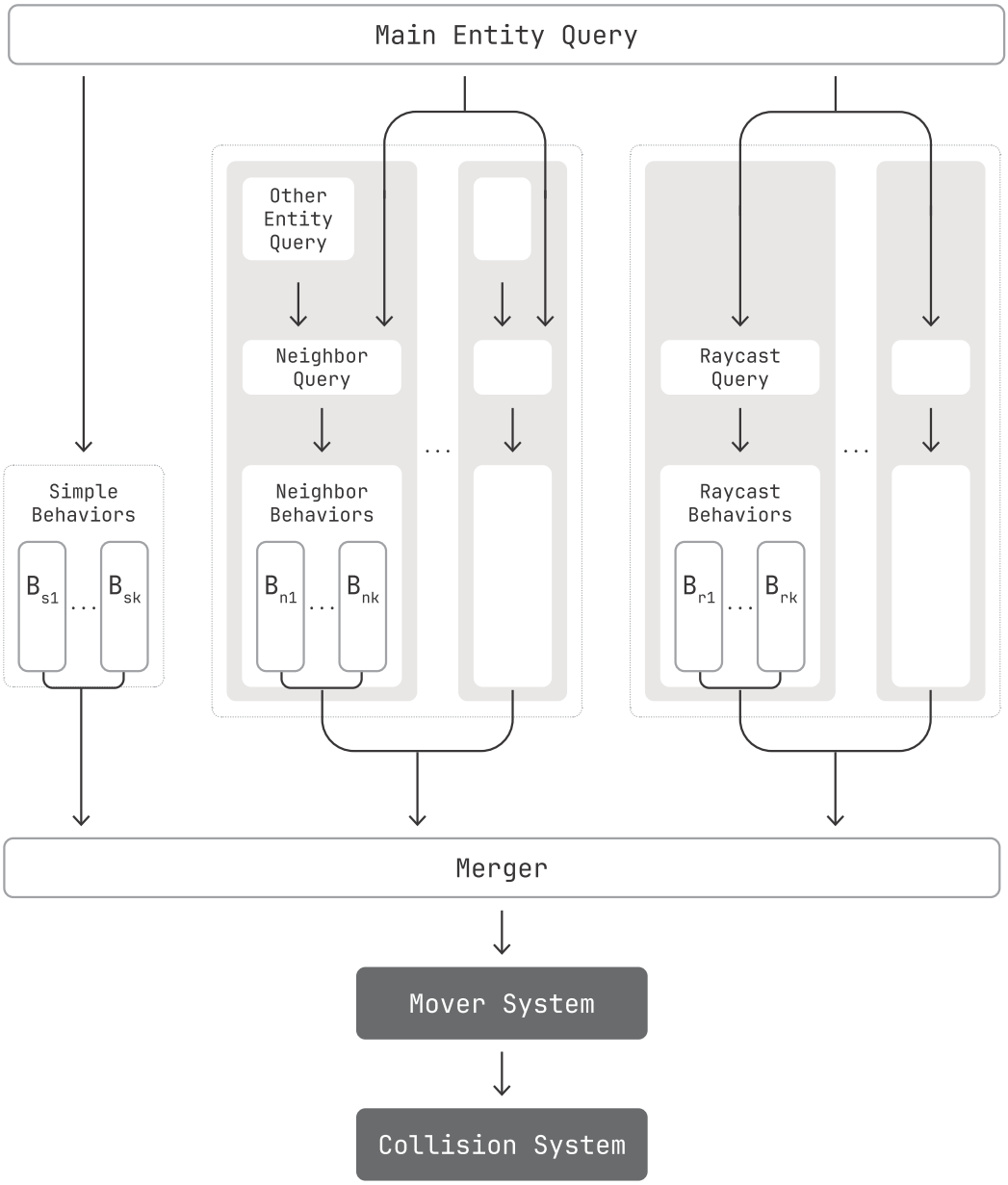

Figure 9.1: Diagram of the *Base System* of the framework.

9.2.1 Jobs and Job Wrappers

The core idea behind the framework design in Figure 9.1, is that it is possible to configure the *Base System* by giving it an array of jobs, which implement some interface, for example an array of *Simple Behaviors*. As far as we know, this requires a small workaround, because instance of a job needs to be scheduled through the `Schedule` extension method. This method only becomes avaliable for the concrete job structs. However, in order to have an array of jobs, the concrete types would have to be hidden behind an interface.

A simple workaround is to wrap the scheduling of a concrete job type into a class implementing some shared interface, and have an array of these wrapper classes instead. We will call these classes *Job Wrappers*. An example of this concept is shown in Listing 9.1. The snippet shows a job named `MyJob`, scheduled from inside of class `MyJobWrapper`, which implements `IFloatArrayJobWrapper` interface. In this case, it would be possible to have an array of `IFloatArrayJobWrapper`, and call `Schedule` on every element.

The whole *Base System* is composed of these *Job Wrappers*, which implement some shared interface, for example `ISimpleBehaviorJobWrapper` for *Simple Behaviors*. A potential downside of this is that the *Base System* has to use instances of classes so it cannot be burst compiled. This is not an issue, because the jobs themselves can still be burst compiled, which is where all the computation is done. One benefit is that the schedule method of the *Job Wrappers* can contain additional set up, or it could even schedule a chain of multiple jobs.

Listing 9.1: Example of a Job Wrapper.

```
struct MyJob : IJobParallelFor
{
    public NativeArray<float> Data;
    public void Execute(int index)
    {
        float a = Data[index];
    }
}

class MyJobWrapper : IFloatArrayJobWrapper
{
    public JobHandle Schedule(NativeArray<float> data, JobHandle inDependency)
    {
        var outDependency = new MyJob { Data = data }.Schedule(data.Length, 1,
            inDependency);
        return outDependency;
    }
}
```

```
interface IFloatArrayJobWrapper
{
    public JobHandle Schedule(NativeArray<float> data, JobHandle inDependency);
}
```

9.3 Base System and Relevant Types

Section 9.2 outlined the structure of the framework, which is centered around the *Base System*, illustrated in Figure 9.1. This section will explore this further, from perspective of how this is set up in C#.

The *Base System* is an abstract class named `BaseSteeringSystem`, inheriting from Unity's `SystemBase`. Concrete implementation of `BaseSteeringSystem` is a system for one type of animal. The user only has to override a method that returns path to load a `SteeringSystemAsset` (described in Subsection 9.3.1). This asset is a `ScriptableObject` containing serialized *Job Wrappers* of different types, all of which will be discussed in Subsection 9.3.4.

The system first queries all entities with a specified tag component provided by `SteeringSystemAsset`. This is the *Main Entity Query* in Figure 9.1. After finding all these entities, it caches some commonly used data from their components, like their positions and velocities, into `BaseBehaviorParams` struct, described in Subsection 9.3.2. The system schedules the *Job Wrappers* as shown in Figure 9.1 and passes the `BaseBehaviorParams` into them. The rest of this section is focused on the contents of `BaseBehaviorParams`, `SteeringSystemAsset` and types of *JobWrappers* which can be implemented.

9.3.1 Steering System Asset

The `SteeringSystemAsset` is a `ScriptableObject` that declaratively specifies behaviors to run, along with queries to use. All of this information is serialized in editor and loaded at runtime by an implementation of `BaseSteeringSystem`. An instance of `SteeringSystemAsset` can be editted in an editor window (see Chapter 10). The `SteeringSystemAsset` holds the following information:

- **`SimpleBehaviorJobWrappers`** – An array of *Simple Behaviors* to run.
- **`NeighborBehaviorGroups`** – An array of neighbor behavior groups. Each group contains an array of *Neighbor Behaviors* to run, grouped by a *Neighbor Query*.
- **`RaycastBehaviorGroups`** – An array of ray behavior groups. Each group contains an array of *Ray Behaviors* to run, grouped by a *Ray Query*.
- **`MergeJobWrapper`** – The *Merger* which turns results from all behaviors into a final result, suggesting where each entity wants to go.
- **`MainTagComponentType`** – The type of tag component used for the *Main Entity Query*.

To illustrate contents of the `SteeringSystemAsset`, see Listing 9.2. This is how creating an instance of a `SteeringSystemAsset` would look like in code, rather than in the editor. Note, the actual implementation of `SteeringSystemAsset` class is slightly different, because of code related to the editor. The sample below only illustrates the class' main structure, and the main idea behind it. The concrete implementations in the sample, such as `CohesionJobWrapper`, are from the framework's *defaults* (See 9.5).

The asset in the sample defines a system for sheep (`MainTagComponentType`), which exhibit flocking with other sheep (first `NeighborBehaviorGroup`), avoid wolves (second `NeighborBehaviorGroup`), avoid obstacles (`RaycastBehaviorGroup`), and wander around (`SimpleBehaviorJobWrappers`). Implementation of merging the behaviors results is assigned to the `MergeJobWrapper` property. A more detailed breakdown of the set up is described below the snippet.

Listing 9.2: Example of a `SteeringSystemAsset`.

```
new SteeringSystemAsset {
    MainTagComponentType = typeof(SheepTagComponent),
    MergeJobWrapper = new MergeVelocitiesJobWrapper(),
    SimpleBehaviorJobWrappers = new ISimpleBehaviorJobWrapper[] {
        new WanderJobWrapper(),
    },
    NeighborBehaviorGroups = new NeighborBehaviorGroup[] {
        new NeighborBehaviorGroup {
            BehaviorJobWrappers = new INeighborBehaviorJobWrapper[] {
                new CohesionJobWrapper(),
                new AlignmentJobWrapper(),
                new SeparationJobWrapper(),
            },
            NeighborQueryJobWrapper = new KDTreeKNNJobWrapper(),
            EntityQueryDesc = new EntityQueryDesc {
                Any = new ComponentType[] {
                    typeof(SheepTagComponent),
                },
            },
            NeighborhoodSettings = new NeighborhoodSettings
            {
                MaxNeighborDistance = 10,
                MaxNumNeighbors = 10,
                MaxFOV = 270,
            }
        },
        new NeighborBehaviorGroup {
```

```
            BehaviorJobWrappers = new INeighborBehaviorJobWrapper[] {
                new FleeJobWrapper(),
            },
            NeighborQueryJobWrapper = new KDTreeKNNJobWrapper(),
            EntityQueryDesc = new EntityQueryDesc {
                Any = new ComponentType[] {
                    typeof(WolfTagComponent),
                },
            },
            NeighborhoodSettings = new NeighborhoodSettings
            {
                MaxNeighborDistance = 20,
                MaxNumNeighbors = 1,
                MaxFOV = 360,
            }
        },
    },
    RaycastBehaviorGroups = new RaycastBehaviorGroup[] {
        new RaycastBehaviorGroup {
            BehaviorJobWrappers = new IRaycastBehaviorJobWrapper[] {
                new AvoidObstaclesJobWrapper()
            },
            CreateRaysJobWrapper = new ConeCreateRaysJobWrapper(),
            RaySettings = new RaySettings
            {
                MaxDistance = 10,
                NumRays = 10,
                LayerMask = 1 << 8,
            }
        },
    },
}
```

Listing 9.2 will now be described in more detail. First, **`MainTagComponentType`** tells the *Base System* to run the system for all sheep, that is entities with the `SheepTagComponent`. The tag is used by the *Main Entity Query* from Figure 9.1.

- **`MainTagComponentType`**: `typeof(SheepTagComponent)` – Run the *Base System* for all sheep.

Then, **`MergeJobWrapper`** is assigned. It specifies the implementation of a *Merger*, used to merge results from all behaviors after they are finished. Different methods

of achieving this were discussed in Chapter 4, users can implement and use their own approach.

- **MergeJobWrapper**: `MergeVelocitiesJobWrapper` – Merge behaviors' results into one desired velocity per sheep using this *Job Wrapper*.

Next property is **SimpleBehaviorJobWrappers**, an array of all *Simple Behaviors* to run for the sheep. Here, only the wandering behavior, discussed in Subsection 6.4.6, is used. The *Base System* passes infomation about all the main entities (sheep) as input into the behaviors, and the behaviors produce results, which are later passed into `MergeVelocitiesJobWrapper`.

- **SimpleBehaviorJobWrappers**
  - `WanderJobWrapper` – Sheep will wander around using this *Job Wrapper*.

Next there are two groups of *Neighbor Behaviors*. These behaviors were the main focus of Chapter 6. First group is set up for behaviors for flocking with other sheep. For each group, an entity query to find potential neighbors is ran first. In Figure 9.1, this corresponds to the *Other Entity Query*. The entity query here, configured by the **EntityQueryDesc** field, looks for all sheep using their tag. In this case, the *Main Entity Query* and the *Other Entity Query* look for the same entities.

After the information about the main entities (sheep) and the other entities (also sheep) is queried, neighborhood of the main entities can be queried. In Figure 9.1, querying neighborhood corresponds to a *Neighbor Query*. These queries were discussed in Chapter 5. The implementation of the *Neighbor Query* itself is assigned to the **NeighborQueryJobWrapper** field. Here, the `KDTreeKNNJobWrapper` uses a k-d tree to find k nearest neighbors restricted by maximum distance and maximum field of view. The parameters like maximum distance are configured in the **NeighborhoodSettings** field.

Finally, after the *Neighbor Query* is finished, information about all the sheep and their neighbors is passed into all behaviors of the **BehaviorJobWrappers** array. This corresponds to the *Neighbor Behaviors* in Figure 9.1. Here, there are three implementations of *Neighbor Behaviors*: cohesion, alignment and separation.

- **NeighborBehaviorGroup** – A group of *Neighbor Behaviors* to run.
  - **EntityQueryDesc** – All entities satisfying this entity query are potential neighbors.
    - **Any** – Entity must have at least one of the following components.
      - `typeof(SheepTagComponent)` – The entity query targets sheep.
  - **NeighborhoodSettings** – Settings for querying neighbors.
    - **MaxNeighborDistance**: `10` – Neighboring sheep must be within 10 units.
    - **MaxNumNeighbors**: `10` – Up to maximum 10 neighbors per sheep.
    - **MaxFOV**: `270` – Restrict each sheep's field of view to 270 degrees.
  - **NeighborQueryJobWrapper**: `KDTreeKNNJobWrapper` – *Neighbor Query* implementation for querying the neighbors.

  - **BehaviorJobWrappers** – An array of implementations of *Neighbor Behaviors* to run.
    - `CohesionJobWrapper` – Implementation of a cohesion behavior.
    - `AlignmentJobWrapper` – Implementation of an alignment behavior.
    - `SeparationJobWrapper` – Implementation of a separation behavior.

There is one more group of *Neighbor Behaviors*. It is used by the sheep to flee from wolves. The set up is very similar to the previous group. The main differences are the following. The wolves are queried using the `WolfTagComponent`, and the **NeighborhoodSettings** field is set up to look for only one neighbor, using a wider field of view and higher maximum distance. Furthemore, there is only one *Neighbor Behavior*, the `FleeJobWrapper`, which makes sure the sheep flee away from the wolf.

- **NeighborBehaviorGroup**
  - **EntityQueryDesc**
    - **Any**
      - `typeof(WolfTagComponent)` – The entity query targets wolves.
  - **NeighborhoodSettings**
    - **MaxNeighborDistance**: `20` – Wolves can be detected farther than sheep.
    - **MaxNumNeighbors**: `1` – At most 1 wolf can be detected.
    - **MaxFOV**: `360` – Use full field of view of 360 degress.
  - **NeighborQueryJobWrapper**: `KDTreeKNNJobWrapper`
  - **BehaviorJobWrappers**
    - `FleeJobWrapper` – Implementation of a flee behavior to escape the wolves.

At last, there is one group of *Ray Behaviors*. It contains one behavior, which is responsibile for collision avoidance. The set up is quite similar as for the *Neighbor Behaviors*. First, rays are cast into the environment, this corresponds to *Ray Query* in Figure 9.1. Here, the `ConeCreateRaysJobWrapper` creates rays in a conic pattern. The *Ray Query* is parametrized by the **RaycastSettings** field, which tells the query how far to cast the rays, how many rays to cast, and which physics layer to target. After the *Ray Query* is done, its results are passed together with information about the sheep into the *Ray Behavior* named `EnvironmentAvoidanceJobWrapper`.

- **RaycastBehaviorGroup** – A group of *Ray Behaviors* to run.
  - **RaycastSettings** – Settings for ray casting.
    - **MaxDistance**: `10` – Cast up to maximum 10 units away.
    - **NumRays**: `10` – Cast 10 rays.
    - **LayerMask**: `1 << 8` – Use 8th layer as a `LayerMask` for the ray casts.

- **`CreateRaysJobWrapper`** : `ConeCreateRaysJobWrapper` – Casts rays in a conic pattern using this implementation of a *Ray Query*.
- **`BehaviorJobWrappers`** – An array of implementations of *Ray Behaviors* to run.
  - `EnvironmentAvoidanceJobWrapper` – Implementation of an environment avoidance behavior.

9.3.2 Base Behavior Params

Each *Job Wrapper* used in Subsection 9.3.1 receives some information about the entities queried by the entity queries. For example, the wandering *Simple Behavior* receives information about all sheep queried by the system's *Main Entity Query*. Some data about the entities tends to be useful for most *Job Wrappers*, so it is cached by the framework into `BaseBehaviorParams` struct. The struct mostly contains arrays of components on the entities. The *Base System* can then pass this struct into each *Job Wrapper*. The data is queried either based on the *Main Entity Query*, or the *Other Entity Query* in case of *Neighbor Behaviors*. In code, a job named `FillBaseParamsJob` is used by the system to fill the `BaseBehaviorParams` for the user. Below is a list of the most important fields in the `BaseBehaviorParams` struct, to illustrate the idea.

- **`NativeArray<LocalToWorld> LocalToWorlds`** – Provides i-th entity's transformation matrix.
- **`NativeArray<VelocityComponent> Velocities`** – Provides i-th entity's current velocity.
- ... – Other commonly used components.
- **`NativeArray<ArchetypeChunk> ArchetypeChunks`** – Provides all `ArchetypeChunks` in the query. An `ArchetypeChunk` can be used to look up any component on an entity, in case the components are not cached in a different field of `BaseBehaviorParams`.

9.3.3 Behavior and Merger Results

Throughout the first part of the thesis, we discussed the importance of the type of results the *Behaviors* pass to the *Merger*. Additionally, one can consider types of results that the *Merger* passes to *Mover*. We call this the "workflow". Often, other authors simply use 3D vector representing a desired acceleration or a desired velocity, for both the *Behaviors* and *Mergers*.

Our behaviors (Chapters 6, 7), need to return more information. Namely the desired direction with direction desire, desired speed with speed desire, and a priority. Our *Merger* (Chapter 4), then uses these results to determine a desired velocity as a 3D vector, which is passed into the *Mover*. While this approach worked well for us, we do no want restrict more advanced users to our workflow. We want to enable them to define their own workflow, for their own *Behaviors* and *Mergers*, if necessary.

Behavior Results

Interfaces such as `ISimpleBehaviorJobWrapper`, used for *Simple Behaviors*, share their results with the rest of the system through `IDelayedDisposable` interface. This interface is almost identical to `IDisposable` interface from the `System` namespace in the .NET framework. The only difference is that the `Dispose` method also receives Unity's `JobHandle`, which should be awaited before disposing. This way, a user can create their own workflow for behaviors, by creating a type that implements the `IDelayedDisposable` interface. This would then enable a user to create their own behaviors that use this workflow.

For example, in the framework's *defaults*, each behavior creates and writes their results to a struct of type `VelocityResults`. The `VelocityResults` struct implements `IDelayedDisposable`, and contains a field of type `NativeArray<VelocityResult>`, where each element represents the behavior's result for an individual entity as `VelocityResult`. The `VelocityResult` struct contains: desired direction with direction desire, desired speed with speed desire, and a priority. For details about `VelocityResult` and `VelocityResults`, see Subsection 9.4.3.

Merger Results

After all behaviors are done, all of their results need to be merged into one final result for each entity. Therefore, *Merger's* interface, `IMergeJobWrapper`, receives an array of `IDelayedDisposable`, one per each behavior. Running a *Merger* is the last step of `BaseSteeringSystem`. For this reason, the final results that the *Merger* calculates are written directly into the entities' components.

For example, in the framework's *defaults*, the `MergeVelocitiesJobWrapper` receives an array of `IDelayedDisposable`. It casts each of them into `VelocityResults`. Then, for each entity, the *Merger* calculates one final result, and writes it into the entity's `DesiredVelocityComponent` component. This component contains a single 3D vector, specifying the entity's desired velocity. Later, a *Mover* system can read this component to decide how to update the entity's current velocity. In *defaults*, all *Movers* update velocity based on an entity's `DesiredVelocityComponent` component.

It is possible that a more advanced user of the framework would want their *Merger's* result to be more complex than `DesiredVelocityComponent`. In that case, the *Merger* could simply write any information necessary to a component of different type. However, the user would then have to implement their own *Mover* system, one that works with their component.

9.3.4 Job Wrappers

Section 9.2.1 described the concept of *Job Wrappers* and how they are used within the framework. In Subsection 9.3.1, their instances were used to create a sample `SteeringSystemAsset` for sheep. In code, each type of a *Job Wrapper*, like `WanderingJobWrapper` conforms to some interface. In this case, wandering is a *Simple Behavior*, and its interface is `ISimpleBehaviorJobWrapper`. This subsection describes all *Job Wrapper* interfaces used by the framework. A user can use these interfaces to, for example, implement a new behavior. Usually, the implementation of a *Job Wrapper* will schedule a single job, however, nothing restricts it

from scheduling multiple of them if needed.

Each *Job Wrapper* interface has only one method named `Schedule`. All `Schedule` methods share the following three input parameters: the `BaseBehaviorParams`, a `JobHandle` of a job that needs to run before it, and a reference to `BaseSteeringSystem` that scheduled the *Job Wrapper*. The `Schedule` method then returns a `JobHandle` as a dependency for the rest of the *Base System*. Additionally, each *Job Wrapper* is expected calculate some result per each entity as part of its contract. The `BaseSteeringSystem` then passes the results as input to the next *Job Wrapper* in the pipeline.

Note, all behaviors need to write their results into some struct implementing `IDelayedDisposable` (see Subsection 9.3.3). For sake of simplicity, while describing the *Job Wrappers*, we will assume that all behaviors write their results into the `VelocityResults` struct, which is used for all behaviors in *defaults*. Additionally, all the *Job Wrappers* that will be used as examples are from the framework's *defaults*. For more information about them, see Section 9.5.

Simple Behavior

As the name suggest, `ISimpleBehaviorJobWrapper` allows implementation of *Simple Behaviors*. Custom implementations of `ISimpleBehaviorJobWrapper` are useful for behaviors that do not depend on the entities' neighbors or raycasts, such as wandering. The interface is shown in Listing 9.3 below. Role of the interface for *Simple Behaviors* was already illustrated in Figure 9.1. Here, Figure 9.2 provides a closer overview with examples.

The figure shows m implementations of `ISimpleBehaviorJobWrapper`, marked with $SB_1 \dots SB_m$, where m-th behavior is the wandering behavior. As indicated with the tag in the *Main Entity Query*, the `BaseSteeringSystem` schedules the behaviors to run for all entities with a `SheepTag`. The tag and instances of the behaviors would be loaded from a `SteeringSystemAsset` (see Subsection 9.3.1).

At the top of the diagram, an `EntityQuery` is executed to find all sheep. Internally, the `FillBaseParamsJob` fills an instance of `BaseBehaviorParams`, named `MainBaseParams`, with information about sheep. The `MainBaseParams` contain useful information about the entities, such as a `NativeArray` of n positions, $[p_1 \dots p_n]$, one for each sheep. The `MainBaseParams` are passed into all the *Simple Behaviors*. Each behavior, for example, the `WanderingJobWrapper`, uses the information about entities to produce `Results`. As discussed in Subsection 9.3.3, `Results` could be any type implementing `IDelayedDisposable`, but normally, it would be an instance of `VelocityResults`. In that case, `Results` contains n instances of `VelocityResult`, one for each sheep – $[v_{r1} \dots v_{rn}]$. Each *Simple Behavior* creates one `Results` in this manner. Later, an array of m `Results` – one per each behavior – would be passed into the *Merger*.

In the figure, all black arrows show flow of dependencies for synchronization. This indicates that all behaviors run in parallel to each other. The grey arrows indicate the main data inputs and outputs. Usually, a *Job Wrapper* in the pipeline receives some input to work with, together with another job's dependency of type `JobHandle`. The job has to wait for the dependency before the input can be read. In the figure, `WanderingJobWrapper` receives `MainBehaviorParams`, together with a dependency from `FillBaseParamsJob`.

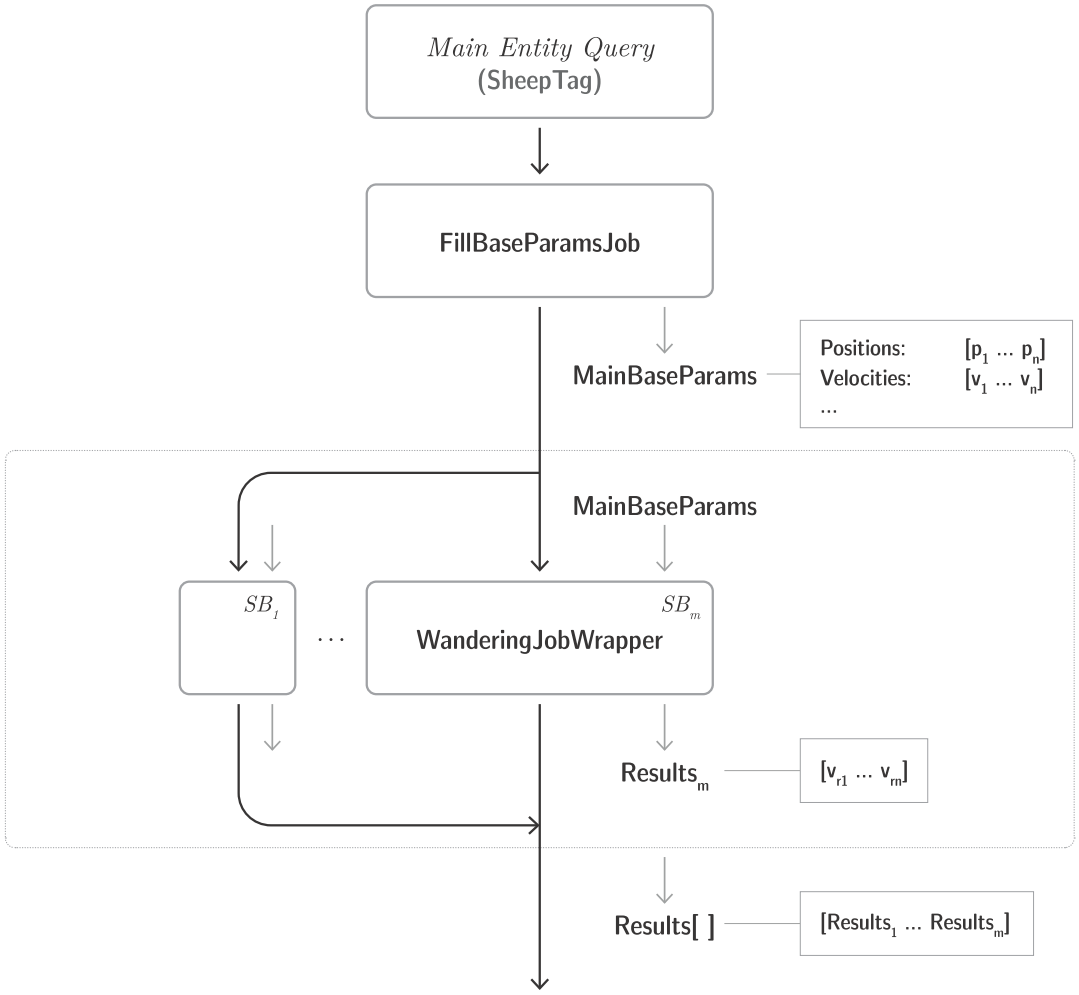

Figure 9.2: Diagram of the role of `ISimpleBehaviorJobWrapper` interface (*Simple Behavior*) in the *Base System*.

