BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ROGUELITE 3D GAME IN UNREAL ENGINE

ROGUELITE 3D HRA V UNREAL ENGINE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PAVEL LUKL**
AUTOR PRÁCE

SUPERVISOR **Ing. MICHAL VLNAS**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

162635

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Lukl Pavel**
Programme:	Information Technology
Title:	**Roguelite 3D game in Unreal Engine**
Category:	Computer Graphics
Academic year:	2024/25

Assignment:

1. Study methods of game development in Unreal Engine and methods of procedural generation.
2. Design a 3D roguelite game with a marine theme.
3. Implement the game in Unreal Engine.
4. Make user evaluation and summarize the results.
5. Create a demonstration video.

Literature:

- Gregory, Jason. *Game engine architecture*. crc Press, 2018. ISBN 1351974289, 9781351974288
- Adams, Ernest, and Joris Dormans. Game mechanics: advanced game design. New Riders, 2012. ISBN 0321820274, 9780321820273
- Koster, Raph. Theory of fun for game design. O'Reilly Media, Inc., 2013.
- Schell, Jesse. The Art of Game Design: A book of lenses. CRC press, 2008.
- Lee, Joanna. *Learning unreal engine game development*. Packt Publishing Ltd, 2016.

Requirements for the semestral defence:
Point 1, 2 and a prototype of point 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Vlnas Michal, Ing.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	14.5.2025
Approval date:	31.1.2025

Abstract

This thesis investigates the design and implementation of a 3D roguelite video game built in Unreal Engine. It addresses procedural world generation by developing a Perlin-noise terrain generator with edge falloff to create seamless island archipelagos. A wind-aware navigation model was implemented to influence ship movement, alongside a Behavior Tree-driven AI framework for dynamic naval combat. A modular upgrade architecture supports both run-based enhancements and persistent progression across sessions. A fully playable prototype was developed and refined through user testing, validating core mechanics and UI elements. The results demonstrate a cohesive integration of procedural generation, AI-driven gameplay, and progression systems. This work contributes practical methodologies and patterns for leveraging Unreal Engine to craft engaging, replayable game experiences.

Abstrakt

Tato práce se zabývá návrhem a implementací 3D roguelite videohry vytvořené v Unreal Engine. Zaměřuje se na procedurální generování světa, přičemž vyvinut byl generátor terénu využívající Perlinův šum s okrajovým falloffem, který vytváří plynulá souostroví. Implementován byl navigační model citlivý na vítr, ovlivňující pohyb lodí, společně s umělou inteligencí řízenou stromem chování pro dynamické námořní bitvy. Modulární systém vylepšení podporuje jak bonusy v rámci jednotlivých spuštění hry, tak trvalý postup napříč herními sezeními. Výsledkem práce je plně hratelný prototyp, který byl otestován uživateli a který ověřil základní herní mechaniky a prvky uživatelského rozhraní. Výsledky ukazují soudržnou integraci procedurálního generování, hraní založeného na umělé inteligenci a systému hráčského postupu. Práce přináší praktické metodiky a vzory pro efektivní využití Unreal Engine k tvorbě poutavých herních zážitků s vysokou znovuhratelností.

Keywords

video game, Unreal Engine, procedural map generation, roguelite genre, Perlin noise, game genres, game development, game engine, artificial intelligence, behavior tree

Klíčová slova

videohra, Unreal Engine, procedurální generování map, roguelite žánr, Perlinův šum, herní žánry, vývoj her, herní engine, umělá inteligence, strom chování

Reference

LUKL, Pavel. *Roguelite 3D game in Unreal Engine.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Michal Vlnas

Rozšířený abstrakt

Videohry představují komplexní multimediální dílo, které propojuje umění, zvuk, příběh i software do interaktivního zážitku. Cílem této práce je ukázat, jak lze v herním enginu Unreal Engine vytvořit 3D roguelite hru zasazenou do procedurálně generovaného námořního světa. Hráč v ní přebírá velení plachetnice, pluje mezi ostrovy, bojuje s nepřátelskými loděmi a v přístavech investuje získané zdroje do vylepšení svého plavidla. Hra kombinuje tradiční prvky roguelite žánru – jako náhodně generované úrovně – s trvalou progresí, která motivuje k opakovanému hraní.

Pro tvorbu herního světa je využito procedurální generování, jež automaticky vytváří variabilní ostrovní řetězce. Díky adaptivnímu využití Perlinova šumu a uhlazujícímu přechodu k moři získává každý ostrov organický tvar a realistickou topografii. Tím je zajištěno, že je při každém spuštění hry prostředí odlišné a nabízí hráči nové výzvy bez nutnosti ručního modelování. Navíc je dynamicky určováno rozmístění klíčových bodů zájmu, jako jsou přístavy a nepřátelské lodě, což podporuje průzkum a strategické rozhodování.

Pohyb lodí využívá simulaci větru, která ovlivňuje rychlost i ovladatelnost plavidla. Hráč musí přizpůsobit úhel plachet a kurz tak, aby využil vítr ku prospěchu nebo se mu vyhnul, pokud hrozí zpomalení. Tento prvek přidává hře strategickou hloubku a nutí k neustálému vyvažování rychlosti a manévrovatelnosti. Námořní souboje jsou založeny na principu broadside – hráč i protivníci se snaží natočit bok lodi ke straně soupeře a vypálit salvou kanónů. Umělá inteligence protivníků využívá stromové chování pro volbu mezi pronásledováním, ústupem a útokem, což zajišťuje plynulé a variabilní souboje, které se dynamicky mění podle aktuální situace na moři.

Progresní systém dává hráči na výběr mezi krátkodobými bonusy, které platí jen pro jednu hru, a trvalými vylepšeními, jež přetrvávají i po zničení lodi. Tento modulární přístup kombinuje krátkodobé bonusy s dlouhodobou motivací. Hráči tak mohou investovat v jedné hře do rychlosti otáčení nebo poškození děl, aby zvládli těžší souboje, ale zároveň se mohou rozhodnout pro permanentní zesílení trupu či plachet, které zlepší startovní podmínky v dalším běhu. Správné vyvážení mezi těmito dvěma typy investic je klíčové pro dlouhodobé udržení zájmu a pocit postupného pokroku.

Pro ověření herních mechanismů byl během vývoje k dispozici plně hratelný prototyp, který prošel několika koly uživatelského testování. Testeři hodnotili zejména plynulost ovládání, srozumitelnost uživatelského rozhraní a vyváženost obtížnosti. Na základě jejich zpětné vazby byly upraveny rychlost plavby proti větru, chování kamery při ostrých manévrech a jasnost indikátorů chladnutí děl. Díky tomuto cyklu
návrh–implementace–testování se podařilo dosáhnout dobré hratelnosti a pozitivního herního pocitu, který podporuje návaznost mezi jednotlivými běhy.

Výsledkem této práce je metodologicky ucelený přístup k tvorbě roguelite herních světů v Unreal Engine, založený na kombinaci procedurálního generování, interaktivní simulace pohybu a adaptivní AI. Navržené postupy lze dále rozšiřovat, čímž se otevírá prostor pro tvorbu stále bohatších a variabilnějších herních zážitků.

Roguelite 3D game in Unreal Engine

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Michal Vlnas. I have listed all the literary sources, publications, and other sources that were used during the preparation of this thesis.

.....
Pavel Lukl
May 12, 2025

Acknowledgements

I would like to thank my supervisor, Ing. Michal Vlnas, for his comprehensive feedback on each chapter, helpful advice, and unconditional positive attitude throughout the thesis. I also thank my family and friends for their support and especially Kryštof Kolář for helping me with some of the game's 3D models.

Contents

Chapter 1

Introduction

Video games have become a universal form of entertainment, inviting players of all ages to explore imaginary worlds, solve challenges, and share experiences. At their heart lies the idea of an interactive story, where each action taken by the player can change the outcome. This combination of creativity and play has made video games one of the most popular pastimes of our time, accessible to anyone with a screen and a controller or keyboard.

Game development is a multidisciplinary field that brings together art, sound, narrative, and software engineering. Modern tools, called game engines, provide a framework that handles core tasks such as graphics rendering, physics simulation, and input handling. On top of this framework, designers and programmers collaborate to craft the rules, visuals, and audio that give each game its unique character. Procedural content generation adds another layer, allowing vast landscapes or levels to be created automatically by algorithms rather than by hand.

This thesis presents the design and creation of a 3D roguelite game set in a sea-faring world. Players command a sailing ship, explore islands whose shapes and features are generated at runtime, engage in naval combat, and visit seaports to repair and upgrade their vessel. The game runs in Unreal Engine, chosen for its powerful visual tools and extensibility, and uses procedural techniques to ensure that each playthrough feels fresh and unpredictable.

The main goals of this work were to learn key features of Unreal Engine, apply procedural generation methods, design a compelling roguelite experience, implement a working prototype, and evaluate it through user testing. Each goal was met: a fully playable game was built, feedback from test sessions led to refinements, and a demonstration video documents the result.

The contents of this document are organized as follows. Chapter 2 reviews fundamental concepts in video game development, roguelite design, and procedural content generation. Chapter 3 describes the game concept, core mechanics, and user interface. Chapter 4 details the technical implementation in Unreal Engine, including world generation and AI systems, including user testing methodology and results. Chapter 5 reflects on achievements and outlines directions for future work.

Chapter 2

Video Games

Video games are a significant cultural, technological, and economic force that have evolved from simple recreational activities into immersive and complex experiences enjoyed by millions of people worldwide [8]. A video game can be broadly defined as an interactive digital medium in which players engage with virtual environments, characters, or scenarios through various input devices. Simply said, a video game is a game that is played on an electronic device. Video games encompass a wide range of genres – such as action, adventure, simulation, or strategy – and offer diverse experiences catering to different types of players.

Since the late 20th century, the video game industry has grown rapidly, driven by advancements in technology that have enabled the creation of more sophisticated gameplay mechanics, realistic visuals, and engaging narratives. This development is supported by the integration of multiple disciplines, including computer science, graphic design, audio engineering, and storytelling, making game development a multidisciplinary field. Beyond entertainment, video games hold cultural significance, serving as tools for education, social interaction, and artistic expression, further emphasizing their versatility [14].

What sets video games apart from other forms of entertainment is their interactivity. Instead of passively observing, players actively participate, influencing outcomes and creating unique experiences within the game world. This interactive element is central to what makes video games such a powerful medium [7].

This thesis focuses on the development of a 3D roguelite video game with an emphasis on procedural generation, dynamic gameplay systems, and the use of Unreal Engine. To provide context for this project, it is essential to examine the video game development process, the capabilities and selection of game engines, the roguelite genre, and the role of procedural generation in creating engaging procedural content.

2.1 Development Process

The development process of a video game is the process of making a video game from the initial concept to a finished product. It is different from the development process of general software because it has to include the development of other disciplines such as art, music, and story [12, 2].

While the initial concept of a game is usually quite simple and only focuses on the main aspects of the game, building the concept and its necessary supporting systems is a lot of work, usually requiring many people with a variety of skills and/or a large amount of time. The scope of the entire project becomes so large that a need for division arises [15, 1].

Developers often follow the basic development pipeline consisting of three distinct phases: pre-production, production, and post-production [1, 2].

Pre-production

Pre-production is the planning phase. The initial concept is defined and refined. During the pre-production phase, details like the game story, the genre, the setting, the environment, the intended audience, and the platforms for the game are clarified. In this phase, it is easy to adjust as a very small amount of the total work is done. The main reason pre-production exists is to eliminate as much guesswork as possible in later stages of development. A concept artist usually creates concept art based on the initial pitch that helps clearly visualize and communicate the category of the game to be built [1].

The outcome of the pre-preduction phase should be a clearly defined plan that makes it easier to decide whether to continue with the project based on market factors, scope, available resources, etc. [1]

Production

Production is the most extensive and resource-intensive phase of game development, during which the majority of the game's content is created and implemented. This phase involves the collaborative efforts of multiple disciplines working towards a common goal. Game designers continue to specify the details of the game systems while communicating with the programmers who work on implementing these systems [12]. When some of these systems get implemented, they are tested, and feedback is provided. The game designers and programmers work in tandem to address the feedback and make further changes. This continues in a cycle where each time the product becomes more refined and complete [1, 15].

The production phase requires effective communication and coordination among team members to maintain consistency. It is also the stage where unforeseen challenges, such as technical constraints or scope changes, must be addressed to keep the project on track [1, 2].

Post-production

The post-production phase in a video game pipeline is crucial for polishing and finalizing the game before release. After the development team has completed the core mechanics, levels, and art assets, post-production focuses on refining these elements to ensure a smooth, enjoyable player experience. This phase includes testing, bug fixing, optimization, and finalizing audio and visual effects [1].

Quality assurance (QA) testers rigorously test the game for bugs, glitches, and performance issues, while developers work to resolve these problems. Optimization is also a key part of post-production, ensuring the game runs smoothly on various hardware platforms and maintaining a steady frame rate [2].

Additionally, the audio team adds final sound effects, music, and voice-overs, enhancing the game's atmosphere and ensuring consistency throughout. Visual polish is applied to animations, lighting, and textures, ensuring the graphics meet the highest standards for the target platform [12, 2].

Finally, post-production also includes creating marketing materials such as trailers and screenshots, preparing the game for distribution, and implementing any final touches based on player feedback from player tests conducted before the official release of the game [12].

After the release of the game, there are usually bugs found by the players that are addressed through updates. Depending on the monetization model of the game and the overall plan, free or paid updates, which include new content for the game, can also be developed [12].

2.2 Game Engines

The phrase "game engine" [8] first emerged in the mid-1990s, particularly associated with first-person shooter (FPS) games such as *Doom*, developed by id Software. In creating *Doom*, id Software clearly separated core programming elements – including the 3D rendering engine, audio management, and collision detection – from the specific game content, such as artwork, environments, and gameplay rules, which directly shaped the player's experience [8, 19].

This division was especially advantageous because it allowed fans and independent development teams to reuse the foundational software while adding their own creative assets, environments, and gameplay mechanics to modify existing games or develop entirely new experiences. Recognizing this trend, original game developers soon began providing official, accessible, and free-to-use development tools specifically designed for the community creating modifications, known as the "mod community". Over time, games started being intentionally developed with this reuse in mind, further encouraging official or third-party modifications. The term game engine, therefore, refers to the extensible, core part of a complete game software package that can be reused in order to create these modifications or entirely new games. Modern game engines are also bundled with a game development environment for ease of use [8].

