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

MULTIPLAYER GAME IN GODOT INSPIRED BY STRONGHOLD

HRA PRO VÍCE HRÁČŮ INSPIROVANÁ STRONGHOLDEM V GODOTU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

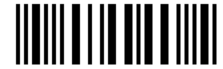
AUTHOR
AUTOR PRÁCE
MAREK KOZUMPLÍK

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. TOMÁŠ MILET, Ph.D.

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment



170533

Institut: Department of Computer Graphics and Multimedia (DCGM)
Student: **Kozumplík Marek**
Programme: Information Technology
Title: **Multiplayer Game in Godot Inspired by Stronghold**
Category: Computer Graphics
Academic year: 2024/25

Assignment:

1. Study the development of games, game design, game mechanics, and the Godot game engine. Explore techniques for multiplayer games and real-time strategy games. Get inspired by games like Stronghold and Hinterland.
2. Design a multiplayer game using a combination of game genres.
3. Implement the designed game in the Godot engine.
4. Test the game from a user perspective.
5. Evaluate the implemented game, write conclusions, suggest expansions, publish the game, and create a demo video.

Literature:

- Ariel Manzur, George Marques, Godot Engine Game Development in 24 Hours, Sams Teach Yourself: The Official Guide to Godot 3.0 1st Edition, Sams Publishing, March 19, 2018, ISBN-10 0134835093
- Juan Linietsky, Ariel Manzur and the Godot community, Godot Docs, https://docs.godotengine.org/en/stable/index.html
- Patrick Felicia, Godot from Zero to Proficiency (Beginner): A step-by-step guide to code your game with Godot, April 22, 2021, ISBN-13 979-8740433844
- Ernest Adams. Fundamentals of Game Design 3rd Edition. New Riders 2013. ISBN-100321929675.
- Buckland, Mat. Programming Game AI by Example 1st Edition. Jones & Bartlett Learning, 2004. ISBN-10 1556220782.

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

Supervisor: **Milet Tomáš, Ing., Ph.D.**
Head of Department: Černocký Jan, prof. Dr. Ing.
Beginning of work: 1.11.2024
Submission deadline: 31.7.2025
Approval date: 10.7.2025

Abstract
This project focuses on the design and implementation of a 3D multiplayer computer game that combines the mechanics of real-time strategy and action RPG games. The game includes several key systems. The main system is the combat system, which combines magic, physics, and a destructible environment. Another key element is the construction of destructible castles and their buildings using resources. A tile-based system is used for building walls. Resource collection is carried out by both players and AI-controlled worker units, with workers having different work cycles. These resources can then be used for building structures or training guards for castle defense. The game also features other units, such as wild animals or bosses. Players can also obtain equipment by defeating enemies, which improves their statistics and unlocks new abilities. The game was developed in the Godot engine and was continuously tested with multiple players simultaneously.

Abstrakt
Tato práce se zabývá návrhem a implementací 3D počítačové hry pro více hráčů kombinující mechaniky real-time strategií a akčních RPG her. Hra obsahuje několik klíčových systémů. Hlavním systémem je soubojový systém, který kombinuje magii, fyziku a ničitelné prostředí. Dalším klíčovým prvkem je stavba zničitelných hradů a jejich budov pomocí surovin. K výstavbě hradeb je použit dlaždicový systém. Sběr surovin zajišťují hráči i počítačové jednotky dělníků, přičemž dělníci mají různé pracovní cykly. Tyto suroviny lze následně využít ke stavbě budov nebo výcviku stráží pro obranu hradu. Ve hře se dále nachází další jednotky, jako je divoká zvěř nebo bossové. Hráč může také získávat vybavení poražením neptřátel, které zlepšuje jeho statistiky a odemyká nové schopnosti. Hra byla vytvořena v herním enginu Godot a průběžně testována s několika hráči současně.

Keywords
Multiplayer game, 3D, Godot Engine, game development, action, strategy, grid tiling system, destructible environment, building system, combat system, visual effects

Klíčová slova
Hra pro více hráčů, 3D, Godot Engine, vývoj počítačových her, akční, strategie, systém mřížky dlaždic, ničitelné prostředí, systém stavění, soubojový systém, vizuální efekty

Reference
KOZUMPLÍK, Marek. *Multiplayer Game in Godot Inspired by Stronghold*. Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet, Ph.D.

Rozšířený abstrakt

Tato bakalářská práce se zabývá návrhem, implementací a testováním 3D počítačové hry pro více hráčů, která kombinuje herní mechaniky strategií v reálném čase (RTS) a akčních RPG. Hráč ovládá svou postavu z izometrického pohledu, volně se pohybuje po herním světě, sbírá suroviny, staví budovy, rekrutuje jednotky (např. dělníky a stráže) a zapojuje se do dynamického soubojového systému proti ostatním hráčům či počítačem řízeným protivníkům. Dále se práce věnuje obecným konceptům herního návrhu a vývoje s využitím herního enginu Godot.

Cílem projektu bylo vytvořit komplexní herní prostředí, které umožňuje jak budování opevněných měst a hradů, tak i přímou akci v soubojích. Hlavními herními prvky jsou získávání surovin a předmětů ničením prostředí a poražením nepřátel, sloužící k odemykání nových schopností a zvyšování hráčových statistik. Hráč zároveň chrání své město a útočí na základny nepřátel, přičemž se přirozeně kombinují mechaniky strategických a akčních her.

Jádrem celé hry je soubojový systém, který probíhá v reálném čase a klade důraz na fyzikální interakce, ničitelné prostředí a originální způsob tvorby kouzel. Hráč může kombinovat klasické útoky, manipulaci s objekty ve světě a magii. Schopnosti mají rozmanité vizuální i mechanické efekty. Hráč může použít ohnivé koule, provádět sečné útoky mečem nebo ovládat složitější kouzla, jako jsou černé díry, tornáda či telekineze. Tyto schopnosti často využívají prostředí a herní fyzikou.

Systém magie je inspirován hrami Arx Fatalis a Magicka. Spojuje kreslení symbolů pomocí myši, charakteristické pro Arx Fatalis, s principem kombinace run, známým hrou Magicka. Každá schopnost a kouzlo má specifický způsob aktivace: buď se spustí okamžitě, podržením nebo s časovým zpožděním. Kromě toho má stanovenou dobu obnovy a určený typ cíle. Schopnost může mířit na postavu (vlastní, nepřátelskou), ve směru kurzoru nebo na vybraný bod na zemi. V systému jsou implementovány i podmínky jako viditelnost cíle, maximální dosah a kolize. Kouzla navíc interagují s fyzikálními objekty. Například tlaková vlna může způsobit zničení objektů, jako jsou budovy nebo stěny, nebo odhodit předměty či postavy tak, že při dopadu způsobí sekundární poškození.

Fyzika zásadně ovlivňuje podobu soubojového modelu. Například objekt vržený pomocí telekineze, který má dostatečnou hmotnost, rychlost a velikost (například kus zdi nebo část stromu), může způsobit poškození hráčům či NPC jednotkám. Tornádo dokáže vymrštit skupinu nepřátel a zároveň ničit budovy, zatímco černá díra vtahuje objekty a postavy do jediného bodu.

Suroviny hrají klíčovou roli ve všech aspektech hry. Získávají se buď manuálně hráčem, nebo pomocí dělníků řízených počítačem, kteří mají přidělené pracovní cykly. Dělníci se dělí na tři hlavní typy: sběrači, farmáři a řemeslníci. Každý typ vzniká v určité budově a pracuje automaticky. Například těžbu surovin, jejich zpracování a následné přenášení do skladů.

Vedle souboje je druhým pilířem hry budovatelský systém. Každý hráč má možnost budovat vlastní město, skládající se z obytných, produkčních a obranných budov. Výstavba probíhá na mřížkovém systému a respektuje kolize s prostředím, ostatními strukturami a jednotkami. Mezi dostupné budovy patří například těžební tábory, farmy, mlýny, kovárny, sklady, tréninkové tábory, věže, hradby a brány. Důležitou součástí systému stavění je automatické spojování hradeb v mřížce tak, aby podle konfigurace sousedních buněk vytvářely vizuálně i funkčně ucelenou strukturu. Budovy i hradby mají vlastní životy a lze je ničit. Některé slouží k tréninku strážných jednotek (např. lučištníci, kouzelníci, rytíři), jiné zajišťují produkci nebo skladování surovin.

Systém frakcí umožňuje hráčům vytvářet aliance, sdílet zdroje a území. Každá frakce má vizuální identifikaci (barvu) a její členové jsou navzájem přátelští. Interakce mezi frakcemi ovlivňují chování jednotek a efekt kouzel (např. léčivá vlna působí pozitivně na spojence, ale škodí nepřátelům). Každá entita ve hře má přiřazenou frakci (od hráčů přes budovy po počítačem řízené jednotky.

Ve hře jsou přítomny i nepřátelské frakce řízené počítačem, jako jsou bandité, zvířata nebo bossové. Tito protivníci mají vlastní bojové styly a kombinace kouzel, které odpovídají jejich typům. Rytíř může mít například seknutí meče, omráčení a bodnutí, zatímco kouzelník kombinuje ohnivé koule, zmražení a meteority. Počítačem řízené jednotky používají podobný soubojový systém jako hráči, včetně dosahu kouzel a systému zorného pole.

Jednou z inovativních funkcí hry je možnost vytvářet a ukládat vlastní hrady, včetně rozmístění budov, jednotek a chování řízeného umělou inteligencí. Takto vytvořené hrady lze následně načítat ze souboru jako nepřátelské základny, přičemž je možné jim přiřadit libovolnou frakci, například pro hru lokálně proti umělé inteligenci. Tento systém výrazně zvyšuje znovuhratelnost a podporuje tvorbu komunitního obsahu.

Hra je vyvíjena převážně pomocí otevřených technologií jako je Godot Engine, Blender a Krita a z velké části je tvořena vlastními assety (3D modely, vizuální efekty a podobně) Hra byla průběžně testována ve vícero hráčích a optimalizována pro běh v síťovém prostředí. Návrh umožňuje snadné rozšiřování systému o nové typy kouzel, jednotek, budov nebo frakcí bez nutnosti zásahu do základního herního jádra.

Multiplayer Game in Godot Inspired by Stronghold

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Tomáš Milet, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.
I have used ChatGPT to improve the formulation of the text and to correct grammatical errors.

. . . .
Marek Kozumplík
July 31, 2025

Acknowledgements

First and foremost, I would like to thank Ing. Tomáš Milet, Ph.D. for his help in solving the problems related to this work, as well as for his positivity and motivation to keep moving forward with the project. I would also like to thank my family for their support throughout my studies, without which I would not have made it to this bachelor's thesis. Last but not least, I would like to thank my roommate for his unwavering mental support and patience.

Contents

1 Introduction **5**

2 Fundamental Concepts of Game Design **7**

3 Game Development with Godot Engine **11**
3.1 Core Runtime Systems . . . 11
3.2 Scene Structure and Communication Patterns . . . 12
3.3 Physics and Collisions . . . 13
3.4 World Building . . . 14
3.5 Artificial Intelligence . . . 15
3.6 Visuals . . . 15
3.7 User Interfaces . . . 16
3.8 Persistence and Serialization . . . 17
3.9 Debugging and Logging . . . 17
3.10 Multiplayer Components and Architecture . . . 17

4 Design of Core Game Systems **19**
4.1 Character Control and Movement . . . 19
4.2 Combat System . . . 20
4.3 Faction System . . . 22
4.4 Building System . . . 22
4.5 Worker System . . . 23
4.6 Guard Units . . . 23
4.7 NPC Combat System . . . 23
4.8 Items and Equipment . . . 24
4.9 The Game World . . . 25
4.10 Win Condition . . . 25
4.11 Game Chat . . . 25

5 Market Research **26**
5.1 Main Inspirations . . . 26
5.2 Other Similar Games and Inspirations . . . 27

6 Implementation **28**
6.1 Multiplayer Architecture . . . 28
6.2 Game World and Entities . . . 29
6.3 Entity Structure: Players, NPCs, and Buildings . . . 29
6.4 Camera . . . 30

6.5 Character Controls . . . 32
6.6 Character Skeleton Animations . . . 33
6.7 Combat System . . . 35
6.8 Visual Effects . . . 43
6.9 Building System . . . 47
6.10 Non-playable Characters . . . 60
6.11 NPC Combat System . . . 63
6.12 Workers . . . 64
6.13 Navigation . . . 67
6.14 Items and Inventories . . . 69
6.15 The World . . . 74
6.16 Game Chat . . . 77

7 Testing **78**
7.1 Main Complaints from User Testing . . . 79

8 Conclusion **81**

Bibliography **82**

A Contents of the included archive **83**

B Used Assets **84**

C Used Tools **88**

D Used Addons **90**

List of Figures

1.1 Screenshot of the current game prototype featuring workers, buildings and a castle. 6

6.1 Third-person camera setup using a SpringArm3D and Camera3D. The spring arm automatically shortens to avoid clipping through obstacles. The pivot allows rotation. 30
6.2 Raycasting from the camera through the cursor to determine the 3D position. 31
6.3 Annotated player node structure for movement and environment detection. 32
6.4 Using the AnimationPlayer node to call functions through a call method track. 34
6.5 Overview of the modular system implementation, showing interactions between subsystems. The lower section represents the game scene, while the upper section depicts abstract systems (scripts) attached to nodes in the scene. 35
6.6 User interface visuals with an example input for casting special abilities, using Particles2D, Label, and Line2D nodes. 39
6.7 Selection of the target is based on the closest distance to the mouse cursor. 40
6.8 Target representation in game using name label and quad mesh. 40
6.9 Overview of line of sight calculation for targeting. 41
6.10 Target line of sight excluding other entities with the same collision layer. . . 41
6.11 Combat UI showing player and target information, a cast bar, and ability buttons with keybinds and cooldowns. 42
6.12 Example of a VFX scene inside the Godot editor, showcasing billboard and trail particles with scrolling textures animated using the AnimationPlayer. . 44
6.13 Seamless VFX texture made in Krita applied to cylinder mesh inside Blender. 45
6.14 Example of VFX scenes inside the Godot editor using billboard and trail particles with scrolling textures animated by the AnimationPlayer. 46
6.15 Simplified class diagram of the building system and its subsystems, including constant data. 47
6.16 Comparison between a non-centered and a centered matrix with the center point indicated. 48
6.17 Screenshot of the building menu user interface. 49
6.18 Placement cursor with price label and collider. Blue collider indicates allowed placement while red indicates that placement is not allowed. 50
6.19 Collision array showing overlapping physics bodies for a line shaped selection. 51
6.20 Before and after using the wall editing system to remove structures and walls. 52
6.21 Each individual 3D tile mesh made in Blender. 53
6.22 Tile shape is determined by the configuration of orthogonal (green) and diagonal (purple) neighbors used in the tiling algorithm. 54

6.23 Example of tiling before and after adding a tile at the cursor position. . . . 54
6.24 Difference between non-diagonal and diagonal tiling. 55
6.25 Tile health visualized by vertical offset after ability impact. 55
6.26 Visual difference between tiling with and without towers and gates included. 57
6.27 Woodcutter building scene made in Godot Engine. 58
6.28 Guards spawned using the NPC spawner on ground, walls, and gates. . . . 60
6.29 EntityGenerator configuration and its runtime behavior. 61
6.30 Guard behavior state machine with parameters. The spawn point is a 3D global position where the entity is instantiated, idles, and returns to. 62
6.31 NPC combat detection using `Area3D`, pathing with navigation agents and line of sight, and attack range of the current ability. 62
6.32 Game screenshot of worker buildings placed in the settlement. 64
6.33 Gatherer behavior defined using state machine. 65
6.34 Farmer behavior defined using state machine. 66
6.35 Artisan behavior defined using state machine. 66
6.36 Baked navigation mesh displayed as blue mesh in the game's debug build. . 67
6.37 Illustration of batching rebake requests showing multiple construction actions within a short time frame and a single navigation mesh rebake occurring after the timer expires. 68
6.38 Screenshot of instanced items in the game world. 70
6.39 Stone item instances with different count: small stack size on the left and full stack on the right. 70
6.40 Screenshot of the inventory user interface showing example items and equipment 72
6.41 Screenshot showcasing three different equipped hats and weapons. 73
6.42 Destroyed trees with wood items instanced nearby. 74
6.43 Stone deposit with stone items dropped after damage. 74
6.44 A 3D terrain mesh in made Blender and textured using a splatmap shader. 75
6.45 Foliage rendered with a MultiMesh using the Simple Grass Texture addon. 76
6.46 Generated paths using Proton Scatter and Stylized Nature Mega Kit assets. 76

Chapter 1

Introduction

In recent years, video game development has become significantly more accessible, largely due to the increasing availability of free and open-source software. Game development, which once required substantial financial investment and large development teams, is now accessible to individual developers and small independent teams thanks to the availability of free and open-source tools. These tools are not only free to use, but are also supported by active communities that continually contribute improvements, documentation, and tutorials. The **Godot Engine** reflects this trend, offering a fully open-source codebase and robust support for both 2D and 3D development. While relatively niche at the start of this work, it has since expanded significantly and is now widely used by independent developers.

The goal of this thesis is to draw inspiration from titles such as *Stronghold* and *Hinterland*, and to study the fundamentals of game design, gameplay mechanics, multiplayer systems, and the technical capabilities of the Godot engine, while also addressing game design, implementation, user testing, evaluation, and the distribution of a working prototype.

This work presents the development of a 3D multiplayer game that combines two genres: *real-time strategy (RTS)* and *action role-playing games (ARPG).* The aim was to explore how RTS mechanics can be merged with the fast-paced, character-focused gameplay of ARPGs in a multiplayer environment.

The resulting game allows players to build and defend castles, gather and manage resources, and engage in real-time combat using melee, ranged, spellcasting, and physics-based abilities. Combat features destructible structures and trees, with abilities that have unique effects and targeting rules, used by both players and AI-controlled enemies. Players can form factions, train units, and choose to defend their territory or attack opponents. Character progression is achieved by collecting equipment and unlocking new abilities. Core RTS systems include resource collection through worker units or combat, and a tile-based construction system with dynamic wall connections.

All aspects of the game, from gameplay logic to 3D models and user interface, were developed primarily using open-source tools. **Godot Engine** served as the main engine and scripting environment, **Blender** was used for modeling characters and objects, and **Krita** was used for creating textures and interface graphics. Multiplayer functionality was implemented using Godot's built-in networking features and was continuously tested during development to ensure performance, balance, and stability.

This work demonstrates the potential of free and open-source software to support the creation of technically ambitious games by individual developers. It examines how diverse gameplay genres can be effectively combined and how multiplayer systems can be

implemented using accessible tools, while also serving as a case study in independent game development from concept to complex working prototype.

The remainder of this thesis is structured as follows. Chapter 2 introduces core game design concepts. Chapter 3 discusses development practices using the Godot engine. Chapter 4 presents the key systems and architecture of the game. Chapter 5 explores similar games and the current indie market. Chapter 6 outlines the technical implementation. Chapter 7 presents testing methods, results, and user feedback. Chapter 8 summarizes the outcomes and discusses possible future improvements. Figure 1.1 below shows a screenshot of the current working prototype of the game.

Figure 1.1: Screenshot of the current game prototype featuring workers, buildings and a castle.

Chapter 2

Fundamental Concepts of Game Design

Game design is the process of creating the rules, systems, and experiences that guide how players engage with a game. A well-designed game introduces clear mechanics and meaningful challenges, while also encouraging long-term interest through progression, social interaction, and replayability.

This chapter outlines key elements of game design, based on established principles from game design literature *The Art of Game Design: A Book of Lenses* [9] and *Rules of Play: Game Design Fundamentals* [8], as well as personal observations from both development and gameplay experience. It covers core concepts such as game loops, objectives, rules, and system design, with specific considerations involved in designing multiplayer games.

Game Loop

The core game loop is set of fun actions or activities that player repeatedly executes while playing the game. Often with slight variations for each cycle. It is the backbone of the game that keeps the player engaged and motivates him to continue playing. In this project, the core game loop involves exploring the world, defeating enemies, gathering items, and becoming stronger to explore more challenging areas and increasingly powerful enemies.

Objectives

Objectives are specific goals or tasks that players work to complete in a game. They give players a sense of direction and purpose.

Objectives can be short-term like defeating an enemy or long-term like finishing the story. They include main goals and optional challenges to keep the game engaging.

In this project, short-term tasks include defeating enemies, cutting trees, and mining resources. Long-term goals involve building a castle, collecting gear to unlock spells, and destroying enemy castles to win.

Rules

Rules in games set the boundaries for how players interact with the game world, other players, and the game systems. They affect the decisions players make, the strategies they develop, and the overall way the game is played.

In this game, the rules cover resource gathering, building placement, combat mechanics, and faction behavior. For instance, players must collect materials to build structures on a grid, manage stamina-limited movement for their hero, cast spells with cooldowns and range restrictions and faction alignment influences interactions.

Progression Systems

Progression systems are associated with the long term objectives, giving players a feeling of growth and goals to work toward. While common in RPGs, many other games have similar features like experience points, upgrades, or unlockable rewards.
Progression typically falls into two types:

- **Vertical progression:** Players become stronger or more powerful over time by gaining better stats, abilities, or gear. This type often affects gameplay balance and is common in RPG games.
- **Horizontal progression:** Players unlock new content, abilities, or cosmetic items that offer variety without increasing power or effectiveness. This type maintains game balance while still rewarding players and keeping them engaged.

Linear Scripted Games vs Games as Systems

Linear games lead players through a set series of events or levels, focusing on storytelling and intentionally scripted experiences. System-based games, on the other hand, rely on interacting systems that create gameplay through the way mechanics, players, environments, and AI work together.

This project is a sandbox game that offers mostly open-ended gameplay with only some direction. It is a system-based game, relying on interacting systems to create dynamic and emergent experiences.

Emerging Complexity

Phenomenon where simple game rules or mechanics can lead to deep and unpredictable gameplay through player interactions with the game's systems. In multiplayer games, this phenomenon often emerges when basic mechanics combine in unexpected ways. This complexity is not directly designed but arises naturally and helps keeping players engaged by encouraging new strategies over time.
Examples include:

- Rocket jumping in *Team Fortress 2*
- Camp stacking in *Dota 2*
- Body-blocking

Multiplayer Game Design

Designing multiplayer games introduces different challenges than single-player games in both design and technical implementation, including more complex systems and stricter balancing.