Figure 9.2 illustrated the role of implementations of `ISimpleBehaviorJobWrapper`, such as `WanderingJobWrapper`. The `ISimpleBehaviorJobWrapper` interface has a single method – `Schedule`, shown in Listing 9.3. The method receives: a reference to the system that scheduled it – `SystemBase`, a dependency it needs to wait for – `Dependency`, and information about the entities – `MainBaseParams`. It is expected to write output of the behavior through `out` parameter `Results`. The `Schedule` method returns a `JobHandle` that should be awaited before reading from `Results`. The `MainBaseParams` and the `Dependency` are based on the `FillBaseParamsJob`, which ran before it (see Figure 9.2). As discussed, `Results` could be any `IDelayedDisposable`, but normally it would be an instance of `VelocityResults`.

Listing 9.3: Interface of `ISimpleBehaviorJobWrapper`.

```
public JobHandle Schedule(
            SystemBase SystemBase,
            in BaseBehaviorParams MainBaseParams,
            out IDelayedDisposable Results,
            in JobHandle Dependency);
```

Neighbor Queries and Neighbor Behaviors

As already seen Figure 9.1, and in Subsection 9.3.1, *Neighbor Behaviors* are grouped into groups by *Neighbor Queries*. For example, in Subsection 9.3.1, the sheep used one group for flocking with other sheep, and a second group to flee from wolves. In that example, `FleeJobWrapper` (*Neighbor Behavior*) was responsible for the flee behavior, and the `KDTreeKNNJobWrapper` (*Neighbor Query*) was responsibile for finding their neighboring wolves. A new *Neighbor Query* can be implemented with `INeighborQueryJobWrapper` interface, and a new *Neighbor Behavior* can be implemented with a `INeighborBehaviorJobWrapper` interface. Both interfaces are shown in Listing 9.5. The pipeline tying these two interfaces together is illustrated in Figure 9.3.

The figure shows one group of *Neighbor Behaviors*. The group contains m implementations of `INeighborBehaviorJobWrapper`, marked with $NB_1 \dots NB_m$, where the m-th behavior – `FleeBehaviorJobWrapper` is a behavior for fleeing. The behaviors are ran for all sheep, as indicated with `SheepTag` in the *Main Entity Query*. Sheep use the behaviors to react to wolves, as indicated with `WolfTag` in the *Other Entity Query*. In this example, `KDTreeKNNJobWrapper` is the *Neighbor Query* responsible for finding neighboring wolves for all sheep. The tag for neighbors, instances of the behaviors, and an instance of the query are loaded from a `SteeringSystemAsset` (see Subsection 9.3.1).

The figure will now be described from top to bottom in more detail. Same as with *Simple Behaviors* from Figure 9.2, information about the sheep is queried into an instance of `BaseBehaviorParams` named `MainBaseParams`. This is the same instance that was passed into the *Simple Behaviors*. Inside the group of *Neighbor Behaviors*, another instance of `BaseBehaviorParams`, named `OtherBaseParams`, is created. It contains data about the wolves.

Once information about both types of entities is queried, the *Neighbor Query*, here `KDTreeKNNJobWrapper`, can start. It receives `NeighborQueryParams`, containing data about sheep and wolves, together with an instance of `NeighborhoodSettings`. These settings, already discussed in Subsection 9.3.1, are used for parameters of the *Neighbor Query*, like maximum distance or maximum number of neighbors. In the figure, k is used for maximum number of neighbors. The *Neighbor Query* determines a neighborhood for each sheep, and writes it into `Neighbors`. The format of the `Neighbors` array is expected to be a flat 2D array. That is, indexes $[0, k-1]$ contain k neighbors, $[u_{11} \dots u_{1k}, \dots]$, of the first sheep, next k entries contain neighbors of the second sheep, and so on until neighbors of the last sheep $[\dots, u_{n1} \dots u_{nk}]$. The neighbors are stored as `NeighborMatch` type, which mainly contains an index of the neighbor inside the arrays of `OtherBaseParams`.

After the neighborhoods are queried, the *Neighbor Behaviors* can start running. Each behaviors receives `NeighborBehaviorParams`. This struct contains data that was passed into the *Neighbor Query* – `NeighborQueryParams`, and `Neighbors` produced by the query. Each *Neighbor Behavior* uses this information to calculate its `Results`. The `Results` are normally of type `VelocityResults` (see Subsection 9.3.3), containing n results of a behavior, $[v_{r1} \dots v_{rn}]$, one result for each sheep.

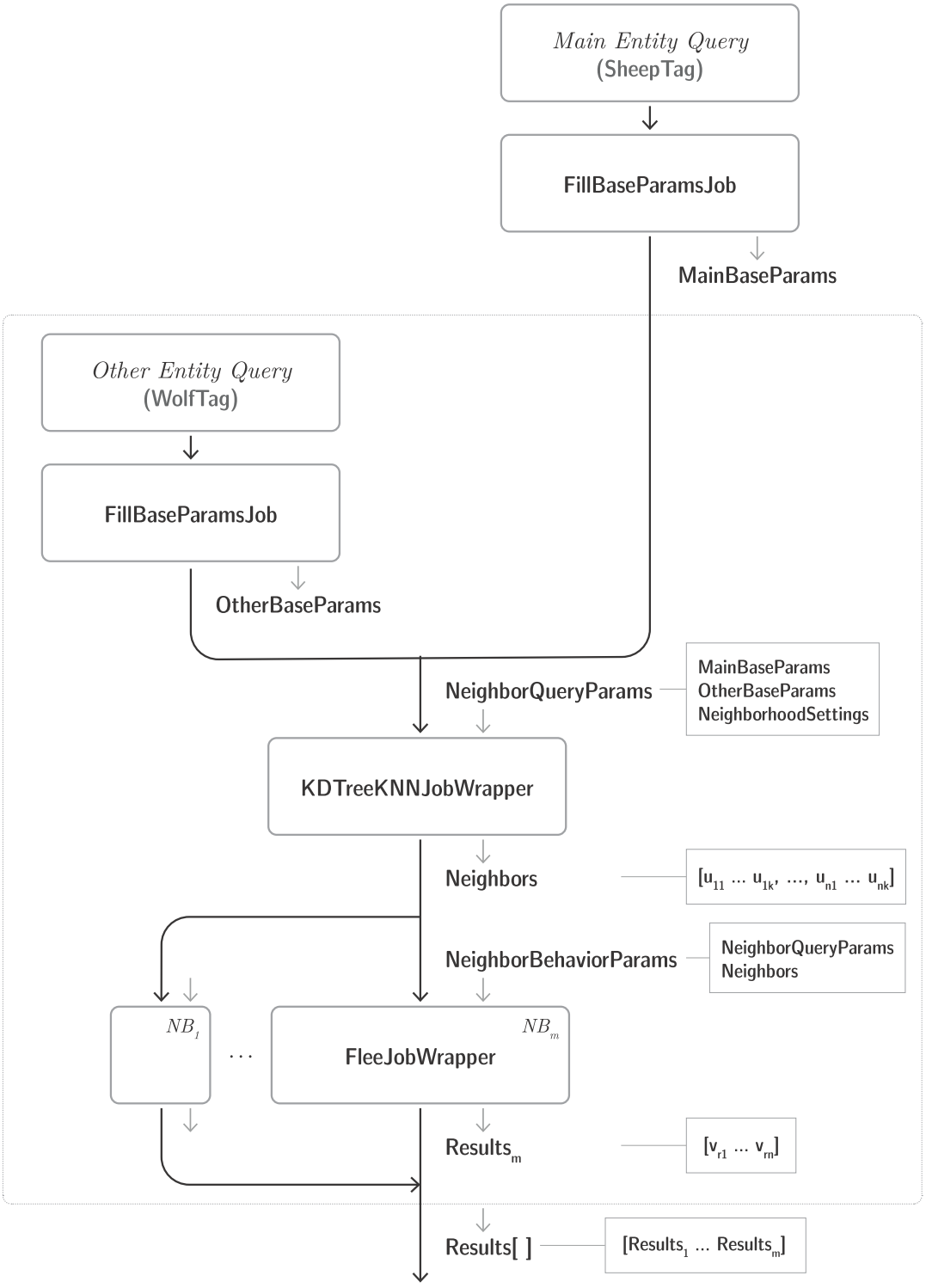

Figure 9.3: Diagram of the role of `INeighborQueryJobWrapper` interface (*Neighbor Query*) and `INeighborBehaviorJobWrapper` (*Neighbor Behavior*) in the *Base System*.

The `Schedule` method of `INeighborQueryJobWrapper` (see Listing 9.4) receives an instance of `NeighborQueryParams`. It also receives preallocated array, `Neighbors`, where it has to write the neighborhoods. It is expected to write the neighborhoods as a flat 2D array, as discussed above. As usual, `Schedule` also receives a `Dependency` to wait for, and returns a dependency to complete before reading from `Neighbors`.

Listing 9.4: Interface of `INeighborQueryJobWrapper` and types related to it.

```
public JobHandle Schedule(
    SystemBase SystemBase,
    in NeighborQueryParams NeighborQueryParams,
    in NativeArray<NeighborMatch> Neighbors,
    in JobHandle Dependency);

public struct NeighborQueryParams {
    public BaseBehaviorParams MainBaseParams;
    public BaseBehaviorParams OtherBaseParams;
    public NeighborhoodSettings NeighborhoodSettings;
}

public struct NeighborMatch {
    public int OtherIndex;
    ...
}
```

The `Schedule` method of `INeighborBehaviorJobWrapper` (see Listing 9.5) receives an instance of `NeighborBehaviorParams`. Same as when implementing a *Simple Behavior* using the `ISimpleBehaviorJobWrapper` interface, an implementation of `INeighborBehaviorJobWrapper` should allocate `Results` and write to them. As always, `Schedule` has an input `Dependency` to wait for, and an output dependency that should complete before reading from `Results`.

Listing 9.5: Interface of `INeighborBehaviorJobWrapper` and types related to it.

```
public JobHandle Schedule(
    SystemBase SystemBase,
    in NeighborBehaviorParams NeighborBehaviorParams,
    out IDelayedDisposable Results,
    in JobHandle Dependency);

public struct NeighborBehaviorParams {
    public NativeArray<NeighborMatch> Neighbors;
    public NeighborQueryParams NeighborQueryParams;
}
```

Ray Queries and Ray Behaviors

The *Ray Queries* and *Ray Behaviors* are set up in a very similar way to the *Neighbor Queries* and *Neighbor Behaviors*. The *Ray Behaviors* are grouped by a *Ray Query*, which they need to wait for, before producing their results. In the example in Subsection 9.3.1, sheep used `ConeCreateRaysJobWrapper` (Ray Query) to create rays querying information about colliders in the environment, and avoid the colliders using `EnvironmentAvoidanceJobWrapper` (Ray Behavior).

A *Ray Query* is implementation of `ICreateRaysJobWrapper` and a *Ray Behavior* is implementation of `IRaycastBehaviorJobWrapper`. For reference, see Listings 9.6, 9.7. Note, that as the name suggest, the `ICreateRaysJobWrapper` only creates the rays. The *Base System* then makes sure to actually cast them and pass the results further.

The Figure 9.4 shows one group of *Ray Behaviors*, where the main entities (sheep) use `ConeCreateRaysJobWrapper`, to create rays in a conic pattern. Results of these raycast are then used to avoid collisions with the environment using the `EnvironmentAvoidanceJobWrapper`. Same as with figures illustrating *Simple Behaviors* or *Neighbor Behaviors*, the group of *Ray Behaviors* first receives infomation about the sheep through an instance of `BaseBehaviorParams`, here named `MainBaseParams`.

The `MainBaseParams` are passed together with `RaySettings` into a *Ray Query*, here `ConeCreateRaysJobWrapper`. Instance of `RaySettings` determines parameters for the *Ray Query*, similarily to `NeighborhoodSettings` that determined parameters for a *Neighbor Query*. The settings contain, for example, which `LayerMask` to use, or how many rays to cast. In the figure, k is used to the number of rays. The *Ray Query* uses the `MainBaseParams` and `RaySettings` to create rays, and writes them into `RayDatas` array. Same as with `Neighbors` of *Neighbor Queries*, the rays should be written into `RayDatas` in a 2D flat array format – $[r_{11} \dots r_{1k}, \dots, r_{n1} \dots r_{nk}]$. The *Base System* then uses the `BatchRayCastsJob` to cast rays based on `RayDatas`, and writes the results of these raycasts into `RaycastHits` – $[h_{11} \dots h_{1k}, \dots, h_{n1} \dots h_{nk}]$.

Once `RaycastHits` array is filled, the *Ray Behaviors* can start running. They receive `RayBehaviorParams`, containing information about the sheep – `MainBaseParams`, information about the *Ray Query* – `RayData`, `RaySettings`, and information about the ray cast hits – `RaycastHits`. As usual, each *Ray Behavior* uses the information to produce `Results`. Normally, `Results` would be of type `VelocityResults` (see Subsection 9.3.3).

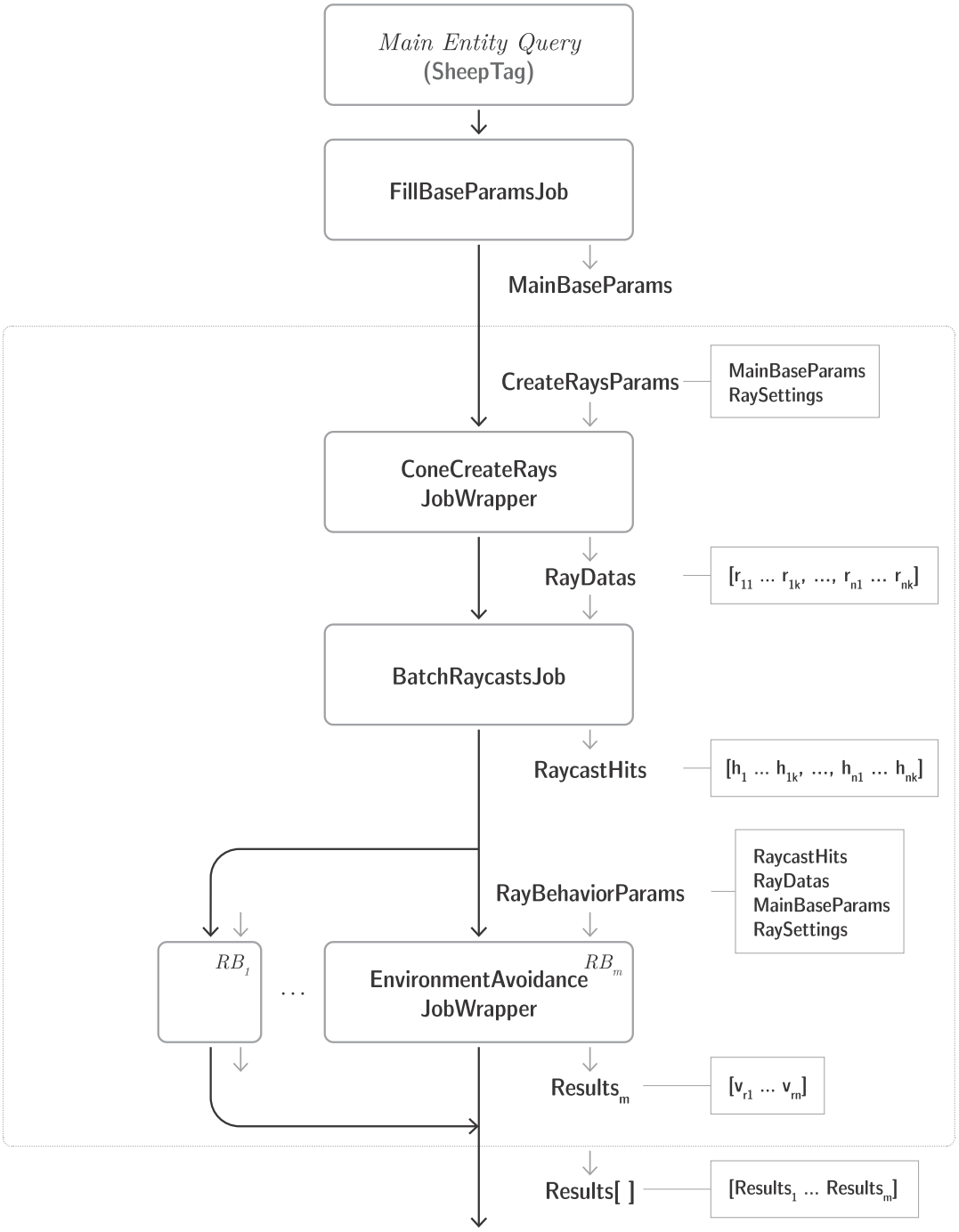

Figure 9.4: Diagram of the role of `ICreateRaysJobWrapper` interface (*Ray Query*) and `IRaycastBehaviorJobWrapper` (*Ray Behavior*) in the *Base System*.

The `Schedule` method of `ICreateRaysJobWrapper` (see Listing 9.6) receives an instance of `CreateRaysParams`. It also receives preallocated array, `RayDatas`, where it has to write the rays it wants to cast. It is expected to write the rays in a flat 2D array format, as discussed above. As always, there is an input `Dependency`, and an output dependency to complete before reading from `RayDatas`.

Listing 9.6: Interface of `ICreateRaysJobWrapper` and types related to it.

```
public JobHandle Schedule(
    SystemBase SystemBase,
    in CreateRaysParams CreateRaysParams,
    in NativeArray<RayData> RayDatas,
    in JobHandle Dependency);

public struct CreateRaysParams {
    public RaycastSettings RaySettings;
    public BaseBehaviorParams MainBaseParams;
}

public struct RayData {
    public float3 Origin;
    public float3 Direction;
    public float MaxDistance;
}
```

The `Schedule` method of `IRaycastBehaviorJobWrapper` (see Listing 9.7) receives `RaycastBehaviorParams`. Same as other behaviors, it allocates `Results`, and writes to them. As always, `Schedule` receives `Dependency` it should wait for before reading from `RayBehaviorParams`, and it returns a dependency that should complete before reading from `Results`.

Listing 9.7: Interface of `IRaycastBehaviorJobWrapper` and types related to it.

```
public JobHandle Schedule(
    SystemBase SystemBase,
    in RaycastBehaviorParams RayBehaviorParams,
    out IDelayedDisposable Results,
    in JobHandle Dependency);

public struct RaycastBehaviorParams {
    public NativeArray<RaycastHit> RaycastHits;
    public NativeArray<RayData> RayDatas;
    public BaseBehaviorParams MainBaseParams;
    public RaycastSettings RaySettings;
}
```

Merger

Once all `Results` arrays are filled by all the behaviors, the *Merger* can merge them together into one final result that specifies what each entity wants to do. For example, in Subsection 9.3.1, sheep used `MergeVelocitiesJobWrapper`, which is an implementation of algorithm described in Section 4.4. Merging can be implemented using the `IMergeJobWrapper` interface (Listing 9.8). Custom implementations of `IMergeJobWrapper` can be created to experiment with different merging strategies.

The role of the interface is illustrated in Figure 9.5. The figure uses the already mentioned *Merger*, `MergeVelocitiesJobWrapper`. A *Merger* receives `MainBaseParams` as usual, and an array of `Results` from all behaviors. In the figure, this is illustrated as `SB-Results[]` from all *Simple Behaviors*, `NB-Results[]` from all *Neighbor Behaviors* and `RB-Results[]` from all *Ray Behaviors*.

For each entity, the *Merger* should calculate one final result suggesting what the entity wants to do, and write it into some component on the entity. This could be any component on the entity, as discussed in Subsection 9.3.3. For example, `MergeVelocitiesJobWrapper` from the framework's *defaults* writes into a `DesiredVelocityComponent`. This component contains a 3D vector suggesting what velocity the entity wants to have. In the figure, this is illustrated as *Merger* writing into n components – $[v_{d1} \dots v_{dn}]$. The component can later be used by the *Mover* system to update the entity's current velocity. All *Movers* in the *defaults* of the framework (Subsection 9.5.7) use `DesiredVelocityComponent`.

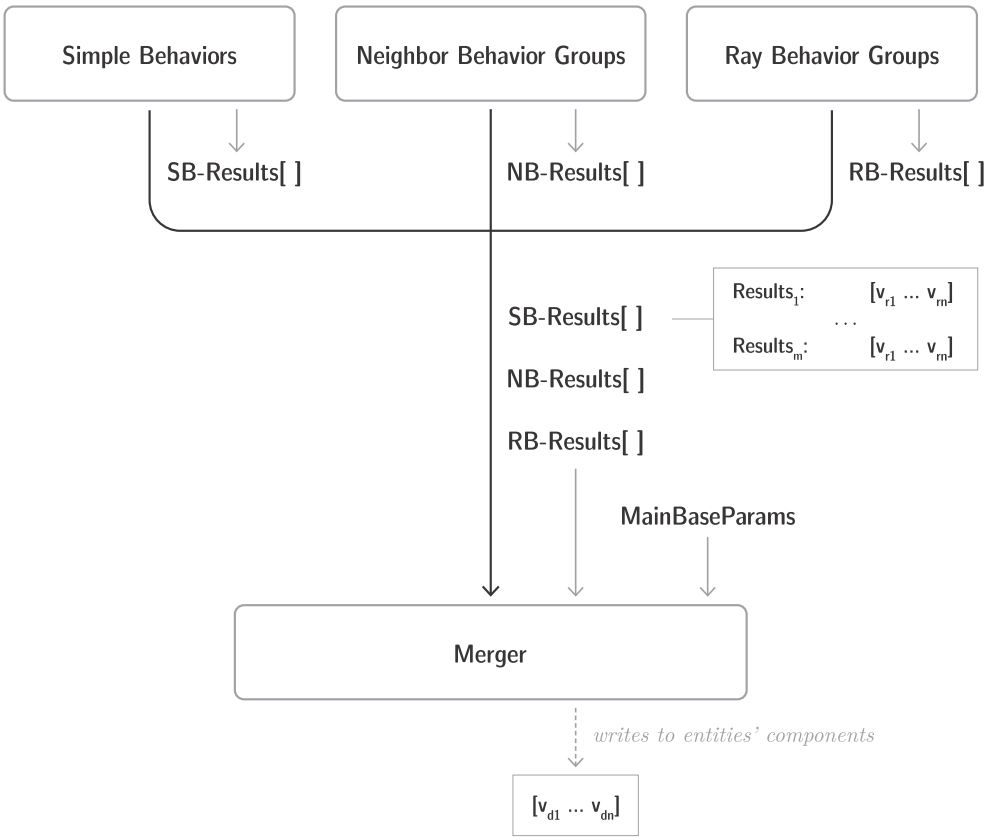

Figure 9.5: Diagram of the role of `IMergeJobWrapper` interface (*Merger*) in the *Base System*.

As usual, the `Schedule` method of `IMergeJobWrapper` (see Listing 9.8) receives `MainBaseParams`. Additionally, it receives an array of `IDelayedDisposable`. It contains `Results` from all the behaviors that were scheduled. An implementation of this *Job Wrapper* should write the final result into a component on each entity. Normally this component would be of type `DesiredVelocityComponent`. This is the last step of the *Base System*, a *Mover System* can later look up the components, and update the boid's current velocity accordingly.

Listing 9.8: Interface of `IMergeJobWrapper`

```
public JobHandle Schedule(
    SystemBase system,
    in BaseBehaviorParams MainBaseParams,
    in IDelayedDisposable[] Results,
    JobHandle Dependency);
```

9.4 Base Jobs

In the previous Section 9.3, the *Base System* was described together with *Job Wrapper* interfaces it schedules. The user can create implementations of the *Job Wrappers* with any code they want to run. This section focuses on simplifying the implementation of these interfaces.

While dicussing *Neighbor Behaviors* in the previous Chapter 6.4.2, we identified a pattern shared by all *Neighbor Behaviors*. This section builds on this concept, abstracting the shared logic to allow users to focus only on implementation details of different behaviors. For example, implementations of *Neighbor Behaviors* like cohesion, alignment or separation would all need to reimplement the following logic. Each entity iterates over all its neighbor accumulating an intermediate result. While iterating, some data about the boids and their neighbors, such as their positions, may need to be queried from the `BaseBehaviorParams` or from components attached to them. Later, the intermediate result is processed into a final result, which is written into the `Results` of the behavior's *Job Wrapper*.

To prevent reimplementing this logic for each new behavior, custom jobs [72] (see Subsection 9.4.1) hiding the shared logic are provided. This section gives an overview of what types of custom jobs are provided by the framework, what they handle for the user in the background, and how they can be used.

9.4.1 Custom Job Types

In Unity's ECS, there are several job types. The simplest jobs implement the `IJob` [73] interface. There, the job's `Execute` function is ran once. Another example is `IJobParallelFor` [74]. Implementations of it run their `Execute` method once for each index in the range of $[0, length]$.

Unity provides a way of defining new custom job types like `IJobParallelFor`. Going into details of defining new custom jobs is outside the scope of this thesis. A good explanation of this can be found in a blog post by Jack Durstann [75]. In the framework, jobs hiding shared logic from the user are implemented as custom

jobs. These custom jobs can be used by the user to simplify creation of new behaviors. The job can then be scheduled by a *Job Wrapper*. This way, the use only has to implement the custom job's interface to create a new behavior, which can then be scheduled from a *Job Wrapper*.

9.4.2 Entity Information

All our custom jobs iterate over all the boids in parallel, and invoke one or more methods, which the user has to implement. Usually this is the `Execute` method, with a parameter of type `EntityInformation<C>` [1]. It contains the entity's data from `BaseBehaviorParams`, and an ECS component of type `C`. This component should be attached to every entity using the custom job. It can be used to parametrize a behavior per each entity.

Listing 9.9 shows a snippet of `EntityInformation<C>`. Figure 9.6 shows how instance of `EntityInformation<C>` is extracted from `BaseBehaviorParams` by the custom jobs. For example, for the first entity, e_1, the instance is filled with the first position, p_1, and velocity, v_1, from `BaseBehaviorParams`. Additionally, an arbitrary component of type `C`, relevant for the given behavior, is looked up on the entity and assigned to the `Component` field. The component `C` is useful to specify parameters of the behavior per entity. For example, `CohesionJobWrapper` from *defaults* relies on the entites having a `CohesionComponent`. That component contains parameters like the behavior's weight.

