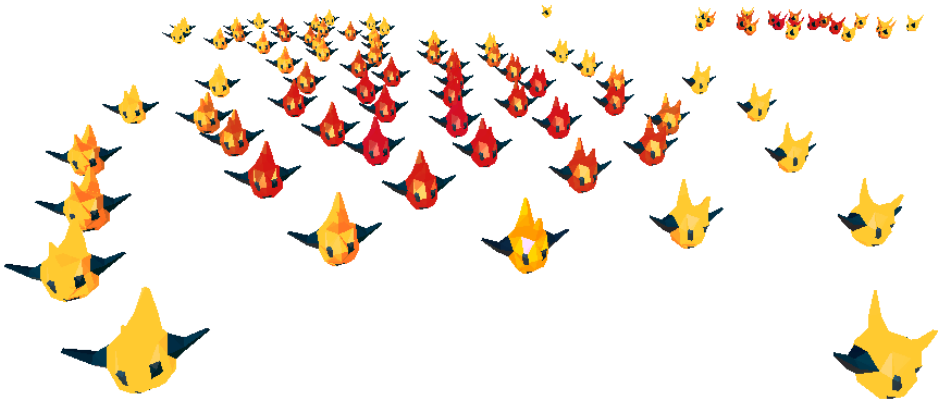

Flocking Behaviour as Demonstrated in a Tower-Defense Game

Bachelor's thesis in Computer Science and Engineering

Alexander Simola Bergsten, Erik Bock, Hampus Ekberg, Noel Hall, Axel Karlsson, Erik Karlsson-Nordling

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020
www.chalmers.se

Bachelor's thesis DATX02-20-22

Flocking Behaviour as Demonstrated in a Tower-Defense Game

Alexander Simola Bergsten, Erik Bock, Hampus Ekberg, Noel Hall,
Axel Karlsson, Erik Karlsson-Nordling

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

Flocking Behaviour as Demonstrated in a Tower-Defense Game
Alexander Simola Bergsten, Erik Bock, Hampus Ekberg, Noel Hall, Axel Karlsson, Erik Karlsson-Nordling

Supervisor: Marco Fratarcangeli, Department of Computer Science and Engineering
Examiner: Michael Heron, Department of Computer Science and Engineering

Bachelor's Thesis DATX02-20-22
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualisation of flocking agents on a 2-dimensional area created using the Unity engine.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Abstract

The research on flocking behaviour pioneered by Craig W. Reynolds has been applied in both movies and video games. However, flocking is seldom used in the video game genre of Tower Defense. Thus, the main purpose of this project is to develop such a game and apply flocking behaviour as a core element in the gameplay. The game is realised with the Unity game engine. The project is comprised of three main activities. First, to create a serviceable video game with dynamic 3D-graphics and audio. Second, to implement and compare different solutions for efficiently simulating flocking behaviour in a correct manner. Third, to design the gameplay with flocking in focus and reward the player for taking advantage of it. Playtesting of the game shows that the majority of people who tried to play the game without regard for the flocking lost. Conversely, almost all the people who played the game with the flocking behaviour in mind managed to win. The end product[1], albeit just a proof of concept, succeeds in showing the potential in the idea of altering Tower Defense gameplay by applying flocking behaviour.

Keywords: flocking; tower defense; boids; k-d tree; spatial hashing; game development

[1]Link to trailer and download of the finished game: https://lmntz.itch.io/lmntz

Sammandrag

Forskningen på flockbeteende av Craig W. Reynolds har använts i både filmer och datorspel. Dock används flockbeteende sällan inom datorspelsgenren Tower Defense. Huvudsyftet med detta projekt är att utveckla ett sådant spel med flockbeteende som en väsentlig del av spelet. Spelet kommer att förverkligas med hjälp av Unitys spelmotor. Detta projekt består av tre huvuduppgifter. Först, att skapa ett funktionsdugligt datorspel med dynamisk 3D-grafik och ljud. För det andra så ska flockbeteende implementeras effektivt och korrekt på ett antal olika sätt som sedan kan jämföras med varandra. Slutligen så ska även spelet designas med flockingsbeteendet i fokus och spelaren ska bli belönad om de utnyttjar det. Testning av spelet av fokusgrupper visade att majoriteten av spelarna som testade spelet utan att ta hänsyn till flockbeteendet förlorade. Å andra sidan, de som spelade med hänsyn till flockandet lyckades nästan alltid vinna. Slutprodukten[2], även om det enbart är ett koncepttest, lyckas det visa potentialen i idén om att förändra Tower Defence-spelstilen genom att tillämpa flockbeteende.

[2]Länk till trailer och nedladdning av spelet: https://lmntz.itch.io/lmntz

Acknowledgements

We would like to thank our supervisor Marco Fratarcangeli for his excellent guidance and support all throughout this project. We also send our thanks to all of the people who participated in the playtests to provide feedback on our game. Lastly, we would like to thank The Division for Language and Communication at Chalmers University of Technology for the feedback during the work with this report.

Contents

List of Figures

List of Tables

1
Introduction

Flocking behaviour has interested the scientific community for a long time. What follows is relevant background information concerning our project of creating a Tower Defence game that utilises flocking behaviour.

1.1 Background

Nature is a complex system consisting of many individual actors such as birds, insects and people [1]. When these actors come together and interact with each other, new kinds of behaviour emerges. Large flocks of birds can seem to be under the influence of a hive mind, yet, its behaviour is simply the result of all individual interactions.

1.1.1 Emergent Behaviours

Emergent behaviour is the phenomenon where many individual entities display complex properties as a group even though each entity might only follow a simple set of rules [2]. One example many are familiar with is the flocking of birds. In this case, there is no force that is guiding the flock to act as it is, only many independent birds acting on their own. The same applies to many other species of animals, like fish in a school or cattle in a herd. There has even been research done that indicates that humans follow flocking behaviour to a certain extent [3].

The study of emergent behaviours is important to be able to understand and replicate natural behaviours. The need for realistic special effects in movies and games is also an area of study that has pushed for the development of flocking behaviour, examples are the passengers in *Titanic* and armies in *The Lord of the Rings: The Two Towers* [4]. In order to achieve realistic behaviour in computer animations, so called behavioural models can be used [5], which are discussed in the section below.

1.1.2 Computer Animation and Flocking Behaviour

In the domain of computer animation, as stated in [5], two approaches exist for animating a group of actors. The first involves the animator in person animating each actor manually. This becomes tedious when the amount of actors in the scene grows. The animator is responsible for making each actor behave in a specified way along with the rest of the group. The second approach is to let the individual actors carry the responsibility of behaving correctly and to animate themselves. This

automates the process, relieving the animator of manually drawing each actor every frame and of having to keep track of each character's state. The animator becomes the director, who provides high-level instructions to the actors who themselves are responsible for the minute details of their behaviour while obeying the animator's commands.

The automated approach of animation is based on the concept of behavioural models [5], which has been applied by Reynolds [1]. Each actor is given a common, often simple, set of rules to follow, that defines the actor's behaviour. This set of rules is the behavioural model. The automated approach facilitates animation of much larger groups. Some movies that have taken advantage of this are *The Lion King* and *The Hunchback of Notre Dame* [5]. However, it also presents an opportunity to observe emergent behaviours like flocking in systems of multiple agents.

As demonstrated by Reynolds [1], flocks can be simulated on a computer by utilising a behavioural model. The flocking entities in the simulation are referred to as boids, from 'bird-oid object', which is the denotation that will be used here and onward. Reynolds formulates three simple rules that each boid should follow: separation, alignment and cohesion. Separation is a boid's tendency to avoid crashing into other boids. Alignment is the tendency to match the velocity of nearby boids. Cohesion is the tendency to stay close to other boids. With only these rules the boids follow each other in flocks that closely resembles what you can see in nature, like seen in figure 1.1.

Flocking as described by Reynolds [1] can be used to solve the problem of individual boids ignoring the movements of their neighbours. This could cause unnatural movement patterns in groups of boids where they bump into each other. Implementations of flocking are thus relevant to those who wish to create realistic simulations of intelligent moving agents, like crowd movement for city planning [3] [6]. It is also relevant to developers of computer/video games, who require such implementations to be efficient as well as realistic. There has been implementations of flocking behaviour in for example real-time strategy games (RTS) in which the characters are moving in flocks [7].

1.1.3 The Tower Defence Genre

A genre of games where flocking behaviour has not been widely implemented or tested yet is known as Tower Defence (TD). It contains games such as *Bloons TD* [8], *Sanctum* [9], *Dungeon Defenders* [10], and many more. All of these games have different takes on the genre, but the common elements between most of them are that there are enemies that try to get from point A to point B and the player is tasked with stopping them by building different types of towers that destroy the enemies or hinder their progress [11]. The enemies are often divided into waves/rounds, and the player wins if they manage to defeat all of the rounds. The lose condition is usually

Figure 1.1: Birds flocking in the sunset.

that a certain amount of enemies reach point B. In some cases the player loses when the first enemy reaches point B, but in other cases the player has Health/Hit Points (HP) that reflect how many enemies can reach point B before the player loses. There are usually also economic aspects, where the player needs to earn currency by defeating enemies to be able to purchase towers. One of the core gameplay aspects of a TD game is being able to budget resources and figuring out which towers should be bought and when. Another related core gameplay aspect is the strategy used when placing towers to effectively stop the enemies.

1.2 Purpose

Considering that flocking behaviour is scarcely implemented in Tower Defence games, this presents an opportunity for investigation on the applicability as well as appropriateness of flocking in such a game. The TD genre allows the player to directly influence the flocking behaviour by deciding the paths of the boids with the placement of towers. Hence, the main purpose of this project is to develop a TD game that utilises efficient flocking behaviour as AI for the enemies. Moreover, the purpose of the flocking is to add unique gameplay mechanics to the TD genre. This purpose has been concretised into the following narrower purposes which are listed in order of importance.

Implications of Flocking on Gameplay: Modify the flocking rules and the gameplay mechanics in such a manner that the player needs to regard the flocking behaviour

in order to conceive an effective strategy in the game.

Develop a Video Game: Create a serviceable video game with dynamic 3D-graphics and audio, with the user experience in mind, by adopting different common techniques used in professional game development.

Efficient Flocking Behaviour: Implement and compare different solutions for efficiently and correctly simulating flocking behaviour, to be able to have many enemies in the game at once.

1.3 Related Works

While there seemingly has been little research done on implementing flocking in a Tower Defence games before, there has been other interesting utilisations of flocking in games and media. A big research topic nowadays is the application of AI together with other emergent behaviours besides flocking with the aim to create realistic-looking simulations.

1.3.1 Flocking

Flocking behaviour, as first defined by Reynolds [1], has been applied in many different scenarios. In the paragraphs below several of these scenarios will be described.

Flocking is one kind of behaviour that can be implemented when working with real-time particles [12]. However, there are also many other applications such as simulating particles being affected by gravity or particles interacting with the environment. What all of these concepts have in common is that rules will have to be created to determine how the particles will act, and in most cases how they can act in a way that is as realistic as possible. Drone [12] explains how a developer can go about implementing different kinds of behaviours for particles. In this project the GPU will not be utilised for the flocking behaviour, but other visuals like explosions might use it which are often important aspects in making a game feel good [13].

Another way of utilising the flocking behaviour was done by Daniel Shiffman with his Swarm system [14]. He wanted to create a non-photorealistic rendering system that could paint images in real-time resembling those of old American Action painters. He was inspired by Reynolds' paper on boids and chose to use a flock to achieve his goal. This shows that flocks can be utilised in different ways than simulating different kinds of living beings, which means that the idea of a game utilising flocking seems more realistic. Allowing the boids in the game to have trails of particles behind them might even create some short lived works of art just like the Swarm.

The Moving Picture Company [15] managed to apply simulated flocking behaviour in order to render space battles in the movie *Guardians of the galaxy*. They based the behaviour of their spaceships on the model presented by Reynolds [1], but also enhanced it with new rules that allowed them to simulate dog fighting and flight

paths. Adding new rules can change how the flocking behaviour can be applied and allow for a lot of interesting uses.

