KANTI BADEN

Kantonsschule Baden

www.kanti-baden.ch

Randomness In Videogames / Zufall In Videospielen

Documentation

Rafael Teixeira da Silva, Vineet Nair, Nino Siegenthaler

Matura Project

Primary supervisor: Michael Schneider

Second assessor: Christophe Bersier

Submission date: 11.11.2025

Contents

1 Introduction

Flipping a coin, pulling a slot machine lever, or rolling a die all share one fundamental characteristic: their outcomes are random. Randomness is not only present in our everyday lives but also in the digital world. In recent years, especially within video games, randomness has become a common design tool used by developers to enhance gameplay and, in some cases, to challenge players. Nevertheless, while randomness can enrich the gaming experience, mastering it requires a clear understanding of its underlying principles — the whys and hows of randomness.

To not only study the theoretical aspects but also apply them in practice, our group decided to divide the project into two parts: a literature-based research phase and a practical proof-of-concept (PoC) game. The first part focuses on understanding the fundamental role of randomness in video games through a literature review supported by case studies that demonstrate how theory has been successfully applied in the past. To maintain a structured approach, we formulated several guiding research questions:

- **How is randomness used as a design tool in video games, and for what purpose?**
  This question is crucial, as its answer supports our own game design and helps avoid meaningless uses of randomness that could reduce player satisfaction.
- **How does randomness affect player satisfaction?**
  Understanding this relationship allows us to design randomness that enhances rather than frustrates the player experience.
- **What are the differences between input and output randomness, and when should each be applied?**
  For example, in a coin flip, the randomness lies in the output, whereas input randomness affects player actions before the outcome is determined. We aim to explore how both types influence perception and engagement.
- **How can developers balance randomness to maintain fairness, enjoyment, and strategic depth?**
  This final question addresses our goal of designing a game that is not only enjoyable but also fair and thought-provoking.

Our Paper's Structure

The structure of this report is as follows. It begins with our literature review and case studies on randomness in video games, presenting the research results and conclusions. Next, we explain our product and its design, giving an objective overview of the game itself. After this, we explain the practical parts about how we implemented the idea into PoC; the methodology. Following that, we get into the design and justification of our questionnaire for the playtesting of the PoC game. This is then followed by two parts of our the PoC's playtesting: presting its results guided by our questionnaire and followed by a discussion, where we try to interpret the result, trying to find any flaws and to detirmine our success. The report concludes by summarizing the key points of our project, reflecting on which research questions we successfully answered and applied, and discussing the knowledge gained throughout the process. Finally, we provide an outlook on future developments and unresolved challenges.

2 Literature Review

2.1 Randomness in Videogames

In modern game design, randomness has become an essential tool for shaping dynamic, engaging, and oftentimes unpredictable player experiences. From procedural level generation in roguelike titles to probabilistic loot systems in online multiplayer games, chance-based mechanics are present in nearly every genre. The relevance of randomness in video games lies not only in enhancing variety and replayability, but also in affecting fairness, difficulty balancing since the game should not just be blatantly spamming some random elements out of context, moreover, psychological reward mechanisms are also of high importance in games with randomness. As gaming becomes an increasingly important field of study within computer science and psychology, understanding the role and effects of randomness is both timely and of high importance which also one of the reasons why we chose to focus on this aspect.

This literature review explores the principle of randomness in video games in a broad manner, looking into game design theory, cognitive science, and case studies from both indie and AAA game development. The aim is to analyze how randomness is applied in digital games, what purposes it serves, and how it influences player

behavior, fairness, and strategic decision-making. Additionally special attention is paid to the distinction between *input randomness* where random events shape the scenario before the player's decision and *output randomness*, where the randomness affects the outcome of the player's action.

Our research were initially guided by the following research questions to give the research some sort of structure:

1. How is randomness used as a design tool in video games, and for what purposes?
2. What is the difference between input and output randomness, and how do these affect player perception and engagement?
3. How do developers manage or limit randomness to maintain fairness, enjoyment, and strategic depth?

To address these questions, we reviewed a diverse set of academic sources, industry reports, developer interviews, and case studies. Key examples include games like *Into the Breach*, *Slay the Spire*[1], *Shadow of Mordor (Monolith productions, 2014)*, and *XCOM*, which demonstrate various implementations and consequences of randomness. This review also includes a closer look at psychological phenomena (e.g. reward cycles, illusion of control, etc.), regulatory discussions surrounding loot boxes, and strategies for managing randomness to optimize game balance. **The ultimate goal of this study was to apply the gathered insights to the development of a proof-of-concept video game that incorporates these findings into its core design.**

2.2 Terminology

To avoid confusion throughout this literature review, we have compiled and explained the key technical terms and concepts that are frequently used when discussing randomness in video games. This section shows relevant terminology that come in context of randomness in video games. Most of the terms are present in this literature review.

Randomness: The occurrence of unpredictable outcomes in a system. In games, randomness is often introduced through random number generators to produce uncertainty in gameplay events, such as item drops or enemy behavior.

[1]Wikipedia: Slay the Spire, last accessed on: 05.11.2025.

Input Randomness: A form of randomness where the random event occurs *before* the player makes a decision. Examples include drawing a hand of cards or procedurally generating a level before the player interacts with it. Input randomness shapes the conditions under which a player must act.

Output Randomness: A form of randomness that takes place *after* the player has made a decision. It determines whether the intended action succeeds or fails. For example, shooting at an enemy in *XCOM* may or may not hit, even if the player chose the target carefully.

Procedural Generation: The use of algorithms to automatically create content (such as maps, levels, or characters) instead of manually designing each element. This method is often used to ensure replayability and variety.

Information Horizon: A term from game design that refers to how far into the future a player can predict the outcome of their actions based on available information. Randomness reduces the information horizon by making future events less predictable.

Loot Box: A virtual item that players can open to receive random rewards, often used in monetization systems. Loot boxes may contain cosmetic items or powerful equipment, and are sometimes compared to gambling mechanisms.

Dungeon Crawl: Dungeon crawl is a type of gameplay or genre where the player explores a labyrinthine-like environment, facing: enemies, loot and traps

Output Bias and Probability Manipulation: A practice in game design where probabilities are subtly altered to match player expectations. For example, a game may make a 90% hit chance function more like 99% to avoid perceived unfairness.

Spiky Information Flow: A design approach where most of the game progresses with manageable complexity, but sudden, disruptive events introduce large amounts of new information at once. This keeps gameplay both strategic and dramatic.

Variable-Ratio Reward Schedule: A reinforcement system where rewards are given after an unpredictable number of actions. This type of reward structure, common in both slot machines and loot boxes, tends to be highly addictive.

Skill Tree Systems: This is a way the game allows players to visually choose the characters features. Often present in role-playing games such as the AAA game *GTA V (Rockstar Games, 2013)*.

2.2.1 What are roguelikes?

One of the key terms used in this study is the genre classification roguelike. The name stems from the orginal game, *Rogue*, developed by Michael Toy and Glenn Whichmann. If features of *Rogue* are present in other games you call it roguelike becuase it is **like** the game **Rogue**. Now, what are features of *Rogue*, or what are roguelikes? A roguelike game is traditionally defined as one that incorporates procedural generation, dungeon crawl and last but not least a roguelike contains a permanent death system, often referred to as "permadeath." These design principles originated from the 1980 game *Rogue*, from which the genre takes its name. In roguelikes, each game session is unique due to the use of randomized environments and item placements, forcing players to adapt strategically to unfamiliar conditions. Additionally, when a player's character dies, they typically lose all progress and must start from the beginning, emphasizing high stakes and replayability. The randomized structure of room progression and stat-based item variations, in roguelikes, provide a fitting context to explore input randomness in practice.

Over time, the term roguelike has evolved, and many modern games incorporate only some of these features. Such titles are often referred to as roguelites but they still preserve the core concept of randomness and high replay value. Examples of popular modern roguelikes or roguelites include *Slay the Spire, Hades.*[2] *(Supergiant Games, 2020)*, and *Dead Cells (Motion Twin, 2018)* [3] [4] [5].

2.2.2 Procedural Generation in Rogueikes

Procedural generation is a method in game design where content, such as levels, maps, items, or events, is created algorithmically rather than manually. Instead of a developer hand-crafting each individual element, a set of rules or algorithms generates new combinations of content during gameplay or each time the game is loaded. This technique allows for vast amounts of variability, increased replayability, and reduced production time for content-heavy games. Classic examples of procedural generation include the terrain in *Minecraft (Mojang Studios, 2011)*, the dungeon layouts in *The Binding of Isaac (2011)*, or the randomized weapons creation in *Borderlands (Gearbox Software, 2009)*.

[2]Wikipedia: Hades, last accessed on: 05.11.2025.
[3]Wikipedia: Slay the Spire, last accessed on: 05.11.2025.
[4]Wikipedia: Hades, last accessed on: 05.11.2025.
[5]Wikipedia: Dead Cells, last accessed on: 05.11.2025.

Roguelike games frequently use procedural generation as a core mechanic to achieve their defining characteristic: unpredictability. Since each playthrough in a roguelike is meant to feel different, procedural generation offers a practical solution for generating fresh challenges without requiring developers to design hundreds of levels manually. In this sense, the two concepts: roguelike and procedural generation—are often tightly linked, especially in popular discourse and game marketing.

However, it is important to highlight that while procedural generation is common in roguelike games, it is not a requirement. A roguelike can still function effectively without full procedural generation, as long as it maintains core genre features such as permadeath, random elements (input randomness), and high replayability. The replacement of traditional procedural generation can be achieved by using pre-designed content segments that are shuffled or rearranged differently in each run. For example, a set of manually created rooms can be randomly ordered to create a new game structure each time, mimicking the effects of procedural randomness while preserving design control. Such form of procedural generation is reffered to as modular procedural generation because it uses premade modules and shuffles them randomly into a dungeon-crawl map. [6]

2.3 Purpose of Randomness in Games

Randomness is commonly used to introduce variety, replayability, and unpredictability into games. Procedural generation, for example, allows games like *Minecraft* and roguelikes to produce infinitely many levels and challenges, ensuring that each playthrough feels fresh. *Shadow of Mordor*'s "Nemesis system" adapts based on player actions, using randomness to dynamically alter storylines and characters.

In multiplayer games, randomness can be a tool for fairness. Games such as *Mario Kart (Nintendo, 1992)* use random item distribution to give weaker players stronger boosts, balancing out differences in skill. This ensures that newer or less experienced players can still enjoy the game, making it accessible and fun for all participants.

[6] Fort: Controlling Randomness, last accessed on: 03.04.2025.

2.3.1 Types of Randomness: Input vs. Output

A key distinction in game design is between input and output randomness. [7] Input randomness refers to situations where a random event happens before the player makes a decision. This includes shuffled cards in *Slay the Spire*, rolled dice in *Dicey Dungeons (Distractionware, 2019)*, or procedurally generated maps. Players adapt their strategies to the conditions set by these random factors.

Output randomness, on the other hand, occurs after a player has made a decision. For instance, in *XCOM*, a player chooses to shoot an alien, but whether the shot hits is determined by chance a.k.a. randomness. Similarly, in *Apex Legends (Respawn Entertainment, 2019)*, the contents of a loot box are only revealed after the purchase. Output randomness is often criticized because it introduces uncertainty after a player has made a decision, meaning that even well-thought-out strategies can fail due to chance alone. This can lead to a loss of perceived control, where players feel that their success or failure is not determined by their skill or planning but by luck. As a result, repeated negative outcomes, especially in high-stakes moments can generate frustration, reduce motivation, and ultimately worsen the overall gameplay experience.

The Ludology Podcast (Ep. 183) and online interviews with other game developers suggest that while output randomness can create tension, it can also cause players to feel a lack of control.[8] Games like *FTL (Subset Games, 2012)* were noted for generating player frustration due to excessive output randomness, leading the developers to shift toward more input-based randomness in subsequent titles like *Into the Breach*.

2.3.2 Strategic Impact and the Information Horizon

Game designer Keith Burgun introduces the concept of the "information horizon" which is the gap between a player's current turn and when new information is revealed. Randomness affects how far into the future[9] a player can plan. If randomness occurs too frequently or without limits, it shrinks the information horizon, making planning difficult.

Well-balanced randomness, especially input randomness with good information horizon, can significantly enhance strategic depth by forcing players to adapt their

[7] Game Maker's Toolkit. 2020. "The Two Types of Random in Game Design." https://www.youtube.com/watch?v=dwI5b-wRLic
[8] Ludology Podcast: Input Output Randomness, last accessed on: 10.04.2025.
[9] Game Maker's Toolkit: Two Types of Random, last accessed on: 24.03.2025.

plans to new and unpredictable conditions. The sudden random input randomness may be reffered to as spiky information flow. This type of randomness encourages thoughtful decision-making and makes players feel that their success or failure is a direct result of their own choices. Because the randomness occurs before the player takes action, it frames the challenge rather than interfering with the outcome. Players are given time and space to analyze the situation and select a response based on the available information. As a result, failure tends to be interpreted as a consequence of one's own strategy rather than bad luck, which strengthens the sense of fairness and personal responsibility. The game thereby supports improvisational gameplay and creative problem-solving, while avoiding the feeling of making progress feel arbitrary and not well programmed.

Some games mitigate chaotic randomness through design tricks. *Pandemic (Z-Man Games, 2008* controls card randomness by shuffling epidemic cards into separate piles to avoid too many or too few appearing in a row thus optimizing the information horizon[10] . Modern *Tetris (1985)* uses a 'bag system' to control randomness in the distribution of pieces. Instead of selecting each new piece completely at random, the game first creates a 'bag' containing one of each of the seven unique Tetris block shapes. These pieces are then randomly shuffled and drawn one by one until the bag is empty. Once all seven blocks have been used, a new bag is filled and the process repeats. This method ensures that players receive every block type within a short span of time, eliminating frustrating situations where, for example, the crucial line piece (the 'I' block) fails to appear for many turns. The bag system preserves randomness while providing a fair and predictable distribution, allowing players to plan better and reducing the chances of losing due to bad luck alone[11] .

2.3.3 Psychological and Behavioral Effects

Random reward systems often tap into psychological effects similar to gambling. Loot boxes, for example, use variable-ratio reward schedules (like slot machines) to stimulate dopamine release, reinforcing the desire to keep playing[12] or spending.

Players often misinterpret probabilities of such slot machines. A 90% hit chance might feel like a certainty, and a string of failed 33% outcomes may cause players to believe the next one "has to" succeed. Developers sometimes adjust probabilities to match player expectations. For example, *Fire Emblem (Nintendo 1990)* improves

[10]Z-Man Games: Pandemic – Rules, last accessed on: 05.11.2025.
[11]TetrisWiki: Random Generator (7-bag), last accessed on: 05.11.2025.
[12]Deterding u. a.: Mastering Uncertainty, last accessed on: 20.05.2025.

hit chances behind the scenes[13] , and *Civilization VI (Firaxis, 2016)* may guarantee a success after repeated failures.

Tharsis (Choice provisions 2016) developer Zach Gage and others suggest using intuitive systems like physical dice or cards to help players better understand randomness.[14] Games like *Slay the Spire* benefit from this clarity, as players can predict and adapt to outcomes more easily based on known card effects. Similarly, *Armello (League of Geeks, 2015)* uses card-based actions and visible dice rolls in combat, which allow players to assess risks and make informed decisions[15] . The game also includes stat-based effects, such as increased Fight or Body stats, earned through quests and gear, which influence how randomness affects outcomes. Because these systems are transparent and familiar and morevover an input-randomness, players feel more in control even when luck plays a role.

By presenting randomness in ways that are easy to visualize and calculate, such games reduce the perception of unfairness and support engagement even in uncertain situations. It is to be considered that such addition may eliminate some surprise factor which randomness-based game try to achieve since you tell the player their chances and it will not come to them as a surprise.