Listing 9.9: `EntityInformation<C>`

```
public struct EntityInformation<C> where C : unmanaged, IComponentData
{
    public LocalToWorld LocalToWorld;
    public VelocityComponent Velocity;
    public C Component;
    ...
}
```

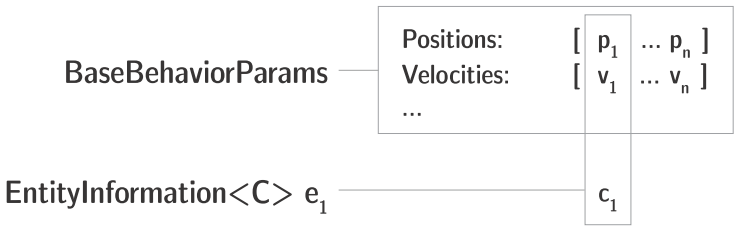

Figure 9.6: Relation between `BaseBehaviorParams` and `EntityInformation<T>`.

[1] In code, the generic parameter is named `ComponentT`. Here the name is shortened.

9.4.3 Velocity Result(s)

In Subsection 9.3.3, we discussed that the framework is set up such that behavior *Job Wrappers* can use any type for their results, as long as that type implements `IDelayedDisposable`. We also discussed that in the framework's *defaults*, all behaviors write into an instance of type `VelocityResults` that contains one `VelocityResult` for each entity. This subsection describes the `VelocityResult` and `VelocityResults` types in more detail.

The `VelocityResult` struct is shown in Listing 9.10. It contains a normalized desired direction – `Direction`, together with its desire – `DirectionDesire`. It also contains a desired speed – `Speed`, together its desire – `SpeedDesire`. Lastly it contains and a priority – `Priority`. An example instance of `VelocityResult` is visualized in Figure 9.7. It shows `Direction` equal to $(0, 0, 1)$ and `DirectionDesire` equal to 0.5. This is drawn as a vector scaled by the desire, with the values written out in the first row of the box. The example `VelocityResult` has `Speed` equal to 1 and `SpeedDesire` equal to 0.5, as indicated by the second row.

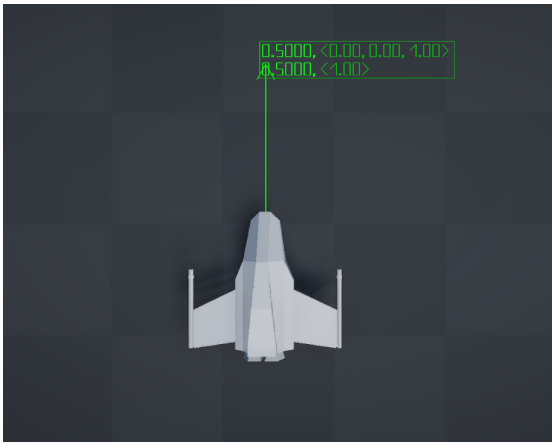

Figure 9.7: Visualization of `VelocityResult`. The arrow is `Direction` $*$ `DirectionDesire`. First row of the box lists the `DirectionDesire` and `Direction`. Second row lists the `Speed` and `SpeedDesire`.

When working with the *Job Wrapper* interfaces described in Subsection 9.3.4, the results only have one restriction – they must implement `IDelayedDisposable`. Later in this section, custom jobs that can ease implementation of these *Job Wrappers* are introduced. When working with these custom jobs, the results, such as `VelocityResults`, have additional restrictions on their type. Furthemore, type restrictions are put on each element in the results, such as `VelocityResult`. This is apparent from Listing 9.10 showing the `VelocityResult` type.

The custom jobs require that the `VelocityResult` struct implements three interfaces – `IDebugDrawable`, `IValidable`, and `IToStringFixed`. They are used internally by the the custom jobs to draw the result's visualization, to check the result's validity, and to turn the result into a fixed string that can be used in burst compiled code. For example, `VelocityResult` is not valid, if `DesiredDirection` is not normalized. In that case, the user is warned in the console, using the `ToStringFixed` method. The `Draw` method gives visualization shown in Figure 9.7. The visualization is drawn in color assigned to the `Color` field. This is especially useful for

debugging. The `Draw` method gets called by the custom job if the user sets the `Debug` flag of the behavior's component to true.

Listing 9.10: Implementation of the `VelocityResult` struct.

```
public struct VelocityResult : IDebugDrawable, IValidable, IToStringFixed {
    public float3 Direction;
    public float DirectionDesire;

    public float Speed;
    public float SpeedDesire;

    public byte Priority;

    public Color Color;
    public void Draw(float3 position, float scale) { ... }
    public bool IsValid() { ... }
    public void ToStringFixed(out FixedString128Bytes string128) { ... }
}
```

As mentioned above, the type `VelocityResults` also has additional type restrictions when using the custom jobs. On top of having to be `IDelayedDisposable`, the struct also needs to implement the `IBehaviorResults<R>` [2] interface. See Listing 9.11, where `VelocityResults` implements this interface. The interface forces `VelocityResults` to contain a field `Results` of type `NativeArray<R>`, where `R` must implement the three interfaces that `VelocityResult` implements in Listing 9.10. The `IBehaviorResults<R>` interface is useful, because it allows the custom jobs to write each `VelocityResult` directly into `Results`, so that a user does not need to do it.

Listing 9.11: Implementation of the `VelocityResult` struct.

```
public struct VelocityResults : IBehaviorResults<VelocityResult> {
    public NativeArray<VelocityResult> Results => results;
    private NativeArray<VelocityResult> results;

    public VelocityResults(int entityCount) {
        results = new NativeArray<VelocityResult>(entityCount,
            Allocator.TempJob);
    }

    public void Dispose(JobHandle dependency) { results.Dispose(dependency); }
}
```

[2] In code, the generic parameter is named `ResultT`. Here, the name is shortened.

9.4.4 Simple Behavior Jobs

In Subsection 9.3.4, we discussed that a new *Simple Behavior* could be created by implementing the `ISimpleBehaviorJobWrapper` interface (see 9.3.4). This would mean that the users would have write all the logic themselves. However, there is some shared logic that all *Simple Behaviors* would likely need to reimplement. Each *Simple Behavior* would have to loop through all entities queried by the *Main Entity Query*. For each entity, it would need to extract some data from the `BaseBehaviorParams`, and look up a component of some type `C` specific to the behavior. Then, a behavior's result of some type `R` would need to be calculated, and written into the *Job Wrapper's* `Results`. For each *Simple Behavior*, only the calculation of the behavior's result would be different. For this reason, a custom job that handles the shared logic in the background was created. To use it, the user has to implement the `ISimpleBehaviorJob<C, R>`[3] interface. An example implementation is shown in Listing 9.12. Figure 9.8 illustrates what the custom job does in the background.

First, the figure will be described. The custom job runs for all entities queried by the *Main Entity Query* in parallel. For each entity, the job extracts `EntityInformation<C>` from `MainBaseParams`, and passes it into the job's `Execute` method, that the user has to implement. The method returns a single result of generic type `R`, which is written at the entity's index into `Results`. In the figure, you can see n-th invocation of `Execute` receiving information about entity e_n, and returning a result v_{rn}. For a concrete example, see Listing 9.12.

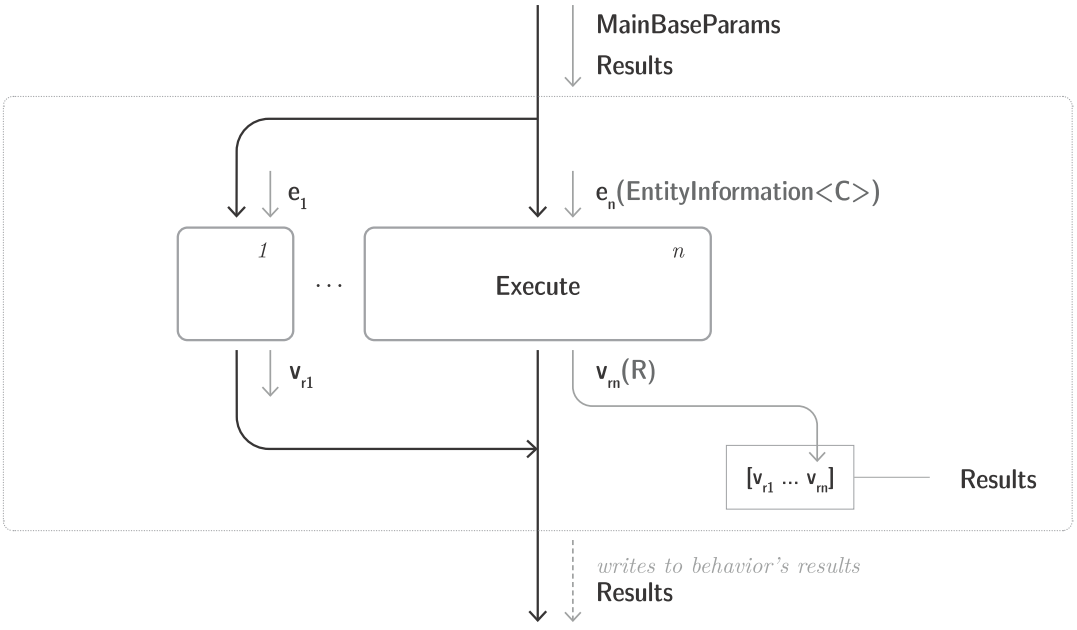

Figure 9.8: Illustration of what `ISimpleBehaviorJob<C, R>` does for the user in the background.

[3] In code, `C` is named `ComponentT`, and `R` is named `SteeringDataT`.

Listing 9.12 illustrates a sample *Simple Behavior* named `GoForwardJob`. It is created as an implementation of `ISimpleBehaviorJob<C, R>`. Here, the type parameter `C` is the `GoForwardComponent` component, and the type parameter `R` is the `VelocityResult`. The `Execute` method receives information about an entity. The `GoForwardComponent` of this entity is accessible through the `entity.Component` field. Based on information about this entity, an instance of `VelocityResult` is returned.

This behavior simply makes the entity go in its forward direction at a specified speed. This is done by assigning `entity.Forward` to `Direction` of the returned `VelocityResult`. The desired speed, desires and priority are assigned directly from the entity's `GoForwardComponent`.

Listing 9.12: Example implementation of `ISimpleBehaviorJob<C, R>`.

```
struct GoForwardJob : ISimpleBehaviorJob<GoForwardComponent, VelocityResult>
{
    public VelocityResult Execute(EntityInformation<GoForwardComponent> entity)
    {
        return new VelocityResult
        {
            Direction = entity.Forward,
            DirectionDesire = entity.Component.BaseData.DirectionStrength,
            Speed = entity.Component.BaseData.Speed,
            SpeedDesire = entity.Component.BaseData.SpeedStrength,
            Priority = entity.Component.BaseData.Priority
        };
    }
}
```

To use the `GoForwardJob` shown above, the user would implement a new *Job Wrapper* implementing the `ISimpleBehaviorJobWrapper` interface, and schedule the job from it. Listing 9.13 illustrates this. In the snippet, `GoForwadJobWrapper` schedules the `GoForwardJob`. From a user's perspective, this is only boilerplate code, needed to set everything up. The `Schedule` method of `GoForwadJobWrapper` only allocates `Results`, and then passes it together with all of its parameters to the `Schedule` method of `GoForwardJob`. Note, scheduling the job returns a `JobHandle` that completes when the job is done and the `Results` are filled out. The user should return this `JobHandle` from the *Job Wrappers* `Schedule` method, as shown in the sample.

Listing 9.13: Scheduling `GoForwardJob` from `GoForwardJobWrapper`, an implementation of `ISimpleBehaviorJobWrapper`.

```
public class GoForwardJobWrapper : ISimpleBehaviorJobWrapper
{
    public JobHandle Schedule(
        SystemBase SystemBase,
        in BaseBehaviorParams MainBaseParams,
        out IDelayedDisposable Results,
        in JobHandle Dependency)
    {
        Results = new VelocityResults(MainBaseParams.EntityCount);
        return new GoForwardJob().
            Schedule<GoForwardJob, GoForwardComponent, VelocityResult>(
                SystemBase,
                MainBehaviorParams,
                (VelocityResults)Results,
                1,
                Dependency);
    }
}
```

9.4.5 Neighbor Behavior Jobs

In Subsection 9.4.4, we showed a custom job that can simplify creation of *Simple Behaviors*. In the same manner, there is a custom job that simplifies creation of *Neighbor Behaviors*. The job's interface is named `INeighborBehaviorJob<C1, C2, A, R>`[4].

What the custom job does for the user in the background will be explained using Figure 9.9. Same as the `ISimpleBehaviorJob<C, R>`, the job runs for each entity in parallel. However, for each entity, there is up to k neighbors to process, so implementing a *Neighbor Behavior* is not as simple as invoking the `Execute` method that returns a result. For each entity, the `Execute` method is invoked up to k times, once for each neighbor. The method receives `NeighborsInformation<C1, C2>`. In contains information about the main entity – `EntityInformation<C1>` and information about its neighbor – `EntityInformation<C2>`. This is, for example, the n-th entity, e_n, and its first neighbor, q_{n1}. The `Execute` method also receives an "accumulator" of type `A`. The accumulator is used to keep track of some state between invocations of the `Execute` method. For the entity e_n, this is illustrated as accumulator going through states $a_{n1} \dots a_{nk}$. In other words, the accumulator is updated for each of the k neighbors. After all the neighbors are processed, the accumulator and information about the entity are passed into `Finalize` method,

[4]In code, the generic parameters have different naming. `C1` is named `ComponentT`, `C2` is named `OtherComponentT`, `A` is named `AccumulatorT` and `R` is named `SteeringDataT`.

which returns a result of type `R`. Same as with `ISimpleBehaviorJob<C, R>`, the result is written into `Results`. This approach of splitting the behavior into an accumulation and finalization part was inspired by generalization of *Neighbor Behaviors* in Subsection 6.4.2.

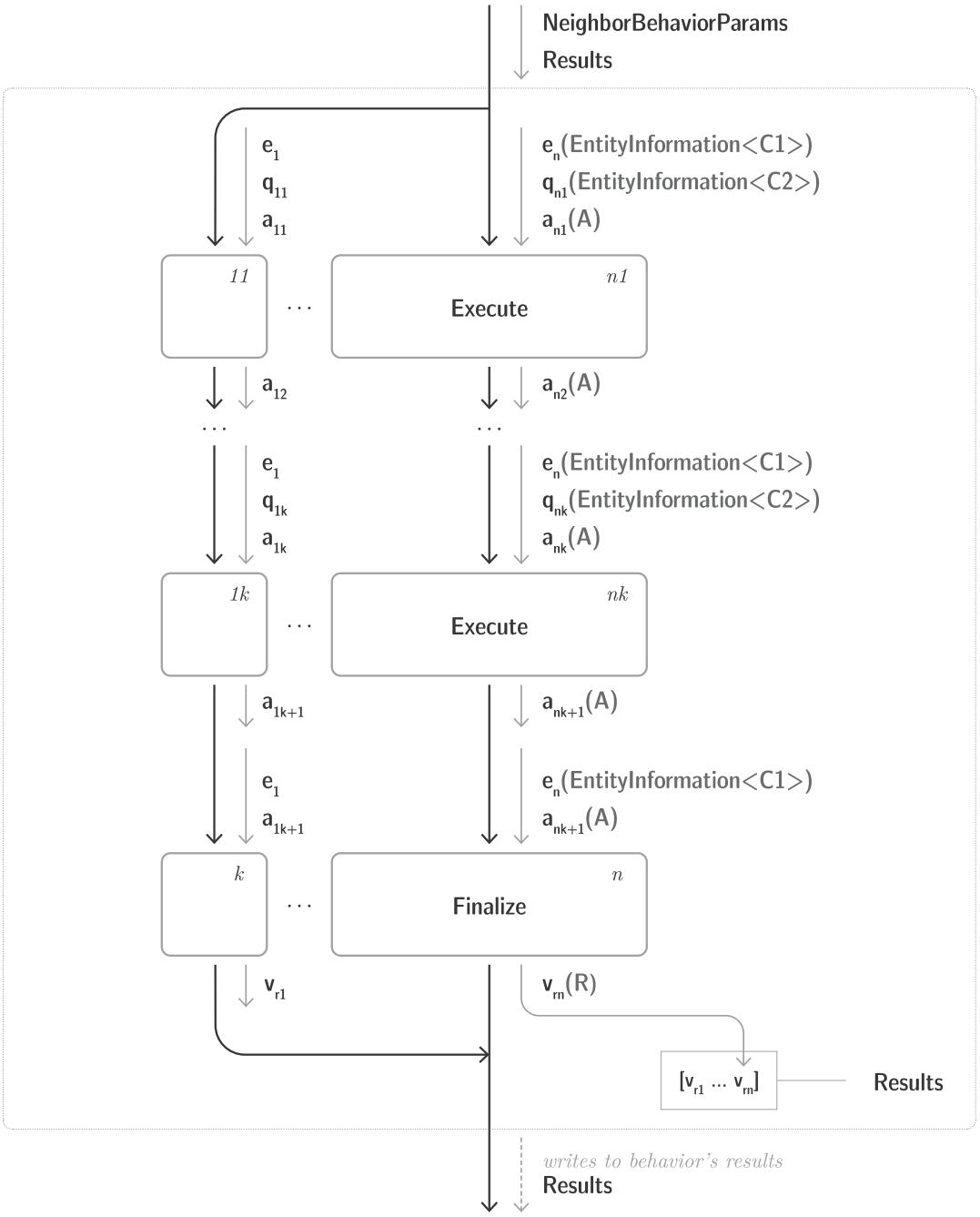

Figure 9.9: Illustration of what `INeighborBehaviorJob<C1, C2, A, R>` does for the user in the background.

Listing 9.14 shows a simple implementation of a cohesion behavior implemented using the `INeighborBehaviorJob<C1, C2, A, R>` interface. Usually, the main idea behind a cohesion behavior is to accumulate centroid of all neighbors, and then go towards it. Therefore, the `Execute` method adds the neighbor's position into the `Sum` accumulator, which accumulates a sum of vectors passed into it. Afterwards, the `Finalize` method uses information from the accumulator to calculate the centroid, and returns a `VelocityResult` with direction towards it. The user would schedule the job from an implementation of `INeighborBehaviorJobWrapper`, similarily as it was done with `GoForwardJob` in Listing 9.13.

Listing 9.14: Example implementation of `INeighborBehaviorJob<C1, C2, A, R>`.

```
struct CohesionJob : INeighborBehaviorJob<CohesionComponent, Comp2, Sum,
    VelocityResult>
{
    void Execute(in NeighborsInformation<CohesionComponent, Comp2> pair, ref
        Sum accumulator)
    {
        accumulator.Add(pair.OtherEntity.Position);
    }

    VelocityResult Finalize(in EntityInformation<CohesionComponent> entity, in
        Sum accumulator)
    {
        float3 centroid = accumulator.SumVector / accumulator.NumNeighbors
        float3 toCentroid = math.normalize(centroid - entity.Position);
        return new VelocityResult { Direction = toCentroid, ... }
    }
}

struct Sum : IAccumulator
{
    public float3 SumVector;
    public int NumNeighbors;

    public void Add(float3 position) { SumVector += position; NumNeighbors++; }
    public void Init() { SumVector = float3.zero; NumNeighbors = 0; }
}
```

9.4.6 Ray Behavior Jobs

In Subsection 9.4.5, we described `INeighborBehaviorJob<C1, C2, A, R>` interface, which simplifies creation of *Neighbor Behaviors*. The `IRaycastBehaviorJob<C, A1, A2, R>`[5] interface simplifies creation of *Ray Behaviors* in a very similar way. No figure is provided for this custom job, since the idea is almost identical to a custom job for *Neighbor Behaviors*. For an example implementation of `IRaycastBehaviorJob<C, A1, A2, R>`, see Listing 9.15 below.

To implement a new *Ray Behavior*, the user has to implement three methods – `OnHit`, `OnMiss` and `Finalize`. As usual, the job loops in parallel through all entities. For each entity with component `C`, the `OnHit` method is invoked per each ray that hit, and `OnMiss` is invoked per each ray that missed. Intermediate results from `OnHit` are accumulated into an accumulator of type `A1`, while `OnMiss` uses an accumulator of type `A2`. After the job has looped through all the rays, `Finalize` method is invoked with information about the entity and the two accumulators, and returns a result of type `R`. An implementation of this job would be scheduled from an implementation of `IRaycastBehaviorJobWrapper`, similarily as it was done with `GoForwardJob` in Listing 9.13.

Listing 9.15 shows a simple implementation of a collision avoidance behavior implemented as `IRaycastBehaviorJob<C, A1, A2, R>`. Here, the `OnHit` method uses `Min` accumulator (`A1`) to find the closest ray hit position. The `OnMiss` method is empty, because it is not needed for this behavior. Therefore, it uses an empty `None` accumulator (`A2`). The `Finalize` method returns a `VelocityResult` (`R`) with `Direction` away from the closest hit. Additional information about the entity that could be used in the behavior are on its `AvoidanceComponent` (`C`).

[5] In code, `C` is named `ComponentT`, `A1` is named `AccumulatorHitT`, `A2` is named `AccumulatorMissT` and `R` is named `SteeringDataT`.

Listing 9.15: Example implementation of `IRaycastBehaviorJob<C, A1, A2, R>`.

```
struct AvoidanceJob : IRaycastBehaviorJob<AvoidanceComponent, Min, None,
    VelocityResult>
{
    void OnHit(in EntityInformation<AvoidanceComponent> entity,
        in RaycastHit hit, in RayData rayData, ref Min hitA)
    {
        hitA.Add(hit.HitPosition, hit.Distance);
    }

    void OnMiss(in EntityInformation<AvoidanceComponent> entity,
        in RayData rayData, ref None missA) { }

    VelocityResult Finalize(in EntityInformation<AvoidanceComponent> entity,
        in Min hitA, in None missA)
    {
        float3 fromClosest = math.normalize(e.Position - hitA.ClosestPosition);
        return new VelocityResult{ Direction = fromClosest, ...};
    }
}

struct Min : IAccumulator
{
    public float3 ClosestPosition;
    private float MinDistance;

    public void Add(float3 position, float distance)
    {
        if (distance < MinDistance)
        {
            ClosestPosition = position;
            MinDistance = distance;
        }
    }

    public void Init() { MinDistance = float.PositiveInfinity; }
}

struct None : IAccumulator { public void Init() {} }
```

9.4.7 Ray Creation Jobs

The framework provides a custom job that simplifies creation of new *Ray Queries*. As discussed in Subsection 9.3.4, an implementation of `ICreateRaysJobWrapper` should write k rays per each entity into the *Job Wrapper's* `Results` array. The array should be filled in a flat 2D array format. This can be simplified through a custom job that requires implementation of `ICreateRaysJob` interface.

The custom job loops through all entities in parallel as usual. For each entity, the `Execute` method is invoked k times, once for each ray to create. In the sample below, k is `NumRays`. The `Execute` method receives information about the entity, `entity`, and an index of a ray to create for this entity, `rayIndex`. Each invocation of the `Execute` method returns one instance of `RayData`, and the custom job writes it at the approapriate index into the `Results` array. Note, information about the entity passed into the `Execute` method is an instance of `EntityInformation<SteeringEntityTagComponent>`. Unlike other custom jobs, the job itself does not have a generic parameter for the component, because we never found a need to have one. Instead, the tag `SteeringEntityTagComponent` is used. This tag is present on all entities used by the framework.

Listing 9.16 illustrates a sample implementation of `ICreateRaysJob`. This implementation creates `NumRays` rays in a circle around each entity. Each ray ranges to distance `MaxDistance`.

Listing 9.16: Example of implementation of `ICreateRaysJob`.

```
struct Circle2DRaysJob : ICreateRaysJob
{
    public int NumRays;
    public float MaxDistance;

    RayData Execute(int rayIndex,
        in EntityInformation<SteeringEntityTagComponent> entity)
    {
        float alpha = 2 * math.PI * rayIndex / NumRays;
        float3 dir = new float3(math.sin(alpha), 0, math.cos(alpha));

        return new RayData
        {
            Origin = entity.Position,
            Direction = dir,
            MaxDistance = MaxDistance
        };
    }
}
```

9.5 Provided Implementations

The framework, as described until now, provides a high level modular abstraction through the `BaseSteeringSystem`, `SteeringSystemAsset`, and the *Job Wrappers*. It also contains custom jobs for some of the *Job Wrappers* to ease the implementation of the user's logic. That is the *core* of the framework. We also want to provide the user with concrete implementations of all the *Job Wrappers*, so that the framework can be used right away, even without programming knowledge. This is the *defaults* part of the package, which will be discussed in this section. The default implementations would also be useful as a reference when implementing new ones. All the implemented behaviors return the `VelocityResult` struct, discussed in Subsection 9.4.3.

9.5.1 Simple Behaviors

The *defaults* contain several useful implementations of `ISimpleBehaviorJobWrapper`. We found these to be useful among different scenarios, or for prototyping. The behaviors are shortly discussed below. More details can be found in the attached documentation A.1.3. As of writting this thesis, there are 7 *Simple Behaviors* provided by the framework.

Wandering

The wandering behavior (`WanderingJobWrapper`) is perhaps the most common behavior to add to the traditional boids model. It adds more randomness to the movement of the boids. The implementation is based on the wandering behavior from the Behaviors Chapter 6.4.6. It uses perlin noise to smoothly change the desired direction and speed. The main properties the user can adjust are the frequencies of the noise functions, and minimum and maximum speeds.

Homing

The homing behavior (`HomingJobWrapper`) makes the boids move towards a target spherical area. The implementation is based on the homing behavior from the Behaviors Chapter 6.4.6. The direction desire to go towards the home grows as the boid's distance to it goes from minimum to maximum radius that the user sets.

Go Forward

The go forward behavior (`GoForwardJobWrapper`) makes the boid continue going in its current direction, at a specified speed. It can be useful as a behavior that drives the boid's speed. This is useful when no other behavior returns non-zero speed and speed desire. It can also be useful for debugging.

Up Alignment

When trying out 3D simulations of fish or birds, we noticed that the boids would sometimes move straight up or down, which looks unnatural. This behavior (`AlignUpJobWrapper`) tries to prevent this. The behavior returns the boid's current

direction projected on the world's up direction. The direction desire grows as the angle between the current forward and world's up decreases.

Keep Height

The keep height behavior (`KeepHeightJobWrapper`) makes the boid stay within a specified range of y coordinates. It is useful to keep the boids from flying too high or too low. The homing behavior already restricts the y coordinate as well, but we often found it useful to limit the maximum and minimum height even more. It also mitigates visible spherical artifacts of the flock when using only homing. The behavior returns direction straight up or down. The direction desire grows the farther the boid moves outside the specified range.

Follow Path

The follow path behavior (`FollowPathJobWrapper`) makes the boid follow a specified path at a specified speed.

Debug Simple

This behavior (`DebugSimpleJobWrapper`) draws a circle around each boid. It is useful, for example, to mark a boid of interest for debugging, so that it does not get lost among others.

9.5.2 Neighbor Behaviors

The *defaults* contain several implementations of `INeighborBehaviorJobWrapper`, covering some common group interactions, like flocking and predator and prey interactions. More details about the behaviors can be found in the attached documentation A.1.3.

Cohesion

The cohesion behavior (`CohesionJobWrapper`) makes the boids move towards the centroid of their neighbors. It is useful for keeping a group of boids together. The implementation is based on the cohesion behavior from Subsection 6.4.5. The direction desire grows with distance to the centroid.

