Project Documentation: Moldus

Filip Bedoya

May 30, 2025

Contents

1 Introduction

This document describes the development process, architecture, and implementation details of the project *Moldus*, a top-down rogue-lite dungeon crawler developed in Unity using C#. The original idea for Moldus emerged from a game jam, with sprites created by Jaromír Procházka.

1.1 Project Goals

- Develop a procedurally generated dungeon system.
- Implement a combat system centered around multiple ways of attack.

2 Game Design

The game is a 2D dungeon crawler with a mushroom-themed aesthetic. Players explore procedurally generated levels, battle enemies, unlock new abilities, and defeat powerful bosses.

2.1 Core Mechanics

2.1.1 Game Loop

The game loop is structured around exploring the dungeon, collecting upgrades and points. Upon reaching the exit, they engage in a boss fight before advancing to the next level. Every time a level is completed, the dungeon gets bigger and bigger.

2.1.2 Combat

The combat system involves using weapons/attacks at the player disposal that damage enemies from short or long range. Enemies spawn from spawners that if not destroyed spawn new enemies.

2.2 Weapons

The player can attack enemies in four different ways: using a basic flamethrower, two types of bullets, or bombs. All weapons function similarly, requiring the player to aim in the desired direction and press the left mouse button to attack, except for the bomb which just placed under the player. All weapons except for the flamethrower correspond to an enemy type that use it as their method of attack.

Weapon	Effect
Flamethrower	Very close range, damages enemies over time.
Purple	Long range high speed bullet
Green	Short range higher damage wave
Red	Places a bomb on the ground, which can also damage the user

Table 1: Weapons

2.2.1 Infections

Initially, the player wields a standard flamethrower. However, after defeating a sufficient number of enemies of a specific type, they can self-infect to temporarily gain the attack abilities of that enemy while also instantly healing themselves, this state is temporary and while active the player can not use any other form of attack. This mechanic incentivizes players to engage in combat and adapt their attacks based on the situation.

2.2.2 Enemy Types

There are 3 types of enemies, red, green and purple, each with different attacks. Spawners repeatedly generate enemies of a certain type up to a fixed limit. Enemies have attacks corresponding to the weapons from Table 1, Red enemies use the Red weapon, Purple enemies the Purple one and Green use the Green one.

2.2.3 Score

Score in this game is tracked by a Depth counter, which increments each time the player finishes the level by defeating a boss. Symbolizing that the player explores deeper and deeper areas of the dungeon.

2.2.4 Level Generation

Dungeon Sctructure Levels are structured as dungeons, which means they are composed of interconnected rooms and corridors. Rooms serve as the primary spaces where most gameplay elements are found, while corridors connect rooms, facilitating movement between them. The whole dungeon is structured as a maze so there is only one exit far away from the spawn.

Dungeon Generation Levels are generated procedurally using the binary space partitioning algorithm. The levels' starting and ending rooms are positioned as far apart as possible, ensuring a sense of exploration. The rooms are populated with pillars that block the line of sight of enemies but can be destroyed if attacked, as well as enemy spawners that fill the rooms with enemies. Each room can contain only one type of enemy, and the floor and walls grow mold that corresponds in color to the enemy type.

2.2.5 Upgrading

The player can enhance their character's abilities by acquiring upgrades. Upgrades are earned by destroying a sufficient number of spawners, which fills the spore counter. Once the counter is full and flashing, the player can choose one of the three randomly chosen upgrades. After selecting an upgrade, the spore counter resets, with the number of spores required for the next upgrade increasing. Some upgrades have levels, so a level 2 speed upgrade is better than level 1, but can be obtained only after level 1 has been acquired.
List of upgrades can be seen in the table 2.

2.2.6 Bosses

Upon reaching the level exit, the player enters a boss room. There are three possible boss encounters, each with distinct attack patterns and behaviors.