On the upside, multiplayer games have clear advantages like more replayability from unpredictable player interactions and stronger social connections. Additionally, deep and well-made systems, especially in PvP games, keep players interested without constantly needing new content updates.

Types of Multiplayer Games

Multiplayer games come in various forms, each offering distinct gameplay experiences and social interactions. Common types include:

- **Cooperative (Co-op):** Players work together to face computer-controlled enemies and challenges presented by the environment.
- **Competitive:** Players are grouped into teams, with each team competing against the others in a shared game world.
- **Massively Multiplayer Online (MMO):** Large-scale games that support hundreds or thousands of players simultaneously in persistent worlds, often combining social interaction, progression, and open-ended gameplay.
- **Party Games:** Focused on casual experiences emphasizing fun and social interaction over deep complexity or competitiveness. often played in groups where everyone can join in without needing serious skill or commitment.

Balancing

Balancing games involves a compromise between keeping the game fair and preserving the fun. Too much emphasis on balance can sometimes reduce enjoyment. On the other hand, no single strategy should significantly outperform the others. A good example are games that use seasonal updates to tweak game systems and shift meta strategies. This refreshes gameplay without needing a constant updates with new content.

Replayability and Player Retention

Multiplayer games are naturally more replayable, as discussed in section 2. However, there are several additional design strategies that can further enhance replayability and support long-term player retention:

- **Skill-based matchmaking:** Fair and competitive gameplay, reducing frustration and keeping players engaged by consistently providing appropriately challenging opponents.
- **Progression and rewards:** Systems such as ranks, levels, unlockable cosmetics, or achievements give players goals to pursue beyond a single match.
- **Seasonal updates and balance patches:** Regular adjustments to game balance or mechanics can refresh gameplay without requiring entirely new content, making familiar systems feel new and engaging.
- **Variety in game modes and maps:** Offering different ways to play prevents the experience from becoming repetitive.
- **Social systems:** Features like clans, friend lists, chat, or team-based which increases emotional investment in the game.
- **User-generated content:** Tools that allow players to create and share custom maps or other content can greatly extend the game's lifespan by introducing new experiences.
- **Emerging Complexity**: as explained in Section 2.

Matchmaking and Server Browsers

Matchmaking automatically pairs players based on things like skill, region, or game mode. It helps set up balanced games quickly without needing players to search manually.

A server browser lets players pick from a list of servers, using filters like ping, player count, or game type. It offers more control but can be slower and harder to use, especially for new players.

Matchmaking is more common in modern multiplayer games, especially in fast-paced genres like shooters and MOBAs. That said, server browsers offer unique benefits. They help build small communities since players often return to the same servers. This can lead to friendlier interactions and more accountability, unlike the more anonymous feel of matchmaking.

Chapter 3

Game Development with Godot Engine

Game development combines programming, design, art, and system architecture to create interactive experiences. It includes many skills beyond coding and game design, such as 2D and 3D art, animation, visual effects, music, sound design, and storytelling. A game is, at its core, is a real-time application that reacts to player input, simulates behavior, and displays the results.

This chapter focuses on the key building blocks of game development and game engines from a technical perspective that are relevant for this project. All concepts are presented using the Godot Engine as the primary tool, but the principles apply broadly across modern game engines. The content is based on established sources, including *Programming Game AI by Example* [1], *Game Engine Architecture* [3], and the *Official Godot Documentation* [2], as well as personal observations and experience gained during development.

3.1 Core Runtime Systems

The core runtime systems of a game engine are responsible for managing the real-time behavior of a game, including how it updates, renders, and responds to input.

The Game Loop

The game loop is a continuous cycle of operations that processes input, updates the game state, and renders frames. This cycle repeats many times per second, usually at the same rate as the display, which allows the game to respond in real time.

In Godot, this loop uses built-in functions like `_process` and `_physics_process`, called automatically each frame. These functions can be defined in any node with an attached script and are automatically called by the engine each frame.

- `_process(delta)` runs every rendered frame and handles non-physics tasks such as input polling.
- `_physics_process(delta)` runs at a fixed rate (usually 60 times per second) for physics logic like movement and collisions, ensuring stable behavior.

Delta Time

In games and real-time applications, frame rates vary with system load and performance. To keep movement and other time-based behavior consistent, delta time is used. Even physics frames, which run at a fixed rate, can have slight differences in time between frames.

Delta time is the time passed since the last frame. In Godot, it is the `delta` parameter in `_process(delta)` and `_physics_process(delta)`. Multiplying changes like movement or animation by `delta` ensures frame rate independent behavior.

Input Handling

Input handling is a core part of interactive applications, especially in games, where it allows players to control characters, navigate interfaces, and trigger actions.

In game development, input is typically abstracted into named actions such as `jump` or `shoot` rather than checking key codes directly, allowing for control scheme flexibility and remapping. In Godot, this is handled exactly the same way through an action-based system where input actions are defined in the `Input Map` and mapped to keys, mouse buttons, or gamepad inputs.

3.2 Scene Structure and Communication Patterns

Game architecture relies on structuring logic into modular units and enabling clean communication between them. Godot supports this through its node-based scene system and signal-driven messaging.

Modularity and Component-Based Design

Modularity in game development means dividing systems into small, reusable parts. This help manage complex systems, reduces duplication, and separates specific features.

Component-based architecture builds game objects from parts like movement or input. For example, Unity uses components like `Transform`, `Collider`, and custom scripts.

Godot uses a similar system based on scenes and nodes. Each scene is a tree of nodes, each handling a task like timing or animation using `Timer` or `AnimationPlayer` nodes. Scenes in Godot are modular and can be reused or extended. Scripts can be added to define custom logic by inheriting the node's class, and values can be exposed with `@export` annotation to edit these values inside the editor.

Event-Driven Development

In event-driven systems, components react to events instead of constantly polling conditions. This is useful for handling asynchronous actions like UI, or gameplay triggers, such as collision checks.

Godot uses a signal system for event-driven logic. Nodes can emit signals, either built-in by the engine or custom. For example, `Button` can emit a `pressed` signal, and a health system might define `health_changed`. Signals can be connected to callback functions of any node within the game scene, allowing decoupled communication. For instance, a UI health bar can update itself when a health component emits a signal. Connections can be made in the editor or by code using `connect` function. `Timer` nodes also use signals, such as `timeout`, which can trigger delayed or repeated actions.

Global Patterns

Godot provides global patterns to organize and manage game logic across the project, making shared data and objects easier to handle. These include:

- **Singletons:** Also known as Autoloads, these are scripts or nodes loaded once at game start and remain globally accessible. Used for management of game state, or systems like audio control and settings.
- **Groups:** Named collections of nodes allowing collective management of objects sharing similar roles or behaviors.
- **Global Enums:** Created using the `class_name` keyword to avoid duplication of IDs or constants across scripts. They ensure consistency and avoids code duplications.

3.3 Physics and Collisions

Collision detection is a core part of most games. It is used to determine when objects collide and stop them from overlapping. It is essential for gameplay elements such as movement, combat, physics reactions, and triggering events.

- **Bodies:** Physics bodies represent objects that interact physically in the world. `RigidBody3D` has built-in physics behavior defined by the engine, responding automatically to forces and collisions. `CharacterBody3D` requires custom movement logic defined by the developer. `StaticBody3D` is typically used for static environment elements.
- **Areas:** Special regions that detect when objects enter, exit, or stay inside. They do not block movement but are used to trigger events or overriding physics parameters inside.
- **Collision Shapes:** Define the physical space of an object for collision. Shapes can be primitives (boxes, spheres, capsules) or custom polygons. They can be attached to physics bodies or areas.
- **Layers:** Control which objects can collide. Each object can belong to multiple layers and detect collisions with selected ones. This allows organization into categories such as players, enemies, projectiles, and environments.
- **Shapecasting and Raycasting:** Techniques used for detecting intersections. Raycasting projects an invisible line to check for hits, useful for shooting or visibility checks. Shapecasting sweeps a shape (e.g., box or capsule) to check if it collides with anything, often used to test movement paths.

3.4 World Building

This section outlines the methods used in this project to construct 3D worlds. Other important systems, such as audio and lighting, are also supported by Godot and utilized in this project.

Tiling and Tilemaps

Tiling is a method of building environments by arranging small reusable elements called tiles into a seamless grid.

This project uses a square tiling system where each tile is selected based on the state of its eight directions including diagonals. This logic determines which mesh to place at a given grid location, allowing for tile transitions and edge blending using predesigned 3D mesh variants.

Godot supports 2D tilemaps using with features like autotiling and collisions. It also includes 3D tiling via `GridMap` and `MeshLibrary`, but this lacks the flexibility needed for custom behavior in this project. This project uses a simple square tiling system selecting tiles based on their eight neighbors to place appropriate 3D castle wall meshes.

Terrain Systems

Terrain systems provide the foundation for large-scale environments. They are commonly implemented using heightmaps or voxel-based solutions with features like LOD, collision, and runtime editing. While Godot does not have built-in terrain system, third party addons exist with navigation and LOD features. However, they are not well suited for this project because of multiplayer.

In this project, the terrain is implemented as a static 3D mesh modeled in Blender and rendered using a custom splatmap shader.

Navigation

Navigation systems allow characters or entities to move intelligently through the world. Godot provides navigation nodes like `NavigationRegion3D` and `NavigationAgent3D` to define walkable areas and path-finding logic. Agents can automatically calculate paths, avoid obstacles and other agents, and dynamically update their routes as needed. The implementation and its optimization are covered in Section 6.13.

3.5 Artificial Intelligence

Artificial Intelligence (AI) in games enables non-player characters (NPCs) to simulate life-like behavior.

Finite State Machines

Finite State Machines (FSMs) are a fundamental concept in computer science, providing a formal model for systems with a limited number of discrete states and well-defined transitions between them. They are widely used in large number of computer science fields such as digital circuit design or language parsing. In game development, FSMs are especially useful for programming the behavior of AI characters, allowing agents to switch between states like patrolling, chasing, or attacking in response to game events. This approach is used in this project to create AI characters, and the specific state machines are discussed in Section 6.10. The concept and practical implementation details of FSMs in game AI are also extensively covered in Mat Buckland's book Programming Game AI by Example [1].

Additional Concepts in AI Systems

- **Behavior Trees:** A modular and hierarchical alternative to FSMs, commonly used for managing complex AI behaviors through a tree of tasks and conditions.
- **Dialog Trees:** Used to structure branching conversations between the player and NPCs, allowing for dynamic dialogue based on player choices or game state.
- **Perception Systems:** Enable AI to detect and respond to the environment using raycasting, line-of-sight checks, or proximity-based triggers.
- **Navigation:** As discussed in its subsection 3.4.

3.6 Visuals

This section covers the major visual systems used in this project, including modeling, shaders, effects, animation, and camera control.

3D Modelling

3D modelling is the process of creating digital objects and environments for the game world. Blender is commonly used with Godot since it is also free and open source. Exporting models in the glTF format is recommended, as it supports materials with textures, colors, UV maps, roughness, and other surface details. Polygon count is important in real-time applications like games to ensure good performance.

Shaders

Shaders control how surfaces are rendered by manipulating geometry and pixel colors to achieve various visual effects. Godot provides its own built-in shading language, similar to GLSL, along with a Material resource that exposes shader settings through the editor interface. The two main types of shaders are:

- **Vertex Shaders:** Handle the manipulation of vertex data, including transforming coordinates and preparing information for the next rendering steps.

- **Fragment Shaders:** Calculate the color and shading of individual pixels, applying lighting and texture effects to create the final visual output.

Visual Effects

Visual effects (VFX) provide visual feedback of a game and are usually composed of multiple elements working together, such as:

- **Materials and Shaders:** Define surface appearance and lighting, including vertex and fragment shaders that control geometry and pixel colors.
- **Scrolling Textures:** Simulate movement on surfaces, like flowing water or moving clouds.
- **Particle Effects:** Used for dynamic elements like fire, smoke, magic spells, and other transient visuals.
- **Billboards:** 2D images that always face the camera, commonly used for effects like fire or light glows.
- **Mesh Effects:** Transform meshes using animation systems to create complex visuals like shockwaves or dissolving objects.

In Godot, these visual effects are implemented using a combination of shaders for custom shading, particle nodes for particle effects, and `AnimationPlayer` nodes that control animated properties like material parameters, mesh transforms, and visibility over time. More information about visual effects and the implementation of specific can be found in the VFX section (6.8) of the implementation chapter.

Animation Systems

Animation systems control changes in properties over time. In Godot, the `AnimationPlayer` node manages multiple animations using keyframes and can modify any property or call functions on nodes during playback, making it useful beyond just visual effects.

Camera Systems and Viewports

Camera and viewport systems define how the game world is presented to the player. In Godot, the `Camera3D` node provides control over the viewing perspective, including field of view (FOV), position, and post-processing effects. `Viewport` nodes enable rendering to textures, which is useful for minimaps, reflections, or split-screen views. Cameras can also be animated or dynamically switched for cinematic sequences.

3.7 User Interfaces

User interfaces in games handle things like menus, inventories, health bars, and dialogue. They let players interact with the game and access important info.

In Godot, UIs use the scene tree like everything else, built with `Control` nodes. Containers handle layout, anchors manage positioning, and built-in elements include buttons, labels, and icons. For 3D games, nodes like `Sprite3D` and `Label3D` display UI in the world with the ability of facing the camera using the billboard setting.

3.8 Persistence and Serialization

Serialization systems allow the game to store and restore state between sessions. This includes player progress, world data, and settings. In Godot, data can be serialized using the built-in `File` and `JSON` APIs, or more advanced systems like `ConfigFile`, `ResourceSaver`, and custom dictionaries. Nodes can export relevant properties, which are written to disk and later reloaded to restore game state.

3.9 Debugging and Logging

Debugging and logging are essential parts of development, helping developers identify errors, inspect runtime behavior, and monitor performance. Godot provides several tools for both visual and code-based debugging:

- **Debug Build Mode:** Running the game in debug build allows rendering of additional visuals such as navigation, paths, avoidance, and collision shapes.
- **Built-in Debugger:** Activates on errors and warnings. Includes tabs like Errors, Warnings, Monitors, Network Profiler, and Multiplayer, showing memory, node activity, performance (including CPU and GPU ms time per function), and network events in real time.
- **Visual Debugger:** Displays function call names along with CPU and GPU time usage to help profile performance. It provides real-time insights into which parts of the code consume the most resources.
- **Network Profiler:** Provides tools to diagnose synchronization issues, showing synchronized nodes, active peers, and RPC calls. Useful for managing multiplayer state and communication.
- **Manual Logging and Breakpoints:** `print` outputs custom messages to the console for basic debugging. `push_error` and `push_warning` send flagged messages to the debugger tabs without halting the game. Breakpoints can be set in the script editor to pause execution at specific lines during runtime.

3.10 Multiplayer Components and Architecture

Multiplayer development introduces additional complexity beyond single-player games by requiring multiple connected instances to maintain a shared synchronized game state. In most cases, this involves division between server-side and client-side logic, along with networking systems that handle communication, replication, and ownership. This makes development more complex and often much slower than building a single-player game.

Architecture Models

Godot supports both common multiplayer architectures:

- **Client–Server:** A central server maintains the game state, while clients send input and receive updates. This model is commonly used in competitive or authoritative scenarios.
- **Peer-to-Peer:** One of the peers also acts as the server, and other players connect to it. This approach is simpler but harder to scale and less secure.

Even in peer-to-peer setups, one peer typically takes on the authoritative role. In Godot, roles are managed using peer IDs and the local authority concept to control which peer owns and synchronizes each node.

Godot 4 Multiplayer Model

In cotrast to Godot 3, Godot 4 allows both server and client logic to coexist within a single project. The engine provides built-in checks to differentiate between server and client at runtime, so scripts can branch logic as needed. This avoids duplicating projects and simplifies multiplayer development.

Synchronization and Replication

Multiplayer synchronization in Godot depends on all peers having the same scene tree structure. Nodes that are meant to synchronize must exist at identical paths on all instances. The engine uses Remote Procedure Calls (RPCs) and synchronized variables to propagate state changes.
Godot 4 also provides dedicated nodes to support this:

- **MultiplayerSynchronizer:** Syncs variables between peers. It can synchronize on value change, on spawn, or at fixed intervals. Authority determines which peer controls the data.
- **MultiplayerSpawner:** Monitors a part of the scene tree and ensures spawned nodes are replicated across all clients. Useful for managing enemies, projectiles, or any dynamically created object.

Scene Tree and Ownership

Reliable multiplayer behavior requires that each synchronized node is present in the same location within the scene tree on all peers. Ownership must be assigned correctly so that only the designated peer controls a node's logic or state updates.

Chapter 4

Design of Core Game Systems

This project is a 3D multiplayer game combining real-time strategy and action RPG mechanics. Players control a hero character from an isometric perspective to gather resources, build settlements, and engage in spell-based combat against AI and other players. The target audience includes both competitive and casual players of multiplayer action RPGs, MOBAs, RTS, and city-building genres. The target platform is Windows and Linux operating systems.

The core systems include:

- **Combat system:** Combat is fast-paced and real-time, focusing on dynamic spell-casting, physics-driven interactions, and destructible environments.
- **Equipment progression:** Gear is obtained by defeating AI-controlled enemies and bosses, improving character stats and unlocking new spells and abilities.
- **Settlement building:** Players create fortified settlements by constructing buildings, walls, and towers to defend against enemy factions.
- **Resource management:** Resources are gathered by the player-controlled hero and AI-controlled units such as workers and soldiers.
- **Production system:** Buildings spawn worker units that collect materials, convert them into refined resources, and return them to storage.
- **Unit training:** Players can train defensive units like archers, mages, and knights to protect their settlement.
- **Multiplayer and AI interaction:** Players compete against both AI and other players in real time, balancing offense, defense, and economic growth.

4.1 Character Control and Movement

The game features a **physics-based movement system** that enables realistic interactions with the environment, characters, and physical objects. Player feedback led to implementation of WASD controls for character movement from the mouse-driven movement in previous version. The character rotates to visually indicate movement direction, and a **stamina system** limits running and climbing. The isometric camera supports zoom and rotation on both horizontal and vertical axes.

4.2 Combat System

The combat system combines **spellcasting** with **physics-driven** interactions. Abilities affect both characters and the environment, including trees, buildings, and other physical objects, creating dynamic and unpredictable outcomes. Damage and healing calculations consider buffs, debuffs, and equipment stats.

Abilities range from simple sword strikes and fireballs to **large-scale spells** like meteors and tornados that alter a target's velocity and rotation, **status effects** such as healing, slowing, or burning, and **physics-based** abilities such as gusts of wind, item throws or black holes that pull in nearby objects.

The full list of currently abilities:

- **Meteor**: Long-cast area spell that deals damage at a target location, including wall tiles and buildings.
- **Tornado**: Instant spell that damages enemies and applies lift and rotation effects to any physical objects it passes through.
- **Flame strike**: Instant strike from above that damages enemies in an area and spawns the `AoE Fire` spell.
- **AoE Fire**: Periodically damages entities in area of effect.
- **Ice Nova**: Freezes nearby enemies by applying a frozen debuff.
- **Frozen debuff**: Block of ice representing being frozen, disables any actions.
- **Heal Totem**: Summons a totem that heals allies over time.
- **Slow**: Applies a slowing debuff to a target enemy.
- **Fireball**: Fires a projectile that damages a single target.
- **Blast**: Area of effect projectile that damages targets it passes through.
- **Speed Buff**: Temporarily increases movement speed.
- **Fire Curse**: Deals damage over time and can spread to nearby enemies.
- **Gust of Wind**: Pushes enemies and items away in a specific direction.
- **Black hole**: Pulls nearby enemies and items toward the caster.
- **Lava Pillar**: Targets an enemy and damages them with a pillar of fire.
- **Heal Nova**: Heals all nearby allies and damages enemies.
- **Slash**: Deals close-range damage in a small area.
- **Slash 2**: Another version of the Slash ability.
- **Stab**: Low range single-target damage ability, treated as a debuff.
- **Leap**: Quickly jumps to a nearby location.
- **Ground Slam**: Stuns and damages enemies when landing.
- **Throw**: Throws a held object that may damage or spread effects.
- **Grip Item**: Grabs and holds a nearby object.
- **Shoot Arrow**: Fires and arrow into a specific location. The curve is calculated based on the speed and the displacement of the target and current location.

Each spell operates independently, with its own casting requirements and internal logic. For example, some single-target spells can become area-of-effect attacks after certain conditions, such as impact or delay.

Some abilities have multiple stages, with each input performing a different action. For example, a telekinesis spell first pulls a nearby object toward the player and then launch it at a target with a second input.

Spell Casting

Each spell has its own cooldown, range, casting logic, and active behavior. Special abilities require players to draw graphs by connecting points with the mouse cursor. The system rewards muscle memory and precise timing rather than repetitive inputs. As described in Section 4.8, spells are unlocked through equipping specific items, allowing players to build unique loadouts.

Targeting Behavior Types:

- **Enemy-targeted:** Selects nearby entities via mouse hover and keybind.
- **Self-targeted:** Automatically affects the caster.
- **Directional:** Casts in the direction of the mouse cursor.
- **Ground-targeted:** Casts at a specific terrain location.

Some spells support both ground and directional targeting but prioritize enemies if one is selected. For example, a sword slash can either swing in a direction of mouse cursor or target.

Casting Requirements:

- **Line of sight:** Some spells require unobstructed view of the target.
- **Range:** All spells have maximum effective distances.
- **Cooldown:** Each ability has a delay before reuse.

Cast types:

- **Casted:** Requires holding the key for a charge duration.
- **Instant:** Casts immediately on input.
- **Channeled:** Repeats while key is held, up to a limit.

Status Effects:

- **Positive:** Beneficial effects like healing, speed boosts, or increased damage. Some are stackable and visually indicated.
- **Negative:** Harmful effects such as stuns, slows, freezes, or damage over time.

Special Abilities

Special abilities are cast by drawing edges between predefined graph points. The system appears when the player presses a specific keybind. Points can be connected while holding a mouse button, allowing for non-continuous drawing and disconnected lines. Neither the direction nor the order of connections is considered, so only the visual shape matters. An example of the visuals is shown in Figure 6.6. This system encourages exploration and progression by allowing players to discover and memorize specific spell patterns, also raising the skill ceiling. The system is blending spell-casting systems from *Arx Fatalis* and *Magicka*.

Physics and Environmental Combat

Some items and objects share physical attributes with players and AI, such as mass, scale, and velocity. When an object exceeds a certain velocity threshold, it becomes dangerous. Collisions with players, buildings, or other objects will cause damage based on its velocity, mass, and scale. The scale is determined by the item's stack count.