Certain game engines focus on specific platforms, genres of games, or even specific features of those games, while other engines are made to be suitable for a wide variety of platforms or to develop many kinds of games. Game engines make necessary trade-offs between the types of games that can be developed in the Engine, the target hardware, ease of use, and optimization of the Engine. This outcome is very natural because, for example, smaller game engines targeting a single hardware platform can be very well optimized, while larger game engines targeting multiple platforms will have a harder time with optimization [19].

As already mentioned, there is a great number of existing game engines, so here are some of the commonly used, freely available options and their features.

Unity

Unity engine [1] is a freely available game engine targeting many platforms such as mobile (Apple iOS, Google Android), consoles (Xbox, PlayStation), handheld platforms (Nintendo Switch, PlayStation Vita), desktop computers (Microsoft Windows, Apple Macintosh, Linux) and many others.

Unity mainly focuses on cross-platform deployment and ease of game development. Unity does not place its focus on any specific game genre, therefore remaining usable for many kinds of games. Like many other modern game engines, Unity provides an easy to use, extensive game development environment, as can be seen in Fig. 2.1, which enables the developer to launch their game while developing in order to preview the game on a specific

[1] More information at: https://unity.com/products/unity-engine.
[2] Image taken from: https://www.megavoxels.com/learn/what-is-the-unity-game-engine/.

Figure 2.1: A showcase of the software development kit for Unity engine.[2]

platform (hardware simulation) while providing analysis, optimization and debugging tools. Unity supports scripting mainly in JavaScript and C#.

Unity has been used to create a wide variety of popular games for many platforms, such as *Hollow Knight* by Team Cherry, *Hearthstone* by Blizzard Entertainment, and *Beat Saber* by Beat Games. Because the Engine is free, it is very popular among small teams and solo developers.

Unreal Engine

Unreal Engine [3], developed by Epic Games, is a widely used, free game engine that supports the creation of high-quality games across various platforms, including PC, consoles (such as Xbox and PlayStation), mobile devices (iOS and Android), and virtual reality systems.

Unreal Engine offers a comprehensive suite of tools designed to streamline game development, see Fig. 2.2. Its Blueprint visual scripting system allows developers to create game logic without writing code, making it accessible to designers and artists. The Engine's rendering capabilities support high-quality visuals, enabling the creation of realistic environments and characters. Additionally, Unreal Engine's physics and animation systems provide lifelike movement and interactions within game worlds. The Engine also supports a wide range of platforms, allowing developers to deploy their games across multiple devices efficiently.

Unreal Engine has been utilized in the development of numerous high-profile games, such as *Fortnite* by Epic Games, *Gears of War* by The Coalition, and *Final Fantasy VII Remake*

[3]More information at: https://www.unrealengine.com/.

[4]Image taken from: https://dev.epicgames.com/community/learning/tutorials/PnXj/unreal-engine-unreal-editor-interface-overview.

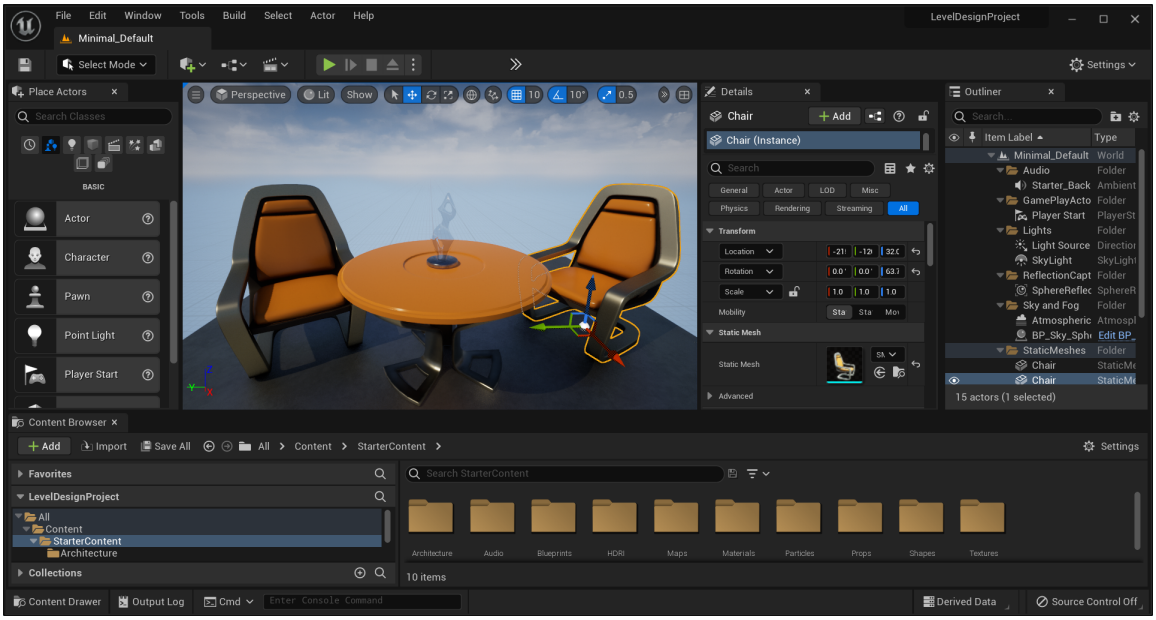


Figure 2.2: A showcase of the software development kit for Unreal Engine.[4]

by Square Enix. Its versatility and robust feature set make it a popular choice among both indie developers and large studios aiming to create immersive gaming experiences.

Godot Engine

Godot Engine [5] is a free and open-source game engine that provides a comprehensive set of tools for game development, as can be seen in Fig. 2.3, supporting 2D and recently even 3D projects.

Godot's architecture is built around a scene system, where each scene is a collection of nodes organized hierarchically. This structure promotes modularity and reusability, allowing developers to create complex game elements by combining simpler components. The Engine supports multiple scripting languages, including GDScript, C#, and visual scripting, catering to different developer preferences. Godot also offers a dedicated 2D engine that operates independently of the 3D pipeline, ensuring efficient performance and a true 2D experience. Its built-in physics engine supports both 2D and 3D physics, making it easy to add realistic physics to games and other interactive applications. Additionally, Godot's open-source nature under the MIT license allows developers to modify and extend the Engine's capabilities to suit their project's specific needs.

Godot has been used to create a variety of games across different genres. Notable titles include *Sonic Colors: Ultimate* by Sonic Team and Blind Squirrel Games, *The Case of the Golden Idol* by Color Gray Games, and *Dome Keeper* by Bippinbits. Its lightweight footprint and versatility make it popular among indie developers and small studios aiming to create 2D and sometimes even 3D games.

[5]More information at: https://godotengine.org/.
[6]Image taken from: https://github.com/godotengine/godot/blob/master/README.md.

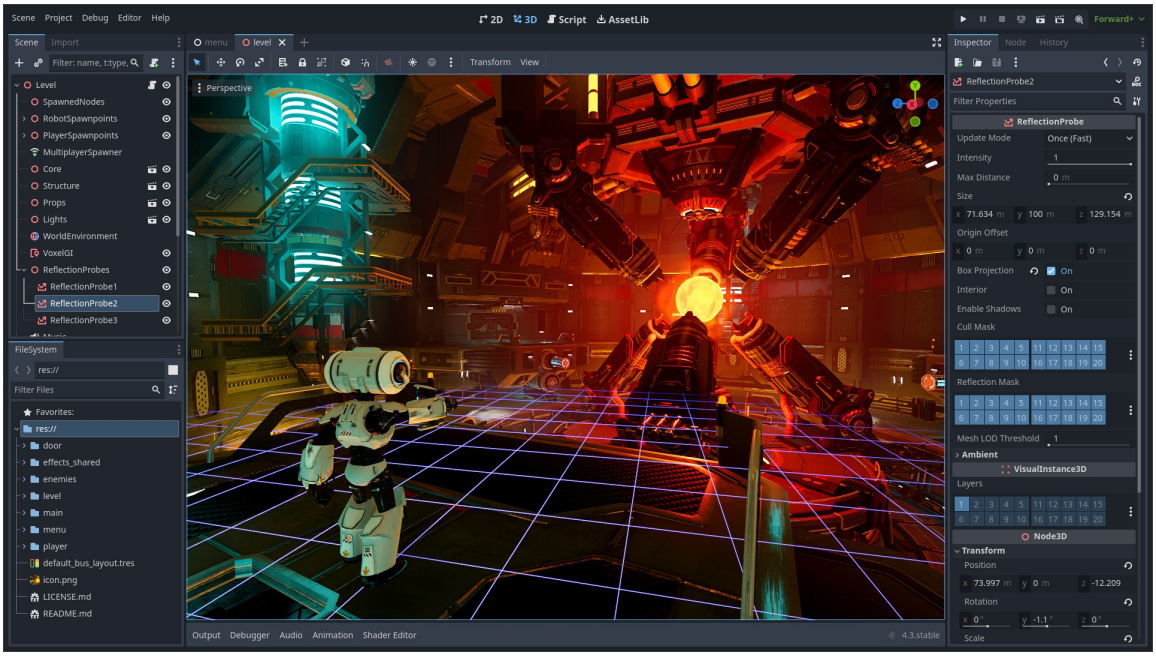


Figure 2.3: A showcase of the software development kit for Godot Engine.[6]

2.3 Roguelite Genre

The term "roguelite" [4] refers to a subgenre of video games that incorporate some aspects from traditional roguelikes but with modifications to make them more accessible to a broader audience. While roguelikes are characterized by features such as procedural generation, turn-based gameplay, grid-based movement, and permadeath, roguelites retain some of these aspects while introducing others that differentiate them from their predecessors.

Definition and Characteristics

Roguelites typically feature procedurally generated levels and permadeath, ensuring that each playthrough offers a unique experience and that players must start over upon character death. However, unlike traditional roguelikes, roguelites often include persistent progression systems, allowing players to retain certain benefits or unlocks across multiple runs. This design choice provides a sense of continual advancement, even in repeated failures. Additionally, roguelites may deviate from the turn-based, grid-based mechanics of classic roguelikes, opting instead for real-time action and more dynamic gameplay [4, 6].

Several games have exemplified the roguelite genre, achieving both critical acclaim and commercial success:

- *The Binding of Isaac*: Combining dungeon crawling with procedurally generated environments and a vast array of items, this game offers deep replayability.
- *Dead Cells*: Merging Metroidvania exploration with roguelike elements, players navigate a constantly changing castle, battling foes in fluid combat sequences.
- *Hades*: An action-packed journey through the underworld, where players control Zagreus, the son of Hades, attempting to escape while uncovering a rich narrative that unfolds over multiple runs.

- *Slay the Spire*: A deck-building game that challenges players to ascend a spire filled with enemies and treasures, offering new card combinations and strategies each run.

Evolution and Popularity

The roguelite genre has grown significantly, particularly in the indie gaming scene. By blending challenging gameplay with progression elements, roguelites have attracted players seeking difficulty and a sense of achievement. The genre's flexibility allows developers to experiment by integrating roguelike mechanics into various game styles, leading to innovative hybrids that expand the genre's appeal [6].

In summary, roguelites offer a compelling mix of challenge and progression, drawing from traditional roguelike elements while introducing features that cater to modern gaming preferences. This balance has solidified their place in the gaming industry, providing players with punishing and rewarding experiences.

2.4 Procedural Generation

Procedural generation [18] refers to the automated algorithmic creation of content through predefined mathematical, logical, or rule-based procedures, instead of manually designing each element by hand. The concept originated primarily in response to the escalating complexity, costs, and resource demands of manually creating detailed, expansive game environments. The exponential growth of the video game industry, characterized by increasingly large-scale worlds, diverse characters, intricate storylines, and sophisticated environments, has rendered manual content creation processes unsustainable in terms of time and cost [10].

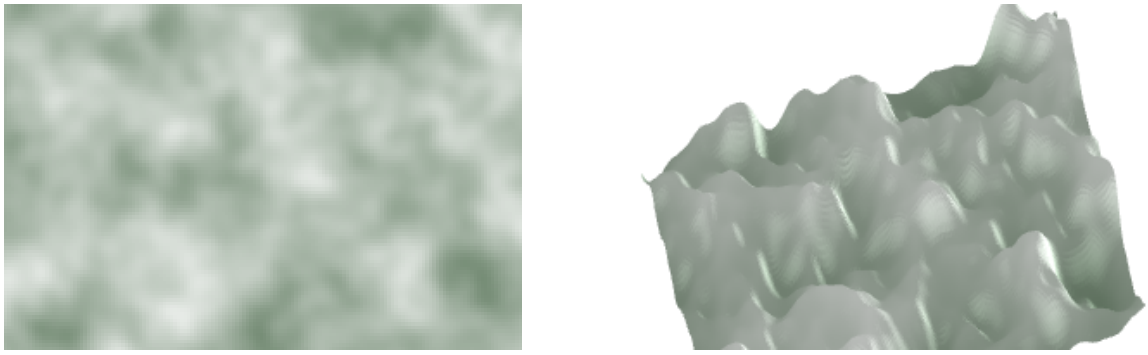
The primary motivation behind procedural generation in video games is twofold. Firstly, it efficiently creates vast, diverse, and immersive environments without proportionally increasing the manual labor involved. Secondly, procedural methods support high variability and replayability, fundamental qualities in numerous game genres such as roguelikes, sandbox games, and open-world exploration titles [18].

Procedural generation spans various levels of complexity and abstraction. It ranges from generating simple textures and assets, such as terrain, vegetation, and items, to more sophisticated processes such as creating complex ecosystems, entire landscapes, urban environments, and even dynamically generated narratives and quests. Despite the variety, all procedural methods aim to maintain a delicate balance between randomness, which ensures diversity, and structured rules, which preserve coherence and contextually appropriate outcomes [10].

2.4.1 Perlin Noise

Perlin noise is a procedural generation algorithm developed by Ken Perlin to create natural-looking textures, terrains, see Fig. 2.4, and variations in computer graphics and video games. It is particularly valued for its ability to produce smooth, continuous patterns that mimic organic phenomena like clouds, landscapes, and textures [10].

At its core, Perlin noise generates a smooth, pseudo-random pattern by calculating gradients at discrete grid points and interpolating values between these points. The process begins by defining a lattice (or grid) of points within the desired space. Each lattice point is assigned a pseudo-random gradient vector generated using a permutation table –



(a) Perlin noise visualization. (b) Generated terrain using Perlin noise.

