“Dungeon Forge”

Dungeon Generator Extension for Godot

Jevgenij Ivanov
20748055

Final Year Project 2024

B.Sc. Computer Science and Software Engineering

Department of Computer Science

Maynooth University

Maynooth, Co. Kildare

Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Computer Science and Software Engineering.

Supervisor: Ralf Bierig

Table of Contents

Declaration

I hereby certify that this material, which I now submit for assessment as part of CS440 Final Year Project module, is entirely my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

I hereby acknowledge and accept that this thesis may be distributed to future final year students, as an example of the standard expected of final year projects.

Signed: Jevgenij Ivanov Date: 15/03/2024

Acknowledgements

I would like to thank Ralf Bierig for giving the opportunity to work on such an interesting project, for all the support throughout the development, and for providing many suggestions and feedbacks.

Abstract

This thesis introduces "Dungeon Forge", a plugin for the Godot Game Engine 4.2, aimed at simplifying the creation and management of dungeon environments for game developers. Leveraging procedural generation algorithms like Delaunay Triangulation and Minimum Spanning Tree, the plugin facilitates the efficient generation of unique dungeon layouts with various rooms, doors, and tunnels. Designed with a user-friendly interface, the plugin comes with a selection of pre-designed textures as part of the project, enriching visual appeal and thrill of created environments. Motivated by a passion for game development and the potential to contribute valuable tools to the Godot community, this project showcases the application of advanced programming skills and 3D modelling techniques. By addressing the technical challenge of automating dungeon design to produce engaging and optimized gameplay environments, "Dungeon Forge" not only streamlines the development process but also opens up new possibilities for creative exploration, making it a great addition to the resources available for developers.

List of Figures

Chapter 1: Introduction

1.1 Topic

This thesis introduces the development of a “Dungeon Forge”, a plugin extension for Godot Game Engine 4.2, designed to help game developers quickly end efficiently create and manage dungeon-like environments. With this tool, users can generate varied dungeon layouts that include rooms, doors, and tunnels. Each part of the layout is marked with coloured meshes to identify layout structures generated. One of the features of this plugin is the ability to save favourite layouts. Users can store their best designs within the plugin and then bring them into other game scenes whenever needed. This saves time and lets developers reuse successful designs without starting from scratch. Other features of the plugin include a tab for spawning various player controllers for the scene to inspect generated content in gameplay environment. A preview tab is also included in the extension, which is used for watching display of your scene from within the plugin. All these features are designed to be easily accessible from within a UI window in the bottom right part of the Godot editor.

1.2 Motivation

The motivation behind this project is rooted deeply in a passion for game development and a fascination with game engines. As the gaming industry continues to evolve, the demand for tools that improve the development process is bigger than ever. This project is a stepping stone towards creating an interesting and useful extension that contributes to the Godot community, and other developers around the world. It is a unique opportunity to delve into the complexities of game engine architecture, understand the nuances of procedural generation, and tackle the challenges of user-friendly design.

During the research phase, a lot of fascinating designs and algorithms have been explored. The concept of procedural dungeon generators seemed quite interesting, the potential for automating complex design tasks was the inspiration for this projects development. One of the blog posts discovered [1] was using a technique called Delaunay triangulation, which offers a methodical approach to generating connected room layouts, thereby enhancing the appeal of dungeon environments. The “Dungeon Forge” project builds upon such foundations, while also creating an interactive and visually appealing UI that serves as an extension to the Godot editor.

Delving into the realm of room and dungeon generation revealed a fascinating use of algorithms like the Minimum Spanning Tree (MST) and Prim’s (also known as Jarník's algorithm), which have creative applications in these kind of systems. These algorithms were already studied in academic coursework for "CS210 Algorithms & Data Structures”, so it was a great opportunity to apply the knowledge to practice. Further investigation revealed that using MST as a subset of the Delaunay triangulation can increase the efficiency of computing operations, a concept detailed in Chapter 5 of Elisa Zancolo's scholarly research [2]. This effective combination has the flexibility to be used in dungeon generation using any of the programming languages and game engines. Discoveries like these have sparked a heightened interest in advancing development in this area.

Furthermore, this initiative is driven by the desire to contribute to the game development community, particularly those interested in creating immersive dungeon-like environments for their Godot projects. By sharing this tool, the aim is to empower fellow developers with the means to bring their creative visions to life more efficiently, thus saving valuable time and resources. The Godot engine

has a strong and active community on platforms like Discord, where people share their work, discuss all things technology, stream their development and much more. Such a community can become a vibrant ecosystem for sharing ideas, providing feedback, and collaborating on projects, thereby enriching the overall experience and knowledge base of all the members involved. Ultimately, the "Dungeon Forge" extension is envisioned not just as a tool, but as a catalyst for learning, growth, and community building among game developers who share a common interest in dungeon environment creation and other aspects of game design.

1.3 Problem Statement

The technical challenge at the core of the "Dungeon Forge" project for Godot Engine 4.2 lies in developing an automated system capable of procedural dungeon generation with high usability and flexibility. Traditional methods of dungeon design often involve manual placement of elements, which could be time-consuming and limits creative exploration due to the repetitive nature of the task. The specific technical problem addressed by this project is the creation a program that uses multiple algorithms that work together to create varied dungeon layouts - with interconnected rooms, corridors, and placements of doors. These algorithms must not only ensure spatial coherence and navigability but also allow for customization to cater to different game design and development needs.

The problem, as defined in academic and practical contexts, involves automating the spatial arrangement of dungeon elements within a given boundary to create environments that are both engaging for players and feasible for developers to produce. The system that was created could then also be implemented into video games and be used during gameplay and runtime. Procedural dungeons could be quickly generated when loading into a zone of a game, as an example. This capability significantly elevates the project's utility by enabling game developers to incorporate procedurally generated dungeons directly into their games, thereby offering players a unique experience with each playthrough. Implementing such a feature requires the algorithm to not only be efficient and versatile in design generation but also optimized for performance to ensure smooth gameplay without noticeable delays. The complexity of this task is magnified by the requirement for these environments to be dynamically generated, offering a unique experience within each game session

1.4 Approach

The journey began with diving deep into the Godot engine, learning it from the ground up, and getting to grips with its scripting language, GDScript. This step into new territory was made smoother by a solid background in programming, thanks to university courses like “CS161 Introduction to Computer Science” using Java, ”CS230 Web Information Processing” with JavaScript, and “CS264 Software Design” with C#. These courses provided a sturdy base in coding principles, making the transition to mastering GDScript more efficient. This educational background proved crucial in effectively tackling the project's technical challenges and applying the logic of procedural generation with accuracy.

It was necessary for the project to expand into exploring Blender 3D, a robust platform for crafting and texturing 3D models. This expertise was pivotal in shaping the dungeon's components, such as doors, walls, and floors. The project benefitted significantly from skills in Shading and UV mapping, learned through the “CS426 Computer Graphics” module. These techniques were applied to develop textures that not only boosted the visuals of the dungeons but also deepened the gaming experience, introducing a sense of depth and immersion to the environments created by the plugin. The knowledge

gained from the Computer Graphics course enabled the creation of simple but effective textures for the dungeon generator, utilizing UV mapping techniques to achieve a higher level of detail and authenticity.
During the exploration of potential algorithms suitable for the dungeon generator, attention was drawn to Delaunay Triangulation and Minimum Spanning Tree algorithms, known for their effectiveness in procedural dungeon generation. Delaunay Triangulation was an ideal candidate to be used for creating tunnels and connections between rooms and dungeons, as mentioned on various forums such as “stackoverflow.com” and “stackexchange.com” [3]. By default, Delaunay method would have too many connections, when the goal is to create interesting dungeon layouts. This is where the “Minimum Spanning Tree” algorithm had to be carefully studied in terms of generating rooms, and how the algorithm can be implemented into the project using code [4].

These algorithms stood out for their ability to create interconnected and realistic dungeon layouts, offering a balance between randomness and structure essential for engaging gameplay. Delving into these algorithms provided insights into crafting dungeons that not only challenge but also intrigue players, by ensuring each dungeon layout is unique and immersive. The decision to incorporate these algorithms marked a significant step in the development process, aiming to enhance the versatility and functionality of the dungeon generator.

1.4.1 Installation and First Steps

Building on the groundwork of mastering the Godot engine and delving into procedural generation algorithms, it's essential to guide first-time users through the process of installing and setting up the dungeon generator plugin. This process is both straightforward and user-friendly, designed to integrate seamlessly into the existing Godot environment. Users begin by adding the plugin folder to the "addons" folder located within their project directory, a standard practice for extending Godot's functionality with custom tools and modules. Once the folder is correctly placed, the plugin needs to be activated through the Godot project settings.

Upon successful activation, the user is greeted with a new UI window, positioned in the bottom right corner of the Godot editor. This window serves as the central hub for interacting with the extension, offering a selection of options. The layout generator is then spawned through this hub, and after selecting the generator node, users can embark on the creative process, selecting and customizing the parameters that will shape their dungeon's layout and characteristics. This straightforward approach to starting with the plugin makes the initial setup quick and easy, inviting users to jump right into creating dungeons without any hassle. It lays the groundwork for an enjoyable and efficient development journey.

Chapter 2: Technical Background

2.1 Topic Material

In the realm of game development, procedural content generation (PCG) techniques have been instrumental in crafting dynamic game environments, with dungeon generation standing as a notable application. “Diablo”, released in 1996 by Blizzard Entertainment, marked a significant milestone in popularizing procedural dungeon generation within the gaming industry [5]. Its sophisticated use of algorithms to create an ever-changing labyrinth of dungeons deeply influenced how games could

leverage procedural generation to enhance replayability and maintain player engagement. Diablo's success demonstrated the commercial viability and appeal of such techniques to a broad audience, setting a precedent for countless games to follow. Although Diablo made procedural generation popular in 1996, it wasn't the first game to use it. That credit goes to "Rogue," a game from 1980. Rogue used this technique to make each game different, introducing a style of game where dungeons change each time you play, and if you lose, you start over [6]. This idea has influenced many games since.

On the indie side of the spectrum, the game "The Binding of Isaac" by Edmund McMillen and Florian Himsl employs procedural generation to create its dungeon layouts, items, and enemy placements. This approach ensures that no two play sessions are alike, providing a continuously fresh challenge that has contributed to the game's enduring popularity [7]. The Binding of Isaac's use of PCG demonstrates the technique's scalability, from major studio productions to smaller indie projects, showcasing its versatility in enhancing game design regardless of budget size.

Like Diablo and The Binding of Isaac, the "Dungeon Forge" project uses procedural generation to make gaming more engaging. But, unlike these games, it gives developers using the Godot Engine the tools to easily create and modify dungeon layouts. This extension not only simplifies setting up dungeons within the editor but also introduces the ability to save and reuse these layouts, aiding the game development process. It's designed to generate dungeons that can potentially enhance gameplay directly, offering versatility beyond initial setup. Additionally, this Godot extension includes a feature to spawn playable characters into the scene, allowing developers to test and interact with the generated environments immediately.

In the course of researching procedural dungeon generators within the Godot ecosystem, it became evident that the community has developed a variety of tools and extensions similar to what the “Dungeon Forge” project is looking to achieve. Among the popular solutions discovered was the “Cyclops Level Builder”, highlighted in YouTube tutorials and showcases, demonstrating the practical applications of these techniques in game development and dynamic level creation. Despite the wealth of practical examples and tutorials, a significant gap was identified in the form of academic writings or scholarly research specifically addressing these Godot-based procedural generation tools.

Even though there aren't many academic papers or research about Godot's tools for procedural generation, there's a wealth of knowledge available through the engine's documentation, guides, tutorials, and community events. This points to a modern way of learning and studying, where the Godot community plays a crucial role. Through forums, social media, and contributions to Godot's library of add-ons, developers share their knowledge and innovations. This approach highlights how learning and developing in Godot leans heavily on community support and collaborative problem-solving, pushing forward advancements without relying solely on traditional academic research.

2.2 Technical Material

The “Dungeon Forge” extension has different features, but the core of this project is the actual generator which relies on two key algorithms: Delaunay Triangulation and Minimum Spanning Tree. These algorithms are quite powerful and have their unique roles. Delaunay Triangulation, in particular, is adept at connecting points to ensure no point lies within any triangle's circumference, ideal for crafting dungeons with interlinked rooms and corridors. An insightful theorem applied in this context shows that for a set of n points, where k are on the convex hull (the outer boundary), Delaunay Triangulation results in exactly $2n-k-2$ triangles and $3n-k-3$ edges [8]. This theorem gives us

formulas to precisely predict the outcomes of triangulation, ensuring that the connections between points form triangles without any point lying inside another triangle's area. This aspect of Delaunay Triangulation is critical for generating interconnected, non-overlapping spaces in applications like dungeon layouts.

The Minimum Spanning Tree (MST) algorithm was selected for its effectiveness in ensuring all dungeon rooms and pathways are correctly connected, eliminating any redundant routes. This decision was influenced by foundational knowledge acquired in the “CS210 Algorithms & Data Structures” module, where MST, alongside other algorithms such as Dijkstra's for shortest paths and Prim's for minimum spanning connections, was thoroughly studied. This educational background provided a good understanding of these types of algorithms, and gave confidence to implement them into the project. Unlike algorithms that may prioritize shortest paths between two points, MST focuses on connecting all points (or rooms) in a way that the overall path length is minimized without creating loops.

Both algorithms have their complexities and challenges, especially when it comes to integrating them into a project like this. In the development process, modifications and optimizations were necessary to tailor these algorithms to the specific requirements of dungeon generation in the Godot Engine environment. This meant tweaking the algorithms to run efficiently within the engine and ensuring that they could generate interesting and varied dungeon layouts.

During the development, the insights from university modules such as "CS426 Computer Graphics" and "CS423 Designing for Virtual Environments" proved to be invaluable. Specifically, "CS426 Computer Graphics" facilitated a lesson for techniques of UV mapping textures [Fig 1] and implementing shading [Fig 2], skills that are essential for adding depth and realism to the dungeon environments created by the extension. Meanwhile, "CS423 Designing for Virtual Environments" offered guidance on crafting basic 3D models (doors and floor in this case) and organizing the project's hierarchy efficiently. The two modules coincided perfectly with the project's development, allowing immediate application of new skills to enhance visual and structural design. This blend of education and practical work highlighted the value of integrating academic learning with project development.

Chapter 3: The Problem

3.1 Technical Problem

The realm of game development is continuously evolving, with game engines becoming more sophisticated and the accompanying tools more powerful. Despite these advancements, the creation of immersive environments and intricate level design remains a formidable challenge. The meticulous process of manually constructing textures to shape a single room, aligning each element akin to assembling a complex structure, can be exceedingly time-consuming and labour-intensive. This painstaking attention to detail, while crucial for building engaging virtual spaces, often demands a significant investment of time and resources.

To mitigate this, modern development tools aim to streamline the workflow, enabling tasks once spanning weeks to be completed with a mere button press. These solutions vary widely, from AI-powered assistance to editor extensions that enhance the engine's native capabilities. The primary goal of this project is to equip developers with the power to effortlessly generate dungeon-like layouts and

environments and managing them, minimizing the manual burden traditionally associated with such tasks.

This extension aspires to be a natural extension of the editor itself, blending functionality with an intuitive and user-friendly interface. It must feel like an integral part of the game development environment, where the functionality aligns with the developer's needs, and the UI facilitates rather than hinders the creative process. The design should be visually appealing and self-explanatory, ensuring users spend less time grappling with the interface and more time actualizing their vision.

3.2 Problem Analysis

In analysing the problem of creating dungeon-like environments within a game development context, several key factors contribute to the complexity of the task. First, the design of a dungeon requires a careful balance between aesthetic appeal and functional design to ensure that the environment is not only visually engaging but also conducive to gameplay. It must feature a variety of spaces, from tight corridors that build tension to larger chambers that serve as arenas for combat or puzzles.

The time investment for crafting such spaces is considerable, as traditional methods necessitate meticulous placement of each element. This extends the development cycle and can introduce delays, particularly for smaller teams or individual creators. Moreover, the repetitive nature of this task can lead to a drain on creative resources, as developers might find themselves mired in the minutiae of environment design rather than focusing on broader narrative and gameplay mechanics.

