BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION INFORMATICA

DIPLOMA THESIS

Ocean explorer - Singleplayer Adventure Game

Supervisor
Lect. dr. Lazar Ioan

Author
Rusu Eduard-Constantin

2024

ABSTRACT

In this thesis is presented a simulation of an endless aquatic environment created with Unity. The main goal of this application is to gain some insight in game development, procedural generation of terrain and also simulating the behaviour of animals and how it evolved over time. So the main feature that the proposed system offers is the possibility to watch how the fishes(boids) learn in real time how to avoid obstacles and also wonder along them on the bottom of an endless ocean.

The first chapter presents a short introduction to the idea of the application. Chapter 2 presents how video games have evolved over time. Chapter 3 presents the technologies used to create this project. Chapter 4 presents how the terrain is being generated. Chapter 5 explores the behaviours of natural flocks of birds, and how this behaviour can be simulated using genetic algorithms. The last chapter presents some final conclusions related to the project, optimizations and possible new features and uses.

Contents

Chapter 1

Introduction

1.1 Context

People have always searched for ways to relax, for amusement and for enjoyable activities that fill their leisure time. For the majority of people, one of such activities is playing video games, which allows them to unwind, use their creativity, and, in certain situations, combat even depression.

The first video games debuted in the 1970s. Due to how simple computers were to operate in that time, they got extremely popular, which led to a period that lasted around a decade known as 'The golden age of arcade video games'. Now, with the massive evolution of computers and the apparition of the internet, the quality of the video games grew exponentially. Nowdays, besides the fact that games are almost indistinguishable from reality, players can interact in real time with one another, which leads to amazing and immersive experiences.

1.2 Motivation

Besides being fun to play, video games that just let you wonder around bring with themselves surprising mental and psychological benefits. Experimental psychologist and director of research at the Oxford Internet Research Institute claimed in an interview that gameplay is positively associated with well-being, as play satisfies the psychological needs for competence, autonomy and relatedness.

The motivation of this project is to get an insight in the process of developing such a game, to get to know the challenges and also the possibilities that such a game provides.

Chapter 2

The history of video games

As ancient as mankind itself, play has always been a desire. For kids, it is an essential tool for preparing them for life, and for adults, it is an opportunity to escape reality and have a few minutes to themselves. Similar to almost every aspect of life, gaming has recently made a the transition to the digital sphere. In 2018, a third of people worldwide frequently engaged in video game play, weather on a computer, a game console, or a mobile device. Although the video game's glorious ascent did not start until the 1970s, its origins may be found around 20 years earlier

2.1 The Emergence of Interactive Entertainment (1950s-1960s)

The 1950s and 1960s marked a significant period in the history of interactive entertainment, as computer scientists and engineers began to explore the potential of computers for gaming purposes. While the technology was in its infancy, these early experiments laid the foundation for the future development of video games.

During the 1950s, computers were primarily used for scientific and military applications. However, some visionary individuals recognized the potential for computers to be used for entertainment purposes. One such person was Claude Shannon, a mathematician and computer scientist known as the "father of information theory." In 1950, Shannon published a groundbreaking paper titled "Programming a Computer for Playing Chess," in which he discussed the possibility of programming computers to play games. This paper laid the theoretical groundwork for computer gaming and inspired future researchers to explore the idea further.

One of the earliest known computer games emerged in 1952, created by A.S. Douglas, a student at the University of Cambridge. Douglas developed "OXO," also known as "Noughts and Crosses" or "Tic-Tac-Toe." "OXO" ran on the EDSAC (Electronic Delay Storage Automatic Calculator) computer, a groundbreaking ma-

chine that was one of the world's first stored-program computers. The game allowed players to compete against the computer in a game of tic-tac-toe, with the computer making strategic moves based on programmed rules. While "OXO" had simple graphics and limited interactivity, it showcased the potential for computers to provide interactive gaming experiences.

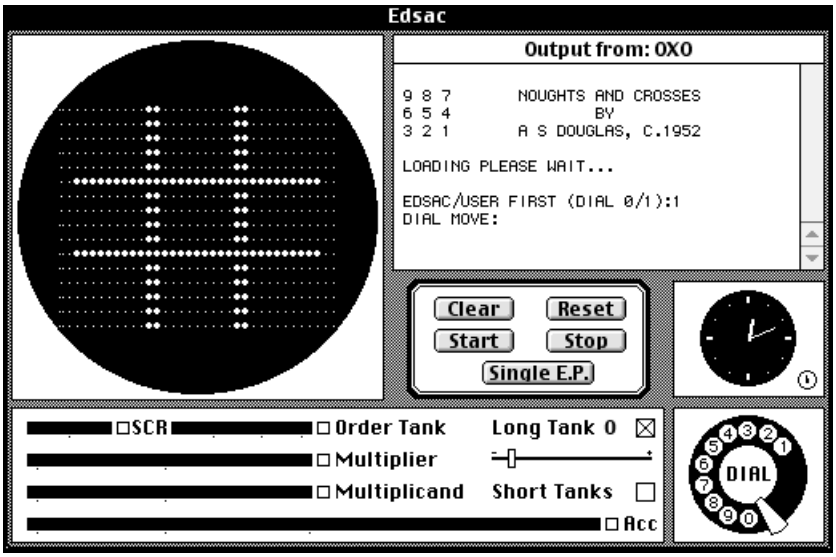


Figure 2.1: OXO played in an EDSAC simulator for the Classic Mac OS [oxo]

Another notable development in the 1950s was "Tennis for Two," created by physicist William Higinbotham in 1958. Higinbotham developed the game as an interactive exhibit for visitors at the Brookhaven National Laboratory in New York. Played on an oscilloscope, "Tennis for Two" simulated a game of tennis, with players using controllers to hit a virtual ball over a net. The game utilized basic physics principles to simulate the motion of the ball and provided a simple yet engaging multiplayer experience. "Tennis for Two" is considered one of the earliest examples of electronic interactive entertainment and laid the groundwork for future advancements in gaming technology.

In the 1960s, computer technology continued to advance, enabling further exploration of interactive entertainment. One of the most influential games of this period was "Spacewar!," developed in 1961 by Steve Russell, Martin Graetz, and Wayne Wiitanen at the Massachusetts Institute of Technology (MIT). "Spacewar!" was designed for the PDP-1, one of the earliest interactive computers. It allowed two players to control spaceships and engage in a simulated space combat scenario. Players had to navigate their ships, utilizing gravitational forces and strategically firing torpedoes to defeat their opponent. "Spacewar!" introduced concepts such as real-time gameplay, gravity simulation, and multiplayer interaction, pushing the boundaries of what was possible in computer-based entertainment.

While 'Spacewar!' gained popularity within the academic and technological

Figure 2.2: Tennis for Two [TfT]

communities, it was not widely accessible to the general public due to the limited availability of computers at the time. However, its impact was significant, inspiring future generations of game developers and serving as a catalyst for the development of subsequent video games. [HoG]

2.2 The Dawn of Commercial Gaming (1970s)

The 1970s marked the dawn of commercial gaming, with significant advancements in technology and the release of pioneering games that laid the foundation for the booming video game industry we know today.

One of the key developments in the 1970s was the rise of arcade gaming. Arcade machines, typically found in public spaces such as arcades, bars, and malls, became popular gathering spots for gaming enthusiasts. These machines offered immersive gaming experiences and quickly gained a dedicated following.

In 1971, "Computer Space" became the first commercially available arcade video game. Created by Nolan Bushnell and Ted Dabney, who would later establish Atari, "Computer Space" was inspired by the earlier game "Spacewar!" However, due to its complexity and unfamiliarity to the general public, "Computer Space" did not achieve widespread success. Nevertheless, it set the stage for future innovations in arcade gaming.

The breakthrough moment for commercial gaming came in 1972 with the release of "Pong" by Atari. Designed by Allan Alcorn, "Pong" was a simplified tennis simulation game that became a massive hit. It featured two paddles and a ball bouncing

Figure 2.3: Spacewar! on a PDP-1 [TfT]

between them on a screen, mimicking the gameplay of table tennis. The intuitive controls and addictive gameplay of "Pong" captured the attention of players across various demographics. The game quickly spread to arcades, bars, and even homes through dedicated consoles.

The success of "Pong" was a game-changer for the industry. It proved that video games could be both entertaining and profitable, sparking a wave of innovation in game design and technology. The simplicity and accessibility of "Pong" laid the foundation for the rapid growth and diversification of the gaming industry.

Following the success of "Pong," Atari released home console versions of the game, allowing players to enjoy the experience in their own homes. This marked the beginning of the home gaming revolution and laid the foundation for the console industry. Other companies, such as Magnavox with the Odyssey and Coleco with Telstar, also entered the home console market during this period.

As the 1970s progressed, more companies began developing and releasing arcade games, creating a competitive market. Titles such as "Space Invaders" (1978) by Taito and "Asteroids" (1979) by Atari became major hits and solidified the popularity of arcade gaming. [HoG]

2.3 The Golden Age of Arcade Games (1980s)

The 1980s is often referred to as the golden age of arcade games, characterized by an explosion of creativity, innovation, and cultural impact. This decade saw the release

of numerous iconic games that left an indelible mark on gaming history.

2.3.1 Pac-Man: A Cultural Phenomenon

In 1980, Namco's "Pac-Man" was released and became a cultural phenomenon. Designed by Toru Iwatani, "Pac-Man" introduced players to a unique maze-like gameplay experience. Controlling the titular character, players had to navigate through a maze, consuming pellets while avoiding colorful ghosts. Power pellets allowed Pac-Man to temporarily turn the tables and chase down the ghosts. With its simple yet addictive gameplay, vibrant visuals, and catchy music, "Pac-Man" became an instant hit. It captured the imagination of players worldwide and transcended gaming to become a pop culture phenomenon. "Pac-Man" introduced character-driven gameplay and set the stage for the development of story-driven games in the future.