Alignment

The alignment behavior (`AlignmentJobWrapper`) makes the boids align their direction and speed with the average direction and speed of their neighbors. It is useful for keeping a group of boids aligned. The implementation is based on the alignment behavior from Subsection 6.4.5. The direction desire grows with how misaligned the current boid is, relative to the average direction. The speed desire grows with how far the current boid's speed is from the average speed.

Separation

The separation behavior (`SeparationJobWrapper`) makes the boids move away from their neighbors. It is useful for preventing collisions within a group. The implementation is based on the separation behavior from Subsection 6.4.5. The direction desire grows with the largest observability value out of all neighbors.

Flee

The flee behavior (`FleeJobWrapper`) makes the boids move away from its neighbors at high speed. It is useful for predator avoidance. The difference from separation is that this behavior affects speed as well. The direction desire, speed and speed desire all grow as the neighbor gets closer.

Seeking

The seeking behavior (`SeekingJobWrapper`) makes the boid move towards a neighbor. It is useful, for example, for a predator that hunts other boids. Same as with the flee behavior, the direction desire, speed and speed desire grow as distance to the neighbor decreases.

Multi Homing

The multi homing behavior (`MultiHomingJobWrapper`) is very similar to the previously mentioned homing behavior from Subsection 9.5.1. The difference is that this behavior can work with multiple different homes placed around the scene. Out of all the homes where the boid's distance to the center is smaller than the home's maximum radius, the boid will travel to the one with the smallest maximum radius. This way, there can be multiple smaller homes scattered around the environment, with large homes making sure the boids do not leave the general area. The direction desire to go towards the home grows as the boid's distance to it goes from minimum to maximum radius which the user sets.

Debug Neighbors

This behavior (`DebugNeighborsJobWrapper`) draws lines from the current boid to each of its neighbors. It is useful for debugging the neighbor relationship given by an implementation of `INeighborQueryJobWrapper`.

9.5.3 K Nearest Neighbor Search

The framework contains two implementations of `INeighborQueryJobWrapper`. From the user's perspective, they both do the same. They find k nearest neighbors restricted by maximum radius and maximum field of view. The only difference is in the used data structure. One implementation uses k-d trees (Subsection 5.3.3), the other uses space partitioning (Subsection 5.3.2). Their comparison from perspective of performance is analyzed in Subsection 9.6.1.

KD Tree KNN

This implementation is based on k-d trees (Subsection 5.3.3). It can be scheduled from `KDTreeKNNJobWrapper`. The implementation is a modified version of k nearest neighbor search implementation found in a public github repository by Arthur Brusse [76]. The main modifications done are the restrictions by maximum radius and field of view.

Spacial Hash KNN

This implementation is based on spacial partitioning (Subsection 5.3.2). It can be scheduled from `SpacialHashKNNJobWrapper`. It is implemented as a sparse grid, so that the user does not have to specify and keep track of a bounding box. Initially, we started experimenting with flocking algorithm which uses spatial partitioning found on Unity's official github [11]. Upon closer inspection, it becomes clear they take a different approach to flocking. They do not calculate quantities such as average velocity of a neighborhood per boid as we do, but per cell of the grid. While this provides a performance improvement, it suffers from "blocky" artefacts, making the grid visible as the boids move. For this reason, we did not continue with this approach further. However, we used their implementation as a base for our implementation of the spacial partitioning grid.

9.5.4 Ray Behaviors

The *defaults* contain a few implementations of `IRaycastBehaviorJobWrapper`. All the provided *Ray Behaviors* are aimed at avoiding collisions. The behaviors are shortly discussed below. More details can be found in the attached documentation A.1.3. As of writting this thesis, there are 3 *Ray Behaviors* provided by the framework.

Environment Avoidance

The environment avoidance (`EnvironmentAvoidanceJobWrapper`) is the most general behavior to avoid collisions. It makes the boids move in direction normal to the surface detected by ray casts of the *Ray Query*. The implementation is based on our obstalce avoidance behavior from Subsection 7.1.4. The direction desire grows with the largest observability value out of all ray hits.

Avoid Verticals

The avoid verticals behavior (`AvoidVerticalWallsJobWrapper`) is very similar to the environment avoidance behavior. The main difference is that the obstacle's normal is projected on the plane defined by world's up (the xz plane). This way, the behavior does not bias the desired direction up or down.

Figure 9.10 shows a trajectory when using the normal environment avoidance in yellow and avoid verticals behavior in red. The boid was driven by *Go Forward* behavior (see Subsection 9.5.1), illustrated in magenta. Note that the yellow path goes up, because the normal of the wall to avoid points up. This behavior was included especially for cases like keeping fish inside a pond, because we often ran

into an issue where the environment avoidance biased the fish towards the water's surface.

Note, this behavior ignores rays that hit surfaces with slope below a set angle, because the behavior is not effective at avoiding those. For example, a normal of flat plane in world's xz plane would give a zero vector, if projected on the same plane.

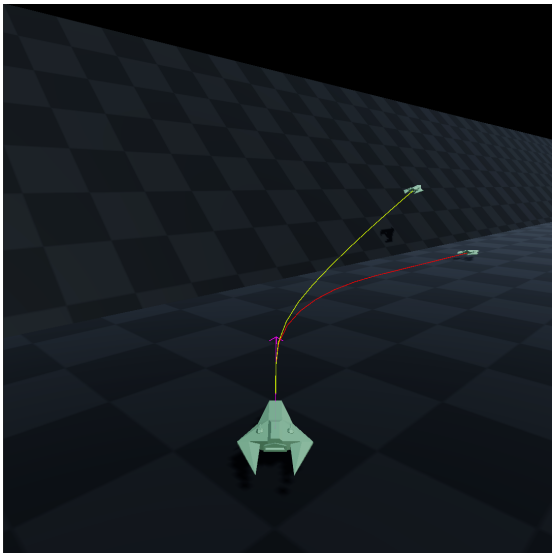

Figure 9.10: Collision avoidance trajectories. Movement direction – magenta arrow, environment avoidance behavior trajectory – yellow, avoid verticals behavior trajectory – red.

Avoid Ground

As the name suggests, the avoid ground behavior (`AvoidGroundJobWrapper`) is aimed at avoiding ground[6] below. It is useful for example for birds flying above an uneven terrain. The Figure 9.11 shows a trajectory when using this behavior. The behavior always returns direction straight up, drawn in green in the figure. The direction desire grows with distance to the closest ray hit, drawn with blue in Figure 9.12.

[6]What is considered ground depends on the `LayerMask` specified in the *Ray Query*.

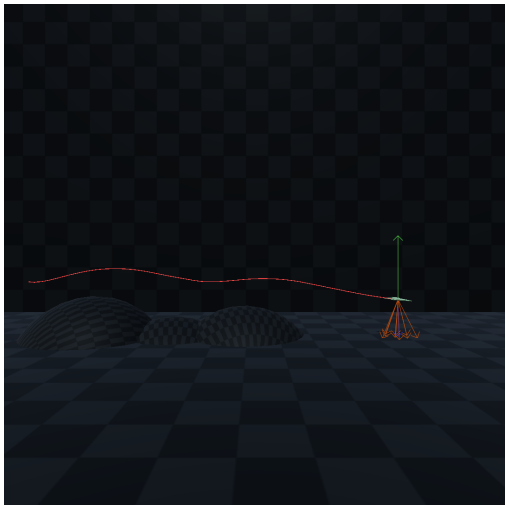

Figure 9.11: Avoid ground behavior trajectory. Red – boid's trajectory, orange – ray hits, green – desired direction.

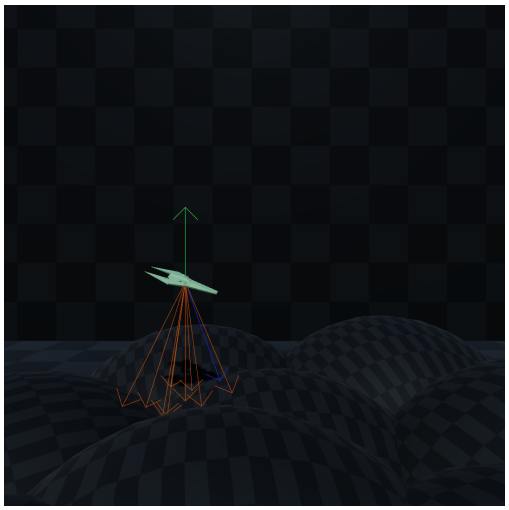

Figure 9.12: Using avoid ground behavior. Blue – closest ray hit, orange – other ray hits, green – desired direction.

9.5.5 Ray Queries

The *defaults* contain two useful implementations of `ICreateRaysJobWrapper`, which are used to create ray casts for implementations of `IRaycastBehaviorJobWrapper`.

2D Cone

The 2D cone (`Sweep2DCreateRaysJobWrapper`) is especially useful for 2D simulations. It creates rays in a 2D cone as shown in Figure 9.13. The cone is centered around the boid's direction. The user can set the field of view of the cone, as well as the maximum distance.

Figure 9.13: Ray casting in a 2D cone, missed rays in yellow, hit rays in red.

3D Cone

This *Ray Query* (`ConeCreateRaysJobWrapper`) is analogous to the 2D cone, but in 3D. Illustration of the rays was shown already in Figure 9.12 as orange arrows. This *Ray Query* was already desribed in Subsection 7.1.3. The user can adjust rotation relative to the boid's direction. In the figure, the rays point straight down in world space. The code is based on implementation by Sebastian Lague [77], who used it in his youtube video about boids [9].

9.5.6 Merging

The *defaults* contain only one implementation of `IMergeJobWrapper`, which merges results of the behaviors. It is scheduled from `MergeVelocitiesJobWrapper`. It accepts results of behaviors of type `VelocityResults`, introduced in Subsection 9.4.3. The job then writes the final desired velocity into the boid's `DesiredVelocityComponent`. The implementation is based on the *Merger* which we proposed in Section 4.4. The behaviors' results are accumulated from the highest priority as long as the sum of their desires is less than one. The results that are not filtered out are merged using a weighted average, where direction and speed are summed up separately, weighted by their desires.

9.5.7 Movement

The movement is handled by a separate system, not a part of the *Base System*. This gives the flexibility to use the same movement system for all entities using the `DesiredVelocityComponent`, which *Merger* writes to. Three implementations of a movement system are provided. One for 3D, one 2D, and one for "2.5D" which is movement in 3D, but on a surface. All implementations are based on our proposed *Mover* from Section 3.5. The main idea is that first, a desired acceleration is calculated based on the desired velocity, the desired acceleration is then split into parallel and lateral components that can be scaled independently, and added to the current velocity.

All three movement systems read the `DesiredVelocityComponent`, and write their results into the `VelocityComponent` and the `Rotation` of the `LocalTransform` component. The movement systems only rotate the boids and set their new velocity. The change of position based on the velocity is done later elsewhere, to enable for example the addition of a collision resolution step (see Subsection 9.5.8).

2D Movement

The implementation 2D movement is in essence directly taken from the Section 3.5. The main implementation detail is that both the current and desired velocities are projected on the xz plane, to ensure the movement stays in 2D. The main properties to specify are the maximum speed, and maximum accelerations in either directions (parallel/lateral). The rotation of the boid is blended from its current rotation, to the rotation given by its current velocity, using spherical linear interpolation as proposed in Subsection 3.6.2.

3D Movement

The implementation of 3D movement is directly taken from the Section 3.5. An interesting detail is that for rotation, we took an idea from Reynolds' paper [7]. In the paper, he described that his boids not only align their direction with the their velocity, but their up direction is aligned to simulate the effects of centrifugal force, making the realistically looking banking turn.

Figure 9.14 illustrates this idea. The red arrow shows the gravitational force. The green arrow shows the centrifugal force, their negated sum gives the new up direction which the boid should align with, shown in yellow. Figure 9.15 shows a path taken by a boid, the yellow arrows show how its up direction progressed.

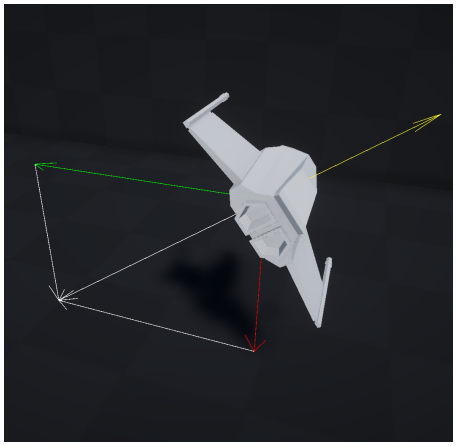

Figure 9.14: The boid's centrifugal force (green), gravitational force (red) and the direction to align its up with (yellow).

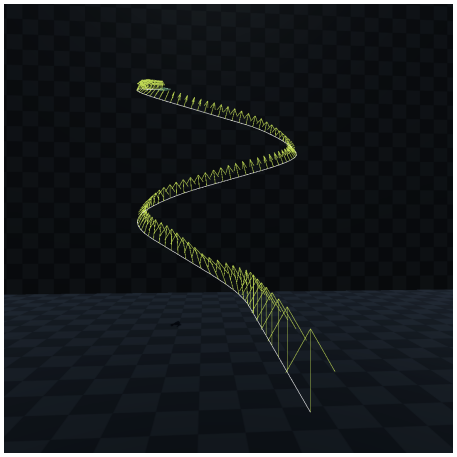

Figure 9.15: Sequence of a boid's up directions (yellow) after moving through trajectory in white.

2.5D Movement

The 2.5D movement system solves walking on surface of a 3D object, usually a terrain. It is handled by two states, if there is no ground below, the boid falls down. On the ground, the algorithm is more complex.

First, the desired velocity is projected on the surface below and scaled to its original length. The user can set two important parameters: a minimum and a maximum slope angle. When walking up, the desired velocity's magnitude is scaled by value from zero to one as the angle of surface below goes from maximum to minimum angle. This makes the boid gradually loose speed as the slope increases. When walking down, the desired velocity is scaled by value from one to two, as the slope goes from minimum to maximum angle. This makes the boid gradually gain speed as the slope increases when walking down. The system then finds the desired acceleration as usual, and adds it to the current velocity.

Afterwards, several sphere casts are used to detect and handle cases such as walking over small obstacles like stairs, snapping to the surface if slightly above ground or switching to the falling state if the surface is too far. While the system resolves potential collision to some extent, by projecting the desired velocity onto the surface, it may still fail in some scenarios. For this reason, it should be used

together with collision resolution system, for example `CollideAndSlide` described in Subsection 9.5.8.

So far, only the linear movement was discussed, but the system handles rotations in quite an interesting way. The boid will align its up direction with the normal of the surface and its forward direction with its current direction of movement. This is illustrated in Figure 9.16, where the boid has a current forward direction (green) and a current up direction (orange). The boid will smoothly rotate to align with its target forward direction (blue) and target up direction (red).

For the rotation, we wanted to have control over how fast the boid aligns its up and forward directions align separately. The usual spherical linear interpolation "slerp" does not support this. We found that a solution to this problem was described by Ming-Lun "Allen" Chou in his blog post [78] about swing-twist interpolation - "sterp". We used his implementation of this algorithm after fixing minor issues and edge cases pointed out by the readers in the comments. The user can set two rotation speeds, one for aligning with the surface, one for aligning with the velocity.

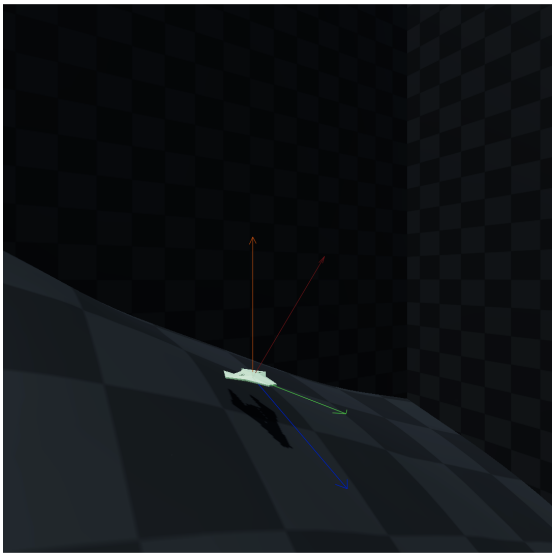

Figure 9.16: Boid before aligning to the surface. Current forward (green), current up (orange), target forward (blue) and target up (red)

9.5.8 Collision Resolution

The collision resolution is handled by `CollideAndSlide` system, which makes sure the boids do not clip through colliders in the scene. The implementation is based on the collide and slide algorithm discussed in Subsection 7.2.2. The `CollideAndSlide` system is the last step after movement systems, and it can be used regardless of the selected movement system. It considers the boid's current position, velocity and radius to update the boid's position. The user can use the `CollideAndSlideComponent` to set the `LayerMask` of meshes to avoid.

9.6 Performance Testing

One of the goals of this framework was to provide a performant solution for flocking. This section will briefly analyze the performance of simulations created using the framework. The main bottleneck are the *Neighbor Queries*, which is why the main focus will be on them, in Subsection 9.6.1. Afterwards, Subsection 9.6.2 will consider performance of an entire flocking simulation. The Unity project used for testing the performance can be found in Attachement A.3.

9.6.1 Neighbor Queries

The *defaults* contain two implementations of k nearest neighbor search. One using k-d trees, one using spacial partitioning grid. In Chapter 5, we were concerned with theoretical worst case performance of neighbor search per one boid. Based on the theory, we expected that k–d trees would be more suitable for smaller flocks, and spacial partitioning would eventually overtake as the total number of boids increases. We also expected k–d trees to scale better as the amount of neighbors within a boid's radius increases. For these reasons, we concluded that k–d trees would likely be better for our use case, since we are interested in hundreds to lower thousands of boids, and also we do not want radius of vision or how dense the flock is at the moment to affect the performance. In this subsection, we will see if our assumptions hold in the actual implementations.

Methodology

In Chapter 5, it became apparent that there is a few important factors which have performance implications. The variables which we will be interested in are:

- n – Total number of boids.
- k – Maximum number of neighbors.
- ρ – Density of the flock. (explained below)

As discussed in Chapter 5, the performance will unsurprisingly depend on n and k. However, it will also depend on how many potential neighbors m there are within a boid's radius of vision r. To account for this, we keep constant radius of vision $r = 5$. This way, m depends only on how dense a flock is. To control a flock's density, expressed in number of boids per meter cubed, we introduce a variable ρ. Then, to try to account for keeping constant ρ, while changing the number of boids n, we restrict boids' positions to a cube, where the cube's side length l is chosen such that $\rho = n/l^3$ is a constant. If a boid travels beyond the cube's extents, it gets teleported to the opposite side.

To illustrate this idea, see Figures 9.17, 9.18. They show flocks of size $n = 100$ and $n = 5000$ with the same constant density $\rho = 0.01$. For reference, the length of each boid is around 0.2 meters. For a much denser flock, see Figures 9.19, 9.20, which show flocks of size $n = 100$ and $n = 5000$ with the same constant density $\rho = 2$.

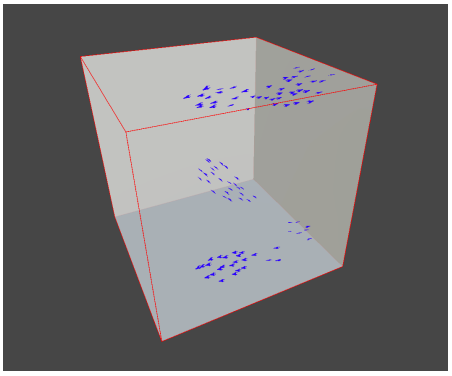

Figure 9.17: Group of boids for $n = 100$, $\rho = 0.01$.

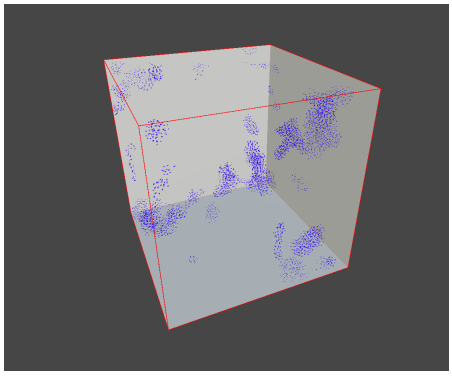

Figure 9.18: Group of boids for $n = 5000$, $\rho = 0.01$.

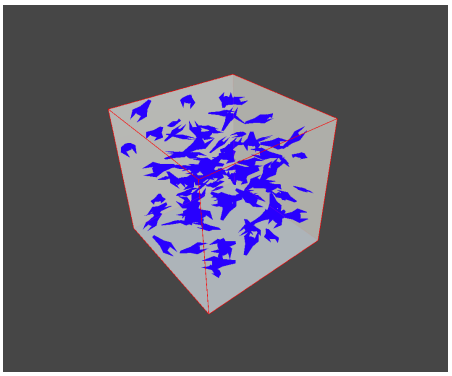

Figure 9.19: Group of boids for $n = 100$, $\rho = 2$.

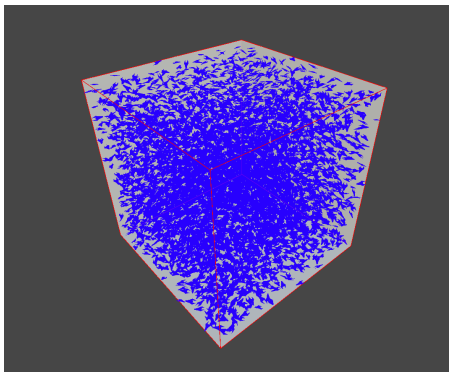

Figure 9.20: Group of boids for $n = 5000$, $\rho = 2$.

When measuring the performance, we spawn a number of boids n randomly around the cube. Then we wait for a few seconds for the simulation to warm up at an increased simulation speed. Afterwards a few seconds is spent collecting the durations it took to complete a *Neighbor Query* in each frame. We use this to find the average time it took for a *Neighbor Query* to complete t_{avg}. A *Neighbor Query* finds k nearest neighbors for all n boids, but it is more interesting to consider how long it takes per one boid. Therefore, we measure $t = t_{avg}/n$, the average time it took to query k nearest neighbors per one boid. The time here is measured in microseconds. The tests were ran on a desktop PC with i9-14900K processor and 64GB of RAM.

Variable Number of Boids

The first performance test was to see how t scales with increasing n, for constant $k = 5$ and $\rho = 0.01$. First see Figures 9.21, 9.22. They show how t progresses for $n \in [50, 1000]$. In both cases, t starts at around 1.6 microseconds and appears to stabilize to roughly a constant around $n = 500$. For k–d tree this constant is approximately $t = 0.3$ microseconds, for spacial partitioning the constant is approximately $t = 0.2$ microseconds. In this case, spacial partitioning is the better choice even for small n, which is not what we expected.

We assume that the reason why the average time per boid is larger for small n is because in those cases, the cube which holds the boids is so small that essentially

all n boids are within the radius of vision. For larger n with a larger cube, only a fraction of n is within the radius of vision. Possibly, constant overhead of scheduling the jobs could also play a role.

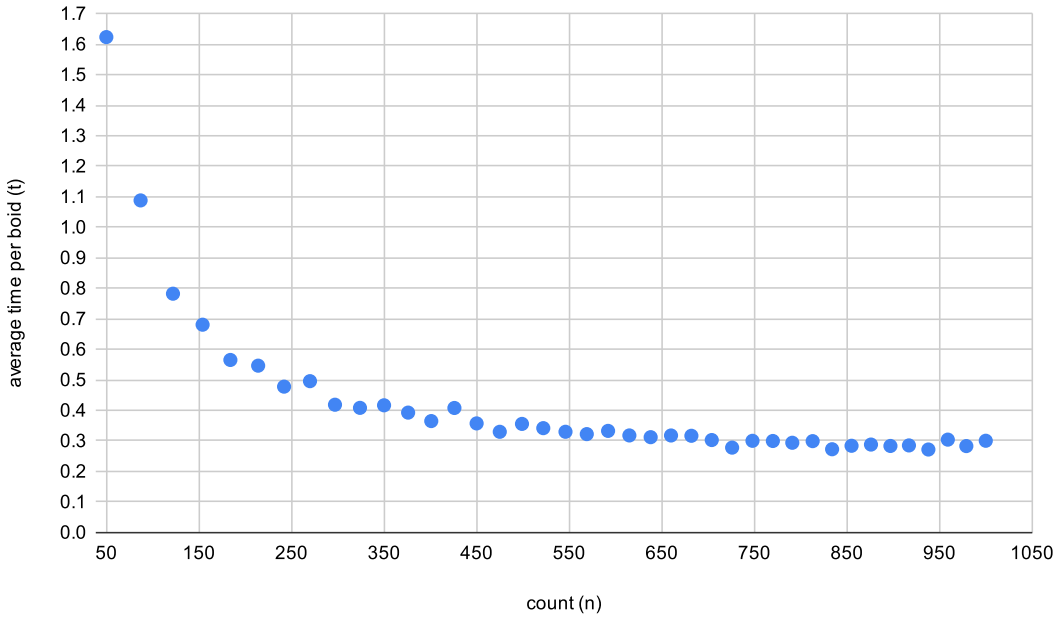

Figure 9.21: K–d tree for $n \in [50, 1000]$, $k = 5$, $\rho = 0.01$.

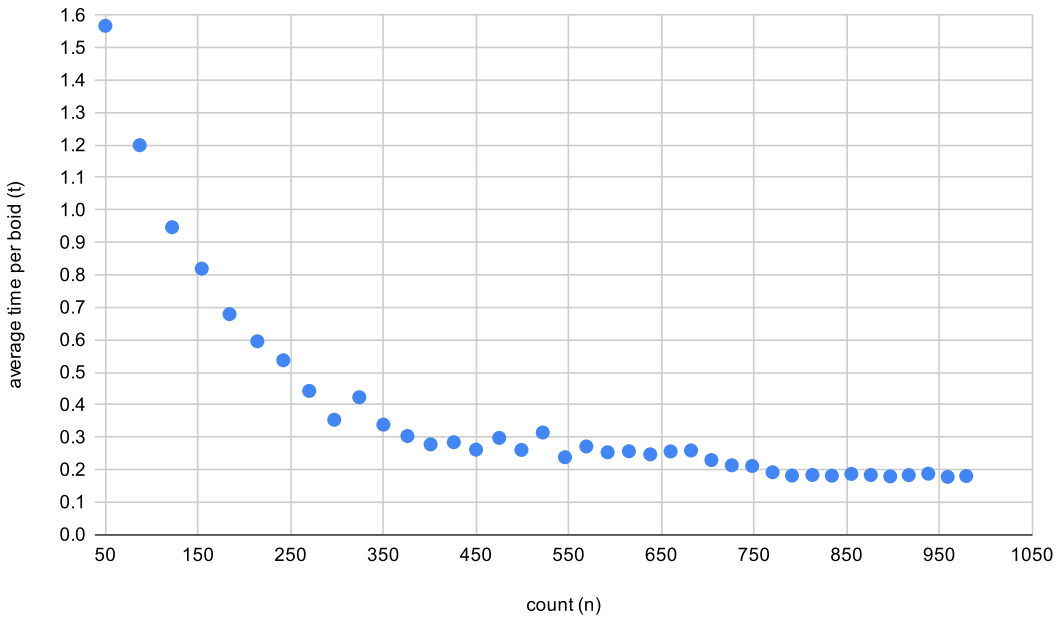

Figure 9.22: Spacial partitioning for $n \in [50, 1000]$, $k = 5$, $\rho = 0.01$.