Figure 2.4: Perlin noise algorithm used in procedural terrain generation.[7]

a predefined table of random numbers from 0 to 255, which ensures reproducibility and consistency in procedural content generation [18].

To illustrate, consider a simplified example in two-dimensional space. Suppose we have a square grid with corners at points $(0,0), (1,0), (0,1)$, and $(1,1)$. Using the permutation table method, we assign gradient vectors at lattice points in Eq. 2.1.

$$\begin{aligned} \mathbf{g}_0 &= (0.6, 0.8) \\ \mathbf{g}_1 &= (-0.8, 0.6) \\ \mathbf{g}_2 &= (0.9, -0.4) \\ \mathbf{g}_3 &= (-0.7, -0.7) \end{aligned} \tag{2.1}$$

Now, we want to compute the Perlin noise value at an arbitrary point, for example $(0.5, 0.5)$. First, we determine the vectors from each lattice point to the evaluation point in Eq. 2.2.

$$\begin{aligned} \mathbf{d}_0 &= (0.5, 0.5) \\ \mathbf{d}_1 &= (-0.5, 0.5) \\ \mathbf{d}_2 &= (0.5, -0.5) \\ \mathbf{d}_3 &= (-0.5, -0.5) \end{aligned} \tag{2.2}$$

Next, we compute the dot product between each gradient vector and its respective distance vector in Eq 2.3.

$$\begin{aligned} \mathbf{g}_0 \cdot \mathbf{d}_0 &= (0.6)\ (0.5) + (0.8)\ (0.5) = 0.3 + 0.4 = 0.7 \\ \mathbf{g}_1 \cdot \mathbf{d}_1 &= (-0.8)\ (-0.5) + (0.6)\ (0.5) = 0.4 + 0.3 = 0.7 \\ \mathbf{g}_2 \cdot \mathbf{d}_2 &= (0.9)\ (0.5) + (-0.4)\ (-0.5) = 0.45 + 0.2 = 0.65 \\ \mathbf{g}_3 \cdot \mathbf{d}_3 &= (-0.7)\ (-0.5) + (-0.7)\ (-0.5) = 0.35 + 0.35 = 0.7 \end{aligned} \tag{2.3}$$

The computed dot products represent how strongly each gradient vector influences the noise value at the evaluation point.

The next critical step involves smoothly interpolating these computed values to generate continuous transitions. Perlin noise employs a smoothing function, commonly the fade function defined in Eq. 2.4.

$$\text{fade}(t) = 6t^5 - 15t^4 + 10t^3 \tag{2.4}$$

This fade function ensures smoothness and continuity, minimizing noticeable artifacts and abrupt transitions in the resulting noise [18].

After interpolation using the fade function and combining the weighted contributions of each lattice corner, the final Perlin noise value for the evaluation point $(0.5, 0.5)$ would be a scalar between -1 and 1, typically normalized to $[0, 1]$ for many applications.

This resulting scalar is crucial in applications such as terrain generation, see example in Fig. 2.4. For heightmaps, each point on a 2D grid is assigned a height value based on its Perlin noise value. These heights are then used to deform a mesh or determine elevation for tiles, producing realistic topographies with hills, valleys, and plains. By modifying frequency and amplitude parameters, developers can control the scale of features: lower frequencies yield larger, smoother landforms, while higher frequencies add finer detail [10]. Beyond heightmaps, Perlin noise is also used for:

- **Texture generation**: to simulate materials like marble or wood.
- **Cloud and smoke effects**: by applying it as input to density functions.
- **Biomes and terrain types**: by thresholding Perlin values to classify regions as water, forest, desert, etc.

Perlin noise's utility extends beyond terrain generation. It is also widely used to create realistic textures such as marble, clouds, smoke, and other natural phenomena. Games like *Minecraft* prominently use Perlin noise for procedural terrain generation, dynamically creating vast, diverse worlds and ensuring uniqueness and exploration appeal in every player experience.

The robustness and computational efficiency of Perlin noise make it a cornerstone algorithm in procedural content generation, significantly influencing numerous applications within game development and computer graphics.

2.4.2 Cellular Automata

Cellular Automata (CA) [11] are discrete computational models that consist of a regular grid of cells, each of which can be in one of a finite number of states. The grid can have any finite number of dimensions. The state of each cell evolves over discrete time steps according to a set of rules based on the states of neighboring cells. This evolution leads to complex, emergent behavior from simple initial conditions [17].

Formal Definition

x A Cellular Automaton is defined by the tuple $\langle G, S, N, f \rangle$, where:

- G is a regular grid of cells.
- S is a finite set of possible states for each cell.
- N defines the neighborhood for each cell, specifying which cells influence a given cell's state.
- $f : S^{|N|} \to S$ is the local transition function that determines the next state of a cell based on the current states of its neighborhood.

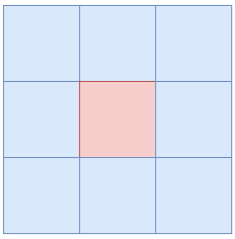
[7] Images taken from https://www.redblobgames.com/maps/terrain-from-noise/

At each time step, the transition function f is applied simultaneously to all cells in the grid, resulting in a new configuration for the entire system [11, 17].

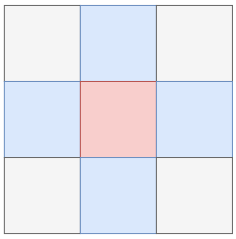
Neighborhood Structures

The neighborhood of a cell comprises the set of cells that influence its state in the next time step. Common neighborhood structures include:

- **Von Neumann Neighborhood** (see Fig. 2.5b): Consists of the four orthogonally adjacent cells (north, south, east, and west).
- **Moore neighborhood** (see Fig. 2.5a): Includes the eight surrounding cells (orthogonal and diagonal neighbors).



(a) The Moore neighborhood.



(b) The Von Neumann neighborhood.

Figure 2.5: A comparison of the commonly used neighborhood definitions.

Conway's Game of Life

One of the most well-known examples of a Cellular Automaton is Conway's Game of Life [11], devised by mathematician John Conway. It operates on a two-dimensional grid where each cell has two possible states: alive or dead. The state of each cell evolves based on the following rules:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors continues to live.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes alive, as if by reproduction.

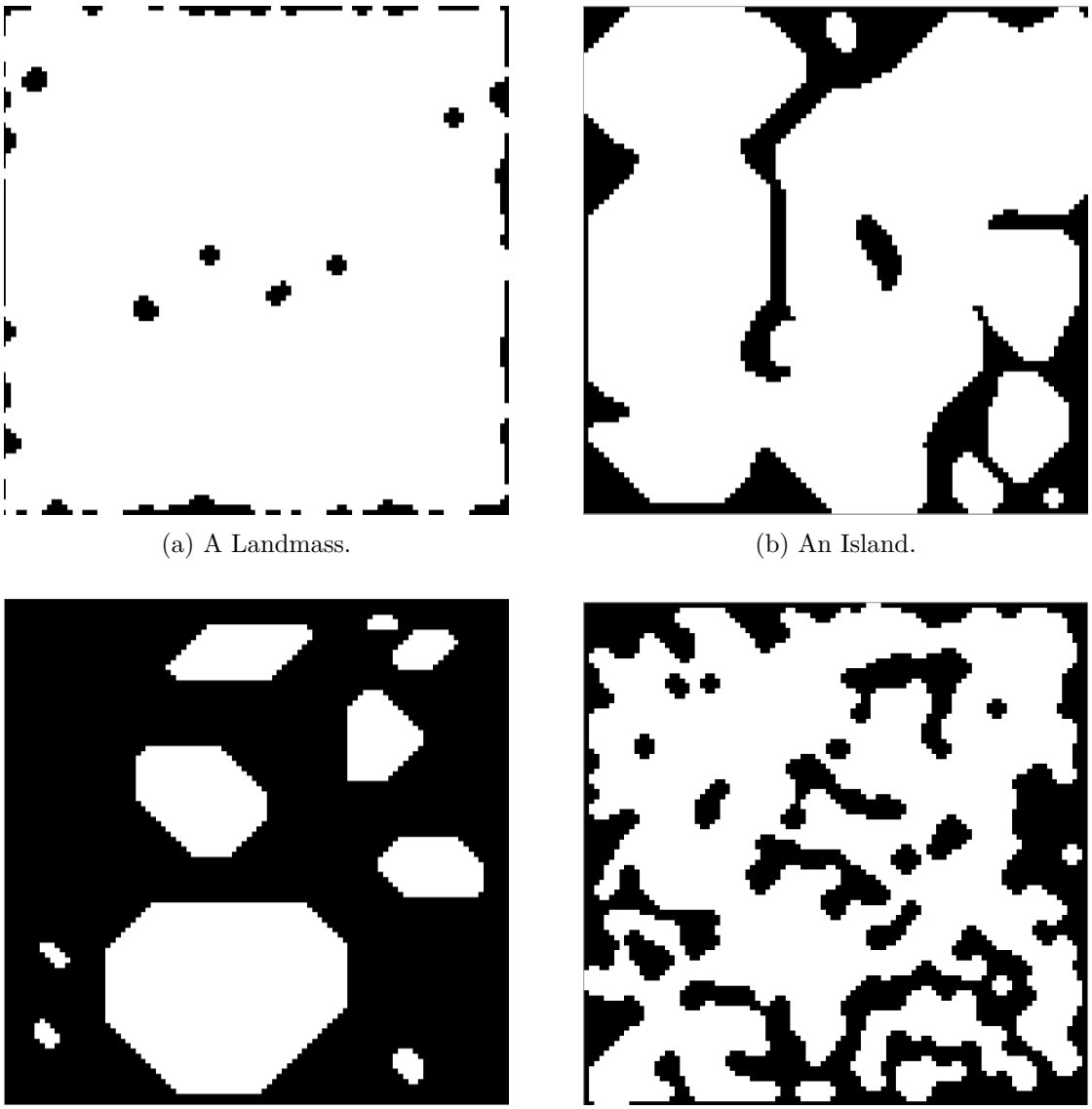
These simple rules can lead to the emergence of complex patterns over successive generations [11].

Applications in Procedural Content Generation

In video game development, Cellular Automata are widely used for procedural content generation. They can create complex and organic-looking structures such as cave systems, dungeon layouts, and terrain features, as can be seen in Fig. 2.6. By defining appropriate

rules and initial conditions, developers can generate diverse and realistic environments that enhance the gameplay experience [18].

By leveraging the principles of Cellular Automata, game developers can efficiently create complex, dynamic, and immersive environments that respond to player interactions and evolve over time.



(a) A Landmass.

(b) An Island.

(c) Multiple islands.

(d) A Cave structure.

Figure 2.6: Examples of 2D terrain and cave generation using Cellular Automata.[8]

2.4.3 Grammar-Based Systems

Grammar-based systems [5] serve as a powerful paradigm in procedural content generation, employing formal grammars to algorithmically produce complex and diverse structures. By

[8]Images taken from https://bronsonzgeb.com/index.php/2022/01/30/procedural-generation-with-cellular-automata/

defining a set of symbols and production rules, these systems enable the recursive generation of intricate designs, such as urban layouts and natural formations, from simple initial parameters. This approach not only enhances the diversity and complexity of generated content but also ensures structural coherence and adaptability, making it particularly valuable in applications like video game design and architectural modeling. For instance, in game development, grammar-based systems can dynamically create levels and environments that are both varied and consistent with the game's aesthetic and functional requirements [18].

Formal Definition

A formal grammar G is defined as a quadruple $G = \langle N, T, P, S \rangle$, where:

- N is a finite set of non-terminal symbols.
- T is a finite set of terminal symbols, disjoint from N.
- P is a finite set of production rules, where each rule maps a non-terminal symbol to a sequence of non-terminal and/or terminal symbols.
- $S \in N$ is the start symbol.

The production rules in P guide the substitution of non-terminal symbols with combinations of terminal and non-terminal symbols, enabling the iterative generation of complex structures [9].

Example: Generating a Simple Building Structure

Consider a simple grammar designed to generate a basic building structure:

- **Non-terminals** (N): {Building, Floor, Room}
- **Terminals** (T): {LivingRoom, Kitchen, Bedroom, Bathroom}
- **Production Rules** (P):
  - Building $\rightarrow$ Floor
  - Floor $\rightarrow$ Room | Room Floor
  - Room $\rightarrow$ LivingRoom | Kitchen | Bedroom | Bathroom
- **Start Symbol** (S): Building

Applying these production rules iteratively can generate various building layouts. For instance:

- Building
- $\Rightarrow$ Floor
- $\Rightarrow$ Room Floor
- $\Rightarrow$ Kitchen Room Floor
- $\Rightarrow$ Kitchen Bedroom Room Floor
- $\Rightarrow$ Kitchen Bedroom Bathroom LivingRoom

This derivation results in a building with a Kitchen, Bedroom, Bathroom, and LivingRoom.

Applications in Procedural Content Generation

Grammar-based systems are widely employed in procedural content generation across various domains [18]:

- **Natural Environments**: Grammars can model the growth patterns of plants, trees, and other natural formations, enabling the creation of realistic ecosystems in virtual environments.
- **Level Design in Games**: Game developers use grammar-based systems to generate intricate and varied game levels, enhancing player engagement through unique experiences.

By leveraging formal grammars, these systems provide a structured yet flexible framework for generating complex and diverse content in procedural design [18].

2.4.4 Comparative Utility of Procedural Generation

Procedural generation techniques offer distinct functional utilities compared to manual content creation methods. They efficiently handle large-scale content creation, significantly reducing the labor-intensive nature of manual methods. Moreover, procedural generation inherently supports high replayability and variability, essential for game genres that thrive on unpredictability and player engagement, such as roguelikes and exploration-based games. Conversely, procedural techniques must be carefully managed to maintain coherent and contextually appropriate results, requiring expert tuning and algorithmic oversight to balance randomness and structure [18].

2.4.5 Usage Examples

Examples of successful procedural generation in video games include:

- *Minecraft* (Mojang Studios): Infinite worlds generated through noise algorithms.
- *No Man's Sky* (Hello Games): Universe-scale planet and ecosystem creation using fractal-based methods, see Fig. 2.7.
- *Spelunky* (Mossmouth): Dynamic, procedural level generation for varied gameplay experiences.
- *FTL: Faster Than Light* (Subset Games): Procedurally generated encounters and narrative-driven events.