Another aspect of the problem is the challenge of ensuring uniqueness and replayability in the dungeon layouts. Procedural generation offers a solution, but without careful consideration, it can result in environments that feel disjointed or monotonous. Therefore, a nuanced approach to procedural algorithms is essential to create spaces that feel intentional and curated.

The user interface of the tool that facilitates this creation process is another critical factor. A poorly designed interface can significantly hinder the efficiency of the tool, no matter how powerful the underlying technology might be. An optimal interface must not only be intuitive, allowing users to quickly grasp and utilize its functions but also flexible, accommodating a wide range of design aesthetics and user preferences.

The class diagram [Fig 3] below provides a visual breakdown of the various components that interact within the plugin, offering insight into how these challenges might be navigated. It shows the potential for an integrated solution that addresses these needs holistically, without committing to a specific decision on implementation. Through this analysis, it becomes evident that the ideal solution would streamline the environment creation process, foster creative flexibility, and enhance the overall efficiency of dungeon design within the Godot Engine.

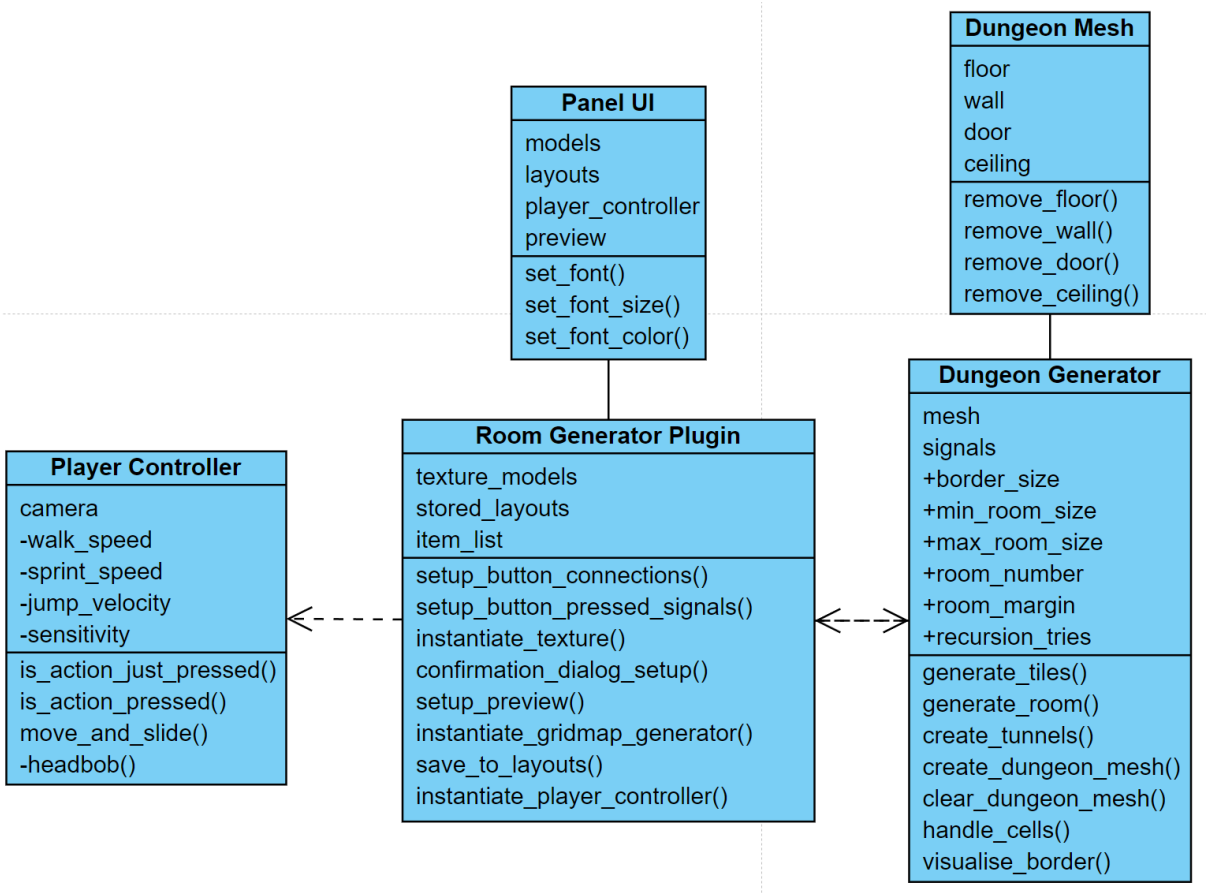
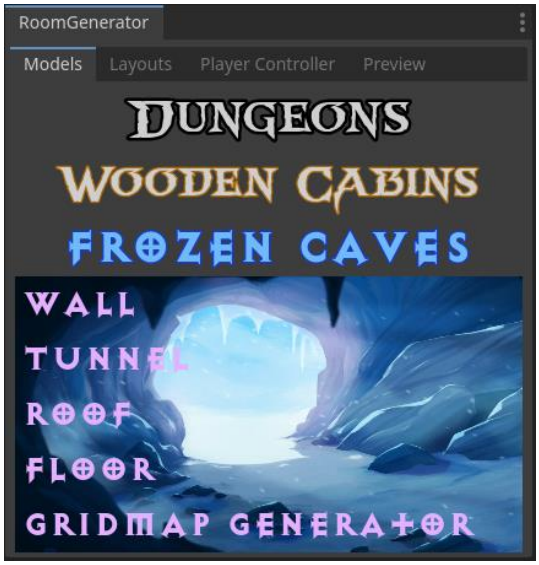


Fig 3. Class Diagram of the “Dungeon Forge” Godot extension.

Chapter 4: The Solution

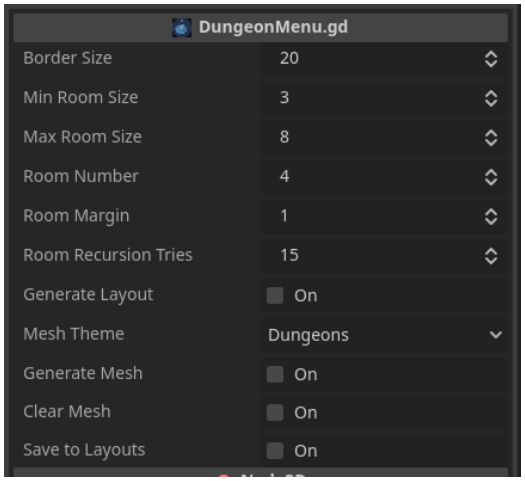
4.1 The Plugin UI (Extension Hub)

One of the most important features of the project is a visually appealing user interface that is located in the bottom right corner of the Godot editor. This UI was designed to contain all the necessary options for spawning various mesh, storing and managing saved layouts, use player controller for runtime testing and more.



The figure on the left shows the design of the plugin UI. The models tab contains submenus for three different dungeon themes, and all of the textures for these themes are available there. The dungeon generator is also spawned through this submenu using the “Gridmap generator” button. The layouts tab is used for storing and managing the saved layouts. It is possible to reuse the saved layouts from this tab, or just simply delete them. The player controller tab contains options for spawning a pre-designed controller that serves as a playable character during runtime. This player controller allows the user to inspect and explore the generated dungeons in a gameplay environment. The preview tab is used for inspecting the generated layouts in an isometric 2D view, so the user can inspect the layout from an angle that has increased visual clarity.

The interesting part about the design of user interface for the plugin, is that in some parts the UI is created from scratch, through code or through UI scenes, but other parts are not. The dungeon generator itself uses a separate UI that is integrated in Godot as shown below.



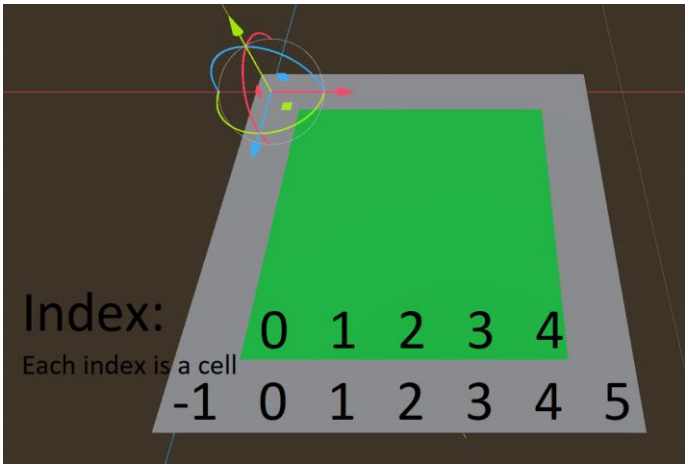
The menu on the right is invoked through export variables and buttons in the code. Parameters such as border size, room size are easily changed through this menu for the dungeon generation. Unfortunately, Godot does not have an export variable for a clickable button as of yet, but a clickable checkbox works just as fine, so they were used for functions like generating layouts or generating mesh. This menu also contains a mesh theme selector exported variable, where the user can chose which type of mesh to spawn. All of these variables are tightly connected with the whole project, and signals are being sent between this menu, dungeon generator and the plugin itself.

Early in development, there was an attempt to create this exact menu inside the plugin UI shown at the start of the chapter. After some time researching and writing code for this task, it was evident that this is not the correct way of doing things in Godot, and it was basically equivalent to rewriting Godot’s source code and fit it into a specific UI window. This approach proved to be extremely inefficient and not scalable, so it was quickly scrapped and the generator menu stayed with the Godot’s export variable system and UI. Even though the generation menu and plugin menu are separate, they are still strongly connected with each other through signals and other methods.

4.2 Dungeon Layout Generation

How does the dungeon generator logic work? Lets start with the very beginning of the generation, which is setting the border which serves as an area where the layout is generated, within the bounds. Lets say our border size is set to 5. In this case, the rooms that can be spawned are allowed to be only

within the 5x5 squares, with indexes ranging between 0 and 4, but the border surrounding that would go from index -1 to 5.




```
func visualize_border():
    if gridmap:
        gridmap.clear() # need to clear every time because the textu
        for pos1 in range(-1, border_size + 1): # border size has to
            gridmap.set_cell_item(Vector3i(pos1, 0, -1), 3)
            gridmap.set_cell_item(Vector3i(pos1, 0, border_size), 3)
            gridmap.set_cell_item(Vector3i(border_size, 0, pos1), 3)
            gridmap.set_cell_item(Vector3i(-1, 0, pos1), 3)
```

Figure 4 & 5. Border and available cells visualised and the GDScript code that creates the borders.

Creating rooms within the designated border follows a straightforward procedure. A random starting point is selected, along with the room's width and height dimensions. From this chosen point, room tiles are placed, progressing from left to right and top to bottom. A critical aspect of this process is that the selection of starting points is constrained by the border's dimensions and the predetermined width and height of the room.

To prevent rooms from overlapping, a check is performed to ascertain whether any tiles are already occupied before room creation proceeds. This involves the use of a variable that contains the room margin value. The implementation mirrors that of room creation, with the notable adjustment being the range set from the negative of the margin value to the height of the room plus the margin variable. Essentially, the range of iterations is expanded by a margin from both ends, as illustrated in the code snippet below.

```
# Minimum distance checker that rooms should keep away form each other (margin)
for r in range(-room_margin, height + room_margin): #for every row in height
    for c in range(-room_margin, width + room_margin): #for every column in width
        var pos : Vector3i = start_pos + Vector3i(c, 0, r) # variable for the posit
        if gridmap.get_cell_item(pos) == 0: #check if the cell is a previously defi
            generate_room(rec-1) #gererate rooms until recursion checker is trigger
            return
```

Fig 6. Code for room margin to avoid room overlaps.

4.2.1 Dungeon Layout Rooms

One of the most complex parts of the dungeon generator is the logic behind calculation and organisation of rooms, connections, and tunnels. Below is a diagram that shows a high level overview of the dungeon generation process.

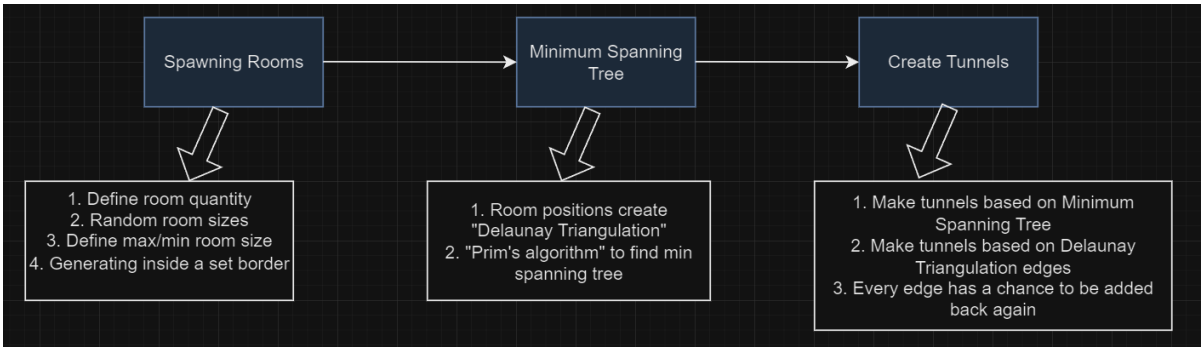


Fig 6.1. High level overview of how the dungeons are generated.

Initially, a variety of rooms, each with random sizes and locations, are spawned within a previously specified border in the generator. This step marks the beginning of the process. Following this, the positions of all these spawned rooms are taken to create what is known as a Delaunay Triangulation. Delaunay Triangulation serves as a straightforward method for drawing triangles from a collection of points, arranged so that no point lies within the circumcircle of any triangle that is drawn.

Although research on triangulation algorithms was conducted, direct implementation of the algorithm as traditionally taught was not required for the project. Godot provides a built-in class for this purpose, named “triangulate_delaunay()”, as illustrated in Figure 4 below. This snapshot, taken from within the Godot editor, showcases the built-in documentation that is readily accessible through CTRL-clicking on the involved functions.

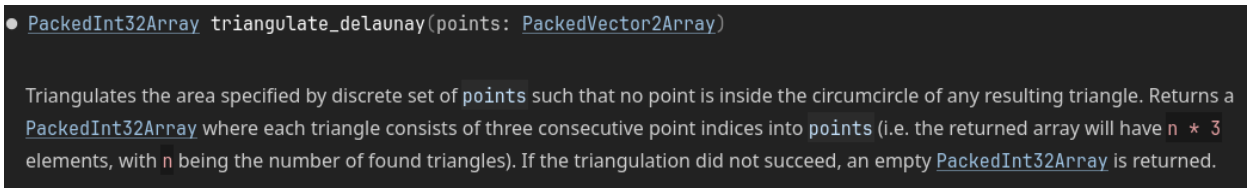


Fig 6.2. How Delaunay Triangulation is done using Godot’s triangulate_delaunay() function of the Geometry2D class.

Following the completion of the Delaunay Triangulation, the next step involves finding the Minimum Spanning Tree within the triangulated network. This ensures that points are connected in such a manner that allows for travel between any two points at any given time.

4.2.2 Tunnels and Minimum Spanning Tree

Tunnels are created based on the Minimum Spanning Tree. Yet, relying solely on the MST algorithm would yield linear layouts with minimal deviation from a straight path, falling short of creating engaging dungeon layouts. To circumvent this issue of monotony, a selection of Delaunay edges is randomly added back into the Minimum Spanning Tree. This approach fosters more complex and intriguing layouts, introducing various tunnel cycles and loops, thereby enhancing the dungeon's sense of adventure.

For managing graph calculations related to Delaunay Triangulation and the Minimum Spanning Tree, Godot's built-in class "AStar2D" was utilized. "AStar2D" simplifies the process of tracking points and their connections. An "AStar2D" object requires an ID and the position for each point, proving invaluable in establishing the graph points for Delaunay Triangulation. This information, combined with Godot's built-in function "triangulate_delaunay()", facilitated the effortless setup of triangulation and the connections required for tunnel creation between the generated rooms.