2.3.2 Donkey Kong and the Rise of the Platformer

In 1981, Nintendo released "Donkey Kong," designed by Shigeru Miyamoto. The game introduced the character of Mario (originally named Jumpman), who had to navigate a series of platforms while avoiding obstacles and enemies to rescue his girlfriend from the clutches of the titular ape. "Donkey Kong" marked the debut of Mario, who would go on to become one of the most recognizable and iconic characters in video game history. The success of "Donkey Kong" propelled Nintendo into the gaming spotlight and solidified the platformer genre as a mainstay of video game design.

2.3.3 Space Invaders and the Invasion of Shoot 'Em Ups

In 1978, Taito's "Space Invaders" hit the arcades, revolutionizing the shooter genre. Players controlled a ship at the bottom of the screen, tasked with defending Earth from an alien invasion. The game featured relentless waves of descending aliens, progressively increasing in speed and intensity. "Space Invaders" not only popularized the shoot 'em up genre but also introduced high-score competition, with players vying for the top spot on arcade leaderboards. The game's addictive gameplay, memorable sound effects, and captivating theme made it a global phenomenon, earning billions of dollars in revenue and paving the way for future shooter games.

2.3.4 Other Notable Games and Innovations

The golden age of arcade games saw the release of several other notable titles. "Galaga" (1981) by Namco, a sequel to "Galaxian," introduced enhanced graph-

ics and gameplay mechanics, further popularizing the space shooter genre. "Centipede" (1980) by Atari, designed by Ed Logg and Dona Bailey, combined fast-paced shooting action with strategic gameplay, as players had to destroy a descending centipede while avoiding other insects. "Defender" (1980) by Williams Electronics offered a scrolling, multidirectional shooter experience with impressive graphics and complex controls.

In terms of technological advancements, the golden age witnessed the introduction of vector graphics, which enabled smoother and more detailed visuals in games like "Battlezone" (1980) and "Star Wars" (1983). Additionally, the development of improved sound hardware allowed for more immersive audio experiences, enhancing the overall gameplay.

2.3.5 Cultural Impact and Legacy

The golden age of arcade games left a profound cultural impact. Arcade gaming became a social activity, attracting players of all ages and backgrounds to gather and compete for high scores. Arcades themselves became vibrant social spaces, fostering a sense of community among gamers.

These games also influenced popular culture, with characters like Pac-Man and Mario becoming household names

2.4 The Rise of Personal Computers and Consoles (1980 - 1990)

The period from 1980 to 1990 witnessed a significant rise in the popularity and accessibility of personal computers (PCs) and gaming consoles, laying the foundation for a revolution in home computing and entertainment. This era marked the beginning of a technological transformation that would shape the future of the gaming industry and personal computing.

The 1980s saw the emergence of affordable and accessible personal computers, which brought the power of computing into the hands of individuals and small businesses. Prominent computer systems during this period included the Apple II, Commodore 64, and IBM PC.

These PCs offered a range of functionalities, from basic word processing and spreadsheet applications to early programming capabilities. They became valuable tools for personal productivity and creative pursuits, enabling users to explore their interests and unleash their creativity.

Additionally, the introduction of the graphical user interface (GUI) by Apple's

Macintosh in 1984 revolutionized the user experience, making computers more intuitive and user-friendly. The GUI allowed users to interact with icons, menus, and windows, providing a visual and graphical way of navigating the computer system.

The 1980s marked the rise of gaming consoles, which brought video gaming into the living rooms of millions of households. These consoles provided a dedicated gaming experience, catering specifically to the entertainment needs of users. One of the most iconic consoles of this era was the Atari 2600, which was widely successful and featured a diverse library of games. The console introduced gaming experiences like "Pac-Man," "Space Invaders," and "Donkey Kong" into homes, captivating players with their engaging gameplay. Other notable consoles included the Intellivision and ColecoVision, which competed for market share and brought a range of innovative and exciting gaming experiences to players.

Throughout the 1980s, both PCs and consoles experienced significant technological advancements that enhanced their capabilities and expanded the possibilities of gaming and computing. PCs saw continuous improvements in processing power, storage capacity, and graphics capabilities. These advancements enabled the development of more sophisticated software and games, offering enhanced visuals and immersive gameplay experiences.Meanwhile, consoles evolved from simple, pixelated graphics to more refined and detailed visuals. The introduction of 8-bit and later 16-bit systems like the Nintendo Entertainment System (NES) and Sega Genesis allowed for more advanced graphics and sound capabilities, bringing games to life with vibrant colors and engaging audio.

During this era, several influential game genres and franchises emerged, setting the stage for the future of gaming. Text-based adventures and interactive fiction games captivated players' imaginations, while side-scrolling platformers like "Super Mario Bros." and "Sonic the Hedgehog" introduced memorable characters and exciting gameplay mechanics. Role-playing games (RPGs) like "Final Fantasy" and "The Legend of Zelda" offered immersive narratives and expansive worlds to explore. Additionally, the rise of first-person shooters began with games like "Doom" and "Wolfenstein 3D," revolutionizing the way players experienced action and combat in video games. These genres and franchises continue to influence and inspire the gaming industry to this day, showcasing the lasting impact of the developments from the 1980s to 1990s.

2.5 The 3D Revolution and Multimedia Experiences (1990s)

The 1990s marked a significant turning point in the world of gaming and personal computing, characterized by the 3D revolution and the emergence of multimedia experiences. This era witnessed a rapid advancement in technology, pushing the

boundaries of graphics, sound, and interactivity.

The 1990s witnessed a groundbreaking shift from 2D to 3D graphics, which revolutionized the visual aesthetics and gameplay experiences of video games. This transformation was made possible by advancements in hardware capabilities and software development.

The introduction of powerful graphics processing units (GPUs) and dedicated 3D accelerators facilitated the rendering of complex 3D models, textures, and environments in real-time. This led to the creation of immersive game worlds with depth, perspective, and realistic visual effects.

Games like "Doom" (1993) and "Quake" (1996) pioneered the use of 3D graphics in the first-person shooter genre, offering players a thrilling sense of exploration and heightened immersion. These titles showcased dynamic lighting, intricate level design, and smooth character movements, setting new standards for visual fidelity in gaming.

The 1990s also witnessed a rise in multimedia experiences, with advancements in audio and video technology. PCs and gaming consoles began incorporating CD-ROM drives, enabling the storage and playback of high-quality audio and video content. CD-based games allowed developers to incorporate full-motion videos (FMV), voice acting, and cinematic cutscenes, enhancing storytelling and player engagement. Games like "Resident Evil" (1996) and "Final Fantasy VII" (1997) capitalized on these multimedia capabilities, delivering immersive narratives and memorable cinematic moments. The integration of Red Book audio allowed for richer and more immersive soundtracks in games. Composers like Nobuo Uematsu ("Final Fantasy" series) and Jeremy Soule ("The Elder Scrolls" series) created iconic musical scores that enhanced the emotional impact and atmosphere of the gaming experience. Furthermore, the emergence of multimedia software applications, such as multimedia authoring tools and media players, enabled users to create and consume a wide range of multimedia content, including interactive presentations, digital art, and educational materials.

The 1990s also saw the early development and experimentation with virtual reality and augmented reality technologies. Although these technologies were in their infancy during this period, they laid the foundation for future advancements. VR systems like the Virtuality Arcade Machines and VFX1 Headgear offered users a taste of immersive virtual environments, albeit with limited graphics and bulky hardware. These early VR experiences paved the way for the future development of consumer-grade VR devices. Augmented reality, although not as prominent as VR, began to make its presence felt with applications like the AR-fueled "Virtual Boy" console from Nintendo. While the Virtual Boy did not achieve commercial success, it laid the groundwork for future AR advancements in gaming and other industries.

The 1990s witnessed a significant rise in multiplayer and online gaming, facilitated by advancements in networking technologies. PC gaming saw the emergence of LAN (local area network) parties, where players connected their computers to play games together in the same physical space. Online gaming took off with the introduction of dial-up internet connections and services like America Online (AOL). Games like "Quake" and "StarCraft" (1998) popularized online multiplayer experiences, allowing players to compete or collaborate with others worldwide. The rise of online gaming not only fostered a sense of community and competition but also laid the groundwork for the future development of massively multiplayer online games (MMOs) like "Ultima Online" (1997) and "EverQuest" (1999).

2.6 The Transition to Online and Multiplayer Gaming (2000s)

The 2000s marked a significant transition in the world of gaming, as online and multiplayer gaming gained widespread popularity. This era witnessed the convergence of advancing technology, improved internet connectivity, and the desire for social interaction in gaming experiences.

The availability of high-speed broadband internet connections became more widespread during the 2000s. This shift from dial-up to broadband connectivity allowed for faster and more reliable online gaming experiences. Players could now connect with others around the world without the limitations of slow and unstable connections. Internet Service Providers (ISPs) began offering affordable broadband packages, making it more accessible for gamers to engage in online multiplayer experiences. This improved infrastructure provided a solid foundation for the growth of online gaming communities.

The 2000s saw the rise of massively multiplayer online games (MMOs) that allowed thousands of players to coexist and interact in persistent virtual worlds. Games like "World of Warcraft" (2004), "EverQuest II" (2004), and "Guild Wars" (2005) gained immense popularity, creating thriving online communities. MMOs offered players the opportunity to embark on epic quests, engage in player versus environment (PvE) or player versus player (PvP) combat, and participate in virtual economies. These games fostered social connections, guilds, and cooperative gameplay, encouraging collaboration and teamwork. The success of MMOs was due, in part, to advancements in server technology and scalable infrastructure that could accommodate large numbers of players simultaneously. Developers continued to expand and update these virtual worlds, providing ongoing content updates, expansions, and events to keep players engaged.