Most spells that affect gravity or motion, such as Tornado, Black Hole, Telekinesis Throw, influence not only players and NPCs but also items. This system creates dynamic and unpredictable interactions that add depth to gameplay.

Instead of relying on fixed damage patterns, players are encouraged to adapt, use the environment to their advantage, and respond creatively to unexpected situations.

4.3 Faction System

Factions allow players to collaborate, share resources, and capture territories, with each faction identified by a unique color. Every entity in the game belongs to one faction, which influences gameplay interactions. For example, healing abilities like Heal Nova benefit friendly players but harm enemies, while area-of-effect attacks ignore allies. NPCs automatically attack other NPCs and players from different factions when nearby, without player input.

4.4 Building System

The building system uses a grid-based placement approach for structures. There are two main structure types: **buildings** and **walls** which connect seamlessly using an automatic tiling system. All construction takes place within the boundaries of a settlement . Settlements act as bases and define the area where players are allowed to build. Each settlement belongs to a specific faction and contains a king NPC. All buildings, walls, towers, workers, and guards are linked to a specific settlement. They can be saved to and loaded from a save file. Players can construct various types of buildings, each serving a specific role:

- **Fortifications**: Defensive structures where units such as archers can be stationed. These include **Walls**, which automatically tile in a 2D array to create seamless barriers, **Towers**, and **Gates** which players and NPCs can walk through.
- **Gathering and Production Buildings**: Produce or convert resources by spawning a specific worker type, which simulates working cycles. Examples include **Mines**, **Farms**, **Crop Fields**, **Mills**, **Woodcutters**, and **Blacksmiths**.
- **Storage**: Serves as the central inventory for the settlement. All construction costs are deducted from this storage before drawing from the player's personal inventory. Workers collect and deposit resources here. Like any other building, it can be destroyed, causing all stored items to be dropped on the ground.
- **Residental Houses**: Increase population capacity, which is required to build other structures and raise an army.
- **Army Buildings**: Used to train and house guards like archers, mages, and knights who defend the castle. These include **Guards**, defensive units that require a corresponding training facility, and **Training Camps**, which differ based on unit type.

Each structure requires specific resources as detailed in Subsection 4.8 and must be placed without overlapping existing objects. All structures and walls have health values and can be damaged or destroyed.

4.5 Worker System

The worker system consists of three distinct categories:

- **Resource Gatherer**: Collects raw materials from the environment.
- **Farmer**: Produces food or wheat.
- **Artisan**: Transforms raw materials into refined goods or items.

Each worker is assigned to and spawned by a specific building, and inherits the faction affiliation of its settlement.

Workers follow defined behavior patterns, cycling through multiple states: **idle**, **working**, **moving to target**, **storing resources**, and **combat** (if provoked). They operate on production cycles, gathering or producing resources at regular intervals. The system includes automated resource storage and management. When storage buildings are available, workers will automatically deposit gathered or produced items there.

4.6 Guard Units

Guard units are defensive troops that players place at specific spots in their castle using unit spawners. Once set, they protect their area by chasing any intruders within a certain range. When an enemy gets close, they automatically attack, keeping the settlement safe even if the player is not actively controlling them. The chase and return distances depend on the unit type. The same unit types also work as workers but switch to defense mode if attacked.

Other NPCs

The guard behavior is also applied to bandits, as well as wildlife and bosses scattered throughout the map, allowing them to respond to nearby threats.

4.7 NPC Combat System

Each non-player character, has a specific combat type. This combat type defines one or more sets of ability combinations. NPCs can use the same spells as players, but their cast times, ranges, health, and power values are different.

When an NPC targets an enemy, it moves closer until it reaches the spell's required range. Once in range, the NPC starts casting. A casting bar appears above its head, similar to a health bar.

The NPC selects a sequence of abilities based on its combat type. It casts all spells in the sequence one after another. Between spells, it may pause to reposition for the next ability. Each combat type has one or more of these sequences.

For example, a knight's sequence might include a melee slash, a stun, and a stab. An archer may only have one ability: shooting an arrow. A wizard boss can have several sequences. One might include a tornado and meteor. Another might use ice nova and damage spells or push enemies away and heal nearby allies.

NPC Spawners

An NPC spawner is placed at a specific location within the game world. The spawner is responsible for creating enemies and it assigns key parameters to these units, such as their 3D model, name, power, health, movement speed, combat type (which determines their spell combinations), faction, and respawn time.

NPC spawners can be directly placed in the game world or constructed using worker or other buildings. In the latter case, the spawner inherits the building's faction and sets the unit parameters accordingly. This approach eliminates the need to create dozens of individual units with different combinations of these parameters.

4.8 Items and Equipment

Items can have three different forms: being instanced in the game world, being stored in an player or building inventory or being equipped by a player (if the item is an equipment). The item system includes two main categories: resources and equipment.

Resources

Resources form the backbone of the game's economy. The inventory system supports stacking of identical items, with different stack limits for various resource types. Resources are used for constructing buildings and training units. The system includes shared storage within settlements. Material items are stackable with different maximum stack size for each item type.

Equipment

Equipment can upgrade hero's statistics like health and power. Some equipment is visible on the player's 3D model, with each piece having a distinct attached model. Each equipment piece unlocks one or two abilities assigned to specific keybind slots based on the equipment class. For example, the primary weapon provides the player with a unique normal attack (such as a fireball or melee strike) and a special attack with a longer cooldown. The offhand item grants a defensive or utility ability, such as a short-duration shield or a dash. Hats and armor also unlock an additional two abilities each.

Item Instances

Items appear in the game world as individual instances, each representing a specific type and quantity. The visual scale of an instance reflects its stack size.

Most item instances have physical properties. As explained in Section 4.2, items in motion can deal damage based on their mass, scale, and velocity, making them part of the combat system.

Items can be dropped through the inventory menu, on the death of an entity, or when destroying objects such as trees, buildings, or resource deposits. This system supports a more interactive game world, where environmental elements can be used strategically in combat. For example, cutting down a tree spawns wood items that can be collected or used as improvised projectiles.

Item instances persist in the game world until they are either picked up by a player or automatically despawned after a certain amount of time, to maintain performance.

Acquiring Items

Items can be acquired through gathering, defeating monsters, or other players. Players can gather resources by attacking trees or resource nodes, which take damage like other entities. When destroyed, they spawn items, such as wood from a fallen tree. Equipment is dropped by defeating NPC units, such as monsters, or bosses. Upon death, all items are dropped (except for equipment) on the ground and can be looted. Workers gather various resources such as wood, stone, and food, which are stored in dedicated storage building.

4.9 The Game World

Each map covers an area of approximately one square kilometer and contains destructible environmental elements such as trees and stone deposits. Wildlife, bandits, and bosses are scattered throughout the world. Maps can include one to three player-controlled settlements, along with predefined AI-controlled settlements. Each settlement can be loaded from a user-generated save file.

Computer Factions

Each map is populated by computer factions, including wildlife, bandits and leaders. A boss is a special NPC that utilizes a combination of different spells and have improved statistics like health and power. Computer factions may control pre-established settlements that do not expand throughout the game.

User Generated NPC Castles

Players can build and save their own settlement (including AI guards and workers), which can be loaded as computer faction castle. This adds user generated content to the game and improves replay-ability by allowing players to play against creations of other players even when playing offline.

4.10 Win Condition

At the start of each match, players are assigned to teams, with each team representing a faction. The core objective for each team is to protect its king while attempting to eliminate the kings of other teams. A match is won by defeating all opposing kings or by being the last remaining faction.

4.11 Game Chat

Players can communicate with each other using the in-game chat. The most recent message from a player is also displayed as a floating label above their character.

Commands and Cheats

The in-game chat also supports admin commands and cheats. For example, settlements can be saved to or loaded from a file using `/save filename` and `/load filename`. Additional commands allow to set player nicknames, grant items, and other functions.

Chapter 5

Market Research

Indie games account for approximately 40 to 50 percent of new releases on platforms such as Steam. According to 2024 data from SteamDB [10], over 70,000 indie titles have been released in the past decade. The number of annual releases continues to rise, increasing competition.

A 2024 analysis of successful indie games, defined as those with over 1,000 Steam reviews [11], highlights popular genres such as horror, simulation, roguelikes, survivor-likes, and open-world survival games. Roguelikes, particularly the survivor-like subgenre, have gained significant traction. Simple but appealing formats, including simulation games and vertical platformers, also perform well. Multiplayer features, especially cooperative modes, often contribute to success.

This project combines base-building, strategy, and action mechanics. While these genres are popular, they are less common in indie development due to the complexity involved in making them. Despite the crowded market, this project explores a niche that mixes real-time action with strategic base management in a multiplayer PvP and PvE setting. The design takes inspiration from both classic games and recent trends. It is driven by personal creative goals rather than a focus on commercial success.

5.1 Main Inspirations

This project is inspired by various games and incorporates features from multiple genres, including action games, role-playing games (RPGs), real-time strategy games, survival games, city-building games, exploration games, and multiplayer player-versus-player (PvP) games. The main inspirations are:

Stronghold Crusader

The main inspirations for this project is Stronghold Crusader, a real-time strategy game developed by Firefly Studios. Several features from the game influenced this project, including the grid-based building system with tileable walls. Another key element is the automatic worker and economy system, where placing a building spawns a worker who follows a predefined behavior loop. The use of faction colors and the scale of the game map also draw from Stronghold Crusader, along with its approach to economy and construction mechanics.

Arx Fatalis and Dark Messiah of Might and Magic

Arx Fatalis, developed by Arkane Studios and released in 2002, is a first-person action role-playing game. One of its most innovative features is its magic system, which requires players to draw rune symbols on the screen using the mouse in order to cast spells. Spells are constructed by combining runes that represent elements, actions, and effects. These runes must be discovered throughout the game. This system emphasizes both player skill and immersion, making spellcasting feel more meaningful than simply pressing a button.

Dark Messiah of Might and Magic, also developed by Arkane Studios, is a first-person action role-playing game known for its physics-based combat system. Players can use the environment to their advantage, such as kicking enemies into hazards or using objects as weapons. The game blends melee combat with environmental interaction, creating dynamic and tactical encounters.

5.2 Other Similar Games and Inspirations

- **Warcraft 3, Age of Mythology**: Real-time strategy games that combine hero units with large-scale armies. They feature base building, resource management, and tech progression, with an emphasis on management of combat and economy.
- **Magicka**: Magicka also has similar spell casting system of combining elements and spell types (shapes) to create different spells. The spell casting in this project is somewhere between Magicka and Arx Fatalis.
- **Dota 2**: The main inspiration from this game is its combat system, especially the Invoker hero, who uses a unique spell-casting system where three basic abilities are combined to create ten different spells, similar to Magicka.
- **Hinterlands, Terraria**: Games centered around a player-controlled character with mechanics focused on exploration, crafting, and building. Progression is driven by resource collection, equipment upgrades, and unlocking new areas or abilities. They also feature base building with NPCs that sell or produce items.
- **Rust**: An open-world multiplayer survival game where players gather resources, craft items, and build bases in a persistent player versus player environment. The gameplay emphasizes territorial control, raiding, and risk-reward survival mechanics.
- **Battlerite, V Rising**: Action-oriented PvP games with real-time, skill-based combat. They emphasize dodging, timing, and ability combos in isometric perspective. V Rising adds persistent progression and base-building elements within a multiplayer world.
- **Kenshi**: A sandbox exploration game focused on character progression and base building, including managing squads that follow work cycles similar to this project. Kenshi was one of the original inspirations for creating a multiplayer game in this style. However, the design of this project shifted away from exploration due to engine and multiplayer limitations as well as time constraints, though it may return in the future.

Chapter 6

Implementation

This chapter explains the implementation of the game systems described in Chapter 4, and references concepts, nodes, and features introduced in Chapter 3. The game is developed using the Godot Engine. Icons and other 2D artwork are created with Krita, while 3D models are made using Blender. Additional assets and development tools are listed in the appendix of this thesis.

6.1 Multiplayer Architecture

The game uses a client-server architecture based on Godot's `ENet` UDP networking. Both the server and clients run from the same executable, with roles determined at startup via command-line arguments such as `-server` and `-ip`.

When a client connects, the server handles their presence using player IDs (peer IDs) and spawns a corresponding player scene. Each player scene is named after its unique peer ID, which simplifies tracking and authority checks.

The server is responsible for authoritative gameplay logic, while clients handle replicated state and visuals. Game logic is split between server and client branches using `multiplayer.is_server`.

Player Authority and Control

Each player scene checks whether it is locally controlled using a local authority ID. This guarantees that only the right peer can control the input, camera, and interactions for their character and avoids unauthorized control of other player's entities. This setup is based on a video tutorial [6].

State Synchronization and RPCs

Game state is synchronized through a combination of:

- **MultiplayerSpawner**: Spawns networked scenes (e.g., players, NPCs, projectiles) and replicates them across clients with correct ownership, eliminating the need for manual instantiation.
- **MultiplayerSynchronizer**: Synchronizes continuous states such as health, position, and animation across the network.

- **RPC and RPC_ID**: Used for discrete actions like spell casting, item usage, and event triggering.

6.2 Game World and Entities

The runtime world is composed of the following types of entities:

- Player and non-player characters
- Environmental resources such as trees and rocks
- Structures including towers, gates, and buildings
- Projectiles and spells
- Buffs, debuffs, and items

All entities exist on both server and client, with server handling logic and clients handling input and displaying the replicated state and visuals. For example, status effects are applied through server logic but shown on clients using particles, meshes, or animations.

Entities are organized into groups such as „`players`", „`resources`", or „`buildings`", and are identified at runtime using their `name`. Since direct node references cannot be passed across the network, remote systems locate and interact with entities dynamically through a shared utility function that retrieves nodes by their group and name.

6.3 Entity Structure: Players, NPCs, and Buildings

Different types of entities, such as players, NPCs, and buildings, share common properties like health, buffs, debuffs (e.g., being stunned, frozen or on fire), inventory and a 3D model with animations. Because of this, these entities are similarly structured from modular components including shared classes like `Condition` and `Inventory`. These base classes are then extended for specific entities, for example the `PlayerInventory` class, which adds functionality for equipment attributes and interaction with nearby items. Below in Table 6.1 is a simplified version of the entity scene tree.

Node	Description
`PhysicsBody`	Root node of the entity
`CollisionShape3D`	Defines the collision bounds
`MultiplayerSynchronizer`	Synchronizes node variables over the network
`Condition`	Manages health, buffs, debuffs, and death
`MagicSystem`	Handles abilities and spellcasting
`Inventory`	Manages items and item interactions
`UI`	Displays health bar, cast bar, and target indicators
`3D Model`	Mesh with animations and skeleton
`Other nodes or systems`	Scene-specific features

Table 6.1: Simplified Entity Scene Tree used for players, NPCs, buildings, and other entities

6.4 Camera

The Godot camera system uses the `Camera3D` node to render the 3D scene from a specific point of view. A `perspective` projection with a narrow field of view (35–40°) creates an isometric effect.
The player camera setup uses the following node hierarchy:

- **Node3D**: A pivot point used to move the camera position.
  - **SpringArm3D**: A RayCast3D node that detects collisions with other objects and adjusts the position of its children to either the collision point or its own endpoint.
    - **Camera3D**: The actual camera node that displays what is visible from its position.

The Figure 6.1 shows the visual representation of this setup in space.

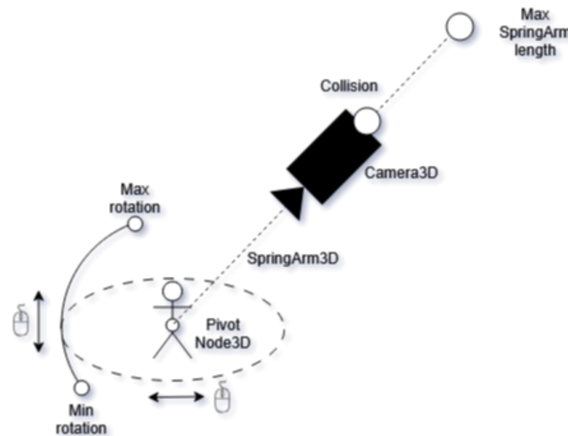


Figure 6.1: Third-person camera setup using a `SpringArm3D` and `Camera3D`. The spring arm automatically shortens to avoid clipping through obstacles. The pivot allows rotation.

Camera Positioning

The player's global position is handled on server and replicated to clients. To avoid jitter from network delays, the client interpolates camera movement using `lerp` on `Vector3`, smoothing motion toward the replicated position. Higher interpolation speeds reduce jitter but may cause minor input lag.

Camera Rotation and Zoom

Mouse movement is handled in `_input(event)` via `InputEventMouseMotion`. The relative vector is scaled by sensitivity and applied to the camera's rotation. Vertical rotation is clamped to prevent flipping. Mouse scroll adjusts the `SpringArm3D` length, clamped within a zoom range.

Cursor Projection in 3D

To determine the cursor's position in the 3D world, a ray is cast from the camera through the cursor's screen coordinates. The ray intersects with 3D bodies on specific collision layers, filtering out irrelevant objects. The point where the ray collides with an object is then used as the cursor's position in world space. The Figure 6.2 illustrates how the mouse cursor is projected into 3D space using raycasting. The resulted position from this function is used as client input by most of the game's systems for actions like getting targeting spells or placing buildings.

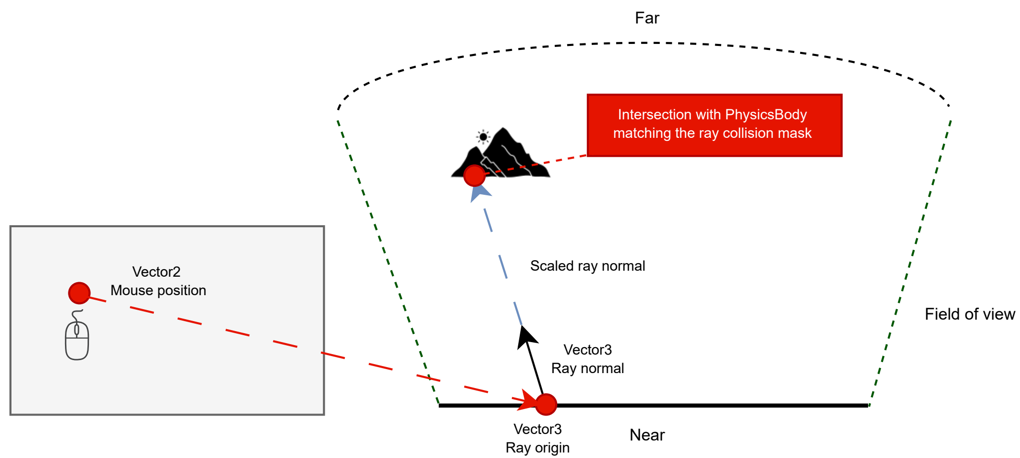


Figure 6.2: Raycasting from the camera through the cursor to determine the 3D position.

6.5 Character Controls

The `CharacterBody3D` node is commonly used for player controllers, but its interactions with `RigidBody3D` are undefined and can lead to unpredictable behavior. Each interaction feature, such as jumping on or pushing objects, would require custom implementation.

To enable consistent physics-based interaction and simplify spell logic, all characters use `RigidBody3D`. This approach supports natural responses to forces from spells like gravity or wind (e.g., via `apply_force`) and guarantees uniform interactions with other physical objects, including players, items, and environmental props like boxes or rubble.

Movement

The player scene includes several nodes essential for movement:

- `RigidBody3D`: Handles physics-based movement and collisions.
- `CollisionShape3D`: Defines the collision boundaries for the rigid body.
- `Area3D`: One detects when the player is on the ground, the second detects walls or objects that allow climbing.
- `RayCast3D`: Detects surface normals for aligning movement to slopes or uneven terrain.

These nodes are organized as shown in Figure 6.3.

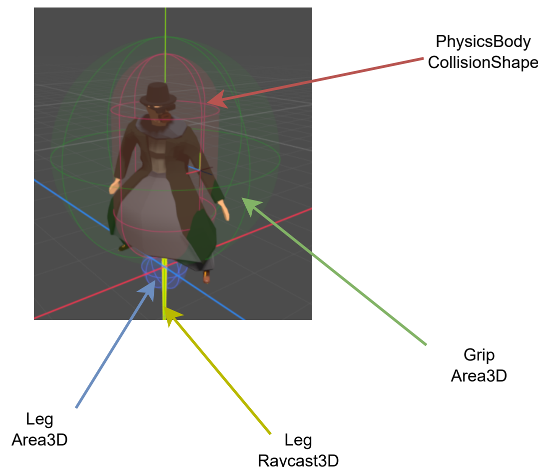


Figure 6.3: Annotated player node structure for movement and environment detection.

Player movement is controlled using the WASD keys, which generate a direction vector relative to input. This vector is normalized and rotated around the vertical axis based on the camera's current orientation to align with the player's view direction. The vertical axis value of the direction vector is set to 0 and then the vector is normalized. The down pointing `Raycast3D` node is used to detect the collision normal value of the terrain below which is then used in the `slide` function as a parameter to enable walking on slopes.

Ground Detection and Movement States

The character uses two `Area3D` nodes and a downward-facing `RayCast3D` to determine contact with the ground. The leg `Area3D` detects whether the character is on the floor and provides broader control for jumping and terrain interaction. When the leg area is not colliding or when the Y component of the terrain normal is below a defined slope threshold, the character is considered airborne. In this state, the physics engine handles falling behavior based on gravity or external forces. Friction and damping features of the Jolt physics engine regulate deceleration.

Jumping and Climbing

Jumping is possible when the leg `Area3D` is colliding. Holding the jump input applies an upward force to the body, allowing partial or full jumps based on input duration. When moving, a horizontal force is applied in the movement direction, but it is reduced while airborne. Jumping is blocked on steep slopes where the terrain normal is too low on the Y axis.

Climbing happens when the grip `Area3D` is colliding while the legs are not. An upward force is applied, weaker than during a jump. A stamina system limits climbing duration. Stamina decreases while holding the jump input and recovers when on ground.

Rotating the Character

The character's rotation is purely visual so only the 3D mesh is rotated to face a direction. To determine the facing direction, the angle between the character and a target point is calculated using Godot's `atan2` function. This angle is then gradually interpolated over time, resulting in smooth and natural-looking rotation.

6.6 Character Skeleton Animations

Humanoid animations for player and AI characters are created using Mixamo animations, along with the Godot Game Tools addon for Blender. The animations were downloaded from Mixamo.