To construct the Minimum Spanning Tree, a random point is selected as the starting point from which the MST begins to take shape, adhering to a specific set of rules. At each step, potential connections are identified, governed by a simple yet crucial rule: a connection must lead from a visited point to an unvisited point, exclusively. The selection process among these connections typically prioritizes the weight of the edges in weighted graphs; however, given that the graph in this context is unweighted, the algorithm opts for the shortest connection, which is logically consistent with the aim of connecting rooms in a dungeon. Once a connection is made, the new point is marked as visited. This procedure of identifying all possible connections from visited to unvisited points is repeated, continuing until all points have been visited, culminating in the formation of the Minimum Spanning Tree.

Up to this point, the locations of room tiles have been stored, and the method by which they should connect has been determined using a graph. With the graph's information extracted through code, the creation of tunnels based on this data is now possible. For each connection present in the tunnel graph, the ends of the connection are identified, and within each room, the tile closest to the other room is designated as the door tile. The next step involves finding the path between these two door tiles, marking the route as tunnel tiles, which is where the "AStar2D" pathfinding algorithm previously mentioned plays a crucial role. Once this process is applied to the entire graph, the dungeon layout is considered complete.

4.3 Dungeon Mesh Generation

Now it's time to apply dungeon textures based on the generated layouts. The textures for walls, doors, floors, and ceilings were carefully designed in Blender 3D, as shown in screenshots [Fig 7] and [Fig 8] in the appendix. These textures were arranged to overlap on every side on a single cell, then exported as a ".glb" file and imported into the Godot project [Fig 9]. Notably, each texture imported includes all the necessary collision shapes, achieved in Blender by appending "-col" to the mesh name before exportation. These textures are organized in a separate Godot scene, which is spawned for each cell within the generated grid map. Once a texture cell is in place, decisions are made regarding which textures need to be removed or retained, determined by the cell's type and the types of adjacent cells.

As an example, if the current cell index is 0 and the adjacent cell to the right also has an index of 0, this indicates the presence of a room tile with another room tile to the right. In response, the program eliminates all walls and doors for that tile to create an opening to the cell on the right. Conversely, if the cell's index is 1, denoting a tunnel tile, and it is adjacent to a room tile, the door texture is removed while the wall between the tiles remains intact. This logic is applied to each side of the cell, as illustrated in figure 10 below.

```
for i in 4: #each side of the wall
    var cell_n = c + directions.values()[i]
    var cell_n_index = gridmap.get_cell_item(cell_n)
    if cell_n_index == -1 || cell_n_index == 3:
        handle_none(dungeon_cell, directions.keys()[i])
    else:
        var key = str(cell_index) + str(cell_n_index)
        call("handle_"+key, dungeon_cell, directions.keys()[i])
```

Fig 10. Logic to remove the necessary textures from each side of the tile.

To facilitate this behavior, a strategy was devised to generate the name of the method intended for invocation, based on the cell information previously calculated. The information obtained from the code informs the creation of a method that is executed at runtime. Thus, if situated on a room cell

adjacent to a door cell, the selected method is "handle_room_door()", which, in this context, is denoted as "handle_02()". Conversely, if the current cell does not correspond to any of the dungeon tiles, the "handle_none" method is invoked, resulting in no mesh being deleted.

4.4 Implementation Nuances

Throughout the implementation of this project, many nuanced challenges were encountered. Notably, the initial version of the plugin UI lacked Undo/Redo functionality - a critical feature for enhancing user experience. For instance, an attempt to undo the addition of a wall mesh through the plugin would result in Godot reverting the last action performed in the Godot editor itself, prior to any plugin activities. Occasionally, the undo function would revert two actions simultaneously - one from the Godot editor and one from the plugin. This behaviour was notably problematic, prompting an immediate focus on rectifying the integration of undo/redo functionality between the plugin and the Godod editor.

Drawing upon previous experiences with the Memento and Command design patterns from “CS264 Software Design” university module, the discovery that Godot's scripting language, GDScript, offered a more direct solution was indeed a pleasant surprise. The availability of the "get_undo_redo()" object, along with a suite of methods like ".create_action" and ".add_undo_method", facilitated a seamless integration of undo/redo capabilities within the plugin. This ensured a smooth and intuitive user experience that aligns with the native behaviour of the Godot editor. Below figure is one example of how the Undo/Redo functionality was implemented into the spawning mesh logic.

```
523 		# For undo/redo functionality:
524 		undo_redo.create_action("Adding gridmap from layouts")
525 		undo_redo.add_do_method(current_scene, "add_child", mesh_from_layouts)
526 		undo_redo.add_do_reference(mesh_from_layouts)
527 		undo_redo.add_undo_method(current_scene, "remove_child", mesh_from_layouts)
528 		undo_redo.commit_action(true)
529 		mesh_from_layouts.visible = true
530 		mesh_from_layouts.owner = current_scene
```

Fig 11. Undo/Redo functionality implemented for generating mesh

Signals played a crucial role during implementation process. There are two ways to implement signals into the project, through code and through Godot UI. As the project continued to grow, it was evident that creating signals through code is the best way to do it. Implementing signals through code, specifically for the feature where saving a dungeon layout triggers a signal to the plugin, carrying data for the grid map and mesh, showcases an advanced utilization of Godot's dynamic connection system. This method not only improved code organization by centralizing signal management within scripts but also contributed to better version control practices. By writing code for signal connections, it was possible to track changes more effectively, as these changes were contained within script files that could be easily inspected and managed through version control systems such as GitHub. Scripted flexibility was a critical aspect of this approach, allowing for dynamic adjustments to signal parameters and connections based on the needs of the systems in the project. The ability to programmatically define which data was sent with each signal - in this case, grid map and mesh data - enabled a more sophisticated interaction between different components of the project. Learning to implement signals through code significantly enhanced the development process.

In the experience of working with Godot, it became evident that the engine struggles significantly with handling file and directory renames within a project. This limitation was discovered through a challenging learning curve, wherein the movement of files between folders and subsequent renaming

of these directories led to unexpected project corruption. The issue went unnoticed until many commits later, highlighting a critical pitfall in Godot's project management system. Such occurrences underscore the importance of tracking and manual updating of references within the project files to prevent loss of data integrity. This experience spotlighted the need for enhanced awareness and caution when reorganizing project assets in Godot, underscoring a potential area for improvement in future versions of the engine.

Chapter 5: Evaluation

5.1 Test Case Documentation

As the development of the “Dungeon Forge” continued, the systems started to become more complex. It was necessary to create some kind of document to keep track of functions and errors. A “Test Case” document [Fig 12] was created with test cases to pass or fail. One of the sections in the document is “Functional Requirements” as shown Figure below.

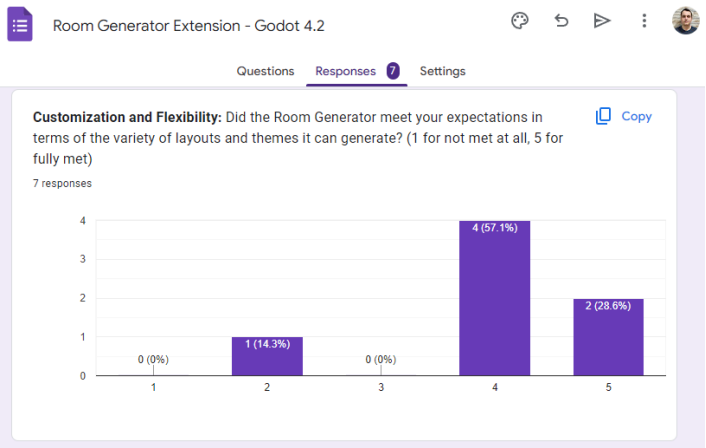
	Functional Requirements (Godot Editor UI):	Results	Comments
8	The extension adds a new UI window in the bottom right part of the Godot Editor.	PASS	
9	The bottom Panel has four tabs: “Models”, “Layouts”, "Player Controller" and "Preview".	PASS	
10	Model works with individual scenes that a designer wants to work with in a project: (Note: These scenes are typically rooms, or sections of a scenery, like a part of a surface	PASS	
11	The user can attach a scene (e.g., of a room) as such a ‘model’ and instantiate it with a button.	PASS	
12	Models can be added and deleted.	PASS	
13	Undo/Redo functionality works.	PASS	
14	The extension UI and text has no overlaps	FAIL	On small laptop screens, the UI might overlap a bit when opened for the firs time. Some adjustments coming.
15	The project saves the extension state when rebooted	FAIL	The saved layouts only stay for the duration of the development session at the moment. If user saves 10 layouts and restarts the editor, he will lose the 10 layouts saved. Working on solution.
16	The user can define layouts from templates, as building blocks used during a design session.	PASS	
17	Models can be rotated and placed into these larger structures.	PASS	

Fig 12.1. Evaluation Test Case document. Functional Requirements section.

This kind of document ensured that no bugs or errors will get lost or forgotten during the development process, and it could be considered as “Functional Tests”. The comment section lists the necessary actions required for the failed test case, which can work as a bug ticket for developer to work on.

After the project gets larger and more complicated elements are integrated, a great addition to the “Test Case” document would be a section for “Repro Steps”, where a detailed step by step guide would be written on how to trigger a specific test case. This will also be extremely beneficial to the developer who may take this project for future development. Additionally, excel allows various calculations of cells, so a section with detailed statistics of results would be a great addition.

5.2 Data Collection



"Google Forms" provides a great tool for developers to gather feedback from users in a form of a questionnaire. One example question from the form could be seen from the screenshot on the left. This form was sent out to other students from the Computer Science course, as well as some of the Godot enthusiast in other communities. The results of the questionnaire highlighted the areas of the project that could have some improvements, and other areas that were done well. This kind of form can be kept for future use when more functionality and improvements are implemented. The form currently has 12 questions, which include each area of the whole plugin. More example questions could be found in [Fig 13] of the appendix.

5.3 Software Testing

Previously, university modules such as "CS265 Software Testing" were thoroughly studied, and it was time to apply some of the learned material to practice. Some of the key concepts such as "Integration Tests" and "Acceptance Tests" were studied during the course, are now part of the core testing methods involved in the project's development. These concepts could be used in combination with Godot's IDE, Editor, and Debugger. Godot has a powerful debugger integrated into the engine. Just like most of the IDE's, you can set breakpoints and perform step-by-step execution of your code. It allows, performance monitoring, as well as keeping track of errors and warning messages. Detailed inspection of variables and calculations were crucial in aiding the development process. The debugger was used in testing the calculations for spawning dungeon layouts. A clear inspection of multiple nested loops was necessary to ensure that the math and logic were done correctly.

5.3.1 Integration Tests

Integration tests were necessary for this application, due to the interaction between main components, such as the plugin UI and the dungeon generator. Integration tests would verify that the functions from dungeon generator would correctly interact with plugins UI, for example when saving a layout from generator into the plugin. Godot allowed the integration tests to be performed in a simple way, through the signal system. The signals in Godot are very powerful, as it is effortless to set up a signal through code, and connect to the signal with other parts of the application. These signals are powered by functions, that accept necessary parameters and return necessary values if needed for specific test. One example of a test done this way was for the saving layouts from dungeon generator into the plugin, and it was done using a signal that passed 3 parameters:

```
emit_signal("save_to_layouts_signal", gridmap_copy, mesh_copy, mesh_theme)
```

```gdscript
func save_to_layouts_function(gridmap, mesh, mesh_theme):
    if !item_list:
        item_list = dockedScene.get_node("TabContainer/Layouts/ItemList")

    # Create a dictionary for the current dungeon layout and its mesh, then
    var dungeon_dict = {
        "gridmap": gridmap,
        "mesh": mesh,
        "mesh_theme": mesh_theme}
    stored_layouts.append(dungeon_dict)

    # Add an item to the ItemList for this dungeon layout
    item_list.add_item("Generated Layout " + str(item_list.get_item_count()))
    print("Layouts: ", stored_layouts)

    #check what I have in the dictionary
    for layout_dict in stored_layouts:
        print("Layout Key: ", layout_dict)
        for key in layout_dict:
            print("   ", key, ": ", layout_dict[key])
```

This signal is then received inside the plugin, and appropriate dissection of these variables is performed and necessary tests are carried out as shown on the code snippet on the left.

Just for testing purposes, this code snippet was set up to test and inspect each element of the dictionary by printing keys and values to the console. This ensured that there is no unexpected interaction between two distant areas of the project, and every element that is passed between the scripts are correct.

5.3.2 Performance Tests

It is quite easy to crash the program by spawning a million nodes per second, as an example. And this is where “Performance Tests” come into play. Since the plugin focuses on dungeon generation within the editor workspace, it must be certain that specific generation outcomes do not overload the computer or crash the program. The designed UI for dungeon generator allows the user to specify really high values for dungeon parameters, so it was clear when the program underperformed when higher values were selected in the menu.

The performance tests began when the Godot editor crashed for the first time. The editor would crash if the dungeon layout consists of hundreds of rooms, and mesh would spawn for each cell at the same time. After tinkering with more values, it was evident that around the 120 room number mark, the performance of the application would drop. A clever solution was to implement a timer for each iteration of spawning mesh nodes. Each time a new cell is processed and a corresponding dungeon cell is spawned, the variable “*t*” is incremented. Every 10th cell (when “*t*” modulo 10 is equal to 9), the code execution pauses for one frame. This pause allows the texture’s rendering process to catch up and display the newly added cells smoothly, avoiding potential performance issues that could arise from adding many cells to the scene at once without any breaks. After the pause, the loop continues to the next cell, and the process repeats until all relevant cells have been processed. This simple addition improved the generation process and fixed crashes for a lot of the cases.

```gdscript
if t%10 == 9: await get_tree().create_timer(0).timeout
```

5.3.3 Acceptance Testing

One of the software testing practices, commonly referred to as "User Acceptance Testing" (UAT), which is where a developer gives the application to a user to test. UAT is a critical phase in the software development lifecycle, which allows the developers to catch issues which might have slipped on the development side. And this project was no exception, as a couple of critical issues were caught during UAT phase. One of the issues caught was that the controls for the player controller would not work on different machines, as the input mapping had to be done on each individual machine to match the controls script, shown in [Fig 14] of the appendix.

Another concept related to this practice, particularly in scenarios where the project is not yet ready for beta testing, involves sharing screenshots and snippets of code on an official Discord server for Godot. In this approach, valuable feedback and advice is gathered, and improvements could be

suggested by experienced developers from the server. This method allows developers to engage with their community and collect insights, which enhances the development process. It can be seen as a preliminary step to user acceptance testing, focusing on feedback from actual users through discussions and comments on visual and code-based previews to polish the quality of the application.

Chapter 6: Conclusion

6.1 Results

The completion of “Dungeon Forge” marked a significant journey of growth and discovery, characterized by a steep learning curve initially. The unfamiliarity with both the Godot engine and its associated programming language, GDScript, presented considerable challenges. This project was largely driven by a learn-as-you-go approach, where understanding and proficiency with Godot's functionalities were developed in tandem with the extension's development. This method of learning, while effective in fostering a deep understanding of the engine, led to instances where tasks were executed less efficiently than might have been possible with prior extensive knowledge of Godot and GDScript.

As the project progressed, tangible examples of the initial inefficiencies became apparent. For instance, early in the development phase, certain elements such as buttons and tabs were integrated using the Godot engine's user interface, a method that seemed straightforward at the time. However, with the accumulation of experience and a deeper understanding of the engine's capabilities, it became evident that incorporating these elements through coding was not only more efficient but also allowed for greater control and customization. This shift in approach, though beneficial for the project's development, introduced a degree of inconsistency in the project's structure. This discrepancy served as a valuable lesson in the importance of understanding the full breadth of tools and methods available within the development environment from the outset, highlighting the benefits of flexibility and the willingness to adapt strategies as one's proficiency grows.

This learning curve and the evolution of development strategies were similarly evident in the handling of signals within Godot. Initially, signals were connected through the engine's user interface, a process that, while intuitive at first glance, occasionally led to confusion and a lack of clarity regarding the relationships between nodes. This method made it challenging to track which node was signalling what, leading to a cumbersome review process. As familiarity with the engine grew, the discovery that signals could also be defined and managed directly through code was a revelation. This approach significantly enhanced flexibility, allowing for a more streamlined and organized setup of signal connections.