2.3.4 Ethical and Design Considerations: Loot Boxes

Surpisingly, loot boxes raise ethical questions due to their similarity to gambling. The ToDiGRA journal article explores this topic in depth, showing how loot boxes operate under the same psychological principles as gambling machines, including near misses, surprise rewards, and dopamine triggers.[16]

Legal frameworks vary: Belgium and the Netherlands classify some loot boxes as gambling and have banned them, while other countries have taken softer stances.[17] Some companies attempt to increase transparency by disclosing item drop odds, which means they publicly state the exact probability of receiving different item types when opening loot boxes. For example, a game might indicate that a legendary item has a 5% chance of appearing, while common items have a 75% chance. This helps players make more informed decisions.[18] and reduces the sense of manipulation.

[13]Serenes Forest: True Hit, last accessed on: 05.11.2025.
[14]Game Developer Staff: Randomness Transparent Tharsis, abgerufen am: 31.03.2025.
[15]Game Developer Staff: Randomness Transparent Tharsis, last accessed on: 31.03.2025.
[16]Deterding u. a.: Mastering Uncertainty, last accessed on: 20.05.2025.
[17]Nielsen; Grabarczyk: Loot Boxes Gambling, last accessed on: 01.05.2025.
[18]Nielsen; Grabarczyk: Loot Boxes Gambling, last accessed on: 01.05.2025.

Academic studies also suggest that younger players are especially vulnerable, as they lack the cognitive control to resist these psychological tricks. Financial harm, lack of transparency, and addictive mechanics make loot boxes one of the most controversial uses of randomness in games today.

In addition, some games implement what's called a pity system—a mechanic designed to guarantee a rare or valuable reward after a certain number of unsuccessful attempts. For instance, *Hearthstone (Blizzard Entertainment, 2014)* uses a pity timer that ensures players receive at least one legendary card if they have opened 40 packs without getting one[19]. This system prevents extremely unlucky streaks and reduces player frustration, while still maintaining the element of randomness.

These mechanisms help reduce the resemblance to gambling by minimizing the sense of total unpredictability. In traditional gambling, outcomes are fully random and players have no assurance that persistence will lead to success. By contrast, pity systems provide a form of safety net, ensuring that effort (or spending) is eventually rewarded. Disclosing drop odds also discourages the illusion of 'beating the system,' which is common in gambling psychology. Together, these features make loot systems feel more controlled, transparent, and less exploitative—although debates remain about whether they go far enough to fully separate games from gambling-like mechanics.

2.3.5 Random Stat-Based Upgrades

One specific and increasingly popular design approach is the use of random stat-based upgrades. A random stat-based upgrade is a game mechanic in which a player's character receives improvements to core abilities—such as health, strength, defense, or special attributes through randomized systems. These upgrades differ from traditional level-up or skill tree systems in that the enhancements are not chosen directly by the player, but rather are assigned or acquired through probabilistic means such as item drops, card draws, or quest outcomes. Unlike cosmetic or inventory-based randomness, stat-based upgrades directly impact gameplay mechanics, often shaping how players approach challenges in subsequent stages of the game.

This mechanic plays an important role in improving input randomness by introducing variety early in the decision-making process. Players are often required to adapt their strategies to the upgrades they receive, while still feeling in control because they are responding to known changes rather than suffering from unpredictable

[19]Hearthstone Wiki: Pity timer, last accessed on: 05.11.2025.

outcomes. In this way, random stat-based upgrades maintain player engagement by providing new opportunities in each playthrough.

A prominent example is found in *Armello*, where players can increase their character's stats—such as Fight, Body, Wits, and Spirit through card-based equipment and successful quest completions. The quests present players with randomized risks and potential rewards, while gear cards are drawn from shuffled decks. Although the specific rewards are not guaranteed, the possible outcomes are visible, allowing players to make informed decisions. This clarity enhances the fairness of the system and preserves agency, as players feel they are strategically navigating through uncertainty rather than being passively affected by it[20] .

Random stat-based upgrades are particularly well-suited to roguelike games, which emphasize replayability and emergent strategy. By introducing new stat combinations in each run, these systems encourage players to explore different playstyles and make dynamic choices that align with the resources available. At the same time, they avoid the potential frustration of output randomness by ensuring that player choices remain central to success.

Table 1: Comparison of Randomness' in Game Design

Type of Randomness	**When It Occurs**	**Impact on Player**	**Example Games**
Input Randomness	Before the player makes a decision	Shapes initial conditions; encourages adaptation and planning; perceived as fair	*Slay the Spire*, *Spelunky*, *Into the Breach*
Output Randomness	After the player makes a decision	Affects outcomes directly; can reduce perceived control; adds tension or frustration	*XCOM*, *Hearthstone*, *Apex Legends*
Stat-Based Randomness	During upgrade or reward phases	Influences future strategy; offers variety without compromising agency	*Armello*, *Dead Cells*, *Hades*
Reward System Randomness	Upon opening loot/reward boxes	Triggers dopamine response; often criticized as manipulative; ethically controversial	*Overwatch*, *FIFA Ultimate Team*, *CS:GO*

[20] Armello Wiki contributors: Armello, last accessed on: 04.10.2025.

3 Hypothesis Formulated from Literature Review

The research findings indicate that randomness enhances engagement and variety most effectively when it preserves player agency and control. Consequently, the proposed prototype will feature **no output randomness**. Combat mechanics will be deterministic: if a player aims and executes an attack, the outcome is guaranteed. Instead, uncertainty is introduced before decision-making through randomized room sequences, upgrade options, and challenge variations.

A particularly promising design direction involves **random stat-based upgrades**, which provide players with varied statistical enhancements or abilities. This mechanic offers unpredictable yet fair variety across playthroughs, ensuring that each run presents a unique strategic experience.

The concept of the **Information Horizon**—the strategic timing of surprise elements—provides a valuable framework for implementation. This principle informs the design of features such as enemy ambushes and environmental effects, which disrupt established patterns in meaningful ways rather than introducing randomness arbitrarily, thereby requiring adaptive player responses.

The prototype will intentionally **exclude loot boxes based on monetized randomness**. The literature demonstrates significant ethical concerns regarding player manipulation in such systems, particularly when rewards are tied to financial transactions. Excluding these mechanics supports the goal of creating an engaging experience that respects player autonomy.

Additionally, the design incorporates a **pity system** to mitigate potential frustration from consecutive unfavorable outcomes. When players receive weaker randomly generated rewards multiple times consecutively, the system increases the probability of receiving superior rewards. This mechanism ensures that randomness does not become the primary determinant of repeated failure.

A practical constraint identified during development planning is the complexity of full procedural generation. Given time and resource limitations, the approach utilizes handcrafted rooms that are randomly assembled into different configurations for each playthrough. This method maintains unpredictability and input randomness—core principles of roguelike design—while ensuring that each room is carefully balanced for optimal gameplay. This approach is referred to as **modular procedural generation**.

3.1 Quick Summary

The prototype aims to demonstrate how randomness can be implemented strategically to create challenging and varied gameplay while preserving player agency and control. The proposed game concept synthesizes the research findings into a cohesive design: a roguelike featuring deterministic combat mechanics, random stat-based progression systems, vertical navigation elements, and dynamically assembled environments that vary with each playthrough.

4 The Game

4.1 Core gameplay loop

Our game is a Roguelike and therefore contains a lot of random elements. This includes random levels, random item placement, randomized drops and randomly generated upgrades. The core gameplay loop consists of starting a run, which means generating a new and unique level, as well as resetting any progress the player may have made in a previous run. Playing the randomly generated level by defeating enemies and collecting upgrades until the player either dies and a new run is started or until the player has fully explored, at which point they can choose to advance to the next level. This generates a new level layout but unlike starting a new run, the player keeps all collected upgrades and the enemies grow stronger, getting more health and increased damage. This loop is repeated as long as the player can keep up with the ever increasing strength of the enemies or as long as the game remains functional and interesting, should the player become powerful enough to keep playing indefinitely.

4.2 Core Mechanics

4.2.1 Player Abilities

Our game is a first person shooter and thus, the player has most of the basic abilities one would expect from such a game. Specifically the player is able to move around using the WASD keys and look around using the mouse, as well as able to jump by pressing the spacebar.

Further, the player is able to use two types of basic attacks: A ranged attack and a melee attack. The ranged attack shoots a bullet in the direction the player is facing and will deal damage to a single enemy it hits. The melee attack will damage any enemies in a certain range in front of the player, the melee attack deals more damage at the cost of being at risk of taking damage because of the proximity to the enemies.

A feature more unique to our game is the ability for the player to change their own gravity at will, by pressing either the arrow keys or left shift and WASD at the same time. The player can do four different types of rotations: The up arrow (W) changes gravity 90° in the direction the player is facing, the left arrow (A) rotates gravity 90° to the right of the direction the player is facing, the right arrow (D) does the same as the left arrow but to the right, the down arrow (S) flips gravity 180° while keeping the player facing the same direction.

Because of level design reasons, the player is not able to change gravity to be any direction, but can only change between six different directions. Whenever the player decides to rotate, the game sets whichever of these directions is closest to the direction gravity would be if it could point in any direction.

By design, the player is unable to aim, and therefore unable to attack enemies, while in the process of rotating. This is due to the fact that the enemies cannot attack the player while mid rotation because the player is moving too fast, hence, if the player were able to defeat enemies while changing gravity, the player could simple perpetually rotate to defeat all enemies without being at risk of being defeated themselves. For the same reasons, the gravity change is designed to be slightly disorienting for a moment.

Because the game is also a Roguelike, the player is able to pick up upgrades. There are four different types of upgrades, one of them being the so-called active items. These are upgrades that give the player a new ability when pressing the active item key (E), and usually have a cooldown. Additionally, the player can only have one active item at a time, but the current item will be dropped when a new one is picked up, and thus is not lost forever.

4.2.2 Progression and Upgrades

Progression is delivered through four upgrade channels, designed to balance predictability with randomness and again to create variety across runs. First, the player

can collect direct, deterministic upgrades to core attributes: health (increasing survivability against burst and sustained damage), armor (percentage damage reduction, mitigating incoming damage multiplicatively), melee damage and melee attack speed (raising close-quarters throughput and responsiveness), ranged damage and ranged attack speed (improving ranged time-to-kill and projectile cadence), movement speed (enhancing positioning, strafing, and gravity-shift execution windows), and mobility enhancements such as additional jumps and jump height (expanding traversal options, especially in vertically oriented rooms).

Second, special upgrades are individually coded, bespoke effects that can alter moment-to-moment gameplay or add entirely new capabilities. Examples include making attacks explode on impact or unlocking a new movement or combat ability. The purpose of this category is to introduce qualitative shifts that make runs feel, sometimes complementing the gravity element, meaning the upgrade works well together with the gravity-element.

Third, active items are player-triggered effects on a cooldown. They can do anything similar to special upgrades but are situational by design, encouraging tactical timing. Because they are not always-on, active itmes create clutch decision points, for instance, using an active to breach a turret-guarded corridor, to stabilize after a poisoned hit, or to reposition during a gravity inversion.

Finally, the system offers a random stat upgrade via a scrap option. Upon picking up an upgrade, the player can elect to scrap it in exchange for a randomly generated stat upgrade. The system allocates a total number of stat increases based on the original upgrade's rarity and distributes this budget randomly across eligible stats. This mechanic uses the theme of randomness as a risk-reward choice: accept a known benefit, or gamble for a potentially higher-variance outcome that may better fit the current option.

4.3 Procedural Level Generation in Three Dimensions

The level is assembled at runtime by selecting from a preset catalog of modular rooms. Each room is authored with one or more exit configurations that can connect along the six cardinal directions in 3D space (north, south, east, west, up, down). Generation proceeds by composing these rooms into a connected structure, ensuring that exit types and positions are compatible where rooms meet. Conceptually, the result can be viewed as a 3D graph in which rooms are nodes, exits are edges, and

the algorithm constructs the map by placing rooms so that their exits align and form traversable links.

Randomness enters this pipeline in the selection and placement of rooms and in the choice among alternative exit configurations for any given room. Because the map operates in three dimensions, the generator is not constrained to a singular floor layouts; vertical adjacency (up and down) is therefore an added element, enabling spatial variety. The design favors variety and thus increasing the uniqueness of each run: although the layout is random, the modular rooms as a base are not, thus in our case this would be called modular procedural generation.

4.4 Enemy types and their behaviors

The enemy roster comprises three principal categories, each designed to read clearly and to create different pressures on the player. Basic enemies form the frontline threat. They primarily deal melee damage and close distance to the player. A variant can switch to a ranged attack if the player is unreachable (for example, due to gravity orientation, elevation differences, or blocked paths). Both melee and ranged versions have poisoned variants that inflict a damage-over-time effect, increasing attrition pressure and forcing resource-conscious play.

Turrets are stationary ranged threats that continuously fire at the player. Their role is area denial and spacing control, especially potent in rooms with long sightlines or limited cover. In addition to poisoned turrets, an elite turret variant is significantly stronger, raising the tactical stakes of how and when to approach or bypass fortified positions. Summoner enemies are stationary units capable, in principle, of calling in any enemy type; however, to prevent overwhelming difficulty spikes, they predominantly spawn basic enemies in practice. This preserves the intended combat rhythm, creating waves that test crowd control and target prioritization, without conflating encounter difficulty with unmanageable unit diversity.

Together, these categories create complementary pressures: basics test kiting and positioning, turrets force line-of-sight management and cover usage, and summoners add a soft timer that rewards aggressive disruption.

4.5 Interactions, Balance and Pacing

The systems are were not just chosen arbitrarily but were designed to work together and interlock. What that means is that gravity manipulation and 3D layouts shape how enemies exert pressure; enemy compositions, in turn, stress different aspects of melee and ranged combat; upgrades mediate the player's response and long-term strategy. Poison variants and elite turrets raise difficulty and burst threats, making armor and health valuable, while movement and jump-related stats aid taking better routes that can bypass or flank static defenses (e.g. due to enemies). Special upgrades can amplify these relationships, for example, explosive attacks improving crowd control against summoner waves and active itmes provide tactical spikes that let the player take riskier and more fun decisions.

5 Implementation

5.1 Project Organisation

At the beginning of this project we had divided the work load clearly into three parts. The programming part; divided into two main parts, consisting of programming which was done by Nino Siegenthaler with support of Rafael Texeira da Silva and the Game Design. The latter was done mainly by Rafael with support from Nino. The remaining non-programming part was done by Vineet Nair. This included the literature review, writing the questionnaire for the playtesting and organinsing the playtestig, what we mean by that will be explained in the later segments.

5.2 Literature Review

To investigate the role of randomness in video games, we used a variety of sources available to us, using a combination of general-purpose and academic sources. Our research began with explanatory content from YouTube and Wikipedia, which provided an accessible introduction to key terms and concepts. This initial phase helped us build a terminology list that served as a foundation for identifying academic keywords (see chapter 2).

The majority of academic literature was sourced from the open-access database *arXiv*, which offered a wide range of research papers in computer science and game

studies. Search terms such as “input randomness,” “procedural generation,” and other were derived directly from our terminology list and entered into the arXiv search tool to locate relevant sources. Additional searches were conducted using Google Scholar and ResearchGate. Although ResearchGate provided a promising insightful papers, we were unable to access them due to paywalls or permission restrictions, and therefore focused primarily on sources available via arXiv and other smaller but trustworthy databases.

Throughout the process, we documented all useful findings in a shared Word document to maintain a clear overview and manage the large volume of information. As recommended by our supervisor, Michael Schneider, we included only research papers published by reputable or verifiably academic institutions. This increased the credibility and reliability of our source base and ensured that the reviewed material met academic standards.

While the inability to access certain ResearchGate papers represents a limitation of our study, the broad selection of high-quality sources from arXiv still allowed us to gather solid information on the topic.

5.3 Pre-Input for Modelling: Blender

This section distills practical Blender techniques we adopted during production. It complements the previous chapters with concise, actionable steps you can follow when modeling for real-time engines like Unity. The focus is on repeatable workflows and hotkeys that speed up iteration while keeping meshes clean and export-ready.