In Blender, the custom mesh was created and rigged using the rigging feature from the Godot Game Tools add-on. The downloaded animations were then also retargeted using this add-on. Afterwards, the animated model is exported as `.gltf` file. The combined animations are converted into an `AnimationPlayer` node and match the animations displayed in Blender.

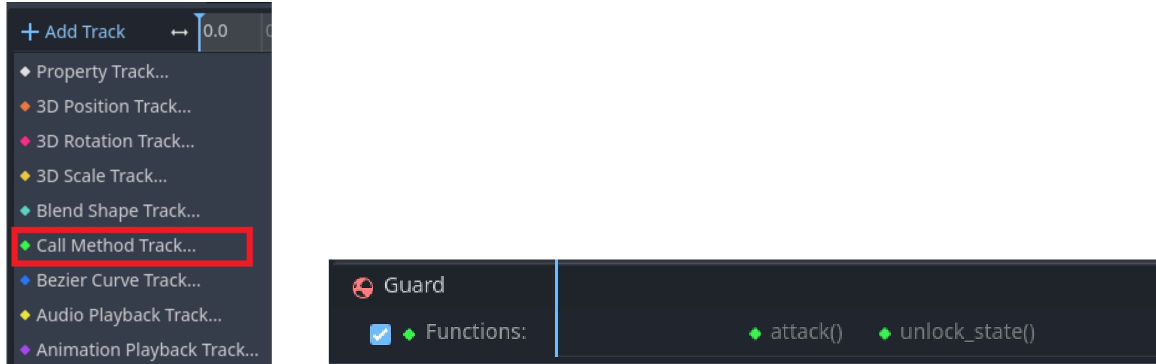
Animation States

A simple state machine is used to control a character's behavior and animation. The character can be in various states such as `IDLE`, `RUN`, `JUMP`, `ATTACK` or `STUNNED`. These states change based on the character's movement and other conditions, such as whether the character is on the ground or in the air.

To maintain consistency in multiplayer settings, only the server is allowed to change the state. A lock mechanism prevents the state from changing during critical actions, such as attacking. Animation is determined by the current state, with checks to avoid

repeating animations unnecessarily. For example, not restarting the jump animation while the character is falling.

To synchronize certain actions with animations, a Call Method Track can be used to invoke functions on any node within the scene. This is useful for features such as the state locking mechanism described above. Figure 6.4 shows an example of this. When a character is frozen, which is set within its `Condition` class, the animation is frozen by setting the speed scale of the `AnimationPlayer` to 0.



(a) Adding call method track.

(b) Using a call method track to trigger an attack and unlock the animation state.

Figure 6.4: Using the `AnimationPlayer` node to call functions through a call method track.

Synchronizing Skeleton Animations with Cast Time

To align visual feedback with gameplay timing, ability animations are synchronized using the animation player. For casted abilities, the animation speed is dynamically adjusted to match the ability's cast duration. This is achieved by setting the animation player's speed and using its built-in callback tracks to trigger logic at specific points in the animation. Instant abilities are locked to ensure the full animation plays before allowing state changes, unless interrupted by higher-priority events like stuns. Channeled abilities combine both approaches: the animation speed is scaled during the cast and slows down near the end, staying that way until the channel finishes or is interrupted.

6.7 Combat System

The combat system is built upon multiple modular subsystems that interact with each other. The primary goal was to reuse the implemented abilities for both player and non-player characters. This system also supports ability spawning mechanism where abilities can be spawned by not only characters but also by other game systems. This allows for more complex interactions, such as environmentally triggered abilities or scripted events.

Items and other physical objects can be generated similarly to abilities, with initial attributes like velocity. When these objects collide with entities such as players, buildings or trees, they inflict damage based on their current speed, mass and collision damage variance. Non-static entities like players and NPCs possess physical attributes just like the items and objects, although with varying values. These physical properties are derived from the Jolt Physics engine.

The next diagram in Figure 6.5 provides an overview of the system's implementation and interactions between its subsystems:

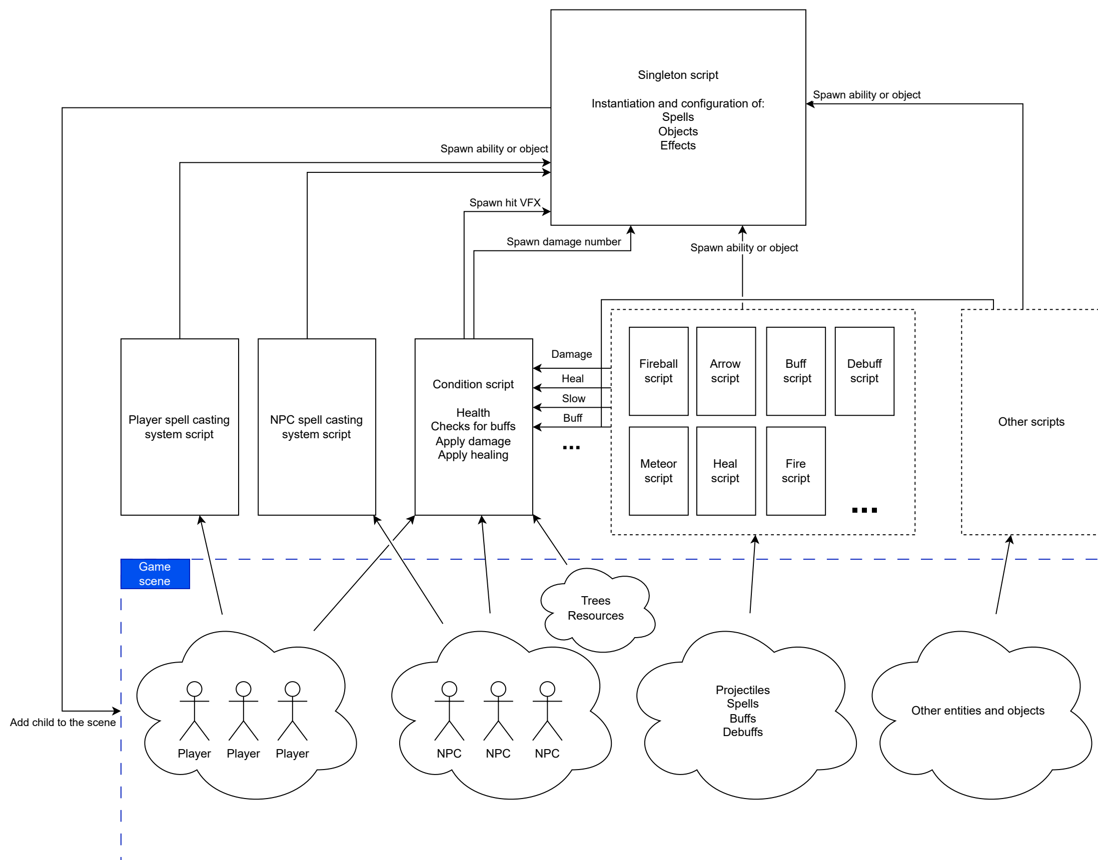


Figure 6.5: Overview of the modular system implementation, showing interactions between subsystems. The lower section represents the game scene, while the upper section depicts abstract systems (scripts) attached to nodes in the scene.

Condition

The `Condition` component is a custom class responsible for managing a character's health, applying healing and damage, and handling status effects and respawning. It evaluates incoming effects based on both the caster's and target's attributes, such as power and active status effects.

Damage and healing are processed by functions in the target's `Condition` node. These functions calculate final values using the caster's power and base ability stats, adjusted for variability and the target's defenses. If health reaches zero, the component handles death and triggers item drops, respawn timers, and visual feedback.

Buff and debuff states are periodically checked and updated (see Section 6.7). The `Condition` class ensures these states influence damage and healing appropriately.

Spells and Abilities

Each ability is defined as a separate scene with a script that controls its behavior such as movement, effects, damage, healing, and duration. Currently, approximately 25 functional abilities are implemented. Their names and descriptions are provided in Section 4.2 of the design chapter.

This subsection discusses examples of abilities and their implementation. Visual effects are detailed in Section 6.8. Most abilities follow a consistent structure, as shown in Table 6.2.

Node	**Purpose**
`KinematicBody3D`	Root node for movement and collision
`CollisionShape3D`	Defines physical collision shape
`Area3D`	Detects overlapping entities
`Timer`	Manages timed events (e.g., despawn)
`MeshInstance3D`	Visual model of the ability
`GPUParticles3D`	Visual effects like particles
`AnimationPlayer`	Plays animations and triggers scripted calls

Table 6.2: Typical node structure used in ability scenes.

Detection is performed using an `Area3D` node, either through signals or querying overlapping bodies. Detected bodies are processed by calling functions on their `Condition` component, which handles damage, healing, or debuffs. `Timer` nodes manage delays, durations, or periodic behavior such as repeated damage or despawn timing. The root node is generally a `CharacterBody3D` or `StaticBody3D`, depending on whether the ability moves or remains stationary. Some abilities, like Throw and Black Hole, interact directly with existing physics objects such as items or player bodies by applying forces.

Examples of Ability Implementations

The following list outlines various abilities and how their behavior is implemented using different node types and logic structures. Other abilities follow similar implementation:

- **Tornado** uses a moving `CharacterBody3D` and applies lift, stun, and damage effects to enemies detected by its `Area3D`.
- **Black Hole** is a stationary ability that uses a timer to periodically pull in physical objects by applying directional force and deal damage to entities within its area.
- **Slash** applies close-range damage to nearby enemies shortly after activation, using a timer to simulate attack timing before checking for overlapping bodies.
- **Heal Nova** gradually expands its collision shape and heals allies while damaging enemies inside its area.
- **Flame Strike** plays an animation before dealing damage and spawning a secondary `Fire AoE` ability at the target location.
- **Fire AoE** uses a timer to apply damage over time to enemies remaining inside its area.
- **Leap Ground Slam** This ability first lifts the caster into the air and starts a timer. When the timer times out, the caster is launched toward the selected position. Upon collision with the environment, a `Ground Slam` ability is triggered at the caster's position. It stuns and damages nearby enemies using `Area3D` node.
- **Throw** This ability has two stages. The first input selects a `RigidBody3D`, such as an item, player or NPC body. Its position is then smoothly interpolated every physics frame to a fixed point above the caster's shoulder. The second input launches the selected body toward a target position.
  The resulting behavior depends on the object type, such as dealing damage or breaking apart on impact. For example, if it is a stone item, it damages other bodies on collision, reduces its own scale, and decreases its item count.

Status Effects

Status effects or buffs and debuffs are a subset of spells that follow specific targeted entities and apply periodic effects. Status effects are instanced like any other ability and take the references to caster and target bodies as parameters.
Status effect scenes typically contain the following nodes (see Table 6.3):

Node	Purpose
`Timer` nodes	Used for managing duration and period of the status effect.
`Particle3D`	Provides particle-based visual effects.
`MeshInstance3D`	Used to display animated 3D models as visual effects.
`MPSynchronizer`	Synchronizes variables across clients.

Table 6.3: The nodes that make up a status effect scene and their purposes.

Some abilities do not apply any periodic effects but are still treated as buffs or debuffs in the code, as they take advantage of the buff system's ability to follow a target body. These abilities are not shown in the debuff list above the player or target health bar.

For example, the `Lava Pillar` and `Stab` abilities are purely single-target and use this system to stay attached to their targets without being visible in the UI.

Checking Status Effects

Buffs and debuffs are checked periodically every few frames by the `Condition` class of each entity. To check a player's buffs, the `get_children` function is called on a parent `Buffs` node within the scene. The system then iterates through the buffs where the `caster` variable is set to the same entity that the `Condition` is a child of.

To determine the type of a status effect, each effect contains boolean parameters such as `is_stun`, `is_slow`, and so on. A single status effect ability can have multiple types simultaneously.

Some status effects, such as stun, are represented as simple `bool` values. Others, like speed buffs, are stackable and are stored as integer counts.

Examples of Status Effects:

Speed Buff

- Increases movement speed and stacks with multiple applications
- Applied through the `Condition` script and handled in the movement logic of each entity
- Visualized with particle effects

Freeze

- Disables input, movement, and animation
- Triggered by the **Ice Nova** spell when hitting enemy entities
- Visualized by an ice block surrounding the target

Fire Curse

- Deals periodic damage over time
- On the second timer tick, checks nearby enemies using an `Area3D` node and spreads by spawning a duplicate with reduced stack count
- Visualized by fire particles and a small explosion when spreading

Instancing Abilities

A globally accessible manager script handles the instancing of abilities, objects, and visual elements such as damage indicators. This manager is responsible for creating these elements, setting their parameters, and placing them correctly in the scene tree. Since it is globally accessible, players, NPCs, and even other abilities can instantiate new abilities through this manager.

All instancing functions include a reference to the caster, which provides access to relevant attributes like faction, power, and condition. These are essential for interactions between the ability and other entities.

When targeting is involved, a reference to the target is also included. This allows abilities to perform specific actions such as tracking, applying effects, or interacting with the target based on custom logic.

Ability Parameters

Each unique ability has a predefined set of parameters including range, cooldown, line of sight requirements, targeting type, and a graph used for casting special abilities. These parameters are described in Section 4.2. They are stored in a `Dictionary` keyed by the spell enum value. Other parameters, such as speed or damage, are defined by the specific behavior of each ability.

Cooldown

Each ability has a predefined cooldown, stored in an array indexed by the ability's ID. Values represent remaining time in seconds. A timer updates every 0.1 seconds, reducing non-zero values. When an ability is used, its cooldown is reset to the base value.

Ability Range

Ability range is determined by checking whether the distance between the start and target positions is less than the ability's maximum range. Godot provides built-in methods to calculate the Euclidean distance between two positions.

Special Ability Detection and Activation

Powerful abilities are created by connecting points with the mouse cursor, as described in the *Design* section. This uses `Area2D` nodes to detect mouse input on each point. Each connection is stored as `Vector2(pointA, pointB)` in an array, with points defined by an `enum` to form a graph. After a second key input, the array is compared to a constant `Dictionary` of spell graphs. A match occurs if all connections are present, regardless of direction (`A to B` is the same as `B to A`) and regardless of the order of connections in the array. Once matched, the spell becomes ready to cast using a separate key. Visual lines are drawn using `Particles2D` nodes, rotated and scaled based on the distance between points. The current visuals are placeholders, as this is a newly reworked feature following the movement and spell casting overhaul. They are shown in Figure 6.6.

Figure 6.6: User interface visuals with an example input for casting special abilities, using `Particles2D`, `Label`, and `Line2D` nodes.

Targeting

Abilities can target either a ground location or an entity (e.g., player or NPC). Ground targeting uses the projected mouse position in world space to select a point on the terrain. Entity targeting relies on a shape query around the cursor to detect nearby valid targets. The closest detected body is selected (see Figure 6.7), and selecting it again will deselect it.

The selected target is replicated to the server via a remote procedure call that passes the target's name and group, as instance references cannot be directly synchronized. Deselecting the target follows the same process by removing the reference on the server. Once selected, the target is visually marked and this is reflected in the UI (see Figure 6.8).

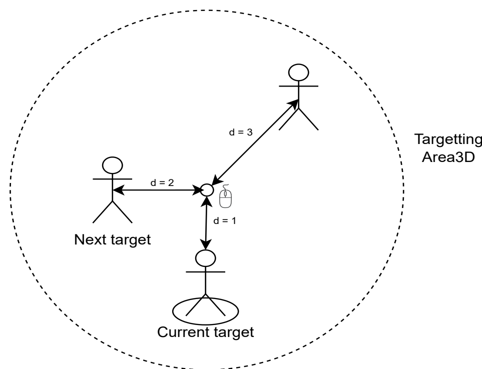


Figure 6.7: Selection of the target is based on the closest distance to the mouse cursor.

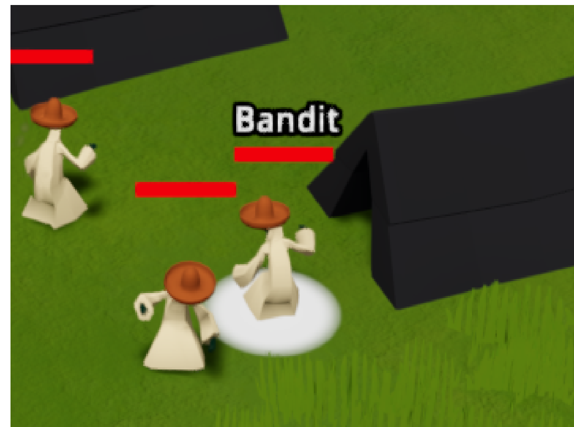


Figure 6.8: Target representation in game using name label and quad mesh.

Line of Sight

Line of sight determines whether a target is visible along an unobstructed path. It is used for spell casting, AI behavior, and damage filtering. Line of sight is implemented using raycasting with a `RayCast3D` node.

- **Targeted Abilities:** Some spells are applied directly to a location or target rather than traveling there. These require a clear line of sight to prevent unfair use through walls or obstacles.
- **AI Behavior:** NPCs can only detect and attack targets within visible range. This prevents unrealistic behavior like attacking through walls or locking onto unseen enemies.
- **Damage Culling:** Certain area effects, such as explosions, only apply damage to entities in line of sight. (Not yet implemented.)

Targeted Abilities

Ground targeted abilities cast a ray from the player's head to the projected mouse position on the terrain while excluding player and NPC layers from the collision mask. Entity targeted abilities cast a ray from the player's head to the target's torso, including player and NPC layers but excluding all characters except the target (see Figure 6.10).

The line of sight is successful if, for entity-targeted abilities, the raycast detects the target object. For ground-targeted abilities, the raycast collision position must closely match the projected mouse position (see Figure 6.9). On success, the target is synchronized with the server, verifying line of sight on both client and server.

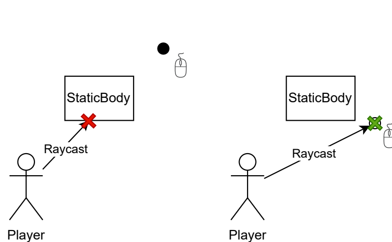


Figure 6.9: Overview of line of sight calculation for targeting.

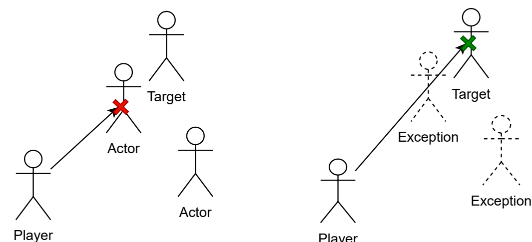


Figure 6.10: Target line of sight excluding other entities with the same collision layer.

Combat System User Interface

The combat UI is created using `ProgressBar` nodes for health and cast bars, `TextureRect` for spell icons, and an `ItemList` node for status effects, along with `Label` nodes for names and other text. Spell activation via key press is animated using an `AnimationPlayer`. The combat user interface is shown in Figure 6.11.

Figure 6.11: Combat UI showing player and target information, a cast bar, and ability buttons with keybinds and cooldowns.

The combat system displays error messages using an `AnimationPlayer` and `Label` nodes, fading them in and out by adjusting their transparency. Examples of such messages include: “Target out of range” or “Spell on cooldown”.

Currently, errors are handled individually on the client side only. A unified system is planned to manage all messages within a shared container accessible by both the client and server.

3D Health Bar

Entities that use the `Condition` component show a health bar above its model. This bar is a `Sprite3D` or `MeshInstance3D` with billboard mode enabled. A `GradientTexture1D` shows the health using two colors: red for health and gray for missing health. The gradient uses constant interpolation. Each instance uses a unique texture and mesh set, and calls `set_local_to_scene` to create a deep copy, ensuring that multiple players do not affect each other’s health values visually.

The current health is divided by the maximum health to get a normalized value between 0 and 1. This value updates the texture offset. A signal from the `Condition` component triggers the update and synchronizes it over the network using RPC.

2D Health and Cast Bar

2D bars use the same logic as the 3D health bar. Normalized value is calculated by dividing the current value (such as health or cast time) by the maximum. Implementation is simpler, since only one instance is needed, and Godot provides built-in nodes like `ProgressBar` and `TextureProgressBar` for this purpose.

The UI references the player with local authority and the target if any. Both have a `Condition` child node that tracks health. The UI updates both using the same logic described in the 3D health bar section.

6.8 Visual Effects

The visual effects of the abilities are created by combining instances of `MeshInstance3D` and `GPUParticles3D`, with an `AnimationPlayer` animating the properties of these nodes.

Particle Effects

The `GPUParticles3D` node has two main components:

- **Draw Pass**: Defines the mesh used for rendering each particle. Typically, this is a simple shape like a quad or a custom mesh. The mesh uses either a `SpatialMaterial` or a `ShaderMaterial`, which controls visual properties such as color, transparency, and animation effects like fading.
- **Process Material**: Usually a `ParticleProcessMaterial`, which controls particle behavior such as lifetime, speed, direction, gravity, and randomness. These can be animated using curves or textures to animate over the particle lifespan

Billboards

Billboards are a technique where flat meshes (typically quads) always face the camera. Billboards are particles that always face the camera. In Godot, this is enabled by setting the `Billboard mode` in material settings of the particle draw pass to `Particle Billboard`. This is useful for effects like fire, magic glows, or explosions that do not require 3D orientation.

Particle Trails

Particle trails are visual extensions that follow the movement of particles over time. These can be enabled using the `trail_enabled` property in the particle system, and are useful for simulating effects like traces behind fast-moving objects, magic projectiles, or motion blur.

Mesh Effects

The properties of a `MeshInstance3D` and its material can be animated using the `AnimationPlayer`. This includes transformations such as scale, rotation, and position, as well as material properties like color, transparency, emission, and UV offset or scale.

UVs and Scrolling Textures

Scrolling textures are a technique where a texture moves across a surface or screen, often used to simulate effects like flowing water, energy waves, or moving other effects with motion. This effect is achieved by adjusting the texture's UV coordinates over time, creating the illusion of motion. In Godot, this can be implemented in a custom shader by modifying the UVs using a time-based offset.

Example of VFX Implementation

All the abilities and their visual effects can be seen in the video in the attachments. Below is an example of the implementation of an effect which combines most of the techniques used for most abilities.

This visual effect consists of three primary components, as illustrated in the image below:

- **Trail particles**: Represent the fire cloud at the top of the effect.
- **Scrolling textures**: Depict the fire descending and making contact with the ground.
- **Billboard particles**: Simulate the explosion upon impact.