Despite the strides made in learning and development, the project is not without its vulnerabilities, as thoroughly documented in the accompanying "Test Case" document. One notable issue arises when the plugin is removed from a project; Godot generates an error indicating that certain textures cannot be found. This challenge underscores a gap in the initial functional requirements and points to an area for future improvement. Plans are in place to address this by prompting users to import the textures used by the plugin, a solution that, while effective, highlights a missed opportunity for foresight in the project's early stages. The experience gained through this oversight serves as a crucial lesson for future projects, emphasizing the value of anticipating potential pitfalls and incorporating comprehensive error handling from the outset. This reflection not only marks a point of learning but

also sets a precedent for more robust and error-resilient development practices in future projects.

6.2 Future Work

Looking ahead, there are a lot of opportunities for enhancing and expanding the plugin's functionality, particularly regarding its user interface. Future iterations could greatly benefit from the inclusion of additional tabs within the UI, offering users more nuanced control over the dungeon generation process. For example, a new tab could be introduced where users have the option to adjust the darkness or visibility levels within the dungeon, enabling them to set a more atmospheric tone or challenge level. Similarly, the introduction of environmental modifiers could allow users to further customize the dungeon's ambiance and mechanics. These proposed enhancements would not only enrich the user experience but also demonstrate the plugin's versatility and adaptability to a wide range of game development needs, opening up new avenues for creativity and customization.

Another promising avenue for future development lies in enhancing the plugin's capacity for customization through user-generated content. Allowing users to import their own dungeon textures directly into the extension represents a significant step towards this goal. By integrating a drag-and-drop functionality into the plugin's UI, users could personalize their dungeons with unique textures, thereby creating environments that better reflect their vision or the specific atmosphere they wish to achieve. This feature would not only empower users by giving them more control over the visual aspects of their dungeons but also foster a more engaging and user-friendly experience. The ability to directly influence the aesthetic and thematic elements of the dungeons would encourage experimentation and creativity from developers.

The potential for scaling and upgrading the dungeon generator system is vast, with numerous possibilities for enhancing its complexity and realism. A particularly transformative upgrade could involve the introduction of multi-level floor generation, complete with staircases that connect different floors, ascending or descending. This feature would dramatically increase the depth and intricacy of the dungeons created, offering developers greater content to work with. Additionally, the capability to randomly spawn wall fixtures, such as torches, would not only enhance the aesthetic appeal of the dungeons but also play a functional role in lighting and visibility.

Expanding the dungeon generator's capabilities to include the generation of outdoor areas would represent another significant enhancement, adding a new layer of diversity and immersion to the generated environments. By incorporating logic for outdoor scenery, such as grassy fields, trees, and various ground decorations, the plugin could offer a seamless transition between the claustrophobic confines of dungeons and the expansive freedom of outdoor landscapes. This addition would synergize well with the proposed environmental tab mentioned earlier, allowing users to not only design the physical layout of these areas but also to customize atmospheric conditions. For example, users could choose between sunshine and rain, generating environments with potentially slippery surfaces and reduced visibility. Such versatility in environmental design would elevate the user experience, offering an environment creation that encompasses both indoor and outdoor scenarios.

References

[1] “Vazgriz”. (18 November, 2019). Procedurally generated dungeons. https://vazgriz.com/119/procedurally-generated-dungeons

[2] Elisa Zancolo. (July 2008). Minimum Spanning Trees Using Delaunay Triangulation. https://netlibrary.aau.at/obvuklhs/content/titleinfo/2410978/full.pdf

[3] “Superflat”. (29 August, 2013). Delaunay triangulation. Where to start?. https://gamedev.stackexchange.com/questions/61424/delaunay-triangulation-where-to-start

[4] “A Adoncaac”. (3 September, 2015). Procedural Dungeon Generation Algorithm. https://www.gamedeveloper.com/programming/procedural-dungeon-generation-algorithm

[5] Boris (14 July, 2019). Dungeon Generation in Diablo 1. https://www.boristhebrave.com/2019/07/14/dungeon-generation-in-diablo-1/

[6] Nic Barkdull. (27 March, 2021). Rogue (1980). https://playbackgames.medium.com/rogue-1980-53d63e37628d

[7] Alexander Baldwin & Johan Holmberg. (Spring 2017). Mixed-Initiative Procedural Generation of Dungeons Using Game Design Patterns. https://www.diva-portal.org/smash/get/diva2:1480353/FULLTEXT01.pdf

[8] Elisha Sacks. (2012). Delaunay Triangulation (chapter 9). https://www.cs.purdue.edu/homes/cs53100/slides/delaunay.pdf

Appendix 1

Code developed for this project.

Please note that Godot's GDScript Editor does not keep code formatting when pasting code snippets, therefore code screenshots of most relevant code are added in this section.

```
83  func setup_button_connections():
84      # Connect the toggle button signal
85      first_person_controller = dockedScene.get_node("TabContainer/Player Controller/First Person Player Controller")
86      use_layout_button = dockedScene.get_node("TabContainer/Layouts/UseLayoutButton")
87      delete_layout_button = dockedScene.get_node("TabContainer/Layouts/DeleteLayoutButton")
88      dungeon_menu_button = dockedScene.get_node("TabContainer/Models/DungeonGeneratorMenu")
89      wooden_cabin_menu_button = dockedScene.get_node("TabContainer/Models/DungeonGeneratorMenu")
90      wooden_cabin_menu_button = dockedScene.get_node("TabContainer/Models/WoodenCabinGeneratorMenu")
91      frozen_caves_menu_button = dockedScene.get_node("TabContainer/Models/FrozenCaveGeneratorMenu")
92
93      first_person_controller.connect("pressed", create_first_person_controller)
94      wooden_cabin_menu_button.connect("pressed", wooden_cabin_menu_button_pressed)
95      frozen_caves_menu_button.connect("pressed", frozen_caves_menu_button_pressed)
96      use_layout_button.connect("pressed", use_layout_button_pressed)
97      delete_layout_button.connect("pressed", delete_layout_button_pressed)
```

Code Snippet 1: Button connections setup in plugin.

```
126 func instantiate_gridmap_from_layouts(gridmap):
127     var current_scene = get_editor_interface().get_edited_scene_root()
128     var gridmap_from_layouts = gridmap
129
130     if current_scene:
131         gridmap_from_layouts.name = "GridMap_" + str(current_scene.get_child_count())
132
133         # For undo/redo functionality:
134         undo_redo.create_action("Adding gridmap from layouts")
135         undo_redo.add_do_method(current_scene, "add_child", gridmap_from_layouts)
136         undo_redo.add_do_reference(gridmap_from_layouts)
137         undo_redo.add_undo_method(current_scene, "remove_child", gridmap_from_layouts)
138         undo_redo.commit_action(true)
139         gridmap_from_layouts.owner = current_scene
140     else:
141         print("No active scene!")
```

Code Snippet 2: How grid maps are instantiated from plugin layouts menu. Undo/Redo included.

```gdscript
func wooden_cabin_menu_button_pressed():
    if wooden_cabins_popup_menu:
        wooden_cabins_popup_menu.clear()
        if wooden_cabins_popup_menu.is_connected("id_pressed", instantiate_wooden_cabin_texture):
            wooden_cabins_popup_menu.disconnect("id_pressed", instantiate_wooden_cabin_texture)

    wooden_cabins_popup_menu = wooden_cabin_menu_button.get_popup()
    var popup_theme = Theme.new()  # Create a new theme

    var style_box = StyleBoxTexture.new()
    var bg_image = WOODEN_CABIN_BACKGROUND
    style_box.texture = bg_image

    var popup_font = FontFile.new()
    popup_font.font_data = load("res://addons/room-generator/fonts/Diablo Heavy.ttf")  # Replace w
    popup_theme.set_font("font", "PopupMenu", popup_font)
    popup_theme.set_color("font_color", "PopupMenu", Color(0.663, 0.91, 0))
    popup_theme.set_font_size("font_size", "PopupMenu", 30)

    wooden_cabins_popup_menu.theme = popup_theme
    wooden_cabins_popup_menu.add_theme_stylebox_override("panel", style_box)

    wooden_cabins_popup_menu.add_item("Wall")
    wooden_cabins_popup_menu.add_item("Entrance")
    wooden_cabins_popup_menu.add_item("Ceiling")
    wooden_cabins_popup_menu.add_item("Floor")
    wooden_cabins_popup_menu.add_item("GridMap Generator")
    wooden_cabins_popup_menu.connect("id_pressed", instantiate_wooden_cabin_texture)
```

Code Snippet 3: How the menus from the “Models” tab are populated upon pressing the button.

```gdscript
func instantiate_frozen_caves_texture(id):
    var current_scene = get_editor_interface().get_edited_scene_root()
    var frozen_caves_texture

    match id:
        0:
            frozen_caves_texture = FROZEN_CAVES_WALL.instantiate()
        1:
            frozen_caves_texture = FROZEN_CAVES_DOOR.instantiate()
        2:
            frozen_caves_texture = FROZEN_CAVES_CEILING.instantiate()
        3:
            frozen_caves_texture = FROZEN_CAVES_FLOOR.instantiate()
        4:
            instantiate_dungeon_gridmap()
            return
        _:
            print("Unknown model selected")
```

Code Snippet 4: How the correct textures are picked for spawning.

```
func instantiate_dungeon_wall():
    var _dungeon_wall = DUNGEON_WALL_UP_LADNO.instantiate()
    var current_scene = get_editor_interface().get_edited_scene_root()

    if current_scene:
        _dungeon_wall.name = "dungeon_wall_" + str(current_scene.get_child_count())

        # For undo/redo functionality:
        undo_redo.create_action("Create Dungeon Wall")
        undo_redo.add_do_method(current_scene, "add_child", _dungeon_wall)
        undo_redo.add_do_reference(_dungeon_wall)
        undo_redo.add_undo_method(current_scene, "remove_child", _dungeon_wall)
        undo_redo.commit_action(true)
        _dungeon_wall.owner = current_scene
    else:
        print("No active scene!")
```

Code Snippet 5: How individual textures are instantiated, with undo/redo functionality.

```
func save_to_layouts_function(gridmap, mesh, mesh_theme):
    if !item_list:
        item_list = dockedScene.get_node("TabContainer/Layouts/ItemList")

    # Create a dictionary for the current dungeon layout and its mesh, then append it
    var dungeon_dict = {
        "gridmap": gridmap,
        "mesh": mesh,
        "mesh_theme": mesh_theme}
    stored_layouts.append(dungeon_dict)

    # Add an item to the ItemList for this dungeon layout
    item_list.add_item("Generated Layout " + str(item_list.get_item_count() + 1), DUNGEON_GENERATOR_ICON, true)
    print("Layouts: ", stored_layouts)

    #check what I have in the dictionary
    for layout_dict in stored_layouts:
        print("Layout Key: ", layout_dict)
        for key in layout_dict:
            print("   ", key, ": ", layout_dict[key])
```

Code Snippet 6: How layouts are saved into the plugin, with debugging prints to see exactly what is saved.

```gdscript
func confirmation_dialog_setup():
    # Create the ConfirmationDialog dynamically
    confirmation_dialog = ConfirmationDialog.new()
    confirmation_dialog.dialog_text = "Are you sure you want to use this dungeon layout?"
    add_child(confirmation_dialog)

    confirmation_dialog.get_ok_button().text = "Yes"

    # Connect signals for the buttons
    confirmation_dialog.connect("confirmed", _on_confirmation_dialog_confirmed)

func _on_confirmation_dialog_confirmed():
    # Assuming you have a function to retrieve the selected layout details
    var selected_index = item_list.get_selected_items()[0]
    var layout_dict = stored_layouts[selected_index]  # Extract the selected layout
    var spawning_gridmap = layout_dict["gridmap"]
    var spawning_mesh = layout_dict["mesh"]
    var spawning_mesh_theme = layout_dict["mesh_theme"]
    spawning_mesh.set_script(load("res://addons/room-generator/dungeon/DungeonMesh.gd"))

    # Spawn both gridmap and mesh upon confirmation
    instantiate_gridmap_from_layouts(spawning_gridmap)
    instantiate_mesh_from_layouts(spawning_mesh, spawning_mesh_theme)
```

Code Snippet 7: How the confirmation popups are set up.

```gdscript
@tool
extends Node3D

@onready var floor = $floor
@onready var wall_up = $wall_up
@onready var door_up = $door_up
@onready var wall_left = $wall_left
@onready var door_left = $door_left
@onready var wall_down = $wall_down
@onready var door_down = $door_down
@onready var wall_right = $wall_right
@onready var door_right = $door_right
@onready var ceiling = $ceiling

func remove_wall_up():
    wall_up.free()
func remove_wall_down():
    wall_down.free()
func remove_wall_left():
    wall_left.free()
func remove_wall_right():
    wall_right.free()
func remove_door_up():
    door_up.free()
func remove_door_down():
    door_down.free()
func remove_door_left():
    door_left.free()
func remove_door_right():
    door_right.free()
```

Code snippet 7: Script for the mesh scenes with textures overlapping. This is used when generator removes necessary textures from the tiles.

```gdscript
func generate_room(rec: int):
    if !rec > 0 || !gridmap: #don't run if recursion limit is reached
        return
    # get random width and heights
    var width : int = (randi() % (MAX_room_size - MIN_room_size)) + MIN_room_size
    var height : int = (randi() % (MAX_room_size - MIN_room_size)) + MIN_room_size

    #pick starting position
    var start_pos : Vector3i
    start_pos.x = randi() % (border_size - width + 1) # need to have +1 there at the end beca
    start_pos.z = randi() % (border_size - height + 1)

    # Minimum distance checker that rooms should keep away form each other (margin)
    for r in range(-room_margin, height + room_margin): #for every row in height
        for c in range(-room_margin, width + room_margin): #for every column in width
            var pos : Vector3i = start_pos + Vector3i(c, 0, r) # variable for the position
            if gridmap.get_cell_item(pos) == 0: #check if the cell is a previously defined te
                generate_room(rec-1) #gererate rooms until recursion checker is triggered
                return

    #we fill in the columns from left to right
    var room : PackedVector3Array = []
    for r in height: #for every row in height
        for c in width: #for every column in width
            var pos : Vector3i = start_pos + Vector3i(c, 0, r) # variable for the position
            gridmap.set_cell_item(pos, 0) #set texture for dungeon walls
            room.append(pos)  #add to room array for each iteration
    room_tiles.append(room) #append whole room to the tiles

    #calculating x an z positions separately
    var avg_x : float = start_pos.x + (float(width)/2)
    var avg_z : float = start_pos.z + (float(height)/2)
    var pos : Vector3 = Vector3(avg_x, 0, avg_z)
    room_positions.append(pos)
```

Code snippet 8: How the room tiles (green tiles) are generated into grid map.

```gdscript
#this is to offset the instances position to allign with the cells in
#the grid map, since they are centered, but our objects are not.
for c in gridmap.get_used_cells(): #for each cell in grid map
    var cell_index = gridmap.get_cell_item(c) #get the index of an item used

    #if the item selected are the ones being used (0-3, 3 excluded because its border cells)
    if cell_index <= 2 && cell_index >= 0:
        var dungeon_cell = mesh_theme_to_spawn.instantiate()
        dungeon_cell.position = Vector3(c) + Vector3(0.5, 0, 0.5) #this position because cells are not pe
        dungeon_mesh.add_child(dungeon_cell)
        t += 1

        for i in 4: #each side of the wall
            var cell_n = c + directions.values()[i]
            var cell_n_index = gridmap.get_cell_item(cell_n)
            if cell_n_index == -1 || cell_n_index == 3:
                handle_none(dungeon_cell, directions.keys()[i])
            else:
                var key = str(cell_index) + str(cell_n_index)
                call("handle_" + key, dungeon_cell, directions.keys()[i])

    if Engine.is_editor_hint():
        var current_scene = EditorInterface.get_edited_scene_root()
        dungeon_cell.owner = current_scene #this allows you to work with spawned cells in your scene
        dungeon_cell.set_script("")
```