Research has been done previously in implementing flocking behaviour in games. Real-time strategy (RTS) is one genre that has had flocking behaviour implemented to test if the behaviour can help with natural movement [7]. Similarly to the Tower Defence genre, RTS often requires characters to move from point A to point B, the difference being that it happens when the player commands it instead of being a constant. Therefore there is much that can be learned from these papers, even if it is a different genre.

1.3.2 Artificial Intelligence in Video Games

Research shows that a Tower Defence (TD) game can be improved with the help of computational intelligence (CI) and how CI research in turn can be aided by being implemented and tested in a TD game [16]. While this research is mostly about how a game can evolve during gameplay to counter the players playstyle, it does give an interesting look into how TD games can be used to test and improve types of artificial intelligence (AI). In our project the developers themselves are in charge of enhancing the AI of the boids to improve their flocking and other necessary behaviour that is needed for the boids to be able to challenge the players. Just like in the CI research a TD game gives the developers a good estimate of how well the AI of the boids is performing, and where improvements need to be made.

There has also been research done on how emergent behaviours affect a game such as Robocode [17]. Robocode is a game in which you program a virtual robot and let it battle with other robots, like a digital version of Robot Wars. In this research there is not a flock of boids that affect each other and create an emergent behaviour like flocking, but instead an agent is given a list of actions that can be taken which evolves into an emergent strategy for defeating an enemy. Some strategies that evolved from this were: *A robot that shot randomly*, *A robot that tailed other robots and accurately shot them*, *A robot that avoided confrontation until the opponent ran out of energy*. This is a form of emergent artificial intelligence, which will also be explored in this project with the boids individual behaviour creating the emergent behaviour of flocking.

1.4 Societal and Ethical Aspects/Considerations

As aforementioned this project will result in a video game. Thus keeping in mind what kind of effect this game will have on its users is crucial. The type of content that the game contains need to be taken into account to create a game which will strive to not be a negative influence on its players. The following paragraphs will discuss research that has been done regarding positive and negative effects of playing games and how that knowledge was used in this project.

A heavily debated area of research is video games leading to aggressive and vio-

lent behaviour. A study was done [18] on the subject and comes to the conclusion that there is some truth to video games having an effect on behaviour. However, while the study shows that video games should be regarded as a factor to aggressive behaviour it is not a big effect, and that it is directly connected to what kind of game it is. For example, if there are pro-social aspects of a game, like fighting for a good cause, it will remedy some of the parts that might otherwise have a negative impact on a person's behaviour. The same can be said for cooperative games according to the study, which help players work together to achieve a task. In the game developed in this project the player will defeat/destroy/kill enemies, which is a concern in this regard. The enemies however are not humanoids or animals and most of the weapons available to the player have no counterpart in reality, which will hopefully help mitigate any effects this game might have on the player's behaviour.

Addiction to video games is another thing to keep in mind when developing games. Research shows that there are negative effects of excessive playing of video games [19], but that it will not affect a large part of the players. However, it is important to take this into account when developing video games so that it does not become too addictive. In this project the game will not have much replayability, since it is able to be finished and will not have a high score. This means that addiction will not be a big concern for the game.

There are no purely negative aspects to making games however, since games can offer a lot of different valuable skills for humans. Playing certain types of games can for example help improve the speed at which a human processes information [20] and also helps improve spatial cognition [21]. The research done in these areas show that video games can be a valuable tool for learning if utilised properly. In the Tower Defence genre a large part of the gameplay is about learning to budget resources and creating a strategy for how to place towers to achieve victory. Hence, it is possible for the player to learn skills that they might be able to use later in life.

2
Problems

This chapter seeks to break down the project of developing a Tower Defence game that utilises flocking behaviour into a set of smaller problems. The problems related to the implementation of flocking behaviour will be presented in the first section below. Following this, the problems pertaining to the development of the game are addressed.

2.1 Flocking Behaviour

One of the goals in this project is to make the enemies in the game move as a natural flock. This problem requires the boids to follow behaviour such as keeping a distance to other boids, avoid hitting obstacles, move towards some goal and all this while moving in a natural way. In this section, these problems are discussed in more detail.

2.1.1 Three Main Rules

As mentioned earlier, Reynolds describes three rules, see figure 2.1, that together make up the model of a flocking behaviour [1]. However, creating an implementation of these rules has not been a straight-forward task. The definitions given by Reynolds does not describe how the rules should be implemented in detail, but rather how they should behave and their purpose. Also the process of getting the flocking behaviour to feel right requires a great deal of experimentation and testing as it is hard to predict how the boids will behave given a certain implementation of the rules.

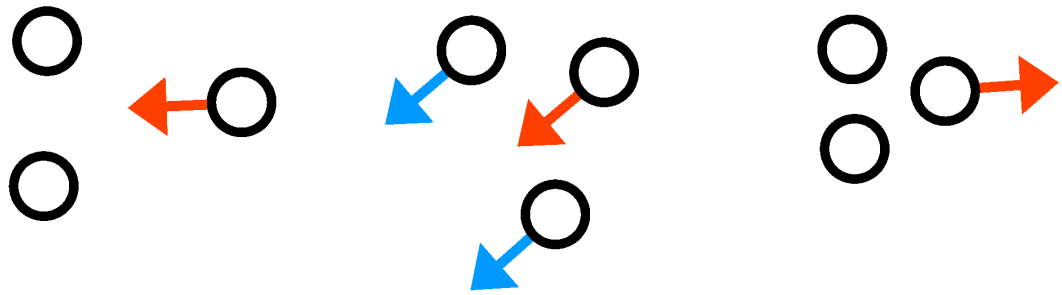

Figure 2.1: The rules, Cohesion (l), Alignment (m), and Separation (r).

The three main rules are enough for the boids to work in a basic simulation, but

for the rules to work in a game context even more rules are needed. First of all in a game there exists non-boid obstacles, which the boids need to be able to avoid. Secondly, in the case of a Tower Defence game, there needs to be a start position and a goal position. The boids then need to be able to navigate from the start to the goal position with no issues.

2.1.2 Obstacle Avoidance Rule

Obstacle Avoidance is an important rule to include in a Tower Defence game since the boids should not clip through any of the towers or other obstacles that are in their way. Therefore the boids need to be able to tell when they are about to collide with an obstacle, and then steer away so that they do not collide. Reynolds writes about this in his paper [1], where he writes about how interesting environmental obstacles are and the effect they have on the boids. He talks about two different ways to achieve a type of obstacle avoidance, the first being the *force field* concept and the second being the *steer-to-avoid* concept.

The *force field* concept is straightforward, and it means that the non-boid objects in the environment will have a field of sorts that repels boids that get to close [1]. The field will grow stronger the closer a boid gets to the obstacle, which in most cases gives a good and realistic behaviour for the boid. However, Renolds make it clear that there are problems with this approach. If a boid is travelling in such a way that its velocity is opposite the direction that the field, then then the field will only slow it down instead of turning it away. There is also the issue of the field affecting boids who have no reason to actually turn away from the object, since it might not be right in front of them and therefore not a problem. Finally Reynolds also bring up the problem of the field not feeling realistic in certain scenarios since it is only really powerful when boids get very close. This can result in the boids bolting away last second, instead of carefully planning their route which would be more realistic.

The *Steer-to-avoid* concept is the approach that Reynolds recommends. This approach entails that the boid has a vision and if it notices anything in front of it within the range of its vision, then it will try to find a path that does not have any obstacles by looking increasingly further to the right and left of itself. This approach might be a bit more costly, since the boids will need to be able to find paths around objects which will take some time to calculate, but overall it gives a much more realistic look with boids that actually seem to plan ahead.

2.1.3 Pathfinding/Goal Rule

Like mentioned previously, in a Tower Defence game the enemies need to be able to go from a starting point A to a goal point B, finding the best and most efficient path to do so. In some Tower Defence games the path is locked and the player is unable to interact with it or block it, but in the case of this project and many other

Tower Defence games the entire playing field is interactive as long as the enemies have some way of going from A to B. If there is no possible path from A to B, it will have to be dealt with. For example by allowing the enemies to bypass obstacles by going over them or destroying random obstacles until a path can be found again.

To be able to go from point A to point B with no preset path, the enemies, referred to as agents in this subsection, will need to be able to calculate what direction to travel in to reach the goal. There are several solutions to this, but many of them are agent based, meaning each agent individually calculates its own path without considering other agents. This approach works very well if you have a small amount of agents that move independently from each other, but with our intended number of agents and the way they are interconnected through the flocking, this approach is not feasible. Instead the fact that every agent has the same goal will be used to implement a goal-based pathfinding algorithm.

A problem that is specific to flocking is that the path finding of the enemies will need to be expressed as a rule. The boids can not stick to a linear path since then the flocking behaviour would be lost, so the path finding will have to fit in with the rest of the rules and be more of a inclination than a direct command.

2.2 Neighbour Problem

In order to calculate the result of the three rules, as well as other possible rules, each boid must know the position and velocity of its neighbouring boids within a certain radius. For a boid to know who its neighbours are it needs to ask every other boid if they are neighbours, as figure 2.2 shows. If n is the total number of boids, then each boid also need to ask n other boids. This leads to the total complexity of $\mathcal{O}(n^2)$. This is a problem as efficient flocking behaviour is is one of the main purposes of this project. However, the complexity can be improved by using different data structures to better manage the boids so not every boid needs to ask every other. The data structures spatial hashing and k-d tree will be compared as two different solution for the neighbouring problem.

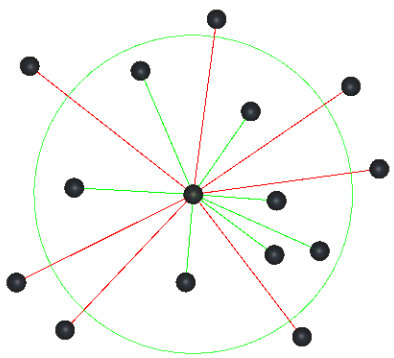

Figure 2.2: The neighbour search for a boid.

2.2.1 Spatial Hashing

A data structure that was considered was spatial hashing. As described in [22], spatial hashing is a method of projecting a 2D or 3D domain to a 1D hash table. Visually it can be represented as dividing the area into a uniform grid of squares, where each square represents a bucket in the hash table, as can be seen in figure 2.3. The hashing function takes a boid's position as input to quickly, $\mathcal{O}(1)$, find the index of its corresponding bucket. Then to find that boid's neighbors only adjacent buckets need to be searched, which also can be found in $\mathcal{O}(1)$ time. The complexity from using spatial hashing can be said to be linear [6], or more specifically $\mathcal{O}(k * n)$ where n is the total number of boids and k the maximum number of boids that the search area can contain.

A problem with this data structure is apparent in the case of very concentrated collections of boids. If a large number of boids were to converge in a small part of the stage, the number of boids per square will still be very large. A way to negate this problem is to use smaller squares. However, as noted in [23], spatial hashing suffers from a memory overhead, since memory is used for all buckets even if no boid is present in it, which is bad for performance. So just decreasing the size of the squares will not always lead to better results. The optimal square size will be the one that has the biggest gain from being smaller relative to the performance lost by the memory cost.

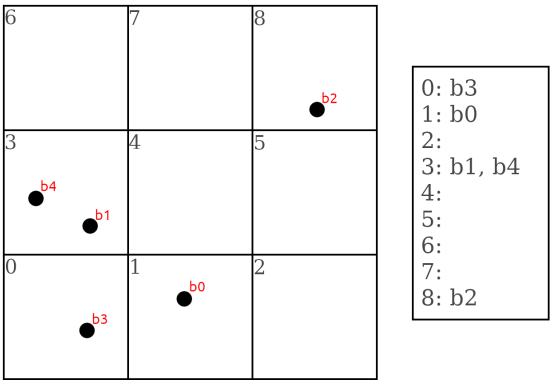

Figure 2.3: Visualisation of the spatial hashing data structure.