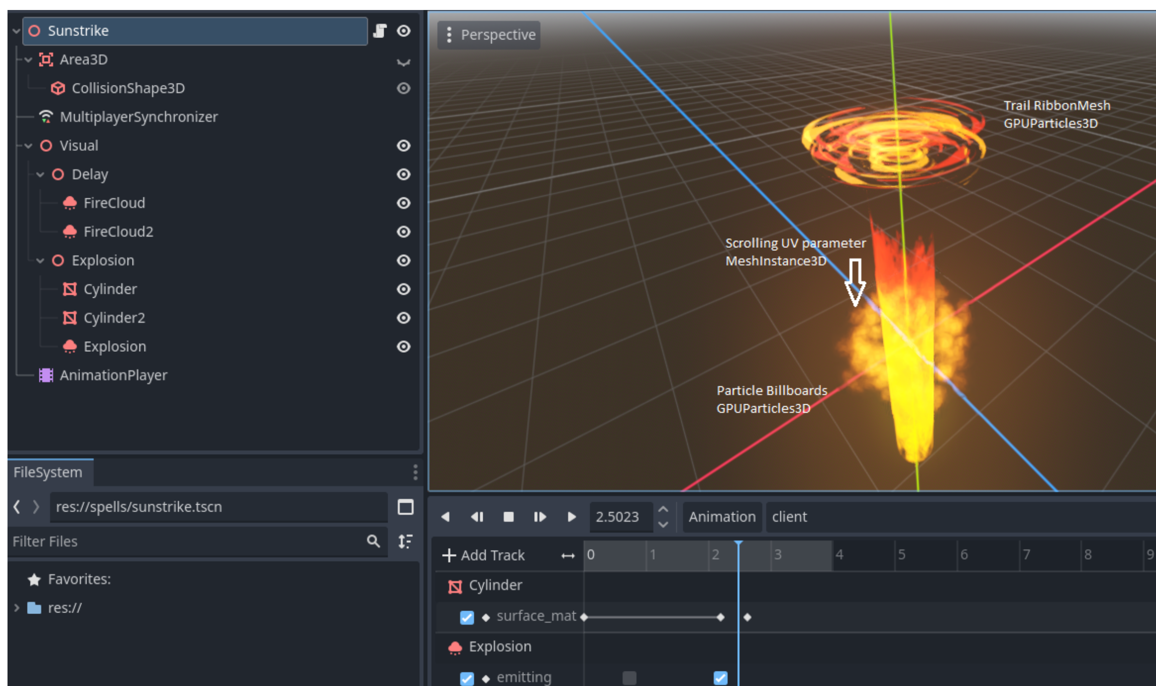


Figure 6.12: Example of a VFX scene inside the Godot editor, showcasing billboard and trail particles with scrolling textures animated using the AnimationPlayer.

Trail Particle Cloud

The trail particle cloud at the top of Figure 6.12 uses a `GPUParticles3D` node with a `RibbonTrailMesh` draw pass. Both `Trails` and `Use Particle Trails` must be enabled. The trail lifetime is set to 0.7 seconds, which is slightly shorter than the particle lifetime to avoid artifacts.

A `Curve` texture defines thickness along the trail. The material uses `Alpha` transparency, `Add` blend mode, and glow via the `Emission` parameter.

A `ParticleProcessMaterial` handles animation. Emission shape is a ring. Y-axis orbit velocity keeps motion in the horizontal plane. A second trail with a different emission color is layered for added depth.

Scrolling Textures

A cylindrical 3D mesh is modeled and UV-unwrapped in Blender (See Figure 6.13 below) to display a seamless fire texture created in Krita. The model is exported as a `.obj` file and imported into Godot.

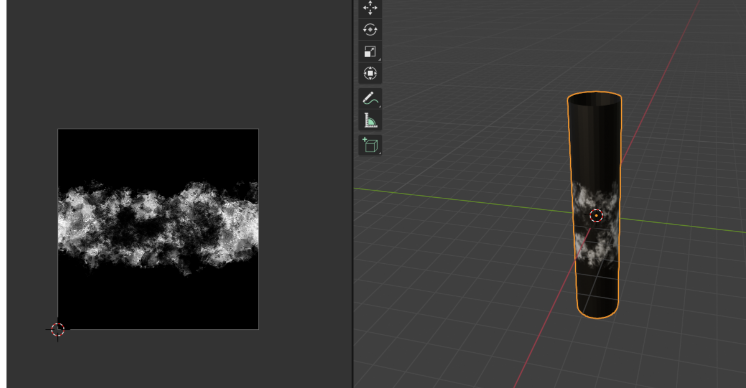


Figure 6.13: Seamless VFX texture made in Krita applied to cylinder mesh inside Blender.

In Godot, the texture is applied to the mesh using a `SpatialMaterial3D`. The material uses `Alpha` transparency and `Add` blend mode with orange `Emission` color with a high multiplier enhances brightness and glow.

Motion is simulated by scaling the `UV Y` parameter and animating the `UV Y Offset` using an `AnimationPlayer`, creating a continuous vertical motion after an initial delay.

A second mesh with the same setup but a different `Emission` color is layered on top. Its offset animation is slightly delayed to create a trailing effect.

Particle Explosion

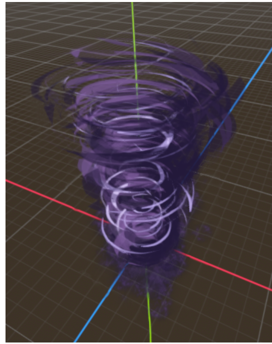
The final component is a particle explosion, implemented using a `GPUParticles3D` node with a `QuadMesh` draw pass. Transparency and emission settings match those used earlier. The material uses `Particle Billboard` mode with `Keep Scale` enabled to preserve size based on `ParticleProcessMaterial` parameters.

Explosive motion is driven by high explosiveness, a short 0.7 second lifetime, and a `Spread` of 180 degrees for full spherical emission. A `Flatness` value of 0.7 slightly flattens particles toward the horizontal plane. `Initial Velocity` is randomized between 4.0 and 6.0 units to add variation. The `Oneshot` option ensures the effect plays only once.

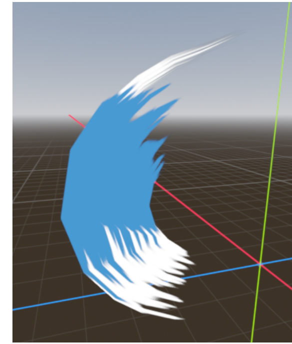
To fade particles smoothly, an `Alpha Curve` texture is applied in the process material, gradually reducing transparency over time.

Other Effects

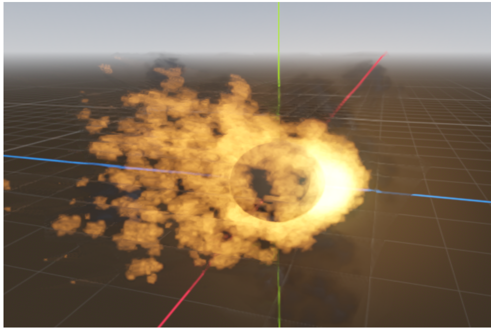
Other effects utilize similar concepts to those described in the previous example. The next Figure 6.14 shows multiple examples:



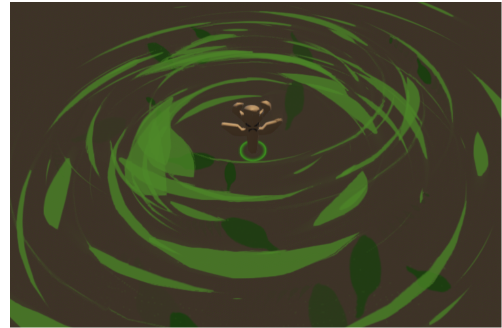
(a) Tornado VFX using trail particles.



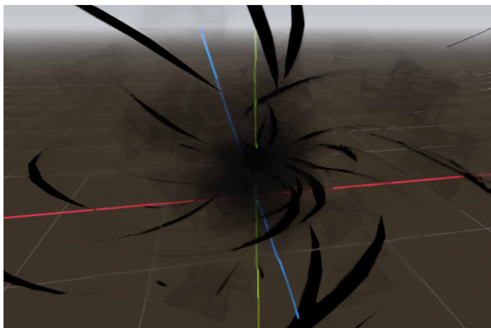
(b) Slash effect using scrolling UV textures.



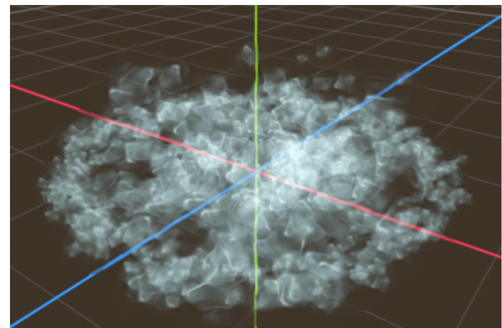
(c) Meteor using particle effects with animated emission for glow.



(d) Heal totem using trail and billboard particles along with orbital velocity.



(e) Blackhole swirling effect using negative radial velocity.



(f) Ice nova expanding particle effect using positive radial velocity and damping.

Figure 6.14: Example of VFX scenes inside the Godot editor using billboard and trail particles with scrolling textures animated by the AnimationPlayer.

6.9 Building System

The building system is divided into two parts:

- **Client side:** Handles input, placement preview, and basic collision and inventory checks to minimize server calls.
- **Server side:** Performs validation, processes building costs, instantiates objects, and updates navigation meshes.

Two main types of objects can be placed in a town are **structures** and **walls**. Structures include houses, towers, and gates, and can be rotated in 90° increments. Walls are auto-tiled and oriented automatically during placement. Each structure is also described in Section 4.4. The simplified diagram in Figure 6.15 shows the main subsystems of the building system. Each subsystem is described in more detail in this chapter.

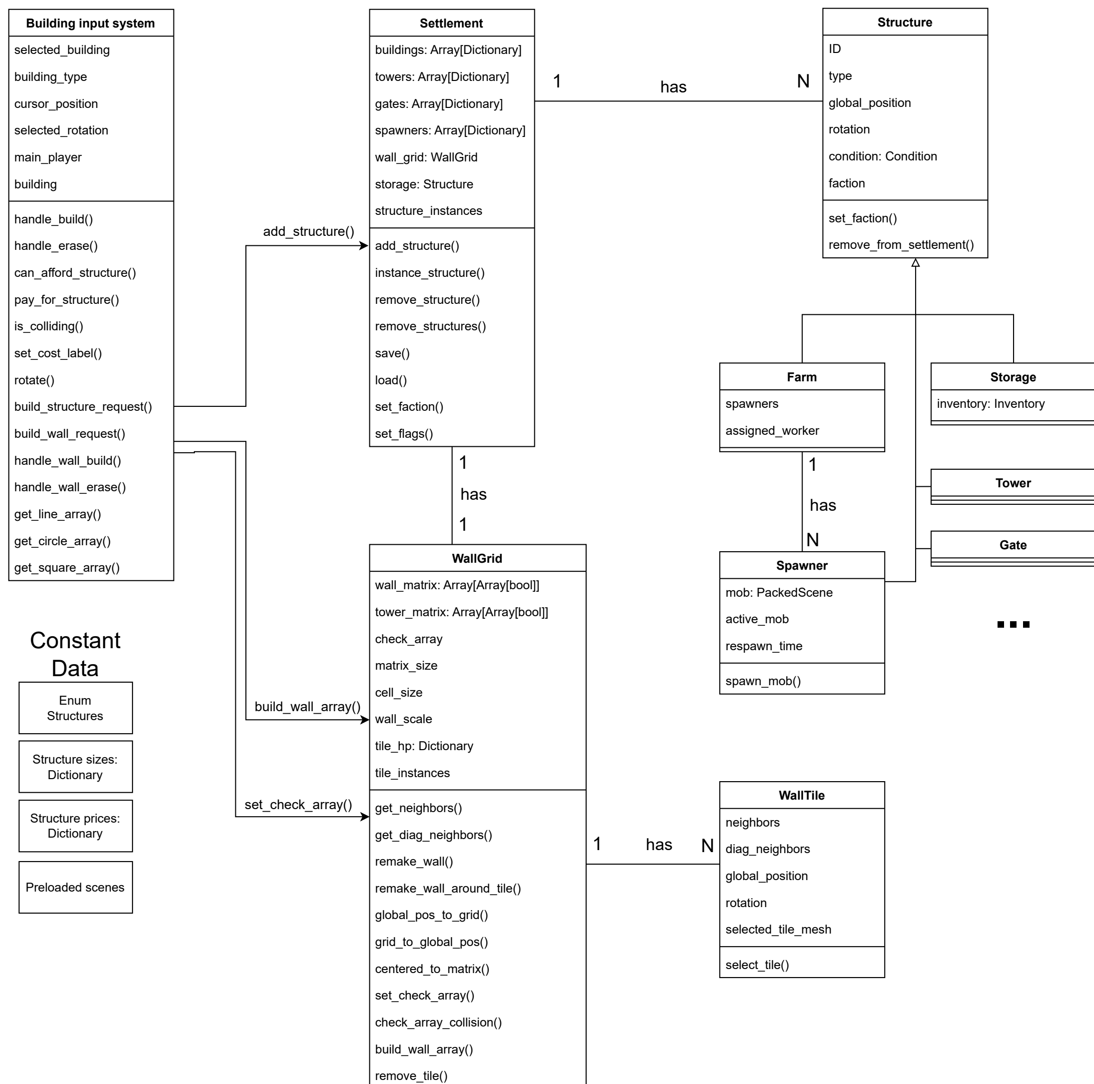


Figure 6.15: Simplified class diagram of the building system and its subsystems, including constant data.

Custom Wall Grid System

All structures are placed on a two-dimensional grid centered around the town center. Each grid cell corresponds to a single wall tile, with buildings sized in whole-cell units (e.g., 2x2, 3x5). Positions are snapped to the nearest grid cell for alignment.

Godot includes a built-in `GridMap` node for grid-based object placement. However, it was not well-suited for this project's requirements. It lacks support for multi-cell structures and cannot handle more advanced behavior such as per-object health, inventory, or custom scripted logic. To address these limitations, a custom grid system was implemented, providing full control over placement, collision checks, and behavior management for each individual structure.

The settlement node contains a child `WallGrid` node that manages a discrete grid system for structure and wall placement. This wall grid system is part of the building system and handles the placement, visualization, and interaction of wall structures within the game.

The backend grid tiling system is implemented using a two-dimensional boolean array, where each `True` value represents an occupied space. A separate `Dictionary` stores references to in-game instances based on their positions in the grid.

Grid Position Conversion

To place structures accurately, global positions (such as the cursor's projected position) are converted into corresponding grid matrix coordinates. This conversion accounts for the settlement's position and grid cell size to align objects precisely within the grid.

The grid is centered around the settlement's origin, ensuring consistent placement even if the grid size changes. This is done by offsetting the grid matrix by half of its width and height, which aligns with the horizontal plane (x and z axes) in Godot's 3D space, as shown in Figure 6.16.

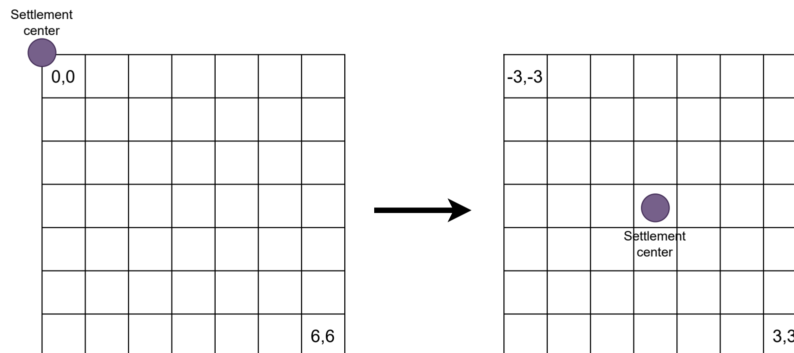


Figure 6.16: Comparison between a non-centered and a centered matrix with the center point indicated.

Since array indices start at 0, negative grid coordinates are shifted into the positive range before indexing. Inverse functions convert matrix or grid positions back to global coordinates.

Structure Management

Each Settlement tracks all buildings in a nested Dictionary. Each entry includes a structure's ID, type, position, and rotation. The ID is assigned from a counter and used to reference the instance in the scene. Preloaded PackedScene instances are mapped to each type in the id_to_scene Dictionary.

When the player initiates placement, the system checks for collision and validates that the required materials are present in the player's and storage inventories. Prices are defined in dictionaries indexed by the Structure enum. Once validated, the placement data (ID, position, rotation) is sent to the server using RPC.

Building Input System

This system handles the selection, placement, rotation, and finalization of buildings and walls. It checks for collisions and resource availability before initiating placement.

The placement cursor snaps to the grid and includes a visual indicator that changes color on collision, along with a label displaying the building cost.

Building Controls and Selection

Entering building mode changes the UI to show build options:

- **Bottom menu:** Rotate, place, and remove (keys Q, E, R, F).
- **Middle menu:** Building category selection (keys 1–6).
- **Top menu:** Shows current structure with left/right cycling (A, D).

Figure 6.17: Screenshot of the building menu user interface.

Buildable objects are categorized into three types and have different building input and management logic:

- **Walls:** Placed using drag-and-drop and auto-tiled.
- **Structures:** Built with a single click. These include towers, gates, and buildings. Buildings support free rotation, while other structures are limited to 90° increments.
- **Spawners:** Built the same way as structures but unique in that they can be placed on top of towers, gates, or walls.

3D Building Input Interface

The input system for both buildings and walls consists of 3D interface that follows the mouse cursor and snaps to the wall grid by rounding the floating-point position. This can be seen in Figure 6.24 below.

The interface includes the following components:

- **Placement Cursor:** Visualizes the position and size of the building. It adjusts based on grid alignment and rotation. Vertical lines face the camera using billboard materials and mark the structure's boundaries.
- **Price Label:** A `Label3D` displays the cost of the selected structure, including available resources. Each building has different price made of one or more resources defined in a `Dictionary` structure.
- **Placement Collider:** Consists of a `BoxMesh` inside a `MeshInstance`, paired with an `Area3D` node. The mesh changes albedo color depending on whether the area overlaps other bodies. This serves as a client-side check to prevent structures from overlapping existing objects or trapping players and NPCs. The collider and its `Area3D` shape scale dynamically based on the selected building's size, defined as a `Vector3i`. The X and Z components determine footprint, and Y determines visual height. Structure types are mapped to their size using a `Dictionary` of `Vector3` values.

(a) Empty collider allowing placement.

(b) Collider overlapping with player body.

Figure 6.18: Placement cursor with price label and collider. Blue collider indicates allowed placement while red indicates that placement is not allowed.

Wall Placement Input and Interface

The wall system supports drag based placement and repair using shape tools such as lines, rectangles, and circles. Shapes can be filled or hollow for flexible editing. The system uses Godot's input handling to draw shapes: the start position is set on `input_just_pressed`, updated while `pressed`, and finalized on `released`.

The 3D input interface for walls shares the placement cursor and price label used in structure placement. However, collision checking is handled with an array of individual collision areas rather than a single scaled collider as used for structures (see Section 6.9). Each area operates independently, and all must be empty (blue) for placement to be allowed. The price label is adjusted based on the number of missing tiles and their missing health. An example can be seen in the screenshot shown in the Figure 6.19.

Figure 6.19: Collision array showing overlapping physics bodies for a line shaped selection.

Spending and Cost Calculation for Buildings and Walls

Before placement, the client checks if enough resources are available by summing the player's inventory and the settlement's storage inventory. If no storage exists, only the player's inventory is used. The server repeats this check to validate the request.

If sufficient materials are found, they are deducted first from storage and then from the player's inventory if needed. This is handled by the inventory method explained in Section 6.14.

- **Buildings:** Each structure has a material cost defined in a `Dictionary` that maps structure types to required materials and amounts.
- **Wall Tiles:** The total cost is calculated based on the percentage of missing health for each selected tile. This allows wall repairs to follow the same process as building new walls in empty spots. While each buildings are treated as single instances rather than arrays.

Adding Walls and Structures

- **Wall Construction and Repair:** Once the shape passes collision validation, a build request is sent to the server. Each selected tile is either instantiated as a new wall or repaired by setting its health back to maximum. Health is tracked both in the tile instance and in the wall grid's `tile_health Dictionary` , indexed by the tile's `Vector2i` coordinate. This design allows interaction across gameplay systems like saving, loading, and damage tracking.

- **Structure Placement:** When a structure is placed, its position, rotation, type, and a generated ID are stored in the settlement's structure `Dictionary`. The corresponding packed scene is then instantiated in the game world with these parameters.

Removing Buildings and Walls

Walls and buildings can be removed using the same drag-based input as construction. The player selects a filled rectangle to define the area, and the start and end positions are sent to the server.

- **Walls**: The corresponding grid coordinates are processed to remove wall instances, update the grid arrays to `false` values, and re-tile the surrounding walls. The instanced wall tiles in 3D world are deleted using `queue_free` using the instance reference `Dictionary`. The navigation mesh is then rebaked to reflect the changes.

- **Structure**; The server identifies all structures within the selected area and removes them from the `Dictionary` using their pos. Each building instance is deleted using `queue_free` using their ID. Structures can also be destroyed when the health value in their `Condition` node reaches 0, following the same process after.

The figure below (Figure 6.20) shows the state before and after removing walls and buildings. The first screenshot displays the selected rectangle area before removal and the second shows the cleared result.

(a) Before: Selected rectangle area highlighted. (b) After: Removed buildings and wall tiles.

Figure 6.20: Before and after using the wall editing system to remove structures and walls.

Wall Tile

Wall tile is a scene responsible for selecting the correct tile `MeshInstace` and its `StaticBody3D` and `CollisionShape`. It is instantiated and managed by and as a child of the `Wall Grid` node. There is one instance of this node for every `true` value in the `wall_matrix` array. To reduce computation time when retrieving the correct tile instance, a separate 2D array called `matrix_instances` is used. This array stores direct references (or null values) to tile instances, allowing for constant-time access using index-based lookup. This approach is significantly more efficient than using a function like `get_node`, which may require a linear search through all wall tiles in the worst-case scenario, resulting in much higher time complexity.

The individual tiles are created in the 3D modeling software called `Blender`. Figure 6.21 shows how the tiles look in the Blender viewport.