Upgrade	Effect	Levels
Damage	Just increases damage dealt	3
Speed	Increases movement speed	3
Fire rate	Increases the shooting frequency	3
Firefly	Ball of light that guides the player to the nearest exit	1
Shield	Absorbs damage and regenerates when depleted	1
Rage	When low on health, grants a damage bonus	1
Poison	Bullets and bombs are poisoned, poison gradualy deals damage	1

Table 2: Available Upgrades

Defeating a boss rewards the player with enough spores to gain a new upgrade, by giving him the maximum possible amount of points at the moment. After defeating the boss, the player can proceed to the next level.

Red Boss Throws bombs at the player and, when close enough, teleports away. Making him difficult to beat with just the flamethrower.

Green Boss Spawns minions to hide inbetween them. Its behavior is the same as the green enemy, so are the minions.

Purple Boss Shoots bullets around him and remains stationary.

3 User Interface

3.1 Main Menu

The main menu, seen in Figure 1, is the starting screen of the game, where the player can:

- Start a new game, green butten in the center
- Quit the game, red button in the lower center

As seen in the figure 1.

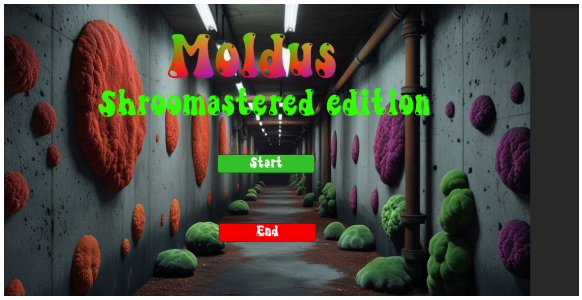

Figure 1: Main Menu

3.2 GUI

Game UI seen in image GUI, consists of:

- Health display, in the top left corner
- Spore counter for Upgrades, under the Health display
- Killed enemies counter for infections, three counters in the top right
- Depth counter to track score, under the Spore counter

As seen in the figure 2.

Figure 2: GUI

3.3 Pause Menu

The pause menu, seen in figure 3, can be accessed by pressing **Escape**. It pauses the game and it allows the player to:

- Volume Slider, in the top right corner
- Resume the game, in the center on top of the menu
- Restarting Last Level that was completed, under the Resume button
- Restart whole game, by generating new dungeon and reseting all values, under the Restarting Last Level button
- Quit the game, under the Restart button in the center bottom of the screen

As seen in the figure 3.

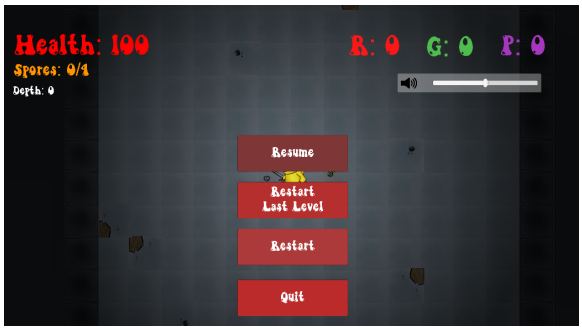

Figure 3: Pause Menu

3.4 Game Over Menu

When the player dies, the Game Over menu appears, seen in figure 4, offers options to:

- Restart the game, red button in the center
- Restart the last level, red button undert Restart
- Quit the game, red button under Restart Last Level

As seen in the figure 4.

Figure 4: Game Over Menu

3.5 Upgrade Menu

When enough spores are collected, the player can open the upgrade menu by pressing **Space**. Can be hidden again by **Escape** This allows them to:

- Choose new upgrades
- View upgrade descriptions via tooltips

As seen in the figure 5.

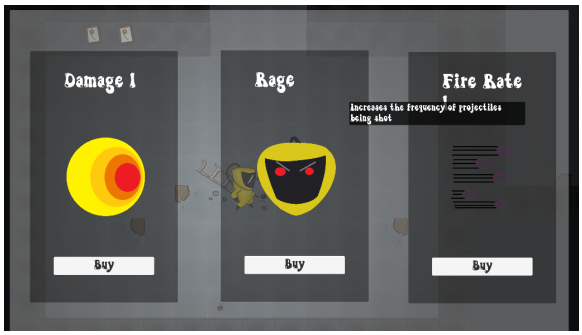

Figure 5: Upgrade Menu

4 Controls

4.1 Movement

Move using the **WASD** keys or Arrow keys.