5.3.1 Viewport Navigation and Transforms

- Orbit with the middle mouse button, pan with Shift + middle mouse, and zoom with the mouse wheel. When precision is needed, type numbers during a transform to enter exact values.
- Core transforms: G (grab/move), R (rotate), S (scale). Constrain to an axis by pressing X, Y, or Z immediately after the command (e.g., G then Z). Hold Shift while transforming to slow down for fine control.

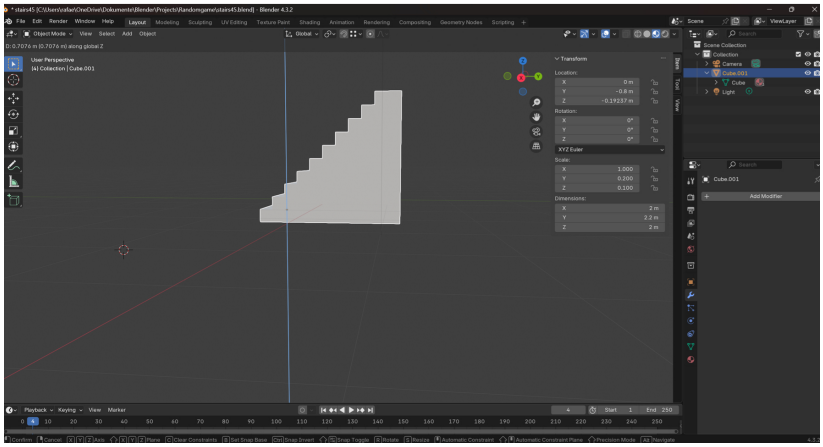


Figure 1: Showing G then Z Grab

- Use the number pad (1/3/7 for front/side/top; Ctrl for opposite) or View menu to quickly re-orient your view. This keeps selections accurate and avoids skewed edits. [21]

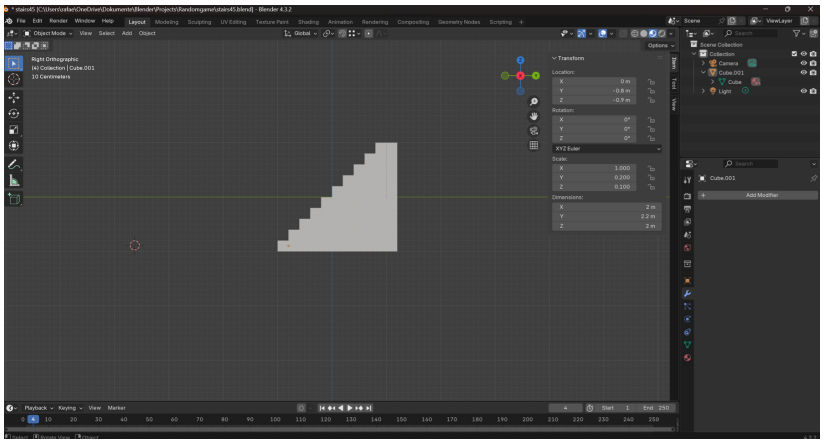


Figure 2: Side View

5.3.2 Edit Mode and Selection Fundamentals

- Toggle Object/Edit Mode with Tab. In Edit Mode, switch selection type with 1/2/3 for vertices/edges/faces.
- Proportional Editing (O) is invaluable for soft, organic changes. Scroll the mouse wheel to adjust the influence radius while moving, rotating, or scaling.
- Link Select (Ctrl + L) expands your selection to all connected elements—useful when a model is composed of multiple shells.

[21] Blender Guru: Donut, last accessed on: 06.11.2025.

Figure 3: Sphere of influence

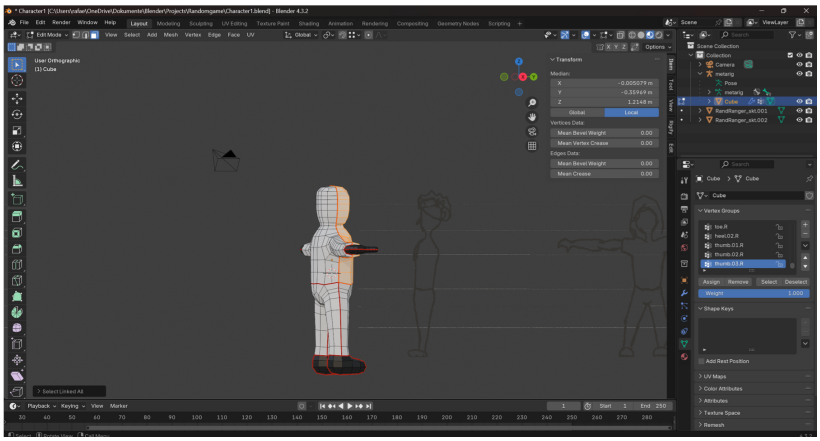


Figure 4: Link select

- Separate (P) splits selected geometry into a new mesh object, helping when you want clean hierarchies or to apply modifiers to only part of a shape. [22]

5.3.3 Fast Modeling Toolkit

- Extrude (E): grows new geometry from selections. Immediately constrain to an axis (E then Z, for example) to avoid diagonal drift.
- Loop Cut (Ctrl + R): adds supporting edges for better control over curvature and deformation. Roll the wheel to add multiple cuts before confirming.
- Bevel (Ctrl + B): replaces razor edges with small chamfers for more realistic shading. Add segments for smoother results but keep budgets in mind.

[22] Blender Guru: Donut, last accessed on: 06.11.2025.

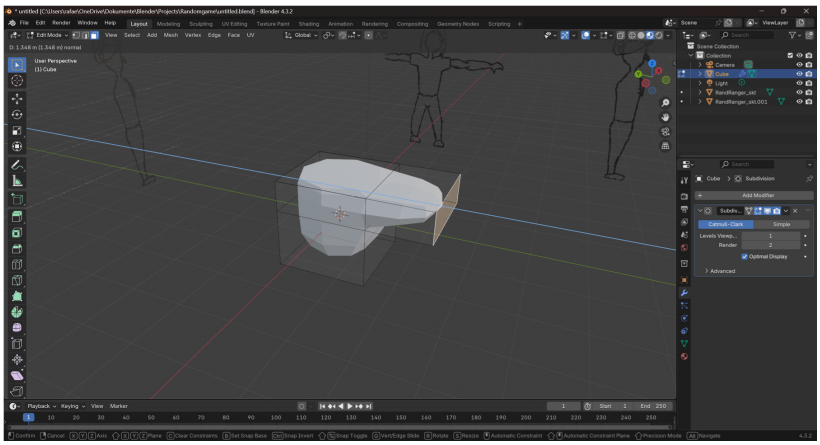


Figure 5: Showing Extrude

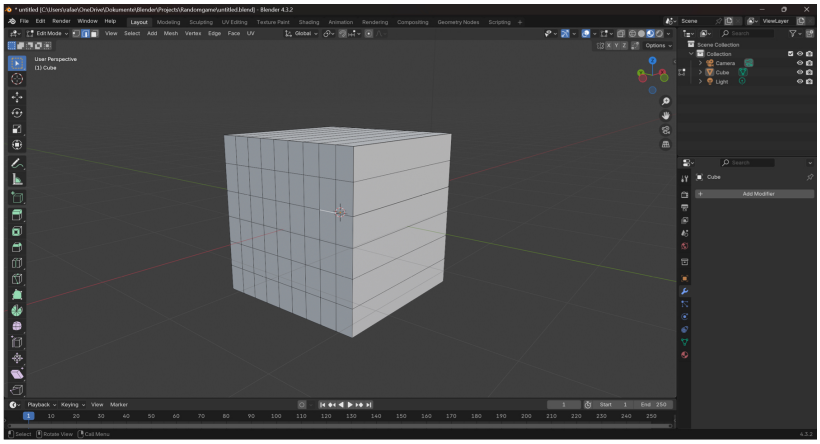


Figure 6: Showing loop cuts

- Knife (K): draw custom cuts to introduce detail exactly where needed. Use snapping to stay orthographic when required.
- Hide/Unhide (H / Alt + H): temporarily removes selections to work without visual clutter. X-Ray view helps when selections must pass through the mesh.[23]

[23]Jelle Vermandere: Character, last accessed on: 06.11.2025.

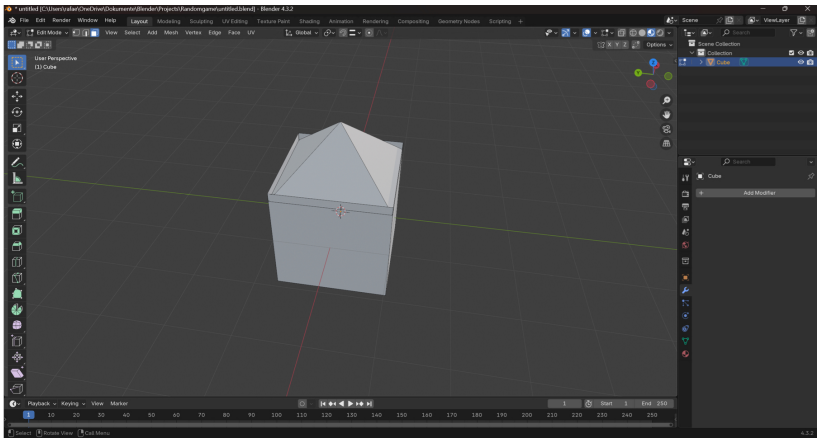


Figure 7: Showing Bevel

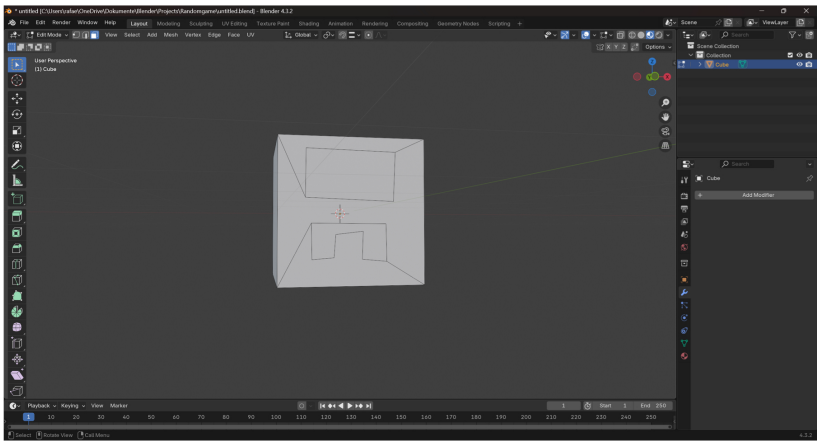


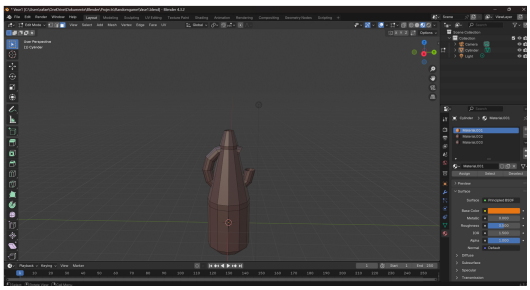
Figure 8: Showing Knife

5.3.4 Productive Use of Modifiers

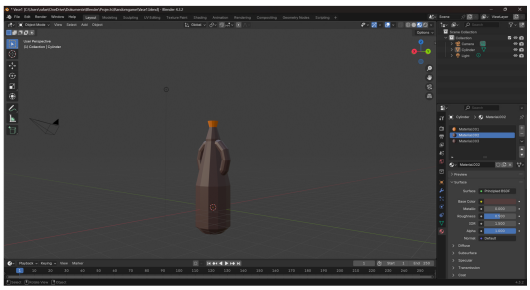
Work non-destructively and commit only when necessary:

- Solidify: adds thickness to shells and panels. Great for ramps, vases, and any surface that needs volume.
- Subdivision Surface: smooths shapes by subdividing and averaging. Combine with supporting loop cuts to preserve form.
- Mirror: model only half (or a quarter) of a symmetrical object; ensure the origin sits on the mirror plane for predictable results.
- Boolean: union, subtract, or intersect volumes to cut holes or blend shapes quickly. Apply and clean up supporting topology afterward.

Keep modifier order intentional: for example, Mirror before Subdivision to keep seam continuity, and Solidify after smoothing to maintain even wall thicknesses.



(a) Hidden Perspective of Hide and Unhide



(b) Unhidden Perspective of Hide and Unhide

Figure 9: Hide and Unhide

5.3.5 Materials and Viewport Display

Assign materials in the Material Properties panel. Use the Principled BSDF shader for most surfaces—tune Base Color, Roughness, and (optionally) Metallic. For quick look-dev, switch to Material Preview and test how your model reads from different angles and light intensities. Name materials clearly so they can be reused across assets.[24] [25]

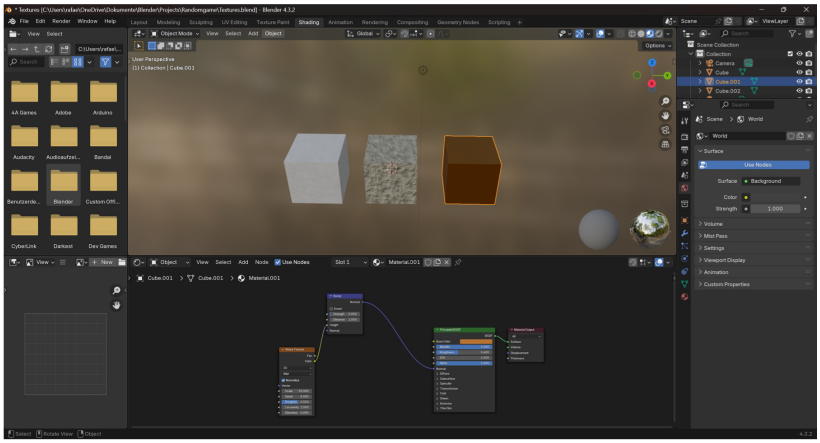


Figure 10: Inexperienced vs. Experienced Mean values

5.3.6 Rigging Primer: Armatures and Weight Painting

When an object must deform (creatures, flexible props), add an Armature and parent the mesh to the skeleton with automatic weights. Enter Pose Mode to test joints early. Then refine in Weight Paint mode: paint more influence (red) on vertices that should follow a bone and reduce influence (blue) where the mesh must remain stable. Keep bone names and hierarchy tidy—clean rigs export to Unity more reliably. [26]

[24] Jelle Vermandere: Character, last accessed on: 06.11.2025.
[25] Blender Guru: Shading, last accessed on: 06.11.2025.
[26] Jelle Vermandere: Character, last accessed on: 06.11.2025.