2.2.2 K-D Tree

A k-d tree is a type of binary data structure which in the 2-dimensional case divides an area into rectangles [24]. Unlike spatial hashing which divides the area into equally sized squares, the k-d tree can divide the area into smaller parts where the concentration of boids is high. This means a low number of boids per part compared to spatial hashing while still keeping the number of parts, and thus memory cost, low. In the tree, each node represent a rectangular part of the total area. The tree is constructed recursively by splitting every leaf into a node with two children if a certain condition is met for that leaf, i.e. the leaf contains a certain number of boids. The area represented by both children together form the area of the parent

node. Boids are only stored in the leafs. How the result looks can be seen in figure 2.4. In order to find the neighbours the tree makes a recursive search for all leafs that coincides with the search radius and checks all boids in them.

The k-d tree approach however introduce different problems. One important problem is that well balanced building and neighbour search have a time complexity of $\mathcal{O}(nlog(n))$ where n is the number of boids [24]. This is somewhat greater than that of spatial hashing. Another problem is the overhead caused by the recursive tree construction.

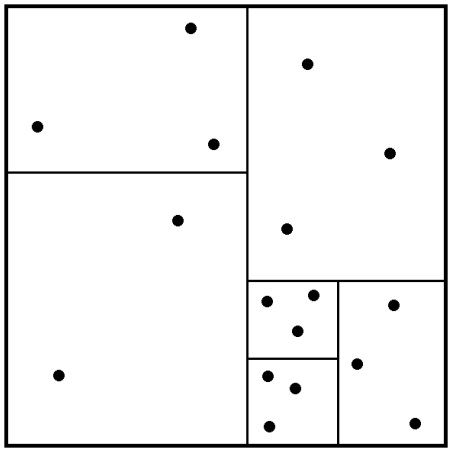

Figure 2.4: Visualisation of the k-d tree.

2.3 Developing a Video Game

Solving the problem with the rules and the neighbour problem will result in an efficient simulation of boids flocking. However, since the goal is not simply to simulate flocking, this is not enough. The problem of using these boids in a meaningful and interactive way through the use of the Tower Defence genre still remains. There are several factors that need to be considered such as game dynamics, difficulty, colour palette and the overall graphical style of the game. It is therefore important to find a design that is well adapted to a computer game setting and helps the player navigate and understand how the game works.

2.3.1 Tower Defence Genre

Since Tower Defence (TD) is a big genre there exists a lot of variations on the core concept [11]. For example how the player interacts with the world. There might just be a fixed top-down camera and the player only uses the mouse to place towers, like in *Bloons TD* [8]. In another game the camera might be able to move around, but the player still only uses the mouse. Games like *Dungeon Defenders* [10] uses a third-person camera and games like *Sanctum* [9] uses a first-person camera, but they both utilise a playable character that the camera is locked on. The player then uses their character to traverse the map, placing towers, and defeating enemies with

various attacks. The games where the default camera is not a "normal" top-down camera often offer an alternative view with this, or at least some type of map that the player can use to get an overview, like in the previously mentioned *Dungeon Defenders* and *Sanctum*. The problem in this project is to determine the aspects of traditional TD that will be utilised and which aspects need to be changed to allow for flocking to become a big part of the game.

2.3.2 Art Style Versus Performance

Another consideration that was made in this project was which art style should be used, since a more realistic art style would be more computation heavy [25]. This is because a more realistic art style requires the use of more polygons, and the more polygons that are used the more time it takes for the Unity engine to render the art. Therefore it would create a big problem since the flocking algorithm itself is already computation heavy. The Unity engine does all the rendering and calculations once per frame, and the frame rate is the amount of frames that the Unity engine has time to process within one second. If a frame takes to long to process, then the frame rate will become lower and this will lead to a worse experience for the player [26]. So by having boids that look like realistic birds the render time would be large as well as the time to calculate their behaviour, which would result in a lower frame rate with the same amount of boids or the same frame rate but with less boids.

2.3.3 User Interface

An important aspect of a game is the user interface (UI). The most common way for users to interact with games these days is through a graphical user interface (GUI), which is a subset of UI. There is also command line interfaces (CLI), but that will not be used in this project. The user experience (UX) on the other hand is not as clearly defined since it is a newer term, but can be though about as how the user feels when they use the UI. UI and UX are therefore closely related, since a bad GUI will almost always lead to a worse experience for the user. UX will be discussed in the next section.

The problem is making sure that the GUI looks good and is easy to understand, so that any user that uses it will enjoy their experience. In order to work efficiently with the UI, three main aspects have been considered: The art style, the colour palette and the placement of elements.

Art Style: In order to make the game appealing to the eye and keep a uniform style throughout the game the GUI needs to be created with an art style that matches the art style used in the rest of the game. An example of this is *Bloons TD* [8] where the cartoon style GUI works well with the other elements of the game.

Colour palette: In order to make the GUI look good and be easy to understand it is important to carefully consider what colours to use in the game. In the article [27] the author discusses the use of colours in game to guide the player and make the

gameplay intuitive, one example mentioned in the article is *Mirrors Edge* [28] where interactive surfaces are coloured red. In order to create a game which is intuitive and interesting what colours to use and how to use them needs to be considered carefully.

Placement of GUI elements: In order to make the game intuitive and good looking it is important to place each element of the GUI in an intuitive manner. Tidwell describes in [29] why this is important and she also discusses how to think when designing a layout. One of the things she mentions is that the "X" or cross button often is in the top right corner of a window, but this is mainly a cultural aspect.

2.3.4 User Experience

An important aspect in games is the user experience (UX) [29]. The user experience is determined by many different factors that need to be addressed, with the goal of creating a game that feels good to play. Some of the most important aspects will be described in the paragraphs below.

One of the largest contributors to how a player feels about the game is the camera. The camera determines the player's perspective of the game environment. Haigh-Hutchinson asserts that an effectively implemented camera should not be distracting to the player and also aid the player in the game [30]. A key challenge here is thus to implement camera behaviour that satisfies these criteria. On top of this, the camera should respond to user input in ways that are obvious to the player.

The camera will support multiple different views. For instance, the camera will be able to provide an overview of the game level to facilitate the placement of new towers and to provide easy access to modify existing towers. Additionally, it will also sometimes follow the avatar of the player to display first person perspective. Since there are several available camera modes, a way to change between these is required, what Haigh-Hutchinson refers to as "transitions" [30]. Finally, there are more specific problems like handling the camera when it is close to a plane which can lead to the plane intersecting the projected image. The result is similar to when the lens of a real camera is positioned just on the surface between water and the air; you can see what is above and below the intersecting surface. That problem is, according to [30], closely related to the problem of collision handling.

Besides the visual feedback that the player is given, another important aspect of UX and the user immersion is the game audio [31]. The audio of a game means as much as the theme of the game if not more [32], which means that the audio needs to be carefully considered when developing the game. The problem is finding the audio that fits with the game and gives the player a good experience.

2.3.5 Implications of Flocking on Gameplay

The use of flocking behaviour as a major gameplay component is one of the main purposes of this project. The idea is that flocking should enhance the normal Tower Defence genre by adding more depth to the gameplay by promoting and encouraging strategies that revolve around the flocking. However the merge of this genre and flocking is not a straightforward process and there exists situations where these two concepts clash and trade-offs needs to be considered.

One such trade-off is having a stronger pathfinding rule versus having more natural flocking. As seen in [33], normal TD strategies often revolve around restricting the enemies movement. For the boids to be able to traverse these kind of environments effectively the pathfinding need to be fairly strong. But this leaves less room for the other rules to influence the boids movement and the flocking will look less natural.

2.3.6 Difficulty

The player should be able to win the game. On the other hand, the player should still feel challenged. Research has shown that games are most fun and exciting when they can challenge players [34]. If the game is too easy, they might get bored. On the other hand, the same research shows that excessive challenge instead leaves the player frustrated which, just like boredom, can make the player not want to play the game again.

The degree to which the player wants to be challenged depends on the type of player [35]. A casual player will play the game to relax or to have fun, whereas a hardcore player is looking to be challenged to the limits of their ability. As demonstrated by [35], there is some evidence suggesting that casual and hardcore players alike will enjoy the game more if the difficulty is reflective of their gaming experience. To target both types of players, there needs to be a system in place that can alter the difficulty.

Two different types of difficulty adjustment systems are discussed in [35], static difficulty options and dynamic difficulty adjustment (DDA). Static difficulty options refer to when the player can choose between for instance *Easy*, *Medium* and *Hard*. DDA makes the difficulty adapt to the performance of the player during the game session.

The difficulty of the game is dependent on many different parameters. To name a few, it is dependent upon the power/effectiveness of each tower, the speed and health of the boids, and the available amount of money. In particular, the benefit that boids get from flocking needs to be adjusted to an appropriate degree. The problem is to find a balance between these parameters and to be able to adjust them depending on the difficulty level.

3
Methods

This chapter contains the theories, methods, and processes used to resolve and handle the problems presented in *Problems*. First, the tools and development methods will be described, then how the flocking behaviour was implemented and tested. Afterwards the different visual and auditory aspects of the game will be discussed, and finally the gameplay aspects.

3.1 Tools

The main tool used in this project is the Unity engine [36]. The main reason the Unity engine was chosen as a development tool was because a lot of documentation exists and it is easy to start using for those who have not used it before [37]. It also is also possible to change parameter values during game play which makes it very efficient for tweaking values. The version used was 2018.4.17f1 personal edition, since at the time of version selection for the project this was the most recent with long time support.

The programming language used for scripting was the object oriented language C# and the programming language D. C# is used since it is the primary language for creating scripts in the Unity engine. There will also be some code written in D, which is a language similar to C++ [38]. D is used since the rules for the boids and spatial partitioning will have to be optimised more than is possible with C#. The D language, like C++, has much less overhead than C# [39], which leads to code running faster and also leaves room for more optimisation. This optimisation turned out to be necessary after the game was unable to handle a boid amount of over the thousands without the game slowing down considerably when the flocking behaviour was implemented in C#.

3.2 Development Method

The development method that was chosen for this project was the agile method Scrum [40]. This method was chosen since several members of the development team has experience with it, and since using a agile method over the standard waterfall method is heavily recommended to achieve greater software quality and to have a stable development [41]. In scrum development is divided into sprints [40], and in this project each sprint was two weeks long. The project also utilised a scrumboard to track the progress of each sprint and the project as a whole.

3.3 Minimum Viable Product (MVP)

An important step in development is determining what the Minimum Viable Product (MVP) should be [42]. A MVP is as the name implies the absolute minimum that has to be developed for the product to be able to represent the final product so that it can be tested by stakeholders. Before starting development a MVP was created for this project, and it looked like the following:

- The player can view each level from a top-down view
- One type of tower with the ability to attack enemies of the player
- One type of enemy with the following properties:
  - Enemies use flock behaviour
  - They are able to die, for example if they get hit by attacks from towers
  - They are able to go from some point A to another point B
  - They take less damage depending on the number of close neighbours.
- The player can win/lose
  - Win: Survive all waves
  - Lose: Too many enemies reach point B.
- One level with the following properties:
  - Completely empty except ground
  - Only one wave of enemies
  - The player has unlimited resources to use in order to place towers on the level.
- It is possible to run a level with at least 100 enemies at a minimum frame rate of 60 frames per second.
- The graphical theme of the game is a minimalist art style.

However the MVP is not the same as the finished product, which will be described in the result. The MVP is the first prototype of the product, that can be seen as a proof of concept and used for evaluation with testers.