4.2 Attack

All attacks are performed by pointing the mouse in the desired direction and pressing the **left mouse button**. The only exception is bombs, as they do not rely on mouse direction.

Switching infections is done by pressing **1, 2, or 3**:

- **1** for **Red** (Bombs)
- **2** for **Green** (Waves)
- **3** for **Purple** (Bullets)

4.3 Upgrades

When enough **spores** (upgrade points) are collected by destroying spawners, the player can press **Space** to open the **Upgrade UI**. Upgrades can be selected using the **mouse**. Hovering over an upgrade displays a **tooltip** with a description.

5 Technical Architecture

The game is built using the Unity Engine with the Universal Render Pipeline (URP) for enhanced lighting and performance. The core game logic is implemented in C# using Unity's component-based architecture. Procedural generation, combat mechanics, and entity management are handled through custom scripts.

5.1 Combat System

A lot of the objects in the game have colliders, one use for them is in combat, where they allow hit detection, so all enemies and the player have colliders so their attacks. Every attack that collides calls the `HealthController.cs` script on the target, invoking its TakeDamage function. It is a script that allows the object to manage its own health, it can take damage, heal damage and die. If the object lacks this script, it cannot take damage. So, most of the combat is implemented by spawning bullets and then those calling TakeDamage functions.

5.2 Infection and forms of attack

The players default mode of attack is a flamethrower, which is just a trigger(an area that registres if another object with collisions entered it) that the enemies take damage in. Infections are managed by `PlayerStatsController.cs` that takes care of the animation and attack changes and if it is possible to change, and `EnemyKillCountController.cs` that counts slain enemies.

5.3 Restarts/Reloads

These are being handled by invoking UnityEvents that have mainly in the inspector assigned everything that should be done to restart the game, such as reconfigurating the player, regenerating the dungeon, reseting or loading all the stats such as score and reseting or reloading upgrades. Reloads get values from a saved json file. The difference between reloading and restarting is that in reloading stats such as upgrade spores, upgrades, score and killed enemy points are loaded and the level also regenerates with the same seed, so it is the same. While restarting resets all those values and generates a new level. Saving occurs when a boss is defeated so in practice when the last level was completed.

5.4 Architecture of Upgrades

UpgradeSO Upgrades are stored in a data containers known as scriptable object that in practice allow definition of what an object should contain and then instances with filled fields can be created, it contains field that define its

name, description, upgradeicon, but also two function, Activate and Revert. That allows it on demand to apply the upgrades but also remove them. This allows upgrades to be abstracted.

UpgradeManager Applies and reverts upgrades. It randomly chooses upgrades from list that can be shown on the UI and then activates it if chosen.

UpgradePointsCounter Just updates the UI if a spawner has been destroyed, if enough spawners have been destroyed, it calls the `upgrademanager` to choose new upgrades and show them to the player.

5.5 Objects

5.5.1 Characters

All characters have a `HealthController.cs`.

Player Includes `PlayerMovementController.cs`, `PlayerShootingController.cs`, `PlayerStatsController.cs`, and `UpgradeManager.cs`. These components manage player movement, attacks, infection changes, and upgrades by storing, applying, and interfacing with relevant UI scripts.

Enemies Enemies inherit from `BasicEnemy.cs`, thats a script that sets up the basic pathfinding and functions, it also implements the IEnemy interface. That demands that the enemy should have an Attack function. Path inding is handled by the external library A* Pathfinding project, which scans an area and by assigning relevant scripts to an object the enemy pathfinds toward a target, while avoiding obstacles. So each enemy has relevant script from the project assigned and toggles it on or off, or changes the target destination.
Since all enemies controllers inherit from `BasicEnemy.cs`, that tells the enemy that if it can see the player, by using a raycast, which is a line that allows as to determin if there are any obstacles, by checking if the line is interupted by a collider between the enemy and the player, than it should path-find into its firing range, and than attack.
Standard enemy AI in `BasicEnemy.cs` is that it waits until the player enters vision distance, than checks if the player is not behind a wall by RayCast which draws a straight lines to the player and checks if it doesnt collide with anything. If it can see the player that it enters its pursue state and tries to path find to the player into its shootingRange and than attacks. The attacks themself are defined by enemy types.