The 2000s also witnessed the integration of online gaming capabilities into home consoles. Companies like Microsoft with the Xbox Live service and Sony with the PlayStation Network introduced robust online platforms that connected console players worldwide. Online console gaming allowed players to compete against each other, join multiplayer matches, and communicate through voice chat. Games like "Halo 2" (2004), "Call of Duty" series, and "FIFA" series brought intense multiplayer experiences to consoles, fostering a competitive and social gaming environment. Console manufacturers introduced features like achievement systems, leaderboards, and downloadable content (DLC), enhancing the longevity and replayability of games. The ability to connect with friends, join parties, and form online communities further contributed to the popularity of online console gaming.

The 2000s also witnessed the rise of competitive gaming and the emergence of esports as a global phenomenon. With the increasing availability of online multiplayer games, competitive gaming leagues and tournaments gained popularity. Esports events like the World Cyber Games, Electronic Sports World Cup, and Major League Gaming attracted professional players and showcased their skills in games such as "Counter-Strike," "StarCraft," and "League of Legends." These events garnered large audiences and significant prize pools, legitimizing competitive gaming as a professional sport. Streaming platforms like Twitch and YouTube allowed gamers to broadcast their gameplay live, further boosting the visibility and accessibility of esports. Spectatorship of competitive gaming became a major aspect, with fans watching matches and supporting their favorite teams and players.

The 2000s witnessed the rise of social networking platforms like MySpace, Facebook, and later, Twitter. These platforms provided avenues for gamers to connect, share experiences, and form online communities centered around gaming. Online forums, gaming communities, and dedicated gaming websites allowed players to discuss strategies, share tips, and participate in vibrant discussions. This sense of community and interaction transcended geographical boundaries, fostering friendships and connections among gamers worldwide. Developers also incorporated social features within games, allowing players to form in-game friendships, create clans or guilds, and engage in cooperative gameplay. Online gaming became a social experience that extended beyond the virtual world, with players organizing meetups and conventions to connect in person.

2.7 Gaming Goes Mobile (2000s-2010s)

The 2000s to 2010s witnessed a significant shift in the gaming industry as mobile gaming gained prominence. With the rapid advancement of mobile technology and the widespread adoption of smartphones and tablets, gaming became accessible to

a broader audience than ever before. This era marked the rise of mobile gaming as a major sector within the gaming industry.

The advent of smartphones and tablets in the late 2000s revolutionized the way people interacted with technology. These portable devices offered powerful processors, high-resolution displays, and touchscreens, providing an ideal platform for gaming on the go. Apple's introduction of the App Store in 2008 and Google's Play Store in 2012 provided centralized platforms for developers to distribute their games to millions of mobile users worldwide. This opened up new opportunities for independent developers and small studios to create and publish games without the need for expensive physical distribution.

Mobile gaming brought a new breed of casual games that catered to the growing audience of casual gamers. These games were easy to learn, quick to play, and could be enjoyed in short bursts of time, making them ideal for mobile devices. Titles like "Angry Birds" (2009), "Candy Crush Saga" (2012), and "Flappy Bird" (2013) became immensely popular, appealing to both seasoned gamers and newcomers. These games featured simple mechanics, addictive gameplay, and intuitive touch controls, capturing the attention of millions of players.

The mobile gaming market popularized the free-to-play (F2P) model, where games could be downloaded and played for free, with revenue generated through in-app purchases (IAPs) or advertisements. This model allowed players to experience games without upfront costs, but offered optional microtransactions to enhance gameplay or unlock additional content. Virtual currencies, power-ups, cosmetic items, and level boosts became common monetization strategies. Popular games like "Clash of Clans" (2012), "Pokémon Go" (2016), and "Fortnite" (2017) utilized this model effectively, generating substantial revenue through IAPs and building dedicated player communities.

Mobile gaming brought social interactions and multiplayer experiences to the palm of players' hands. Online multiplayer games, including real-time multiplayer and asynchronous gameplay, allowed players to compete or collaborate with friends and strangers worldwide. Social features such as leaderboards, achievements, and friend challenges added a competitive element and encouraged player engagement. Games like "Words with Friends" (2009), "Draw Something" (2012), and "Clash Royale" (2016) exemplified the social gaming aspect of mobile games, fostering friendly competition and social connections.

One of the key advantages of mobile gaming was the portability and ubiquity it offered. Players could carry their favorite games with them anywhere, allowing for gaming experiences during commutes, waiting in lines, or in idle moments throughout the day. The integration of cloud-based services and cross-platform compatibility further expanded the reach of mobile gaming. Games could be synced across

multiple devices, enabling players to seamlessly transition between their smartphones, tablets, and even PCs or consoles.

The mobile gaming landscape provided a platform for independent developers and small studios to showcase their creativity and innovative ideas. The relatively low barrier to entry and direct distribution channels empowered indie developers to experiment with unique game concepts and art styles. Games like "Monument Valley" (2014), "Limbo" (2010), and "Alto's Adventure" (2015) demonstrated the artistic and storytelling potential of mobile gaming. These titles pushed the boundaries of what was considered possible on a mobile device, delivering memorable and immersive experiences.

2.8 The Era of Immersive and Virtual Reality (2010s)

The 2010s witnessed a significant advancement in immersive technologies, particularly in the field of virtual reality (VR) and augmented reality (AR). This era marked a new frontier in gaming and interactive experiences, pushing the boundaries of immersion, realism, and interactivity.

The 2010s saw a resurgence of interest in virtual reality, driven by advancements in hardware, software, and increased accessibility. Head-mounted displays (HMDs) with improved optics, tracking systems, and motion controllers were introduced, offering more immersive and accurate experiences. Oculus Rift, released in 2016, played a significant role in popularizing modern VR. Its Kickstarter campaign and subsequent acquisition by Facebook generated widespread attention and investment in VR technology. Other major players like HTC Vive, PlayStation VR, and later, Oculus Quest, further expanded the availability of VR experiences. VR gaming allowed players to step into virtual worlds, providing a sense of presence and immersion like never before. Games like "Superhot VR" (2016), "Beat Saber" (2018), and "Half-Life: Alyx" (2020) showcased the potential of VR, offering unique gameplay mechanics and realistic interactions.

The 2010s also marked significant breakthroughs in augmented reality technology, blurring the line between the real world and digital overlays. Mobile devices became primary AR platforms, leveraging their cameras and sensors to overlay virtual objects onto the real-world environment. "Pokémon Go" (2016) became a global phenomenon, introducing millions of players to the concept of AR gaming. Players could explore their surroundings, capture virtual creatures, and engage in battles, blending the real world with the virtual realm. Additionally, companies like Microsoft introduced dedicated AR devices such as the HoloLens, enabling users to interact with holographic digital content overlaid onto their physical environment. This technology found applications beyond gaming, ranging from industrial design

and architecture to education and training.

The era of immersive and virtual reality opened up new possibilities for storytelling and narrative experiences. VR allowed creators to transport users into captivating virtual worlds and deliver compelling narratives through first-person perspectives. Narrative-driven VR experiences like "Lone Echo" (2017), "The Walking Dead: Saints and Sinners" (2020), and "Moss" (2018) showcased the power of immersion in building emotional connections and engaging storytelling. Beyond gaming, VR also found applications in other industries, such as virtual tourism, virtual training simulations, and therapeutic experiences. Medical professionals, for example, utilized VR to simulate surgeries and treatments, while architects and designers used VR to create immersive walkthroughs of building projects.

While the 2010s saw significant advancements in VR and AR, there were also challenges to overcome. The cost of entry, hardware limitations, and issues with motion sickness and comfort remained barriers for widespread adoption. However, ongoing developments and innovations in technology continued to address these concerns. The future potential of immersive and virtual reality is vast. Advancements in haptic feedback, eye-tracking, and wireless technology are enhancing the realism and comfort of VR experiences. The convergence of VR with other emerging technologies like artificial intelligence, 5G connectivity, and cloud computing holds promise for even more immersive and interactive experiences. Additionally, the potential of mixed reality (MR), which combines elements of VR and AR, is being explored. MR seeks to seamlessly merge the virtual and real world, enabling users to interact with both digital and physical objects simultaneously.

Chapter 3

Used Technologies

3.1 Unity

Unity is a cross-platform game engine and development platform that is widely used in the video game industry to create 2D and 3D games, as well as other interactive experiences such as virtual reality (VR) and augmented reality (AR) applications. It was developed by Unity Technologies and first released in 2005.

Unity allows developers to create games and other interactive experiences using a visual editor and a scripting language. The engine provides a wide range of tools and features, including physics simulation, 3D rendering, animation, audio processing, and networking. It also supports a wide range of platforms, including desktop computers, mobile devices, and game consoles.

One of the key features of Unity is its accessibility. It is designed to be easy to learn and use, even for beginners with no prior experience in game development. This has helped to make it a popular choice for indie game developers and small studios who may not have the resources to develop their own game engines from scratch.

Unity also has a strong community of developers who create and share assets, plugins, and other tools to extend the engine's capabilities. This helps to make it a flexible and versatile platform that can be customized to meet the needs of a wide range of projects and development workflows.

Overall, Unity is a powerful and popular game engine that has helped to democratize game development and make it more accessible to a wider audience. Its ease of use, wide range of features, and support for multiple platforms have made it a popular choice for both professional and amateur game developers around the world.

3.1.1 MonoBehaviour

In Unity, MonoBehaviour is a base class for scripts that are used to add behaviors and functionality to game objects. It is a fundamental component of the Unity engine and is used extensively in game development.

MonoBehaviour provides a range of methods that can be overridden in derived classes to control the behavior of a game object. For example, the Start() method is called when the script is first enabled, and can be used to set up initial state and behavior. The Update() method is called every frame, and can be used to update the behavior of the script based on input or other factors.

MonoBehaviour also provides access to a range of Unity-specific functionality, such as physics simulations, audio processing, and 3D rendering. By using these features, developers can create complex and engaging game experiences that take full advantage of the Unity engine.

One of the key features of MonoBehaviour is its ability to be attached to game objects. When a MonoBehaviour script is attached to a game object, it can interact with other components on that object, as well as with other objects in the game world. This allows developers to create complex and interactive game objects that can respond to player input and other environmental factors.