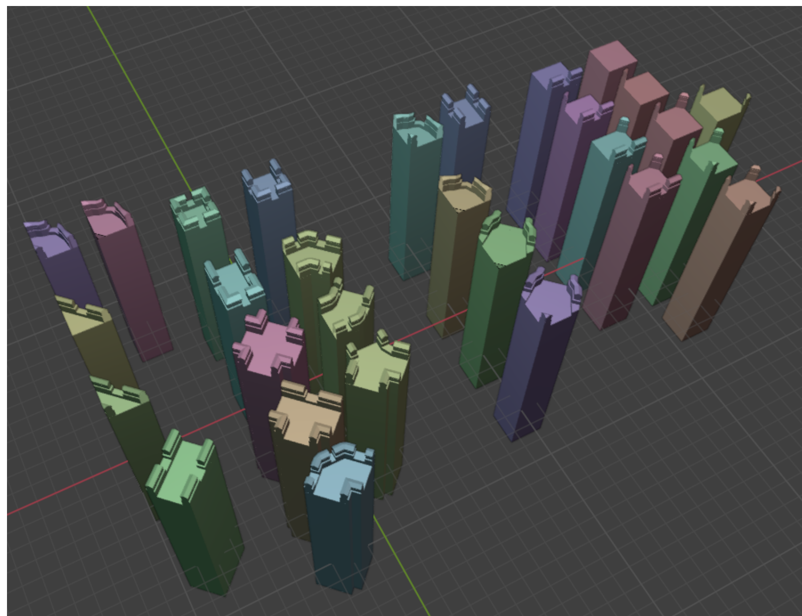


Figure 6.21: Each individual 3D tile mesh made in Blender.

Tiling

Tile selection is determined by the configuration of neighboring tiles. During tiling, each tile evaluates the number and positions of its orthogonal and diagonal neighbors (shown in Figure 6.22) to choose the correct visual and collision shape. The algorithm outputs a `PackedScene` containing the appropriate mesh, static body, and rotation. This ensures that wall segments connect seamlessly in all directions and behave correctly in both visual appearance and physical interactions.

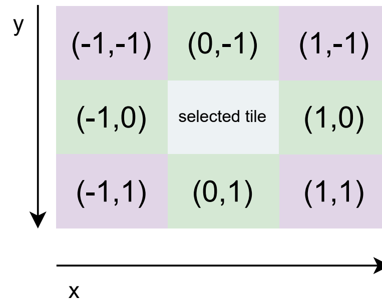


Figure 6.22: Tile shape is determined by the configuration of orthogonal (green) and diagonal (purple) neighbors used in the tiling algorithm.

Once selected, the corresponding `PackedScene` is instantiated with the correct Y-axis rotation and synchronized across clients. When a tile is placed, removed, or destroyed, the system re-evaluates only the eight surrounding tiles using `remake_walls_around_tile`, avoiding the need to reprocess the entire grid.

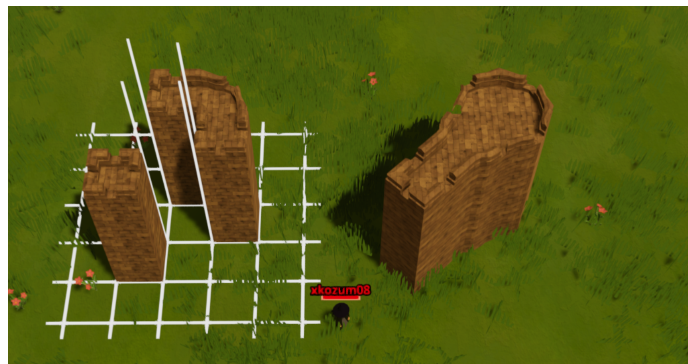


Figure 6.23: Example of tiling before and after adding a tile at the cursor position.

Diagonal Tiling

To enable building walls with a 45-degree angle, the diagonal neighbors must be incorporated into the algorithm. Without this, the tiling would appear as shown in the image on the left (see Figure 6.24a). After including the diagonal neighbors of a tile, the same tiled wall is adjusted as shown in the image on the right (see Figure 6.24b). This update is not only visual but also modifies the static bodies and subsequently the navigation mesh, allowing both players and AI characters to walk on walls that are 1x1 diagonally.

(a) Tiling issue caused by excluding neighboring tiles.

(b) Tiling after including diagonal neighboring tiles.

Figure 6.24: Difference between non-diagonal and diagonal tiling.

Wall Tile Health and Bulk Damage System

Each wall tile keeps track of its health and its position in the wall grid stored as `Vector2i`. The health is also saved in the wall grid's `tile_health Dictionary`, using the tile's position as the key. This way, the health information is available both inside the tile itself and in the overall grid, which helps with things like saving, loading and external queries.

The health of each tile is represented by its vertical position offset, as shown in Figure 6.25.

Figure 6.25: Tile health visualized by vertical offset after ability impact.

When a wall's health reaches zero and it is close to the ground, it is removed on the server, and this change is synchronized with all clients. The corresponding tile is also removed from the health tracking system to maintain consistency. After removal, the nearby wall tiles are updated to reflect the new configuration and ensure correct tiling.

Area of Effect Abilities and Wall Health

To handle area-of-effect (AoE) damage, where multiple wall tiles may be hit simultaneously, the system uses the functions `damage_tiles_bulk` and `remove_tiles_bulk`. These take arrays of coordinates and corresponding damage values, applying damage in a batch. This approach avoids redundant processing and improves performance.

Without bulk processing, the damage workflow for each tile would follow this sequence:

`apply damage` → `remove tile (if health = 0)` → `rebuild surrounding tiles`

For n damaged tiles, this would result in n separate wall rebuilds, which is inefficient.

In contrast, the bulk damage workflow proceeds as:

`apply damage to all tiles` → `remove all with 0 health` → `rebuild walls once around affected region`

This reduces the number of costly wall update operations and significantly improves performance, especially during large-scale damage events such as explosions.

Texturing the Tiles

The challenge with texturing each tile is the transition between each tile needs to be seamless.This is difficult and time-consuming because each side of every wall variant must smoothly transition into every other wall tile.

Instead of manually texturing each tile, a world tri-planar texturing technique is used in the mesh's material UV settings. An example of this texture can be seen in Figure 6.25.

This is a temporary solution since using the world tri-planar UV setting introduces two issues:

- The texture stretches if the plane is not aligned strictly at a 90-degree angle, causing distortion and artifacts on 45-degree angled wall tiles.
- The heightmap cannot be enabled when the tri-planar setting is active in Godot's material system, which makes the textures appear flat.

Building Towers into Walls

When a tower is selected through the building system, its placement area does not interact with the wall collision layer. Instead, any wall tiles within the tower's footprint are removed. The corresponding values in the `wall_grid` are set to `false`, and the associated wall instances are deleted. The number of tiles affected is determined by the tower's `Vector2i` size. Walls are then re-tiled around the affected area.

Building Spawners onto Walls and Towers

Spawner placement behaves similarly to structures. When selected through the building system, spawners ignore collision with walls, towers, or gates. Instead, the system checks the `wall_matrix` and `tower_matrix` for occupied tiles. If either matrix contains a `true` value at the selected location, the vertical position of the spawner is offset accordingly. This ensures that NPCs spawn on top of the wall or tower, rather than getting stuck inside the structure.

`tower_matrix` is a `bool` 2D matrix with the same size as, and aligned with `wall_matrix`. Each `true` value indicates that the corresponding tile is occupied by a tower or gate instead of a regular wall. This matrix is updated whenever towers or gates are built or removed from the settlement based on the the towers size and rotation.

Tiling to Towers and Gates

Walls tile to towers and gates using the same logic as with other walls. To support this, an additional `bool` matrix is maintained for towers as explained above.

When a tower or gate is placed, this tower matrix is updated based on the selected position and size of the object. Neighbor calculations are then modified to include both the `wall_matrix` and `tower_matrix`, allowing the tiling system to treat towers and gates as valid neighbors during tile selection.

Figure 6.26 below shows the difference between having towers excluded from and included in the tiling algorithm.

(a) Tiling without towers included.

(b) Tiling with towers included.

Figure 6.26: Visual difference between tiling with and without towers and gates included.

Structures Overview

Each structure uses a unique type from the globally accessible `Structure` enum to avoid duplication and maintain consistency.

All buildings described in Section 4.4 can be placed or destroyed. However, the residential houses, blacksmith, and farm are still under development and currently have no functional purpose. The other buildings are functional.

Structure Scenes

The scene structure of buildings closely follows the entity scene structure (see Section 6.3) used for players, NPCs, and other dynamic entities. The node structure is also described in Table 6.4 below.

Node	Description
`StaticBody3D`	Root node for physical interaction.
`  MPSynchronizer`	Keeps position, rotation, and scale in sync across the network.
`  Collision shapes`	Aligned with the visual mesh for accurate collisions.
`  Condition`	Manages health and damage, using the same system as characters.
`  Inventory`	Stores items in buildings that support storage.
`  NPCSpawner`	Creates non-player characters in residential or production buildings.
`  MeshInstance`	Represents the visual model of the building.
`  GPUParticles3D`	Adds effects like smoke, fire, or dust.

Table 6.4: Structure of building scenes and the use of their corresponding nodes.

An example in the Godot editor can be seen in Figure 6.27. Screenshot of more examples of worker buildings presented in Figure 6.32.

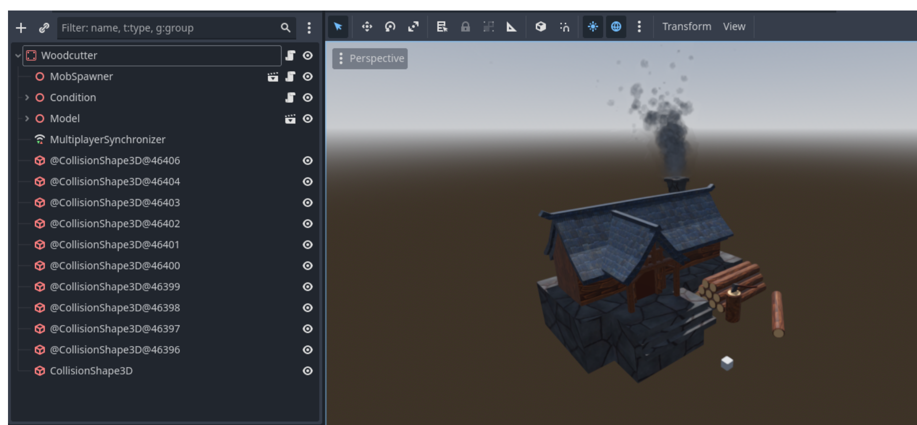


Figure 6.27: Woodcutter building scene made in Godot Engine.

Settlement

The **Settlement** scene acts as a container and manager for all buildable objects. Each instance defines its own area and coordinates. Structures and NPCs within a settlement share properties such as faction ownership.

The class provides loading, saving, placement, and destruction functionality using a scene `Dictionary`. It also supports multiplayer synchronization and tracks player interaction, updating UI and shared systems accordingly.

The **Settlement** scene contains the following nodes (see Table 6.5):

`Node`	**Description**
`MPSynchronizer`	Synchronizes core settlement state across clients
`NPC Spawners`	Container for networked entity generators (e.g., units or NPCs)
`Structures`	Containers for constructed structures like buildings, gates and towers
`MultiplayerSpawners`	Handles structure placement and network replication
`WallGrid`	Manages wall construction, snapping, and pathing grid
`Cursor`	Manages user interaction and placement preview
`Collision area`	Used to detect overlapping bodies
`Check Array`	Used to detect overlapping bodies for array of wall tiles
`Area3D`	Defines the spatial extent of the settlement

Table 6.5: Nodes contained in the `Settlement` scene and their descriptions.

Saving and Loading

Settlement data, including managed arrays and ID counters, is saved to and loaded from a `.save` file using Godot's `store_var` and `load_var` functions. Dictionaries for buildings, towers, gates, and spawners are grouped within a container `Dictionary`.

When loading, the save file is read and structures are instantiated via the settlement and wall grid systems. Loading can be customized by toggling `bool` variables such as `load_walls`, `load_buildings` or `load_spawners` in the editor. Settlements can be loaded by the players via the in-game chat command `/load save_name` or automatically at server startup using the `load_on_start` flag and related export variables.

Faction and Ownership

A settlement is associated with a specific faction. When the faction is set, the assignment is recursively propagated to all structures and characters within the settlement to maintain consistency. This faction information is to determine responses to abilities or other interactions based on team alignment.

6.10 Non-playable Characters

The structure of non-playable characters (NPCs), like the player scene, is based on the entity structure described in Section 6.3. Any action that an NPC can perform is influenced by its status effects, which are managed through the `Condition` component—just like in the `Player` scene. For example, actions may be limited when the unit is stunned, frozen, slowed, or dying. The NPCs are separated into different categories based on their behavior:

- **Guard**: Defensive units including castle guards, archers, bandits, wildlife, and bosses.
- **Worker**: Further divided into three specific behavior types:
  - **Resource Gatherer**
  - **Artisan**
  - **Farmer**

Figure 6.28 shows guard units spawned using NPC spawners (see Section 6.10) on the ground, walls, and gate. Spawning on towers, walls, or gates is further explained in Section 6.9. The guard units also display orange faction representation and have a different set of abilities (see Section 6.11), which is also represented by their equipment.

Figure 6.28: Guards spawned using the NPC spawner on ground, walls, and gates.

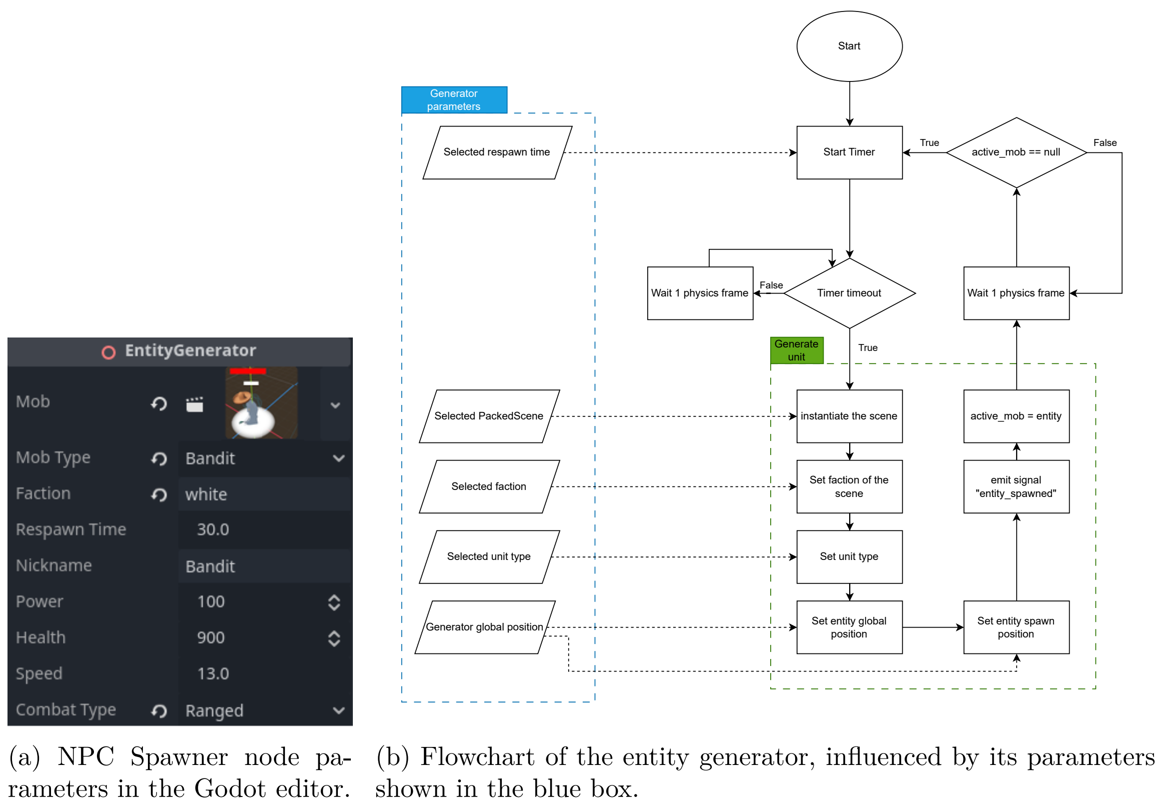
Faction Representation

AI faction is shown by the mesh material color. To avoid shared colors between instances, call `set_local_to_scene(true)`, making a deep copy of the material. Otherwise, all units use the last changed color. This also applies to shared resources like health bar textures.

NPC Spawner

Units need to be instantiated during runtime, with the ability to be de-spawned and have their de-spawn events synchronized across all clients. Additionally, some entities require respawning. Others need predefined positions on the map or within other scenes (e.g., a building scene that spawns a worker unit), placed directly in the editor. NPC spawners can also be placed by the player using the building system interface.

For these purposes a custom scene was built named `EntityGenerator`, with the parameters displayed in Figure 6.29a using the `@export` annotation in its script before the variable declaration. The flowchart of the NPC Spawner runtime behavior is presented in the Figure 6.29b.



(a) NPC Spawner node parameters in the Godot editor.

(b) Flowchart of the entity generator, influenced by its parameters shown in the blue box.

Figure 6.29: EntityGenerator configuration and its runtime behavior.

Guard Unit

Guard is the base behavior for the majority of NPC units, defining their scene tree structure and implementation. Various NPC types inherit this behavior by modifying parameters, either through the NPC spawner (see Section 6.10) or by using fully custom scenes. This design supports a wide variety of enemies with different characteristics and abilities.

Unit examples:

- **Close-combat units**, like knights, have low attack range abilities.
- **Archer units** have high detection and attack range but stay near their spawn point by limiting their maximum distance from it (e.g., a castle wall or tower).

Guard unit behavior is described by the state machine shown in Figure 6.30 below:

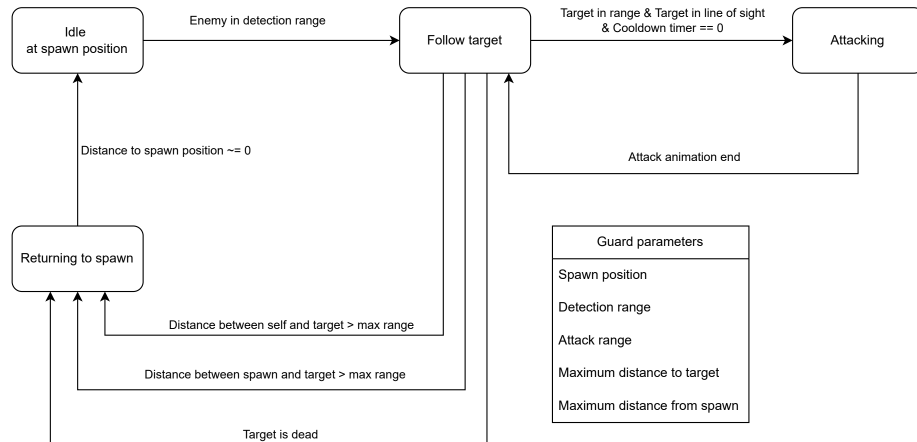


Figure 6.30: Guard behavior state machine with parameters. The spawn point is a 3D global position where the entity is instantiated, idles, and returns to.

Detection and Following

An NPC unit can become hostile towards both playable and non-playable characters when its `Area3D` node detects overlapping bodies that do not share the same faction value and when the unit is not already provoked. Detection is based on the `Area3D` node's collision shape (see Figure 6.31), whose size is scaled by a detection range parameter. Additionally, the NPC can set a target if damaged by another entity and if it is not already provoked. After that, the character follows the target until it is within the range of its next ability and also in direct line of sight (see Section 6.7).

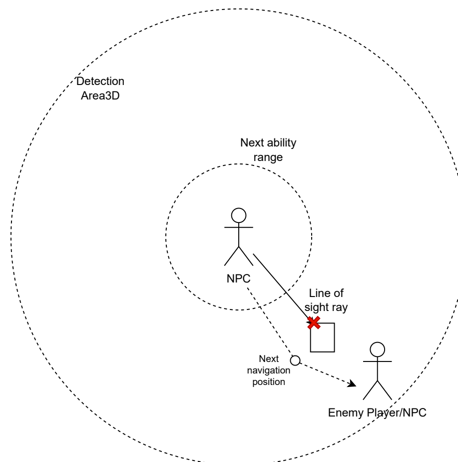


Figure 6.31: NPC combat detection using `Area3D`, pathing with navigation agents and line of sight, and attack range of the current ability.

6.11 NPC Combat System

NPCs use the same spells and abilities as players, as spell and ability scenes are independent of the spellcasting systems. Spells are instantiated on the server with references to the caster and, if applicable, the target, along with parameters such as spawn position, direction, and velocity when needed.

Spell Casting Parameters

Each spell includes a cast time and range, similar to the player system. These values are stored in dictionaries indexed by the global spell enum ID.

This combat system is tied to the guard behavior system 6.30, mostly the attacking and following states. The next Table 6.6 shows the casting process.

Step	Description
1. Condition Check	Verifies NPC is not stunned, dead, or frozen.
2. Line of Sight Check	Ensures the target is visible. Otherwise, returns to Follow state.
3. Start Cast Timer	Begins casting the selected spell using its configured cast time.
4. Display Cast Bar	Server sends RPC to clients to show the cast bar above the NPC.
5. Interrupt Check	Continually checks if casting should be stopped due to status or distance.
6. Spell Execution	When the timer ends, the spell is instantiated and cast at the target.

Table 6.6: Steps of the spell casting process during attacking.

Spell Combinations

NPCs use predefined spell combinations tied to their combat type (see Table 6.7). A random combination is selected and executed in sequence. Between spells, the NPC may reposition to meet range requirements. Once a combination is completed, a new one is chosen.

Combat Type	Example Spell Combinations
Magic	[Tornado, Sunstrike, Meteor], [Fireball, Lava Pillar]
Ranged	[Shoot Arrow]
Melee	[Slash, Slash2, Stab]
Boss 1	[Fireball, Ice Nova, Heal Nova], [Lava Pillar, Ground Slam, ...]

Table 6.7: Example ability combinations by combat type stored in a 2D array of abilities.

6.12 Workers

There are currently three different behavior types of workers: resource gatherer, farmer, and artisan. Each type is further divided into subtypes (e.g., resource gatherer includes miner and woodcutter). All workers inherit their structure and scripts from the base NPC guard. When provoked (for example, attacked by another entity), they switch to the guard behavior described in Section 6.10 and remain in this mode until they reach the `Returning` state, at which point they switch back and resume their original behavior.