Based on Figures 9.21, 9.22, it appears that t stabilizes to a constant. We wanted to see if this is true for even larger n. The same set up was tested for $n \in [500, 90000]$. The results are shown in Figures 9.23, 9.24. For k–d trees, t

starts to slowly increase around $n = 10000$. This curve appears to be logarithmic, which is what we expected based on the theory.

For spacial partitioning, at first, t drops further from 0.2 microseconds to 0.1 microseconds where it stays constant until around $n = 70000$. However, afterwards t starts to increase at what appears to be linear or even quadratic rate. Based on the theory, we expected t to drop to a constant as n increases. We did not expected it to increase afterwards. We are not sure what the cause is, but it suggests that perhaps for extremly large flocks, k–d trees might be a better choice, which is opposite to our expectations. Since such large groups were not the target use case of the framework, we did not investigate this further.

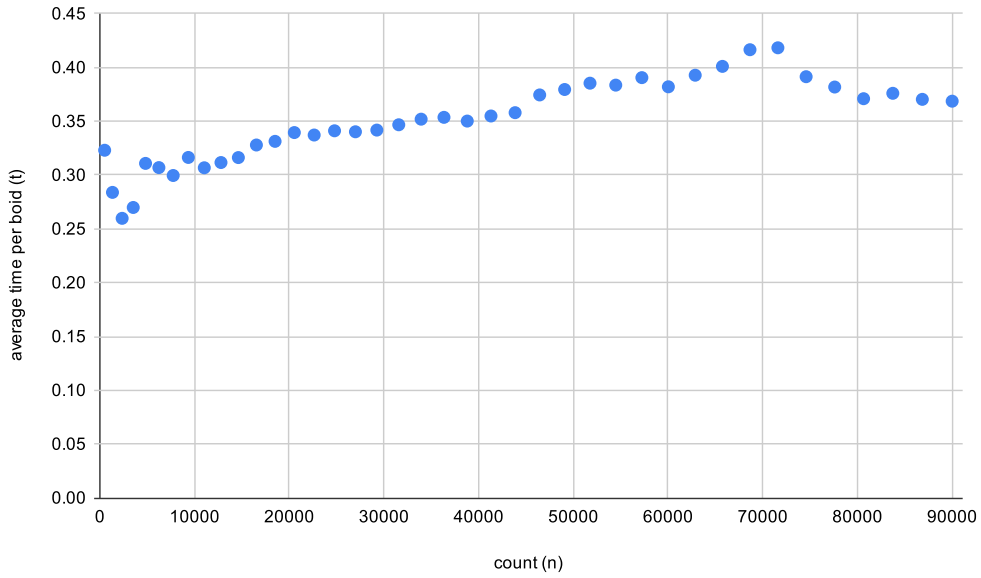

Figure 9.23: K–d tree for $n \in [500, 90000]$, $k = 5$, $\rho = 0.01$.

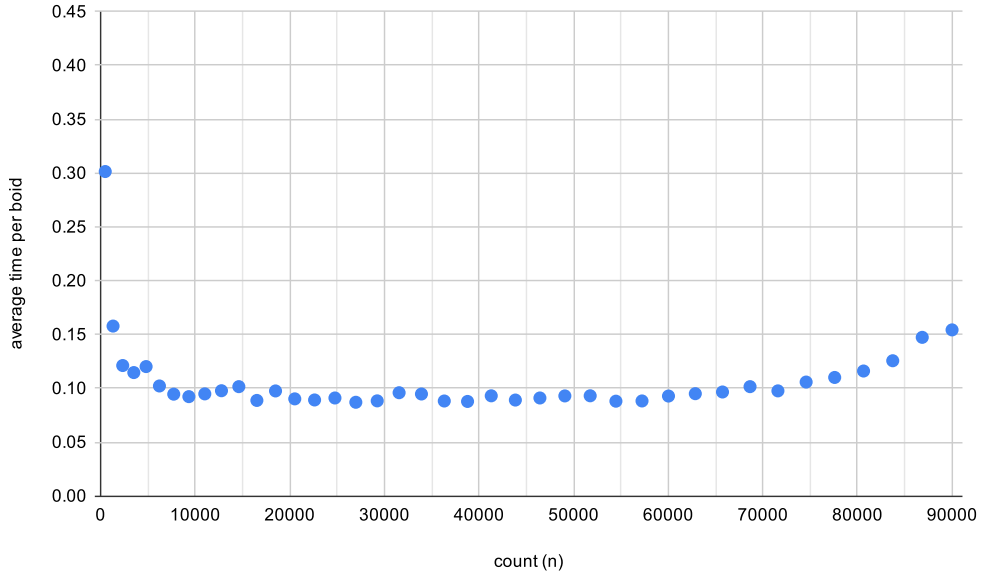

Figure 9.24: Spacial partitioning for $n \in [500, 90000]$, $k = 5$, $\rho = 0.01$.

Variable Density

Based on the previous experiment, it seems that the spacial partitioning implementation is a better choice. However, that experiment only tested a scenario for an arbitrarily chosen density $\rho = 0.01$. Here, we tested how the performance scales with an increasing density $\rho \in [0.01, 2]$ for constant $n = 1000$ and $k = 5$. The results are shown in Figure 9.25, 9.26. In both cases, t appears to grow logarithmically with ρ, and t roughly doubles as ρ goes from 0.01 to 2. In this experiment, spacial partitioning also performed better overall.

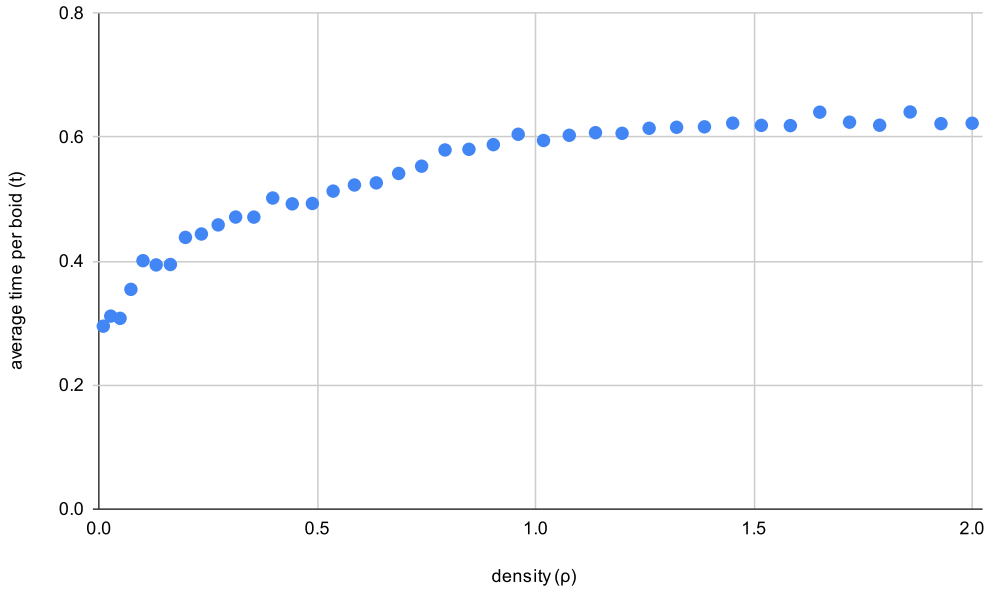

Figure 9.25: K–d tree for $n = 1000$, $k = 5$, $\rho \in [0.01, 2]$

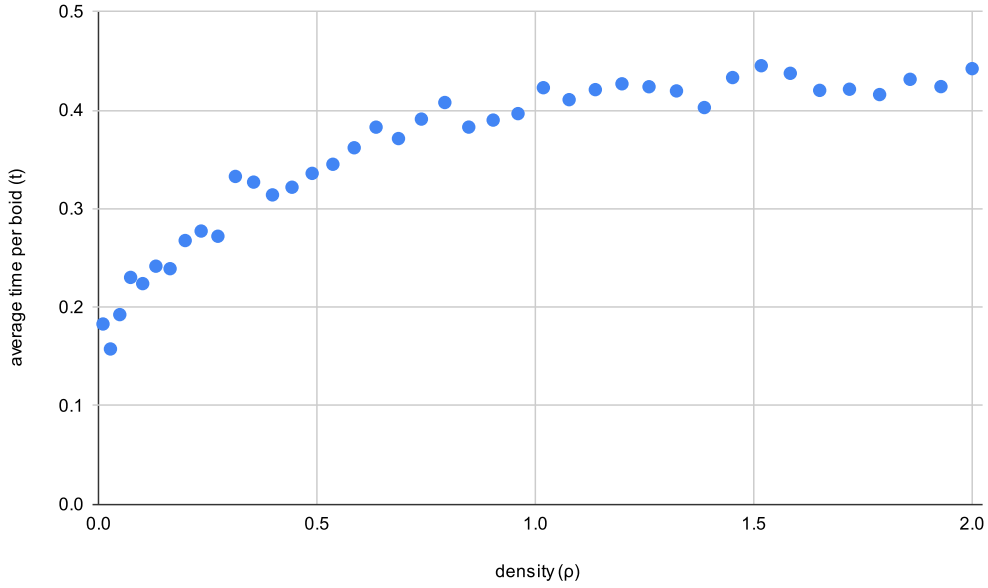

Figure 9.26: Spacial partitioning for $n = 1000$, $k = 5$, $\rho \in [0.01, 2]$

Variable Number of Neighbors

Lastly, we wanted to see how t scales as k increases. This was tested for $n = 1000$, $\rho = 0.2$ and $k \in [0, 1000]$. Larger than usual density of 0.2 was chosen because we noticed that as k grows, the flocks became more cohesive, and we did not want this effect to influence the experiment. At $\rho = 0.2$, the space becomes quite uniformly filled so this should not have a strong impact. In both cases, t appears to scale linearly as k grows until around $k = 250$ and stays roughly constant afterwards. We assume that the point after which t stays constant is for $k > m$, where m is the number of neighbors within a boid's radius of vision. When the same scenario was tested with a larger radius of vision, t continued to grow linearly longer, which supports this assumption further.

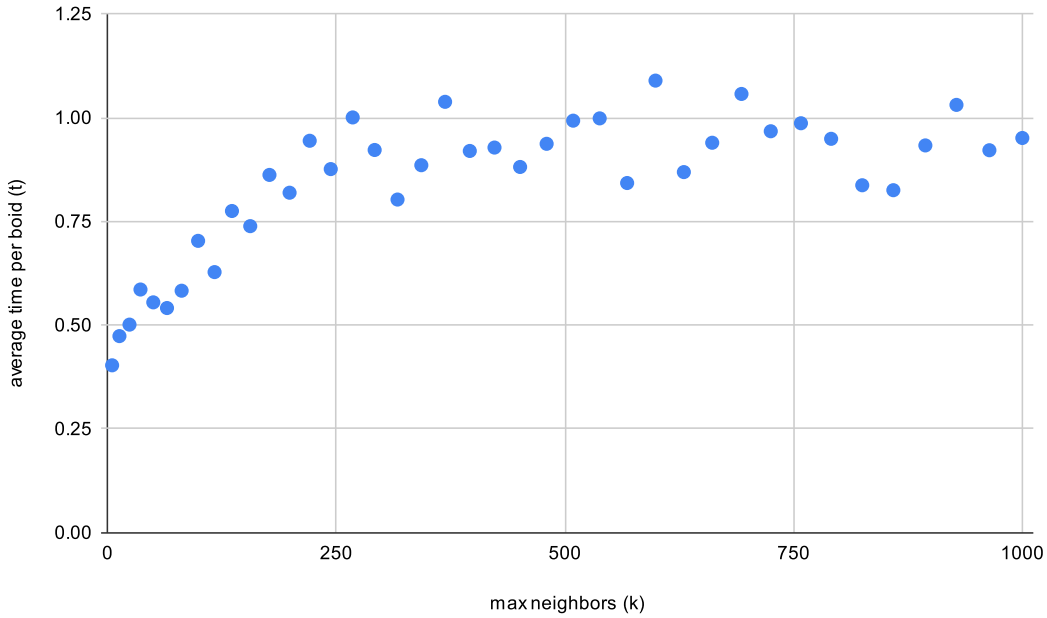

Figure 9.27: K–d tree for $n = 1000$, $k \in [0, 1000]$, $\rho = 0.2$

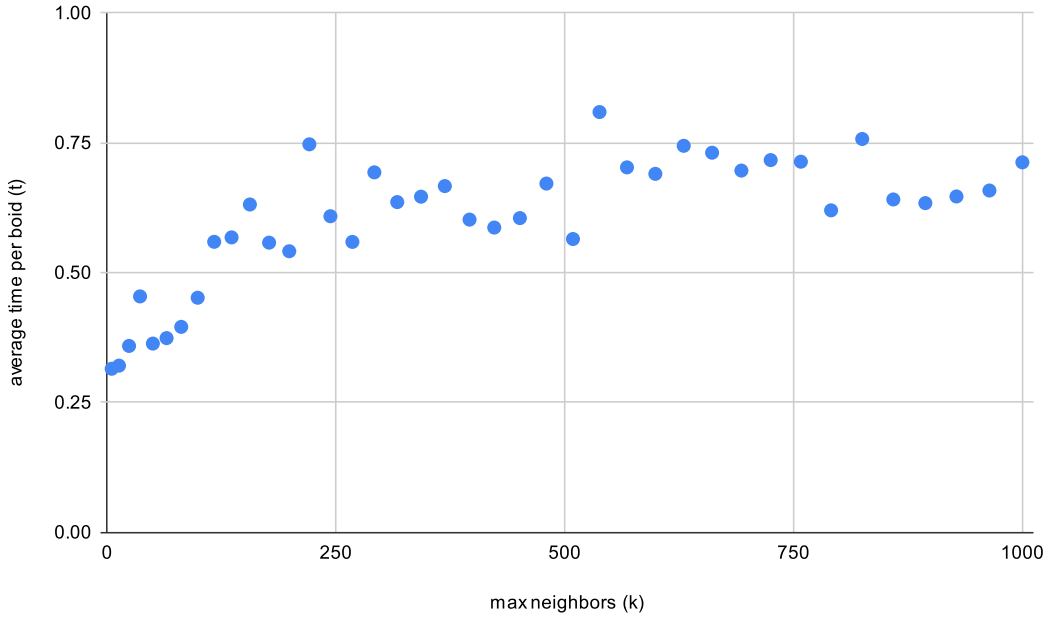

Figure 9.28: Spacial partitioning for $n = 1000$, $k \in [0, 1000]$, $\rho = 0.2$

Conclusion

This subsection conducted a test of two *Neighbor Query* implementations, one based on k–d trees, the other based on spacial partitioning. In all circumstances we tested, the spacial partitioning implementation performed better, and so it seems to be the better choice for our use case of hundreds to lower thousands of boids. However, it is still possible that in some different circumstances or on different hardware the situation would be different.

9.6.2 Full Simulation

One of the goals of this thesis was to create a framework that enables a basic simulation of a flock of 1000 boids under 1 millisecond. The goal was to achieve this on an older gaming laptop with i7-8750H CPU and 32 GB of RAM. We tested this with the same set up as in Subsection 9.6.1. The test for $n = 1000$ was run with $\rho = 0.01$, $k = 5$, $r = 5$. The boids used the three usual behaviors for flocking – cohesion, alignment and separation. On average, the simulation needed around 1.5 millisecond in each frame on the laptop. Therefore, our goal for performance was not entirely achieved, but we are not too far from it. We have two ideas how the performance could be improved.

The implementation of behaviors used in the framework contains many different parameters to experiment with. For example, the exponent for observability and activation functions can be set by the user. Therefore, the implementation has to use `math.pow`. However, we often left the exponent at default value of 2. In that case, `x * x` would run faster. Also, some behaviors have other additional parameters, such as setting a minimum desire of the behavior. For this reason, `math.max` had to be used. This introduces branching, which may also negatively

impact performance. The option to specify a minimum desire could also be removed, at the expense of some additional control over the behaviors.

The aformentioned optimizations could provide some speed up, but it would likely not be too significant. However, we believe that running the simulation on a GPU could improve the performance by orders of magnitude. The biggest challenge would be implementing the k nearest neighbors search on GPU. Afterwards, the rest of the model is "embarrassingly parallelizable", which makes it an ideal task for a GPU.

10. Editor Window Documentation

In the previous Chapter 9, the design and implementation of the framework was discussed. The main idea is that an implementation of `BaseSteeringSystem` loads in a scriptable object `SteeringSystemAsset` (Subsection 9.3.1). The asset contains information about which *Job Wrappers* to run and which entities to target using tags. This is achieved through serialization of the *Job Wrappers* and tags. The *Job Wrappers* are then run by the implementation of `BaseSteeringSystem`.

This chapter will discuss an editor tool created to manage the assets of type `SteeringSystemAsset`. The editor should provide a user friendly way to build the asset in editor, without the need to write code, or know implementation details of the framework. For this reason, it has to be robust, minimizing the possiblity for user's mistakes. Setting up the `SteeringSystemAsset` as a scriptable object through editor has the additional benefit compared to defining it in code: it avoids triggering recompilation with every change, increasing the speed of experimentation. The editor's source files are located in **`com.o-vaic.steering-ai/Editor`** after importing the framework (see Attachment A.1.1).

10.1 User Perspective

This section focuses on the editor window from the point of view of the user, and how it relates to the fields of the `SteeringSystemAsset` class, which were shown in Listing 9.2 in Subsection 9.3.1. Usage of the editor window for a regular user is explained in the attached documentation A.1.3. Figure 10.1 shows a screenshot of the editor window. The list below gives an overview of its main elements. The following Subsection 10.1.1 goes into more detail.

1. The user selects the tag of entities to use here.
2. The user can drag and drop the entity's prefab here.
3. The four tabs enable switching between editing the *Neighbor Behaviors*, *Ray Behaviors*, *Simple Behaviors*, or *Merging*. In the figure, the user is editing the *Neighbor Behaviors*.
4. List of components the entity's prefab should have.

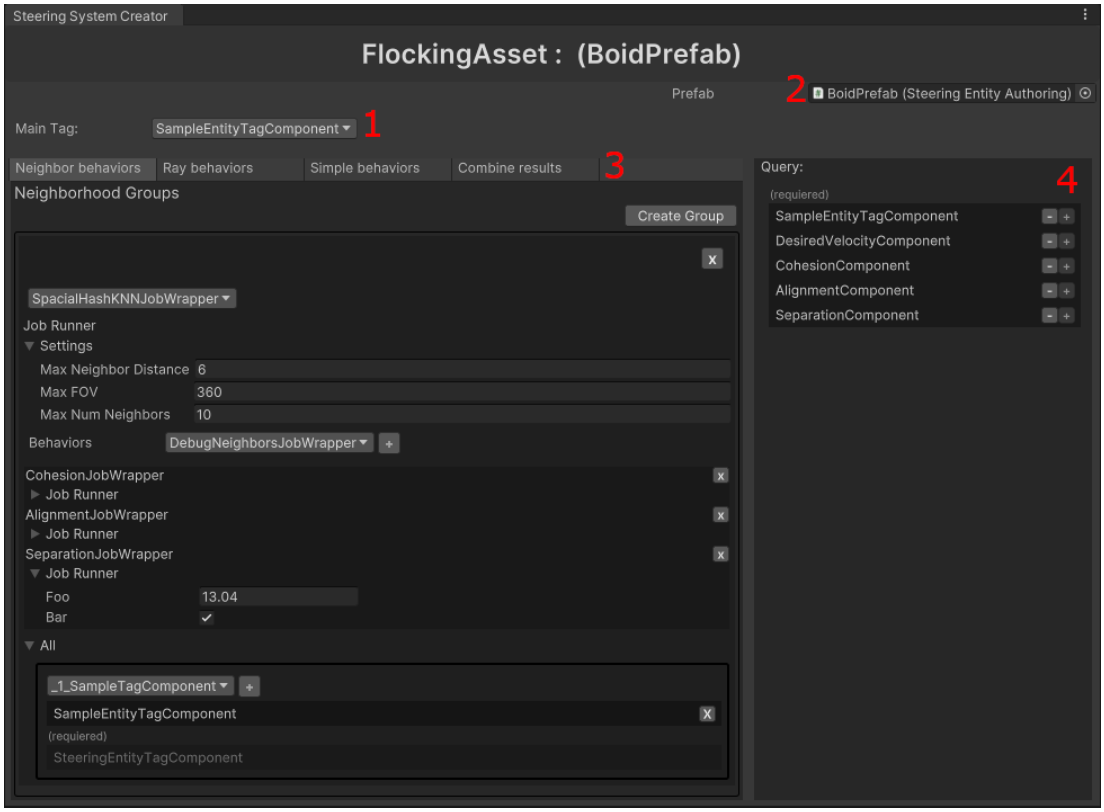

Figure 10.1: Editor for `SteeringSystemAsset`, with *Neighbor Behaviors* tab selected.

10.1.1 Editing Steering System Asset

See Figure 10.2, focusing on the area where the user can edit *Neighbor Behaviors*. The list below describes all the elements of one group of *Neighbor Behaviors*.

1. The "Create Group" button adds a new group of *Neighbor Behaviors*.
2. A dropdown containing all implementations of `INeighborQueryJobWrapper`, with `SpacialHashKNNJobWrapper` currently selected for this group.
3. Settings for the selected implementation of `INeighborQueryJobWrapper`, such as maximum distance, can be set here.
4. A dropdown of all implementations of `INeighborBehaviorJobWrapper`. In the figure, `DebugNeighborsJobWrapper` is selected. Clicking the "+" button adds the selected behavior to the list below.
5. A list containing instances of `INeighborBehaviorJobWrapper` added for this group. The list shows the three flocking behaviors. As shown with the separation behavior, a *Job Wrapper* can have extra serialized properties associated with it.
6. Two lists of components used by an `EntityQuery` to match potential neighbors. The query matches entities with any component from list (a) and all components from list (b).

(a) This list allows adding tags of potential neighbors, using the dropdown above it.

(b) This list shows all components that must be attached to the potential neighbor, as inferred from the *Job Wrappers* used. How this is inferred is discussed in Section 10.2.

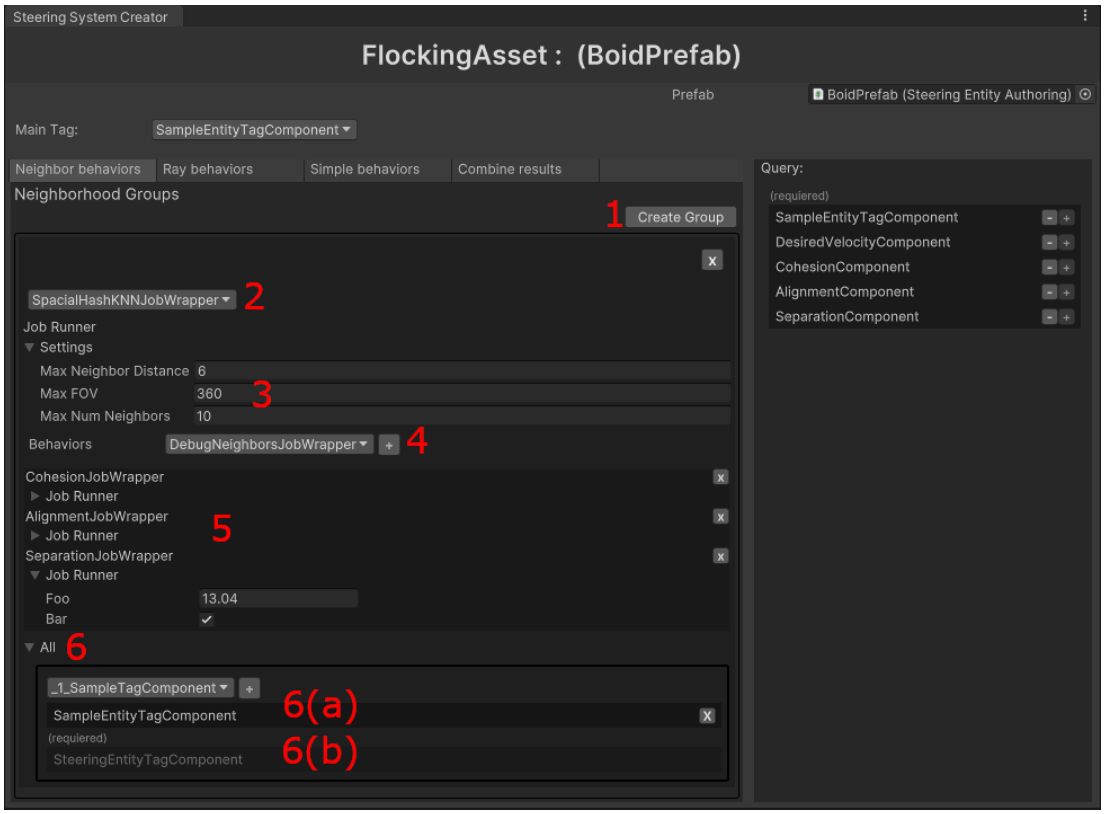

Figure 10.2: Editor for `SteeringSystemAsset`, with *Neighbor Behaviors* tab selected.

Above, only the "Neighbor behaviors" tab was described. The "Ray behaviors" tab is set up in a very similar way. The user can also add groups, and then instead of selecting a `INeighborQueryJobWrapper` for it, the user selects an implementation of `IRayQueryJobWrapper`. Afterwards the user adds behaviors which are implementations of `IRayBehaviorJobWrapper`, instead of `INeighborBehaviorJobWrapper`. Under the "Simple behaviors" tab, implementations of `ISimpleJobWrapper` can be added into a single list. In the "Combine results" tab, the user can select an implementation of `IMergeJobRunner` to use.

As previously mentioned, the list marked with number 4 in Figure 10.1 contains all components which should be attached to the entity's prefab. This is inferred from all the *Job Wrappers* used by the asset (see Section 10.2). The plus and minus buttons next to the components allow adding or removing the components from the entity's prefab.

Merging and Workflow

By selecting the implementation of `IMergeJobRunner` under the "Combine results" tab, the user also implicitly selects the "workflow", that is what types of results

the behaviors return. That could be multiple different types, if the implementation of `IMergeJobRunner` allows it, but normally, if using the implementation of from *defaults*, it would be the `VelocityResults`. After the implementation is selected, the editor only displays behaviors which return this type of results in all dropdowns where behaviors can be selected (under tabs "Neighbor behaviors", "Ray behaviors", "Simple behaviors"). This is a sort of type checking, to prevent the user from selecting behaviors incompatible with the currently selected implementation of `IMergeJobRunner`.

10.2 Reflection

The previous section suggests that the editor needs to query types in the project. For example, to find all implementations of `INeighborBehaviorJobWrapper` compatible with the currently selected implementation of `IMergeJobRunner`. This can be achieved through meta programming. In C#, meta programming can be done through reflection [79]. An important concept here is attributes [80], which can be used to store additional information about a type. For example, an attribute on a `INeighborBehaviorJobWrapper` can specify what type of results it uses. The editor can then use reflection to find all types of implementations of `INeighborBehaviorJobWrapper` compatible with the selected implementation of `IMergeJobRunner`. This section focuses on the attributes used by the framework.