Code snippet 9: How the mesh is generated on top of the grid map, and then removed if necessary.

```gdscript
var tunnels : Array[PackedVector3Array] = [] # used to store the two door tiles positions
for p in tunnel_graph.get_point_ids(): #loop to get access to our connections
    for c in tunnel_graph.get_point_connections(p): #for connection
        if c > p: #to make sure we dont run our code on the same connections
            var room_from : PackedVector3Array = room_tiles[p]
            var room_to : PackedVector3Array = room_tiles[c]
            var tile_from : Vector3 = room_from[0]
            var tile_to : Vector3 = room_to[0]

            #to find shortest connection
            for t in room_from:
                if t.distance_squared_to(room_positions[c]) < tile_from.distance_squared_to(room_positions[c]):
                    tile_from = t
            for t in room_to:
                if t.distance_squared_to(room_positions[p]) < tile_to.distance_squared_to(room_positions[p]):
                    tile_to = t

            var tunnel : PackedVector3Array = [tile_from, tile_to]
            tunnels.append(tunnel)
            gridmap.set_cell_item(tile_from, 2)
            gridmap.set_cell_item(tile_to, 2)
```

Code snippet 10: How the door tiles are marked on the grid map.

```gdscript
for tun in tunnels: # for every tunnel in tunnels, we are making a hall variable to store the path
    var pos_from = Vector2i(tun[0].x, tun[0].z) #from this point
    var pos_to = Vector2i(tun[1].x, tun[1].z) #to this point
    var hall = astar.get_point_path(pos_from, pos_to)

    for t in hall:
        var pos = Vector3i(t.x, 0, t.y)
        if gridmap.get_cell_item(pos) < 0:
            gridmap.set_cell_item(pos, 1) # finally, set the cell to a tunnel tile
```

Code snippet 11: How the pathfinding is done.

```
98      # Below code is the Dlaunay and minimum spanning tree algorithms
99      var room_positions_v2 : PackedVector2Array = []
100     var delaunay_graph : AStar2D = AStar2D.new() #AStar2D requires and ID and a position for each point
101     var min_span_tree_graph : AStar2D = AStar2D.new()
102
103     #turn room positions into Vector2's, this only places the points
104     for p in room_positions:
105         room_positions_v2.append(Vector2(p.x, p.z))
106         delaunay_graph.add_point(delaunay_graph.get_available_point_id(), Vector2(p.x, p.z))
107         min_span_tree_graph.add_point(min_span_tree_graph.get_available_point_id(), Vector2(p.x, p.z))
```

Code snippet 12: how the points are placed on the AStar2D graph.

```
30  func _unhandled_input(event):
31      if event is InputEventMouseMotion:
32          head.rotate_y(-event.relative.x * SENSITIVITY)
33          camera_3d.rotate_x(-event.relative.y * SENSITIVITY)
34          camera_3d.rotation.x = clamp(camera_3d.rotation.x, deg_to_rad(-40), deg_to_rad(60))
```

Code snippet 13: How the camera is set up for the first person controller.

```
46      # Handle Jump.
47      if Input.is_key_pressed(KEY_SPACE) and is_on_floor():
48          velocity.y = JUMP_VELOCITY
49
50      # Handle Sprint
51      if Input.is_key_pressed(KEY_SHIFT):
52          speed = SPRINT_SPEED
53      else:
54          speed = WALK_SPEED
55
56      var input_dir = Vector2(
57          int(Input.is_key_pressed(KEY_D)) - int(Input.is_key_pressed(KEY_A)),
58          int(Input.is_key_pressed(KEY_S)) - int(Input.is_key_pressed(KEY_W))
59      )
60      var direction = (head.transform.basis * Vector3(input_dir.x, 0, input_dir.y)).normalized()
61
62      if is_on_floor():
63          if direction:
64              velocity.x = direction.x * speed
65              velocity.z = direction.z * speed
66          else:
67              velocity.x = lerp(velocity.x, direction.x * speed, delta * 7.0)
68              velocity.z = lerp(velocity.z, direction.z * speed, delta * 7.0)
69      else:
70          velocity.x = lerp(velocity.x, direction.x * speed, delta * 3.0)
71          velocity.z = lerp(velocity.z, direction.z * speed, delta * 3.0)
```

Code snippet 14: How the first person controller movement is done.

```
73      #Head bobbing
74      t_bob += delta * velocity.length() * float(is_on_floor())
75      camera_3d.transform.origin = _headbob(t_bob)
76
77      #FOV
78      var velocity_clamped = clamp(velocity.length(), 0.5, SPRINT_SPEED * 2)
79      var target_fov = BASE_FOV + FOV_CHANGE * velocity_clamped
80      camera_3d.fov = lerp(camera_3d.fov, target_fov, delta * 8.0)
81
82      move_and_slide()
83
84  func _headbob(time) -> Vector3:
85      var pos = Vector3.ZERO
86      pos.y = sin(time * BOB_FREQ) * BOB_AMP
87      pos.x = cos(time * BOB_FREQ / 2) * BOB_AMP
88      return pos
```

Code snippet 15: Continuation of movement for first person controller, with extra effects.

Appendix 2

UML Class Diagram, Test Case File, Google Forms

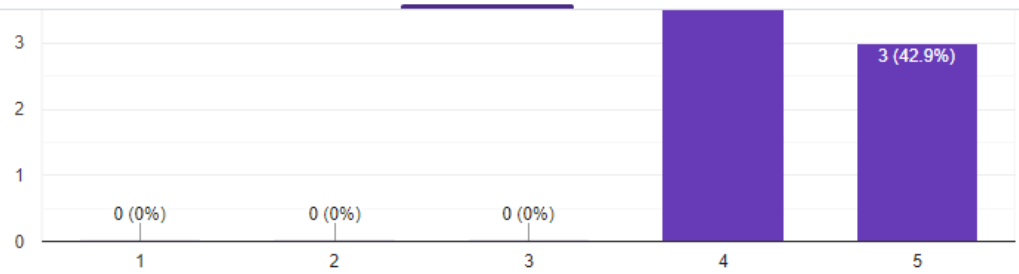
F16

	A	B	C
1	Non-functional Requirements	Results	Comments
2	The tool works as an editor extension for Godot 4.x and works as such (within the editor at design-time and not at game-runtime).	PASS	
3	The extension has a user interface that becomes available when the extension is installed.	PASS	
4	The extension can be added to individual projects.	PASS	
5	The output of the extension does not generate a dependency to the extension. In other words, removing the extension after having done the work should not break the project, meaning the extension does not need to be included to maintain the project and does not need to be compiled into a finished game/experience.	FAIL	When the addon is completely removed from the project, and the dungeon textures are present in a scene, there will be an error at some point that indicates the textures cannot be found. This issue will be fixed by prompting the user to import the mesh textures into the project when saving the generated dungeons. Another great solution for this, would be to implement functionality that allows the user to import their own textures into the dungeon generator by simply dragging and dropping them into a specific UI slot.
6			
7	Functional Requirements (Godot Editor UI):	Results	Comments
8	The extension adds a new UI window in the bottom right part of the Godot Editor.	PASS	
9	The bottom Panel has four tabs: “Models”, “Layouts”, "Player Controller" and "Preview".	PASS	
10	Model works with individual scenes that a designer wants to work with in a project: (Note: These scenes are typically rooms, or sections of a scenery, like a part of a surface	PASS	
11	The user can attach a scene (e.g., of a room) as such a ‘model’ and instantiate it with a button.	PASS	
12	Models can be added and deleted.	PASS	
13	Undo/Redo functionality works.	PASS	
14	The extension UI and text has no overlaps	FAIL	On small laptop screens, the UI might overlap a bit when opened for the firs time. Some adjustments coming.
15	The project saves the extension state when rebooted	FAIL	The saved layouts only stay for the duration of the development session at the moment. If user saves 10 layouts and restarts the editor, he will lose the 10 layouts saved. Working on solution.
16	The user can define layouts from templates, as building blocks used during a design session.	PASS	
17	Models can be rotated and placed into these larger structures.	PASS	
18			
19	Dungeon Generator Checks	Results	Comments
20	The user should be able to generate a single room.	PASS	
21	Two rooms, connected by a door	PASS	This interraction happens, but is uncommon. Most of the time the generated rooms are connected by a tunnel that is longer than 1 cell.
22	Three rooms, connected by tunnels	PASS	
23	Two rooms, connected by a staircase going upwards.	FAIL	Multi floor dungeons are not implemented yet, but is the plan for the future.
24	Three rooms, connected	PASS	
25	Ten rooms, connected	PASS	
26	Generator UI Buttons and setters	PASS	
27	Generating mesh	PASS	
28	Clearing mesh	PASS	
29	Generating different mesh over existing mesh works	PASS	
30	Saving to layouts	PASS	It works, however, the layouts get deleted when restarting the project.
31	Free of errors	FAIL	Currently, if the generator stays in the scene when restarting project, an error will appear indicating theres is something wrong with the scene parenting. It doesn't break the functionality but needs to be looked at.

Fig 12. Full list of test cases in the Test Case document.

Room Generator Extension - Godot 4.2

Questions Responses 7 Settings



Overall Satisfaction: What features did you find most useful in the Room Generator extension?

4 responses

The ability to generate rooms

The ease of using the generator

The way you can select the amount of rooms you want to generate

The amount of settings allowing you to personalize this tool

Further Improvement: What improvements or additional features would you like to see in future updates of the Room Generator extension? Are there any aspects of the extension that you believe could be improved to enhance your workflow?

2 responses

I would make the textures a little better, apart from that it's great!

I would love to see some openings (windows, ect) in the rooms to not feel like you are trapped in a very secluded place

General Feedback: Do you have any other comments or feedback regarding the Room Generator extension?

2 responses

Great work!

The idea is great, and I hope that with more refinement, it will help other people.

Fig 13. Collected feedback and statistics from online questionnaire using Google Forms.

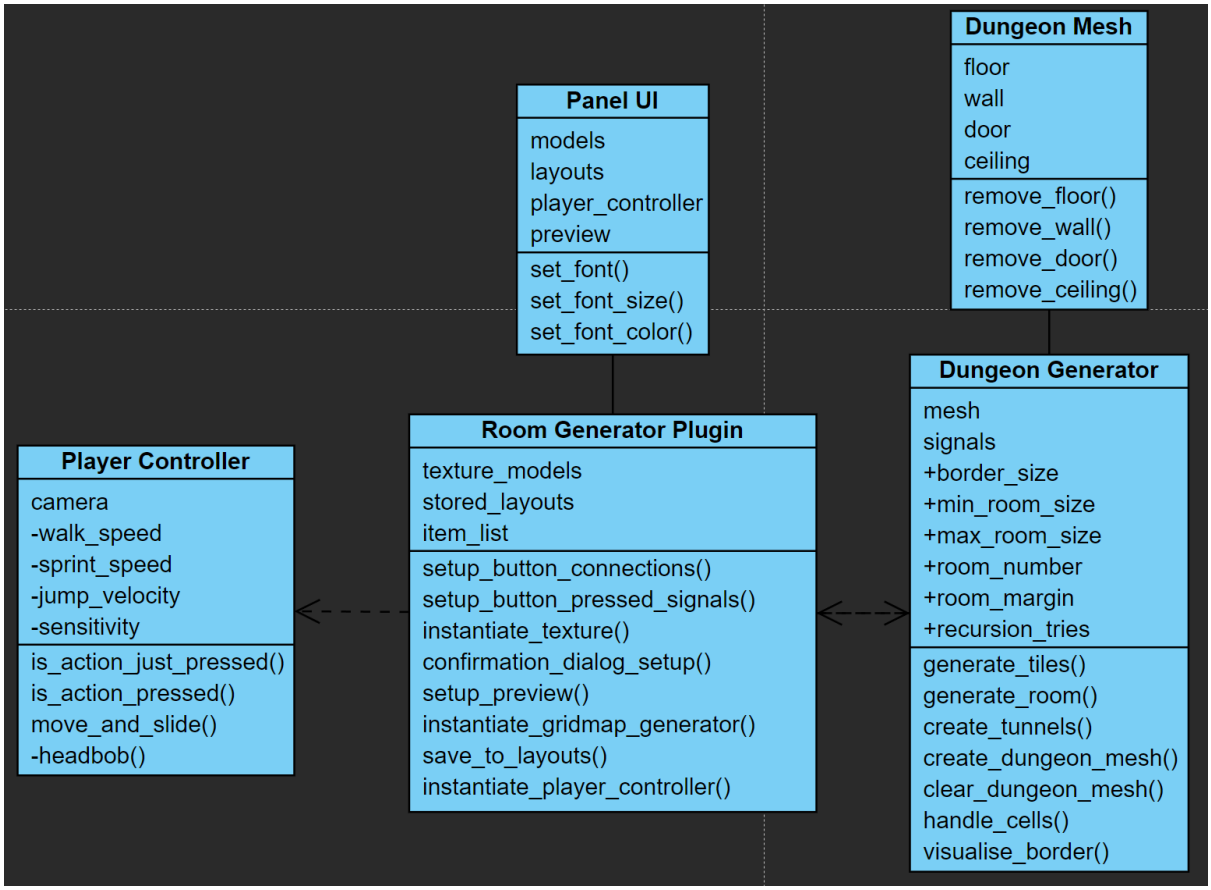


Fig 13.2. UML Class diagram for the “Dungeon Forge” extension for Godot.

Appendix 3

Screenshots of the project implementation

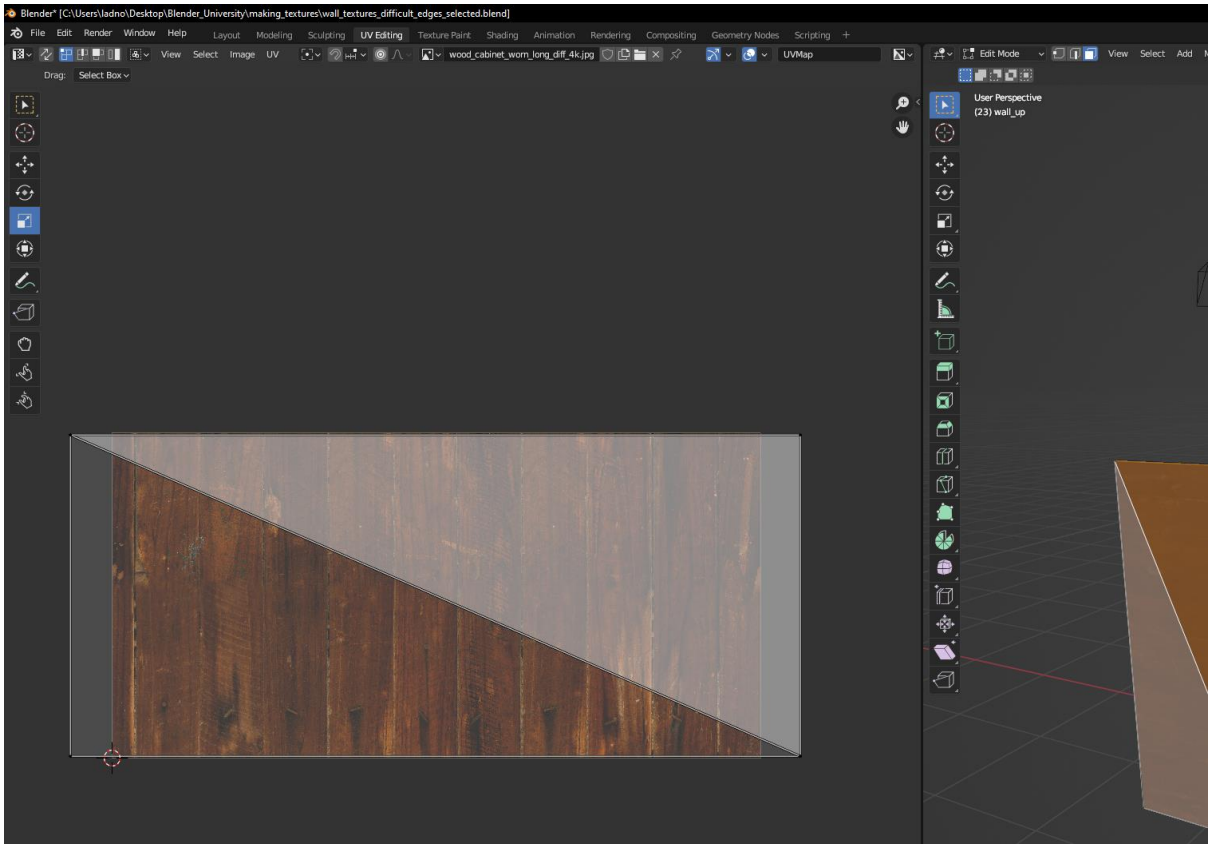


Fig 1. Setting up UV mapping for wooden walls. Techniques used were learned from “CS426 Computer Graphics” university module.