Each worker type also has different 3D meshes attached to the main meshes skeleton using the `BoneAttachement3D` node. These meshes represent the tools and clothing (mainly hats) which differentiates between the workers. This is done similarly to the player equipment representation from this section 6.14

Worker Buildings

Workers are usually generated by NPC spawners (explained in Section 6.10) that are placed inside their dedicated building scene. Each worker type has a unique building based on both the general category and the specific subtype, such as resource gatherers including woodcutters and miners having distinct buildings.

Figure 6.32: Game screenshot of worker buildings placed in the settlement.

Storage

Each worker stores gathered or produced items in a settlement's `Storage`, which uses an `Inventory` component (explained in Section 6.14) derived from the same base as player and NPC inventories. Storage buildings must be constructed by players and can be destroyed.

If no storage exists, workers either wait or continue working and store items in their own `Inventory` until one becomes available. This logic is handled within the worker behavior state machines.

General Worker Behavior

When entering the `Working` state, a `Timer` is started based on unit type. `AnimState` is set to `Working`, triggering the `AnimationPlayer` to animate the `Skeleton3D` to simulate working.

The animation stays locked until the timer ends or is interrupted by a higher-priority event, such as a stun.

When the `Timer` times out, resources are added to the unit's `Inventory`, the animation lock is released, and the next behavior begins.

Resource Gatherer

Resource gatherers are units that detect and collect materials, then store them in the inventory of a designated storage building. Currently, there are two types of gatherers: **Miners** and **Woodcutters**.

As described in Section 6.15, these materials include trees and mineral deposits such as stone. Detection is performed using the `Area3D` node, which identifies nearby resource nodes. The nearest unoccupied node is selected and marked as occupied until the worker finishes the task.

The behavior of gatherers is defined using a state machine, as illustrated in Figure 6.33.

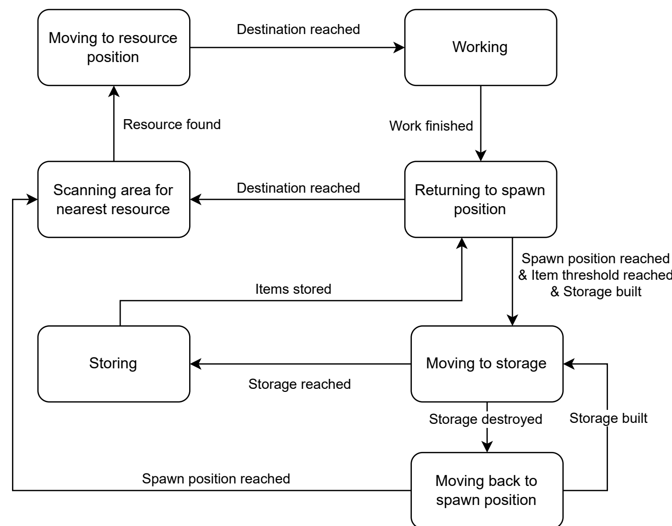


Figure 6.33: Gatherer behavior defined using state machine.

Farmer

A farmer worker type which cycles between working and moving to another crop. A crop is a `Vector3` position. When the farmer reaches crop position, state is switched into `Working`. New items are generated after each time the `Working` state is successfully finished. After attending to all crops within his farm building, the worker moves to the settlement's storage, where he stores the generated resources. The overall behavior of the farmer is illustrated in Figure 6.34 below.

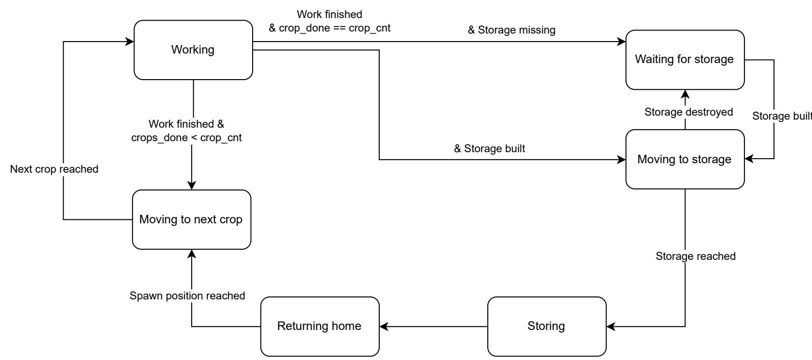


Figure 6.34: Farmer behavior defined using state machine.

Artisan

The `Artisan` is a worker that converts raw materials into other items. It first checks the settlement's `Storage` for the required materials. If unavailable, it enters a waiting state and periodically rechecks the inventory.

Once materials are available, the artisan moves to the `Storage` building and revalidates item availability. If items are still missing (e.g., taken by others), it waits nearby. When the materials are successfully retrieved, the artisan returns its building, switches to the `Working` state, and starts a `Timer` for the crafting process. Subtypes of the artisan include mill workers and blacksmiths. However, blacksmiths are not yet implemented. The behavior of these workers is defined using a state machine, illustrated in Figure 6.33.

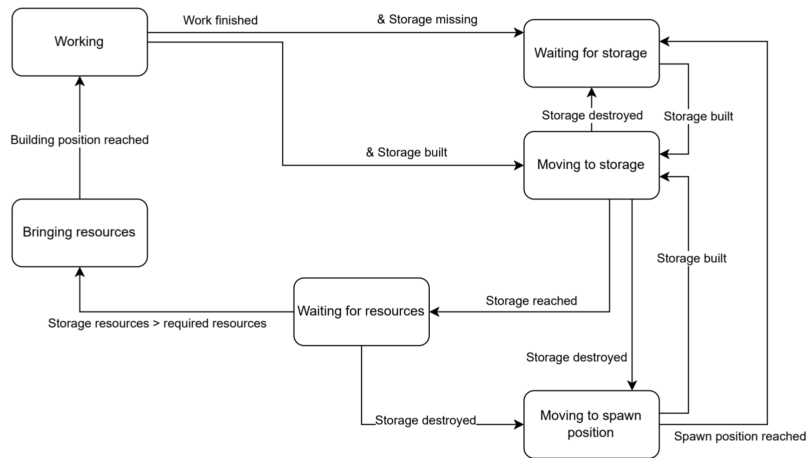


Figure 6.35: Artisan behavior defined using state machine.

Bandits, Bosses and Wildlife

In addition to faction NPCs in settlements, the game world features independent characters like bandits, wildlife, and bosses. These are not tied to settlements and belong to separate factions. They spawn at predefined locations using `NPC Spawners` placed in the Godot editor. All follow the `Guard` behavior type (see Section 6.10). They vary in stats (health, power, speed, combat style) and use distinct 3D models for bodies and equipment.

6.13 Navigation

Navigation is used exclusively by non-player characters, as player characters use direction based movement with physics and jumping. NPCs move using Godot's navigation system, which handles path-finding and obstacle avoidance. The official Godot documentation [2] also includes a written tutorial on the navigation system.

Navigation Region

`NavigationRegion3D` defines the navigation mesh for the `NavigationServer3D`. Any static body that should be included in navigation must be a child of the navigation region in the scene. The navigation mesh can be baked in the editor or at runtime using the `bake_navigation_mesh` function. Figure 6.36 below shows an example of a baked navigation mesh on terrain and buildings.

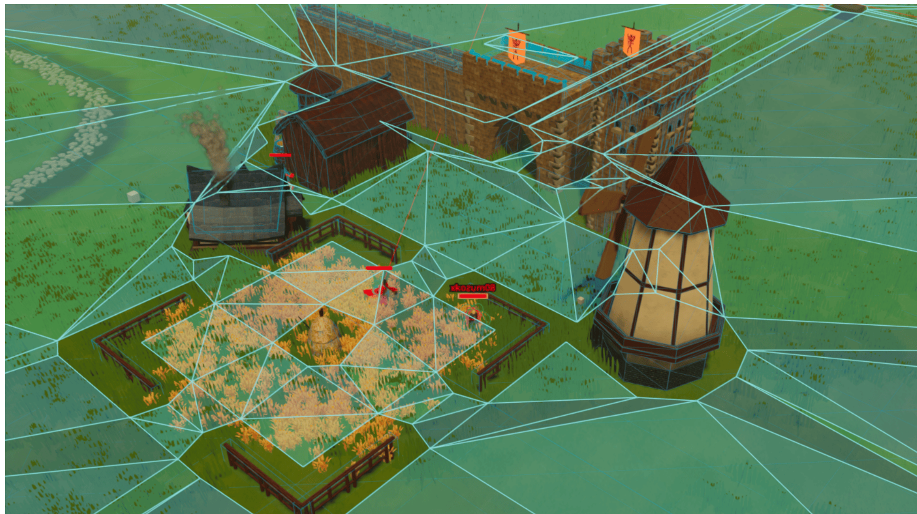


Figure 6.36: Baked navigation mesh displayed as blue mesh in the game's debug build.

Navigation Agents

Navigation agent is a scene child of every character with navigation, it helps with avoidance and API calls to the `NavigationServer3D`. The avoidance setting on the navigation agents helps to prevent direction collision with other agents or other moving obstacles. Other notable parameters of the navigation agent is the `Target Desired Distance`, which is a threshold distance before the target is considered to be reached.

Rebaking the Navigation Mesh

The world in this game is dynamic and changes throughout the game. Trees, buildings, towers, walls, and other environmental objects can be destroyed and rebuilt. The navigation mesh must be updated accordingly. otherwise, AI characters might attempt to walk through walls or avoid obstacles that no longer exist.

Navigation can be rebaked using the `bake_navigation_mesh` function of the navigation region. Because this is a performance-intensive operation, it must be executed on a separate thread. Godot supports this natively through the `on_thread` parameter of the function.

This setup requires the server to have at least two threads. If not, the game stutters for a few seconds each time the navigation mesh is rebaked.

Navigating the Character

To reach a destination, the character uses Godot's navigation system, which handles pathfinding and dynamic avoidance. A target position is set for the navigation agent, which calculates a path and determines the next point to move toward. The direction to this point is used to compute a velocity, which is passed back to the agent for adjustment based on nearby obstacles and agents. The character is then rotated toward the direction of movement. The final safe velocity is returned asynchronously by the engine and applied as force during the physics update.

Optimizing the Navigation

Since the `get_next_path_position` function of the navigation agent has a relatively high computational cost (and is used by many characters) it is not called every physics frame, as this is unnecessary. Instead, a flag is used to control updates, which is set to `true` by a timer every 0.5 seconds. This introduces a slight delay in character response when the navigation mesh changes, such as when a obstacle is placed in front of them. However, this delay does not cause issues because characters still collide with obstacles due to their `PhysicsBody`. And even in real life, human reaction time is not instantaneous.

Batching Rebake Requests

Very often, when a building or wall is constructed or destroyed, additional objects are placed shortly after. For example, a player might build a wall or building three times within a few seconds. Instead of rebaking the navigation mesh after each individual change a timer is started at the first request, predicting the following actions. While the timer is active, any requests are ignored. Once the timer times out, the navigation mesh is rebaked, which includes all of the previous changes. An example illustrating this process is shown in Figure 6.37, which depicts example of the actions in time and the corresponding rebake event.

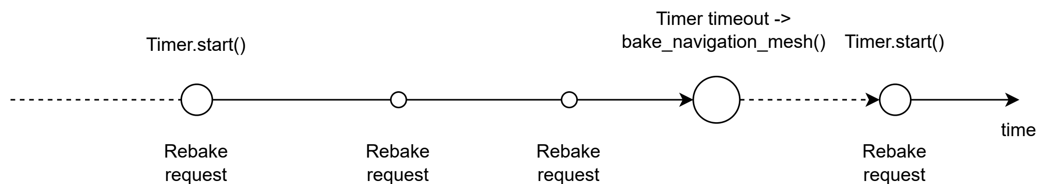


Figure 6.37: Illustration of batching rebake requests showing multiple construction actions within a short time frame and a single navigation mesh rebake occurring after the timer expires.

6.14 Items and Inventories

Items are identified by a unique string ID. A global `Dictionary` in the `ItemManager` singleton stores all item properties. Inventories, item instances, and scripts reference items only by their ID and count, e.g., `{„name": „Stone", „cnt": 10}`. The `ItemManager` handles access to item data and includes utility functions for item spawning, queries between two inventories, and other item-related logic. All transactions are being handled by the server to avoid cheating by manipulating the dictionaries on the client side.

The item properties are stored in a `Dictionary` indexed by item ID. Each item has the following attributes (see Table 6.8):

Key	**Description**
`item_name`	Display name of the item
`scene_3D`	Associated 3D scene for world instance
`icon`	Path to the inventory icon texture
`is_resource`	Whether the item is a raw resource
`is_stackable`	Can be stacked in inventory
`max_stack`	Maximum count in a single stack
`is_equipment`	Can be equipped
`equipment_class`	Equipment category (e.g., Weapon)
`spells`	Spells unlocked by equipping this item
`stats`	Dictionary of stat bonuses (e.g., Power)
`model`	Reference to model used as 3D equipment

Table 6.8: Item property keys and their descriptions used in the item data structure.

Item Instances

Item instances are 3D scenes consisting of nodes described in the Table 6.9. Each instance holds an item ID and count, and its scale reflects the stack size. It also enables detection by the inventory system through collision layers monitored by an `Area3D` node. The physics body allows interaction with the world. For example, thrown stones deal damage based on their velocity and scale.

Node	**Purpose**
`RigidBody`	Handles physics interactions
`CollisionShape`	Enables interaction and pickup detection
`MPSynchronizer`	Syncs position, rotation, count and item name across clients
`MeshInstance`	Visual representation of the item
`Label3D`	Displays the item's name in the world
`Area3D`	Used to detect colliding entities and apply damage to them

Table 6.9: Nodes composing item instances and their functions.

Figure 6.38 shows a screenshot of multiple instanced items, including both equipment and materials, in the game world.

Figure 6.38: Screenshot of instanced items in the game world.

Item Stacks

Multiple identical items are combined into a single inventory slot or item instance with a count to improve performance and organization. Item instances in the 3D world each represent a full or partial stack. Items that can be stacked are defined through their properties in the global item `Dictionary` managed by the `ItemManager` singleton. As shown in Table 6.8, the stackability and maximum stack size are defined for each unique item.

Without item stacks, managing large numbers of instances would cause problems. For example, destroying a storage building containing 1,000 separate items without stacking could lead to severe lag or even crash the server.

Figure 6.39 shows stone item instances with different counts: on the left, two stones holding only a small fraction of a stack, and on the right, a stone representing a full stack (64 stones).

Figure 6.39: Stone item instances with different count: small stack size on the left and full stack on the right.

Inventory

The inventory is a class used by players, AI characters, and buildings and allow unified system for item management of these entities. Items in inventories are stored as an `item_list` array of dictionaries, with each entry representing a stack of a specific item name and its quantity. Inventories support adding, spending, and dropping items. Player inventories expand on this by allowing item pickup, dropping via the user interface, and management of equipment.

Adding Items

When adding an item, the system retrieves its item info using the `ItemManager` to determine if it can be stacked. If stackable, it fills existing non-full stacks in the `item_list` before creating new ones for any remaining quantity.

Spending Items

To spend items, the inventory system iterates through stacks in the `item_list` of the specified item, decreasing their quantities or removing them entirely until the required amount is deducted. It also calculates the total available quantity beforehand to ensure enough items are present. This process is primarily used for paying the cost of building structures (see Section 6.9).

Dropping Items

When an entity's health reaches zero, all of its carried items are dropped into the game world. The items are removed from its inventory and instantiated as 3D scenes at the entity's position. If a specific 3D scene exists for the item, that scene is used. If no matching scene is found, a placeholder is spawned instead.

These item scenes are added under a designated world node responsible for managing items. This node is monitored by a multiplayer spawner that replicates all newly spawned item instances across clients in a multiplayer session.

Synchronizing Inventory

Inventory synchronization is managed using signals and remote procedure calls (RPC). When the server's inventory changes, it emits a signal with the updated item list. This triggers a remote call that updates the item list on all clients. Upon receiving the updated list, clients emit the same signal locally, allowing interfaces such as the inventory UI to refresh immediately without the need to poll for changes.

Player Inventory

The player inventory class extends the inventory class and adds functionalities specific to players. It introduces functions for picking up and dropping items, equipment slots and its statistics and 3D representation.

Picking up Items

Item pickup is handled using an `Area3D` node that monitors a dedicated collision layer for item instances in the game world. When the player triggers the pickup action, a server-side

function is called remotely to process the pickup. This function identifies all items within the pickup area, adds them to the player's inventory (by merging stacks), and removes the items from the world. Finally, an inventory update signal is emitted to synchronize other systems like the user interface.

Dropping Items

Item dropping is initiated through the inventory UI. When a player with authority triggers a drop, the client remotely calls the server's drop function with the selected item. The item is then spawned in the game world using the `ItemManager`, with its position randomized around the player's body.

The drop-all function, used when the player dies, is customized to exclude items marked as equipment (those with `is_equipment` set to true in their item info).

Inventory UI

The inventory UI uses `Control` nodes with an `ItemList` to display items and `Button` nodes for actions below the list. Callbacks connected to the buttons handle item selection and equipment management. Both item icons and action buttons use icons from the preloaded item info `Dictionary` (see Section 6.8).

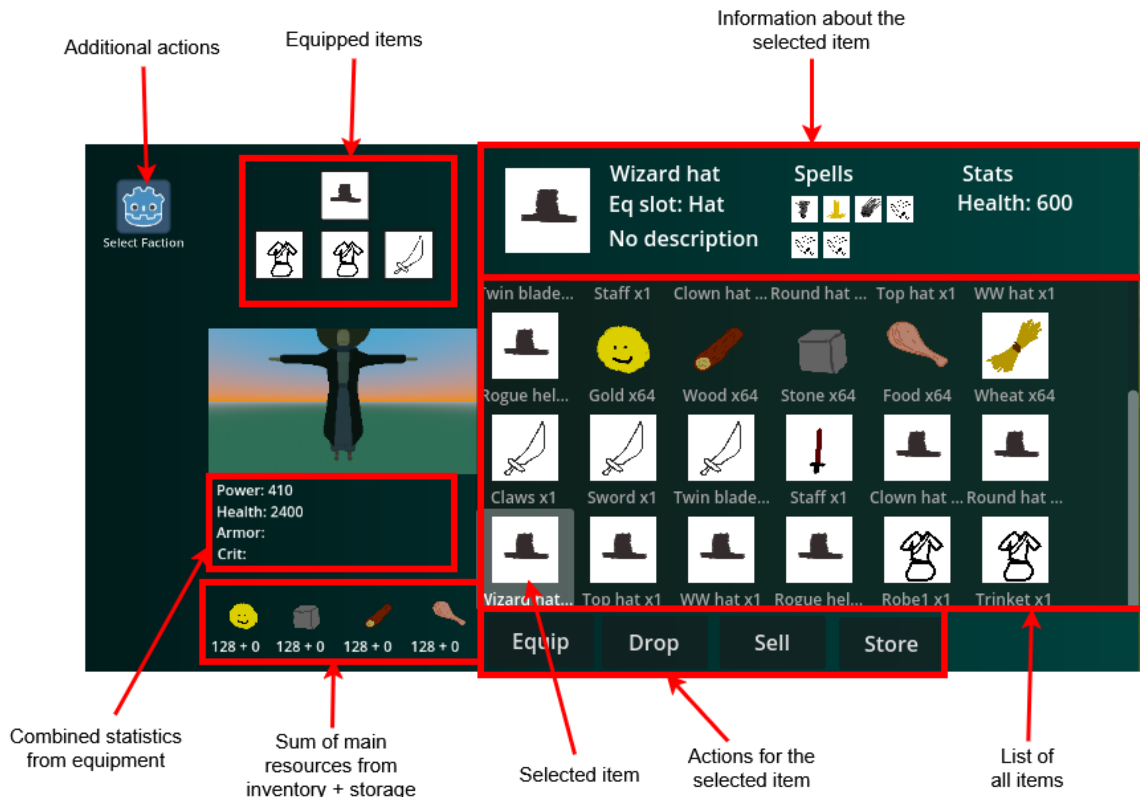


Figure 6.40: Screenshot of the inventory user interface showing example items and equipment

Equipment

The player inventory introduces equipment slots for items such as weapons, hats, trinkets, and chest armor. Players can equip or unequip items through the inventory UI, similar to how items are dropped into the game world (see Section 6.14). Equipping items affects player stats like health and power and can unlock spells associated with those items.

When an item is equipped, the `recalculate_stats` function is called. It iterates through all equipment slots, aggregates stats (e.g., health, power), and emits a signal with the updated values. This signal is connected to functions in the `Condition` class and user interface, which update the player's health, health bar, and inventory stat labels.

Equipped items also determine the player's available spells and abilities. This is handled by iterating through all equipment slots and collecting unique spells listed in each item's `spells` field.

3D Representaion of Equipment

Equipment items are attached to the bones of the 3D character mesh using the `BoneAttachment3D` node. The mesh instances for different equipment items are already placed under their respective bone attachments, but their visibility is initially set to false. When an item is equipped, the relevant 3D models become visible and the change replicated through RPC. Each equipment mesh is assigned a unique ID using an enumeration, and a `Dictionary` maps these IDs to arrays of scene paths for the associated 3D models (see Section 6.8). This allows support for equipment that includes multiple parts, such as dual weapons or shoulder armor attached to different bones. Figure 6.41 shows examples of different visualized 3D equipment sets.

Figure 6.41: Screenshot showcasing three different equipped hats and weapons.

6.15 The World

Apart from players, NPCs, and castles, the world also includes various resource nodes, trees, and material deposits that can be interacted with to simulate resource gathering. The terrain is modeled in Blender using custom meshes and shaders. Foliage such as grass and paths is generated with Godot add-ons using multimesh techniques.

The game world uses Godot's `WorldEnvironment` node for lighting, including volumetric fog to add depth and atmosphere.

Resource Nodes

Resource nodes are entities from which materials can be collected by players and workers. Players can attack these objects directly, causing the corresponding items to be instantiated near the resource's position. Workers simulate collection by playing a working animation and triggering item creation through scripts. They are static resource entities that include `StaticBody` and `Condition` components (see Section 6.3).

Two main types of resource nodes currently exist: trees and material deposits.

- **Trees**: When destroyed, they spawn wood items and update their model and collision shape to represent stripped bark. They also exist on a separate collision layer, so not all spells affect them. Tree destruction triggers navigation mesh recalculation due to the change in walkable space.
- **Material deposits**: Spawn resource items, such as stone, when attacked by players or NPCs.