3.4 Spatial Partitioning Data Structures

Spatial partitioning data structures are used in order to solve the problem of $\mathcal{O}(n^2)$ complexity when calculating the three rules. These data structures divide an area, in our case a 2-dimensional area into smaller pieces. When searching for the neighbours of each boid you then only need to search in the adjacent parts of the data structure. If the data structure is properly organised these parts will be very easy to find. To be able to evaluate the different spatial partitioning data structures, performance was tested by having a 600x600 units area and seeing how many boids with a neighbour search radius of 10 units could be simulated while maintaining a stable 60 frames per second.

3.4.1 K-D Tree

The k-d tree was developed over three iterations. The first iteration was a "simple" version written in C# without any thought of data orientation. New node instances

were created each time a node was divided and the rule calculation iterated through a list of methods implemented in different classes. The performance was 800 boids.

This was improved by changing to a different version of a k-d tree only storing positions and reusing previously created nodes to reduce instantiation costs. This was not written by us, but was found at Github [43]. On this job stack parallelism was applied and the parallelism was optimized by starting the threads at the end of a frame and collecting them again at the beginning of next. This last was particularly efficient, approximately doubled the frame rate. The way the rules were calculated was also simplified. Instead of calling methods of different objects the rules were written directly in the manager. The number of boids at 60 fps was approximately 2,800.

The last version was written in C++ with the k-d tree mentioned in the last paragraph as outline. The Unity engine game object approach was changed to instead use the Unity engine ECS entities. ECS is a data-oriented methodology of working with Unity that seperates identity, data and behaviour [44]. The performance then became 12,000. Since this version had best performance only this one will be described in more detail.

The project by Github user viliwonka [43] was used as a reference for our implementation, as mentioned previously. The reasoning behind this was that this implementation used algorithms for building and neighbour searching that is easy to implement and the code used a computationally cheap method to store and sort the points, namely storing and sorting indexes of points instead of the points themselves. However, the choice was made to implement the k-d tree in C++ rather than in C#. The reason for this was as mentioned in 3.1 that C++ has less overhead than C#. Thus there was still work to be done translating the pre-existing C#-code into its C++ counterpart. The reason C++ was chosen instead of D which was used in spatial hashing was that the group members who worked on the k-d tree had more knowledge about C++ than D and it has lower overhead than C#.

The algorithms used for building the tree can be described as recursive leaf splitting using median of three as partition algorithm. It works as follows:

- Create new node with no children containing `start` and `end`. If `end - start` $<$ `maxElementsPerLeaf` stop, otherwise continue.

- Use some partitioning method to chose a coordinate `partitionAxis` ,either `x` or `z`, and a value `partitionCoordinate`.

- Permute `indexArray` such that the following criteria becomes true:

$$\forall \mathrm{i}, \texttt{start <= i < partitionPivot} : \tag{3.1}$$

$$\texttt{points[permutation[i]].partitionAxis < partitionCoordinate} \tag{3.2}$$

$$\forall \mathrm{i}, \texttt{partitionPivot <= i < end} : \tag{3.3}$$

$$\texttt{points[permutation[i]].partitionAxis >= partitionCoordinate} \tag{3.4}$$

- Create the left child by jumping to 2 with `start <- start` and `end <- partitionPivot`. When that is done create the right child by jumping to 2 with `start <- partitionPivot` and `end <- end`.

In order to find `partitionAxis` and `partitionCoordinate` in each node a type of median of three algorithm was used. It should be noted that this was not used by GitHub user viliwonka in his original tree. The reason this was chosen was mainly because it required less information to be passed down in the tree but also because previous tests of trees written in C# showed a very slight increase in search performance when using median of three instead of binary splitting which was used in the original.

3.4.2 Spatial Hashing

The spatial hashing implementation also had a few different iterations and was started after the k-d tree implementation when some problems with the way things had been done became apparent. The first version of the spatial hashing algorithm was also written in C# but introduced a shift from representing the boids in an object-oriented manner to a more data-oriented manner where most of the data was laid out in contiguous arrays instead of spread out in different objects. This resulted in being able to simulate 3,000 boids, but it is hard to say what part of the performance gains came from the spatial hashing algorithm and what came from the cache benefits of the data being more tightly packed.

The flocking simulation was later rewritten in D to get the performance benefit of using a compiled native language. The result was that 5,000 boids could be simulated at 60 fps, but profiling the performance showed that almost no time was spent in the actual flocking simulation, and that the simulation was instead bottle necked by various other manipulations of the game objects.

To remedy this bottle neck, a version that used Unity engine's ECS was implemented. Initially this switch resulted in being able to simulate around 17,000 boids in the previously outlined test setting. With some parallelization and moving more code to a native implementation, this version was eventually able to simulate 35,000 boids, and ultimately was the version that was used in the final product.

3.5 Boid Rules

The boid rules are one of the main aspects of this project, since without them the enemies would not be flocking. The three main rules, Separation, Cohesion, and Alignment, gives the correct flocking behaviour while the Collision Avoidance and Path Finding rules gives the behaviour necessary for a tower defence game to work. Each rule calculates a separate acceleration vector for each boid, and each rule vector is then added together to create the actual new velocity of a boid.

When reading this section, it should be taken into account that aside from the criteria for good flocking the choices of rule calculation has in a number of cases been based on subjective opinions such as if the flocking posses "a natural look" and it is specifically chosen to work well with the tower defence setting. If the reader wishes to use flocking behaviour in some context she or he is advised to test different approaches to determine what works best for that setting.

Before the discussion of the rules it is necessary to understand how the boids in our implementation of flocking handles the application of forces. Thus the way boids behave when at large speed will be mentioned first.

3.5.1 Speed Restrictions of Boids

In order to avoid the problem of unnatural behaviour due to unlimited velocity boundaries on the speed of the boids are used. When the speed of a boid exceeds the upper boundary a friction force is applied. To be more precise, if the calculated boid velocity $\boldsymbol{v}_1$ after applying acceleration during a frame has larger size than some maximum speed S_M ,i.e. $||\boldsymbol{v}_1|| > S_M$, the following was set:

$$\boldsymbol{v}_2 = \frac{\boldsymbol{v}_1}{||\boldsymbol{v}_1||} \max(S_M, ||\boldsymbol{v}_1|| - a_f \Delta t) \tag{3.5}$$

where $\boldsymbol{v}_2$ is the velocity after restrictions have been applied, a_f is a constant describing the amplitude of the friction force and Δt is the time since the last frame. Thus, if a_f is large enough, the boids will never accelerate to infinity and the behaviour is more natural.

In addition to the upper speed limit a lower limit is also used S_m. This can be used to prevent unnatural behaviour such as if a boid of some reason where to stop. In our testing however it has shown very little effect, as long as it is significantly lower than S_M.

Of course, there are other ways to solve this problem, such as setting a hard upper speed limit. The main reason 3.5 was chosen as the velocity restriction is that it is easy to understand and implement, and it allows boids to be pushed faster than S_M by external forces. Because of its simplicity it is also simple predict how it will work with other aspects of the game, such as the rules.

3.5.2 Distance Functions Used for the Three Rules

This section will describe how the strength of the three rules are calculated with respect to the distance between each boid and its neighbours. Some previously tested and rejected methods are also mentioned.

The following function is used in order to calculate the strength of all three rules (although alignment uses a slightly different version also taking into account the relative velocity, see 3.5.5 for more details). $\boldsymbol{x}$ is the distance vector from the boid to a neighbour.

$$\boldsymbol{f}(\boldsymbol{x}) = \frac{a\boldsymbol{x}}{||\boldsymbol{x}||/s + ||\boldsymbol{x}||^2} \tag{3.6}$$

Where a is the amplitude of the rule and s is the sensitivity.

These acceleration vectors are calculated for each neighbouring boid within a certain radius, they are then superpositioned to create the final rule acceleration vector. After all three rule accelerations are calculated, they are simply applied to the velocity of the boid. This is then repeated for every boid

The reason function 3.6 has the term $||\boldsymbol{x}||/s$ in the denominator is in order to prevent $\boldsymbol{f}(\boldsymbol{x})$ from tending to infinity as $||\boldsymbol{x}|| \longrightarrow 0$ as well as controlling how sensitive the function is to closer boids.

Note that $||\boldsymbol{x}|| \longrightarrow 0 \implies \boldsymbol{f}(\boldsymbol{x}) \longrightarrow as$ and $||\boldsymbol{x}|| \longrightarrow \infty \implies \boldsymbol{f}(\boldsymbol{x}) \longrightarrow a/||\boldsymbol{x}||$. This shows how a higher value of s increases the strength at close range but has negligible effect at very long range. If a_s and s_s are the parameters for the separation rule and a_c and s_c are the parameters for the cohesion rule, then if they are chosen such that $a_s s_s > a_c s_c$ the separation will be stronger at close range and if $a_s < a_c$ the cohesion will be stronger at long range. This way it is possible to give both rules the same range while still preserving separation when close and cohesion when far away.

In order to find a good shape of the distance function for the separation function $\boldsymbol{f}(\boldsymbol{x})$ a number of different functions were tested. The most notable one is the following:

$$\boldsymbol{f}(\boldsymbol{x}) = \frac{a\boldsymbol{x}}{1/s + ||\boldsymbol{x}||^2} \tag{3.7}$$

The behaviour of this function is very similar to the behaviour of 3.6 at long distances, but will tend to zero as $||\boldsymbol{x}|| \longrightarrow 0$. [1] The reason this function was not used is because boids could "clump together" two and two when getting very near each other. It is also obviously not natural behaviour if the avoidance tendency

[1] Naturally other aspects such as maximum strength will be affected if a and s are the same in the two functions 3.6 and 3.7, but these aspects are not relevant since they can be counteracted by balancing a and s.

disappears when getting to close to a neighbour. Tests were also performed where 3.7 was only used in the cohesion calculation, but unnatural effects that probably was caused by the weak cohesion at close range occurred.

3.5.3 Separation

The separation rule on a boid is calculated by first determining for each of its neighbours (i.e. all boids within a certain search radius) their difference in position $\boldsymbol{x}$ and then uses function 3.6 to calculate the acceleration of the boid. Naturally, $\boldsymbol{x}$ is the distance vector from the neighbour to the boid, otherwise the force would be attractive.

3.5.4 Cohesion

The cohesion rule is implemented by having each boid determine its distance vector $\boldsymbol{x}$ to all boids within a certain radius. This time $\boldsymbol{x}$ is the distance vector from the boid to the neighbour. The distance function 3.6 is then applied to all distance vectors and the result of the calculations is superpositioned to create the total cohesion vector.

3.5.5 Alignment

The alignment rule is calculated in a similar way as separation and cohesion, but since the idea of alignment is to match velocity instead of position a slightly different function is used. if $\boldsymbol{x}$ as before is the distance vector from the boid to its neighbour and $\boldsymbol{v}$ is the relative velocity vector of the neighbour compared to the boid, then the alignment rule has the following function:

$$\boldsymbol{f_a}(\boldsymbol{x}, \boldsymbol{v}) = \frac{\boldsymbol{v}}{||\boldsymbol{v}||} \frac{a_a}{1/s_a + ||\boldsymbol{x}||} \tag{3.8}$$

Not much thought has been put in to this function except that the rule strength is stronger for closer boids. Observe also that the rule strength does not depend on the size of the relative velocity vector $\boldsymbol{v}$, only the direction. The reason for this is that the rule behaviour became more coherent during testing, there is no theoretical reason for this choice.

3.5.6 Rule Calculation Using Weighted Mean

The method for rule calculation which have worked best is to simply superposition the results from the rule calculation on each neighbour, this is however not the only way to calculate the rules. One of the main aspects of superpositioning is that the rules become stronger the more neighbours are present. In one sense it is very natural for a flocking agent to be more careful when surrounded with many neighbours, but for rules like cohesion it is not an obvious choice.