2.5 Artificial Intelligence in Video Games

Artificial Intelligence (AI) [13] in video games refers to a set of algorithms and computational techniques that simulate aspects of intelligent behavior such as reasoning, learning, and decision-making. In games, AI primarily governs non-player characters (NPCs), enabling

[9]Images taken from https://extremelyroomypockets.wordpress.com/2016/08/30/its-not-just-no-mans-sky-procedural-generation-doesnt-work/ and https://www.researchgate.net/figure/One-of-the-many-procedurally-generated-planets-in-No-Mans-Sky-9_fig4_334416222

Figure 2.7: Examples of procedural generation in *No Man's Sky*, which generates entire planets complete with flora and fauna.[9]

them to react dynamically to player actions, navigate environments, and exhibit believable behavior. As modern games grow in complexity and scale, AI becomes a crucial tool for enhancing immersion, challenge, and narrative depth [13].

The motivations for using AI in games are manifold. Firstly, AI supports the creation of dynamic and interactive gameplay experiences that cannot be achieved through static scripting alone. Secondly, it allows developers to scale behavior complexity without requiring hand-crafted logic for every possible interaction. Finally, AI enhances player engagement through emergent gameplay, adaptive difficulty, and personalized experiences [3].

2.5.1 Finite State Machines

Finite State Machines (FSMs) [16] are among the most fundamental and widely used AI models in game development. They allow developers to represent behavior as a set of clearly defined states and transitions, forming a simple but effective control mechanism. The model is inspired by formal automata theory and finds practical application in controlling character logic, especially in cases where behavior patterns are well-defined and state-driven [20].

FSMs are especially useful for controlling enemy guards, interactive objects, or quest logic where predictable, testable flows are necessary. They are also easy to visualize and debug, making them accessible to designers as well as programmers [3].

Formal Definition

A Finite State Machine is formally defined by a tuple $\langle S, A, T, s_0 \rangle$ [16], where:

- S is a finite set of possible states.
- A is a finite set of actions, events, or inputs.
- $T : S \times A \to S$ is a transition function that maps a state and an action to a new state.
- $s_0 \in S$ is the initial state.

Each NPC controlled by an FSM occupies exactly one state at a time. Transitions between states are triggered by input events, such as detecting a player or reaching a time threshold. The model is deterministic in its basic form, meaning the same input will always result in the same output [13].

Example: Enemy AI

Consider a basic enemy AI with three states: *Idle*, *Patrol*, and *Attack*. The following transitions apply:

- From *Idle* to *Patrol* after a time delay.
- From *Patrol* to *Attack* if the player is spotted.
- From *Attack* to *Patrol* if the player is lost and a cooldown timer expires.

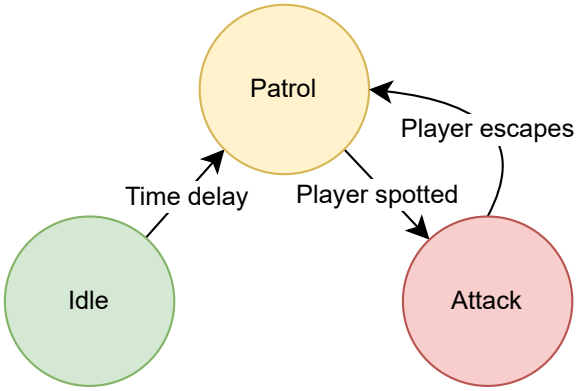


Figure 2.8: Example of a simple finite state machine used to control enemy behavior.

This configuration (Fig. 2.8) provides readable, maintainable behavior logic that can be expanded with more states or transitions. Many classic stealth and action games use FSMs to handle enemy alert levels and scripted sequences [3].

Applications and Utility

Finite State Machines are widely used in scenarios where behaviors follow a predictable, rule-based flow. Their clarity makes them ideal for early prototyping and for systems that must be deterministic or easily auditable [13].

- **Advantages**: Easy to implement, debug, and extend; clear visual representation.
- **Use cases**: Puzzle elements, tutorial scripts, simple enemy AI.

Examples of FSM usage in video games include:

- *Metal Gear Solid* (Konami): Uses FSMs to manage enemy alert levels, transitioning between patrol, caution, and alert states.
- *DOOM* (id Software): Early AI for enemies was structured around state transitions like idle, search, and attack, with simple triggers based on visibility and damage.
- *Half-Life* (Valve): NPCs such as security guards and aliens follow FSM-driven patterns for patrolling, engaging the player, or seeking cover.
- *Pac-Man* (Namco): Ghost behaviors are governed by state machines with chase, scatter, and frightened states based on player interactions and timers.
- *The Legend of Zelda: Ocarina of Time* (Nintendo): Many interactive objects and boss behaviors are implemented with FSMs, allowing for state-driven event progression.

2.5.2 Behavior Trees

Behavior Trees (BTs) [13, 3] are a widely used AI architecture in modern video games. They are designed to address the scalability and flexibility challenges often encountered when using Finite State Machines in increasingly complex games. Behavior Trees structure character logic as a hierarchical tree of tasks and decisions, allowing game developers to implement more nuanced and context-aware behaviors.

Unlike FSMs, which can become difficult to manage as the number of states and transitions grows, BTs are modular and more maintainable. Their structure encourages the reuse of components and supports layered logic that can dynamically respond to changing gameplay conditions. According to Yannakakis and Togelius [13], one of the most significant distinctions is that FSMs hard-code control logic into transitions between states, whereas BTs abstract this logic into composable nodes, decoupling control from individual behaviors. This makes BTs inherently more scalable, better suited for collaborative agent behaviors, and more reusable across multiple characters or scenarios.

Architecture and Logic

A Behavior Tree is composed of nodes that determine the flow and execution of AI decisions. These nodes are arranged in a tree-like hierarchy:

- **Selector (?)**: Evaluates child nodes in order until one succeeds; returns success if any child succeeds.
- **Sequence (→)**: Evaluates all child nodes in order; fails if any child fails.
- **Decorator**: Modifies the behavior or result of a single child node (e.g., invert success/failure).
- **Leaf Nodes**: The actionable elements, such as animations, movement commands, or decision checks.

The flow starts at the root node and propagates downward, with composite nodes (e.g., selectors and sequences) managing the execution logic of their children. This structure is particularly effective for modeling reactive behavior that must prioritize certain actions (e.g., evading danger) over others (e.g., patrolling).

Example: Dynamic Patrol and Combat AI

Fig. 2.9 illustrates a behavior tree used by an autonomous agent such as a robot or NPC. The root node is a repeater that continually reevaluates behavior. The first decision is handled by a selector node that chooses between combat and wandering.

If a target (like a droid) is in range, a sequence checks whether it is weaker. If so, the agent proceeds to attack. If not, the tree branches into another selector where it chooses either to attack immediately or to first find a safer position and move there before attacking.

If no droid is in range, the fallback behavior is a wander sequence that simply instructs the agent to move to a random location.

This structure highlights how Behavior Trees allow conditional logic and action prioritization without exploding into a mass of states and transitions, as would likely occur in a comparable FSM implementation.

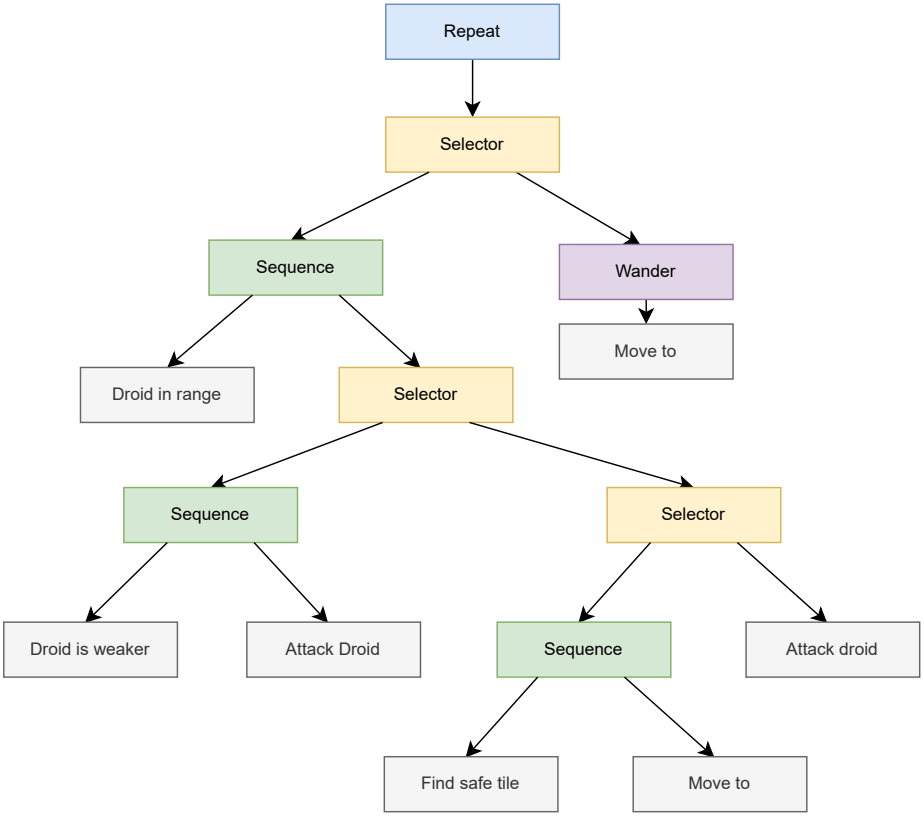


Figure 2.9: A Behavior Tree showing a repeated patrol and combat routine with dynamic prioritization.

Applications and Utility

Behavior Trees are widely used in modern game development due to their flexibility, modularity, and readability [20]. They support complex, layered decision-making and are suitable for both simple agents and highly advanced NPCs.

- **Advantages**: Scalable, reusable components; clear structure for prioritizing tasks; easier maintenance than FSMs.
- **Use cases**: Enemy squads, boss logic, companion characters, dynamic story agents.

Examples of Behavior Tree usage in video games include:

- *The Last of Us* (Naughty Dog): Enemy AI uses BTs to determine cover usage, flanking maneuvers, and coordinated group behavior.
- *Halo* (Bungie): Enemy squads use BTs for hierarchical combat strategies, adapting their actions based on player aggression and battlefield position.
- *Far Cry* (Ubisoft): Enemies use BTs to switch between investigating noise, returning to patrol, or attacking, depending on context.
- *Killzone* (Guerrilla Games): Complex combat AI is structured using BTs to manage awareness states, cover seeking, and aggressive flanking.

- *Shadow of Mordor* (Monolith Productions): BTs manage the layered logic behind the dynamic Nemesis system.

2.5.3 Pathfinding Algorithms

Pathfinding algorithms are a core component of video game AI [3, 20, 13], enabling non-player characters (NPCs) to navigate virtual environments intelligently. These algorithms determine optimal or near-optimal routes between two points in a game world, accounting for obstacles, terrain, and movement costs. Without effective pathfinding, AI agents would appear unrealistic or become stuck, breaking immersion and gameplay [13].

Among the many pathfinding techniques, the A* (A-star) algorithm is the most widely used due to its balance of efficiency and optimality. It provides a flexible framework that can be adapted to different types of grids, movement styles, and gameplay mechanics [13].

The A* Pathfinding Algorithm: Definition and Mechanics

A* is a best-first search algorithm that evaluates nodes using a cost function [20], which can be seen in Eq. 2.5.

$$f(n) = g(n) + h(n) \tag{2.5}$$

- $g(n)$ is the exact cost to reach node n from the starting point.
- $h(n)$ is a heuristic estimate of the cost to reach the goal from node n.

The heuristic $h(n)$ is crucial to the algorithm's performance and is often based on the Manhattan distance (for grid-based maps) or Euclidean distance (for continuous space). A* maintains an open list of nodes to explore and a closed list of already evaluated nodes, expanding the search frontier in an informed manner [3].

A* guarantees the shortest path if the heuristic is admissible (never overestimates the cost) and consistent. It is well-suited to grid-based worlds, navigation meshes, or waypoint graphs and can be modified for dynamic environments or weighted terrains [13].

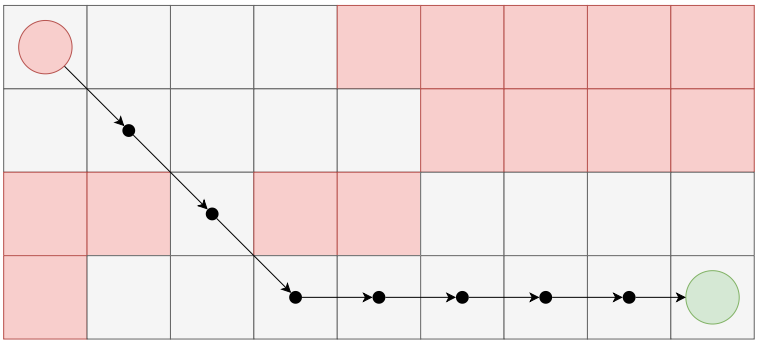


Figure 2.10: Example of A* pathfinding on a 2D grid map using Euclidean distance as the heuristic with obstacles and a goal.

Example: Navigating a Grid World

Fig. 2.10 illustrates a 2D grid-based environment where A* is used to find a path from a red starting point to a green goal. The black cells represent impassable terrain. The algorithm evaluates multiple possible routes, ultimately selecting the shortest viable path that avoids all obstacles.

This basic setup can be extended to support additional game logic, such as avoiding danger zones, dynamically updating the grid in real time, or supporting diagonal movement and terrain penalties. These adaptations make A* suitable for a broad range of genres and gameplay situations.

Applications and Utility

Pathfinding algorithms like A* are fundamental to believable game AI [13]. They ensure that agents move efficiently and naturally within complex environments, often while adapting to dynamic elements like moving platforms or destructible terrain.

- **Advantages**: Guarantees optimal paths (with admissible heuristic), adaptable to many world representations, widely supported in engines.
- **Use cases**: Navigation in RTS, RPG, FPS, and open-world games; movement of enemies, allies, animals, and autonomous agents.

Examples of A* pathfinding in games include:

- *StarCraft II* (Blizzard): Unit navigation uses A* on a grid with continuous updates based on player commands and enemy positions.
- *The Elder Scrolls V: Skyrim* (Bethesda): NPCs use pathfinding through complex 3D environments, including elevation and doorways.
- *Age of Empires* (Ensemble Studios): A* guides armies around terrain, buildings, and dynamically changing battlefields.
- *XCOM: Enemy Unknown* (Firaxis): Movement within tactical grid maps relies on A* to calculate cover-aware paths.
- *The Sims* (Maxis): A* pathfinding enables characters to move around furniture and rooms while fulfilling needs.