MonoBehaviour is a key component of the Unity engine and is used extensively in game development. By providing access to a wide range of functionality and by allowing scripts to be attached to game objects, it helps to enable the creation of engaging and immersive game experiences.

3.1.2 Game Objects

In Unity, a game object is the basic building block for creating interactive experiences. It represents a single entity in a game world and can be used to represent everything from characters and scenery to lights and sound effects.

A game object in Unity is essentially an empty container that can have various components attached to it. Components are scripts or other assets that add functionality to a game object, such as rendering graphics, controlling physics, or handling input. Each game object can have any number of components attached to it, and each component can have its own properties and behaviors.

Game objects can be created and managed using the Unity Editor, which provides a visual interface for creating and manipulating game objects and their components. Developers can add, remove, and modify components on a game object, as well as adjust the game object's position, rotation, and scale.

Game objects can also be organized into hierarchies, which allow developers to create complex scenes with multiple objects that are all positioned and scaled

relative to each other. This can be useful for creating scenes with complex lighting, physics interactions, or other dynamic behaviors.

In Unity, game objects are at the center of the game development process, and understanding how they work is essential for creating interactive experiences. By creating and managing game objects, developers can create complex and immersive game worlds that engage players and provide hours of entertainment.

3.1.3 Scriptable Objects

Scriptable Objects are a unique feature of the Unity game engine that allow developers to create data containers that can be easily reused and shared across multiple game objects and scenes. They are essentially asset files that contain data but no behaviors or functionality.

Scriptable Objects are a way of separating data from the code that uses it, which can make it easier to manage and modify game data. They can be thought of as templates for creating data objects, which can be customized and modified as needed. In Unity, Scriptable Objects can be created using C# scripts and then saved as assets in the project.

One of the main benefits of using Scriptable Objects in Unity is that they can be easily shared and reused across multiple game objects and scenes. This can help reduce the amount of duplicate code and data that needs to be created and maintained, which can make development faster and more efficient.

Another benefit of using Scriptable Objects is that they can be modified at runtime without affecting other instances of the same Scriptable Object. This can be useful for creating dynamic game content or for creating game systems that can be easily customized by players.

Scriptable Objects can be used for a wide range of purposes in Unity, including creating game data such as character stats, item definitions, or game events. They can also be used to create reusable behaviors such as AI actions or UI elements.

Overall, Scriptable Objects are a powerful and flexible feature of the Unity game engine that can help developers create more efficient, reusable, and customizable game data and behaviors.

3.1.4 Compute shaders

Compute shaders are a type of shader in Unity that allow developers to perform general-purpose computations on the graphics processing unit (GPU) using the parallel processing power of modern graphics cards. Compute shaders are written in a special language called HLSL (High-Level Shading Language) and can be used for a

wide range of tasks, including physics simulations, image processing, and artificial intelligence.

Unlike traditional shaders, which are used to control the visual appearance of game objects, compute shaders are focused on data processing and manipulation. They are executed in parallel across multiple threads on the GPU, which allows them to process large amounts of data quickly and efficiently.

Compute shaders are often used in situations where a large amount of data needs to be processed in real-time, such as in particle systems or physics simulations. They can also be used to create complex procedural effects, such as dynamic fluid simulations or dynamic weather systems.

One of the benefits of using compute shaders in Unity is that they can help to offload processing tasks from the CPU to the GPU, which can improve performance and reduce the workload on the main processor. This can be particularly useful in situations where complex computations are required, such as in AI or physics simulations.

In addition to their performance benefits, compute shaders are also highly flexible and customizable. They can be used to create a wide range of effects and behaviors, and can be customized and tweaked as needed to achieve the desired result.

Overall, compute shaders are a powerful tool for game developers working in Unity. They provide a way to offload processing tasks to the GPU, which can improve performance and reduce the workload on the CPU. They are also highly flexible and customizable, making them a valuable tool for creating complex and dynamic effects in games and other interactive applications.

Chapter 4

Creating the environment

4.1 Procedural terrain generation using the GPU

Procedural terrains have traditionally been limited to height fields that are generated by the CPU and rendered by the GPU. However, the serial processing nature of the CPU is not well suited to generating extremely complex terrains—a highly parallel task. Plus, the simple height fields that the CPU can process do not offer interesting terrain features (such as caves or overhangs).

To generate procedural terrains with a high level of complexity, at interactive frame rates, we look to the GPU. By utilizing several new DirectX 10 capabilities such as the geometry shader (GS), stream output, and rendering to 3D textures, we can use the GPU to quickly generate large blocks of complex procedural terrain. Together, these blocks create a large, detailed polygonal mesh that represents the terrain within the current view frustum.

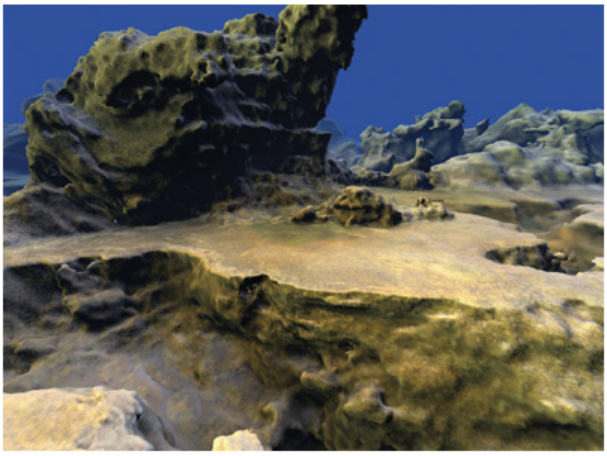


Figure 4.1: Terrain Created Entirely on the GPU [GCP]

4.2 Marching Cubes and the Density Function

4.2.1 Marching Cubes Algorithm

Marching cubes uses a divide-and-conquer approach to locate the surface in a logical cube created from eight pixels; four each from two adjacent slices (Figure 1.2)

The algorithm determines how the surface intersects this cube, then moves (or marchs) to the next cube. To find the surface intersection in a cube, we assign a one to a cube's vertex if the data value at that vertex exceeds (or equals) the value of the surface we are constructing. These vertices are inside (or on) the surface. Cube vertices with values below the surface receive a zero and are outside the surface. The surface intersects those cube edges where one vertex is outside the surface (one) and the other is inside the surface (zero). With this assumption, we determine the topology of the surface within a cube, finding the location of the intersection later.

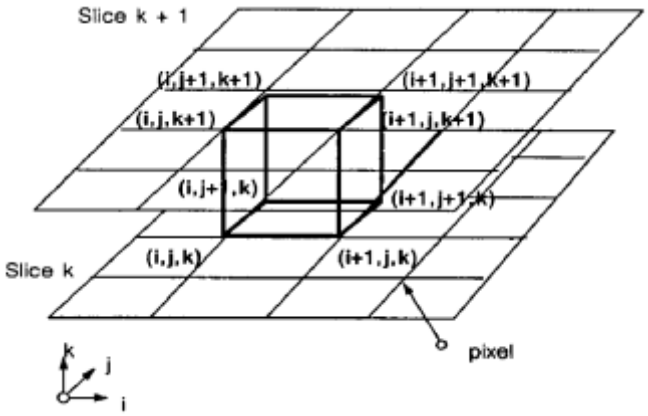


Figure 4.2: Marching Cube [WEL87]

Since there are eight vertices in each cube and two slates, inside and outside, there are only 28 = 256 ways a surface can intersect the cube. By enumerating these 256 cases, we create a table to look up surface-edge intersections, given the labeling of a cubes vertices. The table contains the edges intersected for each case.

Triangulating the 256 cases is possible but tedious and error-prone. Two different symmetries of the cube reduce the problem from 256 cases to 14 patterns. First, the topology of the triangulated surface is unchanged if the relationship of the surface values to the cubes is reversed. Complementary cases, where vertices greater than the surface value are interchanged with those less than the value, are equivalent. Thus, only cases with zero to four vertices greater than the surface value need be considered, reducing the number of cases to 128. Using the second symmetry property, rotational symmetry, we reduced the problem to 14 patterns by inspection. Figure 1.3 shows the triangulation for the 14 patterns.

The simplest pattern, 0, occurs if all vertex values are above (or below) the se-

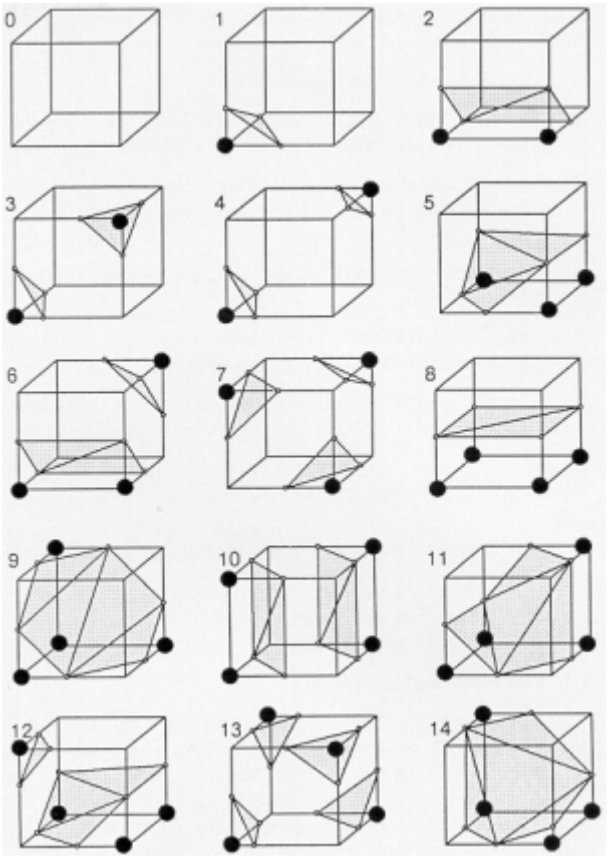


Figure 4.3: Triangulated Cubes [WEL87]

lected value and produces no triangles. The next pattern, 1, occurs if the surface separates on vertex from the other seven, resulting in one triangle defined by the three edge intersections. Other patterns produce multiple triangles. Permutation of these 14 basic patterns using complementary and rotational symmetry produces the 256 cases.