An attempted approach was to calculate the cohesion using a weighted mean of forces from the neighbours. The function looks as follows:

$$\boldsymbol{f_T}(\boldsymbol{x}) = \frac{\sum_{n \in N} w_n \boldsymbol{f}(\boldsymbol{x} - \boldsymbol{x}_n)}{\sum_{n \in N} w_n} \tag{3.9}$$

where N is the set of neighbours, $\boldsymbol{f_S}$ is a function to calculate the force from one neighbour, $\boldsymbol{x}_n$ is the position of neighbour n and w_n is a weight correlated to neighbour n. To calculate w_n an inverse distance function was used. The closer the neighbour, the larger the weight. However, this was not used in the final product, one of the main reason was that each boid could become "content" with just one or two neighbours, not bothering to merge together into larger flocks very often. It is not certain, but it might also have been a contributing factor to a strange behaviour where a ring of boids formed at a distance around the rest of the flock. The mentioned ring only tended to be one layer of boids thick, but the distance between them was very small.

3.5.7 Obstacle Avoidance

Obstacle Avoidance was initially implemented as the recommended *steer-to-avoid* approach by having each boid check for any non boid obstacles in front of it within a certain range. If any obstacle is found in the boids direct path, then the boid will try to find a path which does not have an obstacle, It does this by alternating looking left and right with larger and larger turns until it finds a clear path. The closer the boid gets to the obstacle, the bigger impact the clear path will have on the boids actual velocity. The range of the rule can be adjusted, so that it can be set to a distance that seem like a realistic view distance for the boids.

This approach worked well, but as more and more boids were added to the game, the performance cost of the obstacle avoidance eventually got too big, and the simpler *force field* approach was implemented. This new implementation means that each occupied tile has a force pointing away from it. As boids get closer to the occupied tile, the force pushing them away gets stronger. This approach turned out to be a lot cheaper performance wise and has allowed us to simulate more boids at the same time.

3.5.8 Pathfinding/Goal Rule

The pathfinding rule is implemented with the help of a grid. The entire map is divided into a grid in the logic, and the goal is placed within a certain module of this grid. This same grid is used for towers and for the different obstacles on the map. Although the grid is used to facilitate the pathfinding algorithm, the enemies should not look as if they are bound to the grid when moving.

The development of the pathfinding algorithm was an iterative process which was not finalised until late in the project. From the start it was decided that an agent-based approach to pathfinding was not going to work due to the number of enemies

and the dynamic nature of their movement. Instead, a goal-based algorithm was used that assigns each tile a vector that corresponds to the direction an enemy should move in to reach the goal. The initial version of this was split in two steps, see figure 3.1. First a *wavefront expansion algorithm* was used to calculate the manhattan distance from each reachable tile to the goal tile. Then every tiles vector was set to point to the neighbouring tile with the lowest distance.

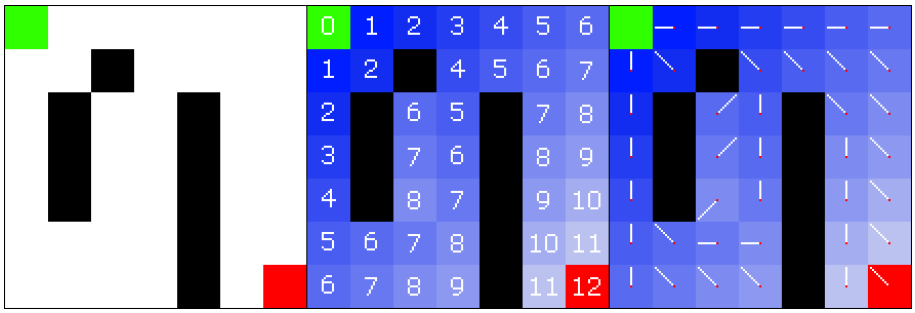

Figure 3.1: An example map(left) going through the different steps (middle and right) of the initial algorithm.

There were a few problems with this way of doing it. First was that since each tile only looked at the immediate neighbours to set its vector, the movement was limited to 8 different directions. Another issue, related to the first one, was that the enemies ended up always going diagonally until they were able to walk horizontally or vertically to reach the goal. These issues made the movement of the enemies look very unnatural. To remedy these problems, changes were made to both steps. In the distance calculating step, the algorithm now kept track of the previous tile of the wavefront expansion algorithm. The vector generation step then used that information to look further back than just the immediate neighbours. When setting a tiles vector, visibility checks were done through raycasting from the tile to the previous tiles in the sequence leading up to it, and if a tile closer to the goal than the immediate neighbours are is found, the vector would point to that tile instead(see figure 3.2).

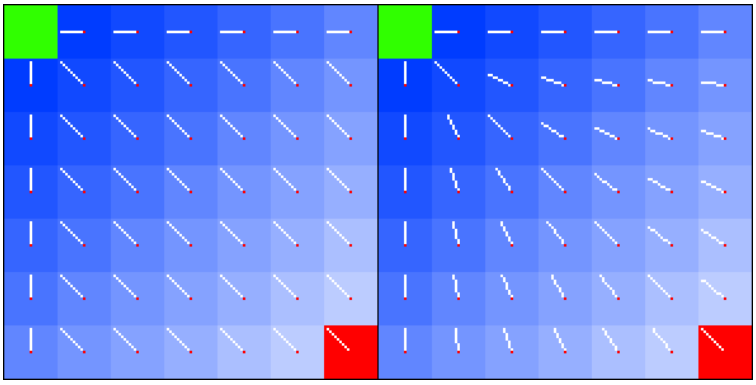

Figure 3.2: Comparison between the old(left) and the new(right) approach on a map without obstacles

When a boid then calculated its pathfinding movement, it would simply check what

tile it is in, and take the vector from that tile as its movement. The problem with this approach was that there could be big differences between the tiles vectors, and it was quite obvious when a boid had passed a tile border to another tile. To fix this problem a bilinear interpolation between the vectors of the 4 nearest tiles is now used as the movement instead. To maintain flocking behaviour, the pathfinding behaviour is implemented as one of the rules affecting boids, so the resulting movement is simply a suggestion instead of a demand which would leave no room for flocking.

3.6 Developing a Video Game

This section is about the solutions that were implemented to create a game that adhered to common game techniques. To be able to achieve this a lot of different aspects needed to be addressed, and the choices and motivations is described in the subsections below.

3.6.1 Art Style and Theme

As mentioned in the problem section the issue with choosing an art style was that more intricate art styles take more time to render by the Unity engine than less intricate art styles, which would result in fewer boids being able to be simulated if the frame rate should stay constant. Therefore a *low poly* art style was chosen, since a low amount of polygons take less time to render but still offer some room for creating a distinct style. After a low poly approach had been chosen the theme was decided to be medieval fantasy, since it fit well with the low poly art style. Illustrated below in figure 3.3 are some of the assets [45] used as obstacles.

Figure 3.3: The assets used for obstacles in the game.

3.6.2 Graphical User Interface (GUI)

The GUI was created with an iterative process. The first draft of the GUI was simple, as can be seen in figure 3.4, and was was put in place for two reasons. Firstly, it was to be able to test the functionality that was implemented in a straightforward way. Secondly, it could be used as a way of testing how the player interacts with the game to see what works and what does not. In the final versions of the GUI there

is a clear resemblance to the original GUI even if it has improved a lot, because the functionality that worked well in the original draft was kept in later iterations.

Figure 3.4: The first draft of the GUI.

Before the actual layout of the GUI was determined, several drafts on paper were made and modified with input from the entire team. An example of these drafts can be seen in figure 3.5. These drafts were then implemented in the Unity engine, and the final layout of the GUI will be presented in the Results section.

One of the layout drafts which were considered can be seen in figure 3.5. In this draft all the information about the towers is kept on the left side of the screen while a mini map is in the top right corner and different options are placed in the bottom right corner. Relevant information such as health and wave count is placed below the mini map. However, the decision was later made to put the options button in the top right corner together with the mini map since this would make the interface more intuitive because of cultural aspects in games [29].

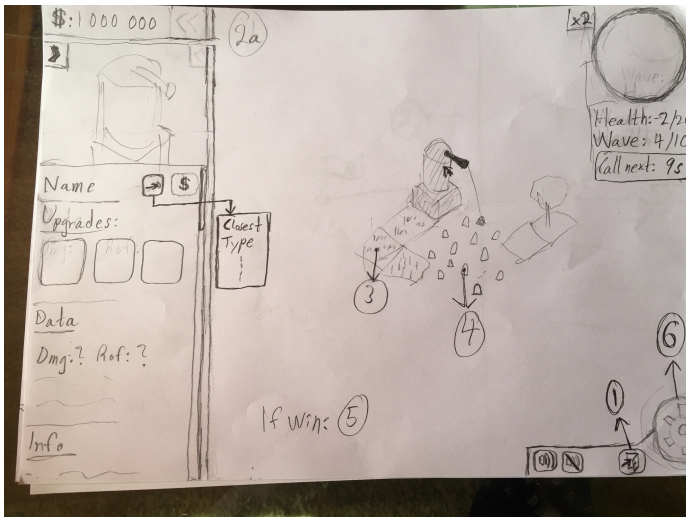

Figure 3.5: Paper draft of the user interface during gameplay.

The art style was also an element of discussion. As mentioned in 2.3.3 it is important to use an art style in the UI that makes the functionality simple to understand and

the game as a whole visually appealing. Since the objects in the game uses a low poly art style it was decided to use a simple design with few small details. Another reason to use a simple design was to decrease the work needed in this project. The decision was made to use a cartoon art style with hard edges. How this later turned out can be seen in 4.9.

Next the colour palette of the GUI was discussed. Based on the discussion of colours in videogames in two articles [27] [46] some important conclusions were made. First of all, there should be a clear difference in colour between the interactive parts of the GUI and the background, furthermore the colours used in the interactive elements should have a high value, in the HSV colour system, compared to the background colours in order to catch the players attention. Secondly, the colours used in important elements of the GUI should be grouped by functionality. In other words, the same colour should be used in all elements which have a common functionality such as closing a window, switch to a certain scene etc. This is in order to make the UI as intuitive as possible, for example if the player wishes to close the current window it can easily find the right button since it knows what colour the button should have. The colour palette that was chosen can be seen in 4.9.

The final thing considered when developing the GUI was visual and auditory response when using interactive elements. This is an important aspect of the interface in order to make it easy for the user to understand all functionality that exists [29]. Visual response was implemented by making buttons and other elements grow in size or change colour ,or both, when hovering over them with the mouse. A short sound when clicking on buttons was added as auditive response.

3.6.3 Camera

The files that represent a level in a game made with the Unity engine are called "scenes" [47]. A Unity scene needs to contain at least one camera or else the player will not see anything [48]. The camera object has a position and rotation which can be altered dynamically by writing a script for it that uses functions from the various Unity libraries. This forms the basis for working with the camera. In the following paragraphs, the solutions to the problems explained in section 2.3.4 will be presented.

To make the camera helpful to the player, it first and foremost provides the player with a top-down perspective on the level. In this view, called pan mode, many towers can be seen at once and manipulated in various ways. In contrast, the player can also switch to inhabit the perspective of a person walking around in the level. In that view, called FPS mode, the player does not have as good of an overview on the towers. The camera can also be positioned at a place where it can capture the entire level, and thus give the player an even better awareness of current events. For pictures that demonstrate these aspects of the camera, see section 4.10.

Transitions between cameras can be implemented in multiple ways. One option

would be to instantly change the position of the camera to its position in the next mode. However, a smoother transition where the camera interpolates from the final position in its current mode to the initial position in the next mode was chosen instead. In this way, the player will not lose its orientation in the world when the camera transitions between modes.

Another problem that interferes with the immersion of the player is that the camera could, if not prevented, travel through objects in the game environment. The solution is to recognise when the camera is in contact with a surface and then stop it from moving closer to that surface. This could be implemented in different ways. In pan mode the camera cannot move if the movement vector originating from the camera's position intersects a surface.

Finally, the player should ideally feel like it is easy and intuitive to control the camera. To verify this, along with the criteria for an effectively implemented camera outlined in section 2.3.4, the game has been tested by external people that provided feedback on the game once it was completed. These tests will be explained in more detail in section 3.6.11.