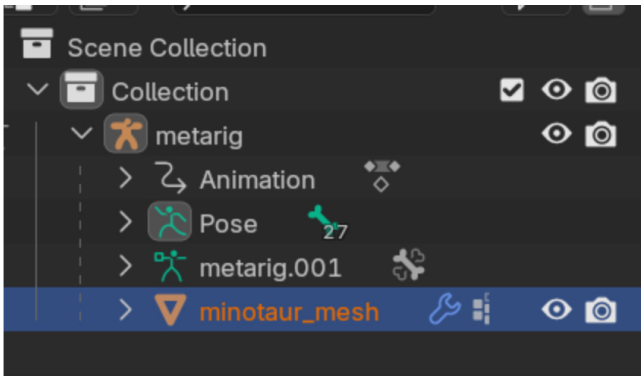


Figure 11: Showing the armature as parent

5.3.7 Keyframe Animation, With or Without Rigs

For rigid objects, animate directly in Object Mode: enable Auto Keying, set a first key, transform at new frames, and Blender records changes. For skinned meshes, switch to Pose Mode and key bone transforms. Use the Dope Sheet to manage clips per action and the Graph Editor to smooth easing. Keep clips short and purposeful (idle, walk, open, close) to simplify reuse in Unity.[27]

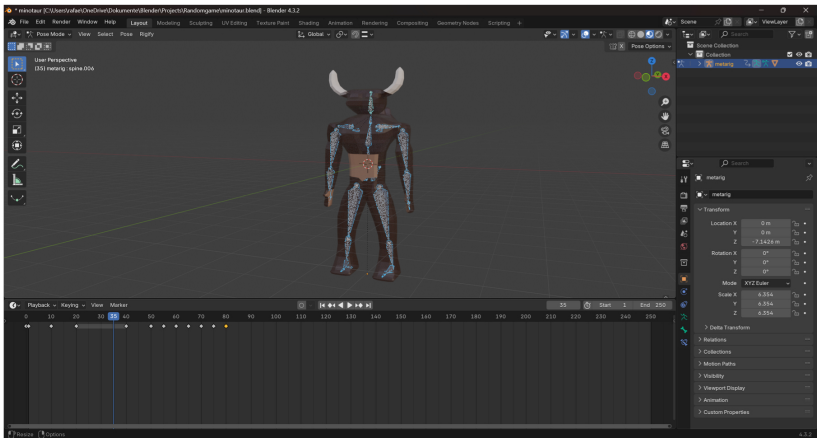


Figure 12: Keyframe animation with rig Fig

5.4 Modeling with Blender

This section distills practical Blender techniques we adopted during production. It complements the previous chapters with concise, actionable steps you can follow when modeling for real-time engines like Unity. The focus is on repeatable workflows and hotkeys that speed up iteration while keeping meshes clean and export-ready.

[27] Jelle Vermandere: Character, last accessed on: 06.11.2025.

The creation stage covers the modeling of all objects in the project: ramps, stairs, a pavilion-like platform, a vase, and characters. In Blender, every object is represented as a mesh, which is composed of vertices, edges, and faces. Vertices are points in three-dimensional space that define the corners of geometry. Edges are the straight lines that connect vertices, and faces are the surfaces enclosed by edges. By manipulating these components, it is possible to shape complex objects from simple primitives.

5.4.1 Ramps at 45°, 35°, and 25°

The ramps serve as inclined surfaces to test movement and interactions in Unity. Each ramp was based on a flat rectangular plane, which provides a simple surface for modification. To achieve the desired slopes, the vertical positions of vertices along the Z-axis were adjusted according to trigonometric calculations, using the tangent of the angle multiplied by the base length to determine the correct height. This ensures that the ramps are mathematically accurate and consistent with the intended gradient. To give the ramps depth, a Solidify Modifier was applied. This modifier adds thickness to a mesh along the direction of its normals, which are vectors perpendicular to each face of the mesh. Normals are critical because they define how surfaces interact with light and physics calculations, such as collision detection in a game engine. It is important to ensure that all normals are correctly oriented, so the Recalculate Normals function was used to automatically align them outward, guaranteeing correct shading and interactions. The ramps were scaled and transformed in three-dimensional space to match the planned dimensions. Applying these transformations sets the scale to the default while preserving size, ensuring consistency when exporting to Unity. This combination of accurate vertex placement, solidifying, and correct normals produced functional ramps with realistic thickness and proper surface orientation.

5.4.2 Stairs at 45°

The staircase was designed to connect surfaces at a steep angle and allow smooth transitions for movement. It was initially constructed from a cube, which was subdivided horizontally to define each individual step. Subdividing a mesh adds additional geometry, allowing precise shaping and control over proportions. To duplicate the steps evenly, the Array Modifier was employed, automatically repeating the geometry along a specified vector while maintaining consistent spacing and

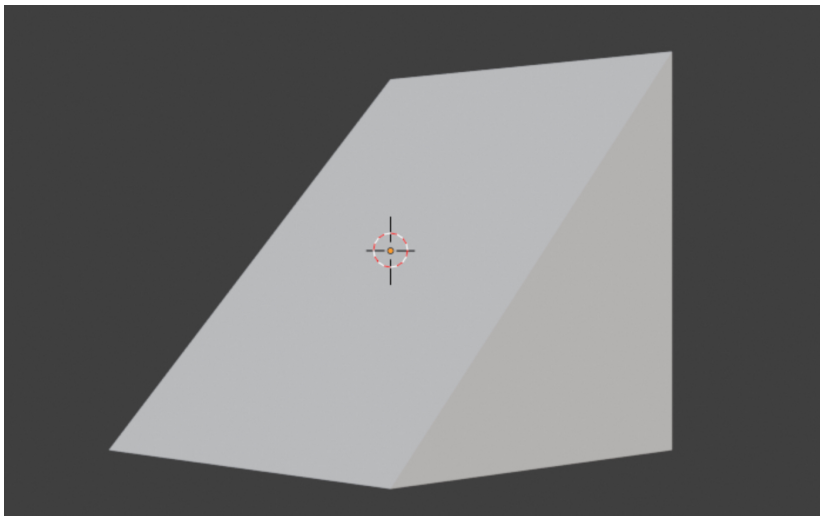


Figure 13: Slope with 45°

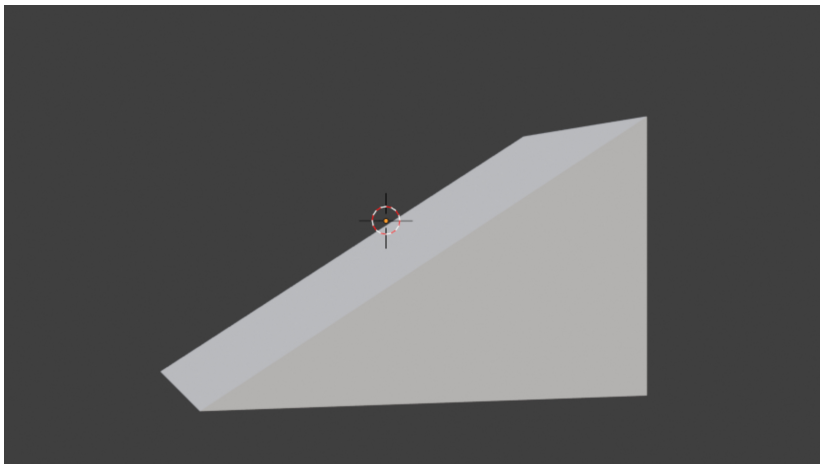


Figure 14: Slope with 25°

alignment. This method avoids manual repetition and ensures uniformity across all steps. Edges were refined using the Bevel Modifier, which adds small angled surfaces along edges. This softens sharp edges and produces more realistic lighting interactions by avoiding harsh shadows. By combining subdivision, duplication, and beveling, the staircase achieved both geometric accuracy and visual realism while remaining structurally consistent with the ramp slope.

5.4.3 Pavilion-Like Platform

The pavilion combines multiple primitive shapes to create a modular architectural structure. The floor was created from a scaled cube, forming a flat base. The roof was developed from flat planes extended through extrusion, a process that generates new geometry by projecting a selected surface along a direction. Pillars

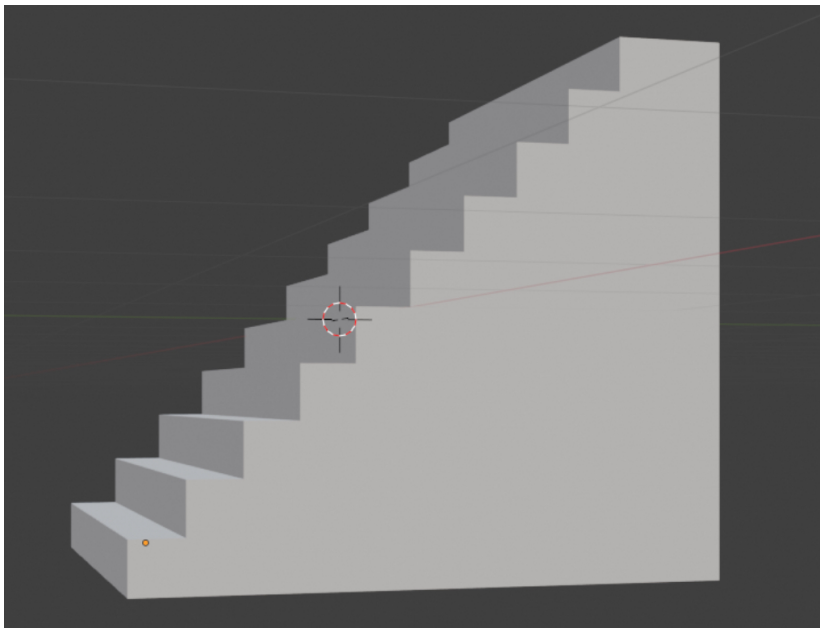


Figure 15: Stairs with 45°

were created from cylindrical meshes and positioned along the edges of the platform. To maintain symmetry and reduce repetitive work, a Mirror Modifier was applied, which automatically duplicates and inverts geometry along a chosen axis, ensuring consistent placement. All elements of the pavilion were organized into a hierarchical structure using parenting, where the movement or transformation of the main base affects all child objects. This allows the entire pavilion to be moved, scaled, or rotated as a single unit while preserving the relative positions of roof, pillars, and base, which is essential for integration into Unity as a modular structure.

5.4.4 Vase-Like Structures

The vase was modeled starting from a cylinder. Vertices were scaled along the vertical axis to produce a tapered profile, forming a smooth curve from the base to the opening. To refine the shape, loop cuts were added, which insert additional edge loops along the mesh to allow precise control over curvature and shape. Proportional Editing was used to adjust multiple vertices simultaneously, creating smooth transitions and natural curves across the surface. To enhance smoothness and realism, the Subdivision Surface Modifier was applied, which subdivides each face into smaller segments and averages their positions to create a refined surface. The Solidify Modifier added thickness to the walls of the vase, transforming it from a flat shell into a volumetric object suitable for physical interaction or collision detection in Unity.

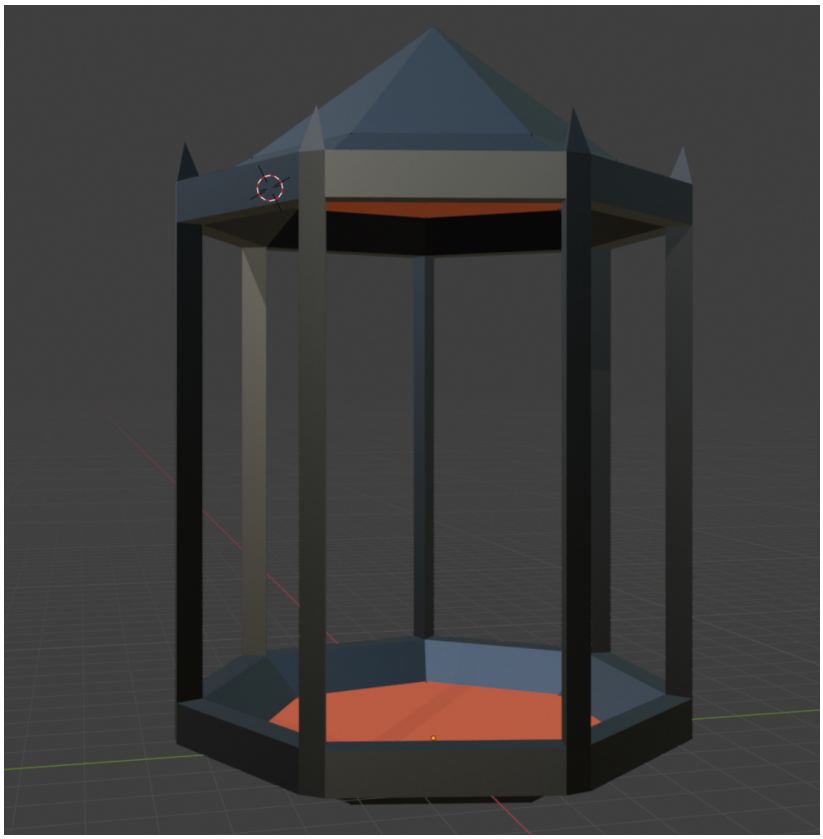


Figure 16: Greek Pavilion created on Blender

5.4.5 Characters

Character models were first developed based on hand-drawn concept sketches that defined the general appearance, proportions of each figure. These were followed by movement sketches, which visualized possible poses and motion sequences for later animations.

The actual models were then created in Blender from basic humanoid meshes. Loop cuts, extrusion, and proportional editing were used to define body contours and limbs. Unique features were added to distinguish individual characters and emphasize their roles in the game. Rigging was performed using the Armature system, which establishes a hierarchical skeletal structure of bones to control the mesh. Each bone influences a specific portion of the model, while the hierarchy ensures that movements in parent bones correctly affect connected child bones.

Weight Painting was applied to assign bone influence to specific vertices, preventing unnatural mesh deformations during animation. Animations such as walking, running, or climbing were created by manipulating the armature in Pose Mode at defined keyframes along the timeline. The Graph Editor was used to refine motion

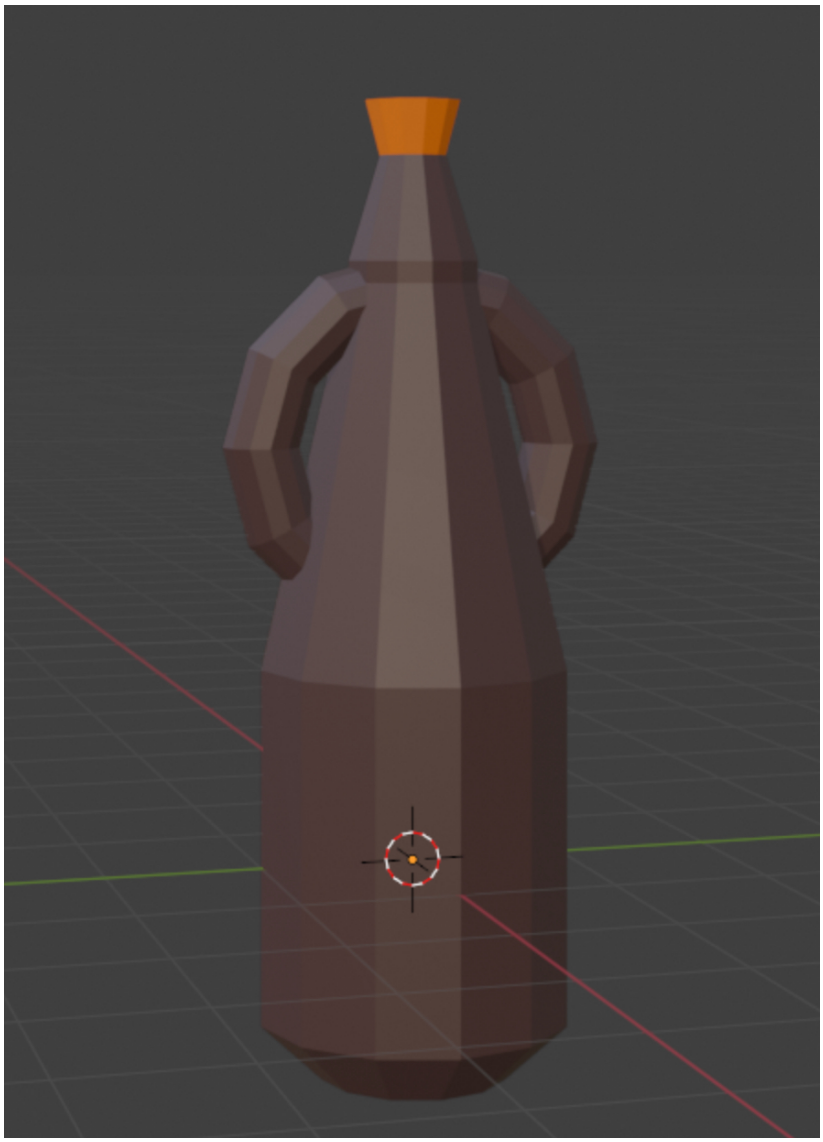


Figure 17: Vase structure example on Blender

curves, smooth transitions, and precisely control timing. Each animation was stored as a separate action in Unity as an independent sequence