PurpleEnemy Uses `PurpleEnemyController.cs`, that Defines the Attack function as shooting a projectile in the players direction. They have a higher range value so they path find to the player less frequently.

GreenEnemy Similar to PurpleEnemy, uses the `GreenEnemyController.cs`, defines the Attack function as shooting a projectile in the players direction.

RedEnemy Uses the `RedEnemyController.cs`, rewrites the behaviour function to focus it on planting bombs and running away.

Spawners Uses the `SpawnerController.cs` that makes it spawn enemies. Enemies are spawned randomly in a radius that can be customized. The spawn period, maximum number of enemies are also adjustable.

Bosses Bosses also mostly inherit from `BasicEnemy.cs`.

Red Boss Uses `RedBossController.cs`, adds a new projectile attack and teleportation to the standard `BasicEnemy.cs`.

Purple Boss Uses the `PurpleBossController.cs`, rewrites most of the behavior, since it doesnt need the checking players proximity and pathfinding, and makes it shoot bullets around itself while stationary.

Green Boss Green Boss has two scripts attached, one for spawning enemies `GreenBossController.cs`, one for behaving like a green enemy `GreenEnemyController.cs`. `GreenBossController.cs` inherits from `SpawnerController.cs`, it add OnDestroy function to kill all enemies it spawned.

5.5.2 Attacks

Attacks describes all objects that are used as a form of attack by the player and or the enemy.

Bullet Standard bullet from the shooting form of attack, can have variances using scriptable object BulletDataSO thats a container that allows to store instaces with specific values in its fields. BulletDataSO store damage, speed, despawn distance, sounds of the bullet and sprite, this allows all the basic bullets to use `BulletController.cs` for setup and OnCollision function, but still be unique thanks to the values from BulletDataSO.

Bomb Just remains stationary until it explodes, then leaves around a purely visual object showing a cloud of spores. It uses `BombController.cs` that handles the time till explosion and damage, the damages from the explosion is done by checking all colliders in a radius and calling TakeDamage, there are no checks if they are friend or foe.

Flame The flame from the Flamethrower is just a an area that registres if other objects with collisions enter it, it is called a trigger and enemies take damage in it, the `FlameController.cs` manages it for the player.

BossBomb Unlike the normal bomb counterpart, this bomb flies toward a destination and explodes when either hitting a wall or reaching the destination. It is handled by `BossBombController`.

5.5.3 Structures

End of Level hatch This just is a trigger that when the player enters runs the boss setup. Its handled by the `EndOfLevelHatchController.cs`.

Pillar Object that appears as a cracked tile in the room. But it can be destroyed. Enemies can not see through them. Their scripts are the `PillarController.cs` that changes the sprite when losing health and animations. Enemies path-find around the pillars and then they are destroyed the pillar makes sure that the path-finding is updated to fit the room without the pillar. In addition, it scans the area for path-finding, when the pillar is destroyed. And the second script is the `HealthController.cs`, the same scripts as all creatures have. It allows it to take damage.