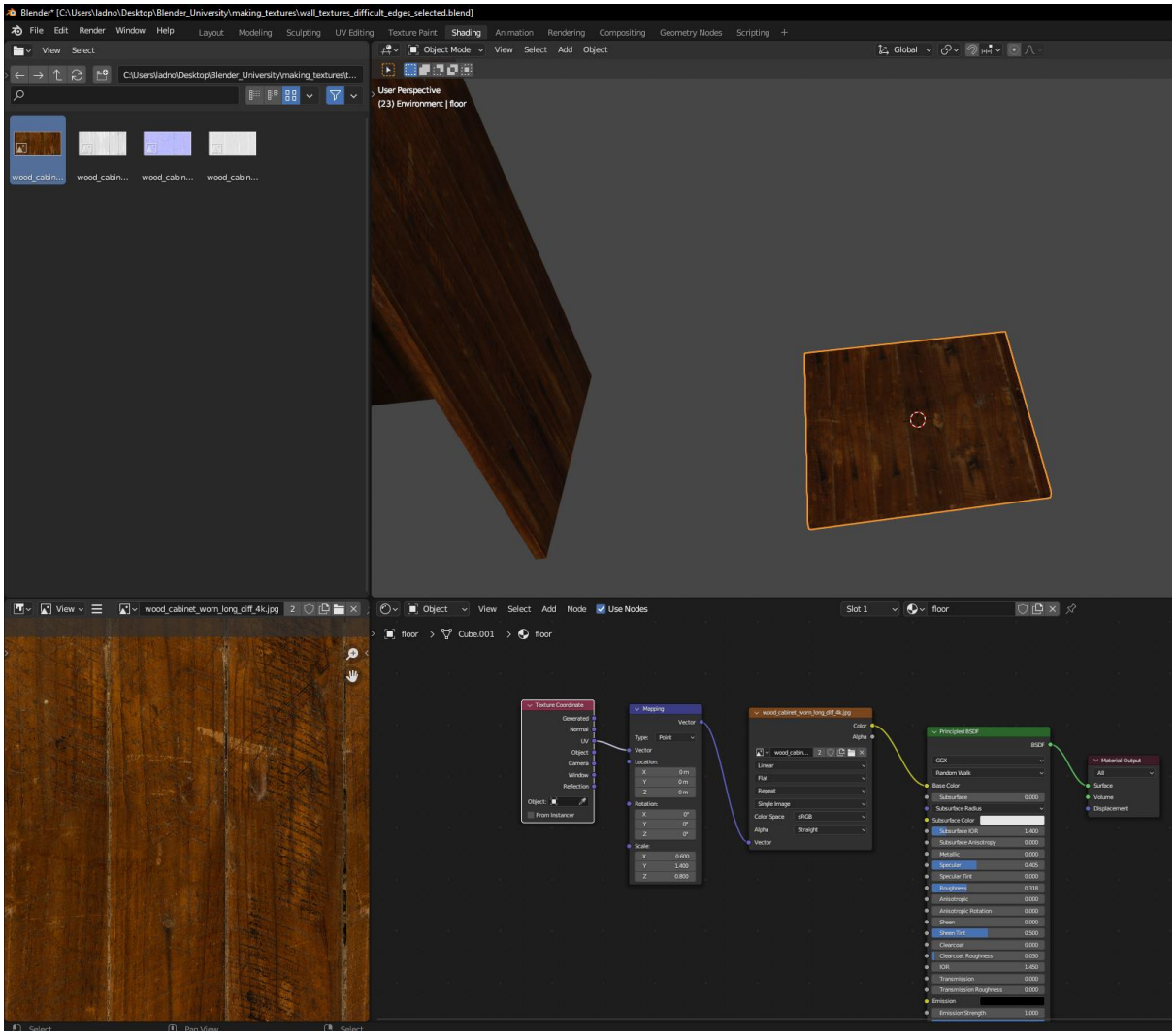


Fig 2. Setting up the Shading tree in the Shading editor. This was needed for the texture to fit perfectly into the floor tile. Techniques were learned from “CS426 Computer Graphics” university module.

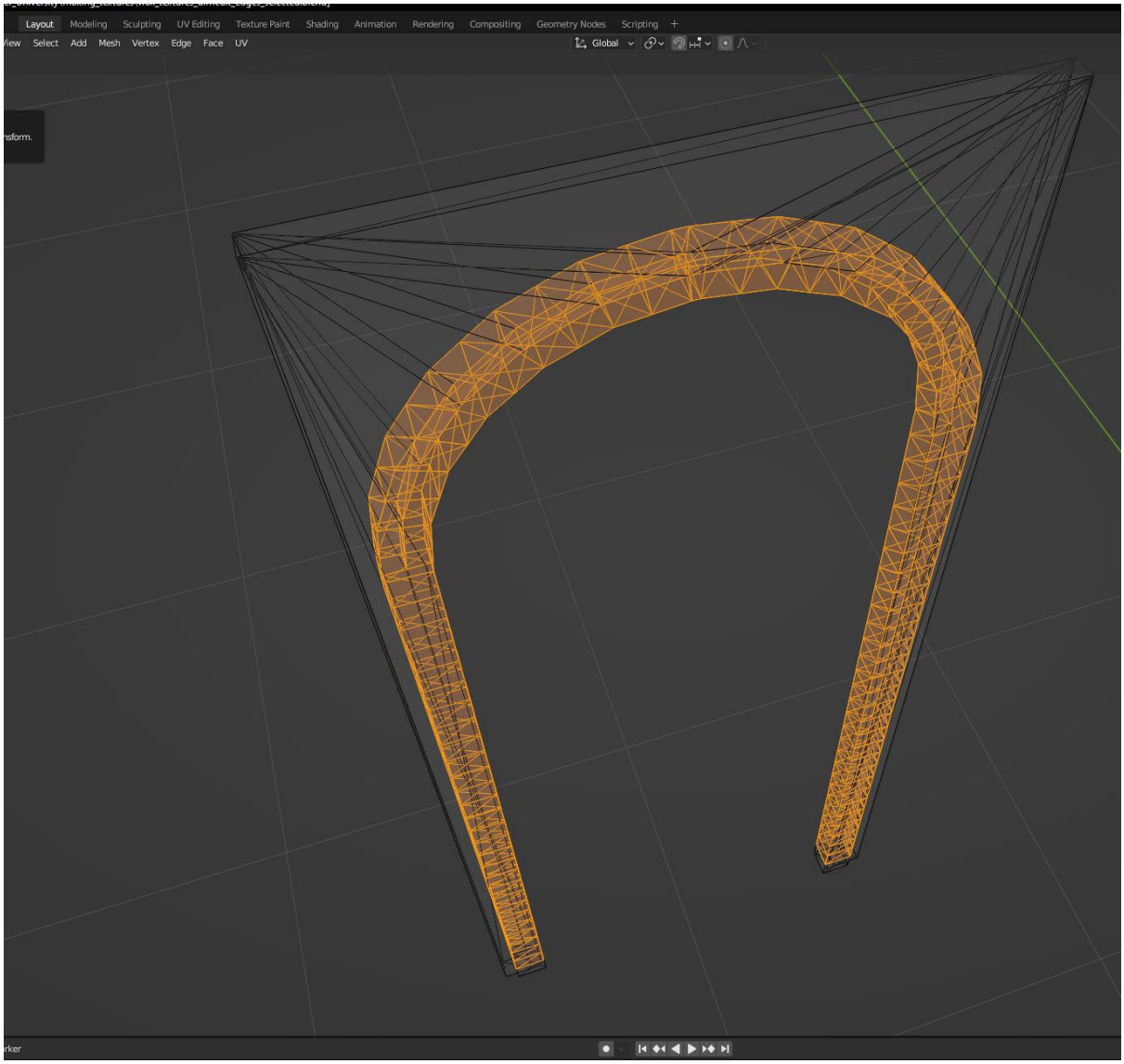


Fig 7. Creating doors inside Blender 3D. Trying to reduce the number of vertices and edges, and make different designs for different themes.

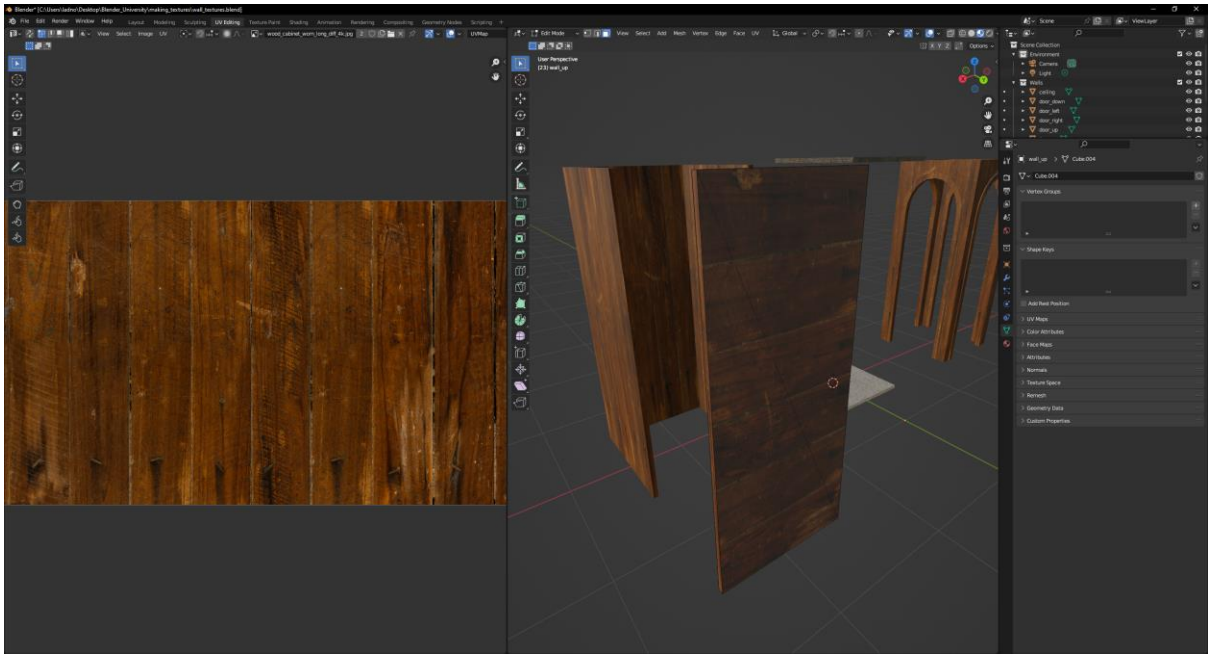


Fig 8. Creating wooden walls inside Blender 3D. Techniques learned from “CS426 Computer Graphics” module.

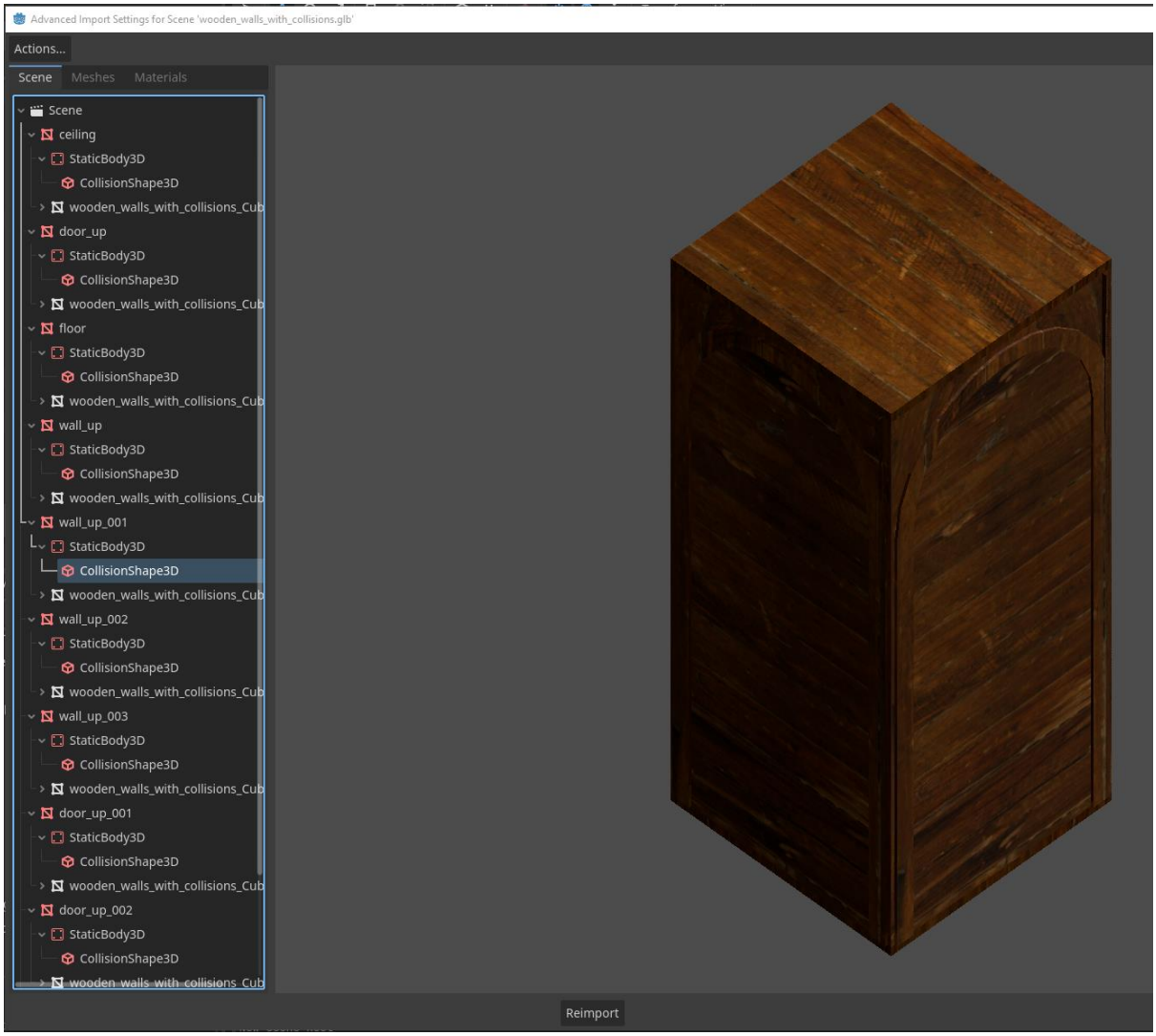


Fig 9. Importing the walls, door, floor and ceiling as 1 cell into Godot, with all collisions set up through Blender3D.