5.4.6 Texturing

After modeling, textures were applied to all objects. Texturing involves creating materials that define how surfaces appear in terms of color, reflection, roughness, and surface detail. UV Mapping was performed to translate three-dimensional surfaces into two-dimensional coordinates, allowing textures to wrap correctly around complex shapes. Smart UV Projection was used to automatically generate UV maps for objects with intricate geometry, ensuring textures are applied without distortion. Materials were developed in Blender's Shader Editor, a node-based system

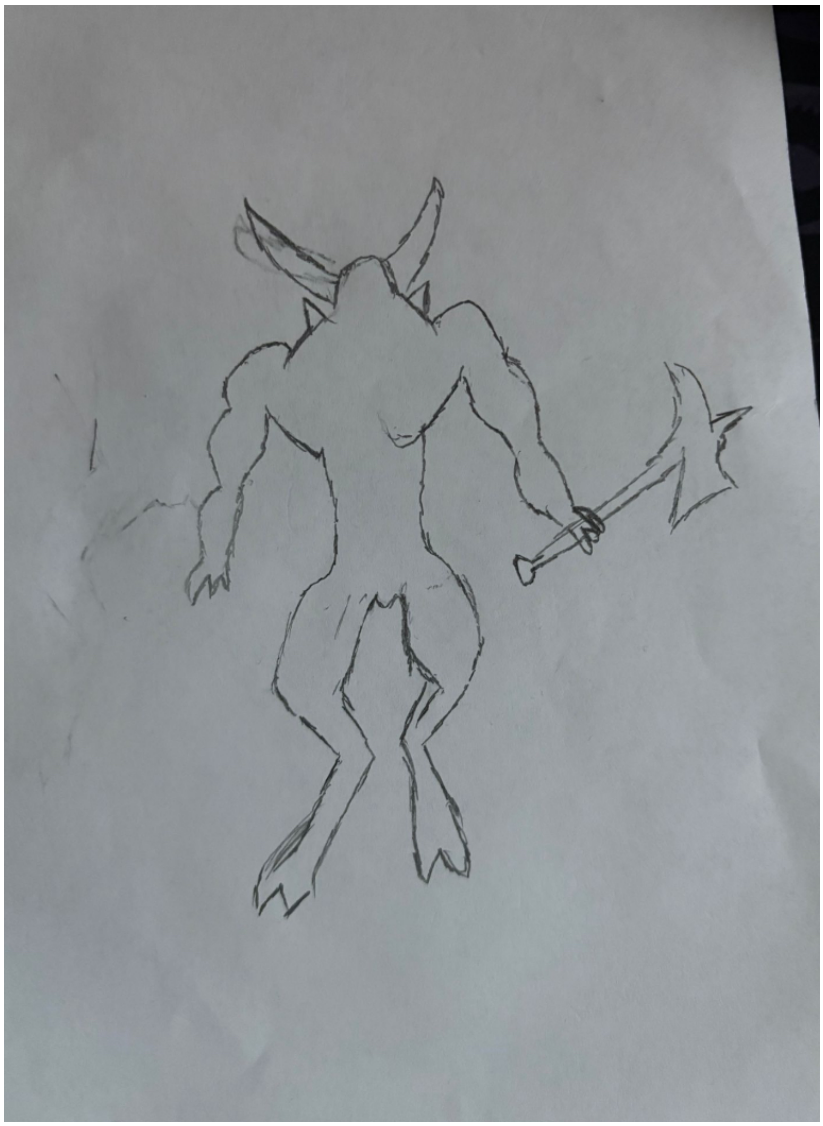


Figure 18: Sketch of Minotaur

that enables combining multiple inputs to create complex surface properties. The Principled BSDF Shader was the primary material, allowing control over base color, metallic properties, roughness, and normal maps. Procedural textures generated variations in color, roughness, or surface detail mathematically, enhancing realism without increasing polygon counts. Concrete textures with roughness and bump maps were applied to ramps, wood textures to stairs, combinations of stone, wood, and metallic materials to the pavilion, ceramic glazing to the vase, and layered materials for characters' skin, clothing, and features. These textures interacted with light and shadows according to the normals and geometry of each mesh, enhancing realism in both Blender and Unity.[28]

[28] Blender Guru: Shading, accessed on: 06.11.2025.

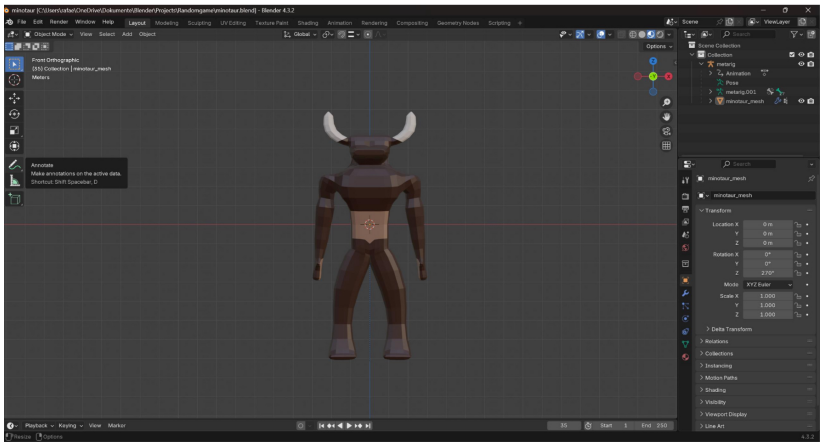


Figure 19: Minotaur in Blender

5.4.7 Export to Unity

The final step was exporting the models and animations to Unity using the FBX format, which preserves mesh geometry, UV maps, armatures, and animation data. Transformations and modifiers were applied to ensure that scale, rotation, and all procedural changes were correctly incorporated. Normals were checked to confirm proper lighting and collision behavior in Unity. Animation sequences were exported alongside characters, allowing Unity's Animator system to recognize them as discrete clips. All objects were imported into Unity, tested for alignment, texture accuracy, collision, and functional interaction. The ramps and stairs maintained their slopes and spacing, the pavilion retained its modular hierarchy, the vase retained its shape and surface details, and characters animated smoothly without mesh distortion. This workflow ensured a seamless transition from Blender to Unity, preserving both aesthetics and functionality.

5.4.8 Rooms Implementation

The implementation of rooms was carried out entirely within Unity, while the 3D assets used to construct them were created beforehand in Blender. These assets included architectural elements and decorative models designed according to the game's ancient Greek theme, characterized by marble textures, symmetrical structures, and occasional bronze statues. The central element of the room system is a custom Unity script that automatically places rooms based on the position and orientation of their entrances. This system ensures that rooms connect logically during level generation and that the layout remains coherent, even when generated dynamically. To enable this functionality, I had to design individual room variations

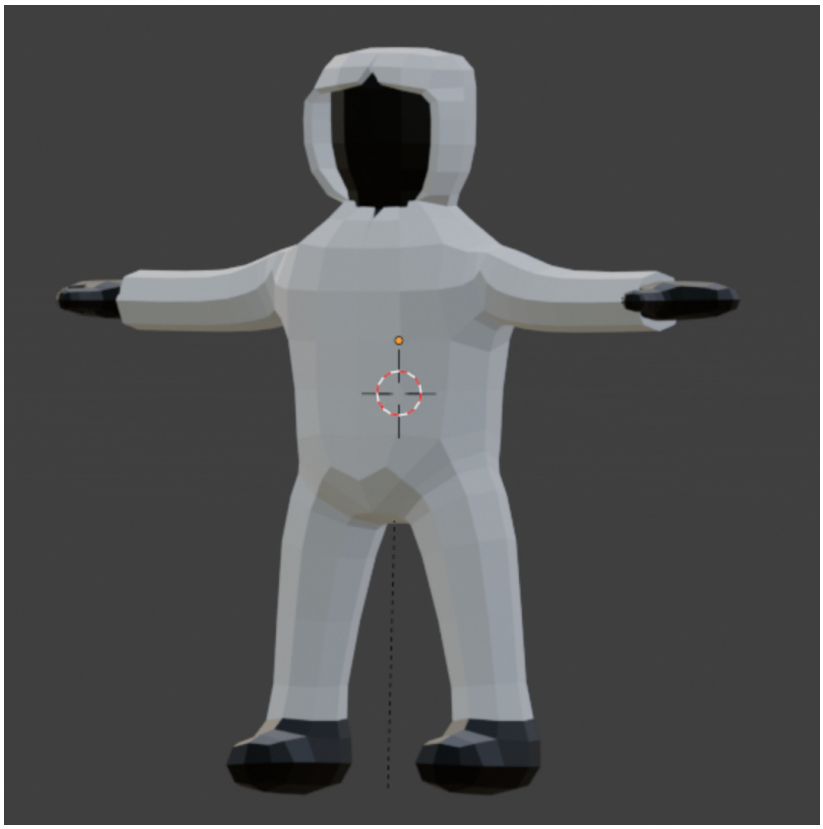


Figure 20: Character/Figure example

for each possible entrance configuration. Each variation was constructed manually in Unity by combining and arranging the available models. In total, 63 unique room variations were implemented, each representing a different combination of entry points. This process required significant planning and iteration, especially due to the game's unique gravity mechanic: the player can alter the direction of gravity individually, meaning that each room must function correctly on all six sides of a cube. To achieve this, every room had to be rotated and tested for each possible gravity orientation to ensure that geometry, objects, and navigation behaved as intended. Designing and aligning all rooms to fit this mechanic proved to be one of the most time- consuming parts of the implementation. It demanded both technical precision—to maintain proper alignment and prevent overlaps—and creative variation, as the limited number of models had to be reused in different configurations to avoid visual repetition. The final result is a flexible and coherent room system that supports the labyrinth-like structure of the game world while maintaining visual consistency and thematic coherence

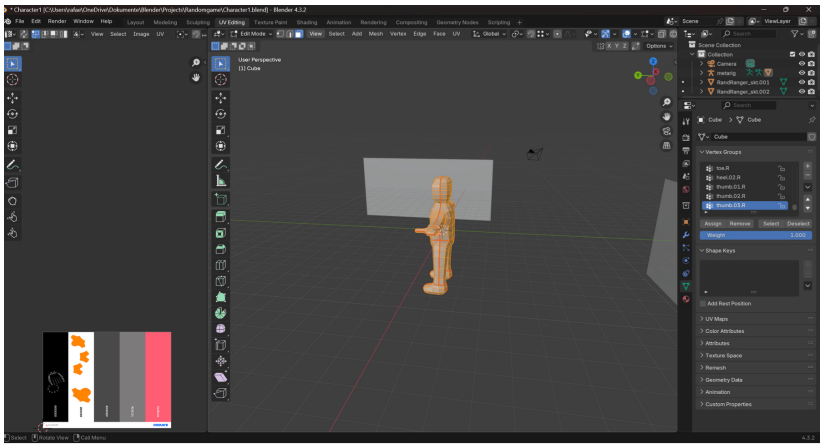


Figure 21: UV Editing Screen

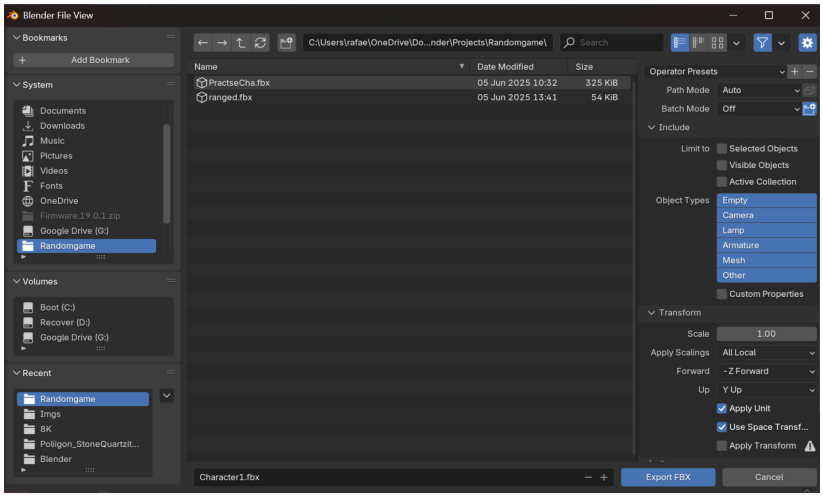


Figure 22: Export Window

5.5 Programming

5.5.1 Random Generation

As mentioned before, all the levels in the game are randomly generated, however we do not have truly unique levels where every part is random, but we created a system that will randomly arrange premade individual rooms into unique levels. For simplicity we decided early on that all rooms are, at least internally, going to be the same size. This means that the exits of a room are always in the same positions relative to one another. This allowed us to still make smaller rooms by blocking off parts of a normal sized room, but, more importantly, we could use a relatively simple grid-based system for the random generation. By having all rooms in a single grid we did not have to account for exits that do not align properly or check if rooms would overlap. Now, we have mentioned exits quite a few times, but what does it

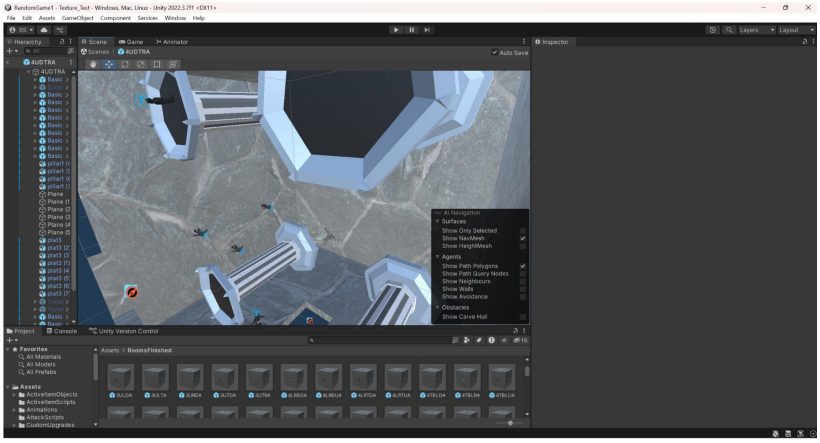


Figure 23: Picture in Unity Showcasing a Room

actually mean for a room to have certain exits?

Since our generated dungeons are 3-dimensional, allowing for up/down exits, a room can have up to 6 exits and those exits can then be arranged in different configuration, allowing, in total, for 63 different types of rooms. In order for the generation to always work, it needs to have a room that can fit in any possible configuration of connecting exits. Therefore we had to manually create all 63 possible rooms. First we created 3D-models for all configurations of exits relative to each other, meaning we did not create all possible rotations of exits. For a room with 2 exits for example, we had to create 2 models, one with the exits opposite one another, and one with the exits forming a corner. Once we had all these models, we exported them into Unity in order to also create all possible rotations, we chose this method because exporting rotations of 3D-models into Unity is complicated, and it was easier to just do the rotations with the already imported rooms. Now, for the system that actually arranges the rooms into unique levels, we once again used the exits as a basis. First we named all the room objects according to their total exit number and which exits they had (Up, Down, Left, Right, Top, Bottom) adding the first letter to the name. For example, a room with 2 exits, one to the left, one to the right, would be named 2LR. Next, we created a 2-dimensional array of strings, which will be our grid, and a list of positions in said grid where there still needs to be a room placed. We used an array to keep track of what exits a certain position in the grid needs to have by adding a letter representing the direction of the required exit to the corresponding item in the array. We once again used the letters we also used in the naming of the rooms. Further we defined an array of objects in which we placed all the room objects that we created earlier and we also defined other variables that let us adjust the parameters of the generation, such as maximum room count or the distance between rooms.