5.5.4 Level Generation

The level generation is done in multiple script, each script has their own specific task in the generation. `RoomGenerator.cs` is the main script that controls the generation by calling other script in creation of the dungeon. They can be subdivided into scripts that just contain algorithms, scripts that generate a specific thing in the dungeon and others that help with specific tasks of the generation.
First, it calculates the coordinates of everything on the tilemap, so we can place them later. A tilemap is a square grid that allows us to easily build a level by filling the squares with tiles, which are images of for example floor we want and allows us to define its properties, such give it a collider for walls. The first think we get are the room coordinates, which are calculated in `BinarySpacePartitioningAlgorithms.cs` using the binary space partitioning algorithm. It returns just a list of bounds, that says the maximal and minimal coordinates of a room on both axes, all the coordinates

in the bounds are calculated and placed into a `HashSet` of coordinates by `BasicDungeonGenerationAlgorithms.cs`, so we can get all of the coordinates into a list and than use it for placing tiles for example, we use a *HeshSet* since we just need a list of coordinates to place floor tiles in and dont care in what rooms they are and there are no duplicates. Then, the center of each room is gathered and sent into `RoomConnectAlgorithm.cs`. It chooses a random room from the connected part (which at the start is just one room) and then connects it to the nearest unconnected room by creating a line of floor tiles, the nearest room is found by checking distances in straight lines for how far are centers of the rooms from each other. It Returns the floor coordinates of the corridors by just adding all the coordinates on the lines from one center to another, which are added to the rest of the coordinates in the `HashSet`. Then, the furthest rooms from each other are calculated in *DungeonContentGeneratorAlgorithms.cs* by checking which rooms are furthest apart using BFS, thanks to this the end of the level is always furthest from the start. Each room is then assigned a color by picking a random color from the enemies.
Then, walls are generated by `WallGenerator.cs`, which are just coordinates that are next to a floor tile but not part of the floor, it manages corners by checking if a floor tile neighbours two walls and if yes fills the diagonal. The `WallGenerator.cs` and `RoomGenerator.cs`, for displaying tiles, call `TilemapVisualizer.cs`, which stores all the relevant data, such as the tilemaps, this helps with using all the saved up coordinates. The `TilemapVisualizer.cs` is called to paint the floor and add random tiles corresponding to each room's color. Random tiles are either colour depended or general, and they are just lists of tiles that are randomly placed in the room to add visual flair, in case of the colours, it chooses just from a list of tiles corresponding to the enemy type that will spawn in the room. Spawners, pillars, the player spawn, and the `EndOfLevelHatch` are placed with the help of `DungeonContentGenerator.cs` and `DungeonContentGeneratorAlgorithms.cs`. The whole map is also rescanned for pathfinding.

Boss room The boss room is generated similarly to the rest of the dungeon. `BossSetupController.cs` contains the main function, which calls `BasicDungeonGenerationAlgorithms.cs` to generate the room bounds and its floor, and then places and colors it using *TilemapVisualizer.cs*. Then, it places pillars using *DungeonContentGenerator.cs*, in the same manner as used in the rest of the dungeon, and sets up the player and the boss itself.

5.5.5 UI

- *Boss Health Bar* - Shows a big red slider in the lower part of the screen. Managed by `BossHealthBarController.cs` and is used to show the

bosses health.

- *Health Text* - UI for the players health, just a number in the top left corner of the screen that is managed by `HealthTextController.cs`.
- *In Game Menu* - Menus are managed by `IngameMenuController.cs`, that invokes relevant events for the buttons. Such as restart, reload , quit or resume.
- *Main Menu* - The main menu is the first screen the player sees when the run the game. It is just two buttons, one for starting the game one for quitting, handled by `MainMenuController.cs`.
- *Volume* - A slider in the pause menu that allows the player to change the games volume, it is handled by `VolumeController.cs`.

UpgradeUI The Upgrades have their own UI. It is divided into the main screen that shows the three available upgrades through cards. It is handled by `UpgradesUIManager.cs`. Those cards are handled by `UpgradeUICardController.cs` and they just allow the easy display of the UpgradeSO. Then there is also the Tooltip that shows the upgrades description when hovered over, that is handled by `UpgradeTooltipController`.

5.5.6 Additional Scripts

These are objects and scripts that dont fit any category, but are still relevant in the project.

- *CameraManager.cs* - Manages the game camera.
- *ColorTypes.cs* - Defines the types of colours/infections in the game by an enum.

6 External Libraries used

For path-finding the AstarPath finding project was used.