10.2.1 Steering Entity Tag Attribute

In Unity's ECS, a tag component is a component with no fields. There can be many tag components for different purposes. Marking a tag component with `SteeringEntityTagAttribute` makes it show up in all the editor's dropdowns where a user can select a tag. For example the dropdown in top left corner of Figure 10.1. This will filter out other tag components in the project, which are not supposed to be used with this framework.

10.2.2 Component Authoring Attribute

An ECS component is a struct implementing the `IComponentData` interface. It is possible to attach the components to entities in code, but Unity also offers a way to attach them to prefabs in the editor. This can be achieved through classical `MonoBehavior` on the prefab, which are responsible for adding the ECS components to the entity. The process is called *baking*, refer to Unity's documentation [81] for more details. However, it is possible that there are multiple `MonoBehavior` components which add the same ECS component to an entity. It is also possible that one `MonoBehavior` component adds multiple ECS components to an entity. In other words, there is no 1 to 1 mapping between `MonoBehavior` components and ECS components.

See again the list of required components on the right in Figure 10.1, the user can add a missing ECS components to the prefab, or remove an unncesessary one. The `ComponentAuthoringAttribute` is used to mark a `MonoBehavior` component, which adds the specific ECS component to an entity. The attribute's constructor takes in the type of the ECS component which the `MonoBehavior` adds. This allows

the editor to find the right `MonoBehavior` component to add or remove from the prefab.

10.2.3 Job Wrapper Attribute

The `JobWrapperAttribute` is used to mark the *Job Wrappers* which should be displayed in the editor's dropdowns. The attribute also contains types of components the entity using this *Job Wrapper* should have. These are then collected by the editor and displayed in the list on the right in Figure 10.1. In the case of *Neighbor Behaviors*, it also contains types of components the neighbors must have. These would be displayed in the list 6(b) in Figure 10.2. In the figure, all neighbors must have a `SteeringEntityTagComponent`.

10.2.4 Out Data Attribute

The last attribute that the editor uses is the `OutDataAttribute`. All behavior *Job Wrappers* should have this attribute. It specifies what type of results the behaviors return, normally it is `VelocityResults`. The implementation of `IMergeJobRunner` should also have this attribute, but there, it declares what types of results it can merge. After selecting the implementation of `IMergeJobRunner`, the editor will only display behaviors which return the compatible results. For more complex workflows, the `OutDataAttribute` on implementation of `IMergeJobRunner` can declare multiple types, if it can handle multiple different types of results.

10.3 Serialization

So far, the `SteeringSystemAsset` was described, together with how it can be editted through the editor window. The asset still needs to have some way of being saved and loaded from the disk. The idea is that the user creates the asset, edits it in the editor window and saves it. We tried two approaches, the first one using JSON serialization with C# code generation, and the second one using Unity's serialization.

10.3.1 JSON Serialization with Source Generation

Initially, we experimented with serializing all the data to a JSON file, and then using this JSON file to generate code with C# source generators [82]. The generated code was a class implementing the `BaseSteeringSystem`, which contained a generated instance of a `SteeringSystemAsset`. The generated instance would look for example like Listing 9.2 from Subsection 9.3.1. This approach was abandoned for multiple reasons. First, it was not clear how serialize fields of the *Job Wrapper* classes. However, the main reason was that each time the asset was saved, the newly generated code had to be recompiled. This would slow down the iteration process, when experimenting with different configurations.

10.3.2 Unity Serialization

The second approach to serialize an instance of `SteeringSystemAsset` was to use serialization which is already built into Unity. The idea is that `SteeringSystemAsset` inherits from `ScriptableObject` [83]. Classes inheriting from `ScriptableObject` can be saved as an asset in the project, and loaded at runtime. This has a number of benefits. First, Unity provides ways to save, edit and load `ScriptableObject` assets, making the editor window easier to implement. Second, users of Unity tend to be familiar with `ScriptableObject` assets, rather than JSON files from the first attempt. Most importantly, because source generation is not used, there is no need for recompilation as was the case with the other approach.

JobWrapper Serialization

Normally, when an asset is selected in Unity, the default inspector window displays serialized fields of the asset. As long as the fields are marked as `Serializable`, Unity can handle them by default. However, in `SteeringSystemAsset`, the fields holding the instances of *Job Wrappers* are lists of interface types. Interfaces are not `Serializable`, and so Unity cannot handle them the same way. Unity provides an attribute `SerializeReference` [84] for these fields, which partially solves the problem.

This attribute allows Unity to handle interface fields – the `Serializable` fields of the concrete type will be displayed. However, Unity lacks a way to select a concrete type to use, an instance has to be assigned in code. This is one of the reasons why the editor window was created. In our editor, dropdowns are used to select the concrete type to instantiate. The concrete types are looked up through reflection, and they are instantiated with `Activator.CreateInstance` [85]. Then, the instances are assigned to the `SerializeReference` fields on the `SteeringSystemAsset`.

10.4 Implementation

This section discusses the implementation of the editor window. Since 2022, Unity provides a new way for creating UI, including editor windows, called *UI Toolkit* [86]. It uses an *XML* based language called *UXML* to define the UI elements. In C#, it is possible to interact with the UI elements, and bind an object's fields to them. There is also a visual editor for creating UI without writing the *UXML* code directly. We decided to use this new system for the editor window. The main motivation was that the visual editor makes it easier to layout the UI elements.

10.4.1 UI Elements

There are some important UI elements that were used when creating the layout for the editor window. They will now be briefly described.

PropertyField

Perhaps the most important UI element is the `PropertyField` [87]. When a field of an object is bound to a `PropertyField`, the default Unity serialization is used

to display and edit the field. This is for example how all the instances of *Job Wrappers* are handeled in our editor window.

Dropdown

One simple, but important, element is the `Dropdown` [88]. Dropdowns are, for example, used to select a concrete type of a *JobWrappers* to use. The dropdowns are populated with a an array of elements, which are displayed and can be selected by the user.

ListView

The `ListView` [89] is used to display lists of elements. In our editor window, they are used for all the lists within the editor window, like the list of all *Neighbor Behaviors* within a group.

10.4.2 Editor Window

The editor window is implemented in class `SteeringSystemEditor` which inherits from `EditorWindow`. After double clicking on an `SteeringSystemAsset`, the window is opened and the asset is loaded with `AssetDatabase.LoadAssetAtPath`. Next, all properties are bound to their respective UI elements. For example, a `ListView` for the *Simple Behaviors* is initialized, and bound to it is a list of `ISimpleJobWrapper` instances from the `SteeringSystemAsset`. There are, however, some UI elements requiring extra work. These exceptions are discussed in the next subsections.

Dropdowns

For simplicity, consider a dropdown for selecting the type of `ISimpleJobWrapper` to add. Dropdowns for other *Job Wrappers* work similarly. The dropdown is populated with all the types which are compatible with the selected `IMergeJobRunner`, as described in Section 10.2 when discussing reflection. Furthermore, all the implementations of `ISimpleJobWrapper` that are already in the list of *Simple Behaviors* are filtered out. Whenever a new behavior is added into the `ListView`, the dropdown has to be refreshed to filter out the added behavior. Lastly, all the dropdowns for selecting the types of behaviors need to be refreshed whenever an instance of `IMergeJobRunner` has changed.

Components List

The `SteeringSystemAsset` object does not contain information about which components the entities should have directly. This information is inferred from the attributes of selected *Job Wrappers*. To populate the list, the editor finds union of all the required components for all the *Job Wrappers* used, based on their `JobWrapperAttribute`. Then, for each required ECS component, it finds the `MonoBehavior` component which adds this ECS component to an entity. These `MonoBehavior` components are looked up using the `ComponentAuthoringAttribute`. The editor then looks at the selected prefab, and if there is no `MonoBehavior` component for a required ECS component, a plus button is displayed next to the component. On click, the editor adds this `MonoBehavior` to the prefab. Once this

is done, the minus button is displayed to remove the given `MonoBehavior` from the prefab. The list is updated whenever a new *JobWrapper* is added or removed, or when the selected prefab changes. To add or remove a component from a prefab, the editor uses Unity's `ObjectFactory.AddComponent` and `Object.DestroyImmediate` methods.

11. Sample Game Scene

One the goals introduced in the first Chapter 1 was to set up a game scene which would demontrate the capabilities of the framework. This chapter outlines what is in the game scene, and how the framework helped with creation of the scene.

The sample game scene's source files are located in **`Game/GameProject`**, see Attachment A.2.1. Builds of the project can be found under **`Game/Builds`**, see Attachment A.2.2. The **`Builds`** folder contains a build of the attached **`GameProject`** folder, and a second build using better graphical assets, which could not be included in the attachments due to their licensing restrictions. The figures in this chapter use screenshots from the version without better assets.

11.1 Game Scene

The game scene is a simulation of a small ecosystem. There is a 3D environment with several types of animals moving around. All the animals in the scene react to the environment, each other, and the player.

11.1.1 Environment

The game environment is a simple 3D world with a terrain and a pond (see Figure 11.1). There are patches of grass placed around the terrain, and obstacles in form of stones and trees. Steep mountains surround the world, creating a natural border.

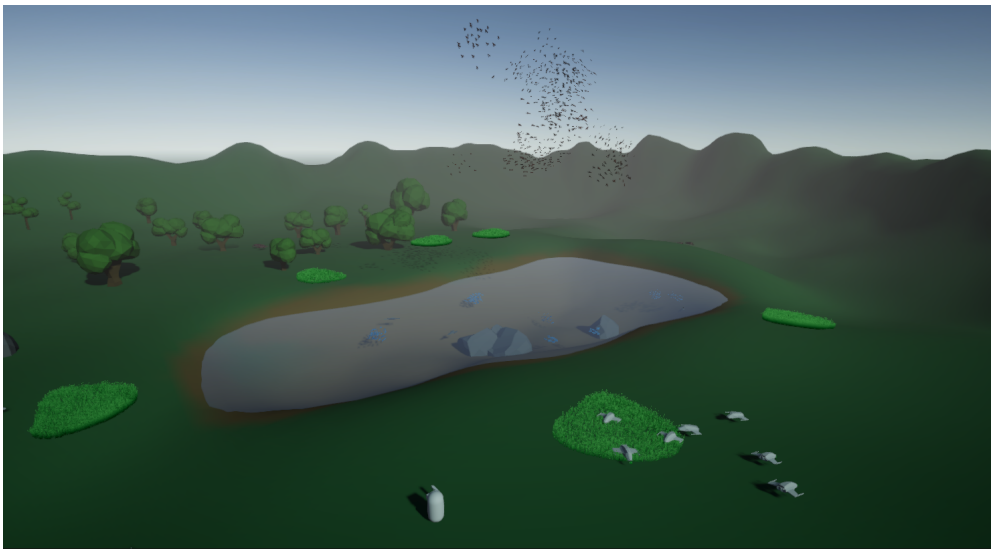

Figure 11.1: Sample game scene.

11.1.2 Player

The player, Figure 11.2, is a 3D character who can run around the world. He can do two actions - throw an apple and fire a gun. Firing the gun at an animal kills it, which makes the animal fall to the ground. Throwing the apple will make some animals chase it.

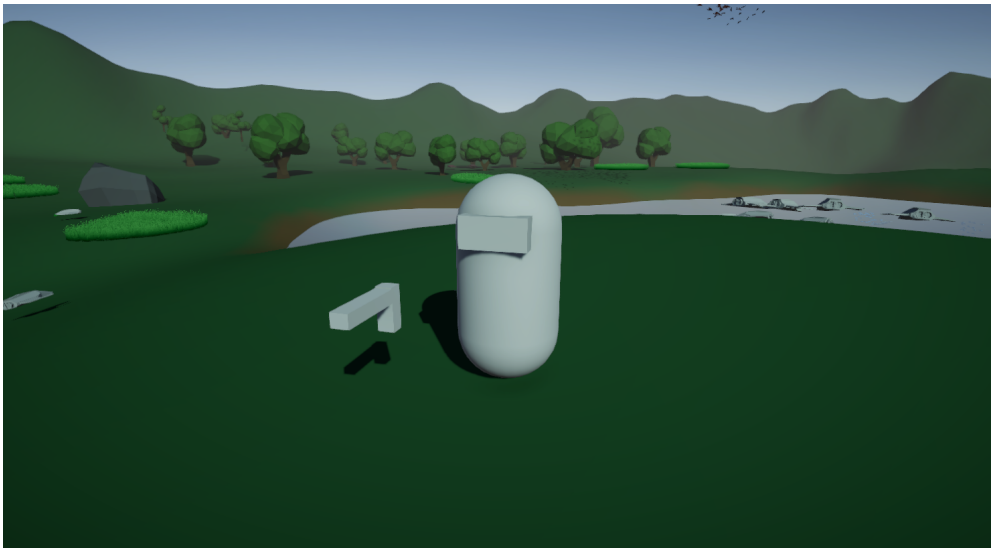

Figure 11.2: The player's character.

11.1.3 Birds

Birds are the only flying animals in the scene. They are the most numerous animals, with two groups of 600 birds each. Figure 11.3 shows one of the groups. The birds exhibit flocking behavior, they are aware of where their "home" area is, and do not stray too far from it. If a gun shot hits anything near the birds, they scatter.

Figure 11.3: Flock of birds.

11.1.4 Fish

Fish, pictured in Figure 11.4, are the second most numerous at 200 individuals in the pond. They exhibit the same flocking and homing behavior as the birds. Morever, they are afraid of the player, loud sounds nearby, and most importantly the sharks, and will swim away from each of these stimuli. They, however, see a dead shark as a potential food source, together with apples, birds and sheep.

When a food source is near, the fish swim towards it and bite it. The tendency to go after food is stronger the hungrier the fish is. Once they have taken a bite of food, they will take a while to chew it. Figure 11.4 shows a fish swimming with pieces of sheep in their mouth. While chewing, they do not search more food, and try to avoid other fish. This is because they consider even the food inside their school mates' mouths as potential food, and may try to steal it.

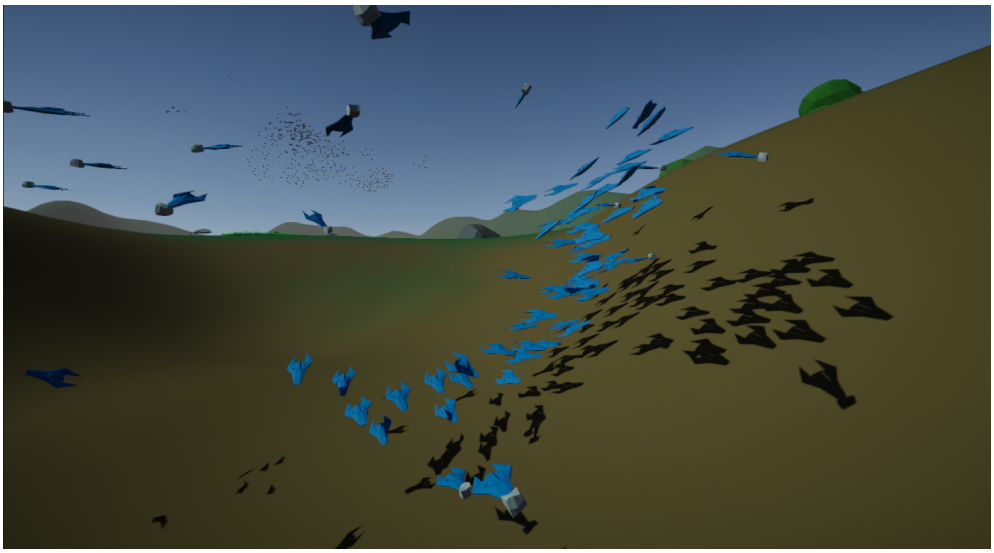

Figure 11.4: Fish with pieces of sheep in their mouth.

11.1.5 Sharks

Sharks are the predators of the pond. There are three of them in the pond. They use the same homes as fish, keep their distance from the other shark (especially when eating), and swim away from sounds and the player. When hungry, they hunt for sheep swimming on the surface, try to catch fish, or go for shot down birds. Their largest maximum speed in the scene is balanced out by their inability to turn as fast as the fish, and their narrow field of view. This gives the fish a chance to escape.

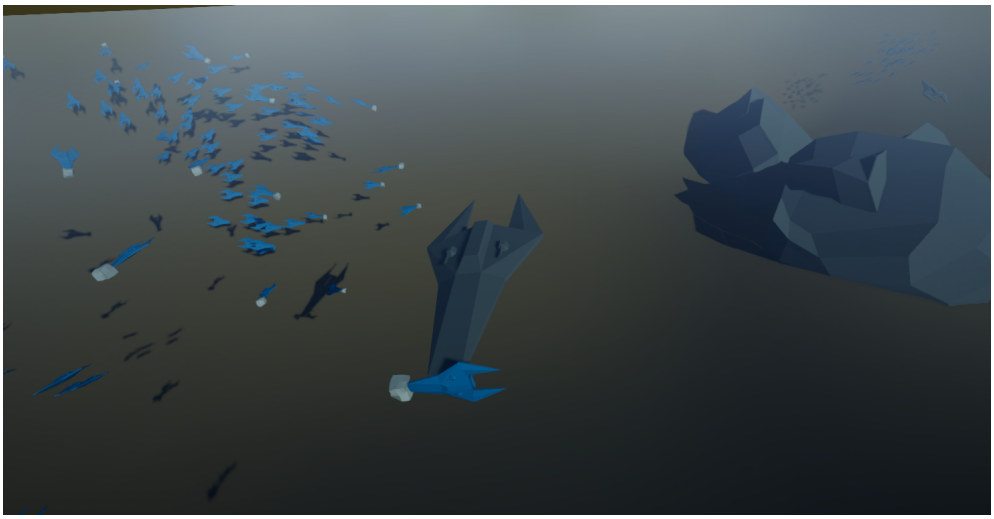

Figure 11.5: Shark with a fish in its mouth.

11.1.6 Sheep

There is 160 sheep walking around the terrain. They have flocking behavior, but no homing to a specific area. They are afraid of the player, the wolves, sounds, sharks and the fish. They eat apples and grass, and after finding a grass patch they slow down to eat it. Same as other animals, they avoid other sheep when eating. The sheep can also swim on the surface of the pond, but the movement is slower than on land.

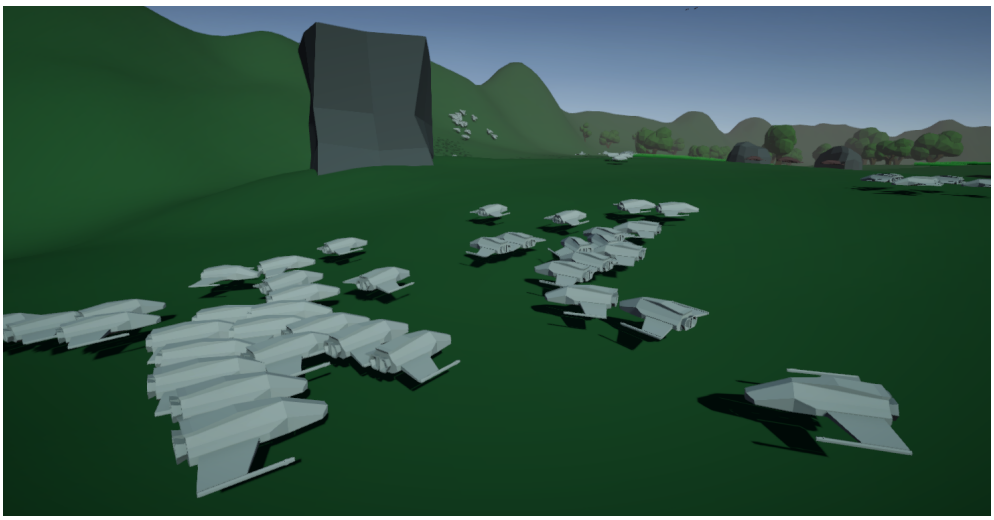

Figure 11.6: Flock of sheep.

11.1.7 Wolves

There are 8 wolves that serve as the predators of the sheep. One wolf is shown in Figure 11.7. The wolves flock, but their tendecy to it is not very strong. Their main food source are the sheep which they hunt. Dead birds are the second possible food source. As other animals, they try to keep their distance while eating. Wolves try to avoid the player, but not as strongly other animals. Unlike other animals, wolves are not afraid of sounds. As for the movement, they cannot swim, and will not enter the pond. Similarly to sharks, they are the fastest animals on the terrain, which is balanced by their low maneuverability, and lower field of view for hunting.

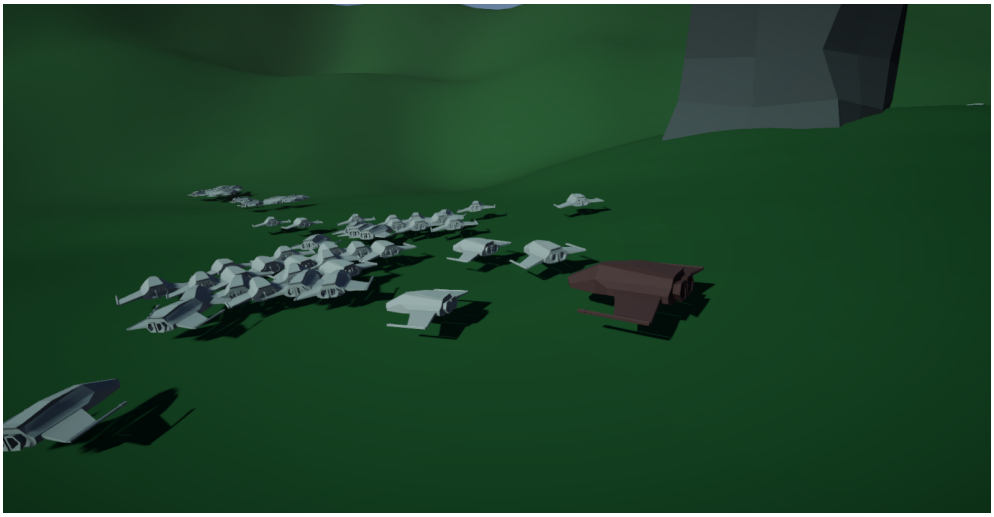

Figure 11.7: Wolf scaring a flock of sheep.

11.2 Framework Changes and Additions

Chapter 9.5 described behaviors and movement systems provided by the framework. While the framework aims to provide a solution that does not need any additional programming, there was still a need to modify and add some behaviors and movement systems for this specific game scene. This section describes changes and additions specific to this game scene, which were not included in the framework's *defaults*.

11.2.1 Abandoned Context Steering Test

While creating the example game, we also experimented with a relatively novel approach to steering behaviors called *Context Steering* [26], introduced by A. Fray in the book titled "Game AI Pro 2". This was done especially to try if our framework can handle other approaches to steering behaviors than the ones the framework was designed for. While this approach is interesting, we found no significant improvements which would justify the added complexity. For this reason, we decided to not use it for the game scene, and to not include it in the framework by default. Therefore we will not go any further into description of this approach.

11.2.2 Movement System with Energy

The movement system for 3D, `Move3DSystem`, was editted to emulate getting tired while running at high speed. This mechanic was added especially to make the predator-prey interaction between the shark and fish more dynamic and interesting. It creates a sort of cooldown period when the shark has to regain energy before being able to hunt.

To do implement this, each entity has an energy value associated with them. Energy is depleted more, the faster an entity moves, and is recovered more, the slower it moves. An entity's maximum speed and acceleration decrease as the energy decreases.

11.2.3 Chase Food Behavior

An additional *Neighbor Behavior* for chasing food (`ChaseFoodJobWrapper`) was created, so that the animals can prioritize between different food sources. Food are entities with `FoodComponent` on them. The component contains the food's "nutriousness". The behavior accumulates a weighted average of directions towards food sources. A food's weight grows with the food's nutriousness and proximity. The final desires and speed grow with proximity to the food source with the highest weight.

An important aspect is that the behavior takes into account a `MouthComponent`, which entities using the behavior must have. The component contains, among other information, a position of the entity's mouth. The direction and distance to the food is calculated from the mouth's position. Calculating it from the center of the entity was especially problematic for sharks, whose mouth is far from the center. Figure 11.8 illustrates the difference. In the figure, $\vec{v_{d1}}$ is the desired velocity calculated from the center, $\vec{v_{d2}}$ is calculated from the mouth. Note that moving in direction of $\vec{v_{d1}}$ would not result in catching the fish.

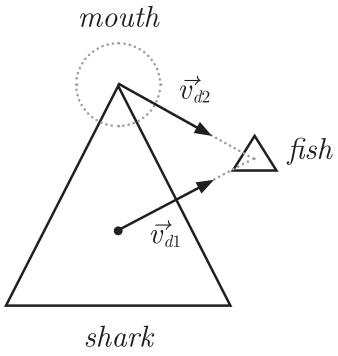

Figure 11.8: Using mouth position to get better desired direction towards food. Boid's position to fish – $\vec{v_{d1}}$. Boid's mouth position to fish – $\vec{v_{d2}}$.

11.2.4 Avoid When Eating Behavior

This *Neighbor Behavior* (`AvoidWhenEatingJobWrapper`) was added to make the animals avoid each other when eating. We added this behavior to, for example, make fish that already took a bite swim away, letting other fish reach the food as well. The behavior accumulates a weighted sum of direction away from neighbors, weight of a neighbor is larger the closer it is. If the current entity has a `MouthComponent` with food, the summed direction is returned, and the entity speeds up. The desires for this behavior grow with sum of the weights.

11.2.5 Avoid Sound Behavior

This *Neighbor Behavior* (`AvoidSoundJobWrapper`) makes the animals avoid sounds. In the game scene, the only sound source is the player's gun. An entity representing sound spawns at player's position and the position of the bullet's impact. The sound is represented by a sphere that expands until it disappears after some time. The entities try to get away from it more, the closer and the louder it is.

The behavior uses a weighted average of all the sounds detected. Furthemore, it makes the entities speed up.

11.2.6 Get Pushed Behavior

The *Get Pushed* behavior (`GetPushedJobWrapper`) is a *Neighbor Behavior*, which tries to simulate animals getting pushed from behind by their neighbors. It was created specifically for sheep, to avoid a problem where the player scared sheep from behind, but they could not move forward because of the sheep in front of them. Note, that this was not a problem for example with the fish, as they can swim over one another. The user can set a "maximum push angle" α illustrated as a cone in Figure 11.9, the cone is always behind the entity's direction of movement. The current entity speeds up away from entities within the cone, if their speed is higher. The desired direction and speed are calculated based on weighted averages. The neighbor's weight grows with how much faster it is, how much it is directly behind, and how close it is.

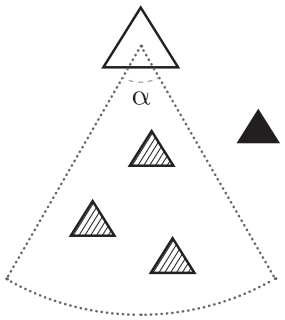

Figure 11.9: A boid (white) and its neighbors considered for *Get Pushed* behavior (striped) within a cone with angle α.

11.2.7 Find Grass Behavior

The game contains patches of grass, this *Ray Behavior* (`FindGrassJobWrapper`) is only used by the sheep which eat the grass. The *Chase Food* behavior is not ideal for grass, because it targets the center of the object. That is not ideal, because the patch of grass can be large, and we wanted to avoid sheep clumping in its center.

The *Find Grass Behavior* handles two cases. One, it makes the sheep go towards nearby grass patches detected by ray casts. Second, when a sheep is above a grass patch and hungry, it makes the sheep slow down to eat it.