With these main factors defined, the generation begins by placing a certain starting room in the center of the grid and adding the corresponding exits to the adjacent positions in the grid. The script then selects the first item on the list of positions that need to be filled and starts the process of eliminating all rooms that do not have the required exits. Thanks to the naming of the rooms this is rather simple, it simply requires us to check if a certain direction of exit is required by reading the string for the selected position, and then looping through all possible rooms, checking if the same letter is contained in the name of the room and eliminating the room, if it does not. After repeating this procedure six times for all directions, we are left with a list of rooms that will connect to all adjacent rooms. You might think that we are already done, having eliminated all rooms that would not connect to the adjacent exits, however we also had to account for the reverse, meaning that we had to check if one of these possible rooms would have exits that would lead in a direction where one of the neighbors does not.

At this point, the script splits into two possible paths, the first is used for closing off the level once it has reached the minimum number of rooms, the second path is used when the minimum number of rooms has not yet been reached.

We will first examine the second path.
The second path is simple but long. Essentially it selects a random one of the possible rooms and then checks all adjacent positions in the grid if there is already a room there and if there is, whether it has a corresponding exit to the room the script is trying to place. For example, if the selected room has 2 exits, one to the left and one to the right, it checks if there is a room to the right of it, and if there is, it checks if it has an exit to the left. It then repeats this for all exits. If something is invalid, the script removes the room from the list of possible rooms and randomly selects another room from the list. This is done randomly so we can have multiple versions of a type of room, without the same version being selected every time. Finally there is one more check; since the minimum number of rooms is not yet reached, the script has to make sure that by placing the selected room it is not closing every remaining exit. This is achieved by taking the total number of exits a room has and then subtracting 1 for each "connected" exit. If the result is larger than zero, the room is valid and it is placed and the script removes the now filled position from the list of empty positions and adds all exits to the array.

As for the first path which is used when the minimum number of rooms is reached. Once the minimum is reached, we want the script to stop and place only as many rooms as necessary to close all remaining exits. This path works similarly to the first path, but a room is only valid if it connects perfectly, leaving no unconnected exits.

Furthermore there is one more condition the script has to validate namely chest rooms. Chest rooms are a second type of rooms that contain an upgrade. In order to make sure that the player finds enough upgrades, there is a minimum number of chest rooms. Chest rooms can be randomly selected just like normal rooms, however, if placing a chest room in every open position would reach the minimum number of rooms, the script activates chest-mode, which means it will only place chests until all open positions are filled.

5.5.2 Changing gravity

In order to make it possible for the player to navigate the 3-dimensional level structure, we decided to implement the ability for the player to change the direction of gravity freely. The gravity script is the longest one we made, however most of it is actually repeated multiple times, so in reality there are only a few unique things in the script.

We begin with is the function that is responsible for actually applying gravity to the player, no matter what direction. Because Unity's default rigidbodies do not support gravity in other directions, we had to make our own gravity. Thankfully however, this is not very complicated as gravity is simply a constant force that has to be applied to the player. In our version, gravity is a vector multiplied by a constant that defines how strong the gravity should be and this vector is then applied as a force to the player in the fixed update, meaning that the code will be repeatedly executed with a constant delay, unlike the "regular" update which runs the code once per frame, which would make gravity dependent on the framerate, as it would be applied more often, and thus be stronger if the framerate is higher.

Now for the part that actually changes gravity, as well as rotates the player object accordingly, this part was rather difficult to code because rotations in Unity are difficult. There is no easy way to just rotate an object, for example, 90° to the right, relative to the direction the object is facing. Because of this, we decided to manually code every rotation the player could rotate to.

At first we thought we would just have to find six rotations, however we soon realized that the rotations also change based on which direction the player is facing when they start the rotation, and thus, we had to manually find and define twenty-four 3-dimensional vectors. To do this we used a system to name the vectors that uniquely identifies each one by the plane the player is standing on, and which direction they are facing. The names of the vectors were constructed by writing the direction the

player is facing as pX/pY/pZ for positive X/Y/Z and nX/nY/nZ for negative, and then doing the same thing again for the plane on which the player is standing. For example, the vector that gives the rotation of the player when they are facing on the positive X and standing on the negative Y, would be pXnY.

After we defined all the vectors, we had to create a function to gradually rotate the player. We did this by dividing the difference between current and target rotation, and rotating the player bit by bit with a short delay in between.

Further, we defined a function to round the relative direction vectors of an object; these direction vectors are the relative up, forward and right direction of any object. This means that the relative "up" vector always points straight up from the object, no matter how you rotate it. In our case, we needed to round these directions, meaning we wanted to know which axis was the "main" one for, for example, the relative up vector. We did this by checking whether each component of the vector was larger than the square root of 0.5. This works because the length of the relative vectors is always 1. For this reason and because one axis is always irrelevant because it is perpendicular, if one of the vector components is larger than the square root of 0.5, the other one can't be as well, because if both of them were larger then $\sqrt{0.5}$ then the length would be larger than 1 (because $\sqrt{\sqrt{0.5}^2 + \sqrt{0.5}^2} = 1$ (Pythagorean theorem)) . Note that we also assume that it will never be the case that both are exactly $\sqrt{0.5}$, because it is too precise to ever realistically happen. The function then returns a vector where all components that are not larger then $\sqrt{0.5}$ are 0 and the one that is, is 1. With that, all required functions are defined.

The script starts when it detects exactly one of four inputs, the arrow keys by default, and then sets the gravity vector to the rounded relative vector corresponding to the button pressed. For example, if the player inputs to rotate to the right, it sets the gravity to the axis that is the closest to the right of the player object. The script then starts one of four separate functions, one for each direction the player can rotate, which has code paths for every direction the player could be facing and standing on. Thankfully we do not have to check if the player is facing the direction they are standing on because this never affects the outcome, however we do still have to check for a lot of rotations. Specifically we do it in the following order:

1: Is the player facing either the positive or negative y axis?
- 1.1: Is it the positive y axis?
  - 1.1.1: Is the player standing on either the positive or negative x axis?
    - 1.1.1.1: Is it the positive x axis?
      - rotate to the correct direction

1.1.1.2 Is it the negative x axis?
rotate to the correct direction
.....etc.

All in all, we have 24 possibilities per direction the player can rotate, so 96 possibilities in total.

5.5.3 Upgrades

In total there are four different types of upgrades the player can pick up in the game: Random upgrades, stat upgrades, custom upgrades and active items. The stat upgrades are relatively simple: All of the player's stats are stored in a single script, therefore applying a stat upgrade, random or not, just requires updating the variables of the player-stats script. Thus, all the stat upgrades need are public variables to assign the increases and decreases in stats and a function to apply the changes.
The random upgrades work by distributing a total amount of increases, given by the rarity of the non-random upgrade, that are randomly distributed across the possible stats. These values are then turned into a string and displayed in the UI once the player picks up the upgrade.

As for custom upgrades, the process of applying them is more difficult. First, what we mean by custom upgrade is any upgrade that requires additional code to be executed in order to function. One example of a such an upgrade would be the melee-dash upgrade, which causes the player to dash a short distance when using a melee attack. This is accomplished by having a new script that, when attached to the player, executes a dash when receiving the input for a melee attack. While the concept of a custom upgrade is simple, the process of actually attaching them to the player object is very much not so. This is because, in Unity, there is no variable type that can be assigned a script type. You can only store specific instances of a script in variables. Hence, we were not able to simply create a script that attaches a given upgrade-script to the player object; instead we had to add the part of attaching the script to the actual upgrade script itself. More specifically, we created a template that contains the code for attaching itself to the player, should the object it is currently attached to not be the player. Because of this method we only need set the name of the script to be added. With this, we could create an object for each upgrade that already had the script attached which could then, unlike the script itself could be assigned to a variable. Through this, when the player chooses to pick up a custom

upgrade, the upgrade object, which handles the generation of the random upgrade, the UI and the pickup, would instantiate the assigned script object, which then contains the upgrade script. The script then realized that it is not attached to the player object and adds itself to said object before deleting the script object.

The active items work a bit differently than the other types of upgrades and are consequently also programmed differently. In our game, an active item is any custom upgrade that is activated by pressing a dedicated active item key, in addition, the player can only have one active item at a time, because otherwise the active item button would trigger all effects at once. Because of this restriction, we wanted the player to be able to switch back to an active item that they discarded earlier and hence, we needed to spawn the correct upgrade object if the player picks up a new item. The pickup of the item has once again the same problems as the custom upgrade, requiring the attaching of a specific script to the player, and so we were able to use the same solution with some minor tweaks. Again, just like with the custom upgrades, we once again had two objects, one with the actual upgrade script, and one that spawns said object. The pickup functions with only two changes: first we want the player to be able to see what item they currently have equipped and, because you cannot assign public variables (variables that can be set in the editor) for a script but only for a script already attached to the object, we need to transfer the public variables from the script on the upgrade object to the script on the player object.

Second, there is a feature in Unity that, while in most cases actually quite useful, is detrimental to the fact that we want to instantiate the same object the player picked up once the item is dropped again. Said feature is the fact that, when assigning a public gameobject variable on a prefab ("templates" for objects that are not present in a scene but can be instantiated from our assets) to be that same prefab and then instantiating the prefab, the variable instead becomes a reference to the specific instantiated gameobject, rather than the prefab. Hence, if we try to instantiate that object, we are trying to instantiate an object that has long since been deleted. While we might have been able to do some workaround where we don't actually delete the object but just deactivate it and then move to where it needs to be, there would still have been problems with multiple item objects and changing levels. Therefore we opted for a less elegant, but still functional, solution, by creating copied of each active item object and assigning the other copy respectively, and, since the assigned object is then not the same prefab the script is on, it remains a reference to a prefab, rather than one to an instantiated object.

5.5.4 Enemies

In the game there are four types of enemies: Melee enemies that deal contact damage, ranged-melee hybrids that switch to ranged attacks when the player is unreachable, turrets that constantly shoot at the player and summoner enemies that spawn either melee or melee-ranged enemies. All the enemies share the same script for their health, containing variables for maximum health and current health that can be adjusted to fit any enemy.

Additionally, all enemies use the default Unity navmesh pathfinding system, which, thankfully had been updated to support different rotations not too long before we started the project, allowing us to use it with only minor tweaks. One limitation that it has however is the fact that it is, while not quite impossible but very difficult to properly have an enemy change gravity. This lead us to create the melee-ranged hybrids, which are a solution to the problem that the player was able to just stand on a wall on which there are no enemies and defeat an entire room using ranged attacks. Because the navmesh package is easy to use, the basic enemy script really only contains a function for dealing contact damage that is shared with the other enemies and a single line of code to continuously set the target of the pathfinding to be the player.
The turrets and summoner enemies are also relatively simple, in fact they both use the exact same script except that the summoner is set to "shoot" other enemies instead of bullets. Said script is doing not much more than instantiating whatever object is set to be the bullet and giving it a velocity towards the player. The only other thing the script does is account for the fact that turrets can only be attacked using melee attacks. We did not want however that it is possible for a turret enemy to take hundreds of melee hits, should the player opt for a ranged build, and thus decided to make defeating a turret require exactly five melee hits, no matter the damage. Because of this, the turret script also sets the health of the turret to be five times the player's melee damage.
The script for the melee-ranged hybrid enemies is by rather more complicated than the others. First it checks whether the player is reachable, by checking if the player has the same gravity as the enemy and if so, whether the path to the player is complete. If the player is reachable, the code is exactly the same as the melee enemy, setting the pathfinding target to be the player. If the player is not reachable on the other hand, the enemy enables ranged mode. It first waits for three seconds and then checks if the player is still unreachable (this was done to prevent the enemies from enabling ranged mode it the player so much as jumped). If this is the case, the enemy

enables a particle system to show that it is currently charging a ranged attack, and then waits for a few seconds to charge the attack. Once it is done charging, the enemy then sets the target position of the attack to the player's current position, but again waits a moment to actually attack in order to allow the player to dodge the attack. After waiting it then simultaneously creates a line renderer to the target position and a spherical collider at the target position. It does this because the beam attack is not actually a beam but only a collider that will deal damage if the player is still close enough to the target position. Further, the enemy assigns the different damage variables of the collidier and then waits for another moment before destroying the collider again.

5.5.5 Pity-system

Our game features a so-called pity-system; this means that with every instance of "bad luck", it becomes more likely for the player to experience "good luck". In our game, this means getting good and bad upgrades, defined by the rarity of the upgrade. To implement this system, we created a script attached to the player that tracks whenever the player picks up a common or uncommon item and adds an percent increase to the chance of getting rare and legendary items with the increase being half as big for uncommon upgrades as for common ones. If the player picks up a rare item, nothing changes, we chose to not decrease the chances when picking up a rare item, because it would lead to the pity system only ever granting rare items. This would happen because the chances would increase enough to get a lot of rare items but not enough for it to be likely to get a legendary item. Consequently, we chose to only reset the pity system when picking up a legendary item, at which point it completely resets.
While, in theory, the actual increase of the odds should only require changing some variables in the upgrade spawning objects, the problem is that the upgrade is chosen as soon as the upgrade spawner is instantiated. Because of this, we cannot retroactively change the odds of spawning different rarities of upgrades and actually get a change in the spawn rates. To solve this problem, we implemented a solution that we later also used for the spawning of enemies. Since all the rooms are on a set grid with a set distance apart, it is possible to exactly determine which room the player is currently in, by determining in which grid cell they are. Through this we can know when the player entered a specific room and, as long as we start with all objects in that room deactivated, can tell the room to only at that point activate the objects. Thus, the upgrade will only be chosen once the player enters the room and is therefore affected by the changed odds of the pity system.

6 Questionnaire for Playtesting

6.1 Method and approach

We designed a questionnaire to assess our proof of concept in terms of seven core themes, all of themes are connected in some way or the other to the conclusion of the literature review. These were the themes: fairness, control/agency, clarity, excitement, satisfaction, replayability, and frustration and a set of prototype-specific checks that confirm key implementation choices, such as deterministic combat (no random hit/miss), the clarity and fairness of stat-based upgrades, whether modular rooms feel coherent yet fresh, whether players notice the pity mechanism, whether they understand when luck matters, and the absence of monetized randomness. In line with AAPOR, we wrote short, as the AAPOR guidleine suggested: single-concept, neutral statements, arranged the flow logically, and used online randomization with AI within constructs, where appropriate, to limit order effects; we also kept the burden low (about five to seven minutes), offered "Not sure" for subtle mechanics, and included an attention check and a consistency probe to support data quality and transparent reporting.

As already mentioned, each topic is linked to specific items and a corresponding statement from our literature review. For fairness, we use C1, C2, and the reverse-worded C3 to validate that prioritizing input randomness and avoiding output randomness supports perceived fairness and that random events should feel rule-consistent rather than arbitrary.

- The results in the game felt fair.
- The randomness elements in the game felt sensible.
- Bad luck mattered more than my choices.

Control/agency is captured with C4, C5, C6 (reverse), and C24 to test the claim that input randomness preserves planning space and agency, whereas output randomness undermines control and can feel unfair.

- My success mostly came from my choices, not luck.
- I could plan ahead because surprises came *before* my choices.
- My actions felt less important because there was too much luck.
- My success was mainly due to my decisions, not luck. (Duplicate of C4)

Excitement is assessed with C7, C8, and C9 (reverse), reflecting the proposition that well-timed unpredictability generates positive tension while poorly timed or opaque randomness harms engagement.

- Surprises in the game made it more exciting.
- Unpredictable moments were good and tense.
- When something random happened, I lost interest.

Clarity is measured by C10, C11, and C12 (reverse) to check the expectation that transparent, understandable randomness—similar to dice/cards paradigms or visible risk envelopes—reduces confusion and perceived unfairness.

- I understood when and why random things happened.
- It was clear how luck could affect my next choice.
- I was confused about how luck worked in the game.

Satisfaction (C13–C15) addresses whether outcomes align with skill and effort even in the presence of luck.

- Results matched my effort and skill.
- I was mostly happy with the results of my choices.
- Even with luck, skill was rewarded.