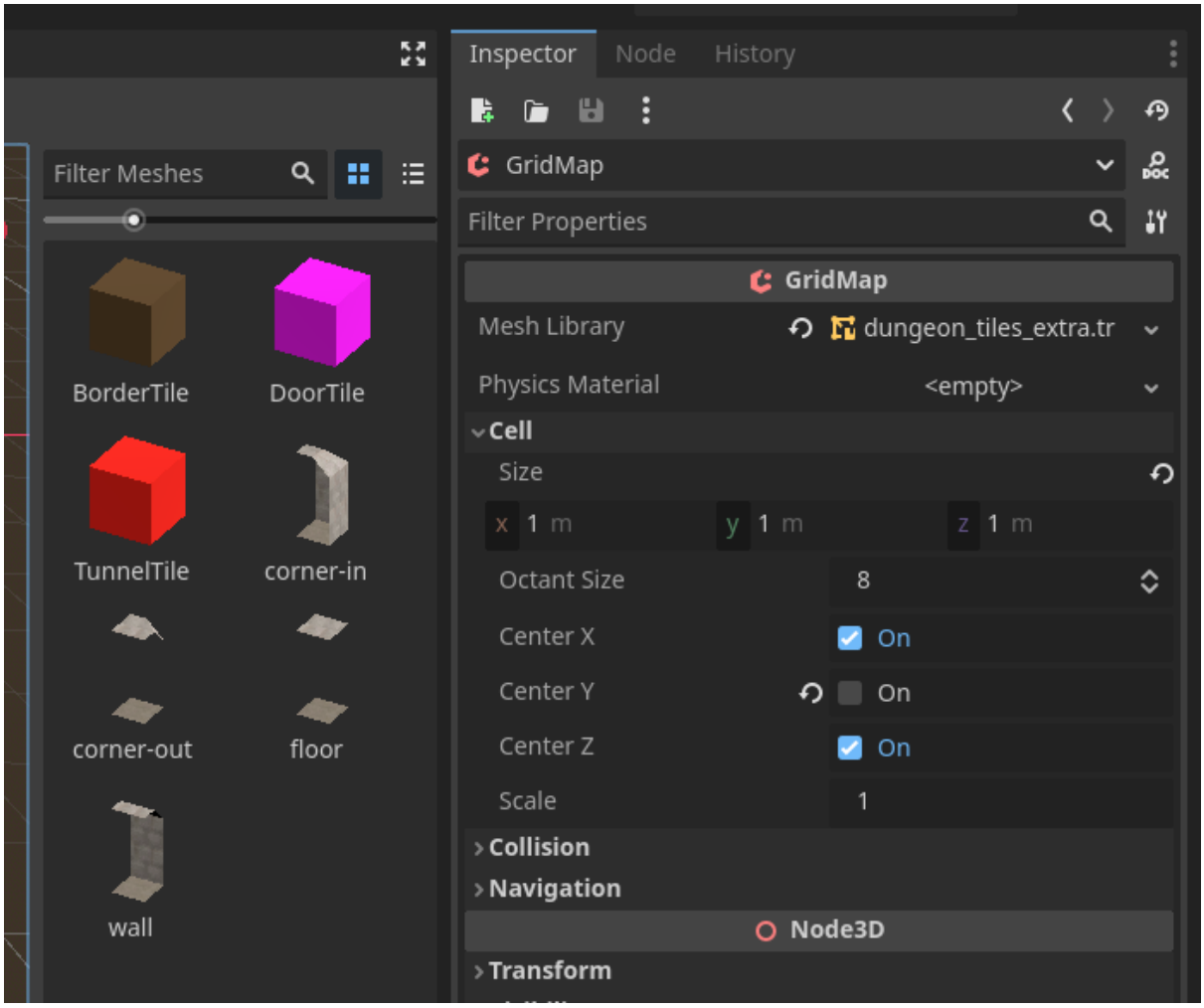


Fig 13.1. Creating my first mesh library which contains coloured tiles for the dungeon layout, as well as first dungeon textures.

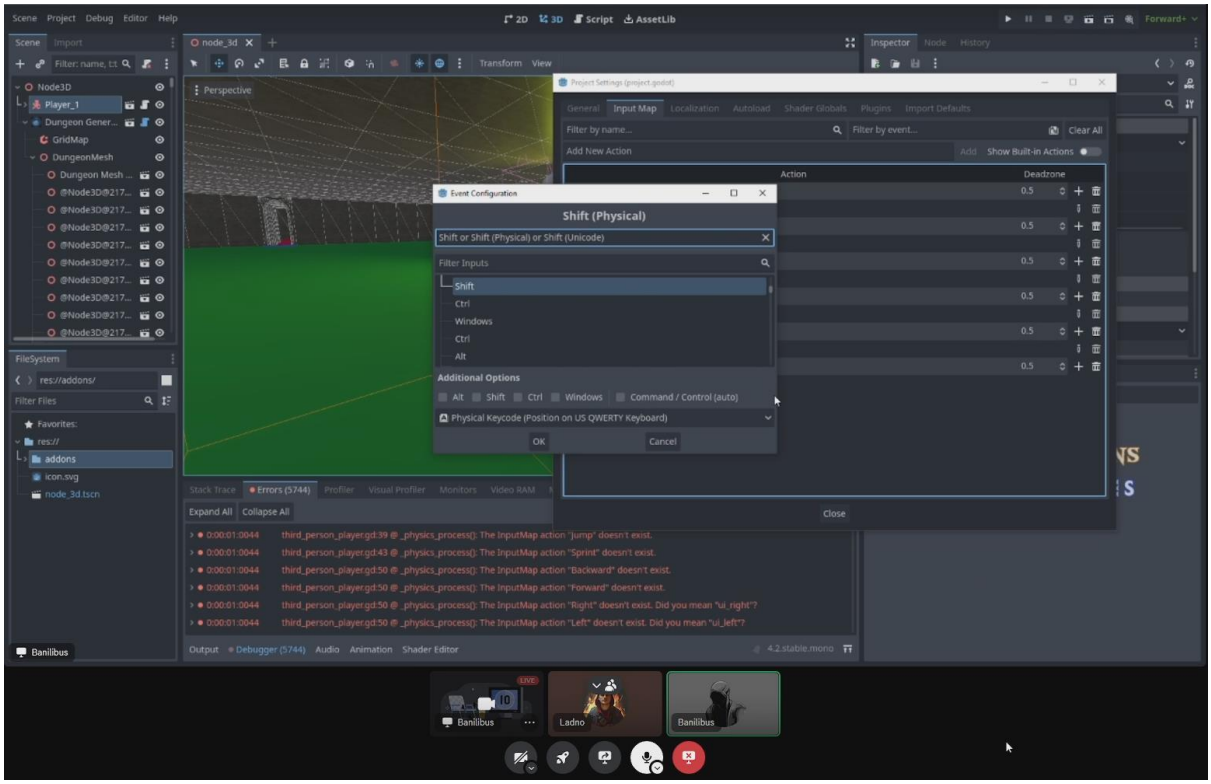


Fig 14. Discovering a critical issue that only triggers on different machines. This issue was found during Acceptance Testing phase, in a discord call with another user doing sanity checks.

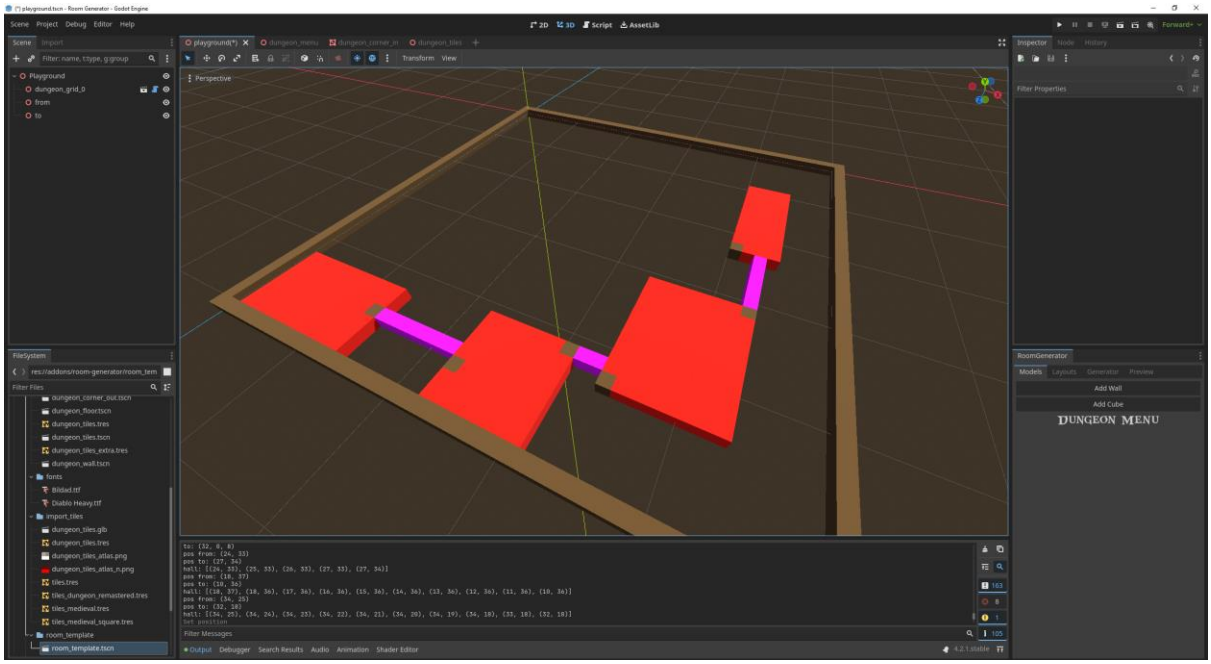


Fig 15. Testing the implementation of tunnel connections. The tunnels are not connected to doors, a lot of debugging was going on during this phase, as seen in the console.

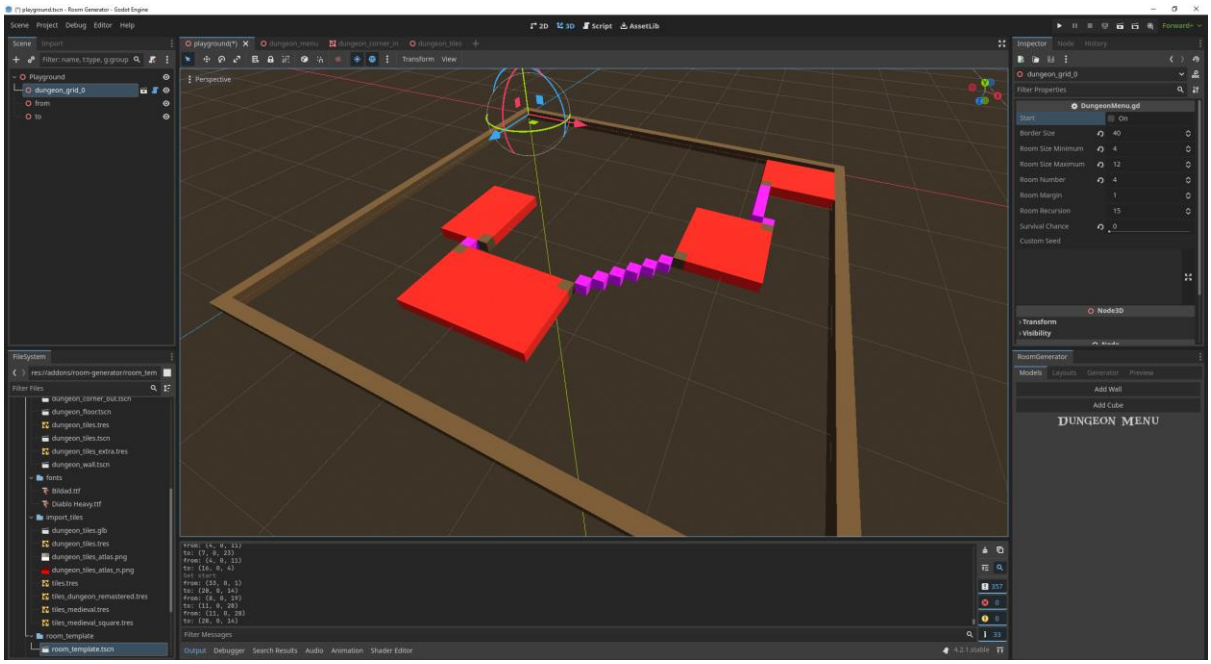


Fig 16. Encountered an issue where the tunnel connections would start spawning diagonally which was not intended and was fixed later.

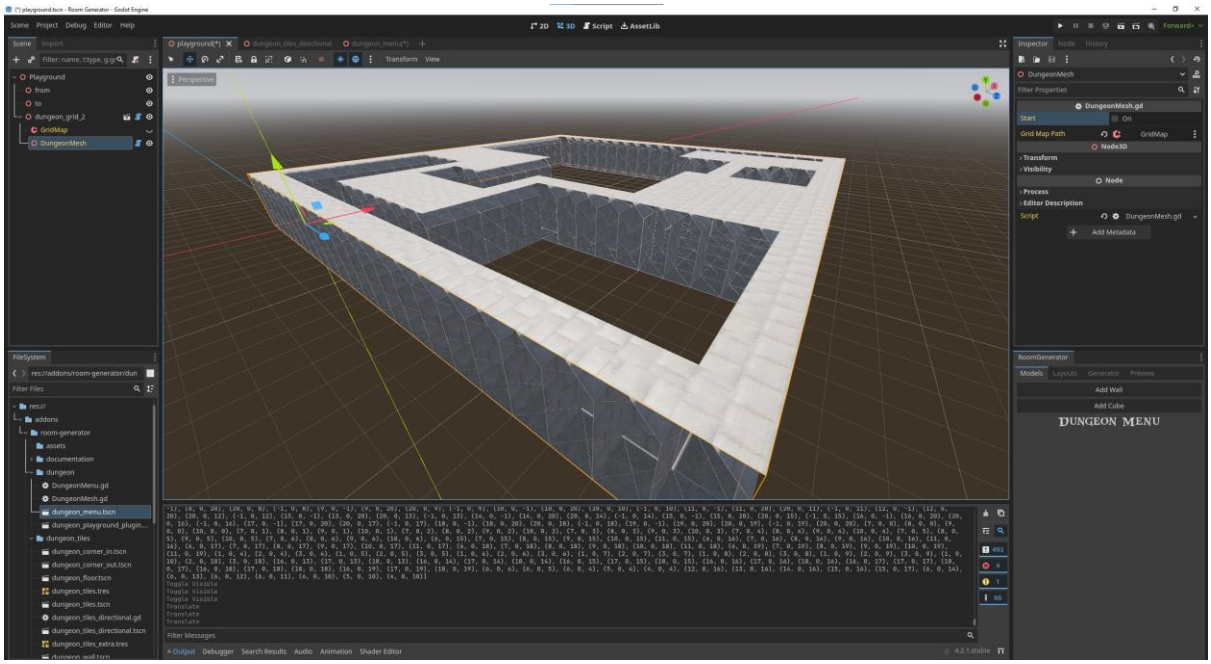


Fig 17. First time the mesh has been spawned on top of the generated GridMap cells. A lot of issues there as the texture spawned had no logic behind it for placing them correctly.

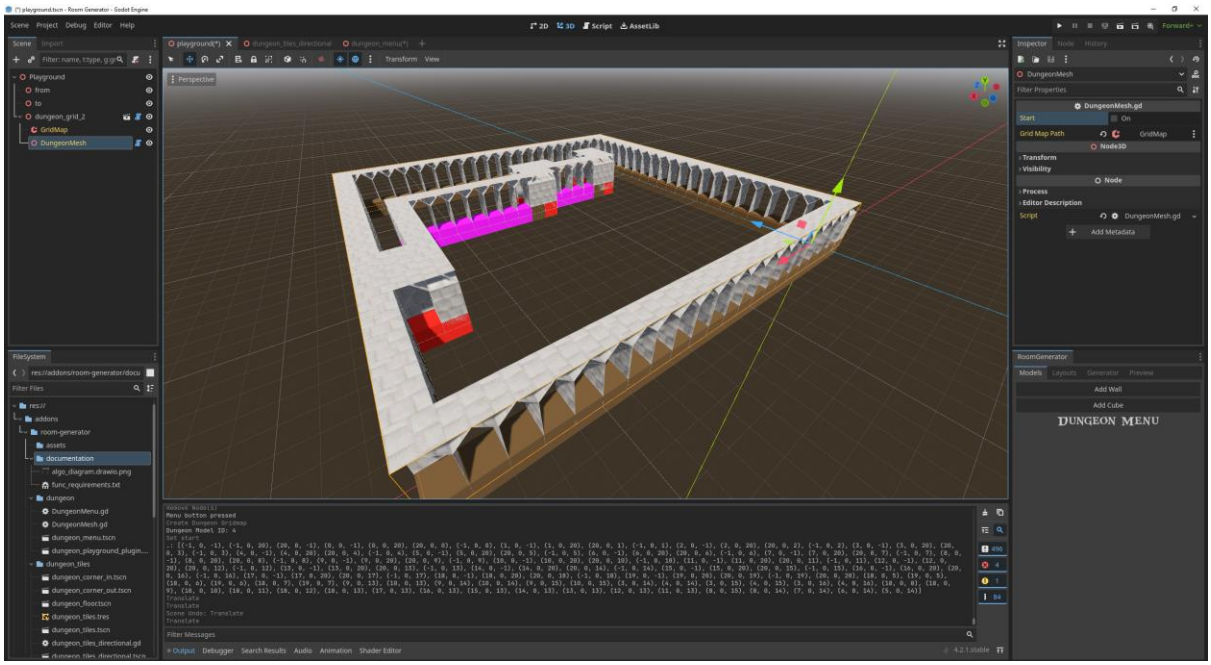


Fig 18. Debugging the mesh that has been spawned on top of the generated GridMap cells. A lot of issues there as the texture spawned had no logic behind it for placing them correctly.