11.2.8 Avoid Player Behavior

A special *Neighbor Behavior* (`AvoidPlayerJobWrapper`) for avoiding the player was created. This was done especially because we wanted better control over how much, and at what distance the player is avoided, so other behaviors like fleeing could not be reused. The entities speed up away from the player, the closer he is. If the player is not moving, the animals will reduce the radius at which they

start avoiding the player. This simulates the player being less threatening when not moving.

11.3 Set Up of Animals

So far, the role of each animal type was discussed (Section 11.1), afterwards, changes and additions to the framework were outlined (Section 11.2). This section describes how the framework and the added behaviors were used to set up each animal type in the scene. The main focus is on the parameters of the movement systems, and the behaviors. Due to the large number of different parameters each behavior has, only the most important ones are mentioned. For example, see Table 11.2, showing *Neighbor Behaviors* for the fish. The table is structured in the following way. The **Tag** column shows which entities the behavior targets, **D** is the maximum distance to them, **P** is *priority* of the behavior, **WD** and **WS** are weight multipliers for *direction desire* and *speed desire* respectively.

The way the behaviors are implemented, the maximum direction and speed desires are always **WD** or **WS** respectively. Note, our method of merging Behaviors' results, described in Section 4.4, assumes a maximum sum of desires. In the framework, this is implicitly one. Therefore, the highest desire each behavior should return is one. For example, if a behavior with the highest priority returned one for desire, no other behaviors would have any effect on the final desired velocity. For these reasons, all **WD** and **WS** weights are in normalized $[0, 1]$ range.

11.3.1 Fish

The fish are tagged with a `FishTag` tag. Description of their role in the game scene can be found in Subsection 11.1.4. They consist of 14 different behaviors, which makes them the most complex animals in the scene.

Fish Movement

The fish use `Move3DSystem` for movement. The values for the associated component are shown in Table 11.1 below. The specific parameters for movement were chosen to make interaction with the shark interesting. The fish's maximum speed is lower than the sharks', but they still have a chance to outswim them, depending on the energy levels. Their acceleration in the parallel diretion is smaller than the sharks', but their acceleration in the lateral direction is higher to give them chance to outmaneuver the sharks.

To make the interaction even more interesting, the fish consume energy faster than the sharks, but they also recover it much faster. The fast recovery is done to make sure that the fish are essentially always ready to speed up. This emphasizes the initial burst of speed after first detecting a predator. The fish's fast energy depletion gives the shark a better chance to catch it during long pursuits.

Maximum Speed	5.5
Maximum Parallel Acceleration	3
Maximum Lateral Acceleration	8

Table 11.1: Parameters for movement of fish.

Fish Neighbor Behaviors

Table 11.2, shows the set up of *Neighbor Behaviors* for the fish. There, are the usual three behaviors for flocking, which all target the *Fish* tag. The specific values were found through experimentation to achieve highly cohesive and aligned flocks where fish do not collide with one another using the *Separation* behavior. High cohesion is achieved through high maximum distance and weight, but when there are neighbors near, the *Alignment* behavior with higher priority can take over. If the neighbors are dangerously close, the separation behavior can take over due to its high priority.

The *Multi Homing* behavior has the lowest priority and low weight. Its main purpose is to slightly bias the fish towards center of the pond, which looks more natural, and helps with keeping a few bigger cohesive groups. The home areas are shown in Figure 11.10. Blue shows the maximum radius where the home can be detected, green shows the minimum radius where fish have zero desire to go towards the home. The *Chase Food* behavior has a high weight and medium priority to emphasize chasing food, while leaving higher priorities for avoiding danger. Note the behavior's *Dead Shark* tag. After a shark dies, this tag is added and its *Shark* tag is removed, to stop fish from avoiding the shark.

The behaviors *Avoid Sound*, *Avoid Player*, *Flee* and *Avoid When Eating* are all aimed at avoiding danger. Avoiding the player and sounds comes secondary to fleeing from the shark. The *Flee* behavior has a high weight to get a very strong predator response whenever possible. It is however lower priority than *Separation* to make sure some distance is kept between the fish. Lastly, *Avoid When Eating* has the highest priority. This is only done to always bias the fish towards avoiding other fish when eating. The weights, however, are low, to let even lower priority behaviors like *Flee* overpower it if needed.

Behavior	tag	P	WD	WS	D
Cohesion	Fish	0	0.8	0.3	20
Multi Homing	FishHome	0	0.2	0	20
Alignment	Fish	1	0.6	0.2	5
Chase Food	Apple Sheep Bird DeadShark	2	0.8	0.65	20
Avoid Player	Player	2	0.4	0.2	10
Avoid Sound	Sound	2	0.65	0.65	10
Flee	Shark	3	0.8	0.7	6.5
Separation	Fish	4	0.6	0	1.5
Avoid When Eating	Fish	5	0.3	0.3	3

Table 11.2: Fish's *Neighbor Behaviors* and their main parameters.

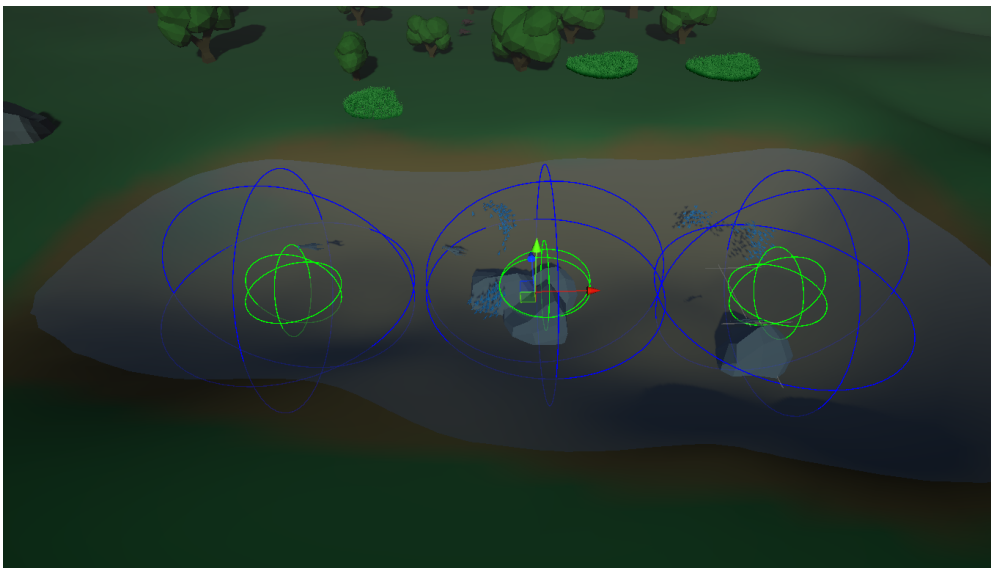

Figure 11.10: Three homing areas inside the pond. Homes' maximum radius in blue, minimum radius in green.

Fish Ray Behaviors

The fish have two *Ray Behaviors*, shown in Table 11.3. Both are aimed at avoiding the physical environment of the scene. Note that both behaviors have priorites especially large, to make sure avoiding collisions with environment is prioritized over everything else. The *Avoid Vertical Walls* is the main collision avoidance behavior, its very high weight and priority makes sure that this behavior can overpower all other behaviors. The main purpose is keeping the fish away from edges of the pond. The *Avoid Ground* has a higher priority, to always bias fish away from the ground, but low weight, to let the fish reach the ground if there is food at the bottom. Both behaviors have a low maximum distance to make sure only imminent collisions are avoided.

Behavior	Layer	P	WD	WS	D
Avoid Ground	Environment Ground Terrain FakeWalls	5	0.2	0	2
Avoid Vertical Walls	Ground Terrain	6	0.7	0	2

Table 11.3: Fish's *Ray Behaviors* and their main parameters.

Fish Simple Behaviors

There are three *Simple Behaviors* for the fish, listed in Table 11.4. The *Keep Height* behavior has the highest priority, and a small weight. It tries to make sure the fish do not swim too close to the surface. However, having small weight makes sure that the fish can still take a bite of sheep swimming on the surface. The behaviors *Align Up* and *Wandering* have the lowest priority, together with *Homing* and *Cohesion*. The *Align Up* behavior has a low weight, to not bias the fish too much if there is more going on, but to still make the fish align to world's up when there is no target or danger. The lowest weight is on the *Wandering*, to make sure it does not interfere with other behaviors. However, it is important to let the fish naturally wander around when no other behavior is active. The small weight for speed is especially important. It makes sure the fish always have some target speed, but other behaviors can easily overpower it if necessary.

Behavior	P	WD	WS
Wandering	0	0.03	0.01
Align Up	0	0.2	0
Keep Height	6	0.35	0

Table 11.4: Fish's *Simple Behaviors* and their main parameters.

11.3.2 Sharks

The Sharks are tagged with a *Shark* tag. Description of their role in the game scene can be found in Subsection 11.1.5. The set up of sharks is very similar to the fish. For this reason, the main focus will be on their differences.

Sharks Movement

The sharks use the `Move3DSystem`, same as the fish. The system's parameters are listed in Table 11.5. As mentioned when discussing the fish, the parameters were mainly chosen relative to the fish, in order to create interesting predator-prey interactions. With the fish, the main focus was on allowing situations where the fish can escape. In general, however, the shark should be the apex predator, and it should have the upper hand in most situations. To ensure this, the shark has higher maximum speed, and it does not deplete energy as fast. However, it recovers energy slower. This creates an interesting dynamic, where the shark has long periods of calm swimming around the lake and scaring the fish in its way,

followed by shorter periods of high speed hunts which often result in catching a fish. The high parallel acceleration let's it achieve its maximum speed quickly, which is balanced out in fish's favour by lower lateral acceleration, reducing the shark's maneuverability.

Maximum Speed	8
Maximum Parallel Acceleration	7
Maximum Lateral Acceleration	5

Table 11.5: Parameters for movement of sharks.

Sharks Neighbor Behaviors

The first noteworthy information in Table 11.6, is that the *Multi Homing* behavior targets the same homes as the fish. This is intentional to force situations where the fish gather in large groups around the homes, only to get dispersed by the shark. Note that the priority is not the lowest as was the case for fish. That is because for the shark, the home is a valuable place to be, because the fish are there. The *Chase Food* behavior has a very high weight, to allow the shark to be "locked in" on its prey, but relatively low priority to let collision avoidance take the lead if necessary. The avoidance of sounds and the player has a relatively low weight, to make the shark hard to scare. *Separation* tries to ensure no collisions occur between the sharks, using a medium weight and high priority. High weight and maximum distance for *Avoid When Eating* makes sure the shark swims far away from other sharks with its food.

Behavior	**tag**	**P**	**WD**	**WS**	**D**
Multi Homing	FishHome	1	0.3	0	20
Avoid Player	Player	2	0.25	0.25	10
Avoid Sound	Sound	2	0.35	0.35	15
Chase Food	Fish Apple Sheep Bird	2	0.9	0.9	15
Separation	Shark	4	0.5	0	3
Avoid When Eating	Shark	5	0.85	0.85	12

Table 11.6: Sharks' *Neighbor Behaviors* and their main parameters.

Sharks Ray Behaviors

Sharks' *Ray Behaviors*, shown in Table 11.7, all try to solve collision avoidance. The *Environment Avoidance* is the strongest behavior. It tries to make sure to stop the shark from colliding with any surface. The *Avoid Vertical Walls* is aimed at avoiding the edges of the pond. It has relatively low priority, because it is not the main collision avoidance behavior. Having higher maximum distance than *Environment Avoidance* makes sure that in most cases *Avoid Vertical Walls* behavior is used to avoid the edges. The priority for *Avoid Ground* is quite low,

to allow the sharks to hunt fish even near the bottom, at the cost of occasionally colliding with it.

Behavior	Layer	P	WD	WS	D
Avoid Ground	Ground Terrain	1	0.5	0	3
Avoid Vertical Walls	Environment Ground Terrain FakeWalls	3	0.7	0	3
Environment Avoidance	Environment Ground Terrain FakeWalls	6	0.7	0	2

Table 11.7: Sharks' *Ray Behaviors* and their main parameters.

Sharks Simple Behaviors

The *Simple Behaviors* for sharks, shown Table 11.8, are very similar to the fish. The only interesting difference is that the *Keep Height* behavior has a very low priority. The reasoning is the same as for the *Avoid Ground* behavior. It let's the shark catch prey at the surface more easily.

Behavior	P	WD	WS
Wandering	0	0.01	0.02
Align Up	0	0.2	0
Keep Height	1	0.45	0

Table 11.8: Sharks' *Simple Behaviors* and their main parameters.

11.3.3 Birds

Birds are tagged with a *Bird* tag. Description of their role in the game scene can be found in Section 11.1.3. The set up of birds is similar to the fish, but they are simpler. The main focus is to make a large flock of birds look realistic while not being too performance heavy.

Birds Movement

Birds use `Move3DSystem`, but they do not use the energy feature, because there are no predators to hunt them. The associated component's parameters are listed in Table 11.9. The birds have a relatively high *Maximum Speed*, because even real birds need to travel at high speed to stay airborne. They also have high acceleration to go along with it. We found high acceleration to be especially important for large flocks, because it allows the flock as a whole to quickly change direction, which looks visually appealing and realistic. It also allows the birds to avoid collisions with their numerous neighbors better.

Maximum Speed	7.5
Maximum Parallel Acceleration	8
Maximum Lateral Acceleration	8

Table 11.9: Parameters for movement of birds.

Birds Neighbor Behaviors

Birds have only five *Neighbor Behaviors* (Table 11.10), the three classical ones for flocking, one for avoiding sounds and one for homing. Especially the flocking behaviors are the biggest concern from performance perspective, since each bird has many potential neighbors. For this reason, the maximum distances for flocking behaviors are very low compared to other animal types, which improves the performance of the neighborhood search. However, this has some negative effects, especially on the cohesion of the flock, which is why the weight for *Cohesion* behavior is unusually high, which increases the flock's cohesion. Weight for *Alignment* is also high, which results in a more organized looking flock. The *Separation*, as usual, has a high priority, with weight over 0.5 to allow it to overpower other lower priority behaviors.

As usual, the birds also avoid sounds. The *Avoid Sound* behavior has a very high priority and weight to show this interaction off, since it is the only way the player can interact with the birds. Note that unlike before, the priority and weight for *Multi Homing* is very high. This is because in this case, homing is used to make sure the birds stay in their intended area. This was not necessary in previous examples, where colliders kept the fish from leaving the pond. Homing is also important, because it forces the boids to be in one cohesive flock, despite the low maximum distance for *Cohesion*.

Behavior	**tag**	**P**	**WD**	**WS**	**D**
Cohesion	Bird	0	0.8	0.1	5
Alignment	Bird	1	0.8	0.3	5
Multi	BirdHome	5	0.8	0	1000
Separation	Bird	4	0.6	0	2
Avoid Sound	Sound	4	0.7	0.7	15

Table 11.10: Birds' *Neighbor Behaviors* and their main parameters.

Birds Ray Behaviors

There is only one *Ray Behavior* for the birds, listed in Table 11.11. It only makes sure that the birds stay far above the ground. Having the birds stay at high altitude has the benefit that they cannot collide with any environment. The *Avoid Ground* behavior works well with a single ray cast downwards. That is more performance friendly than using for example the *Environment Avoidance* behavior, which would need multiple ray casts per each bird.

Behavior	Layer	P	WD	WS	D
Avoid Ground	Water Ground Terrain	3	0.8	0	30

Table 11.11: Birds' *Ray Behaviors* and their main parameters.

Birds Simple Behaviors

The birds use the same three *Simple Behaviors* (Table 11.12) as the previous two animals. The *Keep Height* behavior does not need particularly large priority or weight like fish that had to respect the water's surface to not collide with it. Keeping a maximum height however "flattens" the shape of the flock which is otherwise more spherical due to the strong homing. Based on our observations, a more flat shape looks more realistic. The set up of the other two behaviors is very similar to other animals.

Behavior	P	WD	WS
Align Up	0	0.2	0
Wandering	0	0.045	0.1
Keep Height	4	0.75	0

Table 11.12: Birds' *Simple Behaviors* and their main parameters.

11.3.4 Sheep

The sheep are tagged with a *Sheep* tag. Their main difference to aforementioned animals is that they move on the ground. However, most of the set up is still similar. Description of their role in the game scene can be found in Subsection 11.1.6.

Sheep Movement

The sheep use the `Move25DSystem`, the parameters are shown in Table 11.13. Their predator is the wolf, but unlike with fish, we did not balance the movement parameters around their interaction too much. The reason for this is twofold – first, the area where the sheep can move is much more open, and so if wolf finally finds itself close to a sheep, we want player to see it catch it. Second, sheep are not particularly known for their speed or agility, and so it made sense to make them an easy prey. Moreover, setting the acceleration low makes their movement smoother. This prevents the whole flock from taking sharp coordinated turns, which we personally did not like for sheep as much as we did in the case of birds or fish.

Maximum Speed	7
Maximum Parallel Acceleration	3
Maximum Lateral Acceleration	3

Table 11.13: Parameters for movement of sheep.

Sheep Neighbor Behaviors

The sheep's *Neighbor Behaviors* are shown in Table 11.14. As usual, there are three behaviors for flocking. Note the high maximum distance for *Cohesion*, so that the flock can regroup even over larger distance. This was not so important for fish and birds since the homing helps them regroup. The weights for both *Cohesion* and *Alignment* are relatively low, which makes the flock more "loose". This is based on personal preferance. However, the *Separation* behavior has a very high priority and weight, to prevent sheep walking through each other. This is a bigger concern than with fish or birds, because the sheep cannot move over or under one another. They are also larger, which makes walking through each other more easily noticeable. The *Get Pushed* behavior provides additional help with preventing collisions between the flockmates.

As always, the *Chase Food* behavior has a high priority and weight to emphasize chasing food, but not high enough to interfere with avoiding danger or collision avoidance. Much like in other examples, the sheep are afraid of sounds and the player, but they flee from their predators with the largest priority and weight. It is interesting to mention that weight and priority for *Avoid Player* is quite low, to make sheep appear a little bit domesticated, allowing the player to get close without completely dispersing the flock. Additionally, note that there is no home area for the sheep, this allows them to roam the entire environment. A home area is also not needed to contain the sheep, because the steep mountains around the environment form a natural barrier.

Behavior	tag	P	WD	WS	D
Cohesion	Sheep	0	0.5	0.5	18
Alignment	Sheep	1	0.4	0.5	5
Get Pushed	Sheep	2	0.3	0.3	2.5
Avoid Player	Player	2	0.25	0.25	15
Avoid Sound	Sound	3	0.7	0.6	15
Avoid When Eating	Sheep	3	0.7	0.3	5
Chase Food	Grass Apple	3	0.75	0.3	20
Fleeing	Wolf Fish Shark	5	0.8	0.7	20
Separation	Sheep	6	0.8	0	2

Table 11.14: Sheep's *Neighbor Behaviors* and their main parameters.

Sheep Ray Behaviors

There are two *Ray Behaviors* listed in Table 11.15. One is *Environment Avoidance* targeting only the *Environment* layer and having the highest priority and weight. The other behavior, *Find Grass*, is the only example of a *Ray Behavior* which is not used for collision avoidance, but for getting more information about a target entity. The sheep use this behavior to stop moving when they detect grass below them. Giving it high priority and a high weight makes sure that the only thing to interrupt eating grass is the wolf or other sheep.

Behavior	Layer	P	WD	WS	D
Find Grass	Grass	4	0.7	0.7	5
Environment Avoidance	Environment	7	0.95	0	3

Table 11.15: Sheep's *Ray Behaviors* and their main parameters.

Sheep Simple Behaviors

The only *Simple Behavior* for the sheep is the *Wandering* behavior, shown in Table 11.16. There is no need for the usual *Align Up* behavior, since `Move25DSystem` makes them align with the sufrace below them. The *Keep Height* behavior is not needed either for obvious reasons.

Behavior	P	WD	WS	D
Wandering	0	0.05	0.1	0

Table 11.16: Sheep's *Simple Behaviors* and their main parameters.

11.3.5 Wolves

The wolves are tagged with a *Wolf* tag. They are almost identical in their set up to the sheep, so the main focus here is on their differences, and how they interact with the sheep. A more detailed description of their role in the game scene can be found in Subsection 11.1.7.

Wolves Movement

The wolves use the `Move25DSystem` system. As already mentioned when discussing the sheep, the wolves movement is superior in every way to the sheep's. This is apparent from Table 11.17.

Maximum Speed	10
Maximum Parallel Acceleration	10
Maximum Lateral Acceleration	6.5

Table 11.17: Parameters for movement of wolves.

Wolves Neighbor Behaviors

The wolves *Neighbor Behaviors* are listed in Table 11.18. They use all three flocking behaviors, to allow formation of packs of wolves. However, the weight for *Cohesion* is low, to prevent them from forming one large pack. Furthermore, the distance for *Separation* is unusually high, to make the packs of wolves more spread out, which increases their chance of finding sheep.

Notice that similarly to sheep, the maximum distances tend to be relatively high, due to the openness of the area. This is especially the case for *Chase Food* behavior. One more interesting thing to note is that unlike sheep, wolves have a *Multi Homing* behavior. Three homes were set up on the edges of the map, to create areas of danger. The weight of the *Multi Homing* is low, to let them chase

the sheep if needed. The rest of the wolves' set up should be straight forward from the previous examples.

Behavior	tag	P	WD	WS	D
Cohesion	Wolf	0	0.16	0.1	20
Alignment	Wolf	1	0.4	0.3	9.5
Multi Homing	WolfHome	2	0.3	0	80
Avoid Player	Player	3	0.7	0.7	20
Chase Food	Sheep	3	0.9	0.6	35
Separation	Wolf	4	0.23	0	7
Avoid When Eating	Wolf	6	0.6	0.6	15

Table 11.18: Wolves' *Neighbor Behaviors* and their main parameters.

Wolves Ray Behaviors

The wolves only have one *Ray Behavior*, displayed in Table 11.19. Same as in other examples, it helps the wolves avoid the environment. As usual, the behavior's priority is the highest. Note the high weight and maximum distance, which makes sure that collisions are avoided as soon as possible. This is useful due to their high maximum speed. Unlike sheep, the wolves also avoid collisions with *FakeWalls*. These are invisible walls placed around the pond to prevent the wolves from running into water.

Behavior	Layer	P	WD	WS	D
Environment Avoidance	Environment FakeWalls	7	0.95	0	5

Table 11.19: Wolves' *Ray Behaviors* and their main parameters.

Wolves Simple Behaviors

Same as sheep, the wolves' only *Simple Behavior* is the *Wandering* behavior. See Table 11.20 for its parameters. As always, *Wandering* ensures that the wolves always have something to do.

Behavior	P	WD	WS
Wandering	0	0.03	0.1

Table 11.20: Wolves' *Simple Behaviors* and their main parameters.

12. Conclusion

In this thesis, first the need to create a framework for simulating flocking and similar behaviors was identified (Chapter 1). After looking at other frameworks and use cases, the requirements for the framework were defined (Section 1.1). The main aspects were that the framework should be modular, extensible, flexible, performant and easy to use. Afterwards, a theoretical model of a flocking system was created, based on the work of Craig Reynolds [7] (Chapter 2). The model splits a flocking system into four distinct parts: *Queries* (split further into *Neighbor Queries* and *Ray Queries*), *Behaviors* (split further into *Neighbor Behaviors*, *Simple Behaviors*, *Ray Behaviors*), *Merger* and *Mover* which feed into each other in this order. Then each of these parts was analyzed in detail from a theoretical perspective (Chapters 3, 4, 5, 6, 7). For each part, we analyzed how other authors implementated each part of the model, and discussed advantages and disadvantages of their approaches, often in relation to some constraints which we set. Based on this analysis, we proposed a basis for our implementations of each part of the model.

In the second part of the thesis, a framework was implemented based on the proposed formalization (Chapter 9). The framework was used to implement a number of modular parts of the model, using the proposed implementations from the theoretical part of the thesis. Furthemore, GUI was implemented to make working with the framework faster and less error prone (Chapter 10). Lastly, the framework was thoroughly tested by creating a scene with multiple different animal types interacting with each other, with the player and with the environment (Chapter 11). Creation of this scene allowed us to test that the framework can handle complex use cases. We showed that it was possible to reuse a lot of the code among different animals, while being flexibile enough for each animal to be unique. It also showed that after some initial learning curve, even a non-programmer could set up this scene, given that all the behaviors had already been implemented. That being said, we are satisfied with the result of the thesis, having achieved most of the established goals. The only goal that was not achieved, was the goal of simulating a flock of 1000 boids in 1 milisecond on reference gaming laptop. The performance testing was described in Section 9.6.

12.1 Contributions

Throughout working on this thesis, we spend a very large amount of time coming up with models for flocking and steering behaviors, tweaking them and testing them. For quite long, more basic model was used, where each behavior returned a desired velocity as a 3D vector. These vectors were afterwards merged into one final desired velocity using a weighted sum. Using this model, we found it extremely difficult to add new behaviors. Any added behavior usually required us to rebalance parameters of other behaviors as well, and it usually introduced subtle edge cases which were hard to avoid and debug. With each behavior added, the complexity of adjusting all the parameters started to become unbearable, as the models started to be more and more unstable. The instability was caused by the fact that there is usually many behaviors with conflicting goals.

The decision to split the result of a behavior into two components: the desired velocity and its weight, was crucial for success of this thesis. The trivial addition of one more dimension for *desire* allowed us to create a more robust merging function (Section 4.4), which from our experience handles large number of behaviors much better. In fact the framework uses two desires, one for speed and one for direction, which provided more granular control. Additionally, priority was added to each result, using Reynolds' concept of prioritized allocation [7]. This provided a layer of robustness to make sure that, for example, separation behavior can always overpower cohesion and alignment behaviors. Adding all this extra information to each behavior's result allowed creation of more complex animals. An example can be fish consisting of total 14 behaviors (Subsection 11.3.1). Having a robust merging function was crucial to ensure that the framework can be used in a modular manner, as behaviors can be added and removed without affecting the rest of the system too much. Moreover, having the split between the desire and the desired speed allowed the *Mover* to account for what speed the animal wants to move at. This proved to be essential for creation of believable predator-prey interactions with dynamic speeds. We see the concept of adding one more dimension for "desire" as the main contribution of this thesis, which can be used to improve simple models based on Reynolds' work. This change is simple to understand and implement, while its performance impact is negligible.

Further ideas which made creating and parametrizing multiple different behaviors easier come from Chapter 6. There, the *Neighbor Behaviors* are split into an *accumulator* going through all neighbors to find an intermediate result, which is then used to determine the behavior's final result. This divided each *Neighbor Behavior* into two more manageable problems. The same ideas were later used for *Ray Behaviors*. Chapter 6 also introduced the idea of using *easing functions* to smoothly shape the amount of influence each neighbor has, and to shape a behavior's final desire. This allowed us to create more interesting non-linear behaviors. Furthermore, keeping desires returned from all *Behaviors* in the range of $[0, 1]$, gave a clear maximum limit of how much influence each behavior can have, which makes reasoning about a system with numerous behaviors much easier, and helps with finding the right balance of the parameters.