2.5.4 Summary and Use in Game Development

The integration of AI into video games significantly enhances gameplay depth, realism, and player immersion. Each AI technique serves a particular role depending on the gameplay requirements:

- Finite State Machines are ideal for structured, predictable behaviors that are easy to test and debug.
- Behavior Trees support complex, adaptive decision-making, making them suitable for games requiring realistic and strategic AI.

- Pathfinding algorithms like A* are foundational for enabling movement and navigation in virtually any spatial environment.

Most modern games employ a combination of these systems to balance simplicity and depth. Whether through enemy patrol logic, strategic pathfinding, or layered decision trees, AI contributes meaningfully to the responsiveness and richness of interactive worlds [3, 13].

Chapter 3

Designing a Roguelite Experience

The aim of this thesis is to design and implement a roguelite video game employing procedural generation techniques. This chapter details the concepts, characteristics, and mechanics that shape the gameplay experience, offering insight into design decisions and their intended impact on player engagement and replayability.

3.1 Game Concept

The game designed for this thesis is a roguelite, set in a procedurally generated maritime world filled with scattered islands. Players take command of a sailing vessel (Fig. 3.1), navigating through an oceanic environment, engaging in strategic naval combat with AI-controlled enemy ships. Victories in these encounters reward the player with gold, a currency used to enhance the ship at strategically placed seaports found on certain islands. A final, randomly located encounter determines the ultimate success or failure of the player's journey. Defeat results in the loss of non-permanent upgrades and forces the player to restart, albeit equipped with persistent upgrades, making subsequent attempts progressively more manageable.

Figure 3.1: The player ship in the game world.

3.1.1 Game World

The game world consists of a procedurally generated archipelago, characterized by varied islands and expansive open seas, see Fig. 3.2. Islands may host seaports, where players can repair and upgrade their ship using the currency earned through combat. Enemy ships roam the seas, actively engaging the player and presenting continuous challenges and opportunities for resource gathering through naval engagements. This world structure encourages exploration, strategic movement, and careful resource management, which are vital to successfully navigating the dynamic challenges the game presents.

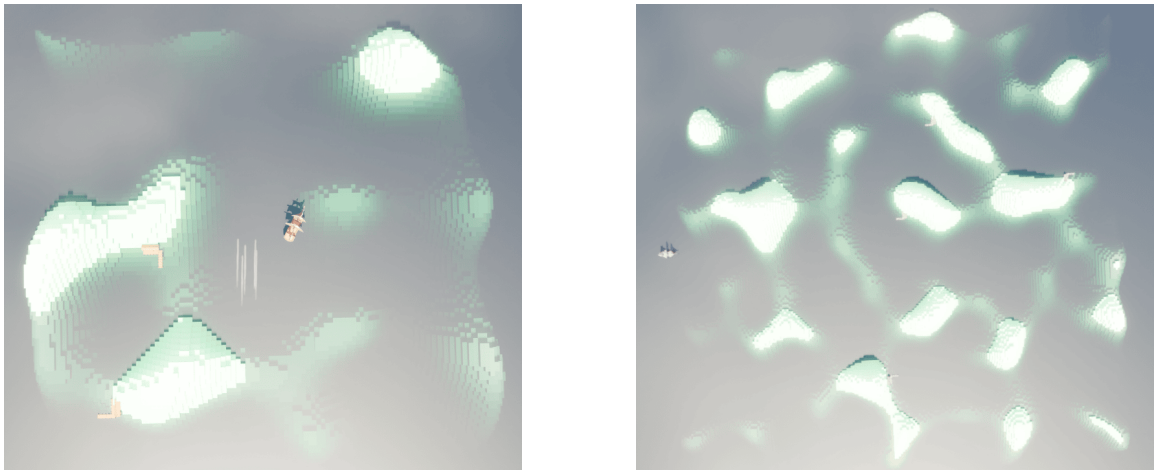


Figure 3.2: Examples of generated worlds with varying size.

3.1.2 Roguelite Genre

The classification of this game within the roguelite genre, detailed in subsection 2.3, is primarily due to its procedural world generation, persistent progression system, and semi-permanent player progression mechanics. Unlike traditional roguelikes, which typically feature complete permadeath without any progression retained, this roguelite design allows players to retain selected permanent upgrades even after losing a session. This persistent progression incentivizes repeated playthroughs and enhances replayability, making each attempt feel rewarding and progressive despite repeated failures.

3.2 Game Mechanics

The gameplay mechanics designed for this thesis emphasize interaction with the procedurally generated environment, strategic combat, resource management, and progression systems.

3.2.1 Movement and Wind

Movement mechanics in the game incorporate a dynamic wind system, significantly impacting gameplay by influencing ship navigation. The wind, characterized by procedurally determined speed and direction, continuously alters sailing conditions. Ships moving with

the wind gain speed advantages, illustrated by the green ship in Fig. 3.3, whereas ships moving against the wind face significant resistance, reducing their effective speed, as depicted by the red ship.

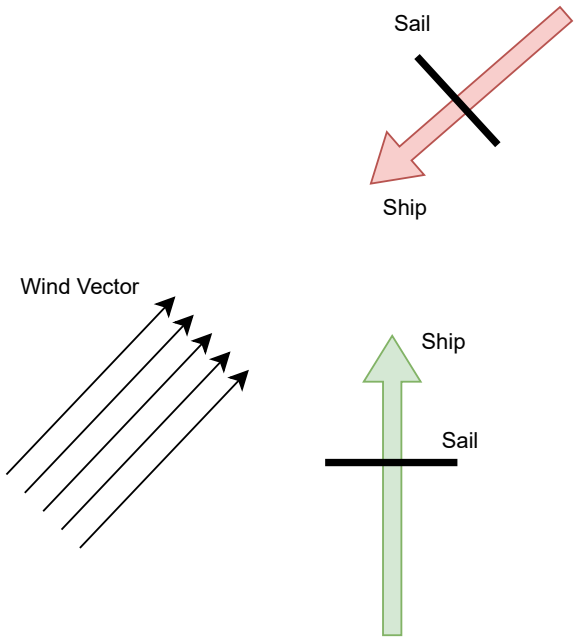


Figure 3.3: Diagram illustrating wind impact on ship navigation.

Players can strategically raise or lower their sails to mitigate or capitalize on wind effects. Fully raised sails neutralize wind effects, allowing ships to maneuver effectively even against adverse wind conditions, while partially raised sails adjust the extent of wind influence. This system encourages strategic sailing decisions and enhances realism and immersion.

3.2.2 Combat and Enemy AI

Naval combat serves as a central gameplay element, requiring tactical skill and strategic positioning. Players engage enemy ships by firing cannonballs from broadside cannons, adjusting shot arcs dynamically to account for distance and relative ship positions. Enemy AI employs similar mechanics, actively pursuing and engaging the player, using procedural decision-making influenced by situational variables such as player positioning, heading, and wind conditions. The AI system is designed to closely mimic the behavior of a human player, operating under the same movement, sail, and combat constraints. It adjusts sails in response to wind, steers based on tactical intent, and only fires when properly aligned – mirroring the actions a skilled player would take. This creates varied and challenging combat scenarios, requiring adaptive player strategies to succeed.

3.2.3 Currency, Seaports and Upgrades

Gold, the game's primary currency, is acquired through successful naval engagements. Defeated enemy ships leave crates containing randomized gold amounts, incentivizing aggressive and strategic combat.

Seaports scattered throughout the world act as key progression hubs, enabling players to repair their ships and purchase upgrades. The player has to find a seaport, sail to it, and interact with it to be able to purchase repairs or upgrades. These seaports offer two categories of upgrades: standard (temporary) and permanent upgrades, each enhancing gameplay in distinct ways. Players can utilize gold at seaports as follows:

- **Ship Repairs**: Restores the ship to full durability, costing gold proportional to the current damage sustained. This action can be repeatedly used.
- **Standard Upgrades**: Temporary enhancements active only during the current gameplay session. See Table 3.1.
- **Permanent Upgrades**: One-time investments that persist across multiple gameplay sessions (see Table 3.2). While more costly upfront, they grant lasting benefits, forcing players to balance immediate power gains against long-term progression.

Upgrade Name	Cost (Gold)	Description
Reinforced Hull	100	Increases ship durability.
Steering Speed	80	Enhances steering responsiveness.
Sail Pulley Oil	60	Enables faster sail adjustments.
Smooth Cannons	120	Cannonballs travel faster and farther.
Aerodynamics	90	Boosts the base forward speed of the ship.

Table 3.1: Available standard (non-permanent) upgrades at seaports.

Permanent upgrades are priced roughly three times higher than their standard counterparts, reflecting their persistent nature. Players must decide whether to spend gold on repeated temporary boosts for immediate advantage or invest in a permanent upgrade for long-term benefit, adding a strategic layer to resource management.

Upgrade Name	Cost (Gold)	Description
Reinforced Hull (Permanent)	300	Increases ship durability permanently.
Steering Speed (Permanent)	240	Enhances steering responsiveness permanently.
Sail Pulley Oil (Permanent)	180	Enables faster sail adjustments permanently.
Smooth Cannons (Permanent)	360	Cannonballs travel faster and farther permanently.
Aerodynamics (Permanent)	270	Boosts the base forward speed permanently.

Table 3.2: Available permanent upgrades at seaports.

3.2.4 End of the Game and Losing

The final encounter represents the climactic challenge of each gameplay session, composed of a group of advanced enemy ships. The location of this encounter is randomly determined, compelling players to explore the procedural world thoroughly. These enemy ships remain passive until provoked, allowing the player discretion over engagement timing.

Losing the player's ship at any stage results in defeat, stripping all acquired standard upgrades while preserving permanent upgrades. This mechanic reinforces the roguelite progression model, incentivizing repeated playthroughs and creating a rewarding cycle of continuous improvement and exploration.

3.3 User Interface

The user interface (UI) of the game is deliberately minimalistic, ensuring immersion while providing clear and concise gameplay information.

Figure 3.4: A showcase of the in-game player camera.

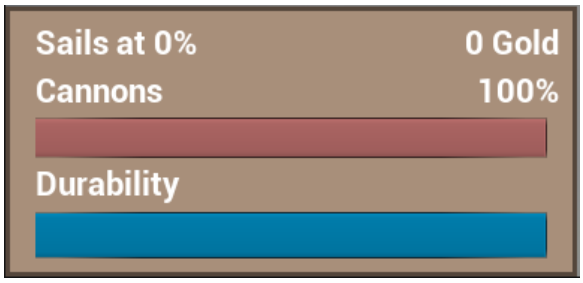


Figure 3.5: A showcase of the in-game UI.

The primary view available to the player is showcased in Fig. 3.4, where the camera is placed strategically behind the player's ship, giving an optimal perspective of the surroundings, and facilitating navigation and combat decisions.

During gameplay, critical information such as ship durability, cannon fire cooldown, and available gold is clearly displayed as depicted in Fig. 3.5. This real-time feedback ensures the player remains informed about their current status and immediate environment.

Figure 3.6: A showcase of the escape menu UI.

An escape menu is also present in order to enable players to pause or exit the game and view all of their accrued upgrades, as can be seen in Fig. 3.6.

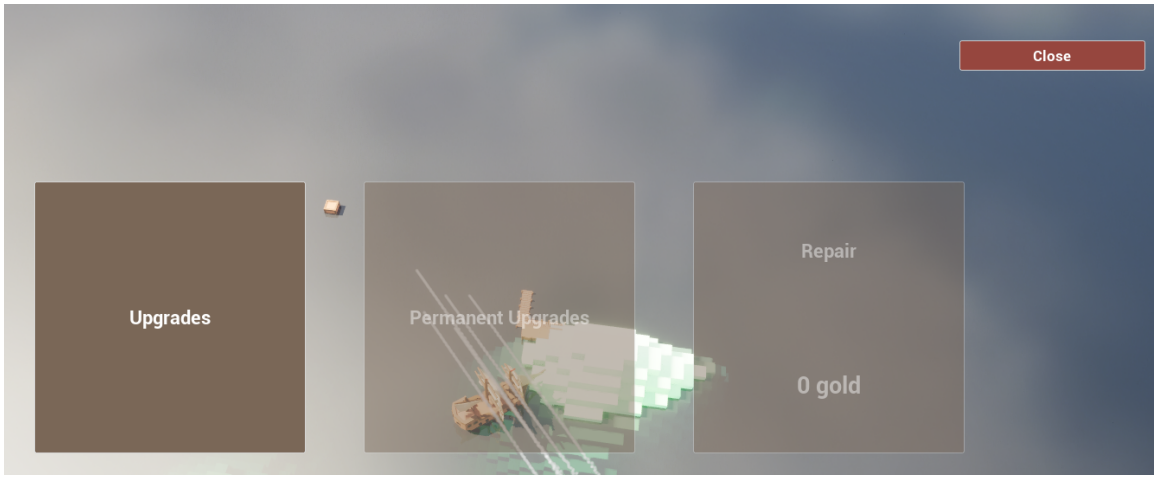


Figure 3.7: A showcase of the seaport menu UI.

Interaction with seaports, a fundamental aspect of the game's progression mechanics, utilizes a dedicated UI illustrated in Fig. 3.7. Players interact with this intuitive menu to

select repairs, standard upgrades, and permanent upgrades, providing strategic options to enhance their ship's capabilities.

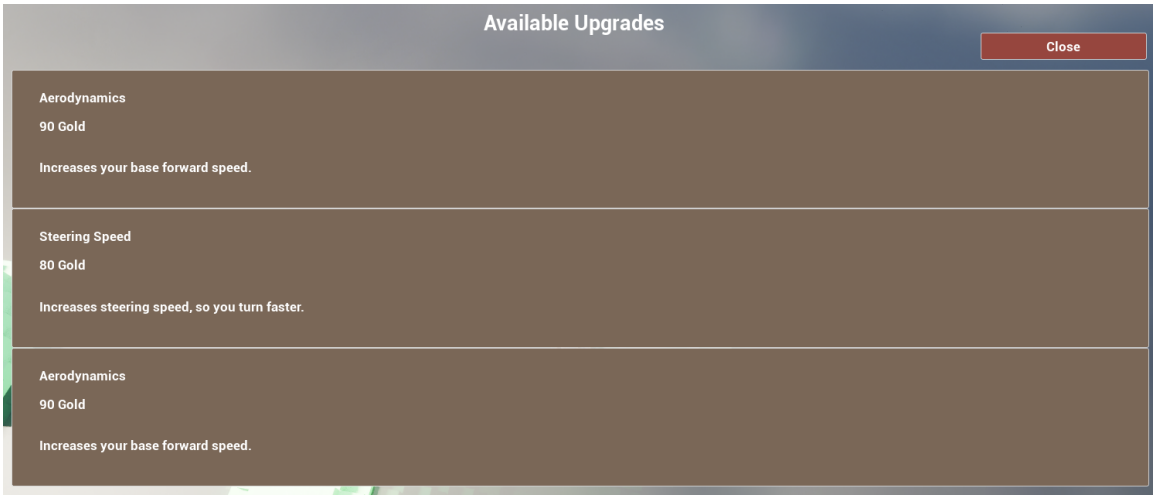


Figure 3.8: A showcase of the upgrades menu UI.