3.6.4 Level Design

The development process of the maps was done in an iterative way. First many rough drafts were created on paper, the best of those were selected out for further testing, lastly the best map of those were then polished and put in the finished game. This process can be seen in figure 3.6 for one of the maps. When creating the levels thought was also put into the interaction the player will have with the flocking enemies. The goal to create situations where flocking had an impact on gameplay by e.g. having flocks spawn at different locations and merge at different points on the way to the goal.

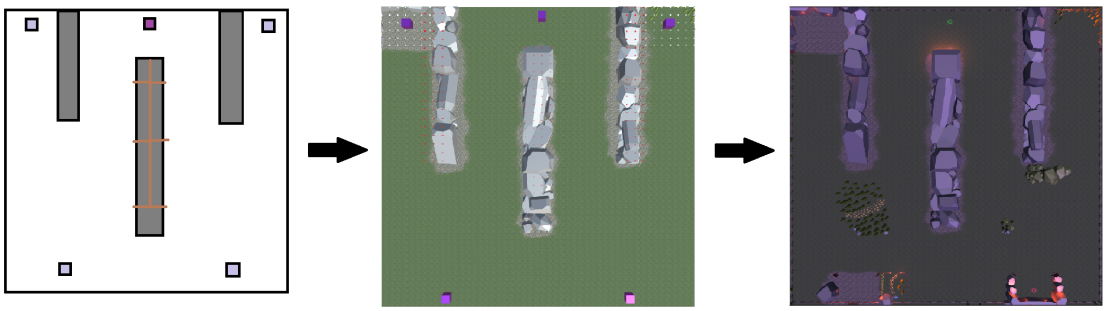

Figure 3.6: The iterative process of creating the maps for the game.

3.6.5 Audio

As mentioned in section 2.3.4 audio plays a large role in the players perception of the game [32]. Therefore a lot of work was put into finding music and sound

effects that would fit with the theme of the game and would help strengthen the players immersion. With the help of the Unity Asset Store and various other similar websites the game was able to be filled with sounds that meshed well together with each other and with the rest of the game.

3.6.6 Particles

As mentioned in section 3.6.1 the game uses a low poly art style and this can also be seen in the particle effects present in the game. The particle effects are used to enhance the players visual experience [13] by indicating actions such as explosions from towers firing, boids dying and even the spawn and goal points [12]. In order to realise these effects the particle system component within the Unity engine was used, and with the help of the image editing software Adobe Photoshop fantasy like particle systems could be created as seen in figure 3.7.

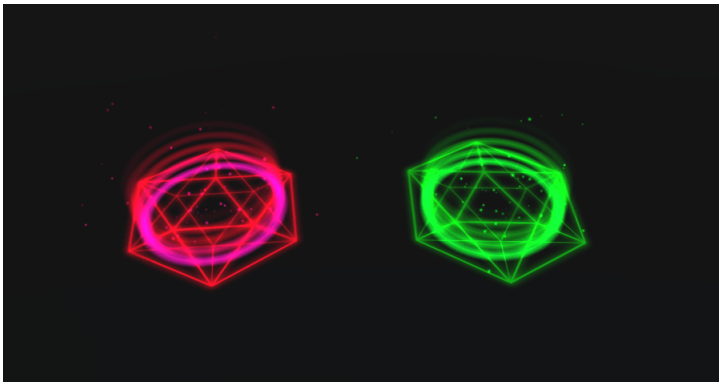

Figure 3.7: Spawn and goal portals.

There are times when a lot of particles are displayed at once perhaps due to a lot of enemies dying simultaneously. An abundance of elements covering the screen can contribute to making it visually difficult to distinguish what is happening in the game [49] and even substantially lower the frames per second at times. In order to address this issue the player is able to set their preferred amount of particles in the options menu.

3.6.7 Towers

In a Tower Defence game the player is able to use towers in order to defend from incoming waves of enemies. The towers often come with a wide array of different abilities which provide the player with a vast number of possibilities [16]. These abilities can range from dealing damage to a single target or in an area to amplifying the abilities of nearby towers or granting the player with extra resources.

In order to create towers that could synergies with each other and contribute to a varied gameplay each tower required their own niche. The first set of towers that were created were basic tower types, a single target tower and a area of effect tower which are commonly seen in other Tower defence games such as *Bloons TD* [8].

Since a lot of the functionality such as the target finding, projectile script and the way to deal damage is mostly the same regardless of the type of tower these towers could later be used as templates to create additional types of towers.

Since the towers are a key aspect of the game, being able to properly control them is crucial in order to be able to effectively withstand the oncoming waves of enemies. Because of this the player can in addition to placing towers also control their targeting style.

3.6.8 Enemies and Enemy Scaling

The enemies of the game are the boids that will utilise flocking. The enemies follow the boid rules, which will result in them flocking from point A to B unless they are stopped. For the player to be able to stop them a boid has Hit/Health Points ,HP, which when reduced to zero will result in the boid getting destroyed.

Each enemy has a scaling value that determines its colour and damage resistance. This value ranges from zero to one and is calculated each frame by counting its number of neighbour using a Sigmoid function, figure 3.8. A value of zero means no extra defence and a light colour, and a value of one means total immunity to damage and a dark colour, but in most cases the value will be somewhere in between.

The purpose of this feature is to add more depth to the game by making it so big flocks will be tougher to defeat, this incentivises the player to try to split up flocks and stop existing from merging with each other. These kinds of strategies seem to involve more creativity and strategy than the alternative where all enemies are funneled into one small area to be defeated, which the neighbour scaling seeks to discourage.

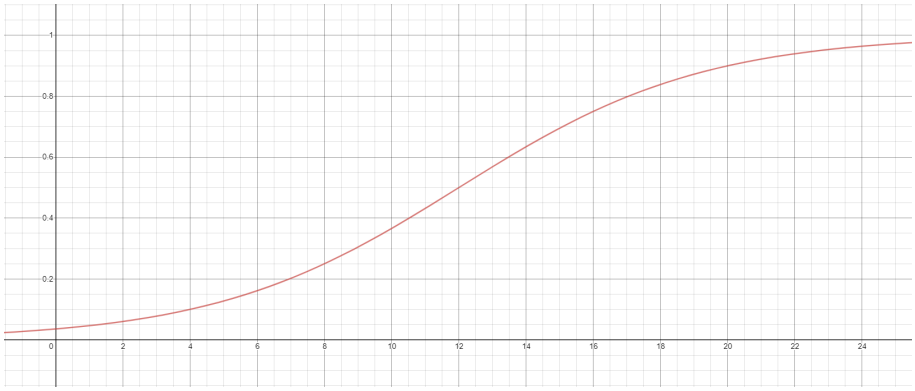

Figure 3.8: The function used to calculate the damage resistance value Y using the number of neighbours X.

3.6.9 Economy

One of the main aspects of a TD game is resource allocation [16], which means that economy plays an important role. The economy in this game revolves around

money with which the player can purchase towers. In the beginning, the player is given a small amount of money. During the game session, they can receive money from three other sources. Those sources are: a passive income for surviving each wave, money received by the farm tower and resources gained by defeating enemies. Income and expenses are balanced in such a way that winning the game will not be too easy nor impossible. More on the topic of difficulty is discussed in the following section.

3.6.10 Difficulty

The game has three different difficulty modes: *Easy*, *Medium* and *Hard*. The difficulty determines how much health the boids will have. On harder difficulties, boids will have more health and thus be more difficult to defeat. This will help make the game more accessible [34]. By providing both casual and experienced players a choice, they can alter the difficulty to fit their desired game experience. In this way the game is targeting a broader audience. Furthermore, this also serves as a remedy if the player gets bored or frustrated from the game being excessively easy or hard, respectively. To check that the levels of difficulty (*Easy*, *Medium* and *Hard*) were appropriately configured, the game was playtested both internally and externally, which will be further explained below.

3.6.11 Playtests

An important part of creating a game is the playtesting [32]. To assess the difficulty of the game, the game has been tested and adjusted iteratively to make it appropriately challenging. Besides assessing the gameplay difficulty, external playtests with focus groups were performed to be able to get an understanding of how well the game has been developed in certain aspects. The main aspect that was of interest was to see what strategies the testers used, and what the results of these strategies were, a win or a loss. The focus group contained people well versed in different games, but not necessarily Tower Defence games. These tests provided feedback on how well the art style and audio works, but were also used to determine how much the flocking behaviour affects how the game is played. Due to COVID-19 the tests had to be done at a distance, and since the game then had to be installed on each tester's computer the chosen test group consisted of people who trusted the developers. This bias has been taken into account, and the information from the tests will be utilised as a new perspective on the game that might have been overlooked by the developers.

4

Results/Discussion

This chapter seeks to present and discuss data gathered during the project. First, the focus is on our specific flocking implementation, then on the results from playtests and lastly on the various parts related to gameplay.

4.1 Spatial Partitioning Performance

Table 4.1: The number of boids that can be simulated with a stable 60 fps for different implementations and field sizes.

Implementation	100x100	200x200	400x400	1000x1000	2000x2000
K-D Tree	2,400	4,000	5,800	10,000	15,000
Spatial hashing	5,400	9,800	15,400	27,000	41,000
Spatial hashing parallel	11,500	22,000	35,000	60,000	90,000

This result indicates that the implementation of spatial hashing works better for our purpose of simulating a large number of boids than the implementation of k-d tree. Reasons for this can be that the memory cost required by spatial hashing is not as high as to negatively affect the performance in a significant extent. It is also possible that the overhead caused by the k-d tree is greater than expected. Also, the implementations developed in this project are not optimal since a limited amount of time was invested in the development. Spatial hashing also has a simpler algorithm than the k-d tree, which can be seen in 2.2.2 and 2.2.1. This means that the final k-d tree implementation may be more prone to inefficient solutions than the implementation of spatial hashing.

These results are from a testing environment where all boids start uniformly spread out across the area and with rendering off. In the actual game the boids will spawn at a limited number of spawn points and as such be more concentrated to the same areas which negatively impacts performance. That along with all the other computations being done to run the game meant that significantly fewer boids could be simulated in the game than in this testing environment. Within the game, approximately 6,000 boids could be simulated while maintaining a stable fps of 30 during gameplay.

4.2 Calculation of Rules

This section describes how the rules are calculated in the final version of the product. First of all the finl values of the parameters for the distance functions of the three rules are presented 4.2. These values were mainly found by testing what worked well with the tower defence setting.

Table 4.2: Parameters used for the flocking rules

Parameter \Rule	Separation	Cohesion	Alignment	Pathfinding
Distance	15	15	15	∞
Amplitude	6	22.5	0.15	0.75
Sensitivity	100	$10^{(-1.5)}$	$10^{(-0.5)}$	-

Aside from these values there is also a master amplitude with a value of 4.
Below is pseudo code that shows how the three rules and the pathfinding rule are calculated for a boid b. It is slightly altered compared to the code actually used in order simplify for the reader, but the principle is the same. deltaTime is the time since the last frame. The vectorField variable that is referenced is the grid of normalized vectors that point towards the shortest path to the goal.

```
cohesion = 0;
alignment = 0;
separation = 0;
pathfinding = bilinearInterpolation(vectorField,b.position)

for each (n in neighbours)
{

    sqdist = squareDistance(b.position, n.position);
    dist = distance(b.position, n.position);

    if(sqdist > 0 && sqdist < (cohesionDistance)^2)
    {
        cohesion +=
        (n.position - b.position) / (dist / cohesionSensitivity + sqdist);
    }

        if(sqdist > 0 && sqdist < (alignmentDistance)^2)
    {

        alignment += (n.velocity - b.velocity).normalized /
        (1 / alignmentSensitivity + dist);
    }

    if(sqdist < (separationDistance)^2)
    {
        separation += (b.position - n.position) /
        (dist / (separationSensitivity) + sqdist);
    }
}

acceleration +=
    (cohesion * cohesionAmplitude +
    alignment * alignmentAmplitude +
    separation * separationAmplitude +
    pathfinding * pathfindingAmplitude)
    * masterAmplitude * deltaTime;
```

4.3 Playtests

In total 16 people tested the game and were then interviewed about their experience. The results of the playtests gave some ideas of what is good and what could be improved in the game. The most important takeaway however was that all of the people who tried to play the game like a usual TD game, like described in [33], did

not manage to win. However almost everyone who played the game in a way that was adjusted to flocking behaviour, such as splitting the boids as much as possible, were able to beat it. The results can be seen in figure 4.3. The balance of the game was still not perfect, but it managed to fulfil the goal of the game being difficult or almost impossible to play like a normal TD game while adjusting the playstyle to regard the flocking behaviour results in a easier experience. Though the game might be a bit too easy with the proper strategies right now, it still works as a proof of concept.