12.2 Future Work

While we are satisfied with the result of the thesis, there are many features and ideas which we would like to implement in the future. Some of them are:

- Implementing a *Merger* which limits influence of multiple behaviors in a different way than priority allocation we currently use. An idea which we will experiment with next is inspired by arithmetic coding compression. For example, currently if there are three results r_1, r_2 and r_3 with priorities $p_1 > p_2 > p_3$ each having a desire of 0.5, then r_1 and r_2 would have the same influence and r_3 would not be included. A better approach could be that r_1 contributes by one half, leaving one half for the other two behaviors. Then r_2 would contribute by one half of the remainder, having only one quater of the influence and leaving one quater for the lowest priority result r_3 which would only contribute by one eighth. This would allow all behaviors to

have at least some influence, which exponentially decays as their priority decreases.

- Trying out a different approach to calculating weighted average of directions in the *Merger*. Currently we use a weighted arithmetic mean of the directions. This way, however, a normalized direction is not defined for two vectors with same length poiting in opposite directions, indicating that this method is not ideal. In our future work, we want to experiment with using spherical means instead of arithmetic ones.
- Experimentation with making the animals more complex by giving them a Finite State Machine as a brain. Based on the current state, different behaviors could be activated or their parameters changed.
- Implementing some sort of histerysis for the behaviors. Currently once a predator leaves the field of view of a prey, the prey immediately stops trying to escape. The acceleration based movement partially mitigates this problem, but a better solution could be found.
- Currently, the same weights of behaviors are used for all animals of a same type in the example scene. It could be interesting to experiment with randomizing these weights slightly. The weights could also change based on parameters like hunger, tiredness, age and similar, to create more unique animals. This could open up interesting game mechanics.
- Extending the flocking with a behavior proposed in a paper [25] by Hartman and Beneš, where some boids start escaping the flock and temporarily become leaders. This could create more interesting flocks of birds. Another interesting option to stimulate flocks would be creating a time based vector field representing wind affecting the flock.
- In 2D, collision avoidance with neighbors was difficult to achieve using only our separation behavior. We would like to experiment with more advanced dynamic obstacle collision avoidance algorithms, like RVO or ORCA.
- Experimenting with more advanced workflow, like one we proposed in the Chapter 4 about Merger. There, we suggested each behavior to return desired direction, speed, acceleration together with their rotational couterparts and a desire for each of them. Having control over acceleration could create even more realistic predator response. Having control over the rotation could be useful to let the animals move in one direction, while observing a target in another. Currently our predators often loose their target because they do not try to keep it in their vision.
- Creating a flow field that the boids can follow, as suggested by Reynolds [16]. This could be a relatively cheap way to navigate the animals around more difficult areas. Currently the boids have a hard time navigating around large obstacles like walls to reach a target. More exact navigation like A* would also be a very interesting addition.
- Further experimentation with *Context Steering* [26], and finding a convenient way of working with these behaviors and implementing them.

Bibliography

[1] National Geographic, "Flight of the starlings: Watch this eerie but beautiful phenomenon | short film showcase." https://www.youtube.com/watch?v=V4f_1_r80RY, 2016.

[2] Proper Games, "Flock!." Video Game, 2010. Flocking as a main feature.

[3] Unity Technologies, "Unity." https://unity.com, 2023.

[4] Developer Nation, "A deep dive into studios game developers work for." https://wordpress.developernation.net/a-deep-dive-into-studios-game-developers-work-for/, 2022.

[5] B. Earth, "Ten million starlings swarm (7 tonnes of bird poo) | superswarm | bbc earth." https://www.youtube.com/watch?v=UVko9jyAkQg, 2021. Accessed on 5.1.2024.

[6] T. Ling, "Starling murmurations: Why do they form and how can i see one?." https://www.sciencefocus.com/nature/starling-murmurations, 2021. Accessed on 5.1.2024.

[7] C. Reynolds, "Flocks, herds, and schools: A distributed behavioral model," in *SIGGRAPH '87 Conference Proceedings*, vol. 21, pp. 25–34, 1987.

[8] BeyondVR, "Groupai." https://assetstore.unity.com/packages/tools/behavior-ai/groupai-227995, 2022. Accessed on 12.7.2024.

[9] S. Lague, "Coding adventure: Boids." https://www.youtube.com/watch?v=bqtqltqcQhw, 2019. Accessed on 6.1.2024.

[10] S. Lague, "Boids." https://www.github.com/SebLague/Boids, 2019. Accessed on 6.1.2024.

[11] Unity Technologies, "Boids." https://github.com/Unity-Technologies/EntityComponentSystemSamples/tree/master/EntitiesSamples/Assets/Boids, 2022. Accessed on 23.12.2023.

[12] Unity Technologies, "Unity's data-oriented technology stack (dots)." https://unity.com/dots, 2023.

[13] Unity Technologies, "About burst." https://www.docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html, 2023. Accessed on 6.1.2024.

[14] Google Scholar, "Google Scholar." scholar.google.com/citations?user=PJm3IXAAAAAJ&hl, 2023. Accessed on 02.12.2023.

[15] C. Reynolds, "Boids." https://www.red3d.com/cwr/boids/, 2001. Accessed on 02.12.2023.

[16] C. Reynolds, "Steering behaviors for autonomous characters," in *Game Developers Conference 1999*, (San Jose, CA), pp. 763–782, Miller Freeman Game Group, 1999.

[17] Warthog Games, "Mace griffin: Bounty hunter." Video Game, 2003. Steering behaviors were used to control AI space ships.

[18] S. L. Tomlinson, "The long and short of steering in computer games," *International Journal of Simulation: Systems, Science & Technology*, vol. 5, no. 1, 2004.

[19] Subatomic Studios, "Fieldrunners 2." Video Game, 2012. Steering behaviors were used to control various units in the game.

[20] G. Pentheny, "Efficient crowd simulation for mobile games," in *Game AI Pro*, pp. 317–323, New York: CRC Press, 2013.

[21] Ben Eater, "Boids algorithm demonstration." https://github.com/beneater/boids, 2020. Accessed on 12.12.2023.

[22] Ben Eater, "Boids algorithm demonstration." https://eater.net/boids, 2020. Accessed on 12.12.2023.

[23] B. Craig Reynolds, Nick Porcino, "Opensteer: Steering behaviors for autonomous characters." https://github.com/meshula/OpenSteer, 2003. Accessed on 12.12.2023.

[24] I. Lebar Bajec, N. Zimic, and M. Mraz, "The computational beauty of flocking: boids revisited," *Mathematical and Computer Modelling of Dynamical Systems*, vol. 13, pp. 331–347, August 2007.

[25] C. Hartman and B. Benes, "Autonomous boids," *Computer Animation and Virtual Worlds*, vol. 17, no. 3-4, pp. 199–206, 2006.

[26] A. Fray, "Context steering: Behavior driven steering at the macro scale," in *Game AI Pro 2: Collected Wisdom of Game AI Professionals*, pp. 183–193, CRC Press, 2015.

[27] L. Wagner, C. Olson, and A. Dockhorn, "Generalizations of steering - a modular design," in *2022 IEEE Conference on Games (CoG)*, (Beijing, China), pp. 580–583, 2022.

[28] P. Klepek, "Annoying Dark Souls 2 Glitch Has Gone Unfixed For Nearly A Year." https://kotaku.com/annoying-dark-souls-2-glitch-has-gone-unfixed-for-nearl-1697992451, 2015. Accessed on 12.12.2023.

[29] H. Fathy, O. A. Raouf, and H. Abdelkader, "Flocking behaviour of group movement in real strategy games," in *9th International Conference on Informatics and Systems*, pp. 64–67, 2014.

[30] S. Alaliyat, H. Yndestad, and F. SanfiLippo, "Optimisation of boids swarm model based on genetic algorithm and particle swarm optimisation algorithm (comparative study)," in *28th European Conference on Modelling and Simulation*, pp. 643–650, 2014.

[31] O. Grönqvist, E. Magnusson, I. Naboulsi, L. Norman, M. Sjöblom, and M. Sundqvist, "Baa! a procedural game based on real-time flocking behaviour," Master's thesis, University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden, 2019.

[32] M. Sajwan, D. Gosain, and S. Surani, "Flocking behaviour simulation: Explanation and enhancements in boid algorithm," *International Journal of Computer Science and Information Technology*, vol. 5, no. 4, pp. 5539–5544, 2014.

[33] Suboptimal Engineer, "Boids." https://github.com/SuboptimalEng/boids, 2022. Accessed on 20.12.2023.

[34] Suboptimal Engineer, "Coding a boids flocking simulation." https://www.youtube.com/watch?v=HzR-9tf0JQo, 2022. Accessed on 20.12.2023.

[35] A. Colas, W. van Toll, K. Zibrek, L. Hoyet, A.-H. Olivier, and J. Pettré, "Interaction fields: Intuitive sketch-based steering behaviors for crowd simulation," *Computer Graphics Forum*, pp. 1–14, 2022. In press. DOI: 10.1111/cgf.14491.

[36] Wikipedia, "Drag (physics)." https://en.wikipedia.org/wiki/Drag_(physics), 2022. Accessed on 22.12.2023.

[37] GDQuest, "Godot steering ai framework." https://github.com/GDQuest/godot-steering-ai-framework, 2024. Accessed on 11.1.2024.

[38] K. Shoemaker, "Animating rotation with quaternion curves," in *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 245–254, 1985.

[39] C. Reynolds, "Interaction with groups of autonomous characters," in *Proceedings of the 2000 Game Developers Conference*, vol. 21, 2000.

[40] C. Reynolds, "Big fast crowds on ps3," in *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, (New York, NY, USA), pp. 113–121, ACM, 2006.

[41] M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, M. Viale, and V. Zdravkovic, "Interaction ruling animal collective behavior depends on topological rather than metric distance: Evidence from a field study," *Proceedings of the National Academy of Sciences*, vol. 105, no. 4, pp. 1232–1237, 2008.

[42] D. S. T. C. Train", "Coding challenge 124: Flocking simulation." www.youtube.com/watch?v=mhjuuHl6qHM, 2018. Accessed on 11.12.2024.

[43] D. Shiffman, "Coding challenge 124: Flocking boids." https://editor.p5js.org/codingtrain/sketches/ry4XZ8OkN, 2018. Accessed on 11.12.2024.

[44] C. H. Ching-Shoei Chiang and S. Mittal, "Emergent crowd behavior," *Computer-Aided Design and Applications*, vol. 6, no. 6, pp. 865–875, 2009.

[45] L. by OCLI Vision, "90 animals ranked by eyesight and how they compare to humans." https://www.lasikbyoclivision.com/90-animals-ranked-by-eyesight/, 2023. Table comparing vision of 90 animals.

[46] B. MacPherson, "Did you know?." https://mexicanwolves.org/did-you-know-4/, 2011. Field of vision of wolves.

[47] Unity Technologies, "Nativearray." https://docs.unity3d.com/ScriptReference/Unity.Collections.NativeArray_1.html, 2024. Accessed on 23.12.2023.

[48] R. Nystrom, *Game Programming Patterns*. Robert Nystrom, 2014.

[49] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, p. 209–226, sep 1977.

[50] M. Skrodzki, "The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time," *CoRR*, vol. abs/1903.04936, 2019.

[51] J. Bentley and J. Friedman, "A survey of algorithms and data structures for range searching," *ACM Computing Surveys - CSUR*, 01 1979.

[52] D. N. M. Hoang, D. M. Tran, T.-S. Tran, and H.-A. Pham, "An adaptive weighting mechanism for reynolds rules-based flocking control scheme," *PeerJ Computer Science*, vol. 7, 2021.

[53] Andrey Sitnik, "Easing functions." https://easings.net, 2023. Accessed on 20.12.2023.

[54] Wikipedia, "Smoothstep." https://en.wikipedia.org/wiki/Smoothstep, 2023. Accessed on 20.12.2023.

[55] B. L. Partridge, "The structure and function of fish schools," *Scientific American*, vol. 246, no. 6, pp. 114–123, 1982. Accessed 14 Nov. 2024.

[56] Wikipedia, "Histogram equalization." https://en.wikipedia.org/wiki/Histogram_equalization, 2023. Accessed on 05.03.2024.

[57] OpenAI, "ChatGPT." https://chatgpt.com, 2024. Accessed on 05.03.2024.

[58] F. Heppner and U. Grenander, "A stochastic nonlinear model for coordinated bird flocks," in *Ubiquity of Chaos*, 1990.

[59] Unity Technologies, "Charactercontroller." https://docs.unity3d.com/ScriptReference/CharacterController.html, 2024. CharacterController documentation.

[60] Unity Technologies, "Character controller package." https://docs.unity3d.com/Packages/com.unity.charactercontroller@1.1/manual/index.html, 2024. Accessed on 11.5.2024.

[61] Poke Dev, "Collide and slide - *actually decent* character collision from scratch." https://www.youtube.com/watch?v=YR6Q7dUz2uk, 2024. Accessed on 11.5.2024.

[62] K. Fauerby, "Improved collision detection and response." https://www.peroxide.dk/papers/collision/collision.pdf, 2003. Accessed on 11.5.2024.

[63] J. Linahan, "Improving the numerical robustness of sphere swept collision detection." https://arxiv.org/ftp/arxiv/papers/1211/1211.0059.pdf, 2012.

[64] Unity Technologies, "Job system." https://docs.unity3d.com/Manual/job-system.html, 2024. Accessed on 11.5.2024.

[65] Unity Technologies, "Entities overview." https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/index.html, 2024. Accessed on 11.5.2024.

[66] Unity Technologies, "Entity concepts." https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-entities.html, 2024. Accessed on 11.5.2024.

[67] Unity Technologies, "Component concepts." https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-components.html, 2024. Accessed on 11.5.2024.

[68] Unity Technologies, "Tag components." https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-tag.html, 2024. Accessed on 11.5.2024.

[69] Unity Technologies, "Archetypes concepts." https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-archetypes.html, 2024. Accessed on 11.5.2024.

[70] Unity Technologies, "Query data with entityquery." https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/systems-entityquery.html, 2024. Accessed on 11.5.2024.

[71] Unity Technologies, "System concepts." https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/concepts-systems.html, 2024. Accessed on 11.5.2024.

[72] Unity Technologies, "Custom job types." https://docs.unity3d.com/Packages/com.unity.jobs@0.50/manual/custom_job_types.html, 2024. Accessed on 11.5.2024.

[73] Unity Technologies, "Ijob." https://docs.unity3d.com/ScriptReference/Unity.Jobs.IJob.html, 2024. Accessed on 11.5.2024.

[74] Unity Technologies, "Ijobparallelfor." https://docs.unity3d.com/ScriptReference/Unity.Jobs.IJobParallelFor.html, 2024. Accessed on 11.5.2024.

[75] Jackson Dunstan, "How unity's c# job types are implemented." https://www.jacksondunstan.com/articles/4857, 2024. Accessed on 11.5.2024.

[76] A. Brussee, "Knn." https://github.com/ArthurBrussee/KNN, 2024. Accessed on 11.19.2024.

[77] S. Lague, "Boidhelper.cs." https://github.com/SebLague/Boids/blob/master/Assets/Scripts/BoidHelper.cs, 2024. Accessed on 11.5.2024.

[78] M.-L. A. Chou, "Game math: Swing-twist interpolation (...sterp?)." https://allenchou.net/2018/05/game-math-swing-twist-interpolation-sterp/, 2018. Accessed on 11.23.2024.

[79] Microsoft, "Reflection in .net." https://learn.microsoft.com/en-us/dotnet/fundamentals/reflection/reflection, 2024. Accessed on 11.5.2024.

[80] Microsoft, "Attributes." learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes, 2024. Accessed on 11.5.2024.

[81] Unity Technologies, "Baking." https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/baking.html, 2024. Accessed on 11.5.2024.

[82] Microsoft, "Source generators." https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview, 2024. Accessed on 11.5.2024.

[83] Unity Technologies, "Scriptableobject." https://docs.unity3d.com/Manual/class-ScriptableObject.html, 2024. Accessed on 11.5.2024.

[84] Unity Technologies, "Serializereference." https://docs.unity3d.com/ScriptReference/SerializeReference.html, 2024. Accessed on 11.5.2024.

[85] Microsoft, "Activator.createinstance method." https://learn.microsoft.com/en-us/dotnet/api/system.activator.createinstance, 2024. Accessed on 11.5.2024.

[86] Unity Technologies, "Ui toolkit." https://docs.unity3d.com/Manual/UIElements.html, 2024. Accessed on 11.5.2024.

[87] Unity Technologies, "Uxml element propertyfield." https://docs.unity3d.com/Manual/UIE-uxml-element-PropertyField.html, 2024. Accessed on 11.5.2024.

[88] Unity Technologies, "Uxml element dropdownfield." https://docs.unity3d.com/Manual/UIE-uxml-element-DropdownField.html, 2024. Accessed on 11.5.2024.

[89] Unity Technologies, "Uxml element listview." https://docs.unity3d.com/Manual/UIE-uxml-element-ListView.html, 2024. Accessed on 11.5.2024.

[90] Meta, "Docusaurus." https://docusaurus.io, 2025. Accessed on 01.02.2025.

A. Attachments

The attachments are split into two main folders, as shown below in a high-level overview of the attachments. The folder **`Framework`** contains files relevant to the framework described in Chapters 9, 10. The folder **`Game`** contains files relevant to the sample game scene described in Chapter 11. Lastly, **`PerformanceTest`** contains project that was used to test performance in Section 9.6. The following sections describe contents of these two folders in more detail.

`Attachments`
- **`Framework`**
- **`Game`**
- **`PerformanceTest`**

A.1 Framework

The main file in the **`Framework`** folder is the **`SteeringAI.unitypackage`**. It can be used to import the framework into a Unity project. To see the framework's code and try out the framework's sample scenes, the reader would have to install the package according to the documentation. Therefore, we include a prepared Unity project with the package installed in the **`SamplesProject`** folder. For completeness, the **`Docs`** folder also contains the source files needed to run the online documentation locally.

`Framework`
- **`SteeringAI.unitypackage`** - *The framework as a Unity package.*
- **`SamplesProject`** - *A Unity project with* ***`SteeringAI.unitypackage`*** *installed.*
- **`Docs`** - *Source files for the framework's online documentation.*

A.1.1 SteeringAI.unitypackage

The framework can be imported into Unity using the file **SteeringAI.unitypackage**. The package was tested with Unity version `2022.3.39f1`. To install it, see Getting Started/Installation page of the documentation. To explore the framework without having to install it, see the Unity project in **SamplesProject** folder (Attachment A.1.2). It contains the unpackaged code and sample scenes.

To understand contents of the package, consider a Unity project **Project**. Its folder structure after importing **SteeringAI.unitypackage** is shown below. The framework's code is imported into **com.o-vaic.steering.ai** inside **Packages**. The framework is split into three main folders. The **Core** folder contains the framework's *core*, described in Sections 9.2, 9.3, 9.4. The **Defaults** folder contains the framework's *defaults*, described in Section 9.5. Lastly, the **Editor** folder contains source code for the framework's GUI editor (see Chapter 10).

The package further allows the user to import several sample scenes into **ECS-Steering-Samples** under the project's **Assets**. The sample scenes can be explored to understand the framework better. Each sample is documented under Samples in the documentation.

Project
- **Assets**
  - **ECS-Steering-Samples** - *Samples showing the framework's functionality.*
    - **1. Minimal Set Up**
    - **2. 2D Movement**
    - ...
- **Packages**
  - **com.o-vaic.steering-ai**
    - **RunTime**
      - **Core** - *The framework's core source files.*
      - **Defaults** - *The framework's defaults source files.*
    - **Editor** - *The framework's editor source files.*
    - **package.json** - *The framework's dependecies.*
- ...

A.1.2 SamplesProject

The **SamplesProject** can be used to try the framework, and explore its source code, without the need to install the **SteeringAI.unitypackage**. The folder can be opened in Unity version `2022.3.39f1`. It contains only the samples, **ECS-Steering-Samples** and the framework, **com.o-vaic.steering-ai**. The **SamplesProject** folder is the result of creating a new Unity project and installing the package following the instructions in the Getting Started/Installation section of the documentation.

SamplesProject
- **Assets**
  - **ECS-Steering-Samples**
- **Packages**
  - **com.o-vaic.steering-ai**
- ...

A.1.3 Docs

The folder **`Docs`** contains source files for the online documentation. This is because the online version may change over time. To run version matching the attached **`SteeringAI.unitypackage`** locally, see **`Readme.md`**. To see its current version online, see:

`https://ondra-vaic.github.io/Steering-AI-Documentation`

The documentation was created using Docusaurus [90]. The most important folder of a Docusaurus project is **`docs`**. It contains `.mdx` files, one for each page of the documentation. The folder structure of **`docs`** reflects the menu on the left in Figure A.1. In the figure, the Getting Started/Installation page is selected, corresponding to path **`docs/getting-started/installation.mdx`**.

The documentation is split into four main sections. The beginner guide (**`2_getting-started`**), description of the framework's samples (**`3_samples`**), and documentation of the framework's *core* (**`4_documentation-core`**) as well as its *defaults* (**`5_documentation-defaults`**).

`Docs`
- **`Readme.md`** - *Explains how to run the documentation locally.*
- **`docs`** - *The documentation's .mdx files, each file is one page.*
  - **`1_intro.mdx`**
  - **`2_getting-started`**
    - **`1_installation.mdx`**
    - *...*
  - **`3_samples`**
  - **`4_documentation-core`**
  - **`5_documentation-defaults`**
- *...*

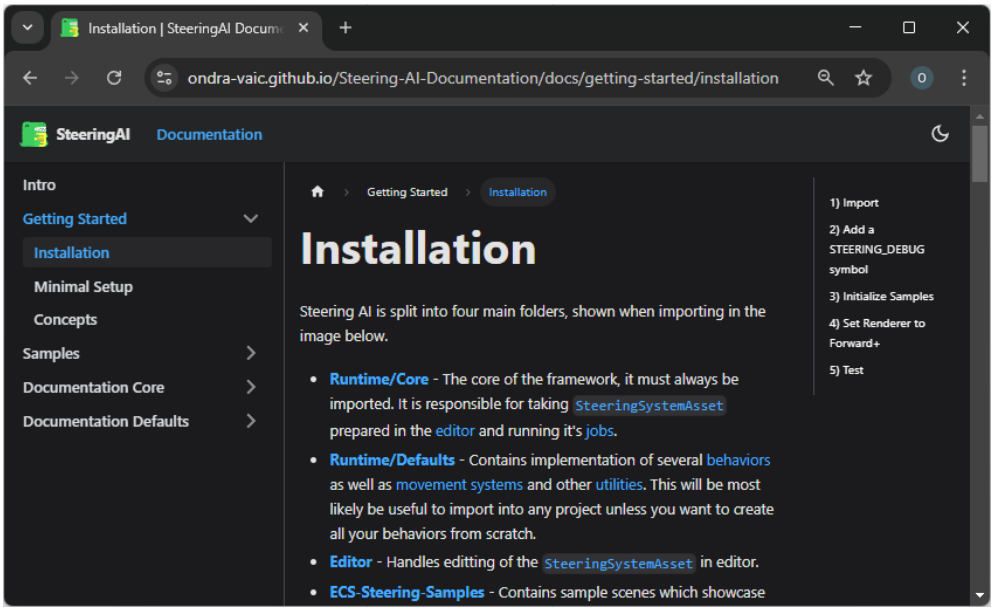

Figure A.1: Framework's documentation with the Installation page selected.

A.2 Game

The **Game** folder contains two subfolders. The **GameProject** folder holds a Unity project for the sample game scene, described in Chapter 11. The **Builds** folder contains builds of the game scene.

Game
- **GameProject** - *Source files for the game scene.*
- **Builds** - *Builds of the game scene.*

A.2.1 GameProject

The **GameProject** is a Unity project, which can be opened with Unity version `2022.3.39f1`. It has the same folder structure as **SamplesProject**. All game–specific assets are in the **Assets/_Project** folder, including some new behaviors described in Section 11.2. As usual, a copy of the framework is located in **Packages/com.o-vaic.steering-ai**. However, this version of the framework contains some minor changes (see Section 11.2).

GameProject - *The game's Unity project.*
- **Assets** - *The game's assets.*
  - **_Project**
    - **Scripts**
    - **Prefabs**
    - ...
- **Packages**
  - **com.o-vaic.steering-ai** - *The framework.*
  - ...
- ...

A.2.2 Builds

The **Builds** folder contains two game builds. The game's controls are described in **Readme.txt**. The first build, **GameProject.exe**, is a build of **GameProject**, which uses simplistic graphic assets that can be shared here thanks to their permissive licensing. The other build, **GameProjectBetterVisual.exe** uses better graphical assets, which, however, are not licensed to be distributed in uncompiled form. This includes several 3D meshes, animations, textures, and a Unity package that resolves animations for Unity ECS. The difference between **GameProject.exe** and **GameProjectBetterVisual.exe** is only visual, Figures A.2, A.3 show how the animals look between the two versions. From left to right, both figures display a wolf, a sheep, a shark, a fish and a bird.

Builds
- **Readme.txt** - *Controls.*
- **GameProject**
  - **GameProject.exe** - *Build of **GameProject**.*
  - ...
- **GameProjectBetterVisual**
  - **GameProjectBetterVisual.exe** - *Build with better visual.*
  - ...

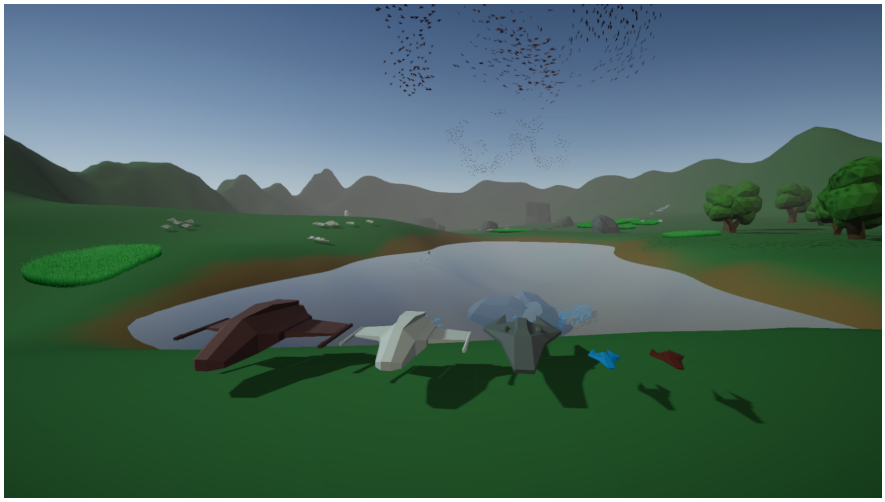

Figure A.2: Scene's visual of **`GameProject.exe`**. From left to right: a wolf, a sheep, a shark, a fish, and a bird.

Figure A.3: Scene's visual of **`GameProjectBetterVisual.exe`**. From left to right: a wolf, a sheep, a shark, a fish, and a bird.

A.3 PerformanceTest

The **`PerformanceTest`** folder contains the Unity project that was used for performance testing in Section 9.6. As always, all regular assets are located in the **`Assets`** folder, and the framework's copy is located in **`Packages/com.o-vaic.steering-ai`**. Here, the framework has some minor changes that enable tracking the performance. A performance test can be configured through `GameObject` named `Tester` inside the `TestingScene.unity` scene.

`PerformanceTest`
- **`Assets`** - *Source files for the performance testing.*
- **`Packages`**
  - **`com.o-vaic.steering-ai`** - *The framework.*
  - *...*