Figure 3.9: A showcase of the main menu UI.

Additionally, the upgrades menu, detailed in Fig. 3.8, offers players clear choices among available enhancements, presented with their respective costs and benefits, facilitating informed strategic decisions to optimize their ship for subsequent encounters.

When the player's ship is sunk, a game over UI is presented (see Fig. 3.10) to the player that allows them to exit to the main menu or exit the game completely.

If the player manages to beat the game, a victory menu is displayed which first allows the player to select from three free random permanent upgrades as a reward, see Fig. 3.11, and upon choosing allows the player to again exit to main menu or exit the game altogether as can be seen in Fig. 3.12.

Figure 3.10: A showcase of the game over UI.

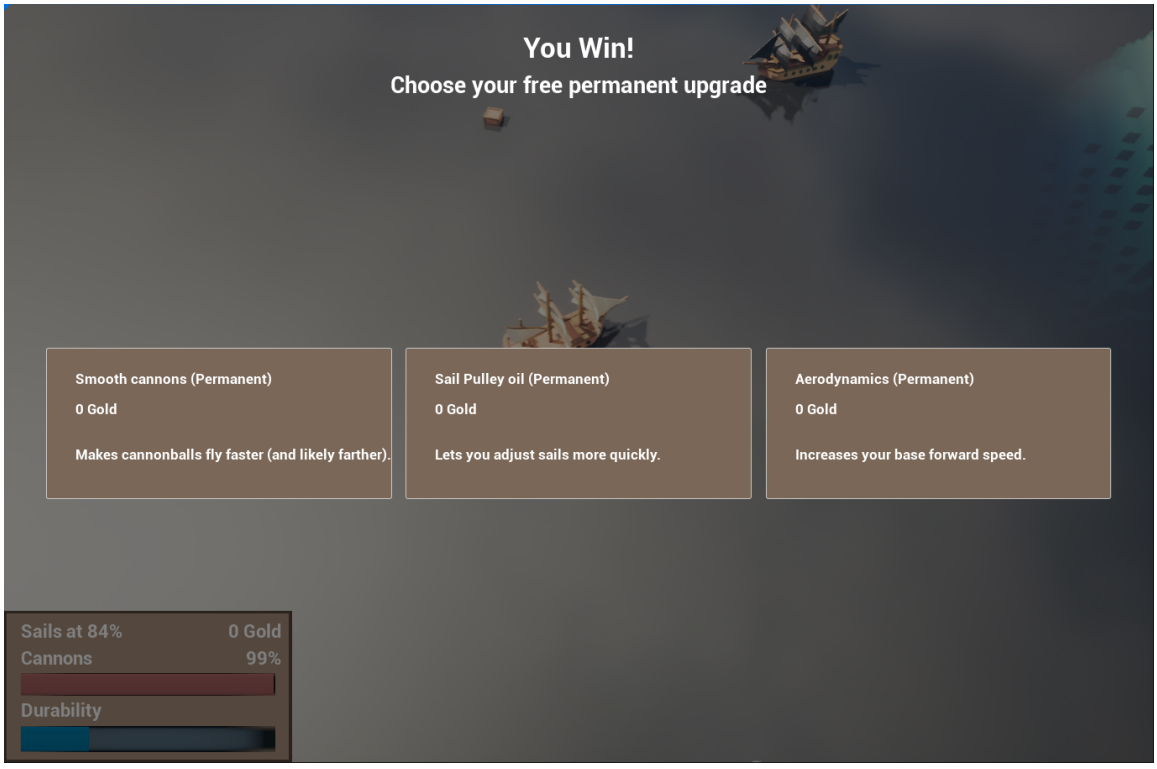


Figure 3.11: A showcase of the first part of the victory UI.

Figure 3.12: A showcase of the second part of the victory UI.

Overall, the game's UI effectively supports the gameplay loop, maintaining player immersion, and providing interaction with essential game mechanics.

Chapter 4

Implementation of a Roguelite Video Game

Having outlined the conceptual foundations and detailed the design aspects of the roguelite video game, this chapter now describes the practical implementation process. It provides a comprehensive overview of how the previously discussed theoretical concepts and mechanics were realized, focusing on critical areas such as procedural generation methods, gameplay mechanics, and artificial intelligence systems.

The sections within this chapter delve into specific implementation details, addressing how procedural algorithms were adapted and integrated to generate the game world, the technical realization of core gameplay mechanics, including ship navigation, combat systems, and upgrade systems, as well as how AI-driven enemy behavior was practically implemented to enhance gameplay dynamics.

Finally, this chapter emphasizes the structure of the implemented system, describing how its individual components function and interact to form a cohesive experience. Attention is given to how the procedural generation algorithms, gameplay mechanics, and artificial intelligence systems integrate, ensuring dynamic gameplay.

To understand this chapter, it is paramount to understand that while most of the code was written in the `C++` programming language, Unreal Engine also has a visual scripting language called `Blueprint`, which is used for UI and minor features in the implementation of the game.

4.1 General Implementation Structure

The game's implementation relies heavily on the underlying structure provided by Unreal Engine, which orchestrates the initialization sequence upon the game's startup. Unreal Engine initiates the game execution by loading the Engine's core modules, assets, and system-level configurations, preparing the runtime environment required for gameplay.

When the initialization sequence completes, the Engine then proceeds to instantiate the `GameMode` class, a central component defining fundamental gameplay rules and logic. Within Unreal Engine, the `GameMode` is responsible for establishing initial game states, player spawning logic, and setting the conditions for match progression.

In the context of this project, a custom subclass named `CustomGameMode` was implemented, extending Unreal's default `GameModeBase`. The instantiation of the `CustomGameMode` marks the transition from the engine-level initialization to the execution

of custom gameplay logic. This is where the game's specific systems – including procedural generation algorithms for world creation, spawning of player ships and enemy vessels, initialization of gameplay mechanics, and user interface setup – begin execution.

The `CustomGameMode` class, as the entry point for custom game logic, is responsible for initializing critical gameplay components and orchestrating interactions between them. Specifically, it performs the following tasks:

- Initializes the procedural world generation system, ensuring islands, seaports, navigable seas, and enemy ships are generated at runtime.
- Handles the spawning and setup of the player's ship as well as the `PlayerController` responsible for the user interface.
- Establishes and maintains the game's progression state, monitoring player interactions, resource acquisition, and transitions between game phases.
- Sets up the user interface elements, such as HUD widgets for displaying crucial information like player ship durability, available currency.

To visualize how these individual modules interact, Fig. 4.1 illustrates the key components initialized and managed by the `CustomGameMode`. The arrows indicate execution flow at the start of the program.

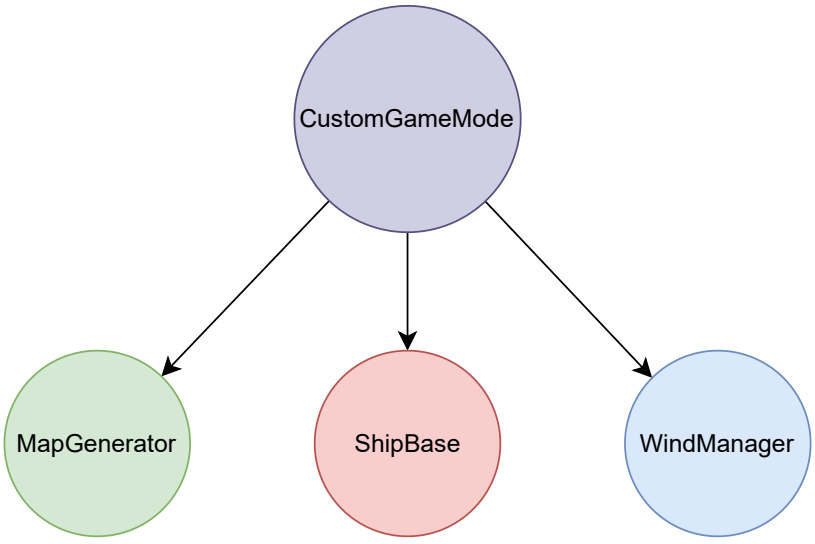


Figure 4.1: A diagram of major gameplay modules initialized by `CustomGameMode`.

The communication diagram clearly depicts the central role of the `CustomGameMode`, acting as a mediator that initializes other main modules such as the procedural world generator, enemy AI controllers, player-controlled ship mechanics, and user interface modules. This centralized structure ensures a consistent, organized execution of gameplay logic, facilitating efficient integration and streamlined data flow between modules.

From this foundational structure, subsequent sections will detail the specific implementation approaches employed for procedural world generation, enemy AI logic, player movement, wind, and other game systems.

4.2 World Generation

The `MapGenerator` class is a core component responsible for generating the game's procedurally created terrain, including points of interest such as seaports. It is initialized at the start of each gameplay session by the `CustomGameMode` class, which invokes the primary method `GenerateTerrain()` to begin terrain generation (see Algorithm 1).

Algorithm 1 Terrain Generation Procedure

procedure GENERATETERRAIN()
    Clear previous cell data
    for $X_{ch} \leftarrow 0$ **to** $worldSize/chunkSize - 1$ **do**
        for $Y_{ch} \leftarrow 0$ **to** $worldSize/chunkSize - 1$ **do**
            GENERATECHUNK(X_{ch}, Y_{ch})
        end for
    end for
    UPDATENEIGHBORINFO()
    SPAWNWATER()
    SPAWNSEAPORTS()
    Calculate Player spawn location
    SPAWNENEMYSHIPS(Player spawn location)
    SPAWNBOSSSHIPS(Player spawn location)
    return Player spawn location
end procedure

procedure GENERATECHUNK(X_{ch}, Y_{ch})
    for $x \leftarrow 0$ **to** $chunkSize - 1$ **do**
        for $y \leftarrow 0$ **to** $chunkSize - 1$ **do**
            $X_w \leftarrow X_{ch} \cdot chunkSize + x$
            $Y_w \leftarrow Y_{ch} \cdot chunkSize + y$
            $noiseValue \leftarrow$ Perlin noise at (X_w, Y_w)
            $falloffValue \leftarrow$ Falloff at (X_w, Y_w)
            $finalHeight \leftarrow \text{Clamp}\left(\frac{noiseValue+1}{2} - falloffValue\right)$
            $cubeHeight \leftarrow \text{Round}(finalHeight \cdot maxHeight + 1)$
            Create new `CellData` with position and cube height
            SPAWNTERRAIN(X_w, Y_w, z)
            Add `CellData` to generated cells list
        end for
    end for
end procedure

Initially, the algorithm clears any existing terrain cell data to prevent residual information from influencing new terrain generation. The generated terrain is divided into square segments known as *chunks*, each subdivided into smaller *cells*. Terrain generation proceeds iteratively by looping through each chunk and invoking the method `GenerateChunk()` for detailed cell-level computation.

Within each chunk, the elevation of each cell is determined by combining Perlin noise and a custom-defined falloff function. Perlin noise provides smoothly varying randomness across the terrain, while the falloff function ensures terrain elevations gradually diminish

towards the edges, effectively blending generated landmass into the surrounding ocean. The falloff function is mathematically defined in Eq. 4.1.

$$\begin{aligned} \text{falloffValue}(x, y) &= \frac{val^a}{val^a + (b - b \cdot val)^a}, \\ \text{where} \quad val &= \max\left(\left|\frac{2x}{\text{width}} - 1\right|, \left|\frac{2y}{\text{height}} - 1\right|\right) \end{aligned} \tag{4.1}$$

The parameters used ($a = 3$, $b = 6$) were chosen empirically to ensure a visually appealing transition from terrain to ocean. Fig. 4.2 visualizes this falloff function, clearly illustrating how terrain elevation smoothly decreases toward map edges.

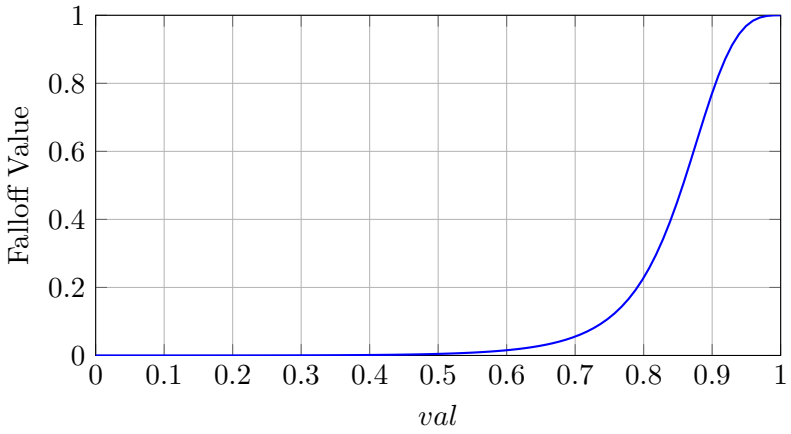


Figure 4.2: Falloff function used to smoothly reduce elevation at terrain edges.

After applying noise and falloff, continuous height values are converted into discrete cube heights using the calculation in Eq. 4.2.

$$\begin{aligned} \text{cubeHeight}(x, y) &= \text{Round}\left(\text{finalHeight}(x, y) \cdot \text{maxHeight} + 1\right), \\ \text{finalHeight}(x, y) &= \text{Clamp}\left(\frac{\text{noiseValue}(x, y) + 1}{2} - \text{falloffValue}(x, y),\ 0,\ 1\right) \end{aligned} \tag{4.2}$$

Each cell's cube height and other relevant information are stored within a `CellData` structure. Subsequently, the method `SpawnTerrain()` is invoked to instantiate physical terrain blocks within the Unreal Engine environment. This method places cubes at computed locations with appropriate height and dynamic materials, and configures collision properties for proper in-game interactions.