Table 4.3: A table of the strategies used by the players compared to their final result.

	Normal TD Strategy	Flocking Adjusted Strategy
Victory	0	7
Defeat	8	1

4.4 Implications of Flocking on Gameplay

In the final version of the game the flocking is paramount with regards to succeeding in the game, as can be seen by the results of the playtests discussed in the previous section. The enemies gain more damage reduction the more neighbours they have, so grouping them together does not benefit the player. The enemies always try to take the shortest path to the goal, which the player can counter by closing certain paths and forcing the enemies to go in a certain direction. It is important however that the player does not let different flocks merge or try to squeeze a flock into too tight of a corridor, since that will result in the enemies gaining a lot of damage reduction becoming difficult to defeat. If the player instead tries to divide the flocks into smaller flocks, then they will instead lose damage reduction and become easier to defeat.

4.5 Towers

In the final version of the game there are six types of towers available to the player, each with three different tiers: bronze, silver and gold. The player is able to purchase the lowest tier from the shop and then spend resources to upgrade an already placed tower to the next tier. The progression is bronze to silver, and silver to gold. Gold is the highest and most powerful tier. Similar to many other Tower Defence games each upgraded tier grants the tower with additional *damage*, longer *range* or faster *attack speed* [16]. Then there are three different targeting styles for each tower as well. The player can choose from a style that focuses the enemy with the lowest health points, the enemy closest to the tower, or the enemy that is closest to the goal. The following is a list describing all of the available towers in the game:

Ballista Tower: This is the basic tower that has high damage to single targets

and a high fire rate.

Cannon Tower: The cannon tower has a slightly lower fire rate than the ballista but instead of dealing damage to a single target it will also deal damage to enemies within a radius around the target.

Laser Tower: This tower deals continuous damage to the enemy it has targeted.

Buffer Tower: This tower improves nearby towers by increasing their damage.

Farm: The farm is more of a passive tower that, while not directly assisting in destroying enemies, will grant the player with a additional income at the end of each round. This will allow the player to afford a greater defence on later rounds.

Ground Slammer Tower: This tower is similar to the cannon tower since it also deals damage to units in an area. Unlike the cannon this tower deals damage to all units around itself within a certain radius.

Figure 4.1: The towers displayed in a level. From top left to bottom right: Ballista, Cannon, Laser, Buffer, Farm, Ground Slammer.

4.6 Game Balancing

As mentioned in section 3.6.10 a game that offers no challenge will leave the player feeling bored while a game that is to hard or even unbeatable will make the player feel stressed and frustrated [34]. To avoid this and make the game appropriately challenging it has to be balanced. In order to achieve this a number of factors needed to be taken into account. They will be discussed in the following sections.

First, considering the amount of enemies spawned each wave, if too many enemies spawn before the player is able to afford sufficient defences the game will be unbeatable, but if there are too few the game will be too easy. Since flocking behaviour is more apparent in a larger flock, and because the aim is to emphasise flocking, the

decision was made to maximising the number of enemies and then base the other balancing parameters after that decision.

Secondly, if the towers are too strong the amount of enemies spawned would have little to no affect on the difficulty, therefore adjusting their stats was of great importance. Since there are a lot of towers, all with a number of tiers, this was no small task, if a tower is too strong in relation to its price there would be no reason to use any other tower. On the other hand, if a tower is to weak or expensive buying it would yield no benefit to the player. We found that the easiest way to balance the towers was to estimate the different values and then through playtesting fine tune them until all towers had something they excelled at and the game felt fair.

Lastly, the income. If the towers are too cheap or the player is granted to much money the player will amass mountains of gold that either will be tedious to spend or which will have little impact on the outcome of the game. The three sources of income need to synergies with the cost of the towers to make the player spend their earnings thoughtfully. Since the number of enemies increases after each round, the income from defeating enemies was disproportionate to the other income sources. It was either too strong during the last few rounds where thousands of enemies are spawned or too weak in the beginning where the enemy count is barely over 100. Therefor this source of income was heavily reduced and instead the passive income after each wave was prioritised as the main income alternative. This lead to greater control of the players economy as it was not related to the number of enemies.

4.7 Art Style

This section will showcase the art style chosen for the game. The art style is simplistic and low poly, because the time it takes to render each entity need to be low because of the reasons detailed in section 2.3.2.

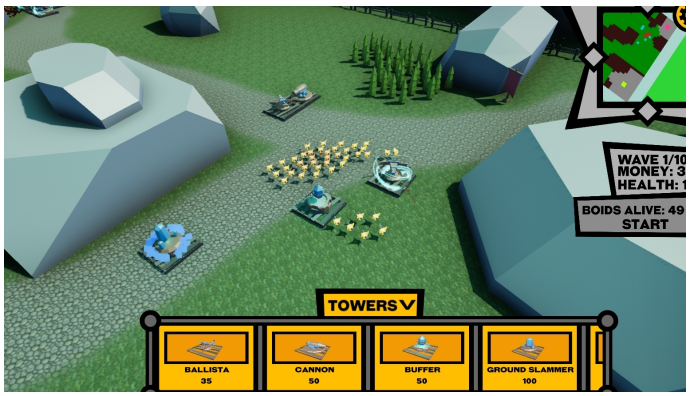

Figure 4.2: A depiction of a normal round in the game.

In figure 4.2 a normal view of the game can be seen. The player has placed their first towers and the first wave of enemies are traversing through them. The scenery is rocks, trees, and banners, while the terrain is grass and stone. There were two

maps created for the game, one which has a daytime setting, figure 4.2, and one that has a nighttime setting, figure 4.3. The different settings helped give a different feel to the maps and make them distinct from each other.

In figure 4.3 a close up of the enemies can be seen. The green portal at the bottom of the image is the goal that the enemies are trying to reach. As can be seen by the darker aesthetics this image is taken on the map with a nighttime setting. Torches have been set up around the map to further improve the aesthetic of the nighttime setting, one of which can be seen slightly in the top of the image.

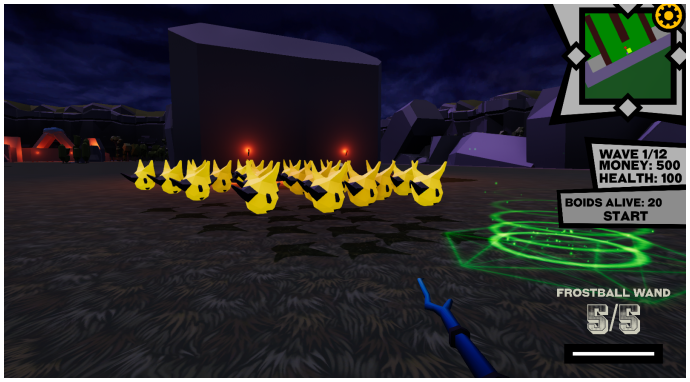

Figure 4.3: Close-up of the enemies, from the first-person perspective, just before they reach the green goal portal.

4.8 Audio

The game has a medieval fantasy theme, so the chosen music had to give the same feeling as the theme for the player to be immersed [31]. The chosen sound effects are therefore similar to the sound effects from RPG games with the same theme as the game, since these type of sound effects are evidently already a good fit for the theme.

A feature that is taken for granted in most games has also been implemented, and that is allowing the player to adjust the sound volume. The player can use the GUI to adjust the master volume, which changes all other volumes. The music and sound effects can be configured separately which allows for a degree of customisation for the player. In the Unity engine this was implemented with the help of an Audio Mixer, which all the games sound can be routed through and altered. This means that the standard volume of different sound effects will be unchanged and differences in decibel between a explosion sound effect and a GUI sound effect will remain the same even if they are lowered by the same amount.

4.9 Graphical User Interface

In order to fit well with the low poly art style of the game, the GUI has a cartoon style with a low amount of details. The colour palette used in the GUI can be

described as grey scale and bright yellow. Figure 4.4 shows the GUI in the level selection screen.

Figure 4.4: The level selection screen.

In order to make the GUI intuitive to use the colour yellow acts as a symbol for interactive elements such as buttons, sliders etc. This makes it easy for the player to discern the relevant elements of the GUI. Every interactive part of the GUI gives visual response when the player hovers the mouse over it. The visual response consists of colour and size changes, and often both. Figure 4.5 shows this.

Figure 4.5: Visual response when hovering over a button.

When deciding what colours to use in the GUI the reasonable working cost of this project was considered. The decision was made to use a simple colour palette with different shades of grey as background, black to mark borders and yellow to mark important/interactive GUI elements. The palette can be seen in figure 4.6.

Figure 4.6: Colour palette used when designing GUI.

4.10 Camera

This section ties back to section 3.6.3 where the different camera modes were presented. The following pictures demonstrate how the camera displays the game environment to the player while a level is active. First, figure 4.2 depicts the default view, pan mode, that provides the player with an overview of a portion of a level. Second, figure 4.3 portrays the first-person perspective of the player, FPS mode. Here, the camera acts as the eyes of the in-game character. The active weapon of the player is also visible here, in this case being the "Frostball Wand". To the right of the weapon is a counter that represents the number of projectiles that the weapon can shoot at this moment. The third view, overview mode, is depicted on figure 4.7 for both of the levels in the game. In this view the entire level is visible to the player.

Figure 4.7: The two maps that are playable in the game

5

Conclusion

The main goal of this project was to develop a Tower Defence (TD) game where flocking behaviour was central to the gameplay. The project was divided into three major objectives. First, it should result in a video game that uses dynamic 3D-graphics and audio. Second, the implementation of the algorithm that realises flocking behaviour needed to be efficient and correctly simulate flocking. Third, the gameplay should be impacted by the flocking behaviour in a meaningful way.

When purely simulating our implementation of the three rules, the desired effect of flocking was achieved. Some parameters needed to be adjusted when applying the flocking to the game to better work with the pathfinding rule and other game mechanics. Here the flocking lost some of its chaotic properties in favour of collectively moving towards a common goal.

The end product is indeed a game with 3D-graphics and audio. Although it is just a demo, or a proof of concept, it succeeds in showing that there is potential in the idea of altering TD gameplay by applying flocking behaviour. The implementation of flocking managed to change which strategy was most effective in order to defeat the enemies, with tower placement being the most prominent change. When flocking behaviour was considered, the player placed their towers in a way that divides the flocks and therefore make each individual enemy easier to defeat.

Spatial hashing turned out to be the more efficient implementation of a spatial partitioning data structure for the circumstances of this project. This does not prove that one of these approaches would be ultimately better overall, because both the data structures could be further improved given more time. However, since the data structures both could be optimised almost indefinitely given an unlimited amount of time, the time constraint was a necessity.

With the spatial hashing method we managed to simulate over 6,000 boids at 30 fps during gameplay. Considering the performance hit from rendering and calculating other gameplay related mechanics, this is sufficiently efficient flocking for our purpose. Therefore, we feel like we have have been able to fulfil the goal that was set at the start of this project, i.e. creating a game running an efficient flocking algorithm.