Figure 6.42: Destroyed trees with wood items instanced nearby.

Figure 6.43: Stone deposit with stone items dropped after damage.

Terrain Mesh and Shader

The Godot Engine does not include a built-in terrain system. Although several add-on solutions are available, none were suitable for the requirements of this game due to performance limitations or missing features such as navigation support and level-of-detail functionality in multiplayer environments.

As an alternative, a terrain mesh was created in Blender using the Terrain Sculpting Tool add-on from Kitfox. This add-on includes features such as ground leveling, ramp creation, and constraining sculpting actions to move perpendicular to the surface.

For texturing, a splat mapping method was used, where a control texture (splatmap) blends up to four terrain textures based on its red, green, blue, and alpha channels. The shader blends these textures using gradients, which is sufficient for creating simple terrain. This method was implemented in Godot's shader language via a visual shader, following the approach demonstrated in a video tutorial [4].

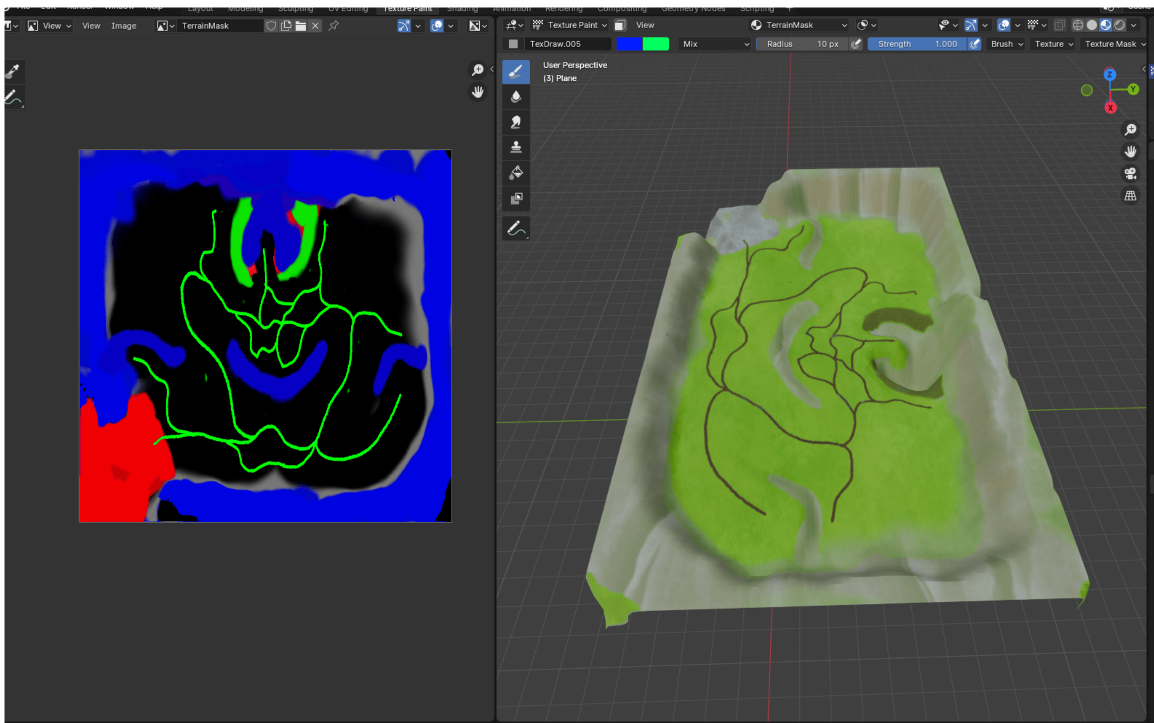


Figure 6.44: A 3D terrain mesh in made Blender and textured using a splatmap shader.

Grass

To generate grass, a `MultiMeshInstance3D` node is used. This approach improves performance by combining many mesh instances into a single draw call, reducing CPU overhead and allowing the GPU to handle the rendering more efficiently.

The grass is created using the `Simple Grass Textured` addon for the Godot engine, which allows painting a grass `Multimesh` directly onto `StaticBody` nodes with selected `Collision Layers`. An example of the resulting foliage is shown in Figure 6.45.

Figure 6.45: Foliage rendered with a MultiMesh using the Simple Grass Texture addon.

Paths

In addition to the paths created using the terrain texture, a `Proton Scatter` Godot addon is used to generate road `multimeshes` with a `Path3D` node and stone path assets from the `Stylized Nature Mega Kit` by Quaternius, as shown in Figure 6.46. These assets and the scatter addon are also used for other foliage elements, such as flowers, bushes, and stones.

Figure 6.46: Generated paths using Proton Scatter and Stylized Nature Mega Kit assets.

6.16 Game Chat

Players can communicate through a built-in game chat system. Text input is handled via a `LineEdit` node. When visible and the `Enter` key is pressed, the input is sent to the server along with the client's unique peer ID using a remote procedure call.

The server then broadcasts the message to all clients. Each client retrieves the sender's nickname using the peer ID and displays the message as a `Label` within a `BoxContainer`. The chat system implementation in this project was inspired by a tutorial example [7] by Rayuse.

Commands and Cheats

When the server receives a message, it first checks whether the first character is a `"/"` to determine if the message is a command. If it is, the text is split: the first word is treated as the command name, which should match a corresponding function, and the following words are used as the function's parameters. This works similarly to how commands are handled in a Linux command line.

Chapter 7

Testing

Since this project is a game, testing has been done continuously, with local tests run after almost every small change or tweak. The game was likely launched and closed hundreds of times each day during development. Fortunately, `GDScript` is an interpreted language, and the `Godot Engine` allows the game to run without needing to recompile or export, which greatly sped up the process.

Debugging

To test and debug performance issues, Godot's built-in debugger and profiler were used (see Section 3.9). The profiler displays, for example, how much processing time (CPU and GPU) each function took during a selected frame. A network profiler was also utilized to monitor download and upload activity, incoming and outgoing RPCs, and the scene paths from which they were sent. Additionally, it tracks the number of objects, total objects drawn, video memory usage, RAM usage, and navigation data. A lot of features were also debugged using `print` statements.

Play Tests

To test larger updates, the game server was hosted remotely and the game executable was sent to testers. The number of testers at a given time ranged from three to five. For both limit testing and financial reasons, the worst available server option was chosen for the first playtest. This server had a shared CPU with only a single thread available. Despite this limitation, the game did not experience any major performance issues or crashes during an hour-long session, with one exception. The main problem was that the shared single-threaded CPU caused the game to freeze for one to two seconds whenever a player built or destroyed a structure, due to navigation mesh rebaking. The conclusion from this test is that the game server requires at least two dedicated threads to run smoothly. Otherwise, the game becomes nearly unplayable.

One other issue was forgetting to turn off debug prints, which ended up causing more performance problems than anything else in the game. It was an easy fix once they were disabled.

For testing, the Linode service with an Ubuntu server has been chosen for the hosting server and was set up using this tutorial [5].

7.1 Main Complaints from User Testing

Aside from the many bugs that are continuously being found and fixed, testers have also raised some significant complaints about the game:

Inventory and UI

The build and combat menus display only keyboard shortcuts, and the buttons are not clickable with a mouse, which may lead to confusion. The inventory interface feels unintuitive, requiring users to manually select an item and then press an additional button to perform any action. The user interface also appears visually unrefined and could benefit from improved design and usability.

Balancing

There are various balancing issues across all systems. For example, some items are too powerful, spells are too weak or too impactful, and building costs can be either too high or too low. Health values for buildings, trees, players, and NPCs are also inconsistent. These values are currently placeholders and will be continuously improved through future testing.

Item Drops

Currently, equipment drops from defeated enemies are overly randomized. All items have an equal chance of dropping, and all NPCs, including both regular enemies and bosses, have the same drop rate.

Manually Downloading New Versions

When a significant bug was discovered or testers requested changes or new features, they were required to delete the old files and re-download the entire game. Additionally, there is currently no system in place to verify that the server and connected client are running the same version of the game

To address this, a game launcher is currently in development, which will automatically download updated versions.

Lack of Error Messages

Some error messages related to the combat system, such as `'target too far away'`, `'out of line of sight'`, or `'ability on cooldown'`, are implemented individually. However, these messages are currently based only on client-side checks. For example when a player does not have the resources to build a structure but receives no feedback from the server. A unified system, similar to the existing chat system, will be needed to handle both server-side and client-side error messages.

Lack of In-game Tutorial

New players needed guidance on how to control the game, particularly the spell-casting system, which is more complex than simply pressing a single button. The main objective and other systems also required some clarifications.

No Feedback on Getting Attacked

Aside from the health bar, there is no visual or audio feedback when the player takes damage or is in a low-health state. This can sometimes lead to the player dying without realizing they were in danger.

Navigation

The NPCs, primarily workers, often get stuck on certain obstacles, despite the navigation mesh being updated with every map change. This issue typically occurs near corners and can hopefully be resolved by testing and adjusting the `Agent radius` and other parameters of the `NavigationRegion3D` node.

Desynchronization Issues on Late Connection

Most of the local testing during development was done by manually starting the server, launching the client, and testing features in a fresh session. However, during larger playtests, a major issue was discovered: When players joined a session with a previously modified game state, such as built structures, destroyed trees, or renamed elements, some data was not properly synchronized. This affected systems like tile placement, player display names and equipment, and tree states, where trees appear uncut on the client even though they are cut on the server. While many of these issues are already fixed, it is likely that some still remain undiscovered.

Mouse Driven Movement

Due to feedback and the project's shift toward a more physics-based gameplay, mouse-driven movement was replaced with WASD controls. Consequently, the magic casting system was also updated to support the new mouse gesture based magic system.

Problems with Resolutions

Although the user interface uses anchors for dynamic positioning rather than fixed pixel values, it does not currently scale with the selected screen resolution. This causes issues on high-resolution displays, such as 4K. Addressing this issue is currently limited by the hardware constraints of the development setup, which does not support testing at such resolutions.

Chapter 8

Conclusion

The game developed in this thesis successfully combines real-time strategy and action role-playing mechanics in a functional 3D multiplayer prototype. All designed core systems from Chapter 4 have been implemented mostly using open-source tools, demonstrating the viability of building complex multiplayer games as a individual developer.

The current version is fully playable as a sandbox where players can experiment with building, combat, and progression systems. While structured game modes are still in development, the existing core systems offer a strong foundation for future features. Combat includes 25 abilities with physics-based effects. The building system supports a variety of structures and auto-tiling walls, with all structures being fully destructible. The world also features destructible trees, interactable objects, and custom terrain. Items affect player statistics, unlock abilities, and serve as projectiles or building materials, and equipped items are visually represented on the character. NPCs simulate a range of worker and soldier behaviors and share the same ability system as players. Their behavior is configurable through spawners that set properties like speed, health, and role. Supporting features include chat, a command console, and a launcher.

Despite the progress, the project still faces challenges. Known limitations include bugs, user experience issues, and the absence of structured content, as discussed in Section 7.1. These are expected at this stage, especially given the project's scope and its development by a single person.

Moving forward, the focus will shift toward testing, bug fixing, refining the player experience, and gradually adding content such as new items, NPCs, and buildings. The core systems already in place simplify the process of expanding the game's features and content.

While a full release remains a long-term goal, the progress so far demonstrates that ambitious multiplayer games can be developed independently using free and open-source tools. The project lays a solid foundation for future development and will continue to evolve with continual iteration and feedback.

Bibliography

[1] Buckland, M. *Programming Game AI by Example.* 1stth ed. Sudbury, MA: Jones & Bartlett Learning, 2004. ISBN 1556220782.

[2] Godot Engine Contributors. *Godot Engine Documentation.* Godot Engine, 2025. Version 4.x, available at https://docs.godotengine.org/.

[3] Gregory, J. *Game Engine Architecture.* 3rdth ed. Boca Raton, FL: A K Peters/CRC Press, 2018. ISBN 9781138035454.

[4] McKay, M. *Making a Terrain Shader for a Game Engine* https://www.youtube.com/watch?v=hpfxUDPxVZ8. December 2022. Accessed: 2025-05-12.

[5] Mitch Makes Things. *Godot Server Cloud Hosting* https://www.youtube.com/watch?v=xl3ddqsh6VM. March 2022. Accessed: 2025-05-12.

[6] MitchMakesThings. *Godot-Things: Networking Tutorials and Mini-Projects for Godot Engine* https://github.com/MitchMakesThings/Godot-Things/tree/main/Networking/Explained. March 2023. Accessed: 2025-05-12.

[7] Rayuse. *Godot-TopDown-Multiplayer-Game: A Top-Down Multiplayer Shooter Game* https://github.com/Rayuse/Godot-TopDown-Multiplayer-Game. May 2021. Accessed: 2025-05-12.

[8] Salen, K. and Zimmerman, E. *Rules of Play: Game Design Fundamentals.* 1stth ed. Cambridge, MA: MIT Press, 2003.

[9] Schell, J. *The Art of Game Design: A Book of Lenses.* 1stth ed. Boca Raton, FL: CRC Press, 2008. ISBN 97801236949669780123694966.

[10] SteamDB. *Steam Database* https://steamdb.info/. Accessed: 2025-07-11.

[11] Zukalous. *What are the top selling indie games of 2023?* 25. january 2024. Available at: https://howtomarketagame.com/2024/01/25/what-are-the-top-selling-indie-games-of-2023/. Accessed: 2025-07-12.

Appendix A

Contents of the included archive

- `README.md`: Documentation including instructions for running and importing the game, archive information, and game controls
- `demo_videos/`: Demo videos including multiplayer playtest footage and narrated demonstration
- `exported_game/`: Exported and compressed game executables
- `source_files/`: Compressed source code files
- `overleaf_source/`: Compressed LaTeX source files
- `xkozum08_thesis.pdf`: Compiled thesis document

Appendix B

Used Assets

Kenney Particle Pack

- **Asset:** Particle Pack by Kenney
- **Description:** A collection of particle effects including smoke, fire, magic, and other visual effects used for game development or simulations.
- **Source:** `https://www.kenney.nl`
- **License:** Public Domain (CC0 1.0 Universal)
- **Usage in Thesis:** Utilized for visual effects in game environment.

Stylized Nature Mega Kit by Quaternius

- **Asset:** Stylized Nature Mega Kit
- **Description:** A large pack of stylized nature assets including trees, rocks, grass, flowers, and other environmental models for use in 3D scenes or games.
- **Source:** `https://quaternius.com`
- **License:** CC0 (Public Domain)
- **Usage in Thesis:** Used for creating natural 3D environments in project.

AllSky Free – Godot Edition (GitHub)

- **Asset:** AllSky Free – Godot Edition (GitHub)
- **Description:** A collection of 10 high-quality skybox textures and resources designed specifically for Godot Engine, featuring pre-configured scenes and PanoramaSky setups.
- **Source:** `https://github.com/rpgwhitelock/AllSkyFree_Godot`
- **License:** MIT License (© 2020 rpgwhitelock)
- **Usage in Thesis:** Used as skybox backgrounds in the project.

World Normal Mix Shader by Arnklit

- **Asset:** World Normal Mix Shader by Arnklit
- **Description:** A shader designed for blending world-space normal maps in Godot, providing an easy way to mix and modify normal maps in 3D environments.
- **Source:** `https://github.com/Arnklit/TutorialResources/`
- **License:** CC0 1.0 Universal (Public Domain)
- **Usage in Thesis:** Used to implement normal map blending for environment shaders.

Stylized spruce bark

- **Asset:** Stylized spruce bark by Tarox
- **Description:** Texture material
- **Source:** `https://www.materialmaker.org/material?id=472`
- **License:** CC0
- **Usage in Thesis:** Used to texture wood materials.

Stylized rooftile

- **Asset:** Stylized rooftile by Tarox
- **Description:** Texture material
- **Source:** `https://www.materialmaker.org/material?id=116`
- **License:** CC0
- **Usage in Thesis:** Used to texture roof materials.

Stylized Stone Rocks

- **Asset:** Stylized Stone Rocks by Swiperino
- **Description:** Texture material
- **Source:** `https://www.materialmaker.org/material?id=643`
- **License:** CC0
- **Usage in Thesis:** Used to texture stone materials.

Stylized Rock Wall

- **Asset:** Stylized Rock Wall by BurritoLord69
- **Description:** Texture material
- **Source:** `https://www.materialmaker.org/material?id=356`
- **License:** CC-BY
- **Usage in Thesis:** Used to texture wall materials.

Mixamo Animations

- **Asset:** Mixamo Animations
- **License:** Free for non-commercial and commercial use
- **Description:** A collection of high-quality humanoid animations available for free, which can be applied to custom character rigs for game development and other 3D projects.
- **Source:** Mixamo
- **Usage in Thesis:** Used in the project to animate humanoid meshes.

Pixabay Sound Effect: Swing Whoosh In Room 5

- **Asset:** Swing Whoosh In Room 5
- **Author:** floraphonic
- **License:** Free for non-commercial and commercial use (Pixabay License)
- **Description:** Audio effect.
- **Source:** Pixabay
- **Usage in Thesis:** Audio effect.

Swing Whoosh In Room 8

- **Asset:** Swing Whoosh In Room 8
- **Author:** floraphonic
- **License:** Free for non-commercial and commercial use (Pixabay License)
- **Description:** Audio effect.
- **Source:** Pixabay
- **Usage in Thesis:** Audio effect.

Near Miss Swing Whoosh 18

- **Asset:** Near Miss Swing Whoosh 18
- **Author:** floraphonic
- **License:** Free for non-commercial and commercial use (Pixabay License)
- **Description:** Audio effect..
- **Source:** Pixabay
- **Usage in Thesis:** Audio effect.

Near Miss Swing Whoosh 17

- **Asset:** Near Miss Swing Whoosh 17
- **Author:** floraphonic
- **License:** Free for non-commercial and commercial use (Pixabay License)
- **Description:** Audio effect.
- **Source:** Pixabay
- **Usage in Thesis:** Audio effect.

Fireball Whoosh 1

- **Asset:** Fireball Whoosh 1
- **Author:** floraphonic
- **License:** Free for non-commercial and commercial use (Pixabay License)
- **Description:** Audio effect.
- **Source:** Pixabay
- **Usage in Thesis:** Audio effect.

Multi Coin Payout 14

- **Asset:** Multi Coin Payout 14
- **Author:** floraphonic
- **License:** Free for non-commercial and commercial use (Pixabay License)
- **Description:** Audio effect.
- **Source:** Pixabay
- **Usage in Thesis:** Audio effect.

Appendix C

Used Tools

Godot Engine

- **Tool:** Godot Engine
- **Description:** An open-source game engine used for developing both 2D and 3D games. It provides a user-friendly environment for game development, offering various built-in tools and scripting languages such as GDScript.
- **Source:** Godot Engine Official Website
- **License:** MIT License
- **Usage in Thesis:** Used in the project for developing and deploying the game.

Blender

- **Tool:** Blender
- **Description:** An open-source 3D modeling and animation software used for creating, sculpting, rigging, and rendering 3D models and environments.
- **Source:** Blender Official Website
- **License:** GPL-2.0 License
- **Usage in Thesis:** Used in the project to create 3D models and assets, as well as for animation and rigging tasks.

Krita

- **Tool:** Krita
- **Description:** A free, open-source digital painting software designed for concept artists, illustrators, and texture artists. Krita provides a wide range of brushes, tools, and features suitable for creating detailed textures, illustrations, and 2D art.

- **Source:** Krita Official Website
- **License:** GNU General Public License v3.0
- **Usage in Thesis:** Used in the project to create custom VFX textures and 2D artwork for game assets and user interface.

Material Maker

- **Tool:** Material Maker
- **Description:** An open-source procedural texture generation software that works with the Godot Engine, allowing the creation of complex, customizable materials and textures without the need for traditional image editing software.
- **Source:** Material Maker Official Website
- **License:** MIT License
- **Usage in Thesis:** Used in the project to create procedural textures and materials for use in 3D environments and game assets.

Draw.io (diagrams.net)

- **Tool:** Draw.io (diagrams.net)
- **Description:** A free, web-based diagramming tool used for creating flowcharts, network diagrams, UML diagrams, and other types of visual content. It is often used for planning and structuring project workflows.
- **Source:** Draw.io Website
- **License:** Apache License 2.0
- **Usage in Thesis:** Used to create diagrams and flowcharts for visualizing the architecture and workflows within the project.

Appendix D

Used Addons

SimpleGrassTextured by IcterusGames

- **Asset:** SimpleGrassTextured by IcterusGames
- **Description:** A simple and efficient grass texture asset designed for 3D scenes, with adjustable parameters for different environmental effects.
- **Source:** `https://github.com/IcterusGames/SimpleGrassTextured`
- **License:** MIT License (Open Source)
- **Usage in Thesis:** Used in the Godot-based project for creating multimesh in the environment.

Scatter by HungryProton

- **Asset:** Scatter by HungryProton
- **Description:** This is an add-on for Godot 4, which automates the positioning of assets in a scene.
- **Source:** `https://github.com/HungryProton/scatter`
- **License:** MIT License (© 2020-present HungryProton)
- **Usage in Thesis:** Used in the Godot-based project for procedural placement of paths.

Godot Scene Object Brush by Sacristan

- **Asset:** Godot Scene Object Brush by Sacristan
- **Description:** A tool for Godot Engine that allows users to easily place and manipulate scene objects within the editor, providing a more efficient workflow for level design.
- **Source:** `https://github.com/Sacristan/godot_scene_object_brush`

- **License:** The Unlicense (Public Domain Dedication)
- **Usage in Thesis:** Used in the Godot-based project for streamlining scene creation and object placement within the editor.

Blender Terrain Sculpt by BlackEars

- **Asset:** Blender Terrain Sculpt by BlackEars
- **Description:** A tool for sculpting in Blender with a focus on terrain. This tool suite provides options for leveling ground, creating ramps and constraining changes to be perpendicular to the ground.
- **Source:** `https://github.com/blackears/blenderTerrainSculpt`
- **License:** GPL License
- **Usage in Thesis:** Used in the project to create and manipulate terrains within Blender, before exporting them for into the Godot Engine.

Godot Game Tools by Vini Guerrero

- **Asset:** Godot Game Tools by Vini Guerrero
- **Description:** A blender add-on that contains tools to ease the creation/import process of assets into Godot Game Engine and others.
- **Source:** `https://github.com/vini-guerrero/godot_game_tools`
- **License:** GNU General Public License (GPL)
- **Usage in Thesis:** Utilized in the project to apply Mixamo humanoid animations to custom humanoid meshes.