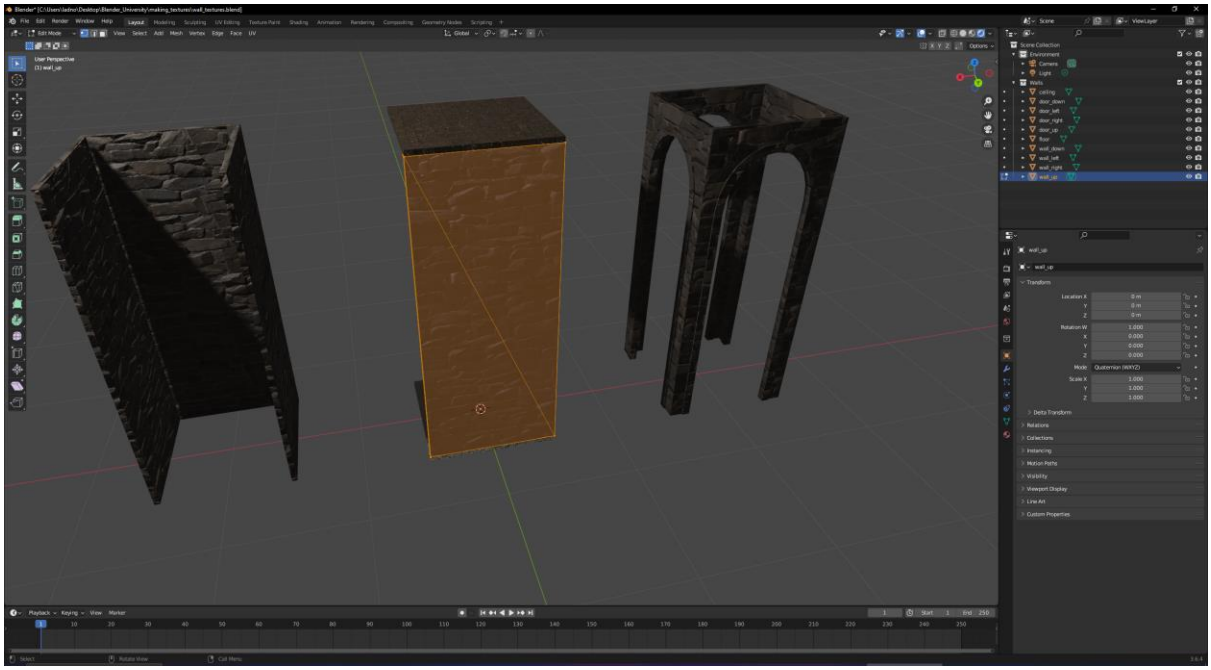


Fig 19. Using Blender 3D to create dungeon themed walls, doors, ceiling and floor. Techniques learned from “CS426 Computer Graphics” module.

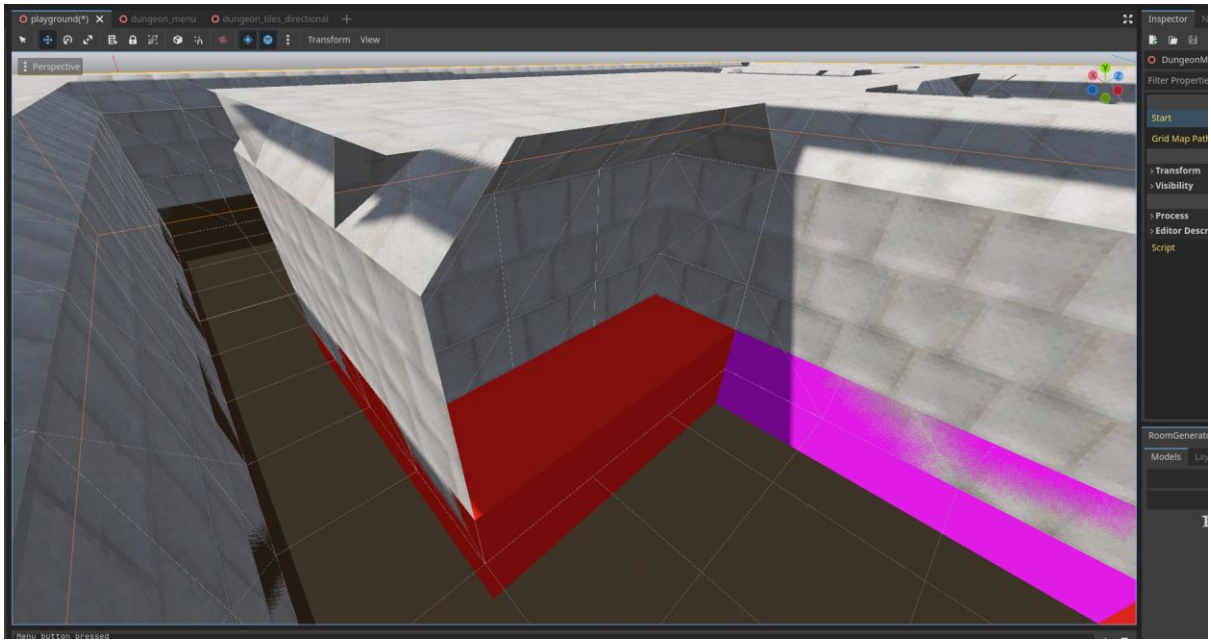


Fig 20. Debugging the mesh that has been spawned on top of the generated GridMap cells. More logic has been implemented here, but the walls were still placed incorrectly.

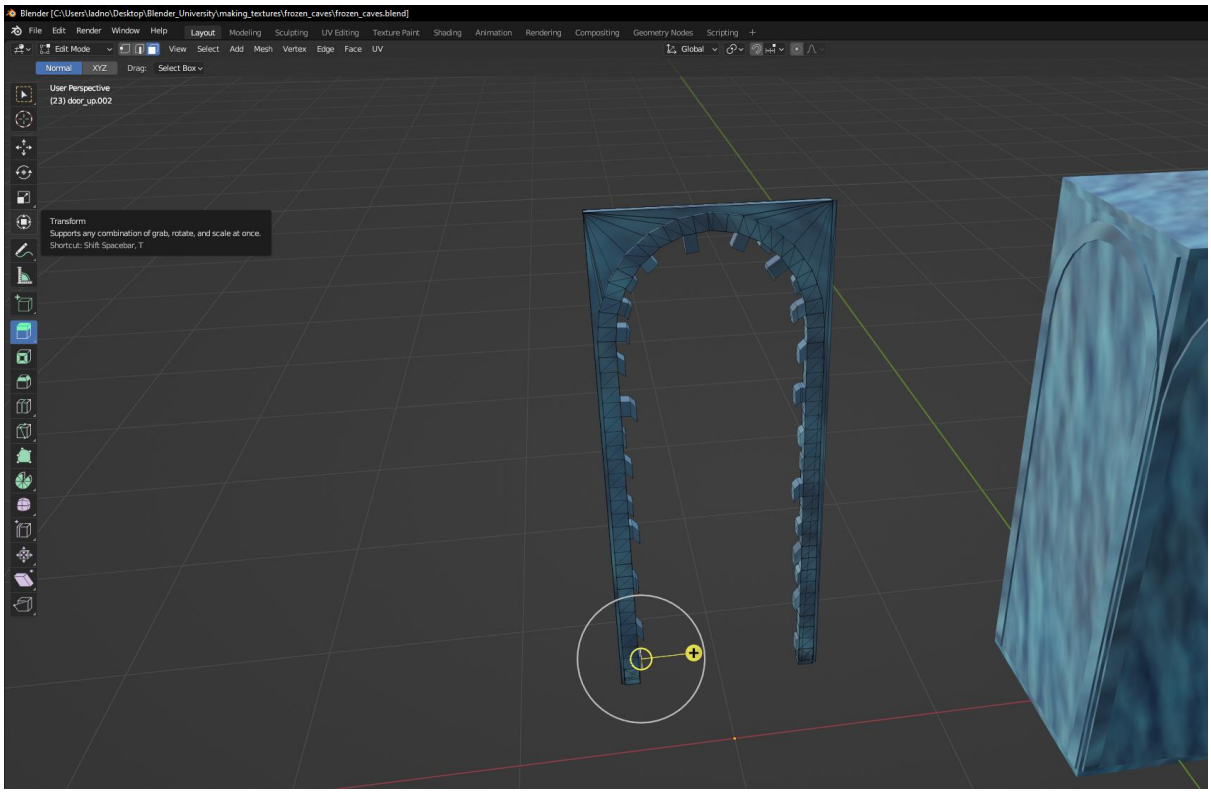


Fig 21. Creating frozen doors inside Blender 3D. Trying to make a unique design for the door to distinguish from other themes that exist in the extension.

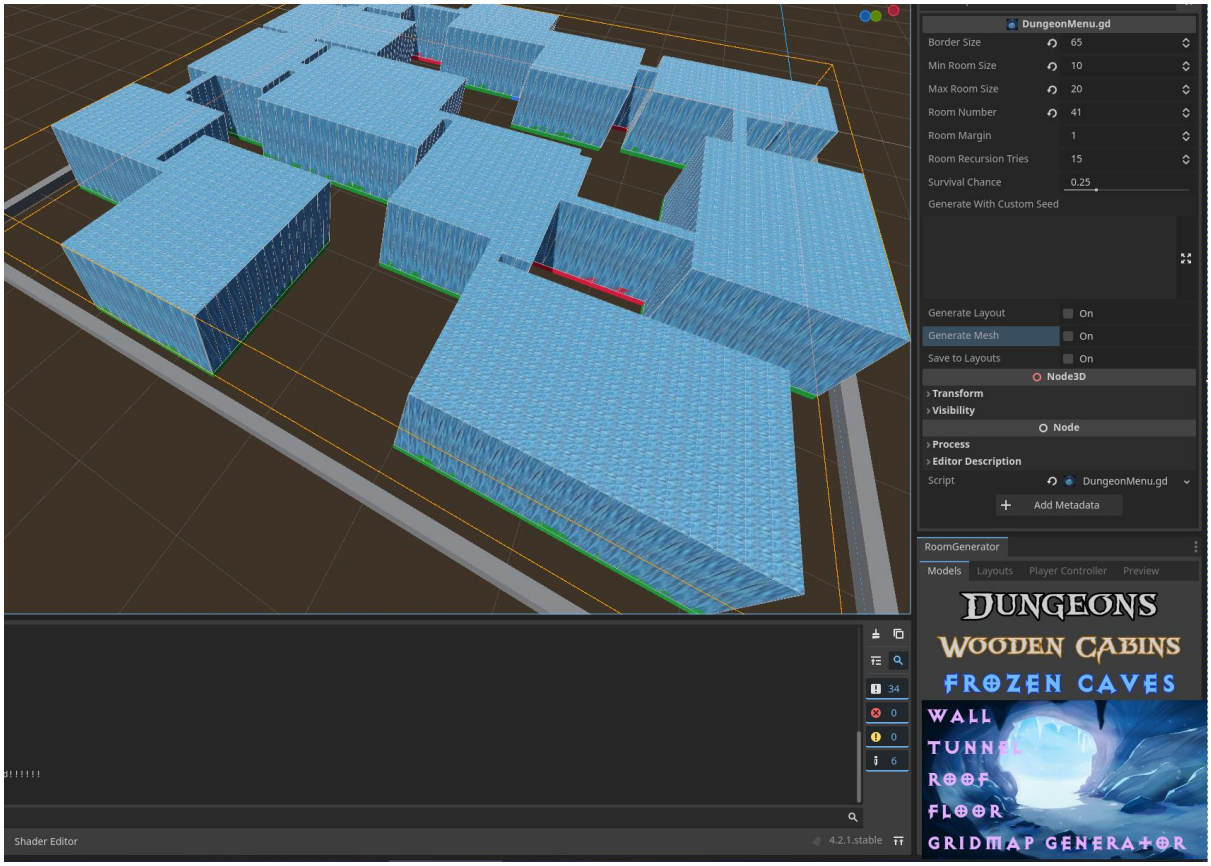


Fig 22. First time the dungeon mesh is spawned correctly onto the grid map tiles. Screenshots shows the plugin in action, with all of the UI open.

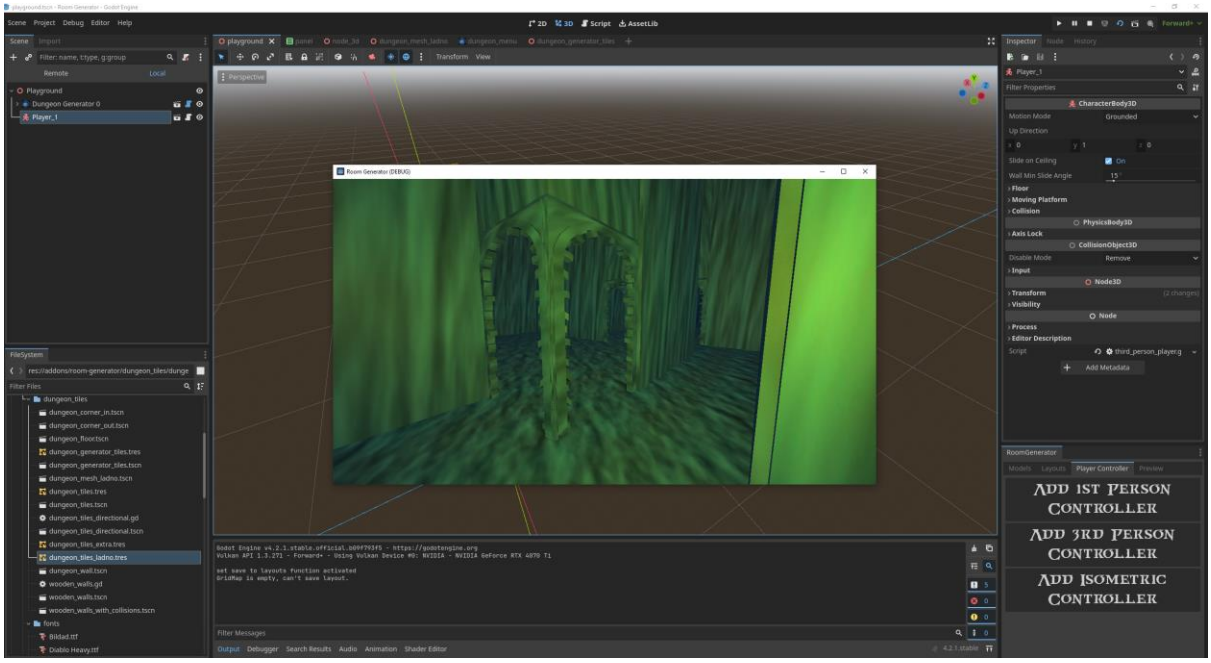


Fig 23. Running the generated dungeon in gameplay environment, using the first person controller spawned from the plugin. This screenshot shows a double door spawn, which is uncommon.