5.1 Suggestions for Further Research

Future studies could see if the concept of the game could be refined even further, by allowing flocking to affect more aspects of the game. For example, new types of towers that are directly affecting the flocking behaviour itself by changing or disrupting the rules that the boids follow. More types of enemies may also improve the gameplay, it is relevant to investigate how different enemies with different flocking rules will act together. It would also be interesting to see if other genres of games could be altered in a meaningful way with flocking behaviour. A interesting idea would be to make a game with the *Hidden Folks* [50] concept, where the player has to find a certain boid within a flock. This boid might behave in a different way, for example by not following the rules or having different parameters for them.

Another topic for future research could be to investigate if other data structures could substitute uniform grid or k-d tree in the neighbour search to make the flocking algorithm more efficient. Also, one could investigate other ways to improve the performance of the flocking behaviour algorithm to support a greater number of boids. For example one could attempt to utilise the GPU to calculate the rules to see if this could further improve the performance of the flocking behaviour.

Bibliography

[1] C. W. Reynolds. "Flocks, Herds and Schools: A Distributed Behavioral Model". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: Association for Computing Machinery, 1987, 25–34. ISBN: 0897912276. DOI: 10.1145/37401.37406. URL: https://doi.org/10.1145/37401.37406.

[2] S. Viscido, J. Parrish, and D. Grünbaum. "Individual behavior and emergent properties of fish schools: A comparison of observation and theory". In: *Marine Ecology-progress Series - MAR ECOL-PROGR SER* 273 (June 2004), pp. 239–249. DOI: 10.3354/meps273239.

[3] M. Belz, L. W. Pyritz, and M. Boos. "Spontaneous flocking in human groups". In: *Behavioural Processes* 92 (2013), pp. 6 –14. ISSN: 0376-6357. DOI: https://doi.org/10.1016/j.beproc.2012.09.004. URL: http://www.sciencedirect.com/science/article/pii/S0376635712001921.

[4] W. E. Carlson. *Computer Graphics and Computer Animation: A Retrospective Overview*. 2017. URL: https://ohiostate.pressbooks.pub/graphicshistory/chapter/19-2-flocking-systems/ (visited on 02/11/2020).

[5] D. Terzopoulos et al. "Behavioral Modeling and Animation (Panel): Past, Present, and Future". In: *ACM SIGGRAPH 98 Conference Abstracts and Applications*. SIGGRAPH '98. Orlando, Florida, USA: Association for Computing Machinery, 1998, 209–211. ISBN: 1581130465. DOI: 10.1145/280953.281582. URL: https://doi.org/10.1145/280953.281582.

[6] M. Quinn and R. Metoyer. "Parallel implementation of the social forces model". In: *Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics* (Jan. 2003), p. 5.

[7] H. Fathy, O. A. Raouf, and H. Abdelkader. "Flocking behaviour of group movement in real strategy games". In: *2014 9th International Conference on Informatics and Systems*. 2014, PDC–64–PDC–67.

[8] *Bloons Tower Defense*. [Flash Browser Game]. Ninja Kiwi, 2007. URL: https://ninjakiwi.com/Games/Tower-Defense/Bloons-Tower-Defense.html.

[9] *Sanctum*. [PC]. Coffee Stain Studios, 2011. URL: https://www.coffeestainstudios.com/games/sanctum/.

[10] *Dungeon Defenders*. [PC, XBOX, PlayStation]. Trendy Entertainment, 2011. URL: https://store.steampowered.com/app/65800/Dungeon_Defenders/.

[11] L. Mitchell. *Tower Defense: Bringing the genre back*. 2014. URL: https://web.archive.org/web/20140203062250/http://palgn.com.au/11898/tower-defense-bringing-the-genre-back/ (visited on 04/25/2020).

[12] S. Drone. "Real-Time Particle Systems on the GPU in Dynamic Environments". In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: Association for Computing Machinery, 2007, 80–96. ISBN: 9781450318235. DOI: 10.1145/1281500.1281670. URL: https://doi.org/10.1145/1281500.1281670.

[13] B. Zhang and W. Hu. "Game special effect simulation based on particle system of Unity3D". In: *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*. IEEE. 2017, pp. 595–598.

[14] D. Shiffman. "Swarm". In: *ACM SIGGRAPH 2004 Emerging Technologies*. SIGGRAPH '04. Los Angeles, California: Association for Computing Machinery, 2004, p. 26. ISBN: 1581138962. DOI: 10.1145/1186155.1186182. URL: https://doi.org/10.1145/1186155.1186182.

[15] R. Pieké et al. "Creating the Flying Armadas in Guardians of the Galaxy". In: *ACM SIGGRAPH 2014 Talks*. SIGGRAPH '14. Vancouver, Canada: Association for Computing Machinery, 2014. ISBN: 9781450329606. DOI: 10.1145/2614106.2614127. URL: https://doi.org/10.1145/2614106.2614127.

[16] P. Avery et al. "Computational intelligence and tower defence games". In: *2011 IEEE Congress of Evolutionary Computation (CEC)*. 2011, pp. 1084–1091.

[17] J. Hong and S. Cho. "Evolution of emergent behaviors for shooting game characters in Robocode". In: *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*. Vol. 1. 2004, 634–638 Vol.1.

[18] T. Greitemeyer and D. O. Mügge. "Video Games Do Affect Social Outcomes: A Meta-Analytic Review of the Effects of Violent and Prosocial Video Game Play". In: *Personality and Social Psychology Bulletin* 40.5 (2014). PMID: 24458215, pp. 578–589. DOI: 10.1177/0146167213520459. eprint: https://doi.org/10.1177/0146167213520459. URL: https://doi.org/10.1177/0146167213520459.

[19] M. D. Griffiths and A. Meredith. "Videogame Addiction and its Treatment". In: *Journal of Contemporary Psychotherapy* 39.4 (2009), pp. 247–253. ISSN: 1573-3564. DOI: 10.1007/s10879-009-9118-4. URL: https://doi.org/10.1007/s10879-009-9118-4.

[20] M. W.G. Dye, C. S. Green, and D. Bavelier. "Increasing Speed of Processing With Action Video Games". In: *Current Directions in Psychological Science* 18.6 (2009). PMID: 20485453, pp. 321–326. DOI: 10.1111/j.1467-8721.2009.01660.x. eprint: https://doi.org/10.1111/j.1467-8721.2009.01660.x. URL: https://doi.org/10.1111/j.1467-8721.2009.01660.x.

[21] I. Spence and J. Feng. "Video Games and Spatial Cognition". In: *Review of General Psychology* 14.2 (2010), pp. 92–104. DOI: 10.1037/a0019491. eprint: https://doi.org/10.1037/a0019491. URL: https://doi.org/10.1037/a0019491.

[22] E. J. Hastings, J. Mesit, and R. K. Guha. "Optimization of Large-Scale , Real-Time Simulations by Spatial Hashing". In: 2007.

[23] S. Lefebvre and H. Hoppe. "Perfect Spatial Hashing". In: *ACM Trans. Graph.* 25.3 (July 2006), 579–588. ISSN: 0730-0301. DOI: 10.1145/1141911.1141926. URL: https://doi.org/10.1145/1141911.1141926.

[24] J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (Sept. 1975), 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007.

[25] Unity Technologies. *Optimizing graphics performance*. English. Unity Technologies. URL: https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html.

[26] M. Claypool, K. Claypool, and F. Damaa. "The effects of frame rate and resolution on users playing first person shooter games". In: *Multimedia Computing and Networking 2006*. Vol. 6071. International Society for Optics and Photonics. 2006, p. 607101.

[27] M. Hunt. "The effects of colour on a player's mood and performance: identifying the positive and negative effects of different colours when implemented into the environment of a puzzle game." In: *Discovery, Invention & Application* (2016). URL: https://computing.derby.ac.uk/ojs/index.php/da/article/view/162.

[28] *Mirror's Edge*. [PC, XBOX 360, PlayStation 3]. DICE, 2009. URL: https://store.steampowered.com/app/17410/Mirrors_Edge/.

[29] J. Tidwell. *Designing Interfaces*. O'Reilly Media, Inc., 2010. ISBN: 1449379702.

[30] M. Haigh-Hutchinson. *Real-time cameras: a guide for game designers and developers*. Morgan Kaufmann, 2009.

[31] N. Gallacher. "Game audio — an investigation into the effect of audio on player immersion". In: *The Computer Games Journal* 2.2 (2013), pp. 52–79. ISSN: 2052-773X. DOI: 10.1007/BF03392342. URL: https://doi.org/10.1007/BF03392342.

[32] J. Schell. *The Art of Game Design: A book of lenses*. CRC press, 2008, pp. 351, 390.

[33] F. Palero, A. Gonzalez-Pardo, and D. Camacho. "Simple Gamer Interaction Analysis through Tower Defence Games". In: *New Trends in Computational Collective Intelligence*. Ed. by D. Camacho, S. Kim, and B. Trawiński. Cham: Springer International Publishing, 2015, pp. 185–194. DOI: 10.1007/978-3-319-10774-5_18. URL: https://doi.org/10.1007/978-3-319-10774-5_18.

[34] G. Chanel et al. “Boredom, Engagement and Anxiety as Indicators for Adaptation to Difficulty in Games”. In: *Proceedings of the 12th International Conference on Entertainment and Media in the Ubiquitous Era*. MindTrek ’08. Tampere, Finland: Association for Computing Machinery, 2008, 13–17. ISBN: 9781605581972. DOI: 10.1145/1457199.1457203. URL: https://doi.org/10.1145/1457199.1457203.

[35] J. T. Alexander, J. Sear, and A. Oikonomou. “An investigation of the effects of game difficulty on player enjoyment”. In: *Entertainment computing* 4.1 (2013), pp. 53–62.

[36] Unity Technologies. *Unity*. Version 2018.4.17f1. Feb. 11, 2020. URL: https://unity.com/.

[37] M. P. Rogers. “Bringing Unity to the Classroom”. In: *J. Comput. Sci. Coll.* 27.5 (May 2012), 171–177. ISSN: 1937-4771.

[38] A. Alexandrescu. *The D Programming Language: The D Programming Lan_p1*. Addison-Wesley Professional, 2010. Chap. Foreword by Scott Meyers.

[39] H. Chen. “Comparative Study of C, C++, C# and Java Programming Languages”. In: (2010).

[40] K. Schwaber and M. Beedle. *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River, 2002.

[41] Ming Huo et al. “Software quality and agile methods”. In: *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* Sept. 2004, 520–525 vol.1. DOI: 10.1109/CMPSAC.2004.1342889.

[42] V. Lenarduzzi and D. Taibi. “MVP Explained: A Systematic Mapping Study on the Definitions of Minimal Viable Product”. In: *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2016, pp. 112–119.

[43] viliwonka. *KDTree*. https://github.com/viliwonka/KDTree. 2019.

[44] Unity Technologies. *ECS*. 2020. URL: https://docs.unity3d.com/Packages/com.unity.entities@0.10/manual/ecs_core.html.

[45] *Low Poly Tower Defence Pack*. Version 1.0. OpenMyGame, 2017. URL: https://assetstore.unity.com/packages/3d/low-poly-tower-defense-pack-93334#releases.

[46] E. Geslin, L. Jégou, and D. Beaudoin. “How Color Properties Can Be Used to Elicit Emotions in Video Games”. In: *International Journal of Computer Games Technology* 2016 (2016), p. 5182768. ISSN: 1687-7047. DOI: 10.1155/2016/5182768. URL: https://doi.org/10.1155/2016/5182768.

[47] Unity Technologies. *Scenes*. 2020. URL: https://docs.unity3d.com/Manual/CreatingScenes.html.

[48] Unity Technologies. *Cameras*. 2020. URL: https://docs.unity3d.com/Manual/Cameras.html.

[49] David Pinelle, Nelson Wong, and Tadeusz Stach. "Heuristic evaluation for games: usability principles for video game design". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2008, pp. 1453–1462.

[50] *Hidden Folks*. [PC]. Adriaan de Jongh, 2017. URL: `https://store.steampowered.com/app/435400/Hidden_Folks/`.