We create an index for each case, based on the state of the vertex. Using the vertex numbering in Figure 1.4, the eight bit index contains one bit for each vertex.

This index serves as a pointer into an edge table that gives all edge intersections for a given cube configuration.

Using the index to tell which edge the surface intersects, we can interpolate the surface intersection along the edge.

4.2.2 The Terrain Generation System

We divide the world into an infinite number of equally sized cubic blocks, as already described. In the world-space coordinate system, each block is 1x1x1 in size. However, within each block are 323 voxels that potentially contain polygons. A pool of around 300 vertex buffers are dynamically assigned to the blocks currently visible in the view frustum, with higher priority given to the closest blocks. As new blocks

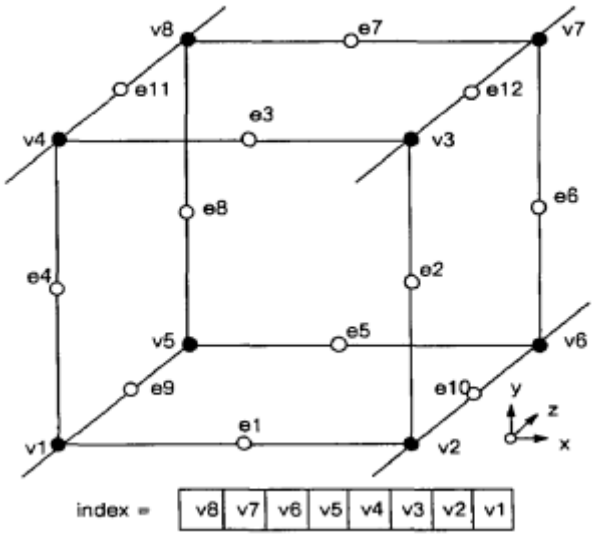


Figure 4.4: Triangulated Cubes [WEL87]

come into the view frustum (as the user moves around), the farthest away or newly view-cullable vertex buffers are evicted and reused for the newly wished-for blocks.

Not all blocks contain polygons. Whether they do or not depends on complex calculations, so usually we won't know if they contain polygons until we try to generate the blocks. As each block is generated, a stream-out query asks the GPU if any polygons were actually created. Blocks that don't produce polygons—this is common—are flagged as "empty" and put into a list so they won't be uselessly regenerated. This move also prevents those empty blocks from unnecessarily occupying a vertex buffer.

For each frame, we sort all the vertex buffers (their bounding boxes are well known) from front to back. We then generate any new blocks that are needed, evicting the most distant block whenever we need a free vertex buffer. Finally, we render the sorted blocks from front to back so that the GPU doesn't waste time shading pixels that might be occluded by other parts of the terrain.

4.2.3 Generating the Polygons Within a Block of Terrain

First, we use the GPU's pixel shader (PS) unit to evaluate the complex density function at every cell corner within the block and store the results in a large 3D texture. The blocks are generated one at a time, so one 3D texture can be shared universally. However, because the texture stores the density values at the cell corners, the texture is 33x33x33 in size, rather than 32x32x32 (the number of cells in the block).

Next, we visit each voxel and generate actual polygons within it, if necessary. The polygons are streamed out to a vertex buffer, where they can be kept and repeatedly rendered to the screen until they are no longer visible.

4.2.4 Making an Interesting Density Function

The sole input to the density function is this:

```
float3 ws;
```

This value is the world-space coordinate. Luckily, shaders give us plenty of useful tools to translate this value into an interesting density value. Some of the tools at our disposal include the following:

- Sampling from source textures, such as 1D, 2D, 3D, and cube maps
- Constant buffers, such as lookup tables
- Mathematical functions, such as cos(), sin(), pow(), exp(), frac(), floor(), and arithmetic

A good starting point is to place a ground plane at y=0:

```
float density = -ws.y;
```

This divides the world into positive values, those below the y = 0 plane (let's call that earth), and negative values, those above the plane (which we'll call air). A good start! Figure 1.5 shows the result.

Figure 4.5: Flat Surface [GCP]

Now we can make the ground look more interesting by adding some randomness. We do this by using the world-space coordinate (ws) to sample from a small repeating 3D texture full of random (noise) values in the range [-1..1]:

```
density += noiseVol.Sample(TrilinearRepeat, ws).x;
```

We can scale ws prior to the texture lookup to change the frequency (how quickly the noise varies over space). We can also scale the result of the lookup before adding it to the density value in order to change the amplitude, or the strength of the noise. Figure 1.7 uses a noise function with twice the frequency and half the amplitude of the previous example:

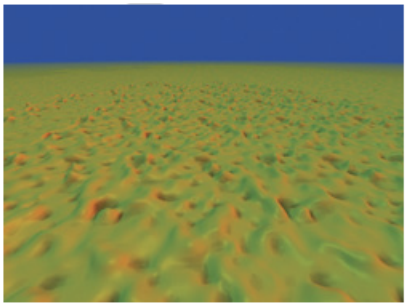


Figure 4.6: Terrain with one level of noise [GCP]

```
density += noiseVol.Sample(TrilinearRepeat, ws*2).x*0.5;
```

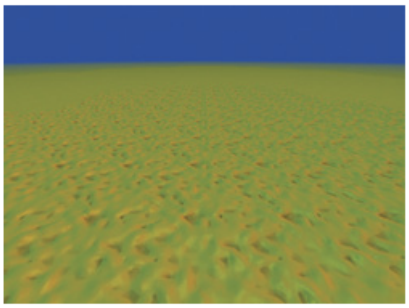


Figure 4.7: Terrain with one level of noise with twice the frequency and half the amplitude [GCP]

To get a more interesting result, we can use more than just one level of noise. The amplitude of each level should be half that of the previous one, and the frequency should be roughly double. It's important not to make the frequency exactly double, though. The interference of two overlapping, repeating signals at slightly different frequencies is beneficial here because it helps break up repetition.

```
density += noiseVol1.Sample(TrilinearRepeat, ws*4.03).x*0.25;
density += noiseVol2.Sample(TrilinearRepeat, ws*1.96).x*0.50;
density += noiseVol3.Sample(TrilinearRepeat, ws*1.01).x*1.00;
```

If more levels of noise are added at progressively lower frequencies (and higher amplitudes), larger terrain structures begin to emerge, such as large mountains and trenches. In practice, we need about nine levels of noise to create a world that is rich in both of these low-frequency features (such as mountains and canyons), but that also retains interesting high-frequency features (random detail visible at close range)

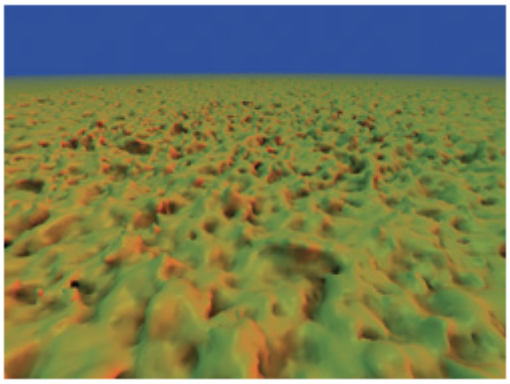


Figure 4.8: Three levels of noise at high frequency generate more details [GCP]

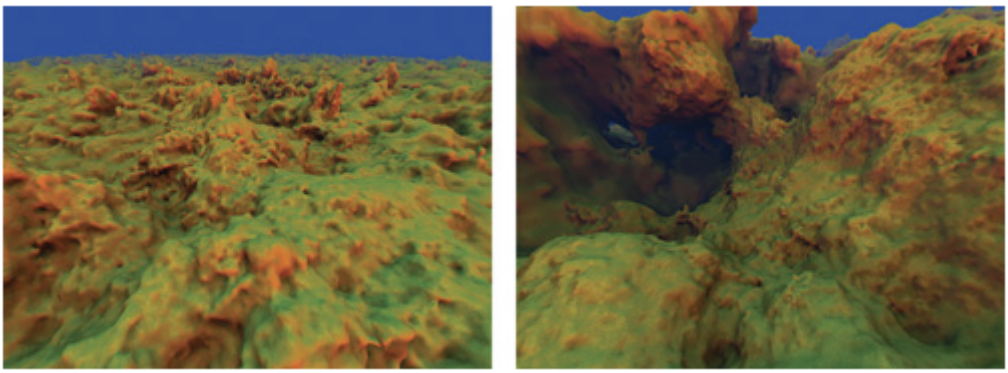


Figure 4.9: Adding Lower Frequencies at Higher Amplitude Creates Mountains [GCP]

4.3 Endless terrain

In order to get an infinite world, we need to find a way to load only the area actually visible to the player, because loading an infinite map would require infinite memory. A solution is to split the map into square chunks, and every time the player moves the chunks that go outside the players vision range get unloaded, and the chunks that get inside the players vision get loaded.