Frustration (C16 and C17, both reverse-worded) examines the known risk that output-style randomness increases annoyance and loss of control, and whether our design minimizes such effects.

- Luck in the game annoyed me.
- I was rarely annoyed by luck.

Replayability (C18–C20, with C20 reverse-worded) evaluates the premise that input randomness via shuffled modules and stat-based upgrades creates fresh runs without sacrificing coherence.

- Because of luck, each run felt different.
- I would play again because new situations can happen.
- The variety felt forced or quickly the same.

For data quality, C23 serves as an attention check, and C24 mirrors the agency theme to diagnose straightlining or inconsistency, as recommended for transparent validity reporting.

- Attention: Please tick **5** for this line.

- My success was mainly due to my decisions, not luck. (Duplicate of C4 and C6 (reverse))

The implementation checks complement these perceptions: D1 confirms the removal of output randomness in combat, D2–D3 verify that upgrades are clear, fair, and meaningfully influence play, D4–D5 test room coherence and freshness under modular sequencing, D6 captures awareness of the pity system intended to reduce streak frustration, D7 checks that players know when luck will matter, and D8 confirms the absence of monetized randomness, aligning with our ethical stance. This mapping is documented in our item-to-finding overview and underpins the instrument's traceability from hypothesis to question and back.

- In combat, there were no random hit/miss outcomes.
- The upgrade choices made me change how I play.
- The upgrade choices were clear and fair.
- The order of rooms made sense and still felt fresh.
- No room felt out of place or unfair.
- If I got several weak rewards in a row, the next one seemed better. (Pity-System Awareness)
- It was clear *when* luck would matter.
- I did not see any paid random rewards.

6.2 Justification of the Questionnaire

The questionnaire was purpose-built to measure the experience we care about and to verify the prototype's key design decisions, which we got with our literature review. We choose to follow AAPOR principles for wording, ordering, burden, randomization, and transparency, as it integrates questions to cross-check multiple answers to see if the answers are consistent if the question varies a bit. Additionally the various built-in check-points make sure that the responses are reliable and can be used for further optimization of our game.

6.3 How do we evaluate the results?

Since our dicsussion and conclusion will follow soon, we would like to take a moment and explain how we evaluated the results gathered from our playtesting. Our evaluation plan follows a simple scoring approach. Reverse-worded items are first flipped so that higher values consistently indicate more favorable perceptions,

using $x' = 8 - x$ on the 7-point scale. For each topic, we compute the mean on the native 1–7 scale and then apply a linear min–max normalization to a 0–100 index,

$$\text{Index}_{0\text{–}100} = \frac{\text{Mean}_{1\text{–}7} - 1}{6} \times 100,$$

such that 50 denotes a neutral position and values near 70 reflect clearly positive responses; frustration is additionally reported inverted as $100 - \text{Frustration}_{\text{index}}$ so that all constructs share a uniform "higher is better" direction. To support methodological rigor and clarity, we planned to report reliability (Cronbach's α) per construct, 95% confidence intervals for group means, and known-groups comparisons (Noobs vs. Pros), in line with AAPOR's transparency guidance. Our success thresholds were set to be decision-friendly yet realistic for early iterations: fairness, agency, clarity, satisfaction, and replayability target ≥ 70, because a score around 70 on the 0–100 index corresponds to a clearly positive average (roughly 5.2/7) rather than mere neutrality; excitement and frustration_inverted target ≥ 65, acknowledging that taste, pacing, and subtle design trade-offs can make these constructs more variable; deterministic combat (D1) targets $\geq 90\%$ "Yes" given its central role in avoiding output randomness, and the absence of monetized randomness (D8) is set at 100% "Yes" as an ethics/policy constraint; other implementation checks (D2, D3, D4, D5, D7) aim for index means ≥ 65 to demonstrate solid performance with room for polish toward ≥ 70. These choices align with AAPOR's emphasis on clear scoring and documented validity checks, and they create a balanced framework that supports iteration without conflating index values with percentages of respondents.

7 Results of our Playtesting

The analysis included 12 participants. 7 participants play videogames rarely and count to the category "Inexperienced" and 5 participants play videogames regularly, already aware of some randomness-knowlege and consequently count to the cateogory "Experienced". Reported run completions for the session were three runs.

7.1 Questionnaire Results of Part C

Results were calculated and for each question the total mean for each group (Noob or Pro) was taken. The mean of questions representing themes were then taken again

and presented in a Inexperienced vs. Experienced column diagramm, as indexed results, as explained in previous chapter.

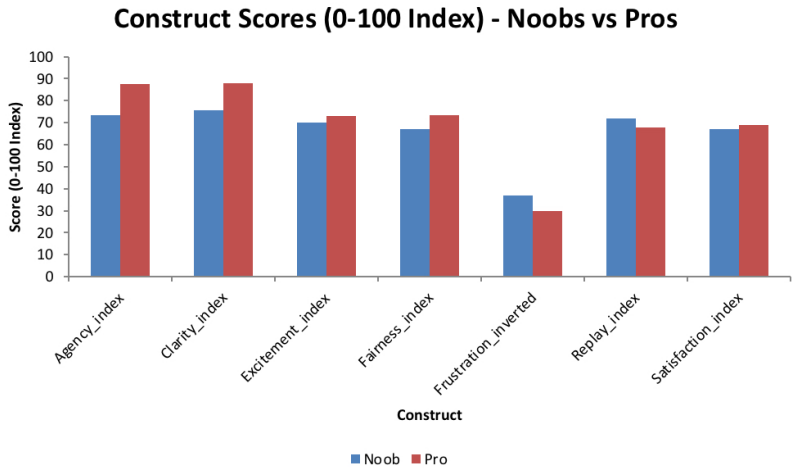


Figure 24: Inexperienced vs. Experienced Mean values

Here the exact numbers of the table:

Inexperienced vs. Experienced

- Agency: 73.4 vs. 87.5
- Clarity: 75.6 vs 88
- Excitement: 70 vs 73
- Fairness: 67.2 vs. 73.3
- Frustration inverted: 37 vs 30
- Replay: 72 vs 68
- Satisfaction: 67 vs 69

7.2 Questionnaire Results of Part D

Inexperienced vs. Experienced

- Random stat-based upgrade (R.S.B upgrades): 100 vs. 100
- Modular room sequencing for "handcrafted" content: 70 vs. 71
- Transparency: 80 vs 83

7.2.1 Results of Detirministic Combat and Pity-System

Inexperienced vs. Experienced

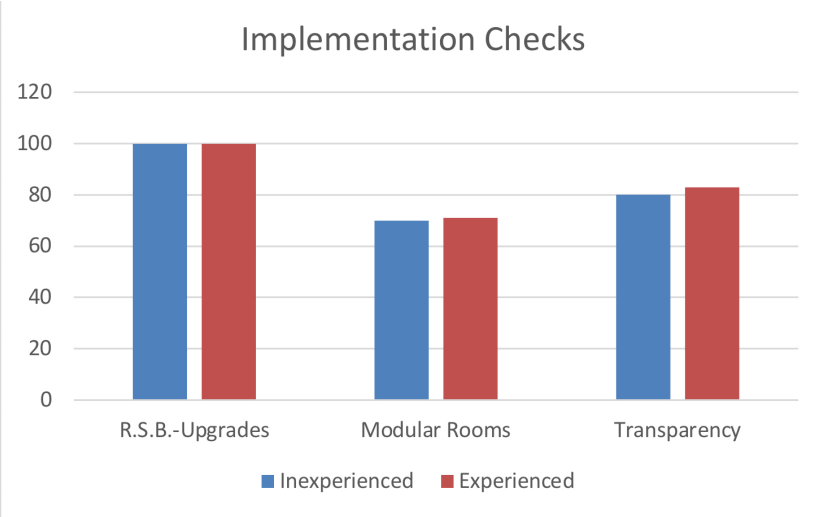
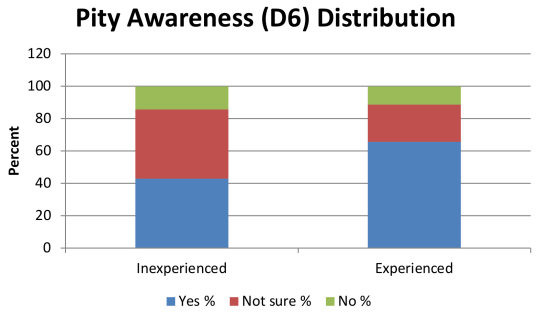
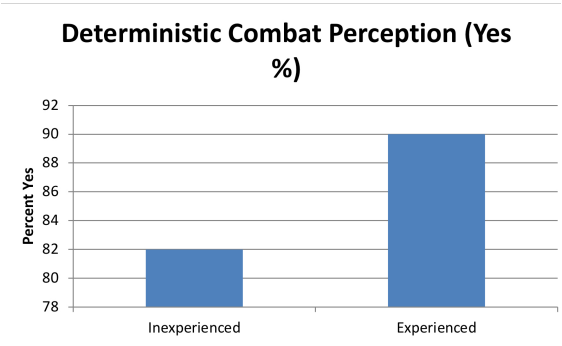


Figure 25: Indexed Mean values

- Pity System: Yes: 43% vs. 66%, Not sure 43% vs. 23%, No: 14% vs. 11.2%
- Detirministic Combat: Yes: 82% vs. 90%, No: 18% vs. 10 %



(a) Pity Awareness in Percentage Comparison



(b) Detirministic Combat Perception (Yes Percentage) Comparison

Figure 26: Implementation Checks

8 Discussion and Interpretation

This chapter examines the implications of the results for the PoC's success. The analysis begins with the core questionnaire section (C) before examining specific implementation outcomes (D). For clarity, the "Inexperienced" group will be referred to as "InX" and the "Experienced" group as "X" throughout this discussion.

8.1 Core Questionnaire

8.1.1 Fairness and Frustration

Fairness

In many roguelike games, fairness represents a significant design challenge. Previous research on games such as XCOM demonstrates that output randomness can undermine perceived fairness. Consequently, questions C1–C3 assessed whether the prototype successfully avoided unwanted output randomness and effectively implemented input randomness. The results yielded a fairness index of 67.2 from group "InX" and 72.3 from group "X". With the success threshold established at an index of 70, the "InX" group's response indicates room for improvement.

The below-threshold fairness rating from group "InX" may be attributed to the inherent variability of input randomness—despite occurring before player decisions, the randomized elements can still produce more or less favorable scenarios. The fully random algorithm may have generated challenging room sequences for certain playtesters. Additionally, in some rounds the pity system appears not to have activated as often as needed, likely due to the short runs and higher death frequency, which proved insufficient for algorithm activation. The pity system only triggers after frequent low-rarity upgrade distributions, subsequently increasing the probability of higher-tier upgrades, therefore, if the rounds are short the player never received sufficient "low-tier" loot to stimulate the algorhythm enough and ,thereby, to increase their chance of receiving better loot. Furthermore, the pity system's memory does not persist across runs; the counter resets each playthrough, which may contribute to perceptions of unfairness and frustration.

Conversely, group "X" responded more favorably to fairness measures. This discrepancy raises the question of whether differential outcomes resulted from chance variation. The more likely explanation is that experienced players possessed greater familiarity with randomness mechanics in game design, enabling them to evaluate fairness from a more informed, conceptual perspective rather than solely from immediate gameplay outcomes.

However, the data indicate that for a broader audience, the current implementation of fairness mechanisms may not meet acceptable standards.

Frustration

The questionnaire results indicate that frustration levels for group "InX" exceeded acceptable thresholds. This outcome may correlate with the fairness ratings and

share similar underlying causes. Questions C16 and C17 suggest that frustration resulted from a combination of challenging input randomness and insufficient pity system activation in scenarios where mitigation would have been beneficial. The pity system's limitations are detailed in the preceding section.

Technical issues also contributed to lower fairness ratings and higher frustration indices. One bug caused enemies to become trapped within floor geometry, preventing players from eliminating them and progressing to subsequent rooms, as complete enemy elimination was required for advancement. To address this unexpected issue, a temporary override button was implemented to eliminate all enemies simultaneously. While this solution undermines the intended challenge, it remains exclusively in the PoC prototype and will be removed following comprehensive bug fixes.

Overall, the data indicates that the PoC's difficulty level may have been excessive for inexperienced players (group "InX"), a factor that requires consideration in subsequent development iterations.

8.1.2 Replayability

Replayability represents another metric that fell below the established threshold. Contrary to the fairness results, group "X" recorded a mean index below 70, while group "InX" exceeded this threshold. Participant feedback from group "X" indicated that the map generation did not achieve the anticipated level of run-to-run uniqueness, suggesting the need for further investigation. This outcome likely resulted from limited enemy variety and an insufficient number of distinct rooms (64 total). These limitations were sufficiently subtle that they were primarily detected by the more experienced player group ("X").

One notable anomaly emerged within this criterion: a playtester who completed twelve runs reported that room uniqueness diminished during the final three runs. This phenomenon likely stems from the constraints of modular procedural generation combined with the limited room pool. After approximately nine runs, the generated room sequences, while technically unique in arrangement, began to exhibit perceptible patterns and repetition. Despite this limitation, the overall performance regarding run-to-run uniqueness remained largely successful.

8.1.3 Agency

Conversely, the PoC demonstrated strong performance in player agency metrics. Agency, in game design terminology, refers to the relationship between player decision-making and control relative to chance-based outcomes. This construct was measured through questions C4–C6 and C24.

The favorable results in this criterion likely stem from successful implementation of the core design principle: input randomness preserves agency while output randomness undermines it. Consequently, the prototype achieved a randomness-based gameplay experience wherein players perceived their success as primarily attributable to strategic choices rather than luck—a perception that extended to failure states as well.

Furthermore, question C5 ("I could plan ahead because surprises came before my choices") validates the theoretical framework regarding the information horizon and input randomness. When the information horizon is effectively implemented—for instance, through modular procedural map generation—it introduces challenge while minimizing frustration, thereby enabling players to engage in forward planning despite the presence of randomized elements.

8.1.4 Excitement

The excitement metric demonstrated that input randomness successfully generated positive tension and drama when properly balanced with decision timing. Although the prototype met the established threshold for this criterion, two responses from group "InX" assigned a rating of 5 to the reverse-phrased question C9, which assessed whether random events diminished player interest. These outlier ratings (5 out of 7) likely indicate that these participants encountered poorly timed randomness events multiple times during their playthroughs, leading to accumulated irritation when subsequent input-randomness-based events occurred. This repetition may have fostered negative associations with randomness mechanics.

While the literature suggests that input randomness does not inherently generate frustration, these specific cases represent exceptions to this general principle, demonstrating that implementation timing and frequency remain critical factors in player reception.

8.2 Implementation-Specific Checks

Analysis of the Implementation-Specific Checks (Part D of the questionnaire) reveals notable findings regarding the pity system. Data from both groups "InX" and "X" indicate that the pity system activated for the majority of participants, with only a minority unable to observe its effects. Participants who selected "Not sure" frequently provided comments indicating uncertainty while noting perceived improvements in loot quality from enemies and random stat-based upgrades following periods of weak rewards. These observations align with the intended pity system functionality; therefore, accounting for the likelihood that a substantial portion of "Not sure" responses would align with "Yes" if certainty were higher, the data suggest successful pity system implementation in the PoC.

The pity system in the current PoC implementation did not affect enemy difficulty scaling. Had such scaling been implemented, frustration levels would likely have decreased while fairness perceptions increased.

Questions D2 and D3 collectively yielded an index of 100, confirming highly successful implementation of random stat-based upgrades. This mechanic also contributed positively to the overall fairness index, suggesting that fairness ratings would have been lower without it. As a source of input randomness, this result supports the theoretical framework that input randomness is perceived favorably because it preserves player agency—at least in the context of random stat-based upgrades.