Following the terrain generation, the `UpdateNeighborInfo()` method is executed. This method iterates through each cell, checking its immediate neighbors (north, south, east, and west). Cells adjacent to water-level cells are flagged as coastal, determining viable locations for seaports.

The ocean surface itself is rendered using the `SpawnWater()` method, which instantiates a single large flat plane that covers the entire generated map, positioned at a predefined sea level to visually represent open ocean surrounding the islands.

Next, `SpawnSeaports()` is used to place seaports strategically. This method selects random coastal cells while enforcing minimum distance constraints between ports, computes

precise spawn locations and orientations based on cell coastline orientation, and instantiates seaport actors.

After all seaports have been placed, the map generator uses the `SpawnEnemyShips` method, which determines the total number of enemy ships to create. It then compiles a list of all terrain cells that lie beneath the water surface and samples from these valid positions. Each candidate spawn point is tested to ensure it is sufficiently far from the player's start location and from any previously chosen ship, preventing overlaps or immediate encounters. To encourage encounters closer to the centre of the map, locations nearer the middle are preferentially selected. Finally, an enemy ship actor is instantiated at each chosen spot just above the water plane, with a randomized yaw angle, producing a well-spaced, centrally focused distribution of AI vessels ready to patrol and engage the player. The `MapGenerator` also executes the `SpawnBossShips` method, which spawns the game's final encounter, a fleet of improved enemy ships acting together using candidate locations far away from the player spawn.

Through the combined use of procedural algorithms, strategic placement logic, and seamless integration of environmental elements, the generated terrain provides a consistently immersive and varied game environment for each playthrough.

4.3 Enemy AI

This section details the technical implementation of the enemy ship Artificial Intelligence (AI) system described in the design subsection 3.2. The goal of the AI is to simulate believable and tactically competent opponents that operate under the same constraints and mechanics as the player. To achieve this, the implementation closely mimics player behavior, including navigation influenced by wind, real-time steering and sail control, and positioning for broadside cannon attacks.

The enemy AI is crucial in order for the game concept to work, as combat is closely tied with progression and, as such, must not be too easy and has to be engaging and challenging.

4.3.1 Spawning Enemy Ships

As was already referenced in section 4.2, the `MapGenerator` generates enemy ships alongside the terrain and seaports.

Once all seaports have been placed, the total number of enemy ships to spawn is computed as shown in Eq. 4.3.

$$N \;=\; 2 \cdot \mathrm{PortCount} \tag{4.3}$$

A candidate list of water-safe cells is assembled by selecting those terrain cells whose world-space Z coordinate lies strictly below the global water height, as can be seen in Eq. 4.4.

$$\mathrm{CellZ} \;<\; \mathrm{WorldWaterHeight} \tag{4.4}$$

From these, positions are iteratively sampled – each new position $\mathbf{x}_i$ must satisfy a minimum separation $D_{\min}$ from every previously accepted ship and from the player's spawn location as described in Eq. 4.5.

$$\|\mathbf{x}_i - \mathbf{x}_j\| \;\geq\; D_{\min} \quad \text{and} \quad \|\mathbf{x}_i - \mathbf{x}_{\text{player}}\| \;\geq\; D_{\min} \tag{4.5}$$

To bias spawns toward the centre of the map, every candidate location $\mathbf{x}$ is assigned a weight calculated in Eq. 4.6.

$$w(\mathbf{x}) \;=\; 1 \;-\; \left(\frac{\|\mathbf{x}-\mathbf{c}\|}{R_{\max}}\right)^{2}, \tag{4.6}$$

where $\mathbf{c}$ is the map centre and $R_{\max}$ is half the map's diagonal. A weighted sampling without replacement then chooses the final N spawn points.

Finally, at each selected $\mathbf{x}$, an enemy ship actor is instantiated at $(x, y, \text{WorldWaterHeight} + \epsilon)$ with ϵ a small vertical offset, and given a uniformly random yaw in $[0, 360°)$. This ensures a well-spaced, centrally focused distribution of enemy vessels.

4.3.2 Enemy Ship Behavior

Enemy ship behavior is governed by a minimalistic behavior tree built around a single monolithic service, `ControlEnemyShip`, which is responsible for all per–frame decision-making. This service continuously evaluates the tactical context of the encounter, updates the AI ship's combat mode, and controls steering, sail settings, and anchor usage accordingly. In addition to this service, the tree includes a single task node, `FireCannons`, which handles the actual act of attacking when appropriate (see Fig. 4.3).

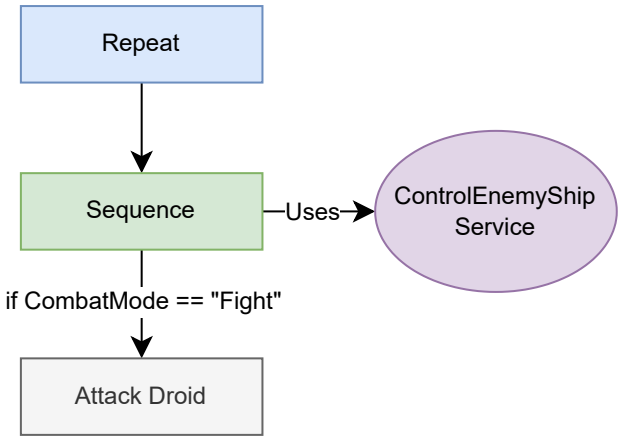


Figure 4.3: Behavior Tree for enemy ships. The service `ControlEnemyShip` evaluates combat context and movement each tick, and `FireCannons` attempts to fire if conditions are met.

The decision to use a monolithic service instead of breaking behavior into smaller tasks stems from the nature of ship control in this game. To realistically simulate naval movement and mimic human player input, the AI must adjust steering, sail height, and reaction to wind direction on a per-frame basis. This requires logic that ticks every frame, continuously processing environmental data and updating ship state with fine granularity. However, Unreal Engine's behavior tree system does not allow tasks to execute every tick, nor does it provide an intuitive or clean way to coordinate between tick-based services and state-based tasks. Splitting the logic into multiple services and tasks led to inconsistent control and unresponsive behavior during testing. As a result, all core logic was consolidated

into a single, tick-driven service to maintain tight control over navigation and engagement decisions.

The `ControlEnemyShip` service determines the ship's current combat state based on distance to the player. These states include *Idle*, *Chase*, and *Fight*. Transitioning between them incorporates buffer zones and minimum durations, preventing constant switching when conditions hover near a threshold. When in *Chase*, the ship turns directly toward the player and pursues at maximum practical speed. When in *Fight*, it maneuvers to position its side toward the player for a broadside attack. If the player escapes, the AI returns to *Idle* once far enough.

Fig. 4.4 illustrates the combat mode transition logic used by the `ControlEnemyShip` service. The enemy ship operates in one of three high-level states: *Idle*, *Chase*, and *Fight*, depending on its distance to the player. Transitions between these states are triggered by proximity checks. When the player is detected within attack range, the ship enters the *Fight* state and prepares for a broadside. If the player retreats slightly, the ship transitions to the *Chase* state to pursue. If the distance increases further, it reverts to *Idle*. These transitions are smoothed by using two separate distance thresholds – *FightDistance* and *EscapeDistance* – which provide buffer zones that reduce oscillation between states. The diagram clearly shows how proximity dictates behavioral shifts, allowing the AI to manage engagement in a context-aware and stable manner.

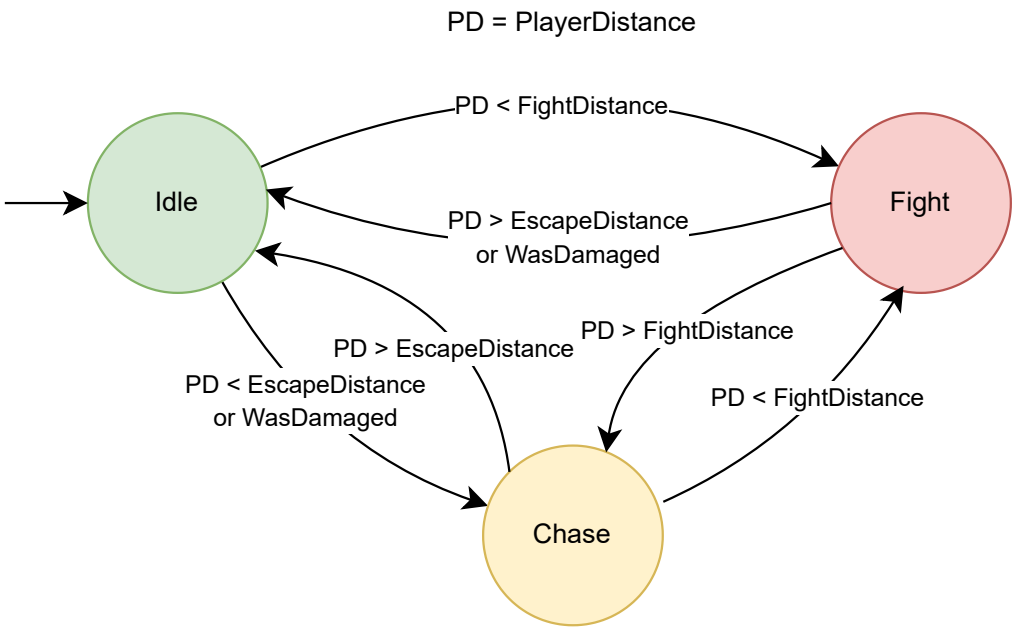


Figure 4.4: A simplified state diagram driving the enemy ship AI state logic.

During all of this, the service adjusts the ship's sails based on current heading and wind direction to optimize speed and responsiveness. Obstacle avoidance is handled by casting rays forward and slightly to the sides of the ship. If potential collisions are detected, steering and sail settings are modified to avoid terrain.

The `FireCannons` task activates only when the ship is in the *Fight* state and correctly aligned for a broadside attack. It checks the AI's current state and orientation, then initiates a cannon volley using parameters that simulate shot distance and angle.

To coordinate between service and task logic, a shared memory structure called a *Blackboard* is used. The Blackboard contains relevant information updated by the service and read by the task. This includes the current combat mode (*Idle*, *Chase*, *Fight*), the anchor state (Raised or Deployed), the yaw direction the ship should aim for, recent distance to the target, and computed control inputs such as steering and sail levels. By decoupling state from execution, the Blackboard allows clean communication between the tick-driven service and the conditionally executed firing task.

Algorithm 2 Tick-based control logic in `ControlEnemyShip` service

```
1:  procedure TickNode()
2:      Measure distance to player
3:      if Distance < fight distance then
4:          Set mode to Fight
5:      else if Distance < chase distance then
6:          Set mode to Chase
7:      else
8:          Set mode to Idle
9:      end if
10:     if mode changed then
11:         Raise or lower the anchor accordingly
12:     end if
13:     if mode == Idle then
14:         return
15:     end if
16:     if mode == Fight then
17:         Compute broadside angle (left/right)
18:         Set yaw target to broadside direction
19:     else
20:         Set yaw target to face player
21:     end if
22:     Compute steering input to reach target yaw
23:     Compute sail input from heading vs wind
24:     Perform obstacle avoidance and override if needed
25:     Apply steering and sail input to the ship
26:     Store control values in blackboard
27: end procedure
```

In this implementation, the Behavior Tree acts more as a scheduling layer for tick-based logic rather than as the central decision-making mechanism. The true intelligence of the AI resides in the continuously running `ControlEnemyShip` service, which handles real-time reasoning and input generation in a way that closely mimics how a human player would operate the ship. This approach results in behavior that is fluid, consistent, and well-suited to the dynamic environment of procedurally generated naval combat. A simplified overview of the service's decision-making process is shown in Algorithm 2.

4.4 Movement Mechanics

The movement system implemented in the game captures the feel of piloting a wind-driven ship while emphasizing player agency, tactical decision-making, and fun navigation, which was outlined in section 3.2. Rather than adhering to strict physical realism, the system is designed to replicate the forces a sailing vessel might experience and translate them into responsive and satisfying controls.

4.4.1 Wind Influence on Movement

A core component of the sailing mechanic is the global wind system, managed by the `WindManager` class. This system defines a constant wind direction and strength for each gameplay session. These parameters are initialized randomly at the beginning of each run, ensuring varied and unpredictable sailing conditions.

Each ship is influenced by the global wind direction, which is represented as a normalized 2D vector $\vec{w}$. The ship's forward direction is likewise expressed as a unit vector $\vec{f}$. The alignment between these two directions is determined using the dot product calculation shown in Eq. 4.7.

$$\begin{aligned} &A = \vec{f} \cdot \vec{w}, \\ &where \quad (A \in [-1, 1]). \end{aligned} \tag{4.7}$$

This scalar value describes how well the ship is aligned with the wind:

- If $A \approx 1$, the ship is sailing with the wind, achieving maximum propulsion.
- If $A \approx 0$, the ship is moving perpendicular to the wind, receiving moderate thrust.
- If $A \approx -1$, the ship is heading directly into the wind and experiences reverse propulsion.

The sail height is represented by a scalar $s \in [0, 1]$, where $s = 0$ corresponds to fully raised sails (no wind influence), and $s = 1$ means the sails are fully lowered (maximum wind capture). The wind-based propulsion force F_{wind} is calculated as shown in Eq. 4.8.

$$F_{\text{wind}} = A \cdot s \cdot S, \tag{4.8}$$

where S is the wind strength constant defined at runtime. Because A can be negative, this formula allows for reverse thrust when the ship sails against the wind.

Finally, the total propulsion force applied to the ship is calculated as can be seen in Eq 4.9.

$$F_{\text{total}} = F_{\text{base}} + F_{\text{wind}}, \tag{4.9}$$

where F_{base} is a constant baseline force ensuring that the ship maintains some maneuverability even without wind. The resulting F_{total} is then multiplied by the ship forward vector, and the result is applied to the ship object as an offset every in-game tick.

This system introduces an interactive and strategic layer to navigation. Players must manage their heading and sail height in real time, adjusting to the current wind direction and strength to optimize speed or control. This contributes to the moment-to-moment skill expression and enhances immersion in the sailing experience.

4.4.2 User Controls: Steering, Sail Adjustment, and Anchor

The ship's control scheme is intentionally simple yet expressive, allowing players to engage with the sailing mechanics without being overwhelmed by complexity. All inputs are handled by the custom `ShipPlayerController`, which communicates with the player's ship to apply movement decisions.