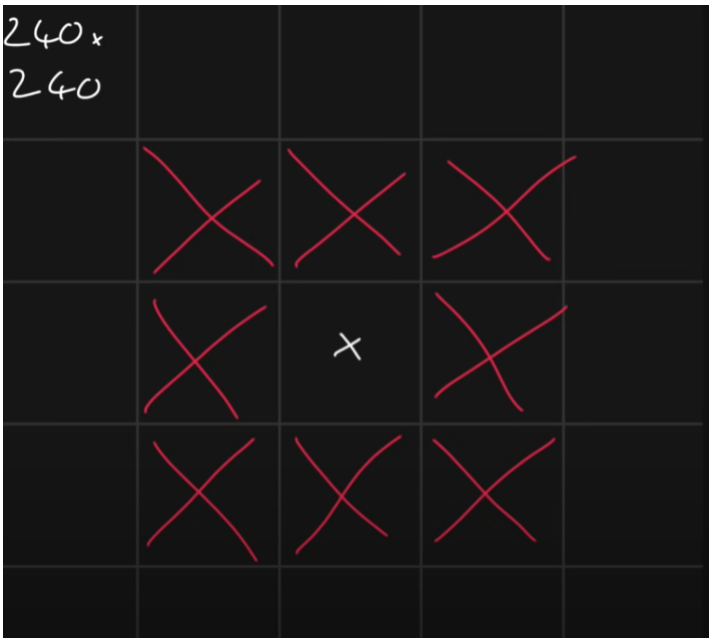


Figure 4.10: Map splitted in chunks. Only chunks marked with X are loaded

A new class can be created, called TerrainChunk. It will have a position and data about the mesh. This data needs to be requested from our MarchinCubes class. Because it takes some time to finish all the calculations for generating the mesh, we can run a separate thread for each chunk that needs to be loaded, and store the results inside a queue. Now, every time there is data inside the queue, it can be sent to the corresponding chunk.

```
public void requestTerrainData(Vector3 position,
   Action<TerrainData> callback)
{
    ThreadStart threadStart = delegate
    {
        TerrainDataThread(position, callback);
    };

    new Thread(threadStart).Start();
}
```

```
void TerrainDataThread(Vector3 position, Action<TerrainData>
   callback)
{
    TerrainData terrainData = GetTerrainData(position);
    lock (terrainDataThreadInfoQueue)
    {
        terrainDataThreadInfoQueue.Enqueue(new
           TerrainThreadInfo<TerrainData>(callback, terrainData));
    }
}

private void Update()
{
    if (terrainDataThreadInfoQueue.Count > 0)
    {
        for (int i = 0; i < terrainDataThreadInfoQueue.Count; i++)
        {
            TerrainThreadInfo<TerrainData> threadInfo =
               terrainDataThreadInfoQueue.Dequeue();
            threadInfo.callback(threadInfo.parameter);
        }
    }
}
```

Now we can call the requestTerrainData from the TerrainChunk's constructor and set a callback to OnTerrainDataReceived function.

```
void OnMapDataReceived(TerrainData terrainData)
{
    this.terrainData = terrainData;
    this.UpdateTerrainChunk();
}
```

The UpdateTerrainChunk method will test if the chunk is still visible to the player. If it is visible and the mesh is not built, it will construct the mesh. If the chunk is no longer visible, the mesh will be destroyed.

```
public void UpdateTerrainChunk()
{
    float viewerDistFromNearestEdge =
       Mathf.Sqrt(bounds.SqrDistance(viewerPos));
    bool visible = viewerDistFromNearestEdge <= maxViewDist;
```

```
    SetVisible(visible);
}

public void SetVisible(bool visible)
{
    this.isVisible = visible;
    if (visible)
    {
        BuildMesh();
    }
    else
    {
        ClearMesh();
    }
}
```

Now that we have a way of requesting the data to build a chunk, we can calculat which are the visible chunks to the player and update them accordingly. If the chunk already exists, we can just call the update method defined above. If not, we crate a new chunk.

```
void updateVisibleChunks()
{
    foreach (var chunk in terrainChunksVisibleLastUpdate)
    {
        var distToChunk =
           Mathf.Sqrt(chunk.GetBounds().SqrDistance(viewerPos));
        if (distToChunk > maxViewDist)
        {
            chunk.SetVisible(false);
        }
    }

    terrainChunksVisibleLastUpdate.RemoveWhere(x => !x.IsVisible());

    int currentChunkCoordX = Mathf.RoundToInt(viewerPos.x /
       chunkSize);
    int currentChunkCoordY = Mathf.RoundToInt(viewerPos.y /
       chunkSize);
```

```
    int count = 0;
    for (int yOffset = -chunksVisibleInViewDst; yOffset <=
       chunksVisibleInViewDst; yOffset++)
    {
        for (int xOffset = -chunksVisibleInViewDst; xOffset <=
           chunksVisibleInViewDst; xOffset++)
        {
            count++;
            Vector2 viewedChunkCoord = new
               Vector2(currentChunkCoordX + xOffset,
               currentChunkCoordY + yOffset);

            ManageChunkAtPosition(viewedChunkCoord);
        }
    }
}

private void ManageChunkAtPosition(Vector2 position)
{
    if (terrainChunkDict.ContainsKey(position))
    {
        terrainChunkDict[position].UpdateTerrainChunk();
    }
    else
    {
        terrainChunkDict.Add(position, new TerrainChunk(position,
           chunkSize, transform, material, gradient));
    }
}
```

Chapter 5

Adding wildlife

5.1 The importance of wildlife in videogames

Wildlife can play an important role in creating a sense of immersion and aesthetic appeal in video games. When players encounter animals in a game world, they may feel like they are exploring a more realistic and believable environment. Additionally, seeing animals in games can simply be visually appealing and contribute to the overall ambiance of the game.

By including realistic depictions of wildlife, game developers can help players feel like they are part of a living, breathing world. For example, in open-world games, players may encounter various species of animals while exploring. Seeing these animals interact with each other and their surroundings can make the game world feel more dynamic and immersive.

Moreover, wildlife can help to tell stories about the game world's history or ecology. Game developers can use extinct animals, for instance, to suggest a world that has been drastically altered by human activity. Alternatively, they can use the presence or absence of certain animals to tell a story about the state of the environment or the balance of power between different factions in the game world.

In terms of aesthetics, many players simply enjoy seeing animals in games because they appreciate their beauty and uniqueness. By including detailed and realistic models of wildlife, game developers can create visually stunning environments that players will want to explore and admire. Even if the animals themselves do not serve a specific gameplay purpose, they can still contribute to the overall experience of playing the game.

So, wildlife plays a key role in creating a sense of immersion and aesthetic appeal in video games. By including realistic and detailed depictions of animals in their games, developers can help players feel like they are exploring a living, breathing world, and create visually stunning environments that players will enjoy.

Figure 5.1: Example of wildlife in popular videogames

5.2 Natural Flocks, Herds, and Schools

A bird has to exhibit behaviors that enable it to sync its movements with those of its flockmates in order to be a part of a flock. These behaviours are not particularly unusual; they are exhibited to a certain extent by all living things. The need to stay with the flock and an instinct to prevent collisions inside the flock appear to be two balanced, competing tendencies that make up natural flocks [Sha75]. It is obvious why an individual bird would prefer to stay apart from its flock members. But what is the reason birds seem to prefer the airborne version of an unpleasant traffic jam?The fundamental urge to flock appears to be the result of evolutionary pressure from a number of factors, including protection from predators, improving the chances of survival, benefiting from a larger and more effective pattern in the search for food, and also social and mating activities.

There is no proof that natural flocks' complexity is constrained in any manner. As new birds join, flocks do not become "full" or "overloaded". Herring run in schools that may be up to 17 miles long and include millions of fish as they move to their breeding sites [Sch83]. Natural flocks appear to function uniformly over a wide range of flock numbers. An individual bird does not appear to be able to pay much attention to every member of its flock. However, a single bird in a large flock that is dispersed over a large area needs to have a localized and filtered perception of the other birds in the flock. A bird may be aware of three groups: the flock as a whole, its two or three closest neighbors, and itself [Par82].

These hypotheses on the "computational complexity" of flocking are designed to imply that since birds employ what would be referred to as a constant time algorithm in formal computer science, they may flock with any number of flockmates. This means that the amount of "thinking" required for a flock to form must be substantially independent of the number of birds involved. If not, we would anticipate a sharp upper limit on the size of natural flocks when the intricacy of the individual birds' navigational job grew too much for them to handle. This hasn't been seen in the natural world.

5.3 Boids

Boids are computer-generated animations that simulate the behavior of flocks of birds or other swarming entities. The concept of boids was introduced by computer scientist Craig Reynolds in 1986 and has since become a popular tool for simulating complex patterns of movement. The name "boid" is short for "bird-oid object," which reflects the fact that the original concept was developed to simulate bird flocks.[Rey87]

The key idea behind boids is that complex patterns of movement can emerge from simple rules governing individual agents' behavior. In the case of boids, each individual bird is programmed to follow three simple rules: alignment, cohesion, and separation [Rey87]. By following these rules, the individual boids can create intricate patterns of movement that resemble the behavior of real-life flocks of birds.

Boids have many practical applications, including video games, animation, and scientific simulations. In video games, boids can be used to simulate the movement of crowds or swarms of enemies. In animation, boids can be used to create realistic-looking flocks of birds or schools of fish. In scientific simulations, boids can be used to model the behavior of real-life flocks or swarms, helping researchers better understand the underlying principles of swarm behavior.

Overall, boids are a powerful tool for simulating complex patterns of movement, and their versatility and applicability have made them a popular tool in many different fields.

5.4 Implementation

5.4.1 Flyweight Design Pattern

The Flyweight design pattern is a software design pattern that aims to reduce the memory footprint of an application by sharing objects that have similar state information. It is a structural pattern, which means it focuses on how objects are

composed to form larger structures.

The Flyweight pattern is based on the idea of dividing objects into intrinsic and extrinsic states. Intrinsic state refers to the properties of an object that are shared among multiple instances, while extrinsic state refers to the properties that are specific to each instance. By separating these states, the Flyweight pattern allows multiple instances of an object to share the same intrinsic state, reducing memory usage and improving performance.

The Flyweight pattern is often used in situations where a large number of objects need to be created, but memory usage is a concern. For example, in a text editor, many characters may need to be displayed on the screen at once, but each character has the same intrinsic state (such as font and size), so it makes sense to share this information between characters to reduce memory usage.

This can be achieved by using Unity's Scriptable Object. In a class called BoidSettings, we can store generic information about a boid such as minimum and maximum speed, the distance in which the boid checks for collisions, it's perception radius and so on. Then, we can create the Boid class, which contains an object of type BoidSettings alongside more fields that store data for a specific boid, like it's position or velocity.