Question D7 ("It was clear when luck would matter") received indices exceeding 70 in both test groups, indicating successful implementation. This suggests that transparency was achieved without compromising excitement (consistent with previously discussed results). However, the data remain insufficient to confirm that randomness transparency reduces perceived unfairness. The wording of D7 was too objective to support such an inference: it assesses whether players could identify when luck would influence outcomes, not whether this transparency enhanced perceived fairness. Additionally, transparency in the PoC primarily applied to loot and upgrades, which may have constrained the index—an anticipated trade-off, as excessive transparency can diminish the surprise elements that contribute to player enjoyment. The findings suggest that transparency alone may not substantially reduce perceived unfairness unless combined with addressing the underlying issues identified in the fairness and frustration analysis.

Questions D4 and D5 achieved marginally passing results, with a notable decline in D5 ("No room felt out of place or unfair"). In rare instances affecting two participants,

the map generation produced predominantly linear configurations, reducing the utility of vertical navigation mechanics. Follow-up inquiry clarified that these participants did not perceive rooms as contextually inappropriate, but rather that the spatial arrangement diminished the functional value of verticality.

These responses correlated with lower fairness ratings, and the affected participants also reported slightly reduced excitement levels compared to other respondents.

9 Conclusion

9.1 Summary of the results

The Proof of Concept achieved mixed results across key design metrics. While it successfully validated input randomness as a viable approach for preserving player agency and generating excitement, significant challenges emerged in fairness and frustration management, particularly for inexperienced players who fell below the 70-point threshold with a fairness index of 67.2. The primary issues included insufficient pity system activation due to short runs and its failure to persist across playthroughs, excessive difficulty for novice players, and technical bugs that hindered progression. Replayability suffered among experienced players due to limited enemy variety and only 64 total rooms, causing perceptible repetition after approximately nine runs. Conversely, the prototype excelled in player agency metrics across both groups, with players attributing outcomes to strategic choices rather than luck, and random stat-based upgrades achieved a perfect implementation score of 100. Transparency and excitement metrics met established thresholds, though isolated cases revealed that poorly timed random events can create negative associations. Overall, while the core design principle proved sound, improvements are needed in pity system mechanics, difficulty balancing, and content variety to meet standards for a broader audience.

9.2 Positives

Throughout the development of our Proof of Concept, we acquired a broad range of new skills and techniques, particularly in the areas of 3D modeling and asset management. Working extensively with Blender allowed us to gain valuable hands-on experience in creating, structuring, and optimizing assets for game engines. One of the major advantages we encountered was Blender's popularity and strong

community ecosystem — the abundance of tutorials, forums, and ready-made tools accelerated our learning process significantly and helped us overcome many early technical hurdles. Additionally, we gained a solid understanding of the workflow between Blender and Unity, which is highly relevant for modern game development and will serve as a crucial foundation for future projects. This combination of creative, technical, and problem-solving progress made the project both educational and rewarding.

Looking further into the programming part, we also learned how to solve or avoid certain common problems in Unity and found some new or improved methods of implementing features. A specific example of this is that we learned to use Unity's navmesh pathfinding system, allowing us to implement it faster in future projects.

9.3 Negatives

However, the project also presented several challenges and limitations that tested our adaptability. A major issue arose from our early assumption that all Blender features—such as simulation nodes for random effects—would export directly to Unity, which proved incorrect and required the creation of various workarounds. Since none of the team members had prior experience with Blender, the initial learning curve was steep and time-consuming, especially when dealing with complex modeling and export workflows. Exporting models and materials into Unity turned out to be particularly exhausting, as textures, shaders, and lighting often behaved inconsistently between the two programs. Another major challenge came from improper prefab setup during early stages: since many prefabs were built before proper models existed, implementing later updates required significant rework. Furthermore, animation proved to be one of the most difficult aspects of the process, demanding careful planning, iterative practice, and fine-tuning to achieve acceptable results. Finally, a considerable amount of trial and error was unavoidable due to unexpected behavior during export and setup, which often slowed progress and tested our patience but also provided valuable lessons for future development cycles.

While we know how to solve the problems we encountered in this project, saving us from having to solve them again, we still had to solve them for the first time during this project. While this is of course always the case, the problem in our case was that none of us have ever really worked on a larger, and especially, 3-dimensional game.

Another problem we encountered is the fact that we made a game in which almost all features depend heavily on one another and the game is not really playable, as we intend it to, until it is almost complete. Because of this we had a lot less time to test the project than we would have liked to, and it was really stressful to fix the great amount of bugs that were only discovered once we were able to properly test the game. This did also not only apply to programming but also to the entire project, more specifically that we did not plan the programming and modeling together. This means that, while the timeline of both the modeling and the programming would have worked just fine one their own, the fact the we first had to learn modeling from scratch created a bottleneck for the programming. Meaning that we were able to complete the 3D models just like planned, we only had the first finished models toward the later half of the project. Hence, the plan for the programming that would have worked fine if we already had the models, fell apart, because, as we realized later on, in order to playtest the game we needed the rooms to be completed which, in turn, required the 3D models to be finished.

9.4 Outlook for the Future

First and foremost, our primary focus for future development will be bug fixing. Although the Proof of Concept (PoC) successfully demonstrated our core idea of randomness, several technical inconsistencies and gameplay issues remain that must be resolved before the game can reach a stable and fully playable state.

Following the initial playtesting phase and based on the valuable feedback we received from the participants, we plan to implement the suggested improvements to enhance gameplay balance, clarity, and engagement. Furthermore, conducting a second round of playtesting will be beneficial. While our first test aimed to compare “inexperienced” and “experienced” player experiences, we noticed that we only vaguely defined what the criteria of "InX" and "X" was, therefore, not distinguishing them perfectly, as we made them choose to which group they belonged to, depending on what games they have played prior. Furthermore, to really distinguish between "X" and "InX", we must come with a better method, that would test the playtester’s skill particularly in randomness. A new test group with more diverse gaming backgrounds might also provide a clearer insight into the learning curve and accessibility of our mechanics.

We also aim to polish our in-game assets and environments, which were intentionally simplified for this PoC. Since our main goal was to explore the concept of randomness,

visuals was secondary. However, moving toward a more complete game requires us to go beyond this conceptual limitation and build a more immersive, visually coherent world. To achieve this, we plan to research better workflows for handling non-exportable Blender features or replicate their effects directly within Unity, ensuring that our asset pipeline becomes more stable and predictable. Additionally, we intend to plan asset creation and prefab setup more carefully in upcoming iterations to minimize unnecessary rework and streamline the implementation of new content.

A key aspect of this visual improvement phase will be increasing our familiarity with Blender–Unity integration, which will help us better manage transitions between modeling, animation, and engine implementation. We also plan to experiment with advanced Blender tools and Unity shaders to achieve a more polished and dynamic look while maintaining efficiency. These enhancements would contribute significantly to both visual appeal and performance optimization. Moreover, we aim to add more animations — not only to characters but also to environmental elements — to make the overall experience more dynamic and visually engaging.

In addition, expanding the variety of rooms, enemies, and builds remains a key step to improving replayability and freshness between playthroughs. The long-term goal is to achieve full procedural generation, allowing every run to feel unique. Eventually, we would like to explore AI-driven procedural generation, further enhancing unpredictability and dynamic content creation — though we acknowledge that this approach introduces high complexity and potential for bugs, especially given our current level of development experience.

Finally, we intend to incorporate a storyline into the game, shifting away from the current endless high-score mode. A narrative layer could help offset the potential frustration caused by unpredictable randomness by providing emotional context and motivation for the player.

9.4.1 Why a Greek Theme?

The Greek mythological theme aligns with the core design philosophy centered on randomness, fate, and challenge. Ancient Greek mythology explores the tension between human agency and divine randomness, embodied in the concept of fate (Moira), where even heroes and gods remain subject to unpredictable forces. This thematic framework parallels the gameplay mechanics, wherein players navigate procedurally generated rooms, encounter unpredictable stat-based upgrades, and respond to random environmental effects such as variable gravity.

The concept of Khaos (Chaos)—the primordial void from which all creation emerged in Greek cosmology—provides a thematic foundation for the randomness systems. Khaos represents the ultimate state of unpredictability and formlessness, the raw potential from which order continuously emerges and transforms. This directly corresponds to the procedural generation approach: each playthrough emerges from a state of possibility, with rooms, upgrades, and challenges materializing through randomized selection. The shifting gravity mechanics can be interpreted as representing a world still influenced by primordial instability, not yet fully stabilized into fixed natural laws.

Furthermore, the mythological setting provides a narrative framework for abstract concepts such as chaos, order, and transformation, all of which relate directly to the randomness systems. The chaotic influence attributed to deities such as Zeus or Poseidon extends the presence of Khaos, while modular procedural rooms evoke the labyrinthine spaces of Greek myth (e.g., Daedalus' labyrinth)—environments that resist fixed form and predictable structure.

From a design perspective, the Greek aesthetic offers both artistic cohesion and implementation flexibility. The setting supports visually distinctive environments (temples, underworlds, floating islands) that adapt readily to modular and procedural generation. It also provides symbolic depth to gameplay elements: random stat-based upgrades can be represented as divine blessings or curses, while each playthrough functions as a mythic trial of endurance and adaptability—a struggle to impose order upon Khaos itself.

References

[1] Armello Wiki contributors: Armello Wiki. https://armello.fandom.com/wiki/Armello_Wiki, last accessed on: 04.10.2025.

[2] DeFusco, Daniel: The Science of Surprise: Using Probability to Create Engaging Video Games. https://www.yu.edu/news/katz/science-surprise-using-probability-create-engaging-video-games, last accessed on: 05.11.2025.

[3] Deterding, Sebastian u. a.: Mastering Uncertainty: A Predictive Processing Account of Enjoying Uncertain Success in Video Game Play. https://www.frontiersin.org/journals/psychology/articles/10.3389/fpsyg.2022.924953/pdf, last accessed on: 20.05.2025.

[4] Fort, Thomas: Controlling Randomness: Using Procedural Generation to Influence Player Uncertainty in Video Games. https://stars.library.ucf.edu/cgi/viewcontent.cgi?article=2706&context=honorstheses1990-2015, last accessed on: 03.04.2025.

[5] Game Maker's Toolkit: The Two Types of Random in Game Design. https://www.youtube.com/watch?v=dwI5b-wRLic, last accessed on: 24.03.2025.

[6] Guseva, Maria u. a.: Instruction Effects on Randomness in Sequence Generation. https://www.frontiersin.org/journals/psychology/articles/10.3389/fpsyg.2023.1113654/pdf, last accessed on: 28.02.2024.

[7] Into the Breach Wiki contributors: Into the Breach Wiki. https://intothebreach.fandom.com/wiki/Into_The_Breach_Wiki, last accessed on: 05.10.2025.

[8] Ludology Podcast: GameTek Classic 183 – Input Output Randomness. https://ludology.libsyn.com/gametek-classic-183-input-output-randomness, last accessed on: 10.04.2025.

[9] Molkara, Farzam: Neuropsychology of Using Randomness in Game Design, Playtesting (and Beyond)! https://tinyurl.com/yj48896d, last accessed on: 31.05.2025.

[10] Nielsen, Rune Kristian Lundedal; Grabarczyk, Paweł: Are Loot Boxes Gambling? https://todigra.org/index.php/todigra/article/view/1774/1774, last accessed on: 01.05.2025.

[11] Staff, Game Developer: Roll for Your Life: Making Randomness Transparent in Tharsis. https://www.gamedeveloper.com/design/roll-for-your-life-making-randomness-transparent-in-i-tharsis-i-, last accessed on: 31.03.2025.

[12] Wikipedia contributors: Into the Breach — Wikipedia. https://en.wikipedia.org/wiki/Into_the_Breach, last accessed on: 03.09.2025.

[13] Wikipedia contributors: Spelunky — Wikipedia. https://en.wikipedia.org/wiki/Spelunky, last accessed on: 02.10.2025.

[14] Wikipedia contributors: XCOM — Wikipedia. https://en.wikipedia.org/wiki/XCOM, last accessed on: 03.08.2025.

[15] XCOM Wiki contributors: XCOM Wiki. https://xcom.fandom.com/wiki/XCOM_Wiki, last accessed on: 03.08.2025.

[16] Zhang, Yilei u. a.: Effect of Input-Output Randomness on Gameplay Satisfaction in Collectable Card Games. https://arxiv.org/pdf/2107.08437, last accessed on: 02.04.2025.

[17] Wikipedia contributors: Slay the Spire — Wikipedia. https://en.wikipedia.org/wiki/Slay_the_Spire, last accessed on: 05.11.2025.

[18] Wikipedia contributors: Hades (video game) — Wikipedia. https://en.wikipedia.org/wiki/Hades_(video_game), last accessed on: 05.11.2025.

[19] Wikipedia contributors: Dead Cells — Wikipedia. https://en.wikipedia.org/wiki/Dead_Cells, last accessed on: 05.11.2025.

[20] Wikipedia contributors: Minecraft — Wikipedia. https://en.wikipedia.org/wiki/Minecraft, last accessed on: 05.11.2025.

[21] Wikipedia contributors: The Binding of Isaac — Wikipedia. https://en.wikipedia.org/wiki/The_Binding_of_Isaac, last accessed on: 05.11.2025.

[22] Wikipedia contributors: Borderlands (video game) — Wikipedia. https://en.wikipedia.org/wiki/Borderlands_(video_game), last accessed on: 05.11.2025.

[23] Wikipedia contributors: Mario Kart — Wikipedia. https://en.wikipedia.org/wiki/Mario_Kart, last accessed on: 05.11.2025.

[24] Wikipedia contributors: Middle-earth: Shadow of Mordor — Wikipedia. https://en.wikipedia.org/wiki/Middle-earth:_Shadow_of_Mordor, last accessed on: 05.11.2025.

[25] Z-Man Games: Pandemic — Official Product/Rules. https://zmangames.com/en/products/pandemic/, last accessed on: 05.11.2025.

[26] TetrisWiki contributors: Random Generator (7-bag). https://tetris.wiki/Random_Generator, last accessed on: 05.11.2025.

[27] Hearthstone Wiki contributors: Pity timer. https://hearthstone.fandom.com/wiki/Pity_timer, last accessed on: 05.11.2025.

[28] Wikipedia contributors: Apex Legends — Wikipedia. https://en.wikipedia.org/wiki/Apex_Legends, last accessed on: 05.11.2025.

[29] Wikipedia contributors: Overwatch — Wikipedia. https://en.wikipedia.org/wiki/Overwatch, last accessed on: 05.11.2025.

[30] Wikipedia contributors: FIFA Ultimate Team — Wikipedia. https://en.wikipedia.org/wiki/FIFA_Ultimate_Team, last accessed on: 05.11.2025.

[31] Wikipedia contributors: Counter-Strike: Global Offensive — Wikipedia. https://en.wikipedia.org/wiki/Counter-Strike:_Global_Offensive, last accessed on: 05.11.2025.

[32] Serenes Forest: True Hit (2 RN) in Fire Emblem. https://serenesforest.net/general/true-hit/, last accessed on: 05.11.2025.

[33] Blizzard Entertainment: Patch 2.0.1 – Loot 2.0 and Smart Loot. https://us.diablo3.blizzard.com/en-us/blog/13141558/patch-201-2014-2-25, last accessed on: 05.11.2025.

[34] Blender Guru: Beginner Blender Tutorial (Donut Part 1). 16.11.2023. https://www.youtube.com/watch?v=B0J27sf9N1Y, last accessed on: 06.11.2025.

[35] Jelle Vermandere: How To Make A 3D Character For Your Game (Blender to Unity). 02.08.2020. https://www.youtube.com/watch?v=ogz-3r0EHKM, last accessed on: 06.11.2025.

[36] Blender Guru: Beginner Blender 4.0 Tutorial – Part 5: Shading. 20.11.2023. https://www.youtube.com/watch?v=fsLO1F5x7yM, last accessed on: 06.11.2025.

List of Figures