- **Steering:** Players rotate their ship left or right using keyboard input. The rate of rotation depends on the ship's steering responsiveness, which is dynamic and can be improved by getting upgrades.
- **Sail Adjustment:** Players can dynamically raise or lower the sails. Lower sails allow the ship to harness wind more effectively and gain speed, while raised sails reduce speed and make the ship more maneuverable. This provides a meaningful trade-off between speed and control. The speed at which the sails can be adjusted is dynamic and can be improved by getting upgrades.
- **Anchor:** A toggleable anchor allows players to slow and stop their ship completely. It activates with a short delay to simulate the time needed to deploy or retract it, making it unsuitable for instant stops but useful for strategic positioning or docking at seaports.

This control scheme encourages the player to actively engage with the ship as a navigational puzzle, constantly adjusting course, speed, and orientation to achieve objectives under varying wind conditions.

4.4.3 Visual Feedback and Wind Awareness

In addition to the interactive wind-driven movement model, the game provides players with intuitive visual feedback to reinforce their understanding of current sailing conditions. This is crucial for maintaining situational awareness in a procedurally generated world where the wind direction and strength are randomized each session.

To support this, the `WindManager` class periodically spawns wind particle effects in the vicinity of the player's ship. These particles (Fig. 3.1) are spawned with a velocity matching the global wind direction and strength. Their visual motion across the screen conveys the heading and intensity of the wind at a glance, allowing the player to visually estimate wind alignment without needing to rely solely on abstract UI indicators. This helps players make quick decisions about whether to adjust their sails or heading to better exploit the current wind.

In parallel, the user interface displays the current sail state as a percentage, reflecting how far the sails are currently lowered, see Fig. 3.5. This percentage corresponds directly to the wind multiplier $s \in [0, 1]$ used in propulsion calculations. A fully lowered sail (100%) means full wind influence, while a fully raised sail (0%) negates all wind-driven thrust. By observing this UI element, players can manage sail settings more precisely during high-pressure situations such as combat or navigation near islands.

Together, the wind particles and UI display ensure that the core sailing mechanics are not only interactive but also legible and accessible. Players receive immediate feedback on their adjustments, which supports strategic mastery of wind-based movement while preserving immersion in the game world.

4.5 Other Features

In addition to the major gameplay systems already discussed – such as procedural terrain generation, ship movement, combat mechanics, and artificial intelligence – the game incorporates several supporting features that contribute to overall polish, strategic depth, and player immersion. These auxiliary systems include the upgrade and currency system, cannon firing and cooldown mechanics, and the user interface architecture.

4.5.1 Upgrades and Currency

A central aspect of progression in the game is the collection and strategic expenditure of gold, which serves as the primary in-game currency. Players earn gold by defeating enemy ships, which drop floating crates that the player can collect by sailing over them. Each crate contains a randomized amount of gold, providing an incentive for aggressive play and successful naval combat.

Gold is spent at seaports, where players can repair their ship and purchase upgrades, which are divided into two categories:

- Standard Upgrades: Temporary enhancements that apply only for the current session. These include improvements to steering speed, sail adjustment speed, cannonball velocity, and overall movement speed. Standard upgrades offer immediate tactical benefits that are useful for surviving difficult encounters.
- Permanent Upgrades: High-cost investments that persist to future playthroughs. Once purchased, they permanently enhance ship performance, creating a sense of long-term progression. For example, a permanent upgrade to ship durability ensures the player starts each run with more resilience.

The upgrade system is implemented using an extensible object-oriented architecture. Each upgrade is defined as a subclass of `UBaseUpgradeEffect`, with overridden methods to apply specific stat modifications to the player ship. This modularity allows new upgrades to be easily introduced with minimal disruption to the existing codebase.

When the player enters the interaction zone of a seaport, a dedicated UI menu becomes available, allowing the player to view current gold, select from available upgrades, and confirm purchases. This menu reflects which upgrades are already owned, greying out or locking them as appropriate. Purchased upgrades immediately apply their effects to the ship through the ship's internal stat management component.

4.5.2 Cannon Firing and Cooldown System

The cannon firing system provides an engaging, skill-based mechanic central to combat. Players initiate a broadside volley by holding either the Q or E key, which corresponds to the ship's left and right cannon banks, respectively. A total of six cannonballs are spawned along the selected side of the ship, spaced evenly and fired with slightly varied vertical arcs depending on how long the firing key was held.

Each cannon volley is governed by a cooldown timer, preventing spamming and encouraging timing and positioning. This cooldown is visually represented in the game's UI as a dynamic bar that fills over time (see Fig. 3.5). The cooldown logic ensures that each volley feels impactful and forces the player to commit to each attack.

Cannonball projectiles are affected by gravity and inherit the ship's forward velocity at the moment of firing, resulting in dynamic and slightly unpredictable trajectories. This makes aiming a skillful process, particularly while the ship is moving or turning. If cannonballs collide with enemy ships, they deal damage, which can ultimately destroy the enemy and reward the player with gold.

4.5.3 User Interface Architecture

The user interface (UI) is implemented using Unreal Engine's UMG Blueprint system and is designed to present only essential information while maintaining immersion. The primary UI components include:

- Durability Bar: Displays the ship's current health as a visual bar on the screen.
- Gold Counter: A simple numerical indicator showing the player's current amount of gold.
- Cannon Cooldown Bar: A visual representation of the remaining cooldown before the player can fire another volley.
- Sail Height Indicator: Displays a percentage representing the current sail setting, helping players make informed decisions about wind usage.

Additionally, specialized menus are implemented for interactions:

- Main Menu: A standard video game main menu with an option to start the game and an option to exit.
- Escape Menu: Allows the player to pause, quit the game, or review all upgrades purchased during the session.
- Upgrades Menu: Shows a categorized list of available upgrades with descriptions, costs, and ownership status.
- Seaport Menu: Provides the interface for ship repair and upgrade purchases when docked at a seaport.
- Game over menu: This menu is displayed when the player's ship gets destroyed. It allows the player to exit to the main menu or exit the game completely.
- Game Victory Menu: This menu first displays a selection of three permanent upgrades for the player to acquire for free as a reward for winning. After a player chooses their free permanent upgrades, they are again given the choice to exit to the main menu or exit altogether.

These UI components are updated in real-time by the `ShipPlayerController`, which holds references to relevant game data such as ship health, upgrade status, and fire cooldowns.

4.5.4 Persistent Data Handling

To support the permanent upgrade system, the game saves selected player data between sessions. This includes a record of purchased permanent upgrades, which are restored when a new run begins. Data persistence is handled using Unreal Engine's SaveGame system, which serializes necessary state data to disk and reloads it at the start of a new playthrough.

This system ensures continuity across sessions and reinforces the roguelite progression loop, where players are rewarded for long-term commitment and skillful play.

4.6 User Testing

To validate the gameplay mechanics, gather user feedback, and identify areas for improvement, a user testing session was conducted following the completion of core gameplay systems. The game was distributed to a small group of selected testers who were instructed to play the game freely and provide subjective feedback on their experience. No specific gameplay paths were enforced, allowing users to naturally explore and engage with the game mechanics.

Participants were provided with a brief overview of the game's controls and objectives, after which they played independently. Feedback was collected both informally through conversations and via written responses, focusing on gameplay feel, UI clarity, difficulty, and overall enjoyment.

The testing yielded several notable observations:

- One user expressed that the ship's base movement speed felt too slow, particularly when sailing against the wind. While they understood that the game incentivized players to align with wind direction, they felt that the penalty for sailing against the wind was overly punishing, especially during exploration. As a result, the base speed value was slightly increased to improve responsiveness without undermining the wind system's strategic value.

- Another tester noted discomfort with the camera behavior, specifically its rotation following the ship's yaw. They stated that the constantly rotating camera made orientation disorienting during turning maneuvers. However, after further investigation, it was concluded that decoupling the camera from ship rotation significantly reduced forward visibility, making navigation and aiming difficult. As such, this feedback was acknowledged but ultimately disregarded to preserve gameplay clarity.

- A third user highlighted frustration with the cannon firing system, particularly the lack of visual indication for the cannon cooldown. Although they understood there was a delay between volleys, they found it difficult to time their shots effectively due to the absence of a clear indicator. In response to this, a visual fire cooldown bar was added to the user interface, dynamically representing the remaining time before the player can fire again. This change improved player awareness and reduced uncertainty during combat.

Overall, user testing provided critical insight into how players perceived core mechanics. The feedback led to meaningful adjustments that improved gameplay feel and usability, while also reinforcing design decisions that balanced strategy and accessibility.

Chapter 5

Conclusion

The aim of this thesis was to study modern techniques in game development within Unreal Engine, design a 3D roguelite game with a nautical theme, implement the game using procedural generation and artificial intelligence, and validate the gameplay through user testing. This objective was successfully met. A fully functional, original game prototype was created, capturing the key features of the roguelite genre while introducing unique mechanics such as wind-based navigation and strategic ship upgrades.

Each requirement outlined in the assignment was fulfilled. Theoretical knowledge of game development, procedural generation methods, and artificial intelligence techniques was studied and described in Chapter 2. The game was then designed with respect to these principles, as discussed in Chapter 3. A complete implementation followed, detailed in Chapter 4, where procedural terrain generation, AI-driven enemy ships, sailing mechanics, and user interfaces were integrated into a cohesive system. Lastly, the game was distributed to testers, and their feedback was collected and used to improve gameplay, thus fulfilling the user testing requirement.

The implemented game features a procedurally generated archipelago, a dynamic wind system affecting movement, and AI-controlled ships that navigate and engage using the same constraints as the player. The player controls a customizable sailing ship, engages in naval combat, collects gold, and chooses between temporary and permanent upgrades at seaports. These systems were integrated into a complete gameplay loop, supported by a modular upgrade architecture, a behavior tree-driven AI system, and an interactive user interface.

Personally, this thesis allowed me to delve into game development for the first time and discover the joy that lies within. I learned how to structure complex gameplay systems, how to prototype, test, and iterate on interactive mechanics, and how to think critically about player experience. It was a challenging but rewarding experience that strengthened both my technical and creative skills.

Future work may involve extending the project by improving the procedural generation algorithm, adding dynamic weather effects, additional enemy and player ship types, more varied upgrades, more points of interest, or procedurally generated quests and events. The AI could also be made more sophisticated through goal-driven or team-based behavior. These additions would improve replayability, depth, and polish, bringing the project closer to a complete indie game experience.

Bibliography

[1] Bartle, R. A. *Designing Virtual Worlds*. 1st ed. New Riders, 2003. ISBN 978-0-13-101816-7.

[2] Bethke, E. *Game Development and Production*. 1st ed. Wordware Publishing, Inc., 2003. ISBN 978-1-55622-951-0.

[3] Buckland, M. *Programming Game AI by Example*. 1st ed. Jones & Bartlett Learning, 2004. ISBN 978-1556220784.

[4] Bycer, J. *Game Design Deep Dive: Roguelikes*. 1st ed. CRC Press, 2021. ISBN 978-1000361988.

[5] Chomsky, N. Three Models for the Description of Language. *IRE Transactions on Information Theory*. 1st ed., 1956, vol. 2, no. 3, p. 113–124. Available at: https://chomsky.info/wp-content/uploads/195609-.pdf.

[6] Craddock, D. L. *Dungeon Hacks: How NetHack, Angband, and Other Roguelikes Changed the Course of Video Games*. 1st ed. Press Start Press, 2015. ISBN 978-1518655312.

[7] Egenfeldt Nielsen, S.; Smith, J. H. and Tosca, S. P. *Understanding Video Games: The Essential Introduction*. 3rd ed. Routledge, 2019. ISBN 978-1138849822.

[8] Gregory, J. *Game Engine Architecture*. 2nd ed. A K Peters, Ltd., 2014. ISBN 978-1466560017.

[9] Harrison, M. A. *Introduction to Formal Language Theory*. 1st ed. Addison-Wesley, 1978. ISBN 978-0201029550.

[10] Hendrikx, M.; Meijer, S.; Velden, J. V. D. and Iosup, A. *Procedural Content Generation for Games: A Survey* online. Northeastern University, 2013. ISSN 1551-6857. Available at: https://doi.org/10.1145/2422956.2422957. [cit. 2025-03-22].

[11] Ilachinski, A. *Cellular Automata: A Discrete Universe*. 1st ed. World Scientific, 2001. ISBN 978-9812381835.

[12] Iuppa, N. and Borst, T. *End-to-End Game Development: Creating Independent Serious Games and Simulations from Start to Finish*. 1st ed. Focal Press, 2010. ISBN 978-0-240-81179-6.

[13] Millington, I. *Artificial Intelligence for Games*. 2nd ed. CRC Press, 2019. ISBN 978-1138486416.

[14] Muriel, D. and Crawford, G. *Video Games as Culture: Considering the Role and Importance of Video Games in Contemporary Society*. 1st ed. Routledge, 2018. ISBN 978-1138655119.

[15] Penazzo, D. *2D Game Development: From Zero To Hero*. Pseudocode Editionth ed. Self-published, 2020.

[16] Ricci, A. *Modeling Software with Finite State Machines: A Practical Approach*. 1st ed. CRC Press, 2011. ISBN 978-1439867079.

[17] Schiff, J. L. *Cellular Automata: A Discrete View of the World*. 1st ed. John Wiley & Sons, 2008. ISBN 978-0470168790.

[18] Shaker, N.; Togelius, J. and Nelson, M. J. *Procedural Content Generation in Games*. 1st ed. Springer, 2016. ISBN 978-3-319-42716-4.

[19] Thorn, A. *Game Engine Design and Implementation*. 1st ed. Jones & Bartlett Learning, 2010. ISBN 978-0763784515.

[20] Yannakakis, G. N. and Togelius, J. *Artificial Intelligence and Games*. 1st ed. Springer, 2018. ISBN 978-3319635187. https://gameaibook.org.

Appendix A

Contents of the Included Storage Media

The contents of the included storage media are inside a root folder. The structure within this folder is as follows:

- **implementation/** – folder containing source files and code of the game
- **thesis/** – folder containing source files for the text of the thesis
- **build/** – folder containing built binaries for Windows and Linux
- **README.md** – file containing directory descriptions, compilation instructions and controls of the game
- **showcase.mp4** – video file showcasing the implemented result
- **thesis.pdf** – file containing the compiled text part of the thesis