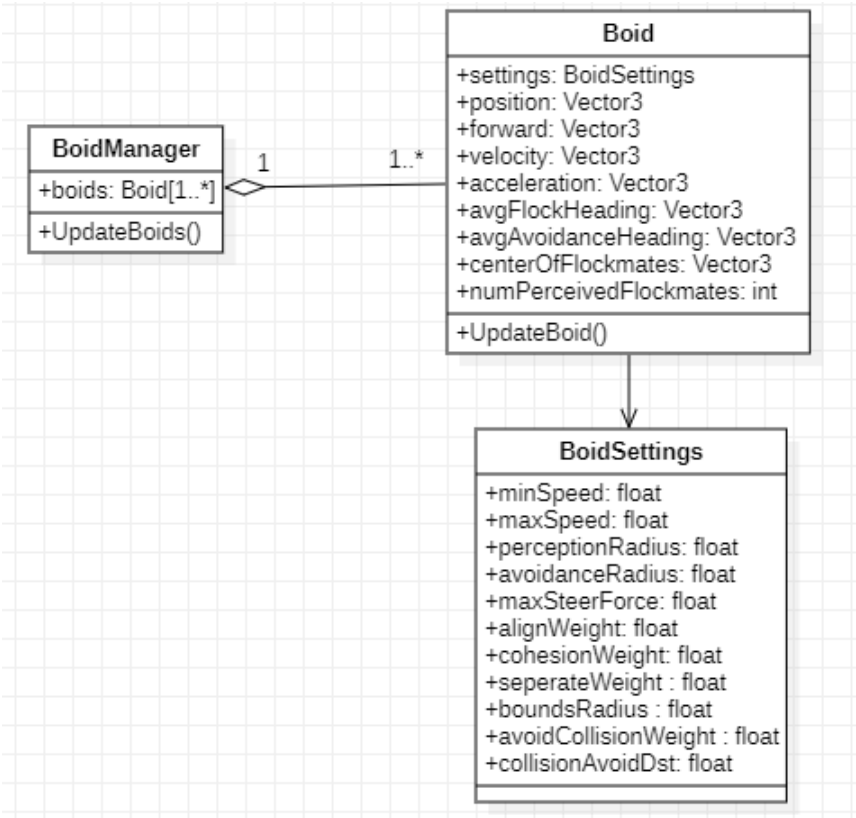


Figure 5.2: Boids using the flyweight pattern

5.4.2 A single boid

Our boid class will store information about the average direction of the flock (alignment rule), the center of the flock (cohesion rule), and the avoidance direction (separation rule), used to avoid other boids from the flock. The class will expose only one public method, UpdateBoid, used to update the velocity and the direction of the boid.

A private method is used to make the boid steer towards a given point.

```
private Vector3 SteerTowards (Vector3 vector)
{
    Vector3 v = vector.normalized * settings.maxSpeed - velocity;
    return Vector3.ClampMagnitude (v, settings.maxSteerForce);
}
```

Now, using this function and the data about each rule, we can determine the forces that will be applied to the boid.

```
public void UpdateBoid()
{
    Vector3 acceleration = Vector3.zero;

    Vector3 offsetToFlockmatesCentre = (centreOfFlockmates -
       position);

    var alignmentForce = SteerTowards (avgFlockHeading) *
       settings.alignWeight;
    var cohesionForce = SteerTowards (offsetToFlockmatesCentre) *
       settings.cohesionWeight;
    var seperationForce = SteerTowards (avgAvoidanceHeading) *
       settings.seperateWeight;

    acceleration += alignmentForce;
    acceleration += cohesionForce;
    acceleration += seperationForce;

    velocity += acceleration * Time.deltaTime;
    float speed = velocity.magnitude;
    Vector3 dir = velocity / speed;
    speed = Mathf.Clamp (speed, settings.minSpeed,
       settings.maxSpeed);
    velocity = dir * speed;
```

```
    position += velocity * Time.deltaTime;
    forward = dir;
}
```

[BM]

Additionally we can make the boid avoid collisions with other obstacles. First, we need to define a method that checks for collisions in a certain distance in the boids path. We do this by casting a sphere in the direction the boid is moving. If the sphere hits something, the function will return true. Else it will return false.

```
bool IsHeadingForCollision ()
{
    RaycastHit hit;
    if (Physics.SphereCast (position, settings.boundsRadius,
       forward, out hit, settings.collisionAvoidDst,
       settings.obstacleMask))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

If the boid is indeed heading for a collision, we need to find a new clear path. In order to do this, we can cast rays at increasing angles from the forward direction, until a clear path is found. For this we need to define a function that equally distributes points on the surface of a sphere [PoS]. We can create a new class called BoidHelper that takes care of this.

```
public static class BoidHelper
{
    const int points = 300;
    public static readonly Vector3[] directions;

    static BoidHelper ()
    {
        directions = new Vector3[BoidHelper.points];

        float goldenRatio = (1 + Mathf.Sqrt (5)) / 2;
```

```
        float angleIncrement = Mathf.PI * 2 * goldenRatio;

        for (int i = 0; i < points; i++)
        {
            float t = (float) i / points;
            float inclination = Mathf.Acos (1 - 2 * t);
            float azimuth = angleIncrement * i;

            float x = Mathf.Sin (inclination) * Mathf.Cos (azimuth);
            float y = Mathf.Sin (inclination) * Mathf.Sin (azimuth);
            float z = Mathf.Cos (inclination);
            directions[i] = new Vector3 (x, y, z);
        }
    }

}
```

Now that we have a function that distributes the points on a sphere, we can cast spheres to each one of them, and return the direction from the center of the sphere (boid) to the first point that does not detect a collision.

5.4.3 Managing multiple boids

Now that we have the logic for a single boid implemented, we can create a new class, BoidManager, that contains a list of boids. This class will be responsible for updating every boid, so each frame it will have to iterate over all the boids two times. The first iteration will calculate the average flock heading, where is the flock center and the avoidance for each boid. The second iteration will update these parameters for every boid and call their Update() function.

5.5 Parametrization

At this moment, our boids have a lot of parameters that need to be set. From the weights of the tree rules defining a boids, we also have a minimum and maximum speed, from how far away does a boid detect a collision, how big is the radius in which a boid detects other boids and so on. It is highly unlikely to get the desired behaviour by setting all there parameters manually. This is an unconstrained optimisation problem so we can solve it using a genetic algorithm.

5.5.1 Genetic algorithms

Genetic algorithms (GAs) are a type of computational optimization technique inspired by the principles of natural selection and genetics. They are used to solve complex optimization problems by mimicking the process of natural evolution.

In a genetic algorithm, a population of potential solutions, represented as a set of individuals or "chromosomes," undergoes a series of iterative operations to evolve towards an optimal solution. Each individual in the population represents a potential solution to the problem, and its fitness is evaluated based on a predefined objective function.

The basic steps of a genetic algorithm include:

1. Initialization: A population of random individuals is created, each representing a potential solution to the problem.

2. Evaluation: The fitness of each individual is determined by evaluating its performance using the objective function.

3. Selection: Individuals with higher fitness have a higher probability of being selected as parents for the next generation. Common selection techniques include tournament selection and roulette wheel selection.

4. Reproduction: The selected individuals are combined through crossover and mutation operators to create offspring. Crossover involves exchanging genetic material between parents, while mutation introduces random changes in the offspring.

5. Replacement: The offspring replace some individuals in the current population, typically those with lower fitness. This ensures the population evolves towards better solutions over time.

6. Termination: The algorithm terminates when a termination criterion is met, such as reaching a maximum number of generations or finding an acceptable solution.

The underlying principle of genetic algorithms is that the fittest individuals in each generation have a higher chance of passing their beneficial traits to the next generation. Over successive generations, the population evolves and improves its fitness, converging towards an optimal or near-optimal solution.

Genetic algorithms have been successfully applied to various optimization problems, such as scheduling, engineering design, financial modeling, and machine learning. They offer advantages such as robustness, ability to handle complex search

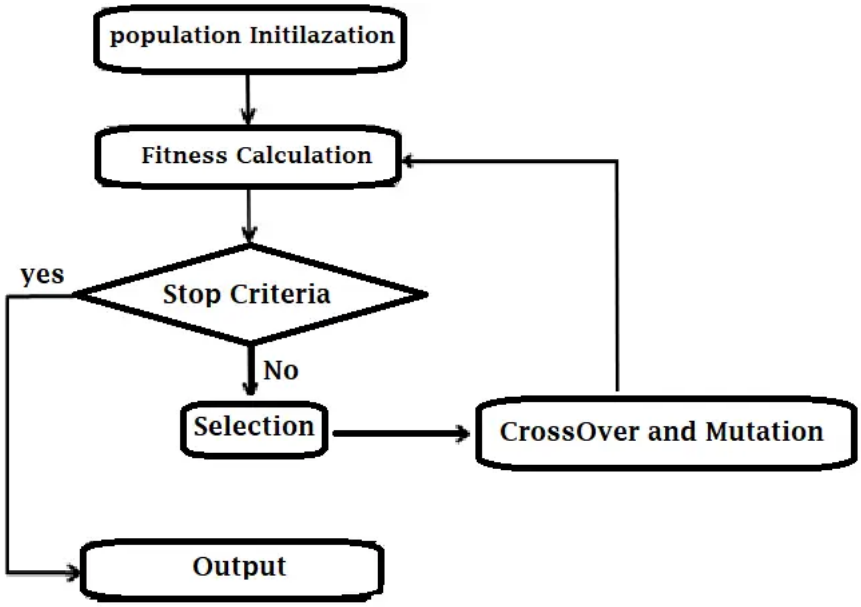


Figure 5.3: Steps of a genetic algorithm [gen]

spaces, and the potential to find global optima. However, they may require careful parameter tuning and can be computationally expensive for large-scale problems.

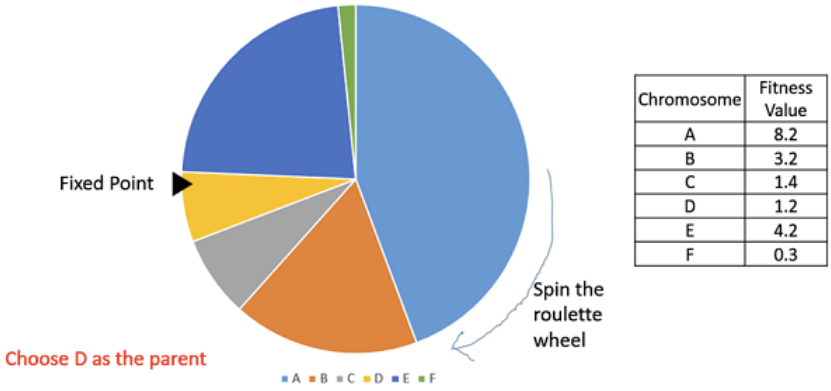
In our case, the chromosomes are represented by the boids, and each parameter of the boid is a gene. for the first generation, each parameter for each boid will be set randomly. We can evaluate the fitness of a boid using the time it spent swimming without hitting any obstacles and the number of boids inside its detection radius.

Once a boid hits an obstacle, we can count it as "dead" and remove it from the scene. Because the boids influence each other's path, it is not a good solution to replace the population once everyone dies, because the last few boids alive will have such little influence over one another that they will only avoid obstacles, so they will never "die". A workaround is to replace a boid right after it dies.

Every time a boid dies, we set it's fitness to 0 so it is not selected for reproduction, and we have to select two other boids as parents for the new offspring. Individuals with higher fitness should have a higher probability of being selected.One of the most common methods of parent selection is fitness proportionate selection. Every person in this has a chance of becoming a parent that is inversely correlated to fitness. Fitter people therefore have a better chance of marrying and passing on their traits to the next generation. As a result, such a selection approach exerts selection pressure on the population's fitter members, improving people over time.

In a roulette wheel selection, the circular wheel is divided into n pies, where n is the number of individuals in the population. Each individual gets a portion of the circle which is proportional to its fitness value. A fixed point is chosen on the wheel circumference as shown and the wheel is rotated. The region of the wheel which

comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Figure 5.4: Roulette Wheel Selection [sel]

After we select the parents, we can combine them in order to create an offspring. Each gene has a 50 percent chance of being selected from each parent, and we can also set, for each gene, a 3 percent chance of a mutation occurring. When a gene is mutated, it's value is randomly increased or decreased by a small amount.

5.5.2 Tracking the evolution

We can create a new scene in order to see how the behaviour of each boid evolves over time. We add a new parameter "fitness" to the boids created above. The boid manager will also be responsible for creating the first population, selecting parents and creating offsprings.

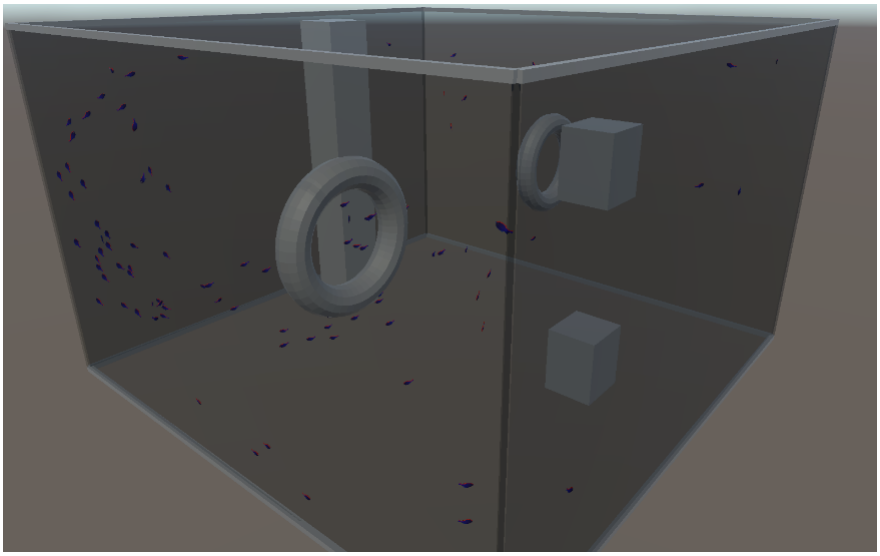


Figure 5.5: Boins training scene

The boid manager can also be responsible for exporting the training data to a csv file. Because we do not have generations, the evolution will be tracked over time, so every 5 seconds, the boid manager will write to the cvs the average for each parameter. Using this data, graphs can be created to get a better overview of how each parameter evolved.

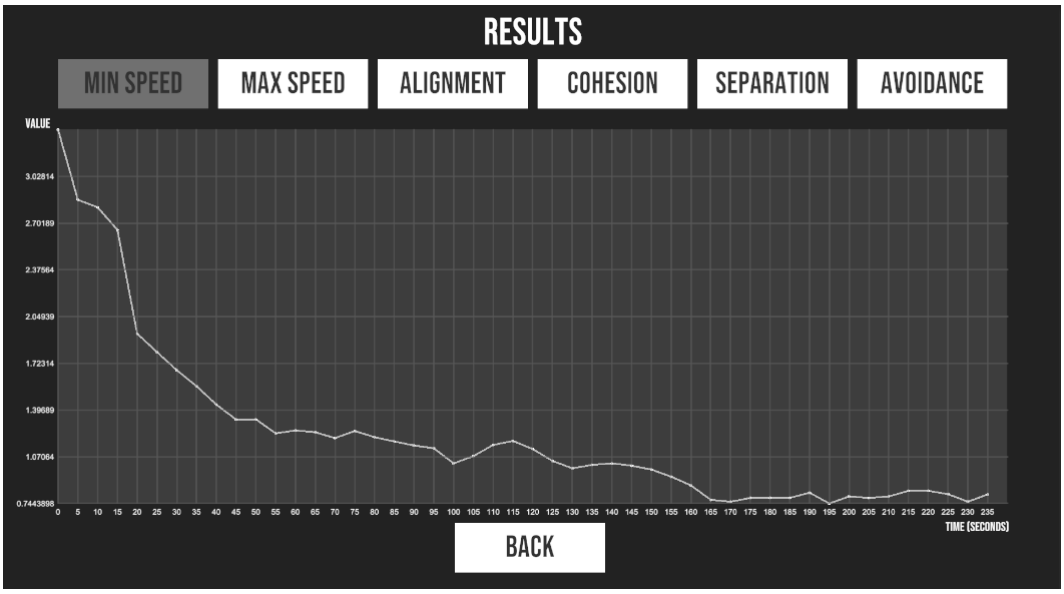


Figure 5.6: Graph from csv file in unity

Chapter 6

Conclusions

This thesis has explored the application of Unity, the marching cubes algorithm, and the boids algorithm to create a realistic and immersive simulation of an endless ocean with swimming fishes. Through the implementation and evaluation of the developed system, several key findings and insights have emerged:

- Realistic Ocean Simulation: The combination of Unity and the marching cubes algorithm proved to be a powerful tool for generating a visually stunning and dynamically changing ocean. The algorithm effectively generated a continuous and endless ocean surface, exhibiting realistic wave patterns, variation in height, and smooth transitions.

- Efficient and Natural Fish Movement: By implementing the boids algorithm, the simulation achieved lifelike fish swimming behavior. The boids algorithm enabled individual fishes to navigate the ocean environment while exhibiting emergent group behaviors, such as schooling and avoiding obstacles. The movement appeared natural and visually appealing, contributing to the overall realism of the simulation.

- Performance Optimization: The thesis also focused on optimizing the performance of the simulation to ensure smooth and responsive interactions. Various techniques, including parallel processing, were employed to maintain a high frame rate and minimize computational overhead. These optimizations allowed for the simulation to run efficiently even with a large number of fishes and complex ocean surface generation.

- User Interaction and Engagement: The developed simulation provided an engaging user experience through intuitive controls and interactive features. Users could navigate the ocean, observe and interact with the fishes, and modify simulation parameters in real-time. The integration of user input enhanced

the sense of immersion and provided a platform for educational or entertainment purposes.

- Limitations and Future Directions: Despite the achievements, there are certain limitations to consider. The complexity of the simulation may impose hardware requirements, and the current implementation may not fully account for advanced physics simulations or complex interactions between fishes and the environment. Future research could focus on integrating more sophisticated behaviors and interactions, expanding the simulation to include additional marine life forms, or incorporating real-time lighting and shading techniques to enhance the visual fidelity

Bibliography

[BM] 3 simple rules of flocking behaviour. https://gamedevelopment.tutsplus.com/tutorials/3-simple-rules-of-flocking-behaviors-alignment-cohesion-and-separation Online; accessed 02 May 2023.

[GCP] Generating complex procedural terrains using the gpu. https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gput. Online; accessed 11 April 2023.

[gen] Gentle introduction to genetic algorithm! https://vidyasheela.com/post/gentle-introduction-to-genetic-algorithm. Online; accessed 15 May 2023.

[HoG] History of video games. https://en.wikipedia.org/wiki/History_of_video_games. Online; accessed 02 May 2023.

[oxo] oxo video game. https://en.wikipedia.org/wiki/OXO_(video_game). Online; accessed 02 May 2023.

[Par82] B. L. Partridge. The structure and function of fish schools. *Scientific American*, pages 114–123, 1982.

[PoS] Generating points on a sphere. https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere/44164075#44164075. Online; accessed 02 May 2023.

[Rey87] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21:25–34, 1987.

[Sch83] V. B. Scheffer. Spires of form: Glimpses of evolution. page 64, 1983.

[sel] Genetic algorithms - parent selection. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm. Online; accessed 18 May 2023.

[Sha75] E. Shaw. Fish in schools. *Natural History 84*, (8):4046, 1975.

[TfT] Tennis for two. `https://en.wikipedia.org/wiki/Tennis_for_Two`. Online; accessed 02 May 2023.

[WEL87] Harvey E. Cline William E. Lorensen. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphic*, 21(4):164–165, 1987.